Cryptfs: A Stackable Vnode Level Encryption File System

Erez Zadok, Ion Badulescu, and Alex Shender
Computer Science Department, Columbia University
{ezk,ion,alex}@cs.columbia.edu

CUCS-021-98

Abstract

Data encryption has become an increasingly important factor in everyday work. Users seek a method of securing their data with maximum comfort and minimum additional requirements on their part; they want a security system that protects any files used by any of their applications, without resorting to application-specific encryption methods. Performance is an important factor to users since encryption can be time consuming. Operating system vendors want to provide this functionality but without incurring the large costs of developing a new file system.

This paper describes the design and implementation of Cryptfs — a file system that was designed as a stackable Vnode layer loadable kernel module[5, 15, 19]. Cryptfs operates by "encapsulating" a client file system with a layer of encryption transparent to the user.

Being kernel resident, Cryptfs performs better than user-level or NFS based file servers such as CFS[2] and TCFS[3]. It is 2 to 37 times faster on micro-benchmarks such as read and write; this translates to 12-52% application speedup, as exemplified by a large build. Cryptfs offers stronger security by basing its keys on process session IDs as well as user IDs, and by the fact that kernel memory is harder to access. Working at and above the vnode level, Cryptfs is more portable than a file system which works directly with native media such as disks and networks. Cryptfs can operate on top of any other native file system such as UFS/FFS[8] and NFS[11, 16]. Finally, Cryptfs requires no changes to client file systems or remote servers.

1 Introduction

There is no easy way for users to transparently protect files. Security systems such as Pretty Good Privacy (PGP)[23] require a lot of application-specific support, consume a lot of CPU cycles, are often incompatible with other systems, and ask the user to be directly involved with their setup and maintenance. Furthermore, many applications are not well integrated (or at all) with security systems. Today's users find themselves in the ironic position of having powerful workstations and fast networks, yet they are unable to reasonably protect their data with minimal effort and a small performance impact.

If the file system is kernel resident, it benefits from increased performance because of the reduced number of context switches and from running in privileged mode. A kernel based file system can offer better security than user-level file systems and encryption tools because information that is kernel resident is harder to get at, and the kernel has better access to private resources not available elsewhere. Encryption as part of the file system automatically offers a uniform encryption for all applications that access files, and reduces the user's involvement.

A file system that transparently allows access to encrypted data is an appealing idea. The idea has long been in existence and was implemented first by Blaze[2] (CFS) and later by Cattaneo and Persiano[3] (TCFS). However, these prior realizations have suffered from poor performance and they are harder to use; TCFS also suffers from limited availability. Consequently, transparent cryptographic file systems have not received wide use. We have remedied these problems in Cryptfs.

Cryptfs is implemented as a kernel-resident file system, and can be mounted on any directory and on top of any other file system such as UFS and NFS — without requiring other daemons running. Users authenticate themselves by using a tool that prompts them for a passphrase which is cryptographically hashed using MD5[14] to form a key which is then passed to and stored in memory by Cryptfs. No information related to encryption is stored permanently, making Cryptfs both easier to use and more secure. Cryptfs uses the Blowfish[18] encryption algorithm and determines key access in one of two modes. In the first mode, it looks up the key based on the real user ID (UID) of the accessing process; this allows a user on one machine to provide a key only once throughout the lifetime of the mount. In the second mode, Cryptfs uses both the UID and the process session ID to lookup keys. This mode offers greater security because only the process authenticated to Cryptfs and all future child processes will have access to the key — as they all share the same session ID. Attackers who break the user's account or can become that UID will not be able to easily decrypt that user's data because they could not join the same session. Cryptfs also achieves greater performance by running in the kernel and by avoiding the NFS (V.2) protocol overheads[11] such as asynchronous writes.

1.1 The Stackable Vnode Interface

Cryptfs 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 notable improvement to the vnode concept is "vnode stacking,"[5, 15, 19] 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 systems 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.

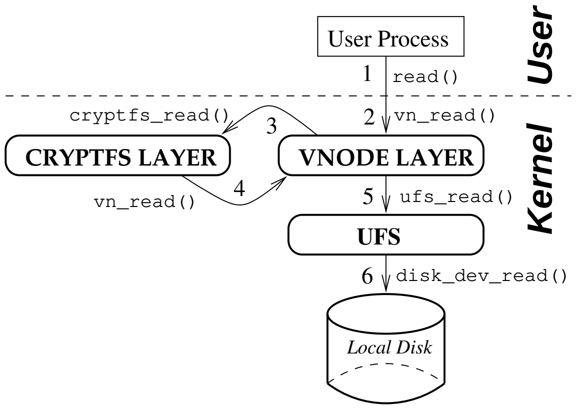

Figure 1: A Vnode Stackable File System

Figure 1 shows the structure for a simple, single-level stackable encryption file system. In this one, system calls are translated into vnode level calls, and those invoke their Cryptfs equivalents. Cryptfs again invokes generic vnode operations, and the latter call their respective "lower level" file system specific operations such as UFS.

The rest of this paper is divided as follows. Next, Section 2 provides discussion on the design of Cryptfs. Following, Section 3 details the implementation. Section 4 evaluates Cryptfs' performance and security among others. We survey related file systems in Section 5 and conclude with a summary and future directions in Section 6.

2 Design

Cryptfs is designed to be simple in principle. The file system interposes (mounts) itself on top of any directory, encrypts file data before it is passed to the interposed-upon file system, and decrypts it in the reverse direction. Our explicit design goals were:

- **Performance**: Cryptfs should exceed the performance of other encrypting file systems.
- **Ease-of-use**: Users should find Cryptfs simple to use or they may opt not to encrypt any files.
- **Security**: Cryptfs should offer strong enough security against most trivial and moderately sophisticated attacks; its design should not be complicated by the desire to protect against very sophisticated attacks.
- **Portability**: Cryptfs should be more portable than other kernel based file systems, by using a stackable vnode interface. It should not require modifications to other file systems or user applications, and it should keep the underlying file system valid.

The next five issues are treated in the following sections:

1. How to make it more difficult for others to decrypt data without authorization while at the same time providing simple encryption services transparently to authorized users?
2. Should encrypted bytes depend on the previously encrypted ones?
3. Does encryption affect file offsets or sizes and if so, how?
4. Should file names be encrypted?
5. How to keep the structure of all affected Unix file systems valid while encryption takes place?

2.1 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, an attacker would not only have to break into an account, but also arrange for his 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. So a user 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.

We made two small additional design decisions here. First, we decided to check for real UIDs and not effective ones. That way a user could run setuid programs and they would work with the runner's UID, not the file's owner. Secondly, if users find it too inconvenient, Cryptfs can be mounted with processing of keys based on UIDs alone (though we do not recommend it.)

We designed a user tool which prompts users for passphrases that are at least 16 characters long. The tool hashes passphrases using MD5[14] 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, and so on. With our design, you 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.

An alternative design option that would allow simultaneous access to multiple keys was to require that each user separately mount an instance of Cryptfs with a different key. This design option was rejected for two reasons. First, users would either require root privileges to mount or the file system would have to allow any user to mount Cryptfs. Secondly, this would not scale well with respect to the number of mounts required on a busy multi-user system.

2.2 Encryption Algorithm and Mode

To provide strong enough encryption it is necessary to encrypt as much data together in a chaining fashion that includes bit substitutions and transpositions, such that each byte encrypted depends on some of the prior ones. At the extreme, we could have designed Cryptfs to encrypt the whole file in this mode; but doing so would mean that each time we need to decrypt a single byte anywhere in the file, all prior bytes would have to be decrypted as well — a major performance problem. We decided to encrypt blocks of data in a size that is natural to the operating system used — 4096 or 8192 bytes. These values were chosen because they are the most common virtual memory subsystem page sizes, making it easier to handle memory-mapped operations (described in Section 3.4.)

Next we picked the algorithm. We rejected patented or licensed ones, and also rejected DES[20] because it is too big and slow. We picked Blowfish[18] — 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.

We selected the Cipher Block Chaining (CBC)[17] encryption mode because it allows us to encrypt byte sequences of any length — 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. This choice also minimizes potential data loss: if one byte is corrupted in a file, at most one page worth of data could not be properly decrypted.

2.2.1 File Offsets

Many programs read or write arbitrary data within files. They seek to a specific offset within the file and perform the read or write operation starting there. Some encryption algorithms may change the size of the input being encrypted — generally increasing it. If the encryption algorithm changes data size, it becomes difficult and costly to perform file operations at arbitrary offsets. The Blowfish algorithm was chosen also to avoid this cost. This algorithm does not change the size of the data being encrypted, making offsets in encrypted and decrypted files the same. Furthermore, since the Blowfish algorithm does not change the total size of the file, operations like `stat(2)` (getting file attributes such as size) can be handled simply by passing them on to the interposed-upon file system layer.

2.3 File Names

Users often choose comfortable file and directory names describing the nature of the data stored within. An attacker who discovers the names of files — even if they cannot access the file data — can still infer much about the nature of the data itself. Therefore, we decided to encrypt all file and directory names as well.

Encryption algorithms use a large subset of possible characters for the ciphertext. This strengthens the encryption by "randomizing" any possible patterns in the cleartext. But when encrypting strings that represent Unix file names, several characters may result that are illegal in file names, such as a forward slash (`/`) or a null. Such encrypted file names cannot be stored verbatim in the normal directory structures as they will corrupt the underlying file system. In addition, there are many non-printable characters that, while legal characters after encryption, are difficult to display on the screen (e.g., the output of `ls`) or may affect the terminal settings.

We decided that after encrypting file names, we will uuencode them to eliminate the unwanted characters and guarantee that all file names consist of printable characters. The uuencoding algorithm chosen is simple and fast. It converts every 3 byte encrypted sequence into a 4 byte sequence of ASCII characters from a set of 64 characters ranging from 48-111. Since each character in the chosen range requires only 6 bits, we were able to convert exactly 3 bytes of encrypted data chosen from a 256 character set to 4 bytes chosen from 64 printable characters.

The above choice also meant that file names become one third longer. This was necessary but it did not have the same ramifications as changing offsets of data within files. Since file names are always read whole and from the beginning of the file name, we can read them from the underlying storage, apply our uudecoding algorithm, and finally the decrypting algorithm. The resulting string would be the original file name and be returned to the caller.

Special consideration was given to the two directories that always exist: the "." and ".." directories. The encryption algorithm leaves them unchanged for two reasons:

1. If these two directories do not exist in the interposed-upon file system, normal directory operations such as changing directories to the parent one and other recursive operations would fail.
2. Since everyone knows that these two must always exist in Unix file systems, encrypting them may reduce the level of security by supplying a potential attacker with known decrypted strings as well as a small set of encrypted ones. An attacker would know that two of the encrypted strings must decrypt to result in "." and ".." — and may try a known-plaintext attack.

Finally, we decided that along with file names, we will also encrypt directory names, symbolic links and the values they point to, and all other special files. The targets of symbolic links will always be encrypted — regardless if they point to "." or "..". These measures provide added security.

2.4 Mount Points

A stackable file system is similar to a loopback file system (lofs) in that the mount point and the directory mounted upon are separate. Cryptfs can provide transparent encryption for, say `/home/ezk/private` mounted on `/mnt/ezk`. Anyone accessing files directly through the mounted directory, `/home/ezk/private`, will see encrypted files and directories with nothing but normal Unix permissions to stop a potential attacker. Only access through the mount point, `/mnt/ezk`, by a valid authenticated user, will provide transparent decryption and encryption of data — which would still be subject to Unix permission checks.

Providing access to the "raw" encrypted files is important for backups: the backup operator should not have to decrypt files because it is CPU intensive and it is insecure to keep plaintext data on backup media. Having this access, however, provides an attacker who gains root privileges or the owner's privileges with the ability to corrupt data files or remove them. For this reason it was also desirable for Cryptfs to overlay the mounted directory with the mount point, making both of them the same. Since overlaying the mount point will prevent backups, we came up with a combined compromise solution: Cryptfs will overlay the mount point by default, will allow valid authenticated users to decrypt files using their keys, deny unauthenticated non-root users any access, and would otherwise behave like a read-only loopback file system to root users who did not provide a key to Cryptfs. In other words, unauthenticated root users who access files via Cryptfs would get to see their encrypted names and data, but will not be allowed to make any changes. This allows backups to proceed quickly and safely, and prevents attackers from corrupting data or removing files.

An additional design decision borne out of these was that the underlying storage must remain a valid file system of whatever type it was before. This is a must for backup programs and other tools to be able to browse the encrypted directories unabated.

3 Implementation

The implementation of Cryptfs proceeded based on the design with one notable exception: memory mapped operations complicated the code further and exposed some general problems with the vnode interface; these problems are discussed in detail in Section 3.4.

Our first implementation concentrated on the Solaris 2.5.1 operating system for several reasons:

- Solaris provides a standard vnode interface that is well understood and is not likely to change soon.
- We have access to kernel sources and other commercial quality file systems offered by Solaris 2.5.1.
- Solaris provides loadable kernel modules, making developing new file systems easier. This was a not a requirement for developing Cryptfs, but a practical decision that sped up the edit-compile-test-debug cycle.

To test a new or modified file system, we unmounted any existing instances of a previous implementation, unloaded the kernel module, loaded a new one, and remounted the file system.

- Solaris is a popular and well supported commercial operating system. Any work we do on such a system is likely to be used by others.

Our next two implementations were for the Linux 2.0 and FreeBSD 3.0 operating systems. We chose these two for the following reasons:

- Linux is sufficiently different than Solaris and it is also very popular; Linux uses a different vnode interface than Solaris. FreeBSD represents the pure BSD section of the Unix market, many variants of which exist.
- Full system sources are also available.
- Loadable kernel modules are supported.

By implementing Cryptfs for these three systems we hoped to prove that practical non-trivial stackable file systems are portable to sufficiently different Unix operating systems. The discussion in the rest of this section concentrates mostly on Solaris, unless otherwise indicated. In Section 3.5 we specifically discuss the differences in implementation between Linux and Solaris; Section 3.6 discusses the differences for the FreeBSD port.

3.1 A Wrapper File System

We began the implementation by creating a "wrapper" file system called "wrapfs" that was initially very similar to the Solaris loopback file system (lofs). Lofs passes all Vnode/VFS operations to the lower layer, but it only interposes on directory vnodes. Wrapfs interposes on every vnode, and makes identical copies of data blocks and pages in its own layer. The reason for this data copy was to make Wrapfs identical to Cryptfs with the exception of the actual encryption of bytes; this allowed us to measure the cost of full stacking separately from the cost of encryption.

3.2 In-Kernel Data Encryption

We used a reference implementation of the Blowfish encryption algorithm[1] and ported it to the kernel. Porting Blowfish to the kernel was easy. Most encryption code (written in C) uses simple arithmetic manipulations and uses very few system calls or C library functions. That made the Blowfish code highly portable.

Next we applied the encryption algorithm to the read and write vnode operations. As per the design, we perform encryption on whole blocks of size matching the UltraSPARC native page size (8192 bytes.) 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 interposed 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 depend on previous bytes, so we have to read and decode parts of pages before writing other parts of them.

Throughout the rest of this paper we will refer to the interposing (wrapping) vnode as V, and to the interposed (hidden or wrapped) vnode as V'. We use F to represent a file at the interposer's level and F' at the interposed one; P and P' refer to memory mapped pages at these two levels, respectively. The following example[2], 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) worth of bytes in it.

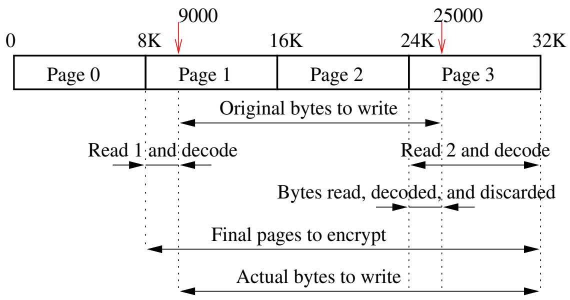

Figure 2: Writing Bytes in Cryptfs

1. An operation `write` is called on vnode V for the range 9000-25000.
2. Compute the extended page boundary for this range as 8192-32767 (3 pages.)
3. Locate the interposed-upon vnode V' from V.
4. Allocate 3 empty pages. (Page 0 of V is untouched.)
5. Read bytes 8192-8999 (page 1) from V', decrypt them, and place them in the first allocated page. We do not need to read or decode the bytes from 9000 onwards in page 1 because they will be overwritten by the data we wish to write anyway.
6. Skip any intermediate pages of data that will be overwritten by the overall write operation. In this case, we ignore page 2 of the opened file (which is also the second page in our write request.)
7. Read bytes 24576-32767 (page 3 of the file) from V', decrypt them, and place them in the third allocated page. This time we have to read and decrypt the whole page because we need the last $32767-25000=7767$ bytes and these bytes depend on the first $8192-7767=426$ bytes of that page.

[1] SSLeay from Eric A. Young

[2] simplified because it does not take into account sparse files.

8. Copy the bytes that were passed to us into the 3 allocated pages. The range of bytes in V' of 9000-25000 is trivially offset and copied into the allocated pages starting at offset $9000-8192=808$.
9. Finally, we have 3 valid data pages that contain unencrypted data. We encrypt the data and call the write operation on V' for the same starting offset (9000), but this time we write the bytes all the way to the last byte of the last page processed (byte 32767.) This is necessary to ensure validity of the data past file offset 25000.

3.2.1 Appending to Files

Files opened for appending only (using `O_APPEND`) do not provide the vnode interface write function any information regarding the real size of the file and where writing begins. If the size of the file before an append attempt is requested is not an exact multiple of a page size, data corruption will occur, since we will begin a new encryption sequence not on a page boundary.

The way we solve this problem is by detecting when a file is opened with an append flag on, turn off that flag before the open operation is passed on to V', and replace it with flags that indicate to V' that the file was opened for normal reading and writing. We save the initial state of the file opened, so that any other operation on V would be able to tell that this file was originally opened only for appending.

Whenever we write bytes to a file that was opened in append-only mode, we first apply the vnode `getattr` operation to find the true size of the file, and add that to the file offsets being written to. The interposed layer's vnode V' does not know that a file has been opened for append-only at the layer above it. For example, an append operation of 430 bytes to a file with an existing size of 260 bytes is converted into a write operation of bytes 261-690 of the whole file, and the procedure described in Section 3.2 continues unchanged.

3.3 File Names and Directory Reading

Every operation that deals with file names (for example "lookup") was modified such that the file name F is encrypted and uuencoded first. The modified file name F' is passed on to V'.

A complication we faced here was the "readdir" vnode operation. Readdir is implemented in the kernel as a restartable function. A user process calls the readdir C library call, which is translated to 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 respect to 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. The way we implemented `readdir` was as follows:

1. A readdir vnode operation is called on V for N bytes worth of directory data.
2. Call the same vnode operation on V' and read back N bytes.
3. Create a new temporary buffer of a size that is as large as N.
4. Loop over the bytes read from V', breaking them into individual records representing one directory entry at a time (`struct dirent`.) For each such entry, the file name within is uudecoded and then decrypted, resulting in the original file name. A new directory entry record is constructed with the decoded file name, and is added to the temporary buffer allocated.
5. We record the offset to read from on the next call to `readdir`; this is the position past the last file name we just read and decoded. This offset is stored in one of the fields of the `struct uio` (representing data movement between user and kernel space) that is returned to the caller. A new structure is passed to us upon the next invocation of readdir with the offset field untouched. This is how we were able to restart the call from the correct offset.
6. The temporary buffer is returned to the caller of the vnode operation. If there was more data to read from V', then the EOF flag would be set to false before returning from this function.

The caller of `readdir` asks to read at most N bytes. When we uudecode and decrypt the file names we reduce the size of the file name — converting every 4 byte sequence into 3. On average, file names are shortened by 25% from the encrypted and uuencoded forms. This means that the total number of bytes we pass back to the caller is less than N. That is all right, because the specifications for directory reading operations call for reading at most N bytes.

The only side effect of reducing the number of bytes returned to the caller is a small inefficiency in the overall number of times `readdir` must be called. Since we are returning less than N bytes, it is possible that the buffer supplied by the user process has enough space to fill in a few more directory entries. When taking into account large directories, it is possible that several `getdents(2)` system call invocations could be saved. With fewer context switches, overall performance could be improved. We did not deem this loss of performance significant since most Unix directories are not very large and the additional code to completely fill N bytes worth of data was going to complicate Cryptfs, increase the overhead, and inevitably lengthen the time required to develop it. (See Section 4.1 for an evaluation of the performance of Cryptfs.)

3.3.1 Multi-User Keys

Next we added support for multi-user keys as described in Section 2.1. Keys are set by a new `ioctl(2)` added to Cryptfs, and may be reset or removed by an authorized user within the same session. The user is prompted for a passphrase and the tool converts it into keys passed to Cryptfs.

We noticed a problem with listing directories containing file names that were encrypted using different keys. Decryption of the directory data occurs not using the original key but the key for the real UID of that session. When a string S is encrypted using key K_1 but decrypted using a different key K_2, the result is a completely different string that may contain any number of characters that are illegal in file names such as a forward-slash "/" or nulls. The latter confuse string operations that expect their input to be null terminated. We solved this problem by adding a 2-byte checksum to the encrypted strings and one more byte indicating original unencrypted length, before we uuencode them. After adding these 3 bytes, we uuencode the sequence using our special uuencoding algorithm. When strings are read from the interposed-upon file system during directory reading operations, we first uudecode them, then decrypt them, and validate the checksum. If it does not match, then we know this file name was encrypted using a different key and we skip the listing of that file altogether.

This solution not only avoids the possible data mishandling of file names, but also has a side effect of a slightly increased security "through obscurity." When a user lists a directory they only get to see files that they encrypted with the proper key. All other files are invisible and inaccessible to that user.

3.4 Memory Mapping

To be able to execute binaries we had to implement memory-mapping vnode functions. We had to decide if and how to cache memory mapped pages. In order to improve performance, we decided that Cryptfs shall have its own copy of cached decrypted pages in memory, and that we would leave the encrypted pages in the interposed-upon file system untouched.

3.4.1 Putpage and Getpage

When a page fault occurs, the kernel calls the vnode operation `getpage`. This function retrieves one or more pages from a file. We followed other implementations in Solaris and created a function that retrieves a single page — `getapage`; this function gets called repeatedly by one of the Paged Vnodes interface functions: `pvn_getpages`. The Paged Vnodes interface functions are not an explicit part of the vnode interface, but are there to assist it when dealing with memory mapped files. This unfortunate but necessary use of operating system specific interfaces exposed portability problems in our stackable file system.

The implementation of `getapage` appeared simple:

1. Check if the page is in the cache. If it is, return it.
2. If the page is not in the cache, create a new page P.
3. Find V' from V and call the `getpage` operation on V', making sure that the operation applied to V' would return only one page P'.
4. Copy the (encrypted) data from P' to P.
5. Map P into kernel virtual memory and decrypt its bytes in place.
6. Unmap P from kernel VM, insert it into V's cache, and return it.

Similarly, `putpage` was written using the Paged Vnodes interface functions. 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 had to 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 had 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 further blurred the distinction between the file system and the VM system.

3.4.2 Zero-Filled Pages

To save space, some Unix file systems support files with "holes" — pages that are not allocated on disk because they contain all zeros. When such a page is read into memory, the VM system fills it with zeros; only when it is modified does the page get physical disk space allocated. When we perform `getapage` on V', we expect to get a page that was previously encrypted with our algorithm and key. When we get a zero-filled page, we have no way of knowing that it was zero-filled and therefore should not be decrypted. (For a given key, there is a valid sequence of bytes that is not all-zeros, which when encrypted, will result in a sequence of all-zero bytes of the same length.) Our solution was to avoid holes in files. The only way to create a file with holes is to open it for write/append, `lseek(2)` past the end of the file, and write something there. We detected this condition in `write` and forced a write of all zeros (after encrypting them.)

3.5 Linux

When we began the Solaris work we referred to the implementation of other file systems such as lofs. Linux did not have one as part of standard distributions, but we were able to locate a prototype one and used it in our port. Also, the Linux Vnode/VFS interface 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 has been 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, the way Solaris moves data between user and kernel space is via structures called User I/Os (UIO). These 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, the file system handles one page at a time, leaving page clustering and multiple-page operations to the higher and more generic code. The Linux 2.0 kernel, however, does not include a `putpage` vnode operation (version 2.1 does.) We had to implement it using `write`. The `write` vnode operation uses different and somewhat incompatible arguments than `putpage`. We had to create the missing information and pass it to the `write()` operation. Since we store information vital for stacking in arguments passed to us, and we had to "fake" some of it here, this limited us to a single level of stacking file systems.

Linux's way of copying between user and kernel memory is a bit outmoded. Some operations default to manipulating user memory and some manipulate kernel memory. Operations that need to manipulate a different context have to set it before and restore it to its previous state afterwards. Solaris simply passes flags to these functions to tell them if to operate on kernel or user memory.

Directory reading was surprisingly simpler in Linux. In Solaris, we had to read a number of raw bytes from the interposed-upon 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 were only two caveats to the portability of the Linux code. First, Linux keeps a list of exported kernel symbols (in `kernel/ksyms.c`) available to loadable modules. In order to make Cryptfs a loadable module, we had to export additional symbols to the rest of the kernel, for functions mostly related to memory mapping. Secondly, 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, which we used 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`, which represents custom per-process virtual memory manager page-fault handlers. Since Cryptfs 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.

3.6 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' and the port was straightforward. FreeBSD's version of the loopback file system is called "nullfs"[12] — a useful template for writing stackable file systems. Two major deficiencies (bugs) in nullfs required attention. First, writing large files resulted in some data pages getting zero-filled on disk; this forced us to perform all writes synchronously. Secondly, memory mapping through nullfs panics the kernel, so we had to implement MMAP functions ourselves. We implemented `getpages` and `putpages` using `read` and `write`, respectively, because calling the lower-level's page functions resulted in a UFS pager error. (When we chose the latest snapshot of FreeBSD 3.0, we knew we were dealing with unstable code, and expected to face bugs.)

4 Evaluation

When evaluating Cryptfs, we compared it to CFS[2] (also using Blowfish) on all three systems and TCFS[3] on Linux ones.[3] The purpose of our work was primarily to create a practical and portable stackable file system. Performance and portability were more important to us than security because the design was such that stronger or weaker security measures could be put in place with relative ease.

We used two sets of performance tests. The first set measured specific common operations such as reading and writing files of various sizes. The second set used a more realistic test of building a large package inside the file system.

[3] All machines listed in this paper had 32MB RAM. The SPARC 5 ones ran at 85Mhz, and the x86 ones ran at 90Mhz. Solaris hosts included all recommended patches including the "jumbo" kernel patches 103640-20 (SPARC) and 103641-18 (x86). Linux systems were a vanilla RedHat 5.1 system. FreeBSD 3.0 machines were installed from the "980520" snapshot.

4.1 Performance

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 to. Since Cryptfs is a stackable file system, we also 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, Wrapfs, or Cryptfs, we mounted them over a local disk based file system. CFS and TCFS are based on NFS, so we included the performance of native NFS. All NFS mounts used the local host as both server and client (i.e. mounting `localhost:/path` on `/mnt`,) and used protocol version 2 over a UDP transport.

CFS is implemented as a user-level NFS file server. TCFS is a kernel module, but it accesses two user-level servers: `nfsd` and `xattrd`. Furthermore, the NFS server in Linux 2.0 is implemented completely in user level, further slowing down performance. As such, we expected that both CFS and TCFS would 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 writing. Lastly, TCFS does not support the Blowfish algorithm so we had to use DES instead; DES consumes more CPU resources than Blowfish.

For the first set of tests, we concentrated on the x86 platform since it was common to all ports. We ran tests that represent common operations in file systems: opening files for reading or writing. In the first test we wrote 1024 different new files of 8KB size each. The second test wrote 8 new files of 1MB size each. Then we read one 8KB file 1024 times, and one 1MB file 8 times. The intent of these tests was that the total amount of data read and written would be the same. Finally we included measurements for reading a directory with 1024 entries repeatedly for 100 times; while that is a less popular operation, cryptographic file systems encrypt file names and thus can significantly affect the performance of reading a directory. All times reported are elapsed, in seconds, and measured on an otherwise quiet system.

Since Linux is the only platform on which TCFS runs, we tested all file systems on it, as reported in Table 1. For Solaris and FreeBSD, we only included figures for the file systems relevant to comparing Cryptfs to CFS; these are reported in Tables 2 and 3, respectively.

Concentrating on Linux (Table 1) first, we see that lofs adds a small overhead over the native disk-based file system, and wrapfs adds another overhead due to stacking on all vnodes and due to performing data copies. The difference between Cryptfs and Wrapfs is that of encryption only. Writing files is 6-12 times faster in Cryptfs than in CFS/TCFS. The main reasons for this are the additional context switches that must take place in user-level file servers, and that NFS V.2 writes are synchronous. When reading files, caching and memory sizes come into play more than the file system in question. That is why the difference in file reading performance for all file systems is not as significant as when writing files. The reason lofs is slower than wrapfs is that the original lofs we used on Linux does not cache data, while Wrapfs and Cryptfs do. Reading a directory with 1024 files one hundred times is 10-37 times faster in Cryptfs than in TCFS or CFS, mostly due to context switches. When Cryptfs is mounted on top of ext2fs, it slows performance of these measured operations 2-3 times. But since these are fast to begin with, users hardly notice the difference; in practice overall slowness is smaller, as reported in Table 4.

File System	**Writes 1024× 8KB**	**Writes 8× 1MB**	**Reads 1024× 8KB**	**Reads 8× 1MB**	**1024× readdir**
ext2fs	3.33	3.06	0.17	0.34	1.49
lofs	3.40	3.35	0.30	0.34	1.51
wrapfs	3.48	3.58	0.18	0.34	1.57
cryptfs	9.27	8.33	0.26	0.34	3.18
nfs	26.85	17.67	0.47	3.17	16.27
cfs	101.90	50.84	0.89	8.77	118.35
tcfs	110.86	84.64	6.45	7.94	34.83

Table 1: Linux x86 Times for Repeated Calls (Sec)

File System	**Writes 1024× 8KB**	**Writes 8× 1MB**	**Reads 1024× 8KB**	**Reads 8× 1MB**	**1024× readdir**
ufs	4.88	3.98	0.48	0.38	0.52
cryptfs	63.95	11.10	10.72	7.23	7.14
nfs	54.17	18.82	1.69	0.38	0.28
cfs	140.78	140.98	27.68	24.57	18.02

Table 2: Solaris x86 Times for Repeated Calls (Sec)

File System	**Writes 1024× 8KB**	**Writes 8× 1MB**	**Reads 1024× 8KB**	**Reads 8× 1MB**	**1024× readdir**
ufs	12.55	6.04	1.00	1.01	0.15
cryptfs	56.59	22.55	1.04	1.05	0.29
nfs	55.69	21.63	1.31	1.09	0.33
cfs	99.34	31.80	2.09	4.80	0.87

Table 3: FreeBSD x86 Times for Repeated Calls (Sec)

Native file systems in Linux perform their operations asynchronously, while Solaris and FreeBSD do so synchronously. That is why the performance improvement of Cryptfs over CFS/TCFS for Solaris and FreeBSD is smaller; when writing vnode operations are passed from Cryptfs to the lower level file system, they must be completed before returning to the caller. For the operations measured, Cryptfs improves performance by anywhere from 50% to 2 times, with the exception of writing large files on Solaris, where performance is improved by more than an order of magnitude.

File System	SPARC		Intel P5		
	Solaris 2.5.1	**Linux 2.0.34**	**Solaris 2.5.1**	**Linux 2.0.34**	**FreeBSD 3.0**
ext2/ufs	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
nfs	1490.8	1440.1	1374.4	772.3	689.0
cfs	2168.6	1486.1	1946.8	839.8	827.3
tcfs	n/a	2092.3	n/a	1307.4	n/a

Table 4: Time to Build a Large Package (Sec)

For the next set of tests, we decided to use as our performance measure a full build of Am-utils[21], a new version of the Berkeley Amd automounter. The test auto-configures the package and then builds it. 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, a fair number of reads and lookups, and a few directory and symbolic link creations. We felt that is a more realistic measure of the overall file system performance, and would give users a better feel for the expected impact Cryptfs might have on their workstation. For each file system measured, we ran 10 successive builds on a quiet system, measured the elapsed times of each run, and averaged them. The results are summarized in Table 4. Results of TCFS are available on the only platform it runs, Linux. Also, there is no native lofs for FreeBSD (and the nullfs available is not fully functional.)

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. Wrapfs adds an overhead of 4.7-6.8% for Solaris and Linux systems, but that is comparable to the 3-10% degradation previously reported.[5, 19] On FreeBSD, however, Wrapfs adds an overhead of 21.1% compared to UFS; that is because of limitations of 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.

Wrapfs is used 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. 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.

Next we measure the overhead of CFS and TCFS and compare them to Cryptfs. When compared to NFS, TCFS is 45.3-69.3% slower on Linux, and CFS is 3.2-45.5% slower. Cryptfs is 40-52% faster than TCFS on Linux. Since TCFS uses DES and Cryptfs uses Blowfish, however, it would be fairer to compare Cryptfs to CFS. Cryptfs is 12-30% faster than CFS. In order to improve performance, CFS precomputes large stream ciphers for the attached directories. Cryptfs, on the other hand, has not been tuned or optimized yet.

Operating System	brk	Synchronous		Asynchronous	
		open	**unlink**	**open**	**unlink**
Solaris x86	813	59342	36216	1280	928
Solaris SPARC	176	15800	10236	862	517
Linux x86	86	-	-	286	106
Linux SPARC	108	-	-	300	162
FreeBSD x86	144	16448	2930	-	-
Frequency	890	185	3	185	3

Table 5: System Call Speeds and Frequencies for One Compilation (microseconds)

A complete and detailed comparison of various operating systems and the performance of their native file system is beyond the scope of this paper. Nevertheless, several interesting observations became apparent from Table 4. It was surprising to find that SPARC Linux surpassed the performance of every Solaris file system on the same SPARC architecture by 3.5-45.9%. Even the performance of the native disk-based performance was 13.2% faster on Linux, and NFS was 12.7% faster.

When we compare x86 based operating systems, Linux and FreeBSD appear comparable. NFS and CFS are 1.4-10.7% slower on Linux 2.0 because the NFS server is in user-level. On the other hand, all other file systems are 5.2-19.3% faster on Linux because of their asynchronous nature.

Solaris x86 is 78-132% slower that Linux. The asynchronous nature of Linux cannot explain why Solaris x86 is 71-136% slower than FreeBSD on identical hardware, since FreeBSD is also synchronous. To find the reason for this, we traced the system calls executing during several compilations of single C source file from the building of Am-utils. These are reported in Table 5. We found out that more than 95% of the time spent by the kernel on behalf of the compiler was spread among three system calls: `open`, `unlink`, and `brk` (to allocate more memory to a running process.) The most frequently called system call, `brk`, was called almost 900 times — more than 4 times the frequency of all other calls. We have found that the cost of a single call to `brk` on Solaris x86 is 813 microseconds; that is 5.6 times slower than FreeBSD, and almost 10 times slower than on Linux x86. Given the frequency of this call, and how slow it is on Solaris x86, it is not surprising that our compilations reported in Table 4 took twice as long on that platform. Luckily, memory allocation speed is not directly impacted by the file system in use.

To be certain, we turned off the default synchronous behavior of Solaris x86, and reran some tests. As expected, the speeds of `read` and `unlink` improved manyfold, and were brought more in line with their speeds on Linux and FreeBSD. Since `brk` is a more dominant call during compilation, and is unaffected by the asynchronous vs. synchronous nature of file systems, we did not expect turning off synchronous operations on Solaris x86 to result in significantly improved speeds. Indeed, when we ran a full build of Am-utils with Solaris x86 using asynchronous operations, the performance as reported in Table 4 improved by less than 6%.

4.2 Security and Ease-of-Use

We used the Blowfish[18] encryption algorithm with the default 128 bit key length. 56 bit encryption had already been broken and efforts are underway to break 64 bit encryption[9]. We felt that 128 bit keys were large enough to be secure for years to come, and at the same time they were not too large to require excessive CPU power. (Increasing the key size would be a simple matter of recompilation.) Blowfish is a newer cipher than DES[20], and as such has not survived the test of time that DES had, but Blowfish is believed to be very secure[18]. At this time, Cryptfs can only use Blowfish. CFS offers the widest choice of ciphers, including DES.

Cryptfs requires a session ID to use keys after they were supplied, to ensure that only processes in that session group get access to the key and thus to the unencrypted data. Requiring a session ID to use an encryption key prevents attackers from easily decrypting the data belonging to a valid user, even if the user's account was compromised on the same machine Cryptfs was mounted. All an attacker could do, even with root privileges, is read the ciphertext, but could not modify or remove files, since we overlay the mount point (see Section 2.4.) In comparison, CFS and TCFS are more vulnerable to the compromising of the users' accounts on the same host because they associate a key with a UID alone, not with a session ID.

Changing keys in Cryptfs is more cumbersome, since only one active key can be associated with a given session. Users have to use a special program we provide that sets a re-encryption key, reads files using old keys, and writes them back using the new key. CFS offers a more flexible solution that allows the user to change the passphrase without re-creating or copying the secure directory.

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 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. CFS' default mode uses no IVs, and we also felt that using a fixed one produces sufficiently strong security.

Both CFS and TCFS use auxiliary files to store encryption related information such as encrypted keys, types of encryption used, IVs[4], and more. CFS is the only file system that can store all of these on a more secure "smart card." TCFS uses login passwords as default keys and those are generally considered insecure and easily guessable. Cryptfs does not store any auxiliary information on disk, because we believe doing so could potentially create security vulnerabilities; it only has a modified instance of the key in use in kernel memory, for the duration of the session, or until the user chooses to remove it.

Since CFS and TCFS use NFS V.2, they are vulnerable to known (or yet undiscovered) attacks on NFS and related servers such as port mappers or mount protocol daemons. Cryptfs does not suffer from these problems.

All three cryptographic file systems suffer from a few similar yet unavoidable problems. At given times, cleartext keys and file data exist in the memory of user processes and in kernel memory and are thus vulnerable to an attack with root privileges through `/dev/kmem`; a sophisticated attacker aided by source access could read kernel memory to follow data structures representing processes, users, file systems, and vnodes — until he reaches plaintext keys and data. Also, if the system pages to an insecure device, there is a chance that parts of active processes containing cleartext will page to an insecure system, but that is not a Cryptfs problem per se.

We conducted a series of tests with a set of users such as letting them store mail and other important files under directories encrypted using Cryptfs. Over a period of one month, these users reported that they found Cryptfs to have a useful balancing of security and convenience. They liked Cryptfs' transparency; once the key was provided, they were able to perform normal file operations without noticing any difference or impact on their workstations.

4.3 Functionality and Portability

Cryptfs encrypts all file and directory data, names, and symbolic link values. In comparison, CFS does not support special files or named pipes. Sparse files' holes get filled in by Cryptfs and encrypted because it was a necessary part of the implementation; however, it also has the benefit of reducing the chance that an attacker could guess the type of file by noticing that it is sparse.

By encrypting and encoding file names into a reduced character set, cryptographic file systems lengthen file names on disk; this reduces the maximum allowed path name and component name lengths at the cryptographic

[4] CFS uses symbolic links to store IVs in the value of the link.

file systems' level. On average, TCFS and CFS halve the lengths of component names and maximum path names. Cryptfs reduces the length only by 25%. In practice, component names are 255 bytes long, and maximum path names are 1024 or more bytes long, so even a reduction of 50% in their lengths is not likely to seriously affect any users.

Solaris and FreeBSD have similar vnode interfaces, but both differ from Linux. Most of the Cryptfs code could not be directly shared between them. 20% of the code we wrote (about 4000 lines for Solaris) were general subroutines shared among all ports. The other 80% of the code was not all a "loss" when it came to other ports. Most of the vnode functions are very similar in behavior: find the interposed vnode V' from the current one V and apply the same vnode operation on V' in some order. The code looks different and the symbol names are not the same, but at its core the same stackable vnode operations occur in all three ports.

It took us almost a year to fully develop Wrapfs and Cryptfs together for Solaris, during which time we had to overcome our lack of experience with Solaris kernel internals and principles of stackable file systems. In comparison, we were able to complete the Linux port in under 3 weeks, and took one week to port to FreeBSD.

5 Related Work

There are two popular Cryptographic file systems in existence: Matt Blaze's CFS[2] and Cattaneo and Persiano's TCFS[3]. Both are compared to Cryptfs in Section 4. The other works mentioned below suffer from one of two problems: their performance is poor, or they are not portable enough to be readily available on systems used these days.

5.1 CFS

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. Any file or directory in a secure one is encrypted.

When users wish to use the secure directory, they run a tool and provide a private key to attach their encrypted directories via CFS. Then they access their cleartext files through the attached point. Once attached, file access is as transparent as any other directory. CFS determines if a user is allowed to access an attached directory based on the UID of the caller.

File data and meta data (symbolic links, file names, etc.) are encrypted. A wide choice of ciphers is available and great care was taken to ensure a high degree of security. A single CFS server can manage multiple directories as well as several users using different keys and ciphers per directory. CFS' performance is limited by the number of context switches that must be performed and the encryption algorithm used.

5.2 TCFS

TCFS[3] is a modified client-side NFS that communicates with a remote NFS server as well as a special RPC based attributes server. TCFS requires the installation of modules and tools on the client, as well as a special attributes daemon on the server. TCFS is available only for Linux systems; both client and server must run a Linux operating system.

TCFS offers a smaller choice of ciphers than CFS does, one of which must be chosen and statically compiled into the file system module. All files are encrypted with the same cipher. User keys default to login passwords that are less secure than passphrases. Encrypted user keys are stored in the file `/etc/tcfspasswd` which further reduces security. All files belonging to one user are encrypted using the same key.

TCFS allows individual files or directories to be encrypted by turning on or off a new and special flag 'X' on the file or directory in question. This provides finer grained control over which files should or should not be encrypted.

5.3 Truffles

Truffles[13] is a distributed file system that uses the Ficus system developed at UCLA to offer replication and file sharing[4, 6]. Truffles uses Privacy Enhanced Mail (PEM) as the method of securely communicating file data over a network. PEM provides authentication and encryption of the data. Truffles was designed to allow users to securely share files without special setup and with minimal system administrator intervention.

Ficus is not as readily available as other network based file systems such as NFS, nor is it very portable because it requires specialized operating system stackable layers support. Also, basing network negotiation on an electronic mail system results in long delays and significantly reduced performance.

5.4 Other Stackable File Systems

Several other operating systems offer a stackable file system interface. Such operating systems have the potential of easy development of file systems offering a much wider range of services than just encryption. Their main disadvantages are that they are not portable enough, not sufficiently developed or stable, or they are not available for common use. Also, new operating systems with new file system interfaces are not likely to perform as well as ones that are several years older.

The "Herd of Unix-Replacing Daemons" (HURD) 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 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[10]. 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.

One work that has resulted from Spring is the Solaris MC (Multi-Computer) File System[7]. It borrowed the object oriented interfaces from Spring and integrated them with the existing Solaris vnode interface to provide a distributed file system infrastructure through a special file system called the *pxfs* – the Proxy File System. Solaris MC provides all of the benefits that come with Spring, while requiring little or no change to existing file systems; those can be gradually ported over time. Solaris MC was designed to perform well in a closely coupled cluster environment (not a general network) and requires high performance networks and nodes.

6 Conclusions

Cryptfs proves that a useful, non-trivial vnode stackable file system can be implemented on modern operating systems without having to change the rest of the system. Better performance and stronger security were achieved by running the file system in the kernel. Cryptfs is more portable than other kernel-based file systems because it interacts with a mostly standard vnode interface, as the quick ports to Linux and FreeBSD showed.

Most complications discovered while developing Cryptfs stemmed from two problems:

1. The vnode interface is not self-contained. The VM system offers memory mapped files, but properly handling them requires some manipulation of lower level file systems and MMU/TLB hardware.
2. Several vnode calls are poorly designed, most likely due to the need to keep compatible with past implementations that were made when resources were more scarce. The `readdir` vnode operation on Solaris and FreeBSD requires special parsing of the file names within the data provided by the file system. Also, it forced us to maintain lots of state to be able to implement it as a restartable operation.

We believe that a truly stackable file system interface could significantly improve portability, especially if adopted by the main Unix vendors and developers. We think that the interface available in Spring[10] is very suitable. If that interface becomes popular, it might result in many more practical file systems developed. We hope through Cryptfs to have proven the usefulness and practicality of non-trivial stackable file systems.

6.1 Future

We plan to add Cryptfs support for other ciphers, especially DES. We also intend to port Cryptfs to newer versions of existing operating systems (Solaris 2.7 and Linux 2.1 in particular,) to take advantage of new system features offered.

The work described in this paper is part of an ongoing research effort to develop "FiST" (File System Translator) — a system that will be used to describe a file system using a high-level language and generate a working implementation for the target operating system from that description[22].

7 Acknowledgments

We would like to thank Fred Korz, Seth Robertson, and especially Dan Duchamp for their help in reviewing this paper and offering concrete suggestions that made Cryptfs better. This work was partially made possible thanks to NSF infrastructure grants numbers CDA-90-24735 and CDA-96-25374.

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] 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.

[4] 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.

[5] 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.

[6] 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.

[7] V. Matena, Y. A. Khalidi, and K. Shirriff. Solaris MC File System Framework. Technical Reports TR-96-57. Sun Labs, October 1996. Available http://www.sunlabs.com/technical-reports/1996/abstract-57.html.

[8] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[9] D. McNett. Secure Encryption Challenged by Internet-Linked Computers. Press Release. distributed.net, 23 February 1998. Available http://www.distributed.net/pressroom/DESII-1-PR.html.

[10] J. G. Mitchel, J. J. Giobbons, 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.

[11] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS Version 3 Design and Implementation. *USENIX Conference Proceedings* (Boston, Massachusetts), pages 137–52. USENIX, 6-10 June 1994.

[12] 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.

[13] P. Reiner, T. Page, G. Popek, J. Cook, and S. Crocker. Truffles – A Secure Service For Widespread File Sharing. *Proceedings of the Privacy and Security Research Group Workshop on Network and Distributed System Security*. PSRG, 1994. Available via ftp in ftp://ftp.cs.ucla.edu/pub/ficus/psrg93.ps.gz.

[14] R. L. Rivest. The MD5 Message-Digest Algorithm. RFC 1321. Internet Activities Board, April 1992.

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

[16] 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.

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

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

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

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

[21] 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/.

[22] 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/ or http://www.cs.columbia.edu/~ezk/research/.

[23] P. R. Zimmerman. Pretty Good Privacy. In *The Official PGP User's Guide*. MIT Press, April 1995.

8 Author Information

Erez Zadok is an PhD candidate in the Computer Science Department at Columbia University. He received his B.S. in Comp. Sci. in 1991, and his M.S. degree in 1994, both from Columbia U. His primary interests include operating systems and file systems. The work described in this paper was first mentioned in his PhD thesis proposal[22].

Ion Badulescu is a staff associate at the computer science department. He is also a B.A. candidate at Columbia University and is expected to graduate in May 1999. His primary interests include operating systems, networking, compilers, and languages.

Alex Shender holds the position of manager of the computer facilities at Columbia University's Computer Science Department. His primary interests include operating systems, networks, and system administration. In May 1998 he received his B.S. in Comp. Sci. from Columbia's School of Engineering and Applied Science.

Mailing address for all authors: Columbia University, Mail Code 0401, 1214 Amsterdam Avenue, New York, NY 10027. Cryptfs is available to U.S. citizens from *http://www.cs.columbia.edu/~ezk/research/cryptfs/*.