A Cryptographic File System for Unix

Matt Blaze

AT&T Bell Laboratories
101 Crawford 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) offers an alternative to ad hoc user-level encryption for protecting file data. CFS supports secure storage at the system level through a standard Unix file system interface to encrypted files. Users can associate a cryptographic key with any 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 distributed file systems 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, making 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 modern computers with our most important information.

Cryptographic techniques offer a promising approach for protecting files against unauthorized access. When properly implemented and appropriately applied, it is widely believed that modern encryption algorithms (such as the Data Encryption Standard (DES)[1] and the more recent IDEA cipher[2]) are sufficiently strong to render enciphered data unavailable to virtually any adversary who cannot supply the correct key. And yet 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 a number of non-cryptographic attacks.

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" as part of 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 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.) New encryption algorithms entail modification of every program, and 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 on-line 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 reliably restore files 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 and network encryption has a number of shortcomings from a 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 is never itself making 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 machines whenever possible[3]. Even if the cryptographic operations are themselves implemented in hardware, additional server software complexity is still required to support it.

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 does responsibility for file encryption properly belong. 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 key that was 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 typed in and authenticated, the user should not be asked to type it again on subsequent operations that can be reliably connected with the previously supplied key. Of course, the user should also have some way to manually destroy a supplied key when it is not in 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 multiple times in 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 offers 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, it should be possible for administrators 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, archival and usage accounting work normally on encrypted files and directories without knowledge of the key. CFS ensures that cleartext file contents and name data are never stored on a disk or transmitted over a network.

CFS works by providing a "virtual" file system on the client's machine, typically mounted on `/crypt`, through which users access their encrypted files. The attach command creates an entry under `/crypt` that is associated with a cryptographic key and a directory on some other file system. Files are stored in encrypted form and with encrypted path names in this directory, although they appear to the user who issued the attach command as if they are ordinary files and directories under `/crypt`. The underlying encrypted directories can reside on any accessible file system, including remote file servers such as Sun NFS[4]. Users control CFS through a small suite of tools that create, attach, detach, and otherwise administer encrypted directories.

User keys in CFS consist of arbitrary-length "passphrases". The passphrase is used to generate the 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. The phrase may include any printable ASCII character, and ideally will consist of an easily remembered nonsense phrase with unusual punctuation, capitalization or spelling (e.g., `"if you have nothing 2 hide you Have nothing too fear!"`). The actual method of encryption is discussed below in section 3.

The `cmkdir` command is used to create encrypted directories and assign their key phrases. Its operation is similar to that of the Unix `mkdir` command with the addition that it asks for a key. For example, 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 must be 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 DES 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 padded out 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 Model

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 offers the security of end-to-end encryption between the client and the server without any actual encryption required at the server side.

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).

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.

3. File Encryption

CFS uses DES to encrypt file data. DES has a number of standard modes of operation, 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 *stream mode* ciphers which 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 follows 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 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 precompute 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 blocks).

This encryption scheme guarantees that identical blocks will encrypt to different ciphertext depending upon their position in the file. It does admit some kinds of structural analysis, however. It is possible to determine which blocks are identical (and in the same place) in two files encrypted under the same key. Chosen plaintext attacks can also reveal something about a file's structure: an attacker can verify that a particular block corresponds to some guessed plaintext by simple bitwise comparison of the ciphertext.

The security of this scheme is not well-analyzed in the literature (it may be new - there appear to be no previous references to this technique), and it is beyond the scope of this paper to attempt to do so. However, at a minimum, it is clear that the level of protection against attack is at least that of a single DES pass in ECB mode but may be as strong as two passes with DES stream mode ciphers. It is possible that this scheme is weakened, in that the attacker can search for the two DES subkeys independently, if there are several known plaintext files encrypted with the same passphrase.

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.

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.

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 are defined to allow attaching and detaching 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. (Attaches 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.

The structure of CFS is described graphically in figure 1.

For each encrypted file accessed through an attach point, `cfsd` generates a unique *file handle* which is used by the client NFS interface to refer to the file. For each attach point,

.

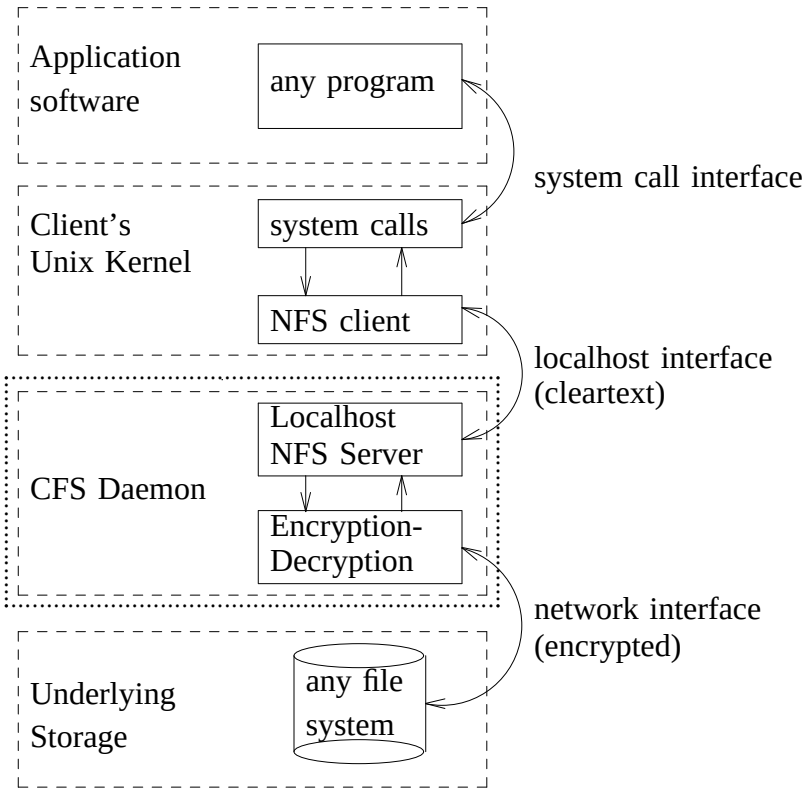

Figure 1 - CFS Prototype

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.

`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 comes at the expense of performance. Because it is at user level, using system calls to store data, and because it communicates with its client through an RPC interface, it must perform several extraneous data copies for each client request. In the diagram in figure 1, each arc represents the crossing of a kernel boundary, with considerable associated context switch overhead. The DES encryption code itself, which is implemented in software, dominates the cost of each file system request. CFS access could, based on worst

case analysis of its components, take up to six times as long as the underlying storage.

CFS performance is much better in practice, 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 1.5 using CFS compared with the underlying file system. This surprising performance can be attributed to two factors. First, the DES software implementation[5] used in CFS is highly optimized (running at about ten times the speed of the standard library implementaton), enabling encryption at close to disk I/O speeds on modern workstations. Second, and perhaps most importantly, the hit rate of the client cache under typical Unix workloads is generally quite high (above 70%), so most I/O for encrypted files never actually reaches CFS. Of course, it is possible to construct a "pathological" workload that will defeat the cache, and such workloads do cause CFS to perform about twice again as slowly.

A forthcoming paper analyzes CFS performance in more detail.

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.

6. Acknowledgements

The author would like to express his sincere 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 on CFS. Howard Katseff 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 highly supportive environment in which this work was done.

7. References

[1] National Bureau of Standards, "Data Encryption Standard." FPSP #46, NTIS, Apr. 1977.

[2] Lai, X. and Massey, J. "A proposal for a new block encryption standard." *Proc. EUROCRYPT 90*, 389--404, 1990.

[3] 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.

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

[5] Lacy, J., Mitchell, D., and Schell, W., "CryptoLib: A C Library of Routines for Cryptosystems." AT&T Bell Laboratories document, 1992.