The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX

Giuseppe Cattaneo Luigi Catuogno Aniello Del Sorbo Pino Persiano
Dipartimento di Informatica ed Appl.
Università di Salerno
Baronissi, SA, Italy, 84081
cattaneo@unisa.it
luicat@dia.unisa.it
anidel@dia.unisa.it
pino.persiano@unisa.it

Abstract

In this paper we describe the design and the implementation of the Transparent Cryptographic File System (TCFS). TCFS is a cryptographic distributed filesystem (à la NFS). It lets users access sensitive files stored on a remote server in a secure way. It combats eavesdropping and data tampering both at the server and over the network by using encryption and message digest. Applications access data on a TCFS filesystem using the regular systems calls thus yielding complete transaparency to the users.

TCFS implementations for Linux, NetBSD and OpenBSD are available at `http://www.tcfs.it`

1 Introduction

Today's networking makes it feasible to share resources over a network. Filesystems have been historically one of the first services to be distributed over a network (see Sun's NFS [13]). Nowaday this services are even more necessary. In facts the wide spread of mobile equipments (such as PDA or lap top computers) strongly require the availability of a common file repository accessible from any place allover the world with different network strategies. Distributing applications and services over a network offers obvious advantages but creates several security problems: unauthorized users might gain access to restricted services.

Within the context of distributed filesystems this phenomenon is easily seen. In a distributed filesystem, we have two types of actors: servers which have direct access to a local filesystem and clients that wish to access files on filesystems local to the server. Servers and clients are connected through a communication network. Let us take NFS as an example. NFS is very naive in its approach to security. Roughly speaking, the server receives requests for block of data from client and sends the data block in clear over the network. It is a simple task to eavesdrop over the conversation and thus read the data [9]. Moreover, the access to the data is granted by the server on the basis of the uid (and gid) communicated by the client. Thus nothing prevents a pirate client from giving the server "the right" information and thus gaining access to the whole file system.

Users who wish to protect their files should adopt measures to prevent exposure of sensible data. This problem can be addressed at several levels: user, application and system level. In this paper we present the Transparent Cryptographic FileSystem (TCFS, in short) that addresses the problem of securing data in a distributed filesystem at the system level. The TCFS project started in 1995 but only in the last 12 months the experimental analysis have demonstrated its efficiency and its robustness.

Before describing the features of TCFS, let us present arguments in favor of a solution at the system level as opposed to solutions at the user or application level.

Several tools exist to encrypt the content of files and directories. However, we point out that this approach suffers of two main drawbacks:

1. Ease of use. Data reside in encrypted form on the filesystem. Before accessing the data, the user needs to decrypt it before and, after he has finished, he needs to re-encrypt the data. This is very cumbersome and users would tend to avoid this step. In general, a well-know security practice principle states that security has to come to little or no operative cost to the user.

2. Network. Encrypting and decrypting data in a distributed filesystem does not guarantee that the data is not exposed to an unauthorized party. Indeed, once the user has decrypted the data, it is stored unencrypted on the server. Thus data is leaked to the filesystem server. Moreover, data is transferred between the server and client in clear and thus can be read by eavesdroppers.

Several widely used applications offer an encrypting service: when data is saved to disk, the user can choose whether to encrypt it or not. This approach addresses the usability problem but data is still vulnerable when it travels on the network.

2 Related Work

Distributed file systems have been the focus of much research in the last decades starting with the early proposals [3, 13].

Along this line of research, transparency (the fact that the filesystem is not local should be hidden from applications and users) and security have been two main issues. Some systems, like Compaq's Cluster File System[4], do guarantee transparency even in presence of faults, but assume that confidentiality of data and authentication of the parties involved is achieved through some other system. The Andrew File System [6] and CODA [14] instead provide mechanism to authenticate servers over public lines and to ensure that client-server communication could not be eavesdropped. The serious key-management issues arising in such file systems is addressed by the Self-certifying FileSystem [8] (more on SFS in Section 2.3). We do point out that both AFS and CODA assume the server to be trusted and store the data in clear on the server machine. TCFS instead stores the files in encrypted form thus denying the server access to the data in clear.

In this section we review some of the work present in the literature stressing the differences with TCFS either in implementation or in architectural design.

2.1 Cryptographic File System

Matt Blaze's Cryptographic File System (CFS)[2] is probably the most widely used secure filesystem and it is the closest to TCFS in terms of architecture. CFS encrypts the data before it passes across untrusted components, and decrypts it upon entering trusted components. CFS users create directories associated with keys and each file created in a protected directory is automatically encrypted.

CFS simulates a remote NFS server which exports on demand encrypted directories. All operations performed in clear by the user on a protected resource are mapped by CFS to the source directory (created by `cmkdir`) encrypted. During (and after) the user session, an intruder could not obtain clear data from the source directory.

CFS, that was the primary motivation of the work presented in this paper, presents the following characteristics.

- CFS is not transparent to the user. Encrypted directories have to be explictly attached to a specific directory by the user before they can be accessed.

- Cryptography granularity is at the level of the directory. This implies that the user must remember a password for each encrypted directory she owns. Moreover, *all* files in an encrypted directory are encrypted as opposed to TCFS where the user can choose which files to keep in encrypted form and which to keep in clear.

- CFS has been implemented as a user application. On the positive side, this approach makes it very easy to port CFS to different operating systems. On the negative side, this increases its vulnerability to attacks to the client machine and reduces its performance.

- CFS does not allow group sharing of protected resources nor it offers data authentication.

2.2 CryptFS

CryptFS[18] is a cryptographic file system implemented at the virtual inode level using the abstraction of Stackable File Systems [5] and can be used on top of local or remote file systems. Like TCFS it uses the cipher block chaining encryption mode within a block (usually 4k o 8k long) and only provides `Blowfish` as encryption algorithm.

CryptFS is part of the FiST (File System Translator) [19] project developed by the same authors. FiST is a system that uses a high-level language to describe a file system and to generate the working implementation for the target operating system, thus improving portability.

We found no source code for CryptFS, so we could not compare it with TCFS. A performance comparison between CryptFS and (an older version of) TCFS is found in [18]. CryptFS does not ensure data integrity and does not allow unencrypted files on an encrypted file system. This has a non trivial impact on the performance as, for example, CryptFS needs not to check if the file is clean or encrypted, nor it needs to check the integrity of blocks upon reading. We also stress that there is no support for threshold group sharing of encrypted files.

2.3 Self-certifying File System

The Self-certifying File System (SFS)[8] addresses the issue of key management in cryptographic filesystems and proposes separating key management from file system security. Servers have a public key and clients use the server public key to authenticate the server and establish a secure communication channel. To allow clients to authenticate servers on the spot without even having heard of them before, SFS introduces the concept of a "*self-certifying pathname.*" A self-certifying pathname contains the hash of the public-key of the server, so that the client can verify that he is actually talking to the legitimate server. Once the client has verified the server a secure channel is established and the actual file access takes place.

Remote SFS file systems are accessed through the `/sfs` mount point. An SFS pathname obeys the following syntax:

`/sfs/location:hostid/real/pathname`, where "`location`" is the name (IP address or DNS Name) of the server exporting the file system and "`hostid`" is the hash of a string containing the server's public key and some other information. SFS does not care on how the pathname has been obtained by the user; a user can eventually obtain `hostid`'s using an existing PKI (Public Key Infrastructure). On the other hand, once a self-certifying pathname for the files he is interested in has been obtained, users do not need to remember any key.

3 The TCFS Architecture

TCFS relies on a very simple architecture. Data is stored in encrypted form on the server filesystem. Each time an application running on a client has to read data, the client kernel requests the appropriate block of data from the server. The server ships the block of data in encrypted form to the client. The client decrypts the block of data before passing it to the application. A write operation is accomplished in a similar way. Suppose a client application wishes to write data on a filesystem. The application passes the data to the client that encrypts the data and passes it to the server. The server, upon receiving data from the client over the network, stores the data on the filesystem.

This architecture has several advantages:

- Minimal trust model. The TCFS architecture does not rely on the the server nor the network being trusted. In fact, the server only sees encrypted data and data travels over the network only in encrypted form. As we will see when we discuss the implementation details, the client can detect any unauthorized modification of data. Of course, since clients can access data only through the servers, TCFS cannot prevent servers from erasing the data or from denying access to the clients. All the encryptions and decryptions are performed by the client on which the application is running. Thus the application and the user have to trust the client kernel used to access the filesystem. This is not a serious limitation for cases in which users employ personal workstations to access files.
- Low system administration impact. TCFS does not require any additional duty to the system administrator of the server. All filesystem maintenance operations on the servers need not to know about TCFS. Actually, the system administrator himself might ignore that his local filesystem is actually a TCFS filesystem.
- Low impact on client applications. TCFS was designed to reduce the impact on the applications. Client applications access files on a TCFS filesystem through the usual system calls and thus they need not to be re-written or re-compiled to work with TCFS. Client applications need not to deal with key management.
- Low impact on the user. Besides issues regarding key management, TCFS has little or no impact on the final user. She can still access her files using the same applications and ignore completely that the files she is accessing are stored on a remote server in encrypted form. TCFS guarantees to the users and to the applications a level of transparency similar to NFS. Nonetheless, TCFS provides users needing a greater control on the encryption/decryption policy, the ability to control which files are encrypted and which are not.

3.1 Authenticating servers

TCFS assumes a very minimal trust model: the user only needs to trust the client machine used to access the TCFS filesystem. We point out that this is a very minimal assumption as it is very hard to conceive a system that preserves security even in presence of untrusted client machines.

On the contrary, a user needs not to trust the server on which the filesystem physically resides. Indeed, the server only has access to data in encrypted form which is of no use. Obviously, the server can modify the data stored and there is nothing that the user can do to prevent that. However, since TCFS includes authentication mechanisms for the data, if the server modifies the data, the user will immediately notice that data has been altered.

Similarly, there is no need for the client to authenticate the server. Suppose that a pirate host has managed to impersonate the legitimate TCFS server. We stress that, even in this case, the privacy of the user is not compromised. Indeed if the client tries to write, then the private server only gets encrypted data. On the other hand, if the client performs a read operation, the data he/she will receive from the server will not be authenticated and thus immediately rejected by the client.

4 Key management

In the design of TCFS we have decided to keep key management issues separated from the actual cryptographic filesystem. In the two implementations of TCFS for Linux and BSD-like kernels, TCFS provides a simple interface to pass key to the kernel (by ad-hoc `ioctl` calls, or by upgrading the filesystem mounting). On top of this basic key-management primitive more sophisticated key management schemes can be built. As part of the TCFS project we have implemented three key management schemes that we termed the Raw, the Basic and the Kerberized Key Management Scheme that we briefly review in the rest of the section. TCFS can perform key management at different levels: at the process level in the sense that each process has its own key to access the TCFS filesystem; at the user level in the sense that each user has its own key and all processes with the same `uid` use the same key. Moreover, TCFS provides a simple threshold mechanism for sharing files in a group of users.

4.1 Group Sharing

TCFS includes the possibility of threshold sharing files among users. Threshold sharing consists in specifying a minimum number of members (the threshold) that need to be "active" for the files owned by the group to become available. TCFS enforces the threshold sharing by generating an encryption key for each group and giving each member of the group a share using a Threshold Secret Sharing Scheme [15]. The group encryption key can be reconstructed by any set of at least threshold keys.

A member of the group that intends to become active does so by pushing her/his share of the group key into the kernel. The TCFS module checks if the number of shares available is above the threshold and, if it is so, it attempts to reconstruct the group encryption key. By the properties of the Threshold Secret Sharing Scheme, it is guaranteed that, if enough shares are available, the group encryption key is correctly reconstructed. Once the group encryption key has been reconstructed, the files owned by the group become accessible. Each time a member decides to become inactive, her share of the group encryption key is removed. The TCFS module checks if the number of shares available has gone under the threshold. In this case, the group encryption key is removed from the TCFS module and files owned by the group become unaccessible.

The current TCFS implementation of the group sharing facility requires each member to trust the kernel of the machine that reconstructs the key to actually remove the key once the number of active users goes below the threshold. Future implementations will remove this requirement by performing the reconstruction of the key in a distributed manner.

4.2 Raw Key Management Scheme

TCFS provides a simple interface for users applications to pass keys to the kernel which we call the Raw Key Management Scheme (RKM, in short). By using the RKM API, an application can provide the key to the TCFS kernel. Subsequently, the TCFS kernel will use the key provided to perform encryptions and decryptions. No check is performed by the TCFS kernel on the key and the application has to make sure that the right key is passed to the kernel. The RKM scheme is not intended for the end user but only as a basis on top of which to build more sophisticated KM schemes.

4.3 The Basic Key Management Scheme

This scheme allows users to generate their keys and to store them in a database in encrypted form using the login password as key. Thus, TCFS users must not remember their master key, but only their login password. To benefit of the BKMS a user must be registered with the key database (typically the file `/etc/tcfspwdb`) by the system administrator. The usage of the BKM scheme follows the phases below:

1. The system administrator registers a user to the key database (Fig. 1) by issuing the command `tcfsadduser`.

2. The user creates his master key by running the `tcfsgenkey` command. `tcfsgenkey` generates a random key, encrypts it with the user's password, and stores it in the entry of the key database associated with the user.

3. When the user needs to access his encrypted files, he must extract his master key from the database (providing his password), and give it to the TCFS layer. This operation can be performed with the `tcfsputkey` command(Fig. 2).

4. The user terminates his session by running the `tcfsrmkey` command which erases the key from the kernel.

Setting up a TCFS group requires the following steps:

1. The system administrator creates a normal UNIX group, then creates a TCFS group by running the `tcfsaddgroup` command. This utility asks for the number of group member, the threshold, the password, and the username of each member of the new TCFS group. For each member, a share is created, encrypted with the password of the respective user and then it is stored in the TCFS group keys database (`tcfsgpwdb`).

2. To become active, a member of a TCFS group pushes her share into the kernel. This can be accomplished by executing the command `tcfsputkey` with the `-g` switch. Note that, user can get access to shared files only if the number of the same group shares pushed to the kernel is greater or equal to the group's threshold.

3. The `tcfsrmkey -g` command ends the user's session.

The aim of the BKMS is to provide the user with a simple to use management scheme. It is not to be considered very secure as the user master key is protected by the user login password that can be compromised in several ways.

4.4 The Kerberized Key Management Scheme

Kerberos is a distributed authentication service developed in the late 80s at M.I.T. [17].

`root#` **tcfsadduser**	
`Username to add to TCFS database:` **jack**	
`Ok`	*now jack has an empty entry in the key db*
	before to have his first TCFS session, jack must run:
`jack$` **tcfsgenkey**	
`Insert your password, please:`	*give his login password*
`Press 10 random keys, please:` **********	*seed*
`Key succesfully generated.`	*now jack's enty in the key db contains his master key, ecrypted with his login password*
	whenever superuser must remove jack from TCFS key database, he must run:
`root#` **tcfsrmuser -u jack**	

Figure 1: Creating a TCFS user with the BKM Scheme

Kerberos requests as a trusted third-party authority that provides authentication to all the actors in a distributed environment. Kerberos makes possible for a client and a server to authenticate each other and to establish a private communication channel. The Kerberized Key Management(KKM) Scheme provides a strong alternative to the BKM Scheme. We introduce a new component: the *TCFS key server* (TCFSKS) that maintains a database of master keys. Clients (*i.e.*, kerberized TCFS utilities such as `tcfsputkey`, or TCFS-*aware* applications) authenticate themselves on Kerberos, and obtaining a session key and a ticket, send to TCFSKS the requests (for example: get the user key or store a new key) over the network. Administrative operations, such as adding/removing users and group, can be performed in the same way. Since no changes have been made to the interface of front-end utilities, an user does not feel any difference between Kerberized and Basic Key Management procedures. The only substantial difference is that now all Key Management operations are performed over the network and thus several TCFS clients can share the same TCFS key database (in the BKMS the key db is local to the client).

Communication among client and TCFSKS follows these steps:

1. At the end of the Kerberos authentication, the client obtains the session key.

2. The client sends its request and its ticket to the TCFSKS.

3. The server decrypts the message and sends back the response and the ticket.

4. The client get the response and discards the ticket.

5 Cryptographic Engine

TCFS does not employ a fixed encryption scheme but for each file a different encryption engine can be specified. Encryption engines need to conform to a specific interface and, in the Linux implementation, can be kernel loadable module. Modules for all the major encryption scheme are provided with the implementations. Having modular encryption, allows user to plug into TCFS its own encryption module for increased user security. Modu-

`jack$` **tcfsputkey -m /mnt/tcfs**	*Jack starts his session*
`Password:`	*giving his login password*
	now, Jack can encrypt/decrypt and access
	transparently to encrypted files.
`jack$` **cd /mnt/tcfs**	
`jack$` **echo "Hello World!" > first**	*the file "first" is still in clear*
`jack$` **tcfsflag +X first**	*toggles first's cryptographic flag*
	now it is stored encrypted
`jack$` **cat first**	*all standard application can access*
`Hello World!`	*encrypted files*
	while Jack's key is available to the kernel
	can be read,
`jack$` **cp first second**	*copied and so on..*
	the file "second" is stored in clear
`jack$` **tcfsrmkey -p /mnt/tcfs**	*Jack removes his master key from the kernel*
`jack$` **cat first**	
`permission denied`	*since the master key has been removed,*
	access to encrypted files is not
	allowed.
`jack$` **cat second**	
`Hello World!`	*second is still in clear, TCFS session*
	has no effect on clear files

Figure 2: A simple TCFS session

lar encryption allows users and system administrator to pick their favorite block encryption scheme. Thus, for this section we denote by $E(\cdot,\cdot)$ and $D(\cdot,\cdot)$ the encryption and decryption algorithm associated with the encryption scheme actually employed (see Figure 3). The size of the block encrypted by the encryption need not to be equal to the block size of the file. Each file has a header that contains some information about the file itself (e.g., TCFS version number, cipher id).

Each user A is associated with a *master key* K_A as described in Section 4. For each file f a *file key* K_f is randomly chosen. The file key is encrypted using the master key of the user and stored in the `file-key` field of the header. Each block of a TCFS file consists of two parts: the data and the authentication tag. Each block of an encrypted TCFS file is encrypted with the encryption algorithm E in CBC mode using a different *block key*. Block of unencrypted TCFS file are stored in clear. The block key is computed by applying the hash function to the concatenation of the file key and the block number. The authentication tag of an authenticated TCFS file is computed by hashing the concatenation of the block data and the block key. On the other hand, unauthenticated TCFS files have an authentication tag consisting of NULL bytes.

This way of encrypting and hashing the blocks exhibits the following security characteristics:

1. If a robust encryption scheme is used then

encrypted files cannot be read without knowledge of the file-key or of the user master key.

2. Since each file is encrypted using a different file-key, it is not possible to check whether two encrypted files correspond to the same cleartext.

3. Moreover, since each block is encrypted using a different block key, it is not possible to check whether two blocks of the same file correspond to the same cleartext.

4. The authentication tag is to prevent that data on the server is modified. Obviously, since TCFS servers have physical access to the filesystem there is nothing that could prevent the server from modifying files. Our goal is thus to guarantee the user that no modification will go unnoticed. Our authentication mechanism guarantees:

   (a) Modification of a block without recomputing its authentication tag is easily detected by the TCFS client. However, recomputing the authentication tag of a block requires knowledge of the block key which in turn depends on the file key which is encrypted using the user master key.

   (b) Since each block authentication tag also depends by the block offset without knowledge of the block key it is not possible to insert foreign blocks of data or to swap two blocks of the same files.

       Moreover, since the authentication tag depends also on the file key, it is not possible to import block from other files.

6 Implementations

TCFS is designed to work in kernel space as an intermediate layer between the Virtual File System (VFS) and the storage file systems (such as EXT2FS, UFS, NFS and so on). In this way, user applications can perform all the usual file operations by means of system calls interface, without being rewritten/recompiled. Users can perform protection/encryption operations with apposite utilities which interact to the TCFS layer by the `mount` and `ioctl` system calls.

The TCFS layer only touches application data, and not file system logical structures (such as inodes, directory organization, etc.). Hence, although protected files result incomprehensible to the server's system administrator, all the disk maintenance tasks (check, backup, recovery), can be performed as usual.

TCFS has been implemented on Linux, and {Net,Open}BSD. The two versions are quite different, due to the different characteristics of the respective operating system.

All key management features (excepting pushing/removing keys) have been implemented at user level. To make easy the development of tcfs-*aware* application, or further key management schemes implementation, TCFS is provided of a development library that includes several functions which allow to manage every aspects of interaction among user and TCFS hiding any OS-specific implementation details.

6.1 The Linux version

The Linux version of TCFS consists in an extended NFS client with cryptographic features; a remote NFS server acts as the *storage file system*.

The client host mounts the remote filesystem as a TCFS filesystem (by means of a patched version of `mount`). Client and server communicate by means of NFS protocol. The user provides the encryption key to the TCFS

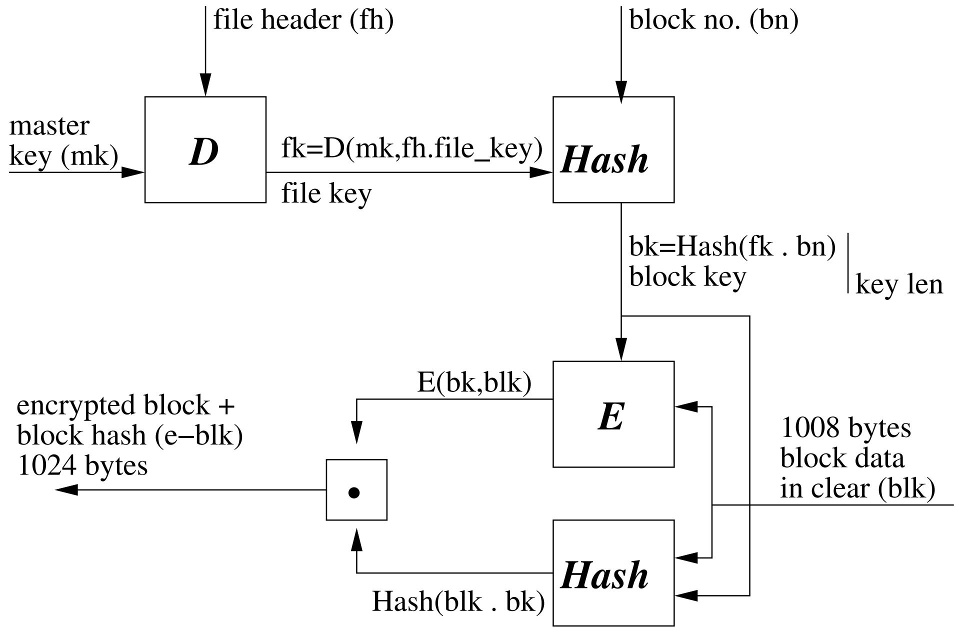

Figure 3: Encryption of blocks in TCFS

client so to allow encryption and decryption of files. The TCFS server instead is a simple NFS server: it needs not to know that the exported filesystem is indeed a TCFS filesystem. This has the advantage of making possible to use TCFS on any host that runs an NFS server.

User-level applications passes keys and directives to TCFS by calling the system call `ioctl` on the filesystem's mount point; for this pourpose, TCFS introduces some new `ioctl` commands.

Since version 2.0, the Linux kernel features Loadable Kernel Modules (LKM) (see [1] and [11]). LKMs consist in some parts of the kernel (usually device drivers or filesystem switchs) which can be loaded at runtime whenever they are needed. Cryptographic services, in TCFS, are implemented as kernel modules. Thus, ciphers can be selected at runtime by loading the proper dynamic kernel module. Users can choose different ciphers to encrypt/decrypt different files or directories. Each cipher module is completely independent from the others and it is possible to load, unload different engines or even to use simultaneously several cryptographic engines.

The Linux TCFS implementation allows to have client and server running on the same host. However, the communication between the two takes place using the NFS protocol. This has the drawback of slowing down the communication.

6.2 The BSD version

An operating system based on 4.4BSD[10] provides to the kernel a generic interface to several kinds of file systems: the virtual-node (vnode) layer. The vnode layer features an object-oriented interface which abstracts the invocation to filesystem-specific operations, implemented at an underlying level. This makes possible to mount and access different kind of file system in the same way. 4.4BSD file system switch has been provided of a mechanism for stacking filesystems on top of one other (proposed by Rosenthal[12], and refined by Heidemann and Popek[5]). The bottom of a filesystem stack is usually a storage filesystem (which directly interacts with the device driver). The layers above, can implement on their own any functions and/or redirect them downstairs (with or without argument transformations).

TCFS for BSD has been implemented as a file system layer. This approach presents an important advantage: since the cryptographic layer can be mounted upon any filesystem by mean of vnode interface, encryption of local filesystems (improving TCFS performances on these ones) does not need the NFS to introduce the cryptographic file-operators. Furthermore, TCFS for BSD has been developed by writing only those operators which required the introduction of cryptographic services whereas other calls have been redirected to undelying filesystems.

Users send their directives to TCFS by updating the mount-point parameters. The TCFS layer adds to the usual arguments of the `mount` system call two new sets of data that cointain directives and their arguments:

1. The data concerning the user's directive: the command, the key, the key's owner, etc.

2. A set of data which represents the status of the filesystem: number of active keys, error codes, information about the cipher.

TCFS flags and optional attributes management is performed by means of some `ioctl` calls.

6.3 Process keys

The BSD implementation of TCFS makes possible to provide different keys to different processes belonging to the same user. Thus users can work with several keys simultaneously and, moreover, he can setup batch jobs which works on encrypted resources. User applications do not need to be rewritten/recompiled and, the process keys management is completely transparent. To make easy to run application with different keys, we developed the `tcfsrun` utility. This utility asks the user for the process key, passes it to the kernel and withdraws it when the user application ends. All kinds of keys (user, group and process) are managed independently, so, user can use his masterkey and group-shares normally even while any applications work with their own key.

7 Performance

In this section we present the overall TCFS performances analysis as a proof of concept.

We have employed a modified version of the Andrew benchmark [7]. The benchmark takes as input a subtree containing the source code of a UNIX application (in our case we used the sources of the GNU `make` application). The benchmark consists of five phases:

1. *Directories creation*: the source directory hierarchy is reproduced several times into a target directory on the tested file system.

2. *File copy*: all files of the source directory are recursively copied to a directory of the target subtree.

3. *Recursive directories stats*: attributes of each file in the target subtree are recursively scanned.

4. *Recursive files scan* : recursive reading of files in the target subtree.

5. *Compilation*: files on target subtree are compiled and linked.

We have performed four suites of tests. The first suite measured the performance of NFS.

In the second suite of tests, we measured the performance of TCFS on files that are not encrypted. Thus, input/ouput is still performed by TCFS but no encryption engine is invoked. The last two suites deal with TCFS in which encryptions is performed by the NULL module (encryption using the indentity function) and the 3DES module (encryption using Triple-Des[16]). TCFS with the NULL module differs from the second test suite as here an encryption engine, albeit a trivial one, is invoked.

All tests have been performed on a Pentium II at 233MHz with 64Mb of memory. We have performed two series of experiments. In the first series the results obtained were more influenced by the performance of write operations as we had the the source tree on a local filesystem and the destination files on the remote filesystem (TCFS with 3DES, TCFS with Null, TCFS without encryption and NFS). In the second series of experiments we did the opposite: the source tree was on the remote filesystem and the destination filesystem was a local filesystem. Thus, the measurement were mainly affected by the performance of a read operations.

The figures reported are the average of 10 runs with the client and the server running on the same machine so that network latency is not an issue in the measurement.

As it is obvious from the experimental data, reading is much faster than writing. This is due to the fact that, unless a whole new block is written, a write operation involves reading a block, decrypting it, modifying it and re-encrypting it. Moreover, TCFS does not perform very well at random accessing encrypted files. Since TCFS encrypts each block using CBC, reading one byte might involve decrypting a whole block considerably slowing down the operation. This however does not have to be considered an inherent limitation of TCFS as it only depends on the specific cryptographic engine employed. Much faster random access can be obtained by encrypting in ECB (Electronic CodeBook) mode even though in this case the confidentiality of the data is considerably weakened.

The overhead introduced by TCFS can be seen by comparing the first column with the second and the third column of Figure 4. As it can be seen, TCFS NONE exhibits performances very similar to NFS. More surprisingly, TCFS NULL is much slower than NFS. This is due to the folowing phenomenon. TCFS forces the remote attribute checking before each read/write operation whereas NFS does not. We expect that removing this check would have no impact on the security and keep the performance of TCFS NULL test much closer to NFS.

TCFS 3DES is the slowest of all and this is mainly due to the time to perform encryption/decryption. Indeed the difference between TCFS 3DES and TCFS NULL is exactly the overhead introduced by the cryptographic engine. Reducing this gap calls for a better cache management strategy, an issue that at the moment has not been considered yet. Also, we stress that when this test were performed with client and “remote” filesystem residing on the same machine. On a loaded Ethernet, the encryption/decryption overhead is likely to be absorbed by the network latency.

We believe that TCFS still has room for improvement (we would like to see TCFS NULL closer to TCFS NONE) but at the moment its performances are acceptable.

8 Acknowledgments

The authors thank Angelo Celentano, Andrea Cozzolino, Ermelindo Mauriello and Raffaele Pisapia that were part of the first TCFS team and took part in the initial stage of the TCFS project.

Exporting DATA to remote filesystem				
Phase	**NFS**	**TCFS NONE**	**TCFS NULL**	**TCFS 3DES**
Creating directories	0.109	0.178	0.648	0.698
Copying files	1.385	2.777	9.047	15.924
Recursive directory stats	2.215	4.798	5.558	6.537
Scanning each file	3.074	7.489	10.047	16.129
Compilation	36.802	57.791	1m3.874	1m27.929
Importing DATA from remote filesystem				
Phase	**NFS**	**TCFS NONE**	**TCFS NULL**	**TCFS 3DES**
Creating directories	0.052	0.065	0.081	0.093
Copying files	0.282	1.545	2.548	5.462
Recursive directory stats	1.355	1.449	2.273	2.388
Scanning each file	2.261	2.464	4.038	4.267
Compilation	34.634	36.653	35.448	48.364

Figure 4: Performance of the TCFS Linux implementation as measured with the Modified Andrew Benchmark

References

[1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, D. Verworner, *"Linux Kernel internals"* Addison-Wesley, 1996

[2] M. Blaze, *"A Cryptographic File System for UNIX"*, First ACM Conference on Communication and Computing Security, pp. 158-165, Fairfax VA 1993.

[3] C. T. Cole, P. B. Flinn, A. B. Atlas, *An Implementation of an Extended File System for UNIX*, USENIX Conference Proceedings, Summer 1985, pp 131-149, Portland OR.

[4] *"Cluster File System in Compaq TruCluster Server"*, Compaq White Paper, Unix Software Division, Compaq Computer Corp, August 2000.

[5] J. S. Heidemann, G. J. Popek, *"File System development with stackable layers"*, ACM Transactions on computer systems, vol 12, no. 1, pp. 58-89, February 1994.

[6] J. H. Howard, *"An overview of the Andrew File System"*, Proceedengs of the USENIX Winter Technical Conference, Feb. 1988, Dallas TX.

[7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West, *Scale and performances in a distributed file system* , ACM Transaction on Computer System, Feb. 1988.

[8] D. Maziéres, M. Kaminsky, M. F. Kaashoek, E. Witchel *"Separating key management from file system security"*, Proceedings of 17th ACM Symposium on Operating System Principles (SOSP '99), Kiawah Island, South Carolina, December 1999.

[9] S. McCanne, V. Jacobson, *"The BSD Packet Filter: a new architecture for user-level packet capture"*, Proceedings of the 1993 winter USENIX conference, pp. 259-269, San Diego CA, 1993.

[10] M. K. McKusick, K. Bostic, M. J. Karels, J. S. Quarteman, *" The Design and Implementation of the 4.4BSD Operating System"*, Addison Wesley, 1996

[11] O. Pomerantz, *"Linux Kernel Module Programming Guide"*, GPL licensed book, 1999

[12] D. Rosenthal, *"Evolving the vnode interface"*, USENIX Association Conference Proceedings, pp. 107-118, June 1990.

[13] R. Sandberg, D. Goldberg, S. Kleimann, D. Walsh, B. Lyon, *"Design and implementation of the Sun Network Filesystem"*, USENIX Association Conference Proceedings, pp. 119-130, June 1985.

[14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, H. Siegel, D. C. Steere *"Coda: A highly available file system for a distributed workstation environment"* , IEEE Trans. Computers, Vol. 39, No. 4, Apr. 1990, pp 447-459.

[15] A. Shamir, *"How to share a secret"*, Comm. ACM v.24 n. 11, November 1979.

[16] D. Stinson, *Cryptography: Theory and Practice*, CRC Press.

[17] J. G. Steiner, C. Neuman, J. I. Schiller, *"Kerberos, an authentication service for Open Network Systems"*, USENIX Association Conferences Proceedings, pp. 191-202, February 1988.

[18] E. Zadok, I. Badulescu, A. Shender, *"Cryptfs: A stackable vnode level encryption file system"*, 1998

[19] E.Zadok *"FiST: A File System Component Compiler "*, PhD Thesis, published as Technical Report CUCS-033-97 (PhD Thesis proposal), Computer Science Department, Columbia University, April 1997.