# KERNEL-LEVEL IMPLEMENTATION OF AN ENCRYPTING FILE SYSTEM

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Science with Honors in Computer Science from the College of William & Mary in Virginia,

by

Brian K. Dewey

Accepted for

Williamsburg, Virginia

May 1996

## Abstract

The need for secure storage of information has become increasingly important in the information age. Data encryption is the most powerful tool available to protect information. A modification to the Linux operating system kernel has been developed that allows the transparent encryption and decryption of an entire block device, providing a high degree of security at a low performance and inconvenience cost to the user.

# Contents

1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2 Design of the Linux Secure File System . . . . . . . . . . . . 7

2.1 Fundamentals of data encryption . . . . . . . . . . . . . 8

2.2 Overview of Linux file system support . . . . . . . . . . 10

2.3 Implementation of LSFS . . . . . . . . . . . . . . . . . . 13

3 Performance analysis . . . . . . . . . . . . . . . . . . . . . . . . 25

4 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

A Selected source code listings . . . . . . . . . . . . . . . . . . . 30

A.1 encrypt.c . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

A.2 make_request() . . . . . . . . . . . . . . . . . . . . . . . . 33

A.3 getblk() . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

A.4 wait_on_buffer() . . . . . . . . . . . . . . . . . . . . . . . 37

A.5 lsfs_read_super() . . . . . . . . . . . . . . . . . . . . . . . 37

# List of Figures

2.1 Algorithm for CBC mode encryption . . . . . . . . . . . . . . . 10

2.2 Algorithm for CBC mode decryption . . . . . . . . . . . . . . . 11

2.3 Steps involved in reading data from disk . . . . . . . . . . . . 12

2.4 Algorithm for creating an LSFS superblock . . . . . . . . . . 16

2.5 Algorithm for loading LSFS superblock . . . . . . . . . . . . 17

2.6 Algorithm for enabling a buffer to perform transparent encryption/decryption . . . . . . . . . . . . . . . . . . . . . . . . 19

2.7 Algorithm for obtaining a buffer . . . . . . . . . . . . . . . . . 20

2.8 Algorithm for doing a device read/write request . . . . . . . . 21

2.9 Algorithm for reclaiming cache memory . . . . . . . . . . . . . 24

3.1 Comparison of file system performance . . . . . . . . . . . . . 28

# List of Tables

2.1 Properties exhibited by single-key cryptographic functions . . 9

3.1 Compilation . . . . . . . . . . . . . . . . . . . . . . . . . . 26

3.2 Large `tar` copy and extract . . . . . . . . . . . . . . . . . . . . 27

3.3 Concurrent `tar` file extraction and source compilation . . . . 28

## 1 Introduction

Ensuring information security has long been a fundamental job of multi-user operating systems. Most operating systems control access to files through a system of user identification and permissions, which provides an adequate level of security for most purposes. However, the persistent nature of disk storage creates a vulnerability that operating system permissions cannot redress. Briefly consider the following thought-experiment: sitting in front of your computer, you create a sensitive document and store it in a file on a floppy disk. You set the file permissions so that only you have the authority to read that document. The computer's operating system will steadfastly refuse to serve that information to any user but you. The document appears to be secure. Now, remove the floppy disk from the drive and hold it in your hand. Your sensitive document, encoded in a well-known and easy-to-read protocol of 1's and 0's, sits on the surface of that disk — and the operating system of the computer no longer controls who has access to that information. Someone who illicitly obtained your disk would have no problem reading your sensitive document on a different computer. Thus, when dealing with files that must remain confidential, a user would be wise to store the information on disk in such a way that even if the integrity of the physical media is compromised, the information stored on it remains safe.

Cryptography provides the best solution to this problem. Modern cryptosystems can protect files by rendering them meaningless to any person who lacks knowledge of a specific numeric key. Many encryption programs exist for Unix-based machines that can provide this type of security. Most run as command-line utilities and work on a file-by-file basis — i.e., the user creates a file using one program and then runs the encryption program separately to protect it. However, these utilities are typically awkward to use as they require the time-consuming and tedious ritual of running a separate program twice for every access to a confidential file.

Further, command-line encryption utilities suffer from three potential security compromises that stem from human neglect or error. First and most obvious, if a user ever forgets to run the encryption program after accessing a file, the data will reside on the disk in a vulnerable form. Second, the user must remember that deleting the unencrypted copy of the file usually won't remove the data from the disk. In most operating systems, deleting a file means its disk space has been marked available for future use. The user needs to remember to take whatever extra steps necessary to actually remove the unencrypted data from the disk. Finally, many applications create temporary files that store unencrypted pieces of your confidential data elsewhere on the disk. The user must make sure these temporary files are wiped off the disk to prevent a security compromise.

The Cryptographic File System (CFS) [3] substantially improves upon the security of command-line utilities by encrypting an entire file system. This approach has several advantages. First, the user must present his encryption key only when he begins his session with the file system, instead of providing it with each individual file access. Second, CFS transparently encrypts and decrypts all data stored in the file system, which ensures no information is ever on disk in clear form. This eliminates the problems of forgetting to encrypt a file when done, dealing with the residual images of deleted files on the disk, and keeping track of temporary files. Finally, instead of being implemented directly into an operating system kernel, CFS is implemented as a Network File Server (NFS) process. This allows people on many Unix-based platforms to use the software, and it also allows the software to store encrypted files on remote machines and keep the data protected as it crosses the network.

CFS is especially suited for network-based file access. When dealing with files stored on the local machine, however, a strong argument can be made that the encrypted file system support should be integrated directly into the operating system. First, the NFS protocol, which allows remote file access, adds a significant amount of unneeded overhead when dealing with a local disk. Second, since disks are slow even without encryption, modern operating systems are designed to minimize the number of times they must access the disk drive. This feature consequently minimizes the number of encryption and decryption steps needed for an integrated encrypting file system, which would boost performance. Finally, from a philosophical standpoint, it is one of the traditional jobs of a multi-user operating system to provide security for local disk contents — providing encryption capabilities is a natural extension of the operating system's responsibility.

This paper describes modifications to the Linux operating system kernel to provide encrypted file system support. The new file system, called LSFS (Linux Secure File System), has the simplified key management and transparent encryption and decryption of CFS. While the kernel implementation lacks the ability to work with remote file systems, it possesses a sizeable performance edge over CFS when dealing with local files.

## 2 Design of the Linux Secure File System

The primary purpose of the Linux Secure File System is to protect sensitive data from the vulnerabilities inherent in persistent storage on a disk. When LSFS is used on a particular block device, it guarantees that no file information will ever be stored on that device in clear format. Thus, even if the security of the disk itself is compromised, no data may be obtained from it without knowledge of the encryption key. LSFS has the following design goals:

- **Transparent access to data.** Files on an encrypted file system should behave like files on any other file system. Applications that access data on an encrypted file system should behave exactly as if they were accessing data on a non-encrypted file system.

- **Minimal performance impact.** While the encryption and decryption process is computationally expensive, the kernel should ensure that the user feels a minimal performance burden when using the file system.

- **Minimal inconvenience for the user.** The kernel should minimize the number of extra steps the user needs to take in order to use an encrypted file system. In particular, the kernel should minimize the number of times the user is required to enter the encryption key.

- **Thorough protection of file contents.** In order to gain the maximum benefit from cryptography, the encrypted file system should appear indistinguishable from a random sequence of bytes to any person who does not have the proper key. This means that file contents must be scrambled in such a way that it would be impossible to discover any structure in the encrypted version of the file, such as repetitions of byte patterns. Additionally, it should be impossible to examine two encrypted files and determine how the plaintext versions of those files differ.

- **Protection of file metadata.** The file system metadata should be encrypted to prevent any knowledge of file names, sizes, access times, etc.

The rest of this section provides an overview of data encryption and Linux file system support before detailing the design and implementation of LSFS.

## 2.1 Fundamentals of data encryption

Data encryption is the process of taking a meaningful sequence of data $P$ (called the *plaintext*) and transforming it with an encryption function $E$ and a key $K_e$ into an apparently random sequence of data $C$ (called the *ciphertext*). In order to restore the plaintext, the user must know the decryption function $D$ and decryption key $K_d$. Put in mathematical terms, one has a viable encryption scheme when $E(P, K_e) = C$ and $D(C, K_d) = P$ for all possible plaintexts $P$. Modern cryptosystems do not rely upon the

1. Given a key $k$ and a message $P$, the cryptographic function produces ciphertext $C$ such that $f(k, P) = C$ and $f(k, C) = P$.
2. Each system encrypts data using a key selected from a large enough key-space to make an exhaustive search of all possible keys practically impossible.
3. It is computationally infeasible to deduce $P$ if you know $C$ without knowledge of the key (and vice versa).
4. Given $P$ and $C$, it is computationally infeasible to deduce the $k$ that generated the transformation.

Table 2.1: Properties exhibited by single-key cryptographic functions

secrecy of the decryption function $D$ to provide security; instead, a modern cryptosystem is secure if, given $D$ and $C$, it is computationally infeasible to deduce the plaintext $P$ without knowledge of the decryption key $K_d$. *Single-key* cryptosystems are a specific subset of modern cryptosystems where $K_e = K_d$ — in other words, the key used to generate the ciphertext is the same key used to restore the plaintext. (See table 2.1 for a summary of the properties of single-key cryptosystems).

### DES

DES, the U.S. Government's standard for data encryption [1], is a single-key cryptosystem whose cryptographic strength has long been acknowledged. The DES encryption function uses a 56-bit key to encrypt 64 bits of plaintext. Plaintext longer than 64 bits is broken down into 64 bit blocks and each block is encrypted separately. However, it is important to ensure that two identical 64-bit blocks of plaintext do not encrypt to identical 64-bit

```
algorithm CBC_encrypt
input:  plaintext, key
output: ciphertext
{
  for(every block of plaintext)
    {
      if(encrypting first plaintext block)
        XOR plaintext block with initialization vector;
      else
        XOR plaintext block with previous ciphertext block;
      encrypt plaintext block and store in ciphertext;
    }
}
```

Figure 2.1: Algorithm for CBC mode encryption

blocks of ciphertext; this repetition could provide clues to a cryptanalyst as to the structure of the unencrypted data. To prevent this, the DES cipher may be used in *Cipher Block Chaining* (CBC) mode [2] (see tables 2.1 and 2.2). CBC mode uses the previous block of ciphertext as a mask for the current block of plaintext, which prevents identical blocks of plaintext from having the same ciphertext representation.[1]

## 2.2 Overview of Linux file system support

Support for file systems in the Linux kernel is implemented in three main layers: the file system layer, the buffer cache layer, and the device driver layer. Of these, the file system layer and the buffer cache layer are of primary importance to LSFS.

User programs view information on disk as variable-length files, while the

[1]When encrypting the first block of plaintext, CBC mode uses an *initialization vector* (IV) as the mask.

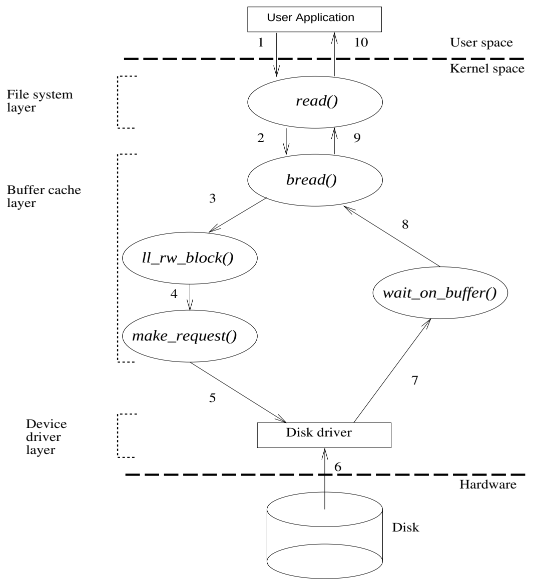
```
algorithm CBC_decrypt
input:  ciphertext, key
output: plaintext
{
  for(every block of ciphertext)
    {
      decrypt ciphertext block and store in plaintext;
      if(decrypting first ciphertext block)
        XOR plaintext block with initialization vector;
      else
        XOR plaintext block with previous ciphertext block;
    }
}
```

Figure 2.2: Algorithm for CBC mode decryption

disk driver views the disk as a numbered sequence of fixed-length blocks. The Linux file system layer translates a user program's request for a particular byte of a particular file into the corresponding request to fetch a block from a disk. The file system accomplishes this feat by maintaining its own metadata on disk that tells it which blocks belong to which files.

The buffer cache layer performs the critical task of maintaining the buffers in the computer's primary memory that hold the images of the blocks on the disk. Further, the buffer cache attempts to keep frequently accessed disk blocks in memory. Since disk access time is much slower than memory access time, the buffer cache can boost the performance of the overall system dramatically.

To illustrate the interaction of the operating system layers, figure 2.3 diagrams many of the steps required during a typical disk read operation.



Some steps involved in reading data from the disk: 1) The user application performs the `read()` system call and asks the kernel to return a specified number of bytes from a file. 2) The system call uses its knowledge of the file system to determine what disk block contains the bytes requested by the user program. It then calls the routine `bread()` to fetch the disk block into the buffer cache. 3) `bread()` looks for the block in the cache. If it is found, steps three through eight are skipped. However, if block is not already in the cache, `bread()` obtains an empty cache entry and tells `ll_rw_block()` to fetch the disk contents. 4) `ll_rw_block()`, having validated the disk access, passes it to `make_request()`. 5) `make_request()` locks the buffer cache entry and places a request in the disk driver's queue. This routine then returns without waiting for the I/O to complete. The process that initiated the disk read will go to sleep in the `wait_on_buffer()` routine (see step 7). 6) The disk driver will eventually find the request in its queue and pull the block off the disk. 7) Once the block has been read, the disk driver releases the lock on the cache entry. This wakes up the original call, which had been asleep in `wait_on_buffer()`. 8) `wait_on_buffer()` returns, which notifies `bread()` that the block has been read off the disk and stored in the buffer cache. 9) `bread()` returns the cache entry to the original `read()` call. 10) The `read()` call finds the requested bytes in the buffer and returns them to the application.

Figure 2.3: Steps involved in reading data from disk

## 2.3 Implementation of LSFS

### Encryption routines

The current implementation of LSFS uses DES encryption (see page 9). However, LSFS hides the actual encryption engine from the bulk of the operating system, making it a simple matter to change encryption algorithms to one that best suits the needs of the system.

### File system changes

As outlined on page 10, the primary job of the kernel file system layer is to translate requests for bytes of a file into requests for pages off the disk. LSFS uses the existing minix file system code to accomplish these tasks. However, in addition to the traditional file system responsibilities, the LSFS file system layer is responsible for managing the disk's encryption key.

LSFS requires that the user enter a pass phrase when doing any of the following operations: creating an encrypted file system (using the `mkfs` command), checking the consistency of the file system (using the `fsck` command), and preparing an encrypted file system for use (the `mount` command). For all three cases, LSFS uses the MD5 message digest algorithm [4] to hash the user's pass phrase down to 128 bits. LSFS does not generate the file system encryption key from the 128-bit pass phrase hash, though, because if the user ever wanted to change his pass phrase, the kernel would have to re-encrypt every block on the disk with the new key. Instead, LSFS generates a random[2] encryption key and encrypts that key using 56 bits of

[2]LSFS uses the *truerand* random number generator that is built into CFS. *truerand* uses fluctuations in machine timing to generate a truly random number.

the pass phrase hash. LSFS stores the encrypted copy of the key on the file system superblock.[3] Therefore, if the user ever wants to change his pass phrase, LSFS only needs to re-encrypt the key stored on the superblock instead of re-encrypting every block on the disk.

In addition to storing an encrypted copy of the key, LSFS stores a known string encrypted with the same hash on the superblock. This enables the kernel to verify that the user has entered the correct pass phrase when he attempts to check the file system consistency or prepare the file system for use — if the hash of the pass phrase decrypts the validation string to its known value, then the kernel knows that the proper pass phrase has been entered.

By storing the encryption key on the superblock, LSFS creates a bootstrapping problem. As described in section 2.3, the buffer cache uses the encryption key to encrypt and decrypt blocks as they are moved to and from the disk. However, when accessing the disk's superblock for the first time, the buffer cache does not yet know what the encryption key is, thus making buffer-cache layer decryption impossible. To circumvent this, the buffer cache treats the superblock as a special case and never performs transparent encryption or decryption when accessing that block on disk. Instead, it is the responsibility of the file system layer to encrypt fields on the superblock

[3] Every disk that contains a file system has a superblock that stores information identifying the type of file system on the device and containing important file system parameters. Before the kernel can use a file system, it reads the superblock from the disk and copies the file system parameters into memory. Since LSFS uses the same file system layout as minix, the structure of their superblocks are nearly identical. The LSFS superblock adds only two fields — one storing the an encrypted copy of the disk's key and the other field storing a validation string. The superblock is conventionally stored on the block number 0 of a device. Some file systems store backup copies of the superblock elsewhere on the disk; minix, upon which LSFS is based, does not.

as it feels necessary. Currently, LSFS only encrypts the key and validation fields of the superblock and, for simplicity, leaves information such as the number of blocks on the device and the number of inodes unencrypted. Simple modifications to LSFS would encrypt these fields as well.

Most of the modifications to the file system layer, therefore, have been confined to the routines that create and read an LSFS superblock. The algorithm *create_lsfs_superblock*, depicted in figure 2.3, is used by the **`mkfs`** command when creating a new encrypting file system; it merely adds the encrypted key and the encrypted validation string to the minix superblock. The algorithm *load_lsfs_superblock* (figure 2.3) is used when mounting an encrypted file system and checking its consistency. Its primary job is to ensure that the **`mount`** or **`fsck`** commands fail if the improper pass phrase has been entered. If the correct pass phrase has been given, it decrypts the key and stores it in the in-kernel copy of the superblock for use by the buffer cache layer (see page 18).[4]

Once the file system has been mounted with the correct passphrase, it follows the same access semantics as any other file system on the machine. Consequently, applications will be able to access the encrypted information without change. Further, while the file system is mounted, the traditional security methods of user identification and file permissions must be depended upon — for while the data resides on the disk in encrypted form, it will be

[4]Keeping a decrypted copy of the key in memory is a potential security compromise. However, the effects should be minimal, since the key resides in kernel memory only when the disk is mounted and only someone with root access could pull the key out of kernel memory. If someone has root access to the computer, then there are much easier ways to access the data while the device is mounted, and the additional risk of storing the decrypted key in memory is minimized.

```
algorithm create_lsfs_superblock
input:  pass phrase used for encryption
output: new superblock structure
{
  create minix superblock;
  create random encryption key;
  encrypt random encryption key with passphrase;
  store random encryption key on superblock;
  encrypt validation key with passphrase;
  store validation string on superblock;
  return superblock;
}
```

Figure 2.4: Algorithm for creating an LSFS superblock

decrypted whenever it is pulled into the computer's main memory.

When no pass phrase is given, a user may still access the data on the device. However, the data will remain encrypted, and the user will not be allowed to see any of the file system structure on the device. Thus, one may use a command such as `dd` that accesses the raw disk to create a backup of the encrypted file system block by block.

### Buffer cache changes

As previously outlined, the Linux buffer cache is responsible for getting blocks of data from a device and storing them for use by the rest of the kernel. The modified Linux kernel that supports LSFS performs the encryption and decryption when moving blocks of data between the buffer cache and the device. This keeps the data stored on the device in encrypted form yet keeps it in the buffer cache in clear form.

```
algorithm load_lsfs_superblock
input:  copy of the superblock from a disk
        pass phrase used for encryption
output: copy of superblock in kernel memory
{
  copy minix superblock information to kernel memory;
  decrypt validation string in superblock with passphrase;
  if(validation string <> known value)
    return with error;
  else
    {
      decrypt random encryption key using passphrase;
      store random encryption key in kernel memory;
      return kernel copy of superblock;
    }
}
```

Figure 2.5: Algorithm for loading LSFS superblock

To accomplish this, additional information is included in each individual entry in the buffer cache. In the unmodified Linux kernel, each entry in the cache contains a pointer to a buffer storing the data block, the device and block numbers that identify where the data comes from, and pointers to organize cache entries into lists (e.g., the cache keeps a list of dirty buffers, a list of locked buffers, a list of clean buffers in least-recently-used order, etc.). To support data encryption, four new fields have been added to each entry in the cache:

1. A flag indicating if the data block needs to be stored in encrypted form on the device.

2. A flag indicating if the kernel is currently in the process of reading

encrypted data from the device.

3. A copy of the key used for encrypting and decrypting the block.

4. A pointer to a *data transfer buffer* that is used to store the encrypted data on its way to or from the device.

When the kernel wants access to a block of data on a device, it first checks if that block is already mapped to a buffer cache. If not, the kernel finds a blank cache entry to store the data. At this point, the kernel checks its copy of the device superblock for an encryption key; if one is found, the kernel knows that the data on the device is stored in encrypted form. Therefore, when it obtains its blank buffer cache entry, it sets the `encrypted` flag to TRUE and generates the key that will be used to encrypt and decrypt that data block. (The subsection on "Encryption strategy" (page 22) describes how and why each block on the device has its own copy of an encryption key.) If the kernel had not found an encryption key in the device superblock, it would set the cache `encrypted` flag to FALSE. (See figure 2.7)

Then, whenever the kernel attempts to read the block from the device to store it in the cache entry, it checks the `encrypted` flag. If the flag indicates that the data is stored in encrypted form on the device, the kernel tells the device driver to read the data block into the data transfer buffer instead of directly into the cache entry. Then, when the device driver signals the kernel that the read is complete, the kernel uses the key stored in the cache entry to decrypt the data transfer buffer and store the plaintext in the cache entry.

Similarly, when the kernel wants to write a block from the cache back

```
algorithm prepare_encrypted_buffer
/* Performs the necessary tasks to prepare a
 * buffer for transparent data encryption.
 */
input:  disk buffer
output: disk buffer prepared for encryption
        or decryption
{
  if(buffer not already marked as prepared)
    {
      generate local key for buffer;
      store local key in buffer;
      get a free buffer from the cache;
      link the two buffers;
      mark the buffer as prepared;
    }
  return buffer;
}
```

Figure 2.6: Algorithm for enabling a buffer to perform transparent encryption/decryption

```
algorithm getblk     /* get a block in the buffer cache */
input:  device number
        block number
output: a buffer in the buffer cache that can now be
        used to store data from the device
{
  if(block in hash queue)
    {
       update LRU list;
       if(device uses encryption)
         prepare_encrypted_buffer;
       return buffer;
    }
  else
    {
       obtain free buffer;
       if(device uses encryption)
         prepare_encrypted_buffer;
       put buffer on hash queue;
       put buffer on LRU list;
       return buffer;
    }
}
```

Figure 2.7: Algorithm for obtaining a buffer

```
algorithm make_request     /* Sends a read or write request to
                              a device driver */
input:  device number
        block number
        locked buffer from buffer cache
        type of request(read or write)
output: none
{
  prepare device to receive request;
  if(buffer uses encryption)
    {
      unlock buffer;
      if(read request)
        mark buffer as waiting for encrypted read;
      else
        encrypt buffer data and store in data transfer buffer;

      /* the following command makes the data transfer */
      /* buffer the buffer that will be used for I/O */
      switch to the data transfer buffer;
      lock data transfer buffer;
    }
  submit request to device;
}
```

Figure 2.8: Algorithm for doing a device read/write request

to the device, it checks the `encrypted` flag to see if it needs to encrypt the block before writing. If it does, it stores the encrypted version of the data block in the data transfer buffer and tells the device driver to write the information in the data transfer buffer to the disk. (See figure 2.8.)

**Encryption strategy**

As described earlier, DES (the underlying encryption scheme used by LSFS) works by encrypting or decrypting an 8-byte block and then moving on to the next block (see page 9). Using DES in CBC mode prevents two identical 8-byte blocks of plaintext from having the same ciphertext representation. This encryption strategy is used when encrypting and decrypting a block of data from the device to mask any patterns in the data on that block. However, just as we want to prevent two identical 8-byte blocks of plaintext from having the same ciphertext representation, we want to ensure that two identical disk blocks on the same device do not appear identical when encrypted. The type of block chaining used in CBC is not an efficient algorithm in this situation because of the random access nature of individual blocks on a device. A CBC-like algorithm would require that if we want to read the $n$th block from a device, we also have to read the $(n-1)$th block to have the bitmask used in the block chaining process. If the user program changes the data on the $n$th block, it would require re-enciphering and re-writing all blocks coming after it on the disk.

A system of "master" and "local" encryption keys was used to solve this problem. The encryption key stored on the superblock of the device is the "master" key for the device. However, instead of being used directly to encrypt/decrypt blocks on that device, it is used to generate a "local" key for each block, and the local key is used for all of that block's reads and writes. (See the section covering the buffer cache, page 16.) Since each block on the device gets its own encryption key, even two identical blocks will appear different when encrypted.

A local key is generated by encrypting the block number with the master key. Because of the properties of DES encryption, if one does not know the master key, there is no feasible way to determine the local key from the block number alone. However, if an attacker is somehow able to determine the local key that was used to encrypt/decrypt a disk block, the knowledge of both the block number and the local key does not give enough information to determine the master key.[5]

### Buffer memory management

The Linux buffer cache does not occupy a fixed amount of memory. Instead, whenever the cache needs to obtain free buffers, it first checks if it can dedicate more memory to the buffers. If so, it will expand the size of the buffer cache to make room for the free buffers; otherwise, it gets rid of old cache data to make room for the new disk blocks. This allocation policy makes the cache more efficient because it enables the maximum amount of data to reside in memory, minimizing the number of times the operating system needs to access the disk.

However, the buffer cache's dynamic growth tends to rapidly claim all

[5]One possible mode of cryptanalysis, called the *known-plaintext* attack, depends upon knowing some plaintext $P$ and its ciphertext representation $C$ and using that information to determine the $k$ such that $E(P,k) = C$. I have found no literature suggesting that the DES cipher is vulnerable to this specific attack. However, even if there is a small vulnerability to the known-plaintext attack, the system of master and local keys should be no less secure than a conventional system that uses the same key to encrypt each block. If the cryptanalyst was able to determine a block's local key from examining the block's data — the first step in conducting the known-plaintext attack against this system — then he would have been able to determine the single key used in a conventional system from examining the same data.

```
algorithm try_to_free     /* attempt to reclaim cache memory */
input:  candidate memory page
output: success or failure indication
{
  for(every buffer on page)
    {
      if(buffer is not a data transfer buffer
         AND it is dirty or in use)
           return failure;
      if(buffer is a data transfer buffer
         AND its plaintext is dirty or in use)
           return failure;
    }
  for(every buffer on page)
    {
      if(buffer is not a data transfer buffer)
        {
          remove buffer from lists;
          if(buffer comes from encrypted device)
            {
               unlink plaintext and data transfer buffers;
               insert data transfer buffer on free list;
            }
        }
      else
        {
          unlink plaintext and data transfer buffers;
          remove plaintext buffer from lists;
          insert plaintext buffer on free list;
        }
      put buffer in unused list;
    }
  free page;
  return success;
}
```

Figure 2.9: Algorithm for reclaiming cache memory

available memory. Thus, in order for the cache to remain efficient, it needs to be able to relinquish its memory when needed by other parts of the operating system. The key algorithm in this process is *try_to_free* (figure 2.3), which checks if a particular memory page used by the cache can be relinquished to other parts of the operating system. The algorithm succeeds if all buffers on the memory page are clean (i.e., the buffer contains the same information as its corresponding disk block) and if no processes are using the buffer.

The presence of encrypted data transfer buffers in the cache memory space complicates the original algorithm. When the algorithm encounters a data transfer buffer on the page it is attempting to free, it needs to see if its corresponding plaintext buffer is clean and unused. If it is, then the algorithm may proceed; otherwise, the algorithm fails, since the data transfer buffer may not be reclaimed by the system as long as its plaintext equivalent is dirty or in use. If the algorithm does reclaim the memory of a data transfer buffer, it marks the corresponding plaintext buffer as available for use.

## 3 Performance analysis

The use of LSFS carries with it a performance price stemming from the overhead of performing a computationally-expensive encryption or decryption every time an LSFS disk is accessed. The degree of performance loss for a particular application will depend upon its I/O intensity and how efficiently the buffer cache performs to minimize actual disk accesses for the application. Three benchmark tests were used to measure LSFS' performance under various workloads. All benchmarks were performed on a 90 megahertz

| File system | Elapsed time (95% confidence interval) |
|---|---|
| Clear FS, regular kernel | $43.8 \pm 0.1$ |
| Clear FS, modified kernel | $44.7 \pm 0.4$ |
| LSFS | $44.9 \pm 0.2$ |
| CFS | $69.7 \pm 0.5$ |

Table 3.1: Compilation

Pentium with 32 megabytes of primary memory and a 700 megabyte IDE hard drive. Each benchmark was run ten times on each of four file systems — the minix file system implemented in the Linux 1.2.8 kernel, the same minix file system code running in the modified, LSFS-enabled kernel, the LSFS file system, and CFS. The tables in this section report the average execution time for the set of runs and the corresponding 95% confidence interval.

Table 3.1 shows the amount of time it took to compile the CFS software on each of the four file systems. This benchmark measures the file system performance under light-to-moderate workloads with moderate disk I/O activity (the CFS source contains roughly 9200 lines of code and 4750 semicolons). This benchmark exhibits the benefits the buffer cache brings to transparent disk encryption. LSFS, which uses the read/write caching built into the kernel, ran with only a 3% overhead. CFS, which uses no write caching, ran 59% slower.

Table 3.2 shows the results from copying a 6.4 megabyte compressed `tar` file from a remote file server onto the four file systems and extracting the archived files. This tests the performance of the file systems under heavy

| File system | Elapsed time (95% confidence interval) |
|---|---|
| Clear FS, regular kernel | $60.4 \pm 0.3$ |
| Clear FS, modified kernel | $62.1 \pm 0.8$ |
| LSFS | $177.2 \pm 1.0$ |
| CFS | $311.3 \pm 0.4$ |

Table 3.2: Large `tar` copy and extract

disk utilization with minimal cache benefits. The lack of efficient caching leads to a pronounced difference between the encrypting and non-encrypting file systems. In this test, LSFS runs slower than a non-encrypting file system by a factor of 2.9; CFS' poor performance, a factor of 5.1 slower than non-encrypting file systems, can be attributed to the additional overhead required to copy data through the NFS protocol.

Finally, table 3.3 shows the time taken to run both the CFS source-file compilation and the `tar` file extraction concurrently. This tests the file systems under a heavy load. It also stresses the cache as two processes compete for buffer space and generate more random disk access patterns. Under this burden, LSFS runs a factor of 2.1 slower; CFS runs slower by a factor of 3.8.

Figure 3.1 summarizes the performance of the four file systems. While LSFS runs slower than non-encrypting file systems, especially when caching benefits are reduced, the low overhead of the kernel implementation allows for a substantial performance improvement over CFS.

| File system | Elapsed time (95% confidence interval) |
|---|---|
| Clear FS, regular kernel | $97.6 \pm 0.3$ |
| Clear FS, modified kernel | $107.9 \pm 0.9$ |
| LSFS | $208.5 \pm 0.5$ |
| CFS | $378.9 \pm 0.5$ |

Table 3.3: Concurrent `tar` file extraction and source compilation

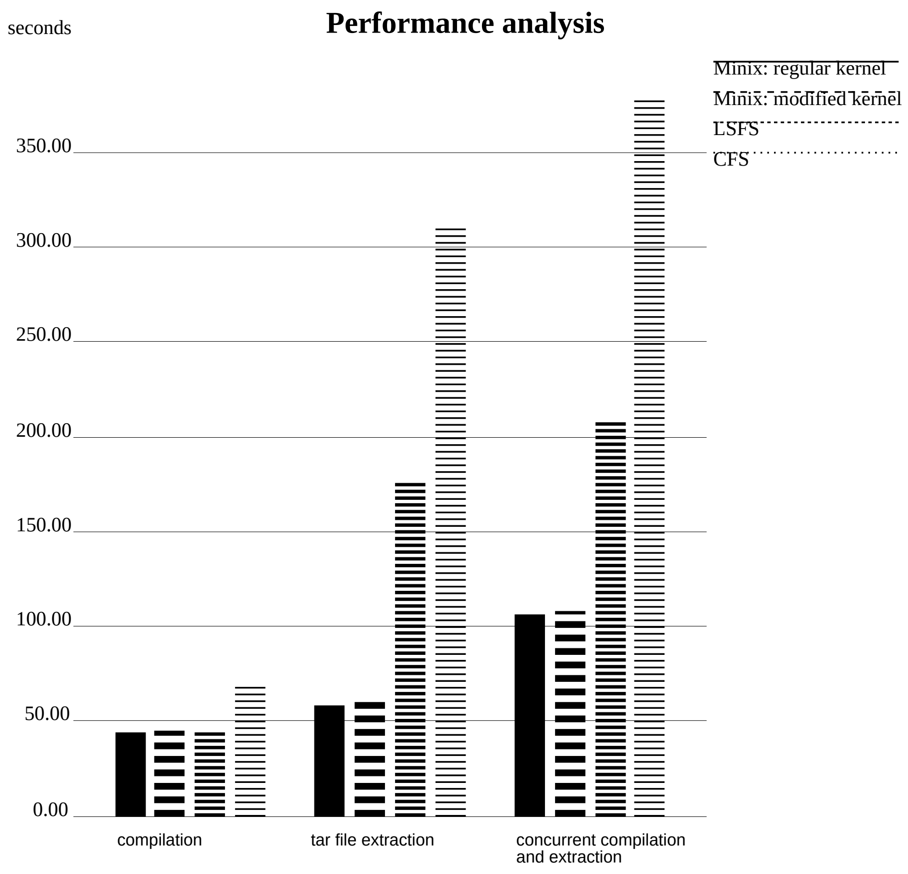


Figure 3.1: Comparison of file system performance

## 4 Conclusion

By integrating data encryption into the operating system kernel, LSFS provides thorough security for information stored on the local machine. Unlike encryption programs that run outside of the kernel, LSFS is able to use the buffer cache to minimize the performance impact of encryption. Thus, LSFS demonstrates that operating system support for encrypted file systems is a viable solution to the problem of providing increased security without decreasing convenience or performance.

# A Selected source code listings

## A.1 encrypt.c

```
/************************************************************
* encrypt.c
*
* This is the file that contains the core encryption/decryption
* routines for the extended Linux kernel.  There are two key routines
* that currently depend upon a simple DES implementation lifted
* shamelessly from CFS.  However, more sophisticated encryption
* can and should be substituted in this file at a later date.
*
* The important routines that MUST be provided to the kernel are:
*
******
* void scramble_key(key master_key, unsigned long block_no, key local_key);
*    This routine is responsible for taking a master key (constant for the
*    entire filesystem) and a block number and generating a local key
*    that will be used for encrypting/decrypting that particular block
*    on the filesystem.
******
* void encrypt_buffer(char *plaintext, char *ciphertext, key local_key, int size);
*    This routine takes a buffer "plaintext" of size "size" and encrypts it using
*    the local_key, storing the result in ciphertext.
******
* void decrypt_buffer(char *plaintext, char *ciphertext, key local_key, int size);
*    This routine needs to reverse what was done above.
************************************************************/

/* $Id: encrypt.c,v 1.8 1996/02/24 00:44:58 bkdewe Exp bkdewe $ */

#ifndef __KERNEL__
#include <stdio.h>
#endif /* __KERNEL__ */
#include <linux/fs.h>           /* Contains the definition of "key" */

#define C_BLOCK_SIZE    8       /* Size of an encryption "chunk" */
typedef char vector[C_BLOCK_SIZE];
void initVector(vector);
void xorVector(vector, vector);
void copyVector(vector, vector);

void printKey(key k);           /* Used for debugging */

/* The following routines are in cfs_des.c */
/* A more sophisticated algorithm will need to replace these calls. */
```

```
int q_block_cipher(key short_key, key text, int decrypting);
int des_block_cipher(key expanded_key, key text, int decrypting);
int des_key_setup(key _key, char *subkeys);
void key_crunch(char *buffer, char *key, int size);

void scramble_key(key master_key, unsigned long block_no, key local_key)
{
  char ex_key[128], *alias_block_no;
  int i;

  alias_block_no = (char *)&block_no;
  for(i = 0; i < KEY_SIZE; i++)
    local_key[i] = alias_block_no[i % sizeof(long)];

  /* Use the master key to encrypt the doubled-binary representation of the */
  /* block number we generated above.  The result is the local key for the */
  /* block.  This process should generate a local key that cannot be determined */
  /* without knowledge of the master key. */
  des_key_setup(master_key, ex_key);
  des_block_cipher(ex_key, local_key, 0);
}

void encrypt_buffer(char *plaintext, char *ciphertext, key local_key, int size)
{
  char ex_key[128], *cur;
  int i;
  vector iv;

  initVector(iv);
  des_key_setup(local_key, ex_key); /* Generate extended key */
  if(size % 8)
#ifdef __KERNEL__
    panic("Buffer size for filesystem encryption not a multiple of 8!\n");
#else
  {
    fprintf(stderr, "Buffer size for filesystem encryption not a multiple of 8!\n");
    abort();
  }
#endif

  memcpy(ciphertext, plaintext, size);
  cur = ciphertext;
  for(i = 0; i < size / 8; i++)
    {
      xorVector(cur, iv);
      des_block_cipher(ex_key, cur, 0);
      xorVector(iv, cur);
      cur += 8;
    }
}

void decrypt_buffer(char *plaintext, char *ciphertext, key local_key, int size)
{
  char ex_key[128], *cur;
  int i;
  vector iv, old;

  initVector(iv);
  des_key_setup(local_key, ex_key); /* Generate extended key */
  if(size % 8)
#ifdef __KERNEL__
    panic("Buffer size for filesystem encryption not a multiple of 8!\n");
#else
  {
    fprintf(stderr, "Buffer size for filesystem encryption not a multiple of 8!\n");
    abort();
  }
#endif

  memcpy(plaintext, ciphertext, size);
  cur = plaintext;
  for(i = 0; i < size / 8; i++)
    {
      copyVector(old, cur);
      des_block_cipher(ex_key, cur, 1);
      xorVector(cur, iv);
      xorVector(iv, old);
      cur += 8;
    }
}

void
initVector(vector v)
{
  int i;
  for(i = 0; i < C_BLOCK_SIZE; i++)
    v[i] = '\0';
}

void
xorVector(vector new, vector base)
{
  int i;
  for(i = 0; i < C_BLOCK_SIZE; i++)
    new[i] ^= base[i];
}

void
copyVector(vector dest, vector src)
{
  int i;
  for(i = 0; i < C_BLOCK_SIZE; i++)
    dest[i] = src[i];
}
```

## A.2 make_request()

```
static void make_request(int major,int rw, struct buffer_head * bh)
{
        unsigned int sector, count;
        struct request * req;
        int rw_ahead, max_req;

        /* ... */

/* If we're going to be doing an encrypted data transfer, set that up before
 * looking for the free request.
 */

        if((bh->b_encrypted) && (bh->b_blocknr > 1))
          {
            unlock_buffer(bh);
            if(rw == READ)
              {
                bh->b_encrypted_read = 1; /* Flag this buffer */
                mark_buffer_clean(bh);    /* Mark us as clean */
                bh = bh->b_cipher;        /* Read into this block */
                bh->b_list = BUF_CLEAN;   /* Prevents refiling... */
              }
            else
              {
                encrypt_buffer(bh->b_data, bh->b_cipher->b_data,
                              bh->b_local_key, bh->b_size);
                mark_buffer_clean(bh);
                bh = bh->b_cipher;        /* Write from this block */
                bh->b_list = BUF_CLEAN;   /* Prevents refiling... */
              }
            lock_buffer(bh);
          }

        if(!bh)
          panic("make_request:Requesting null buffer!\n");
/* look for a free request. */

        /* ... */

/* fill up the request-info, and add it to the queue */
        req->cmd = rw;
        req->errors = 0;
        req->sector = sector;
        req->nr_sectors = count;
        req->current_nr_sectors = count;

        req->bh = bh;
        req->bhtail = bh;
        req->buffer = bh->b_data;
        req->sem = NULL;
        req->next = NULL;
        add_request(major+blk_dev,req);
}
```

## A.3 getblk()

```
void prepare_encrypted_buffer(struct buffer_head *bh)
{
  /* This is an encrypted filesystem! */
  if(!bh->b_encrypted)
    {
      bh->b_encrypted = ENC_PLAINTEXT;
      scramble_key(sb->s_master_key, block, bh->b_local_key);
      if(!bh->b_cipher)
        {
          /* Setup bh->b_cipher, the data transfer
           * buffer that holds the ciphertext.
           */
          bh->b_cipher = getcipherblk(isize, size);
          bh->b_cipher->b_encrypted = ENC_CIPHERTEXT;
          bh->b_cipher->b_dev = bh->b_dev;
          bh->b_cipher->b_blocknr = bh->b_blocknr;
          bh->b_cipher->b_size = bh->b_size;
          bh->b_cipher->b_count = bh->b_count;
          bh->b_cipher->b_dirt = bh->b_dirt;
          bh->b_cipher->b_lock = bh->b_lock;
          bh->b_cipher->b_uptodate = bh->b_uptodate;
          bh->b_cipher->b_flushtime = bh->b_flushtime;
          bh->b_cipher->b_req = bh->b_req;
          bh->b_cipher->b_cipher = bh;
        }
    }
  else if(bh->b_encrypted == ENC_CIPHERTEXT)
    /* If we get here, it means we found a ciphertext
     * buffer in the queues.  THIS SHOULD NEVER HAPPEN.
     * Stop here as a sanity check.
     */
    panic("getblk working with ENC_CIPHERTEXT buffer!");
}

struct buffer_head * getblk(dev_t dev, int block, int size)
{
        struct buffer_head * bh;
        int isize = BUFSIZE_INDEX(size);
        struct super_block *sb;

        sb = get_super_nowait(dev);

        /* Update this for the buffer size lav. */
        buffer_usage[isize]++;

        /* If there are too many dirty buffers, we wake up the update process
           now so as to ensure that there are still clean buffers available
           for user processes to use (and dirty) */
repeat:
        bh = get_hash_table(dev, block, size);
        if (bh) {
                if (bh->b_uptodate && !bh->b_dirt)
                         put_last_lru(bh);
                if(!bh->b_dirt) bh->b_flushtime = 0;
                /* But before we can continue, we need to see if this is an encrypting */
                /* filesystem buffer, and if so, mark it. */
                if(sb && sb->s_encrypted)
                {
                        prepare_encrypted_buffer(bh);
                }
                else
                {
                        bh->b_encrypted = 0;
                        /* Note:  bh could point to a block that used to */
                        /* be encrypted but now isn't.  We need to check on */
                        /* the b_cipher pointer and get rid of it if it */
                        /* exists. */
                        if(bh->b_cipher)
                        {
                                /* Break connection... */
                                bh->b_cipher->b_cipher = NULL;
                                bh->b_cipher->b_dev = 0xffff;
                                bh->b_cipher->b_encrypted = 0;
                                put_last_free(bh->b_cipher);
                                bh->b_cipher = NULL;
                        }
                }
                bh->b_encrypted_read = 0; /* By default... */
                return bh;
        }

        while(!free_list[isize])
        {
                refill_freelist(size);
        }

        if (find_buffer(dev,block,size))
                 goto repeat;

        bh = free_list[isize];
        remove_from_free_list(bh);

/* OK, FINALLY we know that this buffer is the only one of its kind, */
/* and that it's unused (b_count=0), unlocked (b_lock=0), and clean */
        bh->b_count=1;
        bh->b_dirt=0;
        bh->b_lock=0;
        bh->b_uptodate=0;
        bh->b_flushtime = 0;
        bh->b_req=0;
        bh->b_dev=dev;
        bh->b_blocknr=block;
        insert_into_queues(bh);
/* But before we can continue, we need to see if this is an encrypting */
/* filesystem buffer, and if so, mark it. */
/* Code is essentially duplicated from above... */
                if(sb && sb->s_encrypted)
                {
                        prepare_encrypted_buffer(bh);
                } else bh->b_encrypted = 0;
                bh->b_encrypted_read = 0; /* By default... */
        return bh;
}
```

## A.4 wait_on_buffer()

```
extern inline void wait_on_buffer(struct buffer_head * bh)
{
        if (bh->b_lock)
                __wait_on_buffer(bh);
        if (bh->b_cipher)                    /* If we're encrypted... */
          if(bh->b_cipher->b_lock)           /* We must check b_cipher's lock, too. */
            __wait_on_buffer(bh->b_cipher);
        if (bh->b_encrypted_read)            /* If we were reading... */
          {
            bh->b_encrypted_read = 0;
            decrypt_buffer(bh->b_data,       /* Decrypt the data from the disk. */
                           bh->b_cipher->b_data,
                           bh->b_local_key,
                           bh->b_size);
            bh->b_uptodate = 1;
          }
}
```

## A.5 lsfs_read_super()

```
struct super_block *lsfs_read_super(struct super_block *s,void *data,
                                     int silent)
{
        struct buffer_head *bh;
        struct lsfs_super_block *ms;
        int i,dev=s->s_dev,block;
        extern void invalidate_buffers(dev_t);
        key validate;

        if (32 != sizeof (struct lsfs_inode))
                panic("bad i-node size");
        if(!data)
          {
            printk("LSFS-fs: passed NULL pass key\n");
            return NULL;
          }
        MOD_INC_USE_COUNT;
        lock_super(s);
        set_blocksize(dev, BLOCK_SIZE);
        printk("Attempting to load LSFS superblock\n");
        if (!(bh = bread(dev,1,BLOCK_SIZE))) {
                s->s_dev=0;
                unlock_super(s);
                printk("LSFS-fs: unable to read superblock\n");
                MOD_DEC_USE_COUNT;
                return NULL;
}
ms = (struct lsfs_super_block *) bh->b_data;
/* Note: data must hold the user's pass key */
decrypt_buffer(validate, ms->s_validate, data, KEY_SIZE);
if(strcmp(validate, "bkdewe!"))
  {
    s->s_dev=0;
    unlock_super(s);
    printk("LSFS-fs: Invalid pass key.\n");
    MOD_DEC_USE_COUNT;
    return NULL;
  }
s->s_encrypted = 1;
/* By storing the decrypted masterkey in s_master_key,
 * the buffer cache layer will know to encrypt/decrypt blocks
 * from this device.
 */
decrypt_buffer(s->s_master_key, ms->s_masterkey, data, KEY_SIZE);

s->u.lsfs_sb.s_ms = ms;
s->u.lsfs_sb.s_sbh = bh;
s->u.lsfs_sb.s_mount_state = ms->s_state;
s->s_blocksize = 1024;
s->s_blocksize_bits = 10;
s->u.lsfs_sb.s_ninodes = ms->s_ninodes;
s->u.lsfs_sb.s_nzones = ms->s_nzones;
s->u.lsfs_sb.s_imap_blocks = ms->s_imap_blocks;
s->u.lsfs_sb.s_zmap_blocks = ms->s_zmap_blocks;
s->u.lsfs_sb.s_firstdatazone = ms->s_firstdatazone;
s->u.lsfs_sb.s_log_zone_size = ms->s_log_zone_size;
s->u.lsfs_sb.s_max_size = ms->s_max_size;
s->s_magic = ms->s_magic;
if (s->s_magic == LSFS_SUPER_MAGIC) {
        s->u.lsfs_sb.s_dirsize = 16;
        s->u.lsfs_sb.s_namelen = 14;
} else if (s->s_magic == LSFS_SUPER_MAGIC2) {
        s->u.lsfs_sb.s_dirsize = 32;
        s->u.lsfs_sb.s_namelen = 30;
} else {
        s->s_dev = 0;
        unlock_super(s);
        brelse(bh);
        if (!silent)
                printk("VFS: Can't find a lsfs filesystem on dev 0x%04x.\n", dev);
        MOD_DEC_USE_COUNT;
        return NULL;
}
```

```
/* Invalidate any buffers still in memory */
invalidate_buffers(s->s_dev);
for (i=0;i < LSFS_I_MAP_SLOTS;i++)
        s->u.lsfs_sb.s_imap[i] = NULL;
for (i=0;i < LSFS_Z_MAP_SLOTS;i++)
        s->u.lsfs_sb.s_zmap[i] = NULL;
block=2;
printk("LSFS superblock successfully read!\n");
for (i=0 ; i < s->u.lsfs_sb.s_imap_blocks ; i++)
        if ((s->u.lsfs_sb.s_imap[i]=bread(dev,block,BLOCK_SIZE)) != NULL)
                block++;
        else
                break;
for (i=0 ; i < s->u.lsfs_sb.s_zmap_blocks ; i++)
        if ((s->u.lsfs_sb.s_zmap[i]=bread(dev,block,BLOCK_SIZE)) != NULL)
                block++;
        else
                break;
if (block != 2+s->u.lsfs_sb.s_imap_blocks+s->u.lsfs_sb.s_zmap_blocks) {
        for(i=0;i<LSFS_I_MAP_SLOTS;i++)
                brelse(s->u.lsfs_sb.s_imap[i]);
        for(i=0;i<LSFS_Z_MAP_SLOTS;i++)
                brelse(s->u.lsfs_sb.s_zmap[i]);
        s->s_dev=0;
        unlock_super(s);
        brelse(bh);
        printk("LSFS-fs: bad superblock or unable to read bitmaps\n");
        MOD_DEC_USE_COUNT;
        return NULL;
}
set_bit(0,s->u.lsfs_sb.s_imap[0]->b_data);
set_bit(0,s->u.lsfs_sb.s_zmap[0]->b_data);
unlock_super(s);
/* set up enough so that it can read an inode */
s->s_dev = dev;
s->s_op = &lsfs_sops;
s->s_mounted = iget(s,LSFS_ROOT_INO);
if (!s->s_mounted) {
        s->s_dev = 0;
        brelse(bh);
        printk("LSFS-fs: get root inode failed\n");
        MOD_DEC_USE_COUNT;
        return NULL;
}
if (!(s->s_flags & MS_RDONLY)) {
        ms->s_state &= ~LSFS_VALID_FS;
        mark_buffer_dirty(bh, 1);
```

```
                s->s_dirt = 1;
        }
        if (!(s->u.lsfs_sb.s_mount_state & LSFS_VALID_FS))
                printk ("LSFS-fs: mounting unchecked file system, "
                        "running fsck is recommended.\n");
        else if (s->u.lsfs_sb.s_mount_state & LSFS_ERROR_FS)
                printk ("LSFS-fs: mounting file system with errors, "
                        "running fsck is recommended.\n");
        return s;
}
```

# Bibliography

[1] Data encryption standard. FIPS PUB 46, National Bureau of Standards, Washington, DC, January 1977.

[2] DES modes of operation. FIPS PUB 81, National Bureau of Standards, Washington, DC, December 1980.

[3] Matt Blaze. A cryptographic file system for Unix. Pre-print of a paper presented at the First ACM Converence on Communications and Computing Security, Fairfax, VA, November 3–5, 1993. file://research.att.com/dist/mab/cfs.ps.

[4] R. Rivest. The MD5 message-digest algorithm. RFC 1321, April 1992.