A Cryptographic File System for Unix

Matt Blaze

AT&T Bell Laboratories
101 Crawfords Corner Road, Room 4G-634
Holmdel, NJ 07733

mab@research.att.com

Abstract

Although cryptographic techniques are playing an increasingly important role in modern computing system security, user-level tools for encrypting file data are cumbersome and suffer from a number of inherent vulnerabilities. The Cryptographic File System (CFS) pushes encryption services into the file system itself. CFS supports secure storage at the system level through a standard Unix file system interface to encrypted files. Users associate a cryptographic key with the directories they wish to protect. Files in these directories (as well as their pathname components) are transparently encrypted and decrypted with the specified key without further user intervention; cleartext is never stored on a disk or sent to a remote file server. CFS can use any available file system for its underlying storage without modification, including remote file servers such as NFS. System management functions, such as file backup, work in a normal manner and without knowledge of the key.

This paper describes the design and implementation of CFS under Unix. Encryption techniques for file system-level encryption are described, and general issues of cryptographic system interfaces to support routine secure computing are discussed.

1. Introduction

Data security in modern distributed computing systems is a difficult problem. Network connections and remote file system services, while convenient, often make it possible for an intruder to gain access to sensitive data by compromising only a single component of a large system. Because of the difficulty of reliably protecting information, sensitive files are often not stored on networked computers, making access to them by authorized users inconvenient and putting them out of the reach of useful system services such as backup. (Of course, off line backups are themselves a security risk, since they make it difficult to destroy all copies of confidential data when they are no longer needed.) In effect, the (often well founded) fear that computer data are not terribly private has led to a situation where conventional wisdom warns us not to entrust our most important information to our most modern computers.

Cryptographic techniques offer a promising approach for protecting files against unauthorized access. When properly implemented and appropriately applied, modern cipher algorithms (such as the Data Encryption Standard (DES)[5] and the more recent IDEA cipher[4]) are widely believed sufficiently strong to render encrypted data unavailable to virtually any adversary who cannot supply the correct key. However, routine use of these algorithms to protect file data is uncommon in current systems. This is partly because file encryption tools, to the extent they are available at all, are often poorly integrated, difficult to use, and vulnerable to non-cryptoanalytic system level attacks. We believe that file encryption is better handled by the file system itself. This paper investigates the implications of cryptographic protection as a basic feature of the file system interface.

1.1. User-Level Cryptography Is Cumbersome

The simplest approach for file encryption is through a tool, such as the Unix `crypt` program, that enciphers (or deciphers) a file or data stream with a specified key. Encryption and decryption are under the user's direct control. Depending on the particular software, the program may or may not automatically delete the cleartext when encrypting, and such programs can usually also be used as cryptographic "filters" in a command pipeline.

Another approach is integrated encryption in application software, where each program that is to manipulate sensitive data has built-in cryptographic facilities. For example, a text editor could ask for a key when a file is opened and automatically encrypt and decrypt the file's data as they are written and read. All applications that are to operate on the same data must, of course, include the same encryption engine. An encryption filter, such as `crypt`, might also be provided to allow data to be imported into and exported out of other software.

Unfortunately, neither approach is entirely satisfactory in terms of security, generality, or convenience. The former approach, while allowing great flexibility in its application, invites mistakes; the user could inadvertently fail to encrypt a file, leaving it in the clear, or could forget to delete the cleartext version after encryption. The manual nature of the encryption and the need to supply the key several times whenever a file is used make encryption too cumbersome for all but the most sensitive of files. More seriously, even when used properly, manual encryption programs open a window of vulnerability while the file is in clear form. It is almost impossible to avoid occasionally storing cleartext on the disk and, in the case of remote file servers, sending it over the network. Some applications simply expect to be able to read and write ordinary files.

In the application-based approach, each program must have built-in encryption functionality. Although encryption takes place automatically, the user still must supply a key to each application, typically when it is invoked or when a file is first opened. Software without encryption capability cannot operate on secure

This is a pre-print of a paper to be presented at the First ACM Conference on Communications and Computing Security, Fairfax, VA, November 3-5, 1993.

data without the use of a separate encryption program, making it hard to avoid all the problems outlined in the previous paragraph. Furthermore, rather than being confined to a single program, encryption is spread among multiple applications, each of which must be trusted to interoperate securely and correctly with the others. A single poorly designed component can introduce a significant and difficult to detect window of vulnerability. (For example, some versions of the Unix editor `vi` can encrypt files but still leave temporary data in the clear.) Changing the encryption algorithm entails modification of every program that uses it, creating many opportunities for implementation errors. Finally, multiple copies of user-level cryptographic code can introduce a significant performance penalty.

1.2. System-Level Cryptography Is Often Insufficient

One way to avoid many of the pitfalls of user-level encryption is to make cryptographic services a basic part of the underlying system. In designing such a system, it is important to identify exactly what is to be trusted with cleartext and what requires cryptographic protection. In other words, we must understand what components of the system are vulnerable to compromise.

In general, the user has little choice but to trust *some* components of the system, since the whole point of storing data on a computer is to perform various operations on the cleartext. Ideally, however, required trust should be limited to those parts of a system that are under the user's direct control.

For files, we are usually interested in protecting the physical media on which sensitive data are stored. This includes online disks as well as backup copies (which may persist long after the on-line versions have been deleted). In distributed file server-based systems, it is often also desirable to protect the network connection between client and server since these links may be very easy for an eavesdropper to monitor. Finally, it is possible that the user may not trust the file server itself, especially when it is physically or administratively remote.

Physical media can be protected by specialized hardware. Disk controllers are commercially available with embedded encryption hardware that can be used to encipher entire disks or individual file blocks with a specified key. Once the key is provided to the controller hardware, encryption is completely transparent. This approach has a number of disadvantages for general use, however. The granularity of encryption keys must be compatible with the hardware; often, the entire disk must be thought of as a single protected entity. It is difficult to share resources among users who are not willing to trust one another with the same key. Obviously, this approach is only applicable when the required hardware is available. Backups remain a difficult problem. If the backups are taken of the raw, undecrypted disk, it may be difficult to restore files reliably should the disk controller hardware become unavailable, even when the keys are known. If the backup is taken of the cleartext data the backup itself will require separate cryptographic protection. Finally, this approach does not protect data going into and out of the disk controller itself, and therefore may not be sufficient for protecting data in remote file servers.

Network connections between client machines and file servers can be protected with end-to-end encryption and cryptographic authentication. Again, specialized hardware may be employed for this purpose, depending on the particular network involved, or it may be implemented in software. Not all networks support encryption, however, and among those that do, not all system vendors supply working implementations of encryption as a standard product.

Even when the various problems with media and network level encryption are ignored, the combination of the two approaches may not be adequate for the protection of data in modern distributed systems. In particular, even though cleartext may never be stored on a disk or sent "over the wire", sensitive data can be leaked if the file server itself is compromised. The file server must maintain, at some point, the keys used to encipher both the disk and the network. Even if the server can be completely trusted, direct media encryption on top of network encryption has a number of shortcomings from the point of view of efficient distributed system design. Observe that each file access requires two cryptographic operations by the server, once for the network and once for the disk, even though the server itself never makes use of cleartext data. Such a design violates the principle that work should be shifted from the (shared, heavily loaded) file server to the (unshared, lightly loaded) client machine whenever possible[1]. Even if the cryptographic operations are themselves implemented in hardware, additional server software complexity is still required to support them.

Several commercial and research systems incorporate cryptographic techniques for protecting file data against various kinds of attack. In the personal computer (e.g., MS-DOS, Macintosh) world, there are file encryption systems that can create an "encrypted area" on a disk. These packages generally require the preallocation of storage space to a given key, and often support only a particular kind of storage media (such as a local hard disk). Encrypted files typically appear outside the system as a single large file and therefore cannot be readily managed by conventional administration tools or moved to arbitrary storage devices. In larger-scale systems, cryptographic techniques are even less widely used, although a few systems do use encryption for protecting certain vulnerable interfaces. The Truffles system[7], for example, uses a combination of cryptographic authentication and secret-key encryption to protect network access to widely distributed shared files. The files themselves, however, are stored at the server in clear form.

In the following sections, we describe the alternative approach taken by the Cryptographic File System (CFS). CFS pushes file encryption entirely into the client file system interface, and therefore does not suffer from many of the difficulties inherent in user-level and disk and network based system-level encryption.

2. CFS: Cryptographic Services in the File System

CFS investigates the question of where in a system responsibility for file encryption properly belongs. As discussed in the previous section, if encryption is performed at too low a level, we introduce vulnerability by requiring trust in components that may be far removed from the user's control. On the other hand, if encryption is too close to the user, the high degree of human interaction required invites errors as well as the perception that cryptographic protection is not worth the trouble for practical, day-to-day use. CFS is designed on the principle that the trusted components of a system should encrypt immediately before sending data to untrusted components.

2.1. Design Goals

CFS occupies something of a middle ground between low-level and user-level cryptography. It aims to protect exactly those aspects of file storage that are vulnerable to attack in a way that is convenient enough to use routinely. In particular, we are guided by the following specific goals:

- Rational key management. Cryptographic systems restrict access to sensitive information through knowledge of the keys used to encrypt the data. Clearly, to be of any use at all, a system must have some way of obtaining the key from the user. But this need not be intrusive; encryption keys should not have to be supplied more than once per session. Once a key has been entered and authenticated, the user should not be asked to supply it again on subsequent operations that can be reliably associated with it (e.g., originating from the same keyboard). Of course, there should also be some way to manually destroy or remove from the system a supplied key when it is not in active use.

- Transparent access semantics. Encrypted files should behave no differently from other files, except in that they are useless without the key. Encrypted files should support the same access methods available on the underlying storage system. All system calls should work normally, and it should be possible to compile and execute in a completely encrypted environment.
- Transparent performance. Although cryptographic algorithms are often somewhat computationally intensive, the performance penalty associated with encrypted files should not be so high that it discourages their use. In particular, interactive response time should not be noticeably degraded.
- Protection of file contents. Clearly, the data in files should be protected, as should structural data related to a file's contents. For example, it should not be possible to determine that a particular sequence of bytes occurs several times within a file, or how two encrypted files differ.
- Protection of sensitive meta-data. Considerable information can often be derived from a file system's structural data; these should be protected to the extent possible. In particular, file names should not be discernible without the key.
- Protection of network connections. Distributed file systems make the network an attractive target for obtaining sensitive file data; no information that is encrypted in the file system itself should be discernible by observation of network traffic.
- Natural key granularity. The grouping of what is protected under a particular key should mirror the structural constructs presented to the user by the underlying system. It should be easy to protect related files under the same key, and it should be easy to create new keys for other files. The Unix directory structure is a flexible, natural way to group files.
- Compatibility with underlying system services. Encrypted files and directories should be stored and managed in the same manner as other files. In particular, administrators should be able to backup and restore individual encrypted files without the use of special tools and without knowing the key. In general, untrusted parts of the system should not require modification.
- Portability. The encryption system should exploit existing interfaces wherever possible and should not rely on unusual or special-purpose system features. Furthermore, encrypted files should be portable between implementations; files should be usable wherever the key is supplied.
- Scale. The encryption engine should not place an unusual load on any shared component of the system. File servers in particular should not be required to perform any special additional processing for clients who require cryptographic protection.
- Concurrent access. It should be possible for several users (or processes) to have access to the same encrypted files simultaneously. Sharing semantics should be similar to those of the underlying storage system.
- Limited trust. In general, the user should be required to trust only those components under his or her direct control and whose integrity can be independently verified. It should not, for example, be necessarily to trust the file servers from which storage services are obtained. This is especially important in large-scale environments where administrative control is spread among several entities.
- Compatibility with future technology. Several emerging technologies have potential applicability for protecting data. In particular, keys could be contained in or managed by "smart cards" that would remain in the physical possession of authorized users. An encryption system should support, but not require, novel hardware of this sort.

2.2. CFS Functionality and User Interface

An important goal of CFS is to present the user with a secure file service that works in a seamless manner, without any notion that encrypted files are somehow "special", and without the need to type in the same key several times in a single session. Most interaction with CFS is through standard file system calls, with no prominent distinction between files that happen to be under CFS and those that are not.

CFS provides a transparent Unix file system interface to directory hierarchies that are automatically encrypted with user supplied keys. Users issue a simple command to "attach" a cryptographic key to a directory. Attached directories are then available to the user with all the usual system calls and tools, but the files are automatically encrypted as they are written and decrypted as they are read. No modifications of the file systems on which the encrypted files are stored are required. File system services such as backup, restore, usaage accounting, and archival work normally on encrypted files and directories without the key. CFS ensures that cleartext file contents and name data are never stored on a disk or transmitted over a network.

CFS presents a "virtual" file system on the client's machine, typically mounted on `/crypt`, through which users access their encrypted files. The attach command creates entries in CFS (which appear in `/crypt`) that associate cryptographic keys with directories elsewhere in the system name space. Files are stored in encrypted form and with encrypted path names in the associated standard directories, although they appear to the user who issued the attach command in clear form under `/crypt`. The underlying encrypted directories can reside on any accessible file system, including remote file servers such as Sun NFS[8] and AFS[1]. No space needs to be preallocated to CFS directories. Users control CFS through a small suite of tools that create, attach, detach, and otherwise administer encrypted directories.

Each directory is protected by set of cryptographic keys. These keys can be supplied by user entry via the keyboard or, if hardware is available, through removable "smart cards" connected to the client computer. When entered from the keyboard, keys take the form of arbitrary-length "passphrases" which are used to generate the set of internal cryptographic keys used by CFS's encryption routines. Passphrases must be of sufficient length to allow the creation of several independent keys; the current implementation requires at least 16 characters. Phrases may include any printable ASCII characters, and ideally consist of easily remembered nonsense sentences with unusual punctuation, capitalization and spelling (e.g., `"if you have nothing 2 hide you Have nothing too fear!"`). In the smart card-based system, the keys are copied directly from the card interface to the client computer after user entry of a card access password that is checked on the card itself. Section 3 describes the algorithms used to encrypt file contents and file names with the keys.

The `cmkdir` command is used to create encrypted directories and assign their keys. Its operation is similar to that of the Unix `mkdir` command with the addition that it asks for a key. In the examples that follow, we show dialogs for the "passphrase" version; the smart card version is similar but with prompts to the user to insert a card and enter its password. The following dialog creates an encrypted directory called `/usr/mab/secrets`:

```
$ cmkdir /usr/mab/secrets
Key: (user enters passphrase, which does not echo)
Again: (same phrase entered again to prevent errors)
$
```

To use an encrypted directory, its key must be supplied to CFS with the `cattach` command. `cattach` takes three parameters: an encryption key (which is prompted for), the name of a directory previously created with `cmkdir`, and a name that will be used to access the directory under the CFS mount point. For example, to attach the directory created above to the name `/crypt/matt`:

```
$ cattach /usr/mab/secrets matt
Key: (same key used in the cmkdir command)
$
```

If the key is supplied correctly, the user "sees" `/crypt/matt` as a normal directory; all standard operations (creating, reading, writing, compiling, executing, `cd`, `mkdir`, etc.) work as expected. The actual files are stored under `/usr/mab/secrets`, which would not ordinarily be used directly. Consider the following dialog, which creates a single encrypted file:

```
$ ls -l /crypt
total 1
drwx------ 2 mab 512 Apr 1 15:56 matt
$ echo "murder" > /crypt/matt/crimes
$ ls -l /crypt/matt
total 1
-rw-rw-r-- 1 mab   7 Apr 1 15:57 crimes
$ cat /crypt/matt/crimes
murder
$ ls -l /usr/mab/secrets
total 1
-rw-rw-r-- 1 mab  15 Apr 1 15:57 8b06e85b87091124
$ cat -v /usr/mab/secrets/8b06e85b87091124
M-Z,k^]^B^VM-VM-6A~uM-LM-_M-DM-^[
$
```

When the user is finished with an encrypted directory, its entry under `/crypt` can be deleted with the `cdetach` command. Of course, the underlying encrypted directory remains and may be attached again at some future time.

```
$ cdetach matt
$ ls -l /crypt
total 0
$ ls -l /usr/mab/secrets
total 1
-rw-rw-r-- 1 mab  15 Apr 1 15:57 8b06e85b87091124
$
```

File names are encrypted and encoded in an ASCII representation of their binary encrypted value padded out to the cipher block size of eight bytes. Note that this reduces by approximately half the maximum path component and file name size, since names stored on the disk are twice as long as their clear counterparts. Encrypted files may themselves be expanded to accommodate cipher block boundaries, and therefore can occupy up to one eight byte encryption block of extra storage. Otherwise, encrypted files place no special requirements on the underlying file system.

Encrypted directories can be backed up along with the rest of the file system. The `cname` program translates back and forth between cleartext names and their encrypted counterparts for a particular key, allowing the appropriate file name to be located from backups if needed. If the system on which CFS is running should become unavailable, encrypted files can be decrypted individually, given a key, using the `ccat` program. Neither `cname` nor `ccat` require that the rest of CFS be running or be installed, and both run without modification under most Unix platforms. This helps ensure that encrypted file contents will always be recoverable, even if no machine is available on which to run the full CFS system.

2.3. Security and Trust Model

Most security mechanisms in computer systems are aimed at authenticating the users and clients of services and resources. Servers typically mistrust those who request services from them, and the protocols for obtaining access typically reflect the security needs of the server. In the case of a file system, the converse relationship is true as well; the user must be sure that the file system will not reveal private data without authorization. File encryption can be viewed as a mechanism for enforcing mistrust of servers by their clients.

CFS protects file contents and file names by guaranteeing that they are never sent in clear form to the file system. When run on a client machine in a distributed file system, this protection extends to file system traffic sent over the network. In effect, it provides end-to-end encryption between the client and the server without any actual encryption required at the server side. The server need only be trusted to actually store (and eventually return) the bits that were originally sent to it. Of course, the user must still trust the client system on which CFS is running, since that system manages the keys and cleartext for the currently attached encrypted directories.

Some data are not protected, however. File sizes, access times, and the structure of the directory hierarchy are all kept in the clear. (Symbolic link pointers are, however, encrypted.) This makes CFS vulnerable to traffic analysis from both real-time observation and snapshots of the underlying files; whether this is acceptable must be evaluated for each application.

It is important to emphasize that CFS protects data only in the context of the file system. It is not, in itself, a complete, general purpose cryptographic security system. Once bits have been returned to a user program, they are beyond the reach of CFS's protection. This means that even with CFS, sensitive data might be written to a paging device when a program is swapped out or revealed in a trace of a program's address space. Systems where the paging device is on a remote file system are especially vulnerable to this sort of attack. (It is theoretically possible to use CFS as a paging file system, although the current implementation does not readily support this in practice.) Note also that CFS does not protect the links between users and the client machines on which CFS runs; users connected via networked terminals remain vulnerable if these links are not otherwise secured.

Access to attached directories is controlled by restricting the virtual directories created under `/crypt` using the standard Unix file protection mechanism. Only the user who issued the `cattach` command is permitted to see or use the cleartext files. This is based on the *uid* of the user; an attacker who can obtain access to a client machine and compromise a user account can use any of that user's currently attached directories. If this is a concern, the attached name can be marked *obscure,* which prevents it from appearing in a listing of `/crypt`. When an attach is made obscure, the attacker must guess its current name, which can be randomly chosen by the real user. Of course, attackers who can become the "superuser" on the client machine can thwart any protection scheme, including this; such an intruder has access to the entire address space of the kernel and can read (or modify) any data anywhere in the system.

The security of the system is largely dependent on the secrecy of the encryption keys and the inability of an attacker to guess them. Although an exhaustive search of the key space is probably computationally infeasible to all but the most determined and well funded adversary, poorly chosen keys can make the attacker's job much easier. This risk is especially great when keys are chosen directly by the user. To reduce the risk of dictionary-based attacks, and to provide enough entropy to generate several independent subkeys, passphrase-based keys must be fairly lengthy. The `cmkdir` program can be easily modified to enforce passphrase selection rules, such as minimum length and alphabetical variety, that promote the use of good keys. Passphrase-based keys also carry a risk of compromise through user carelessness or "social engineering"; these risks can be reduced somewhat with user training.

The smart card based system uses the cards themselves to generate and store the actual encryption keys; here the user passphrase is used only to control access to the card. Note that it is theoretically possible to design a system in which the keys never leave the smart card and all cryptographic operations are performed on the card itself. CFS, however, transfers the keys from the card to the client machine and performs file encryption there, since the bandwidth to generally available card interfaces is too low for file system use.

We discuss possible attacks against our prototype implementation in Section 4, below.

3. File Encryption

CFS uses DES to encrypt file data. DES has a number of standard modes of operation[6], none of which is completely suitable for encrypting files on-line in a file system. In the simplest DES mode, *ECB (electronic code book),* each 8 byte block of a file is independently encrypted with the given key. Encryption and decryption can be performed randomly on any block boundary. Although this protects the data itself, it can reveal a great deal about a file's structure − a given block of cleartext always encrypts to the same ciphertext, and so repeated blocks can be easily identified as such. Other modes of DES operation include various *chaining* ciphers that base the encryption of a block on the data that preceded it. These defeat the kinds of structural analysis possible with ECB mode, but make it difficult to randomly read or write in constant time. For example, a write to the middle of a file could require reading the data that preceded it and reenciphering and rewriting the data that follow it. Unix file system semantics, however, require approximately uniform access time for random blocks of the file.

Compounding this difficulty are concerns that the 56 bit key size of DES is vulnerable to exhaustive search of the key space. DES keys can be made effectively longer by multiple encryption with independently chosen 56 bit keys. Unfortunately, DES is computationally rather expensive, especially when implemented in software. It is likely that multiple on-line iterations of the DES algorithm would be prohibitively slow for file system applications.

To allow random access to files but still discourage structural analysis and provide greater protection than a single iteration ECB mode cipher, CFS encrypts file contents in two ways. Recall that CFS keys are long "passphrases". When the phrase is provided at attach time, it is "crunched" into two separate 56 bit DES keys. The first key is used to pre-compute a long (half megabyte) pseudo-random bit mask with DES's *OFB (output feed back)* mode. This mask is stored for the life of the attach. When a file block is to be written, it is first exclusive-or'd (XOR) with the part of the mask corresponding to its byte offset in the file modulo the precomputed mask length. The result is then encrypted with the second key using standard ECB mode. When reading, the cipher is reversed in the obvious manner: first decrypt in ECB mode, then XOR with the positional mask. Observe that this allows uniform random access time across the entire size of the pre-computed mask (but not insertion or deletion of text). File block boundaries are preserved as long as the cipher block size is a multiple of the block size, as it is in most systems. Applications that optimize their file I/O to fall on file system block boundaries (including programs using the Unix `stdio` library) therefore maintain their expected performance characteristics without modification.

This combination of DES modes guarantees that identical blocks will encrypt to different ciphertext depending upon their positions in a file. It does admit some kinds of structural analysis across files, however. It is possible to determine which blocks are identical (and in the same place) in two files encrypted under the same key (e.g., in the same directory hierarchy).

The strength of the ECB+OFB scheme is not well analyzed in the literature (it may be new − there appear to be no previous references to this technique), and such an analysis is beyond the scope of this paper. However, at a minimum, it is clear that the protection against attack is at least as strong as a single DES pass in ECB mode and may be as strong as two passes with DES stream mode ciphers. It is likely that the scheme is weakened, in that the attacker might be able to search for the two DES subkeys independently, if there are several known plaintext files encrypted under the same keys.

To thwart analysis of identical blocks in the same positions of different files, each file encrypted under the same key can be perturbed with a unique "initial vector" (IV). Standard block-chaining encryption modes (such as CBC) prepend the IV to the data stream (e.g. the beginning of the file) and XOR successive blocks with the ciphertext of the previous block, starting with the IV. As long as each file has a different IV, identical blocks will encrypt differently. Unfortunately, the chaining modes do not permit random update within a file, so an encrypting file system cannot use them directly. Instead, CFS simply XORs each cipher block with the same IV throughout the file prior to the final ECB mode encryption or after the ECB decryption, just as with the OFB mask.

File IVs are generated from the inode number of the underlying file at creation time, which generally remains unique for a file's lifetime. Since the IV is required for decryption, it must be stored along with the file. Inode numbers may not be preserved after a file system backup/restore operation, so it is not sufficient to rely on the inode number remaining constant over time. The IV could be stored at the beginning of each file, but that would shift file contents away from block boundaries or, if padded out to an entire block, waste space. The natural place to store the IV would be in the inode itself, along with the file's other attributes. However, because CFS sits above the file system used for actual storage and uses system calls for all its I/O, it has no direct access to inode fields and cannot therefore add new file attributes. To store the IV, CFS must co-opt an existing inode field.

Since all of the inode fields are used for some purpose and can be changed outside of CFS's control, using an existing field is a rather treacherous proposition. CFS therefore offers two modes of encryption that are selected at the time an encrypted directory is created. In the standard mode, no IV is used and files are therefore subject to analysis of identical blocks. In the second, "high security" mode, the IV is stored in the group id (`gid`) field of each file's inode. In this mode, CFS reports the group ownership of the root directory of the encrypted hierarchy as the group for each file within it; it is not possible to have different files with different group ownership in the same directory. Note that a file's group could be changed outside CFS, so this mode does carry a small risk of unrecoverable data if both the inode number and group of a file change. Standard backup and restore procedures, however, do ordinarily preserve the group.

Encryption of pathname components uses a similar scheme, with the addition that the high order bits of the cleartext name (which are normally zero) are set to a simple checksum computed over the entire name string. This frustrates structural analysis of long names that differ only in the last few characters. The same method is used to encrypt symbolic link pointers.

4. Prototype Implementation

Of considerable practical significance is whether the performance penalty of on-line file system encryption is too great for routine use. The prototype CFS implementation is intended to help answer this question as well as provide some experience with practical applications of secure file storage.

4.1. Architecture

The CFS prototype is implemented entirely at user level, communicating with the Unix kernel via the NFS interface. Each client machine runs a special NFS server, `cfsd` (CFS Daemon), on its *localhost* interface, that interprets CFS file system requests. At boot time, the system invokes `cfsd` and issues an NFS `mount` of its localhost interface on the CFS directory (`/crypt`) to start CFS. (To allow the client to also work as a regular NFS server, CFS runs on a different port number from standard NFS.)

The NFS protocol is designed for remote file servers, and so assumes that the file system is very loosely coupled to the client (even though, in CFS's case, they are actually the same machine). The client kernel communicates with the file system through 17 *remote procedure calls (RPCs)* that implement various file system-related primitives (read, write, etc.). The server is *stateless,* in that it is not required to maintain any state data between individual client calls.

NFS clients cache file blocks to enhance file system performance (reducing the need to issue requests to the server); a simple protocol managed by the client maintains some degree of cache consistency. All communication is initiated by the client, and the server can simply process each RPC as it is received and then wait for the next. Most of the complexity of an NFS implementation is in the generic client side of the interface, and it is therefore often possible to implement new file system services entirely by adding a simple NFS server.

`cfsd` is implemented as an RPC server for an extended version of the NFS protocol. Additional RPCs attach, detach, and otherwise control encrypted directories. Initially, the root of the CFS file system appears as an empty directory. The `cattach` command sends an RPC to `cfsd` with arguments containing the full path name of a directory (mounted elsewhere), the name of the "attach point", and the key. If the key is correct (as verified by a special file in the directory encrypted with a hash of the supplied key), `cfsd` computes the cryptographic mask (described in the previous section) and creates an entry in its root directory under the specified attach point name. The attach point entry appears as a directory owned by the user who issued the attach request, with a protection mode of 700 to prevent others from seeing its contents. (Attachs marked as *obscure,* as described in Section 2, do not appear in the directory, however). File system operations in the attached directory are sent as regular NFS RPCs to `cfsd` via the standard NFS client interface.

For each encrypted file accessed through an attach point, `cfsd` generates a unique *file handle* that is used by the client NFS interface to refer to the file. For each attach point, the CFS daemon maintains a table of handles and their corresponding underlying encrypted names. When a read or write operation occurs, the handle is used as an index into this table to find the underlying file name. `cfsd` uses regular Unix system calls to read and write the file contents, which are encrypted before writing and decrypted after reading, as appropriate. To avoid repeated *open* and *close* calls, `cfsd` also maintains a small cache of file descriptors for files on which there have been recent operations. Directory and symbolic link operations, such as *readdir, readlink,* and *lookup* are similarly translated into appropriate system calls and encrypted and decrypted as needed.

To prevent intruders from issuing RPC calls to CFS directly (and thereby thwarting the protection mechanism), `cfsd` only accepts RPCs that originate from a privileged port on the local machine. Responses to the RPCs are also returned only to the localhost port, and file handles include a cryptographic component selected at attach time to prevent an attacker on a different machine from spoofing one side of a transaction with the server.

It is instructive to compare the flow of data under CFS with that taken under the standard, unencrypted file system interface. Figure 1 shows the architecture of the interfaces between an application program and the ordinary Sun "vnode-based" Unix file system[2]. Each arrow between boxes represents data crossing a kernel, hardware, or network boundary; the diagram shows that data written from an application are first copied to the kernel and then to the (local or remote) file system. Figure 2 shows the architecture of the user-level CFS prototype. Data are copied several extra times; from the application, to the kernel, to the CFS daemon, back to the kernel, and finally to the underlying file system. Since CFS uses user-level system calls to communicate with the underlying file system, each file is cached twice, once by CFS in clear form and once by the underlying system in encrypted form. This effectively reduces the available file buffer cache space by a factor of two.

4.2. Performance

`cfsd` is considerably simpler than a full file system. In particular, it knows nothing about the actual storage of files on disks, relying on the underlying file systems to take care of this. This simplicity can come at the expense of performance. Because it runs at user level, using system calls to store data, and because it communicates with its client through an RPC interface, CFS must perform several extraneous data copies for each client request. Each copy carries with it considerable potential copying and context switch overhead. The DES encryption code itself, which is implemented in software[3], dominates the cost of each file system request (although it is the fastest software DES implementation of which we are aware). CFS access could, based on worst case analysis of its components, take several times as long as the underlying storage.

We measured CFS under a variety of workloads. For comparison, we also ran each workload on the underlying cleartext file system and again on the underlying system through a user-level encryption filter tool that implements a multimode DES-based cipher similar to that in CFS. All measurements were taken on an unloaded Sun Sparc IPX workstation running SunOS 4.1.2 with 32 MB of memory and a locally connected SCSI Seagate model Elite 16000 disk drive. Each benchmark was also run a second time on similar hardware but with the underlying files on an NFS file system connected over a lightly-loaded network. In the tables below, **CFS-LOCAL** and **CFS-NFS** indicate the

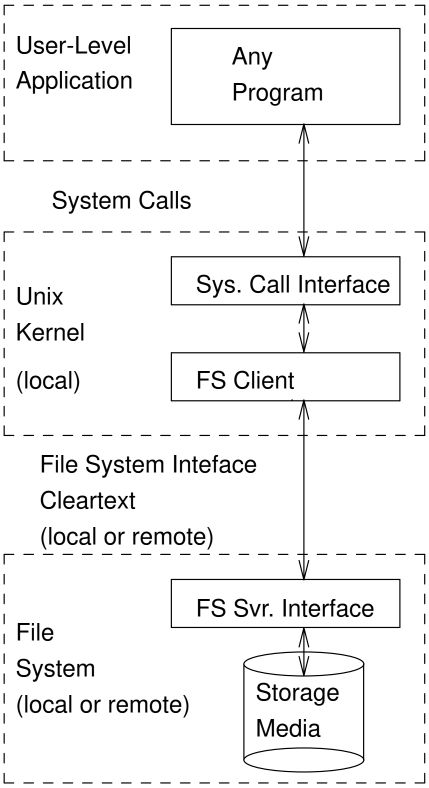

Figure 1 – Data Flow in Standard Vnode File System

CFS measurements for the local and NFS file systems, respectively, **CLEAR-LOCAL** and **CLEAR-NFS** indicate measurements of the underlying file systems, and **USERTOOL-LOCAL** and **USERTOOL-NFS** indicate measurements of the user-level encryption tool under the two file systems. All numbers represent the mean of three runs; variances between runs were low enough to be virtually insignificant. All times are given in seconds of elapsed real time used by each of the various benchmarks.

The first benchmark simply copied in and read back a single large (1.6MB) file. This measured the cost of both writing and reading, including the effects of whatever caching was performed by the file systems. Because CFS is mounted via the NFS interface, it does not perform any write caching, and this limitation was most dramatically reflected in the performance results against the underlying local file system: CFS was roughly a factor of 22 slower. The manual encryption tool, however, fared even more poorly, since its output is uncached on both reading and writing: it was slower than the underlying file system by a factor of more than 200. With NFS as the underlying storage, CFS performance was much more reasonable, less than a factor of four slower than the underlying system. The manual tool slowed performance by a factor of more than 11. These measurements are summarized in Figure 3.

Changing the benchmark to magnify read performance under the cache narrows the performance gap between CFS and the underlying storage. Figure 4 gives the results of copying the same file once but then reading it fifty times. Note the especially poor performance of user-level encryption under this workload, since the encrypted results are never cached.

The cost of creation of small encrypted files is bound primarily to the actual storage system and the system call and context-switch overhead rather than the actual encryption. We

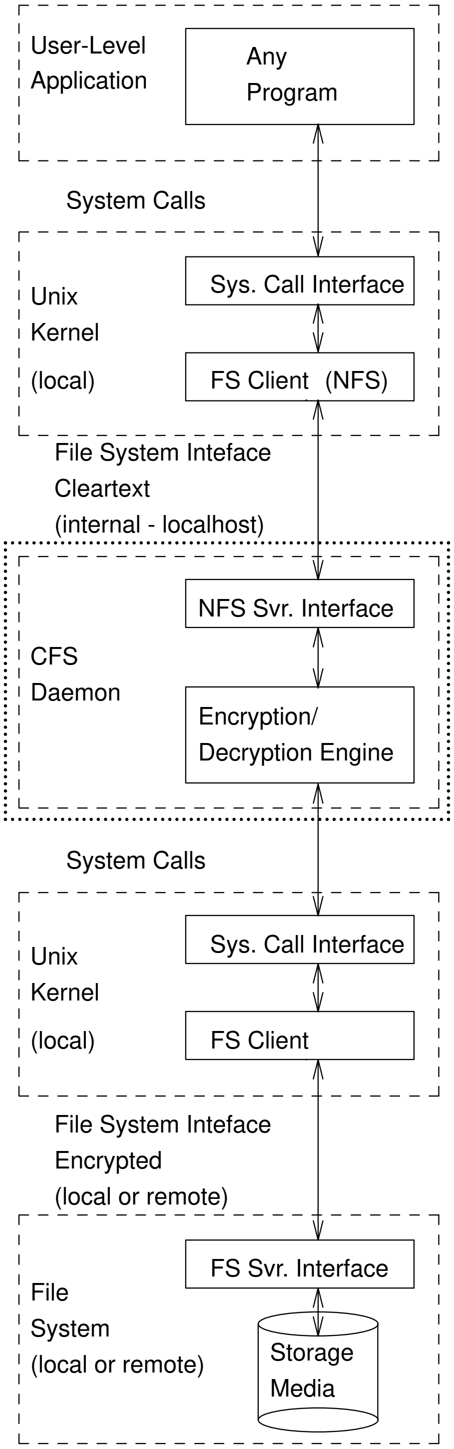

Figure 2 – Data Flow in CFS Prototype

measured the cost of creating an eight byte file one thousand times. On a local system, both CFS and the user-level system added about one third to total latency. Under NFS, CFS increased latency by roughly a factor of two, while the user encryption tool added only about 13%. Figure 5 gives the results of these measurements.

These results suggest that encryption is expensive, although the caching performed by CFS makes it less expensive

File System	Elapsed Time (seconds)
CLEAR-LOCAL	1
USERTOOL-LOCAL	217
CFS-LOCAL	22
CLEAR-NFS	20
USERTOOL-NFS	234
CFS-NFS	74

Figure 3 – Large File Copy + One Read

File System	Elapsed Time (seconds)
CLEAR-LOCAL	4
USERTOOL-LOCAL	5348
CFS-LOCAL	27
CLEAR-NFS	22
USERTOOL-NFS	5371
CFS-NFS	77

Figure 4 – Large File Copy + 50 Reads

File System	Elapsed Time (seconds)
CLEAR-LOCAL	141
USERTOOL-LOCAL	201
CFS-LOCAL	203
CLEAR-NFS	247
USERTOOL-NFS	276
CFS-NFS	496

Figure 5 – Small File Creation

than user-level encryption tools. CFS performance is much better under practical workloads, however. Informal benchmarks (such as compiling itself), with underlying files on both local and remotely mounted file systems, suggest a fairly consistent factor of approximately 1.3 using CFS compared with the underlying file system. In day-to-day operation, where there is a mix of CPU- and I/O- bound processing, the performance impact of CFS is minimal. For example, Figure 6 gives the results of compilation (`make`) of the CFS system itself (a mostly I/O-bound operation) under the various systems. Note that under both local and remote storage CFS adds about one third to the total latency; user-level encryption adds about two thirds. Furthermore, CFS is operationally transparent, while user-level encryption requires manual operation of the encryption software and is therefore likely to introduce considerable user interface delay in practice.

5. Conclusions

CFS provides a simple mechanism to protect data written to disks and sent to networked file servers. Although experience with CFS and with user interaction is still limited to the research environment, performance on modern workstations appears to be within a range that allows its routine use, despite the obvious shortcomings of a user-level NFS-server-based implementation.

The client file system interface appears to be the right place to protect file data. Consider the alternatives. Encrypting at the application layer (the traditional approach) is inconvenient.

File System	Elapsed Time (seconds)
CLEAR-LOCAL	63
USERTOOL-LOCAL	110
CFS-LOCAL	86
CLEAR-NFS	75
USERTOOL-NFS	122
CFS-NFS	106

Figure 6 – Compilation

Application based encryption leaves windows of vulnerability while files are in the clear or requires the exclusive use of special-purpose "crypto-aware" applications on all encrypted files. At the disk level, the granularity of encryption may not match the users' security requirements, especially if different files are to be encrypted under different keys. Encrypting the network in distributed file systems, while useful in general against network-based attack, does not protect the actual media and therefore still requires trust in the server not to disclose file data.

6. Acknowledgments

The author would like to express his thanks to Don Mitchell and Jack Lacy for their help in using their excellent CryptoLib software. Steve Bellovin made a number of helpful suggestions on lines of attack against CFS. Howard Katseff and Tom Reingold entrusted CFS with real data and cheerfully suffered through each new (and incompatible) release. Tom London is owed a particular debt of gratitude for creating the supportive environment in which this work was done. We also thank the anonymous referees for their helpful comments.

7. References

[1] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanaryanan, M. & Sidebotham, R.N. "Scale and Performance in Distributed File Systems." *ACM Trans. Computing Systems,* Vol. 6, No. 1, (February), 1988.

[2] Kleiman, S.R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX." *Proc. USENIX,* Summer, 1986.

[3] Lacy, J., Mitchell, D., and Schell, W., "CryptoLib: A C Library of Routines for Cryptosystems." *Proc. Fourth USENIX Security Workshop,* October, 1993.

[4] Lai, X. and Massey, J. "A Proposal for a New Block Encryption Standard." *Proc. EUROCRYPT 90,* 389-404, 1990.

[5] National Bureau of Standards, "Data Encryption Standard." FIPS Publication #46, NTIS, Apr. 1977.

[6] National Bureau of Standards, "Data Encryption Standard Modes of Operation." FIPS Publication #81, NTIS, Dec. 1980.

[7] Reiher, P. et. al., "Security Issues in the Truffles File System." *Proc. PSRG Workshop on Network and Distributed System Security,* 1993.

[8] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., & Lyon, B. "Design and Implementation of the Sun Network File System." *Proc. USENIX,* Summer, 1985.