# File System Encryption with Integrated User Management

Stefan Ludwig
Corporate Technology
Siemens AG, Munich
fsfs@stefan-ludwig.de

Prof. Dr. Winfried Kalfa
Operating Systems Group
Chemnitz University of Technology
kalfa@informatik.tu-chemnitz.de

## Abstract

*Existing cryptographic file systems for Unix do not take into account that sensitive data must often be shared with other users, but still kept secret. By design, the only one who has access to the secret data is the person who encrypted it and therefore knows the encryption key or password. This paper presents a kernel driver for a new encrypted file system, called Fairly Secure File System (FSFS), which provides mechanisms for user management and access control for encrypted files. The driver has been specifically designed with multi user systems in mind. FSFS also tries to prevent unintentional transfer of sensitive data to unencrypted file systems, where it would be stored in plaintext.*

## 1 Introduction

There exist several projects for the Unix operating system that offer transparent cryptographic protection for files or complete file systems (e.g., *CFS* [1], *TCFS* [2], *ppdd* [3], *loopback device encryption extension*). All these solutions suffer from two major shortcomings:

1. Only the owner of the data has access to the encrypted files. To share such files with other users, the owner needs to give the encryption key to every user who should be able to access the encrypted data. This way of sharing the secret files is often not acceptable because everyone who knows the key has the same status as the legitimate owner and could enable additional users to access the protected files by just giving them the key.

   There is also the risk that one of the users replaces the encryption key and takes over the ownership of the file since he is now the only one who knows the new key. When using one of the existing encrypted file systems, it is usually very costly to revoke access permissions for a user because all the data needs to be re-encrypted with a new key and this new key must be distributed to all other users.

   Apart from the key, users must also have the appropriate file access permissions to read, write or execute an encrypted file. This poses another problem. There is no way for a non-privileged user to easily manage groups of users who have access to the encrypted files, especially if some users have full access, but others only limited access (e.g., read-only).

2. While decrypted, sensitive data could leak out to unprotected areas of the file system (e.g., swap space, temporary files on plain text file systems). Up to now, to maintain a high level of security, it is necessary to encrypt all file systems of the machine. This makes it very difficult to use this kind of encryption on multi-user machines where some directories are used by several users. All these users would have to share the same key making it quite useless to encrypt the data at all.

This paper describes an encrypted file system driver (Fairly Secure File System driver, FSFS) that lets non-privileged users encrypt whole file systems and allows them to manage groups of users who should have access to the secret data. Access rights can be granted and revoked on a per user basis.

The driver also provides mechanisms that help to prevent secret data from accidentally leaking out to non-encrypted file systems.

## 2 Fairly Secure File System - FSFS

FSFS consists of two main components: A *device driver* that operates in kernel space and encrypts blocks of data

before they are written to disk, and a *setup program* in user space that provides a user interface for the encryption driver. It is used for communication with the kernel driver, including the transmission of encryption keys, and to perform different administrative tasks, such as creating new encrypted file systems and managing access permissions.

The encryption driver operates at the lowest layer of the file system stack and behaves like a normal block device driver. All data that is sent to this virtual block device is encrypted and passed on to the driver for the hardware block device where the secret data is stored. When the kernel requests a block of data from the virtual block device, the driver gets the encrypted block from the hardware block device, decrypts it, and returns the plaintext block to the kernel.

Since the encryption driver works with blocks of raw data, it does not know about files, directories, and how a file system implements them. Therefore, FSFS can only encrypt whole partitions (containing some file system), not single files or directories. This requires the user to know in advance how much space to reserve for the encrypted file system. Although this a limitation[1] because either some unused space is wasted or the file system is running out of space, it makes the encryption completely transparent for the rest of the operating system and therefore very flexible to use: Data can be encrypted on any block device supported by the operating system and any file system can be used to store the secret files. Neither the kernel nor any application needs to be modified to make use of the capabilities of the encryption driver.

On a typical system, there are no spare disk partitions available for an unprivileged user to create encrypted file systems on. Therefore, FSFS supports the use of so-called container files. FSFS treats these files like disk partitions and the user can create a file system inside the container. Such a file system can be mounted and used like any other file system. The setup program provides special functions for mounting and unmounting FSFS container files that allow even unprivileged users to mount file systems created within a container file.

This way a user can create one or more container files, create encrypted file systems inside these containers, and store secret files there. Files that belong together can be grouped by putting them inside the same container, protected by a single encryption key. Using container files makes it easy to create backups of the secret data. Container files can be backed up just like any other file, resulting in an encrypted backup of the secret files. The user/process performing the backup does not need to have access to the encrypted data, but only to the container file.

[1]There exist tools for the popular ext2 file system (*resize2fs, ext2resize, GNU parted*, and others) that can resize a partition without corrupting the file system on it, thus eliminating this limitation.

## 2.1 Key Flow and User Management

In order to implement a flexible access control and user management for the encrypted files, FSFS uses several keys and passwords. These are used for user authentication, granting and revoking access rights on a per user basis, and protection against different attacks.

Figure 1 shows how the different keys, passwords and other values are combined to achieve this:

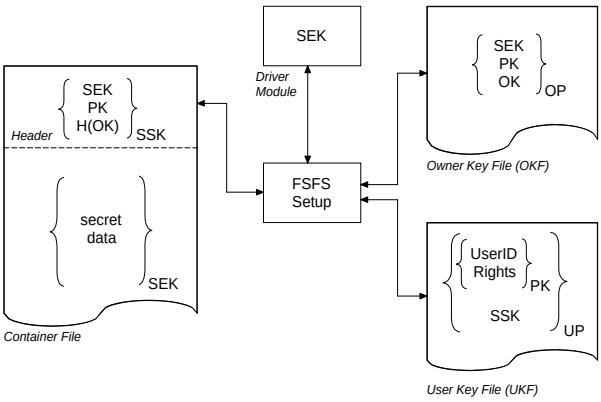


Figure 1: Key flow between FSFS components

Keys, passwords and configuration data are stored in various places of the system. The secret data in a container file is encrypted using a random *Secret Encryption Key (SEK)*. Only the owner of the encrypted data knows this key. It is never stored on backing store in plaintext. The *SEK* is generated when the owner of the files creates a new encrypted partition. At the same time an *Owner Key (OK)* and a *Permission Key (PK)* are generated, both also only known to the owner. The owner can later use the *OK* to prove to the driver that he really is the owner of the secret data. The *Permission Key* plays an important role in the management of access rights and will be described later. *SEK*, *PK*, and *OK* are stored in a so-called *Owner Key File (OKF)* and are protected by an *Owner Password (OP)*, chosen by the owner.

FSFS supports *Pluggable Authentication Modules* (PAMs) [4]. Therefore, not only (potentially weak) pass-

words can be used to protect the Owner Key File but also other methods of authentication (e.g., smartcards, biometrics).

The first bytes of the container file are used as a header where keys and configuration data, such as the name of the encryption algorithm used for protecting the secret files, are stored. The header contains the *SEK*, the *PK* and a secure hash value of the *OK*. These keys are encrypted using the so-called *Secret Shared Key (SSK)*. The container file can only be accessed by its owner and the FSFS setup program, which runs with root permissions.

The *SSK* is given to all users who should be able to access the encrypted files inside the container. The user stores the *SSK* in his *User Key File (UKF)* and protects it with a *User Password (UP)*. When a user wants to work with the encrypted files, he presents his *SSK* to FSFS. The driver then uses the *SSK* to decrypt the header of the container file, receives the *SEK* and is able to decrypt the secret data for the user.

This procedure allows the owner to grant other users access to the encrypted files without giving them the key used to encrypt the data *(SEK)*. This has the advantage that the owner can easily revoke access to the file system by encrypting the *SEK* in the container header with a new *SSK* and distributing this *SSK* to all users except those who should no longer be able to access the files. The layer of indirection added by encrypting the *SEK* with an *SSK* and distributing the *SSK* instead of the *Secret Encryption Key* is not really necessary for access revocation but increases the usability of FSFS: Encrypting the possibly large amounts of data with a new *SEK* would be time consuming. During this time, the file system cannot be accessed because some parts of the data are already encrypted with the new key while for others still the old key is needed. Re-encrypting only the header of the container file is much faster and the encrypted files can be accessed without interruption because the header is only read when a user wants to gain access to the encrypted file system.

To be able to give different users different types of access, the user ID and access rights for each user are stored in his key file *(UKF)*. To protect user ID and access rights from being tampered with, they are encrypted using the permission key *PK*. To access the secret data, a user must present an encrypted user ID/access rights pair. The driver decrypts this pair using the *PK* stored in the container header. Only if the decrypted user ID matches that of the user, access with the specified rights is granted. Since users cannot access the container file header and all access to the encrypted files is controlled by the encryption driver, it can enforce these access rights.

To do this, the driver intercepts all system calls that could initiate data transfers from or to block devices. The original system call is replaced by code that checks if an encrypted partition is involved in the call, and if the process owner has sufficient rights to perform the requested operation. If not, an error is returned; otherwise the original system call is executed.

FSFS modifies system calls such as `open()`, `link()`, `del()`, `chmod()`, and `chdir()`. This modification is achieved using a loadable kernel module that replaces the original system calls with its own code. This way the Linux kernel does not need to be changed and FSFS can be installed or removed without a reboot.

FSFS introduces the following five access rights, which always apply to the entire encrypted file system and take precedence over the Unix file access rights of the files stored in the encrypted file system:

**Read:** The user is permitted to read the encrypted files, navigate through directories, and list files.

**Write:** Files may be written, created or deleted.

**Export:** The right ”read” does not include the permission to copy the secret data that has been read from the encrypted file system, to another (possibly unencrypted) file system. If not granted the ”export”-right, a user could save data read from encrypted files only to the same file system. This right can be used to prevent secret data from being stored in non-encrypted parts of the computer.

To enforce the ”export”-right, the driver keeps a list of all processes that have read from the encrypted file system and belong to users without ”export” permission. If such a process wants to open a file on another file system for writing, the `open()` call fails.

This protection against the transfer of secret data out of the encrypted file system is not complete. It does not prevent a process that has read secret data from communicating with other processes (e.g., using IPC or shared memory), which then could write the data to a file anywhere on the system. The transmission of data over network connections is also not addressed. There is no easy way to establish effective protection in these two areas because standard Unix operating systems cannot tell who the

original owner of a piece of data was after it has been transmitted within the system. Only a complete redesign of the operating system could make the ”export”-right enforceable in all cases.

However, leakage of plain text to virtual memory can be prevented with only minor system modifications. [5] proposes to encrypt swap file data with a session key, which is discarded on system shut down, thus making the contents of the swap file unreadable.

**Introduce** - After creating a new encrypted file system, the owner is the only one who can issue key files to give additional users access to the secret data. This right to introduce new users can be granted to especially trusted users to relieve the owner from administration work. Users with the ”introduce” right can issue new key files or modify existing ones. These files can only contain rights the issuing user owns himself, but never the ”introduce” right.

**Admin** - This right, in combination with ”introduce”, allows a user to issue key files containing any rights (read, write, export, introduce), except ”admin”. If granted the ”admin” right, a user can also mount the encrypted file system exclusively, so that no other user can access it while he is using it.

To access a file system with owner rights (low level access to the partition, file system modifications, introduction of new administrators), the *Owner Key* must be presented. The driver compares the hash value of the presented *OK* with the value stored in the container header and grants access if both are equal.

The rights described above are granted only for processes in the session (a group of processes that share the same controlling terminal) from which the user authenticated himself to the driver. This allows the user to work with programs that need access to the encrypted data on one terminal without having to give all his processes such access rights.

# 3 Implementation

## 3.1 Overview

FSFS has been implemented as a kernel module for the Linux 2.2 kernel. It is based on the idea of the loop block device (like ppdd and the original loop device encryption) and uses parts of its code.

Figure 2 shows how the encryption driver is embedded into the Linux file system stack and how container files are used. The encrypted file system inside the container file is of type *A*, the container file itself resides on a file system of type *B*.

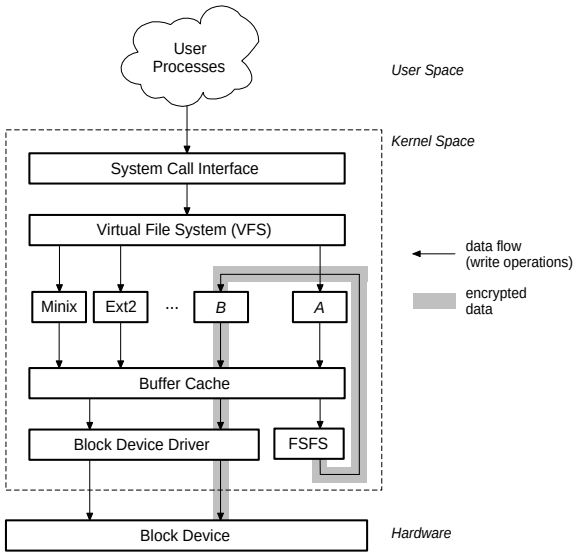


Figure 2: FSFS in the Linux file system stack

The FSFS driver works with stock Linux kernels and requires no modification of the kernel sources. FSFS can be installed and removed without a reboot and requires root privileges only during installation.

## 3.2 Design

FSFS has an extremely modular design to make it as flexible as possible (Figure 3). The authentication and cryptographic modules can easily be replaced to adapt to different security and performance requirements. Furthermore, the key data storage (*Owner Key File* and *User Key File*), currently a file, could be realized as a smart card or some other secure storage media.

The heart of FSFS is the device driver module in the kernel. It consists of three components:

**Key Management:** This component is used to configure the driver. It reads the *SEK* from the setup program in user space and sends it to the cryptographic module to be used.

**Block Transfer Handler:** Here, the access to an encrypted file is translated to an access to the container file. Using one of the cryptographic modules,

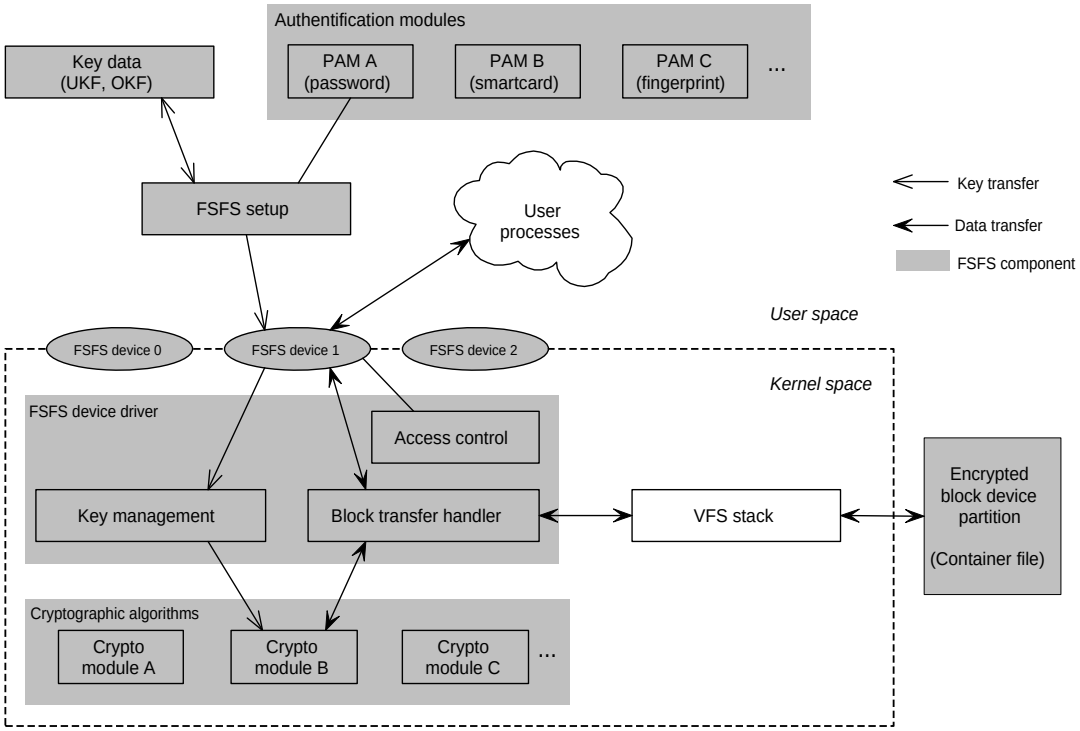


Figure 3: Components of FSFS

blocks of data that are written to/read from the file system are encrypted/decrypted here.

**Access Control:** The Access Control component determines for each access to the FSFS device whether the user/process requesting the transfer is allowed to access the encrypted file system. This component takes care that no secret data is transferred to unencrypted parts of the directory tree without special permission.

The setup program in user space forms the second major part of FSFS. It is used to validate user input, authenticate users, set up encrypted file systems, to transmit the SEK and related parameters to the driver (log on), and to request the driver to destroy the secret encryption key and to release the encrypted partition (log off).

FSFS comes with several cryptographic modules. Each module implements a different cipher. Up to now, the AES (Rijndael), Blowfish, Serpent, Skipjack, and XOR encryption algorithms have been implemented. Additionally, the cryptographic modules provided by the *KernelI project* [6] can be used with FSFS. Each of the currently active encrypted partitions could use a different cipher. All ciphers work in cipher block chaining mode (CBC) and use whitening to increase the level of cryptographic protection.

By default, FSFS uses password based user authentication. The contents of the user key file *(UKF)* can only be accessed with the correct user password *(UP)*. Additionally, the identity of a user could be further verified using Pluggable Authentication Modules that authenticate a user with the help of a smart card reader, fingerprint scanner, or by other means.

## 3.3 Performance

As one would expect, there is quite a performance hit when working with an encrypted file system. A benchmark that eliminates all caching effects showed that the raw block transfer is slowed down to approximately 25% – 60% of the original performance, depending on the encryption algorithm used. To measure how the user is slowed down when performing typical tasks, a real life scenario was used for another benchmark: Uncompress the kernel sources, compile them, create a new

tar archive with all the files, then delete them. When this is done on an encrypted file system, it takes only $15\% - 45\%$ longer than on a regular file system, again depending on the encryption algorithm. These more promising values reflect the performance drop in every day use better than the low level benchmark.

Using a container file instead of a raw disk partition counts for about 60% of the performance loss, the other 40% are caused by the additional work for encryption/decryption during the block transfer.

# 4 Conclusion

FSFS lets non-privileged users encrypt their secret files and flexibly share them with others. Users can encrypt any block device, from floppy disks to complete hard drives. The use of container files appears to be a good compromise between flexibility, performance, and ease of use and makes it easy to create backups of the encrypted files. Some precautions have been taken to prevent secret data from being stored on disk in plaintext. FSFS is useful even if no cryptographic protection of files is needed. The user management that FSFS provides lets users without administrator privileges easily establish groups of users who can access their files.

While FSFS offers a high level of cryptographic protection against attacks while the encrypted file system is not in use, it is still vulnerable to attacks by a malicious user with root privileges or if the operating system itself has been compromised. This is a general problem of all software based file system encryption schemes because they all rely on the integrity of the operating system. A more detailed discussion of some attack scenarios and how FSFS handles them can be found in [7].

## 4.1 Status and Future Work

FSFS has been developed as part of a diploma thesis [7] at Chemnitz University of Technology, Germany. It is now a stable file system encryption driver and has proven its suitability for everyday use in different scenarios on both single and multi user systems. FSFS is currently only available for the Linux 2.2 kernel. The next development steps include adapting FSFS to the 2.4 kernel and making use of the new functionality of this kernel, especially regarding virtual memory encryption and device handling.

Practical experience has shown that a simple way of user key file distribution is needed for the efficient use of FSFS. It is planned to investigate how FSFS can be integrated into different public key infrastructures to facilitate the automated distribution of (new) key files.

# References


[1] M. Blaze. A Cryptographic Filesystem for Unix. In *Proceedings of the First ACM Conference on Computer and Communications Security,* pp. 9-16, November 1993

[2] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic Filesystem for Unix. Unpublished Technical Report, July 1997
`ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz`

[3] A. Latham. ppdd - practical privacy disk driver documentation.
`http://linux01.gwgd.de/~alatham/ppdd.html`

[4] V. Samar and R. Schemers (SunSoft). Unified Login with Pluggable Authentication Modules. Open Software Foundation Request for Comments 86.0, October 1995

[5] N. Provos. Encrypting Virtual Memory.
USENIX Security Symposium. Denver, CO, August 2000

[6] The International Kernel Patch.
Website: `http://www.kerneli.org`

[7] S. Ludwig. Verschlüsselung von Dateisystemen unter Linux (Linux File System Encryption). Diploma Thesis. Chemnitz University of Technology, Department of Computer Science, Operating Systems Group, November 2000
`http://osg.informatik.tu-chemnitz.de/publikat/da/00/Ludwig00.pdf`