# Design of a Cryptographic Filesystem for Linux

**Programming Assignment 2**
**Operating Systems Internals, Spring 2002**

*Project Team Members*
**Madhusudhan Jujare, Andrew Jauri, Maruti Gupta, Satyajit Grover, Harkirat Singh, Ansuya Negi, Sashikiran Rachakonda**

**Table Of Contents**

## Introduction

This document describes the design and implementation of a cryptographic filesystem abstraction for Linux based on the requirements specified in the second assignment, which can be briefly described as follows:

1. Modularity - The filesystem must be implemented as a loadable device driver using the Linux module abstraction
2. Persistence - The filesystem must persist as a file within some other filesystem
3. RAM based – The filesystem must be able to read and open files in the main memory.
4. Cryptographic protection - The filesystem must provide cryptographic protection so that at any time, any data from the filesystem must not be present in an unencrypted form on the disk.
5. Inode based - Full file-system semantics, this would include operations such as opening a file, reading and writing to a file, making directories and all such operations expected from a normal filesystem.

We have come up with a filesystem design, which fulfills all the above requirements, and is referred to as the CryptExt2 filesytem.

The overall description has been organized into five sections. The next section describes in brief the existing Linux file system structure and how our module fits in the scheme of things. The third section investigates the different approaches studied for the design of a cryptographic file system and gives a high-level overview of the design architecture that we came up with, the reasoning behind it. The overview is given from the user's as well as the developer's point of view. In section IV, we get into some of the implementation level details such as the data structures used, functions added and changes suggested and implemented. We conclude in the fifth and the last section with the current status of our implementation and some of our key learnings from this project. In each section, we also show how the design fulfills the requirements of the project.

## Linux Filesystem Architecture

Linux can support multiple filesystems through a concept called the Virtual Filesystem (VFS). VFS is a special kernel interface layer software which handles all system calls related to a standard Unix filesystem. It then proceeds to map these system calls onto the functions supported by the filesytem the file happens to belong to. Thus, any file system in Linux must implement the interface presented by the VFS in order to work as can be seen from figure 1.1.

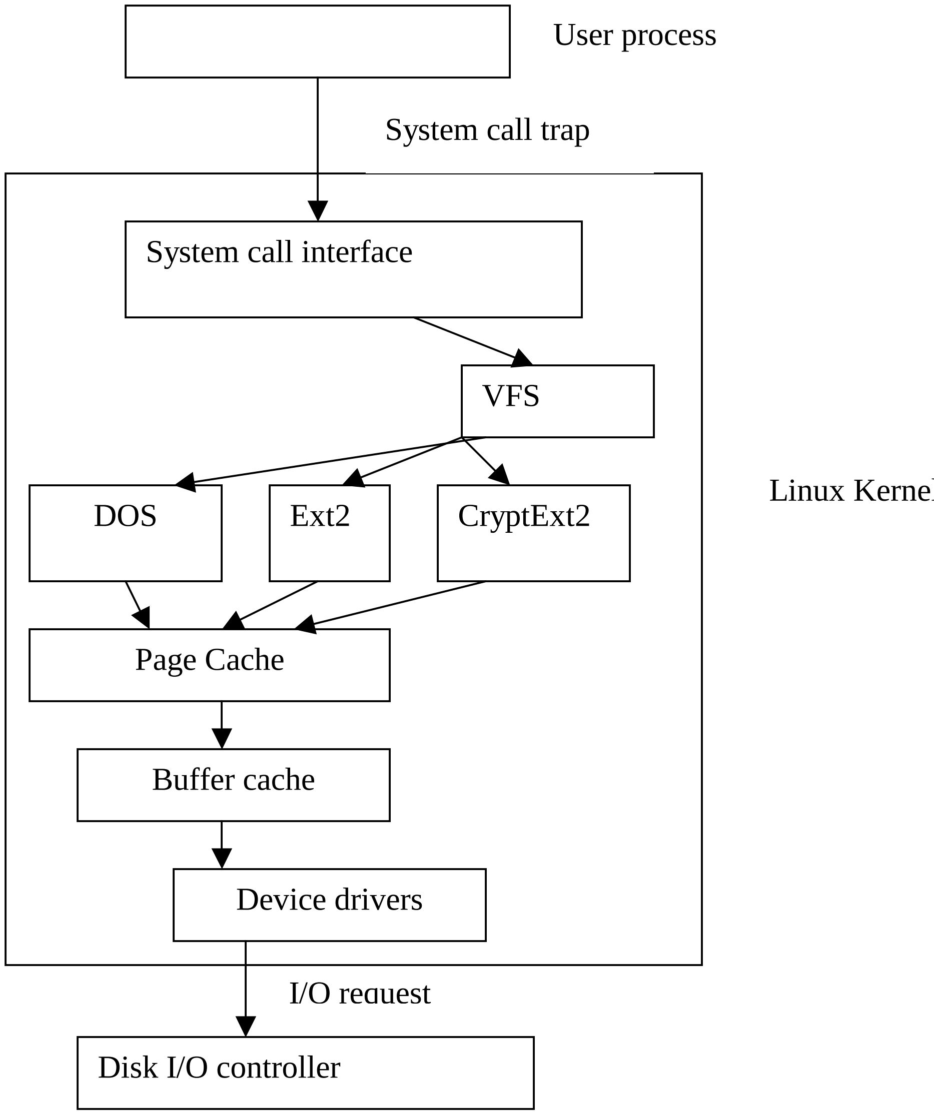


**Figure 1**

Figure 1 traces how file operations are traced from the user process to the disk I/O controller. Each system call related to files first goes through the VFS layer which then redirects the call to the filesystem on which the file belongs through the use of a common file model. The common file model can be thought of as *object-oriented.* VFS defines some data object types, which also contain the functions to manipulate the data within them. These objects types are then used in the implementation of each file system module and can be accessed by VFS when the module is registered with the operating system.

The common file module consists of the following object types:

1. superblock object
2. inode object

3. file object
4. dentry object

The CryptExt2 filesystem sits between the VFS layer and the physical device layer. It is implemented as a loadable device driver, thus satisfying the first requirement of modularity.
The page cache and buffer cache layers are a performance optimization used in Linux in order to minimize disk accesses as much as possible.

## Cryptographic Considerations

This section deals with the requirement of keeping the user data in an encrypted form on the disk. There are currently implementations of cryptographic file system available for example CFS [2] for Unix, TCFS [3] for Linux, and EFS [1] for Windows. Some of these were examined for their potential use in the project. These filesystems provide security to user's sensitive data by keeping data encrypted on the disk and using passphrase based mechanisms to obtain authorization to the encrypted data. Thus the data in the filesystem is protected in the event someone gets physical control over the storage unit.

There are several possible ways to develop a cryptographic filesystem. We could start from scratch and write a filesystem that encrypts and decrypts data before submitting it to the user application or we could take an existing filesystem and modify it so that it enables cryptographic protection at the right places. The second approach is used widely since it avoids duplication of work already done and is faster to implement. In the second approach, the overall design then depends upon the existing file system that is selected for modification. We identified two such possibilities, the RAMFS filesystem and the ext2 filesystem. We also looked at the CFS filesystem as another possibility. We explore the pros and cons of each of these filesystems and explain the reasons why we ended up selecting the ext2 fs.

**CFS**

The CFS filesystem, implemented on the Debian distribution is implemented entirely at user level. CFS runs a daemon called cfsd which uses regular Unix system calls to read and write the file contents, which are encrypted before reading and decrypted after writing as required. It is simple and easy to understand. However, some of the drawbacks with CFS are

1. It does not encrypt directory information and file size and access times, thus making it easy for an attacker to locate the encrypted files on disk.
2. Data in memory may be paged in or out on a paging device in an unencrypted form. Since CFS is implemented on an NFS client-server architecture, this can be especially bad if the paging device is on a remote machine.
3. It incurs considerable performance overhead since it is not implemented as a full filesystem and it uses system calls to store and copy data back and forth
4. It does not use a standard API for encryption and decryption algorithms thus making it harder to change the encryption/decryption mechanism.

5. It requires keys for each directory created by the user under the system thus requiring the user to remember a large set of keys assuming they enter a different one for each directory.
6. The key files are accessible to the superuser.

**RAMFS**

This is entirely a RAM-based filesystem, i.e. it does not have any mechanism for persistent storage of data, which is one of the requirements of the project. Like CFS, it has a very simple design. Data needs to be encrypted/decrypted only during the mount/umount process. However, we found it lacking in several respects some of which are mentioned below:

1. Due to the lack of a facility for persistent storage, it would require one to come up with one's own storage mechanism. It didn't seem a good idea to write something so readily available in other filesystems from scratch
2. Since the entire filesystem is loaded in the RAM during the mount process, it requires the decryption of the entire filesystem data even if all the files are not actually being used. The decryption process considerably slows the mounting process.
3. Keeping the entire filesystem in main memory imposes a huge memory overhead on the system. Also, due to the constraints of main memory size, the file system cannot accommodate files beyond a certain size.

**Ext2 FS**

This is the most widely used filesystem on most Linux distributions. It is stable and gives good performance. Since it is so widely used, it does not make a significant change in the way users need to use the filesystem. However the price to pay for stability is that it is complex in design and therefore requires significant effort in understanding the system. Thus, we decided to go with ext2 filesystem since it seemed more reasonable and would require less work on designing the file system (of which there are several good ones already) and allow us to focus more on enabling the cryptographic protection in the filesystem.

## CryptExt2 User Interface

As stated above, CryptExt2 is based on the Ext2 filesystem. It has been implemented by modifying the current ext2fs implementation by incorporating encryption/decryption and authentication procedures for file data access.

CryptExt2 provides transparent Unix file system interface to the user application. Once the CryptExt2 is mounted in the user's directory, the user is unaware of the underlying encryption/decryption mechanism in place.

The following are the steps a user needs to create and use the CryptExt2 filesystem, once the CryptExt2 module has been installed in the system.

1. The user is first required to create a file system using a modified version of the mke2fs utility. This prompts the user for a passphrase and the type of encryption algorithm they would like to use. Currently, there are only 2 choices available, i.e. none and Triple Des. However, it is easy to add more encryption mechanisms to this list. This step need only be done once, i.e. when the user starts to create a CryptExt2 filesystem. The user can create more than one filesystems, if so desired, but ordinarily it would be used only once.

   mke2fs <filesystem name> <size of the file in 1K blocks>

2. Next, the user issues the mount command (Note: we assume that the user has a limited capability of mounting a filesystem with permissions to user only. Ordinarily this privilege is restricted to root only). The user must first create an empty file as the name of the filesystem to be mounted.

   touch <name of filesystem file>
   mount –t cryptext2 <file to be mounted> <mount point.> -o loop

   where:
   -t : stands for the type of CryptExt2 file system.
   -o loop: to declare a loopback device.

   Here, the user is prompted for the passphrase and algorithm again. This is then verified using an authentication mechanism with the password stored on the disk when the filesystem was created. If authentication succeeds, then the mount proceeds and the user can work as they normally would in any other filesystem. The filesystem would transparently encrypt and decrypt files as they are written or read from the disk, as appropriate. Thus encryption and decryption takes place only when files are specifically being read or written from the disk. At no time is the data on disk kept in an unencrypted format.

3. At the end of the session, the user must unmount the file system, so that the system does not remain vulnerable to an attack from "superuser" as is when the filesystem is mounted. Unmount works the same as usual.

   umount <mountpoint>

To implement the above interface, we modified three sets of source trees

1. CryptExt2 module, (derived from the ext2 module source)
2. mke2fs utility
3. mount utility

Besides these changes, we were also required to make some changes in the source code of Linux 2.4.18 kernel. There are two ways in which the encryption/decryption policy can be implemented depending upon the level of security desired by the user.

1. The first approach takes a paranoid view and keeps data encrypted in memory until it is requested by the user application. Thus, even if the data were paged out, the data would be encrypted and decrypted once it was accessed again. This approach would make the system very secure since until the user accesses the data, it remains in encrypted form and is decrypted only when it is in use. However, keeping the data encrypted in memory imposes a significant performance penalty on the system.

2. The second approach decrypts data as soon as it is read from the disk and keeps it in unencrypted form in the memory until it is written back to the disk. This is more efficient since the performance overhead of encrypting/decrypting is considerably reduced.

The user may select one of these approaches depending upon the level of security desired, however both may not exist at the same time. In the next section, we describe in detail how these two approaches can be implemented in the system.

## Implementation Details

We start with a description of the data structures used and modified for the implementation of CryptExt2.

1. The superblock structure of CryptExt2 mirrors that of the ext2 filesystem except for the following changes. The ext2 superblock structure has a 197 word[32-bit] long padding at the end which is used by CryptExt2 to define some additional fields and reduce the padding size to 184 32-bit words.

*struct crypt_auth_struct:*

| | |
|---|---|
| __u32 | s_algo_type; |
| __u8 | s_key[32]; |
| __u8 | s_hash[16]; |

*s_algo_type* – this field is used for storing a user choice of encryption/decryption algorithm.

*s_key field* – this field is used for storing the randomly generated key at the time of creation of the superblock using the mke2fs utility. This key will be used for encryption and decryption of information.

*s_hash* – this field is used for storing the md5 digest on the passphrase (which can be up to 64 characters long) entered by the user. This digest is used to encrypt the superblock itself.

As stated earlier, there are two keys used, one is derived from the md5 digest of the passphrase and is used to encrypt the superblock. The other key is generated using a random number generating algorithm during the creation of filesystem and is stored in the superblock along with the md5 hash. This other key is used to encrypt and decrypt the filesystem data and the inode descriptor blocks as well as the group descriptor blocks. Since this key is not passphrase dependent, the user can change the passphrase without having to worry about decrypting the entire filesystem data with the old key and then encrypting it again. The random generating algorithm uses the linux /dev/random device to generate random numbers.

### *Changes to mke2fs Utility*

The algorithm for initialization of superblock and its encryption in the mke2fs utility.

```
algorithm  create_crypeExt2fs_superblock
input:
        pass phrase (used for encryption of superblock and user authenticity)
        choice of encryption algorithm (None, DES, TrippleDES etc..)
output:
        encrypted superblock with initialized crypt_auth_struct fields
        create ext2 superblock;
        hash the pass phrase entered by user using MD5;
        generate a random key used for encryption and decryption;
        store the algorithm type in the superblock;
        store random generated key in the superblock;
        store the hash in the superblock;
        encrypt the superblock using the key generated based on passphrase;
```

Encryption mechanism for the other filesystem data blocks: This is implemented in the write_blk() function, which is the one called ultimately to store the other file blocks onto the disc. Thus this function first calls the encrypts the blocks and then stores them onto the disk.

### *Changes to the mount utility*

We have added a function called the crypt_mount() function that is called whenever a user tries to mount a filesystem of type CryptExt2. This function can also be implemented as a separate program and placed in the /sbin/ directory in a file named as mount.cryptext2. The mount utility automatically checks this directory for such a file for each type of filesystem. This would then require no changes in the mount program itself.

The crypt_mount function does the following:

1. Turns off the echo settings for the terminal
2. Prompts the user for a passphrase from the terminal

3. Turns on the echo settings
4. Prompts for the algorithm type
5. Generates a 128-bit hash from the passphrase using the md5 algorithm
6. Packs all the mount options (including the mountpoint and filesystem file etc), the hash and the algorithm type in a string and passes it to the sys_mount system call.
7. The sys_mount system call in turn calls the corresponding read_super function implemented by the CryptExt2 module, which takes care of the authentication process.
8. If the authentication process is successful, then the filesystem is mounted, else an error message is generated and the mount fails.

### *CryptExt2 module*

This is where some of the major changes are made. We start with the changes made for the decryption/encryption of the super block. The following two functions were modified for this process.

1. Cryptext2_read_super function
2. Ext2_sync_super functions.

These functions are implemented in the super.c file of the CryptExt2 module.

The cryptext2_read_super() function is called when the superblock is read during the mounting process and is implemented as follows:

1. Get the sector size from the hardware device and use that as the blocksize, if it is greater than the one defined by the filesystem blocksize.
2. Parses the options sent in from the mount system call, extracts the 128-bit hash generated from the passphrase and the encryption algorithm type.
3. Reads the superblock from the disk
4. Decrypts the superblock using the key and the algorithm passed from the mount function.
5. Extracts the md5 hash stored on the superblock
6. Compares the hash with the one passed from mount. If they are equal, then authentication succeeds and it proceeds with the other checks and initialization calls which are similar to those implemented by the read_super function of the ext2 filesystem. Otherwise it fails and returns a null object. It also writes an error message in the system log.

The ext2_sync_super() function is called when the superblock is flushed to the disk so as to maintain the latest copy on disk. It is called from several functions, mainly from ext2_write_super, ext2_remount, ext2_put_super and ext2_error.

1. Encrypts the superblock
2. Writes the encrypted superblock to the disk
3. Keeps a decrypted copy in memory for reference.

**File and Metadata Encryption**

There are two ways in which this information can be encrypted in the CryptExt2 file system. Either can be used depending upon the security policy desired by the user.

**First Approach**

In this approach, data is kept encrypted in the memory and is decrypted only on user request. Thus, data is present in decrypted form only in user space and is encrypted as soon as it passes out of it. To implement this approach, we change the functions declared in the file_operations structure, which is assigned separately for regular files and directory files.

The file_operations structure contains function pointers to functions describing file operations.

```
struct file_operations ext2_file_operations = {
	llseek:		generic_file_llseek,
	read:		crypto_generic_file_read,
	write:		crypto_generic_file_write,
	ioctl:		ext2_ioctl,
	mmap:		generic_file_mmap,
	open:		generic_file_open,
	release:	ext2_release_file,
	fsync:		ext2_sync_file,
};
```

Here, we have changed the functions for read and write to call crypto_generic_file_read and crypto_generic_file_write respectively instead of generic_file_read and generic_file_write functions implemented by VFS. They are implemented in the kernel in the file mm/filemap.c.

*crypto_generic_file_read():* This function is similar to the generic_file_read function except for the following changes.

1. Figure out the block in the page which is being currently read
2. Decrypt the block using the key in superblock
3. Copy the decrypted buffer to user space

*crypto_generic_file_write():* This function is similar to the above function except that instead of decrypting, it encrypts data.

1. Find the block in the page currently being written.
2. Decrypt the block, write user data onto it.
3. Encrypt the block using the key in the superblock

Note: If user reads/writes more than 1 block of data, then the above steps are repeated until the entire data is decrypted or encrypted as appropriate.

*Directory operations:* The following functions are declared for directory operations using the file_operations structure.

```
struct file_operations ext2_dir_operations = {
        read:       generic_read_dir,
        readdir:    ext2_readdir,
        ioctl:      ext2_ioctl,
        fsync:      ext2_sync_file,
};
```

We haven't made any changes to the functions declared in this structure since all the functions in this structure end up calling ext2_get_page and ext2_put_page functions, which have been changed.

*ext2_get_page():* This function is called each time a directory operation takes place, for example a directory read, or lookup operation.

1. Determines the valid data blocks in the page to be decrypted using the file offset and the number of the page to be accessed in the given inode.
2. Decrypts the valid data blocks and returns the decrypted data

*ext2_put_page():* Once the directory operation is completed, ext2_put_page is called.

1. determines the blocks to be encrypted using similar techniques in ext2_get_page.
2. encrypts the blocks in page

This approach does not take into account memory mapped files. We can use a similar technique as with directory operations for encryption/decryption of inode blocks and bitmaps.

**Second Approach**

In this approach data is decrypted as soon as it is read from the disk and kept in unencrypted form in the memory until it is written back to the disk by changing page cache operations. Linux has generic file operations for data IO through the pagecache. This means that the data will not directly interact with the file- system on read/write/mmap, but will be read/written from/to the pagecache whenever possible. The pagecache has to get data from the actual low-level filesystem in case the user wants to read from a page not yet in memory, or write data to disk in case memory gets low.

To enable encryption/decryption right after data is read/written from disk, we need to change the address_space_operations for the inode object, which is declared in the address_space structure as shown below.

```
struct address_space {
        struct list_head        clean_pages;
        struct list_head        dirty_pages;
        struct list_head        locked_pages;
        unsigned long           nrpages;
        struct address_space_operations *a_ops;
        struct inode            *host;
        struct vm_area_struct   *i_mmap;
        struct vm_area_struct   *i_mmap_shared;
        spinlock_t              i_shared_lock;
};
```

An address_space is some kind of software MMU that maps all pages of one object (e.g. inode) to another concurrency (typically physical disk blocks). The a_ops field defines the methods of this object and host field is a pointer to the inode this address_space belongs to. The a_ops field contains page cache operations as shown below.

```
struct address_space_operations {
        int (*writepage)(struct page *);
        int (*readpage)(struct file *, struct page *);
        int (*sync_page)(struct page *);
        int (*prepare_write)(struct file *, struct page *, unsigned,
                                unsigned);
        int (*commit_write)(struct file *, struct page *, unsigned,
                                unsigned);
        int (*bmap)(struct address_space *, long);
};
```

*readpage*: The readpage function is called each time data is being read from disk for read/mmap operations. This function is modified to decrypt the data once data is read from the disk.

*syncpage*: As the name suggests this function is called whenever data needs to be brought in sync with the disk. This function ultimately ends up in calling the writepage function described below.

*writepage*: For writing to the filesystem two pathes exist: one for writable mappings (mmap) and one for the write(2) family of syscalls.

In mmap case writable mmap pages are marked dirty if they are changed. The bdflush kernel thread that is trying to free pages, either as background activity or because memory gets low will then try to call writepage on the pages that are explicitly marked dirty.

All that the write page function does is to write the full pages to disk. Before these pages are written to disk they are encrypted using the key in the superblock. In normal file I/O write system call case prepare_write and commit_write functions are used. The prepare_write() function (as the name suggests) is called right before data is written to ensure that the write operation has the allocated the number of blocks required. The commit_write() function is called after the data is written to the blocks, to mark the blocks as dirty. These functions are called every time data is modified in the files, and they in turn don't invoke any actual write-back of the data to disk. Instead, when the kernel runs low on memory (invokes bdflush() thread) or needs to update/sync the disk (invokes the kupdate thread), then only are the dirty buffers written to disk. This is done at the buffer cache layer through the try_to_free_buffers function, which eventually calls write_locked_buffers. This function starts the actual I/O operation for the buffers in the dirty list.

This is where we implement the changes for the encryption of data blocks specific to CryptExt2 filesystem. The blocks specific to the CryptExt2 filesystem are identified by tracing the pointer from the buffer_head structure all the way to the superblock structure which, contains a field storing the name of the filesystem to which the buffer belongs. This four level of encryption may result in a degradation in performance, which can be improved by storing a bit in the buffer_head structure that can be used to determine whether the block needs to be encrypted or not.

Similarly, the buffer cache level encryption can be done to the blocks (related to CryptExt2 filesystem) swapped out to disk.

## Current Implementation Status

The current status of implementation in each of source trees as mentioned in earlier sections is given below.

**mke2fs utility:** All changes related to this utility are complete and have been tested to work.

Currently we get the passphrase and the algorithm type from the user. In future, it can be modified to also get the security policy from the user (once both the approaches have been implemented). This utility works completely as desired. It generates afilesystem with an encrypted superblock, which can be mounted using the ordinary mount utility without giving the right password and the algorithm type.

**mount utility:** This utility has also been changed accordingly and the changes have been tested to work. Thus a CryptExt2 filesystem will fail to mount if given the wrong password.

**CryptExt2 module:** All the changed pertaining to filesystem authentication, superblock encryption/decryption and the first approach described above for encrypting and decryption regular file data are complete. We use Triple Des for the encryption/decryption of the superblock and a simple encryption algorithm to encrypt/decrypt the filesystem data.

## Conclusion

Our design of the CryptExt2 filesystem as presented in this document meets all the specified requirements. Under time constraints we have managed to implement a working prototype with some features. It is flexible enough to allow the incorporation of other features.


## Acknowledgements

We would like to acknowledge the following for providing on the web some of the source code used in this project.

1. Message Digest (md5) implementation – L. Peter Deutsch
2. Keymaker algorithm for generating a random key – Chris Holloway
3. Triple DES implementation – FreeSWAN project


## References


- Matt Blaze: A Cryptographic File System for Unix. In Proceedings of 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, November 1993, pp. 9{16.ftp://ftp.research.att.com/dist/mab/cfs.ps
- Daniel P. Bovet and Marco Cesati, " Understanding the Linux Kernel", O'Rilley Publishers, 2001.
- Linux Page Cache: http://www.moses.uklinux.net/patches/lki-4.html
- Linux Virtual memory management: http://kos.enix.org/pub/linux-vmm.html
- Paging and swapping in linux http://home.earthlink.net/~jknapka/linux-mm/pagecache.html