Design and Implementation of Secure File System

Thesis submitted in partial fulfillment
for the award of the degree of

Master of Technology

in

Computer Science and Engineering

by

Rajesh Kumar Pal

Under the supervision of

Dr. Indranil Sengupta

Dept. of Computer Science and Engineering
Indian Institute of Technology
Kharagpur - 721 302, INDIA
May 2008

Dedicated to my wife Sunita

Certificate

This is to certify that the thesis entitled "**Design and Implementation of Secure File System**" submitted by **Rajesh Kumar Pal** for the award of the degree of **Master of Technology (M.Tech.)** is a bonafide research work carried out by him under my supervision and guidance during the period 2006-2008. To the best of my knowledge, the results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma. In my opinion, this thesis is of the standard required for the award of the degree of M.Tech.

Dr. Indranil Sengupta
Professor
Dept. of Computer Science and Engineering
Indian Institute of Technology
Kharagpur -721 302, INDIA

Acknowledgments

Writing this part of thesis is probably the most difficult task. Although the list of people to thank heartily is long, making this list is not the hard part. The difficult part is to search the words that convey the sincerity and magnitude of my gratitude and love. People unknown to me till a few years before; have become indispensable in my life, while the people already known to me, remain pillars of support and encouragement through these long years. It is because of their help and support that I am here now.

First of all, I would like to name my mentor, Prof. Indranil Sengupta, to whom I am sincerely indebted. His creative input and constructive challenges inspired me a lot. I have learnt to lead a systematic and disciplined life from sir, which has helped to build my personality to the fullest. His brilliant guidance catalyzed my research work, and inspired me to face every difficult and challenging aspects pertaining to my studies. His dedication towards our work is inexplicable. If I continue to write instances of his dedication, I shall run out of pages. I would like to thank Indian Air Force for providing me this opportunity to do my masters from this prestigious institute.

My warm thanks are due to all my colleagues of IIT Kharagpur for giving me the feeling of being at home in work. Talking, sharing and discussing with them made my research very interesting. Space precludes a detailing of every contribution, but I am especially grateful to all my friends and well wishers.

I would also like to thank my father and mother for their support and love for me and their blind faith in me. Finally I would like to say thanks to my wife, Sunita the one person to whom I owe this degree most to. I would like to thank her for her patience, her understanding, her encouragement, her kindness and her ever growing love and trust.

Rajesh Kumar Pal

Abstract

Governments, military, financial institutions, hospitals, and private businesses amass a great deal of confidential information about their employees, customers, products, research, and financial status. Most of this information is now collected, processed and stored on electronic computers and transmitted across networks to other computers. Should confidential information about a businesses customers or finances or new product line fall into the hands of a competitor, such a breach of security could lead to lost business, law suits or even bankruptcy of the business. Protecting confidential information is a business requirement, and in many cases also an ethical and legal requirement. For the individual, information security has a significant effect on Privacy, which is viewed very differently in different cultures.

In computer systems the information is stored traditionally in form of files. File is considered as a basic entity for keeping the information. In unix-like systems, the concept of file is so important that almost all input/output devices are considered as a file. Therefore the problem of securing data or information on computer systems can be defined as the problem of securing file data. We all will agree that in today's world, *securing file data is very important.*

There are various approaches available to ensure file data security, such as encryption tools like 'aescrypt' in linux or integrated encryption application software or disk encrypter. But each one has its own inherent disadvantages, rendering them less frequently used. These approaches are generally cumbersome and inconvenient to the users. Therefore, there is a need for a mechanism/system which can ensure reliable and efficient file data security in a transparent and convenient manner. We have taken this as a challenge and

tried to solve the problem of file data security by integrating our proposed Secure File System into the kernel itself.

The Secure File System (SFS), we have designed, provides file data security using cryptographic techniques in a transparent and convenient way. The proposed SFS pushes encryption services into the Linux kernel space, mounting it between the Virtual File System layer and underlying file system. SFS requires that the user creates a directory and name it with the prefix 'ecrypt' to store the encrypted file data, such as ecryptdir. Any directory on the system with the prefix 'ecrypt' will basically tells the system that the newly created directory will contain encrypted data. All files destined to be saved on this directory will be transparently encrypted on the fly without any user intervention. SFS provides this security infrastructure to all the applications transparently as SFS embedded into kernel enables it to provide security as one of its basic functionality. It means irrespective of the application from where data is being saved it will be encrypted thus security is enforced. SFS is fully compatible with all underlying storage file systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The spread of Internet is increasing by each passing day, enabling people all over the world to share variety of resources. We can say that Internet has become the interface of the world to us. Many people are using Internet for sharing information, searching information, gathering skills, communication, social networking, air/train reservation, entertainment and in fact all the tasks of life are now linked to the internet. Computer systems have penetrated in our lives in such a way that its difficult to imagine our life without them. But this, increased dependence and usage of computer systems has also led to an increased security attacks on our personal information, financial data, financial transactions and our privacy. Here, by security attack we mean any action that compromises the security of information. Beside the individuals, there are many organisations and government bodies providing essential services to the citizen are also being hacked, penetrated into, and their sensitive or classified information are being compromised. Figure 1.1 shows Computer (Information) security incidents (or attacks) handled by CERT-in in the year 2007. Indian computer Emergency Response Team (CERT-in) is a national initiative (*under Ministry of Communications and Information Technology*) to tackle emerging challenges in the area of information security and country level security risks and vulnerabilities. We have been witnessing that many financial institutes and banks are also under increased security attacks. The situation is equally bad on the defence

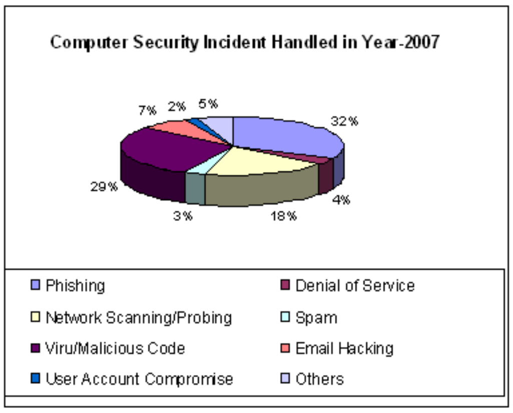


Figure 1.1: Computer security attack in year 2007

systems or military networks, and it is of grave concern given the fact that some of the information compromised can severely hamper our national interest and can give undue advantage to our enemies. I am being an officer from the Indian Air Force, aware of the ways our adversaries are using to gather the classified information and use it to their advantage and bring disadvantage to us. Due to these prevaling scenarios, most of the classified information is kept on removable drives under lock and key, and the internal networks or defence intranets are never been connected to the internet. This shows how low is our confidence on computer systems.

1.1 Background

The nature of risks and vulnerabilities in modern societies is becoming more and more transformational today. An open and innovative approach to newer vulnerabilities at the physical, virtual and psychological levels is needed to create new knowledge and a better understanding of new risks and of their causes, interactions, probabilities and costs.

Information security has therefore assumed a greater importance in today's world. The primary aspects of information security are confidentiality, integrity and availability. These three factors are important not only to business sector but also to the Government and critical infrastructure such as Power, Telecommunications, Transportation, Energy, Banking & Finance and Defence, etc. Information security has become essential part of the day-to-day functioning of these sectors.

Securing information and systems against the full spectrum of threats requires the use of multiple, overlapping protection approaches addressing the people, technology, and operational aspects of information technology. This is due to the highly interactive nature of the various systems and networks, and the fact that any single system cannot be adequately secured unless all interconnecting systems are also secured. By using multiple, overlapping protection approaches, the failure or circumvention of any individual protection approach will not leave the system unprotected. Through user training and awareness, well-crafted policies and procedures, and redundancy of protection mechanisms, layered protections enables effective protection of information technology for the purpose of achieving mission objectives.

1.1.1 Security Principles

Information security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction. The Principles provide general guidance to establish and maintain the security of information. Figure 1.2 highlights the core principles of Information security are Confidentiality, Integrity and Availability also, known as CIA Triad.

These important principles are described below:-

1. **Confidentiality**
   Confidentiality is the characteristic of information being disclosed only to authorized persons, entities, and processes at authorized times and in the authorized manner. Information that is considered to be confidential in nature must only be accessed, used, copied, or disclosed by persons who have been authorized to do so, and only when there is a genuine need to do so. A breach of confidentiality occurs when

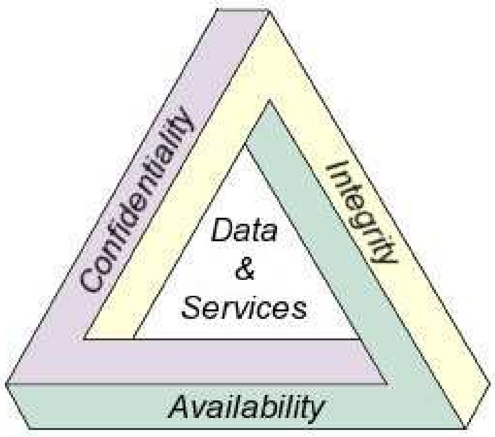


Figure 1.2: CIA Triad

information that is considered to be confidential in nature has been, or may have been, accessed, used, copied, or disclosed to, or by, someone who was not authorized to have access to the information.

2. **Integrity**
   Integrity is the characteristic of information being accurate and complete and the information systems preservation of accuracy and completeness. In information security, integrity means that data can not be created, changed, or deleted without authorization. It also means that data stored in one part of a database system is in agreement with other related data stored in another part of the database system (or another system).

3. **Availability**
   Availability is the characteristic of information and supporting information systems being accessible and usable on a timely basis in the required manner. The concept of availability means that the information, the computing systems used to process the information, and the security controls used to protect the information are all available and functioning correctly when the information is needed. The opposite of availability is denial of service (DOS).

1.1.2 Cryptography as a Security Tool

Cryptography is used to constraint the potential senders and/or receivers of the message. Modern cryptography [20] is based on secrets called keys that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key - the key is source of the message. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message, so that the key becomes the destination.

Encryption is a means for constraining the possible receivers of a message. An encryption algorithm enables the sender of a message to ensure that only a computer possessing a certain key can read the message. In a symmetric encryption algorithm, the same key is used to encrypt and to decrypt. That is, E(k) can be derived from D(k), and vice versa. Therefore, the secrecy of E(k) must be protected to the same extent as that of D(k).The common symmetric encryption algorithm used are data encryption standard (DES), 3DES, twofish, RC5, RC4 and advanced encryption standard (AES). AES can use key lengths of 128, 192, 256 bits and work on 128-bit block. It works by performing 10 to 14 rounds of transformations on a matrix formed from a block. Generally, the algorithm is compact and efficient. In an asymmetric encryption algorithm, there are different encryption and decryption keys. The RSA cipher is a block-cipher public key algorithm and is most widely used asymmetric algorithm. Asymmetric cryptography is based on mathematical functions rather than transformations, making it much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Asymmetric algorithms are mostly used for encryption of small amounts of data, authentication, confidentiality, and key distribution.

Authentication offers a way of constraining the set of potential senders of a message. Authentication is thus complementary to encryption. There are main two variety of authentication algorithm. The first type of authentication algorithm uses symmetric encryption. In a message -authenticated code (MAC), a cryptographic checksum is generated from the message using a secret key. The second main type of authentication algorithm is a digital-signature algorithm, and the authenticators thus produced are called digital signatures. In a digital-signature algorithm, it is computationally infeasible to derive $S(k_s)$ from $V(k_v)$; in particular, V is a one-way function. Thus, Kv is the public key and Ks is the private key. The most widely authentication algorithms are MD5 and SHA.

1.1.3 Defense in Depth

Information security must protect information through out the life span of the information, from the initial creation of the information on through to the final disposal of the information. The information must be protected while in motion and while at rest. During its life time, information may pass through many different information processing systems and through many different parts of information processing systems. There are many different ways the information and information systems can be threatened. To fully protect the information during its lifetime, each component of the information processing system must have its own protection mechanisms. The building up, layering on and overlapping of security measures is called *defense in depth*. The strength of any system is no greater than its weakest link. Using a defence in depth strategy, should one defensive measure fail there are other defensive measures in place that continue to provide protection.

1.2 Security provided by Operating Systems

Operating system (OS) is the most essential part of a computer system, which provides functionality like process management, memory management, storage management, protection and security. In order to enhance the system security, the most common approach used by OS to authenticate a user identity is the use of passwords. UNIX like systems uses encryption to store these passwords securely. The system contains a function that is extremely difficult to invert but is simple to compute. This function is used to encode all passwords and only encoded passwords are stored. When a user presents a password, it is encoded and compared against the stored encoded password. Besides user authentication, OS also provides security by making a clear distinction between kernel space and user space and by enforcing the access-permissions set on the resources.

A modern operating system provides access to a number of resources, which are available to software running on the system, and to external devices like networks via the kernel. Internal security, or security from an already running program is only possible if all possibly harmful requests must be carried out through interrupts to the operating system kernel. If programs can directly access hardware and resources, they cannot be secured. Microsoft Windows has been heavily criticized for many years for window's almost total inability to protect one running program from another, however since windows isn't generally used as a server it has been considered less of a problem. In recent years, Microsoft has added limited user accounts, and more secure logins. However most people still operate their computers using Administrator accounts, which negates any possible internal security improvements brought about by these changes.

Linux and UNIX both have two tier security, which limits any system-wide changes to the root user, a special user account on all UNIX-like systems. While the root user has unlimited permission to affect system changes, programs as a regular user are limited only in where they can save files, and what hardware they can access. This limits the damage that a regular user can do to the computer while still providing them with plenty of freedom to do everything but affect system-wide changes. The user's settings are stored in an area of the computer's file system called the user's home directory, which is also provided as a location where the user may store their work, similar to 'My Documents' on a windows system. Should a user have to install software or make system-wide changes, they must enter the root password for the computer, which allows them to launch certain programs as the root user.

While users generally find regular user accounts on Linux installations provide plenty of freedom for day to day activities, the need to enter a password to install software has generated criticisms from many Windows users who are used to being able to change, delete, create, and rename files anywhere on the system at whim, while also making it extremely easy to accidentally delete important files, and for viruses to infect the operating system.

At the operating system level, there are a number of software firewalls available, as well as intrusion detection/prevention systems. Most modern operating systems include a software firewall, which is enabled by default. A software firewall can be configured to allow or deny network traffic to or from a service or application running on the operating system. Therefore, one can install and be running an insecure service, such as Telnet or FTP, and not have to be threatened by a security breach because the firewall would deny all traffic trying to connect to the service on that port.

However, OS doesn't have a robust mechanism for file data security. Our proposed SFS is designed to provide the file data security as a functionality of Linux kernel using the cryptographic tools.

1.3 File data Security

In computer systems the information is stored traditionally in form of files. File is considered as a basic entity for keeping the information. In unix-like systems, the concept of file is so important that almost all input/output devices are considered as a file. Therefore the problem of securing data or information on computer systems can be defined as the problem of securing file data. We all will agree that in today's world, *securing file data* is very important.

There are have been different approaches used, to solve the file data security problem. Most of the solutions provided works in user space. The simple and naive approach used by many people to secure their file data is to use common utilities like crypt or aescrypt. These utilities take the filename of the file needs to be encrypted and the password as inputs and produces the encrypted file. This type of utility is good for limited use only, as it is very cumbersome and manual. The user needs to provide the key(or password) for each file he is encrypting and he also needs to remember it, in order to retrieve the data back. For a large number of files this can be very tasking thus prone to error. The user may also need to delete the plain file from the disk manually, so that there is only one copy of file and that too in encrypted form. Due to all this it is very inconvenient and seldom used. Second approach is integrating encryption engine in application software itself, where each program that is to manipulate sensitive data has built-in cryptographic facilities. For example, a text editor could ask for a key when a file is opened and automatically encrypt and decrypt the file's data as they are written and read. But the disadvantage here is all application should use the same encryption engine and any change in one will require changes in all. This is a good approach but it has practical difficulties like integrating the encryption engine with all types of application/utility may be difficult. Also any upgradation in application version/major change will need reintegration. These aspects restricts its wide spread use. The third approach is to use commercially available disk controllers with embedded encryption hardware that can be used to encipher entire disks or individual file blocks with a specified bock. It suffers from the fact that key needs to be shared among users, whose data reside on the disk because entire disk is protected as a single entity. It is good for single user system but for multi-user system the key protecting the data needs to be shared between different user. Sharing the key among different users may not be desired in certain cases. The fourth approach is called systems approach where the encryption facility is incorporated at systems level.

So we have seen that each one of the approaches described above; has its own inherent disadvantages, rendering them less frequently used. These approaches are generally cumbersome and inconvenient to the users. Therefore, there is a need for a mechanism/system which can ensure reliable and efficient file data security in a transparent and convenient manner. We have taken this as a challenge and tried to solve the problem of file data security. We have considered various places where this mechanism/system can be placed to fulfill its requirement in the best possible way. The considered places include user space, device layer level, and kernel space. We are of the opinion that the *file data security should be provided as a functionality of operating system*[1], therefore we have decided to push the encryption services into the Linux kernel space mounted beneath the virtual file system. The case study of CFS [4] and TCFS [6] has also been a motivation for the same.

1.4 Contribution of the Thesis

This thesis addresses the core problem of *File data Security* on the computer systems. We have tried to solve this problem by extending the Operating system (Linux) functionality, by integrating our Secure File System (SFS). This section outlines the Secure File System presented in the thesis.

1.4.1 Design of Secure File System (SFS)

The Operating systems provide several important functionalities like process management, memory management, storage management and provide user interface. They also provide many security related functionality, but it lacks a mechanism for file data security. Our Secure File System once integrated with the Operating system, extends the OS such that it provides File data Security as one of its basic inherent functionality. This approach is very efficient, convenient and user friendly. SFS has been mounted beneath the Virtual File System (VFS) layer; therefore it works perfectly fine on all underlying storage file systems without any changes. SFS encrypts clear text file data, on the fly using advanced encryption standard (AES) algorithm and then stores this encrypted data onto the disk. The file is to be encrypted or not, is decided automatically by the SFS based on the selection of the directory where the file will be stored. The file access is controlled using public key cryptography.

1.4.2 Implementation of SFS in Linux Kernel

We have implemented Secure File System on Linux based system. This has been implemented in kernel space. In addition to this, it has some user space utility to facilitate user friendliness and convenience. We were required to implemented two new system calls and few kernel modules, which have been described later.

1.5 Thesis Layout

The thesis is organized as follows:

- **Chapter 1** introduces the main focus of the thesis. It provides a brief background of the work and also lists the major contributions.
- **Chapter 2** This chapter gives a detail literature survey on the previous work done in the field of data security and cryptographic file systems. It also explains the Virtual File System, which is important to understand the work done.
- **Chapter 3** This chapter describes design goals, architecture of the proposed Secure file System, system operation and its usage.
- **Chapter 4** This chapter covers the implementation details.
- **Chapter 5** This chapter gives the functional and performance evaluation of the scheme. We have also discussed the test results.
- Finally **Chapter 6** concludes this work and presents some useful extensions.

Chapter 2

Background and Related Work

2.1 Overview of related work

There has been lot of development [4, 23, 14] taken place since the time when MS DOS device driver [8] was used to encrypt the entire partition. Now a days we have several cryptographic file systems available, which we have briefly described.

2.1.1 MSDOS SFS

SFS is an MSDOS device driver that encrypts an entire partition. Once encrypted, the driver presents a decrypted view of the encrypted data. This provides the convenient abstraction of a file system, but relying on MSDOS is not secure because MSDOS provides none of the protection of a modern OS.

2.1.2 Cryptographic File System (CFS)

Cryptographic File System was developed by Matt Blaze, to provide a transparent UNIX file system interface to directory hierarchies that are automatically encrypted with user supplied keys [4]. CFS is implemented as a user-level NFS server. User needs to create an encrypted directory and assign its key required for cryptographic transformations, when the directory is created for the first time. In order to use an encrypted directory, CFS daemon requires the user to attach the encrypted directory to a special directory /crypt. This attach basically creates a mapping between the encrypted directory and mount point (directory) in the /crypt. This way the actual encrypted data resides in the encrypted directory and the mapping provides a window to access these encrypted file in clear text form to the authenticated user. CFS uses DES to encrypt file data.

The CFS prototype is implemented entirely at user level, communicating with the Unix kernel via the NFS interface. Each client machine runs a special NFS server, cfsd (CFS Daemon), on its localhost interface, that interprets CFS file system requests. cfsd is implemented as an RPC server for an extended version of the NFS protocol. At boot time, the system invokes cfsd and issues an NFS mount of its localhost interface on the CFS directory (/crypt) to start CFS. To allow the client to also work as a regular NFS server, CFS runs on a different port number from standard NFS. The NFS protocol is designed for remote file servers, and so assumes that the file system is very loosely coupled to the client (even though, in CFS's case, they are actually the same machine). The client kernel communicates with the file system through 17 remote procedure calls (RPCs) that implement various file system-related primitives (read, write, etc.). The server is stateless, in that it is not required to maintain any state data between individual client calls. All communication is initiated by the client, and the server can simply process each RPC as it is received and then wait for the next.

CFS is a well designed, portable file system that provides transparent access semantics. Its main disadvantage is that it runs in user mode, thus requires many context switches and data copies from user space to kernel space. A modification over this scheme using smartcard has been given by Itoi in [12].

2.1.3 Transparent Cryptographic File System (TCFS)

TCFS works as a layer under the VFS (Virtual File system) layer, making it completely transparent to the applications [6]. The security is guaranteed by means of the DES (data encryption standard) algorithm. TCFS is implemented as a NFS distributed file system. The TCFS daemon handles the RPC generated by the kernel. RPC relative to read, write etc have been extended to perform security operations. Each time a new file handler is created, the extended attribute 'secure' is tested. If the file is secure, then all successive read and write operation will be filtered through the encryption/decryption layer. In TCFS for file encryption, each user is associated with a file system key and all files of a user are encrypted using this key. This key is encrypted with user's password and is stored in a database in /etc/tcfspwdb. This dependability of user key on login password is one of the major disadvantages of TCFS. Also we are of the opinion that storing encryption key on the same disk containing data reduces security.

2.1.4 BestCrypt

BestCrypt is designed as a loopback device driver which creates a raw block device with a single file [2]. This single file acts as the backing store and thus called container. Each container has an associated cipher key, which will be used in cryptographic transformations. This device can be formatted with any available file system. BestCrypt requires the user to create and mount the container same as if user is mounting a regular block device. This commercially available software is good for single user environment where creation and use of container is limited to one individual. In multi-user environment the users will need to share the cipher associated with the container with the system administrator as some functionality requires superuser access.

2.1.5 Self-certifying File System

The Self-certifying File System addresses the issue of key management in cryptographic filesystems and proposes separating key management from file system security [13, 9].

Servers have a public key and clients use the server public key to authenticate the server and establish a secure communication channel. To allows clients to authenticate servers on the spot without even having heard of them before, it introduces the concept of a self-certifying pathname. A self-certifying pathname contains the hash of the public-key of the server, so that the client can verify that he is actually taking to the legitimate server. Once the client has verified the server the server a secure channel is established and the actual file access takes place.

2.1.6 Encryption File System (EFS)

Microsoft Windows uses EFS which is based on Windows authentication methods and Windows ACLs [18]. EFS stores the encryption keys on disk in a lockbox that is encrypted using the user's login password. It has the obvious disadvantage that each time the users change their login password they also need to re-encrypt the lockbox. If an administrator changes the user's password, then all encrypted files become unreadable.

2.1.7 StegFS

StegFS is a file system that uses steganographic and cryptographic techniques [17]. It is implemented as a modified EXT2 kernel driver that keeps a separate block-allocation table per security level. If the adversaries inspect the system, then they only know that there is some hidden data. They do not know the contents or extent of what is hidden. Data is replicated randomly throughout the disk to avoid loss of data when disk is mounted with an unmodified EXT2 driver and random blocks may be overwritten. Although StegFS achieves strong deniability of data's existence, the performance degradation is very high, making it impractical for most application.

2.2 Virtual File System (VFS)

Linux's key to success is its ability to coexist comfortably with other systems. Linux manages to support multiple filesystem (including those of unix, windows) through a concept called the Virtual Filesystem (VFS) [19].

2.2.1 Role and Structure of VFS

The idea behind the Virtual Filesystem is to put a wide range of information in the kernel to represent many different types of filesystems; there is a field or function to support each operation provided by all real filesystems supported by Linux [5]. For each read, write, or other function called, the kernel substitutes the actual function that supports a native Linux filesystem, the NTFS filesystem, or whatever other filesystem the file is on.

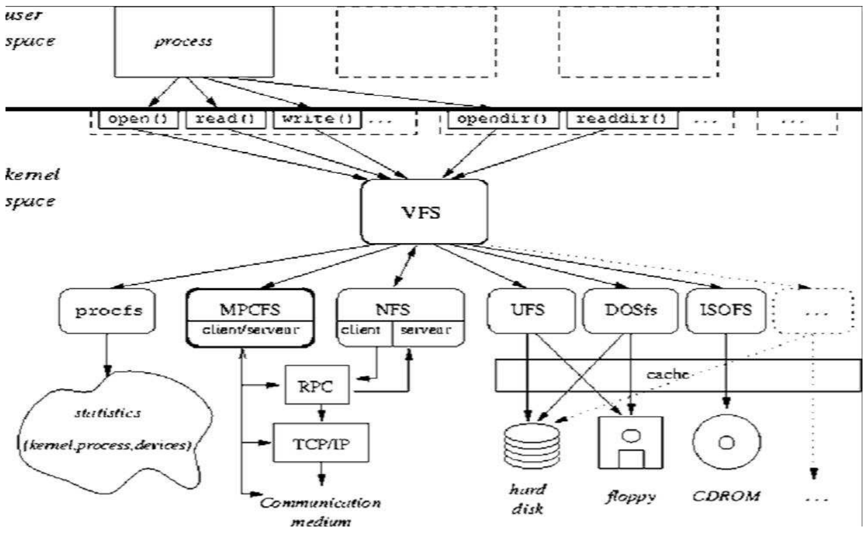


Figure 2.1: Structure of VFS

The structure of Virtual Filesystem is represented by figure 2.1. The Virtual Filesystem (also known as Virtual Filesystem Switch or VFS) is a kernel software layer that handles all system calls related to a standard Unix filesystem. Its main strength is providing a common interface to several kinds of filesystems.

2.2.2 Filesystems supported by VFS

Filesystems supported bythe VFS may be grouped into three main classes:

- **Disk-based File Systems**

  Local disk (HDD, ODD) or Device that emulates disk (ex. USB flash memory)

  Linux: Ext2 (Second Extended Filesystem), Ext3, ReiserFS

  Unix: sysv (System V, Coherent, Xenix), UFS (BSD, Solaris, NEXTSTEP), VERITAS VxFS (SCO UnixWare)

  MS: MS-DOS, VFAT, NTFS

  ISO9660: CD-ROM filesystem, UDF (Universal Disk Format) DVD filesystem

- **Network File Systems** Easy access to files included in other network computers NFS, Coda, AFS (Andrew Filesystem), CIFS (MS Common Internet File System), NCP (Novell's NetWare Core Protocol).

- **Special filesystems** Do not manage real disk space purely virtual /proc.

2.2.3 Common File Model of VFS

The key idea behind the VFS consists of introducing a common file model capable of representing all supported filesystems. It requires that each specific filesystem implementation must translate its physical organization into the VFS's common file model. The file model consists of the following object types:

- **The superblock object**
  Stores information concerning a mounted filesystem. For disk-based filesystems, this object usually corresponds to s *filesystem control block* stored on disk.

- **The inode object**
  Stores general information about a specific file. For disk-based filesystems, this object usually corresponds to a *file control block* stored on disk. Each inode object is associated with an *inode number*, which uniquely identifies the file within the filesystem.

- **The file object**
  Stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file open.

- **The dentry object**
  Stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. Each disk-based filesystem stores this information in its own particular way on disk.

Chapter 3

Secure File System

3.1 Design Goals

We have designed Secure File System (SFS) with the objective that file data security should be provided as one of the primary functionality of the kernel. We have extended the kernel to provide file data security using cryptographic techniques as one of its functionality. The encryption/decryption of file data is performed transparently, making it convenient for the users. The proposed SFS is designed with the following primary objectives:

- **Security**: Confidentiality of data is ensured by use of strong encryption. The files are encrypted on the fly and then saved to the disk or send onto the network. Clear-text data are never stored into the directory prefixed with the word 'ecrypt'. The system uses random symmetric key dynamically generated by the system to encrypt the file data. Therefore it is possible to use difficult-to-guess keys of sufficiently long length.

- **Strong Access Control**: We have also used public-key cryptographic techniques, to control the access of the file. This approach enhances the security of file by avoiding unwarranted access. We will be using smart cards to store the private-key of the users, thus effectively the data and the key are stored on two different physical locations.
- **Transparent Performance**: Encrypted files should behave no different from some other files. The only difference is that the encrypted files are useless without the knowledge of right key. In fact the interface should be uniform that users don't realize that they are working on secure file. Encryption/decryption must be performed transparently without much of user's intervention.
- **Convenience**: The system should be convenient to users; otherwise it will not be used. SFS provides uniform interface to the users except some initial directory creation and set up of access control file.

3.2 Design of proposed Secure File System

The proposed Secure File System is designed to provide the above mentioned goals. Figure 3.1 shows the normal flow of control in standard file system. We have already explained the virtual file system in previous chapter. VFS has namely two main function. First, to handle the file system related system calls like open, close, read, write etc. Secondly it provides a uniform interface to actual filesystems like ext2, ext3, FAT, etc, by acting as a switch.

What we have done is, we have taken the control flow from VFS layer based on some condition and reroute it to our proposed SFS. The condition we are checking is, the location were the file is destined to be saved. If the location is the directory starting with prefix word 'ecrypt' e.g, *ecrypt*dir then we take the control flow to SFS layer. Figure 3.2 shows SFS layer is mounted beneath the VFS and interacts closely with it. VFS and

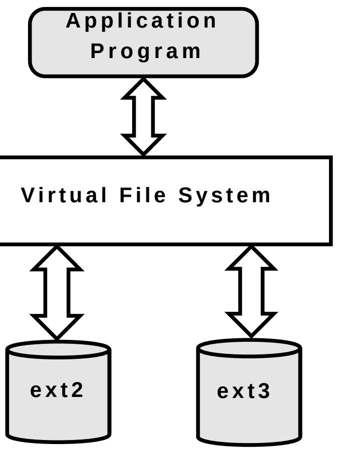


Figure 3.1: Control Flow in Standard Filesystem

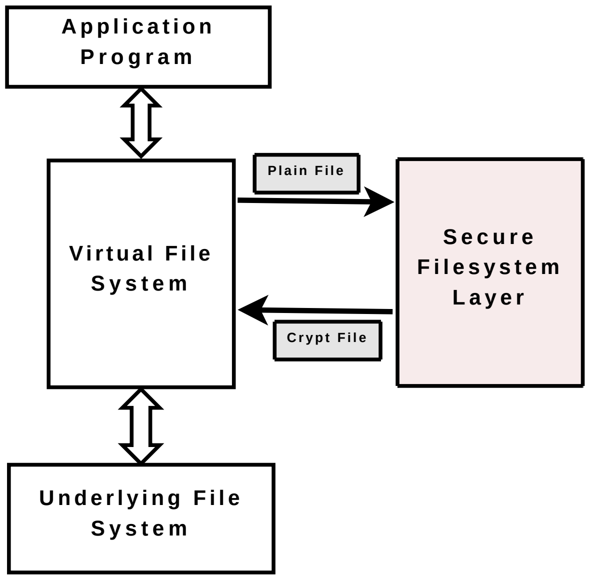


Figure 3.2: Control transferred to Secure File System which provides the encryption services

proposed SFS functions in kernel space, therefore a user cannot access them directly. The above condition will be checked and executed by kernel.

After study of kernel filesystem structure and related work [11, 22, 14], we have found that mounting SFS beneath VFS is the idle place because then we can efficiently use the kernel infrastructure and deviate only were its required. In this way, it has also one major advantage that it provides uniform interface with all the application and the underlying file system. This means SFS transparently handles the data without being bothered of, from which application the data is coming. Therefore SFS is compatible and works with all the applications. The user can use the cryptographic strength provided by SFS with any and every application. This way it provides a very user friendly and convenient environment for user to work on. Similarly, SFS is compatible with all types of filesystem because, after SFS strengthens the data with cryptography, it passes the data to VFS. Therefore this design decision effective utilizes the kernel infrastructure in providing uniform interface and user convenience.

We have used the directory name checking condition to decide whether the control should be diverted to SFS layer. This way it poses a restriction that all directories which contains encrypted files have common prefix 'ecrypt'. This introduces some weakness as an attacker can easily know what all directories are containing secret information and where they are located, but this provides a user convenience. This is just a mechanism we have used, which can be strengthened further either by providing some utility to create directories which will contain secret data.

3.2.1 Architecture of Secure File System

Figure 3.3 shows the architecture of Secure File System in detail. SFS mainly has three components:-

- Key Management Unit (KMU)

- Crypt Engine (CE)
- Access Controller (AC)
- File Header Extractor (FHE)

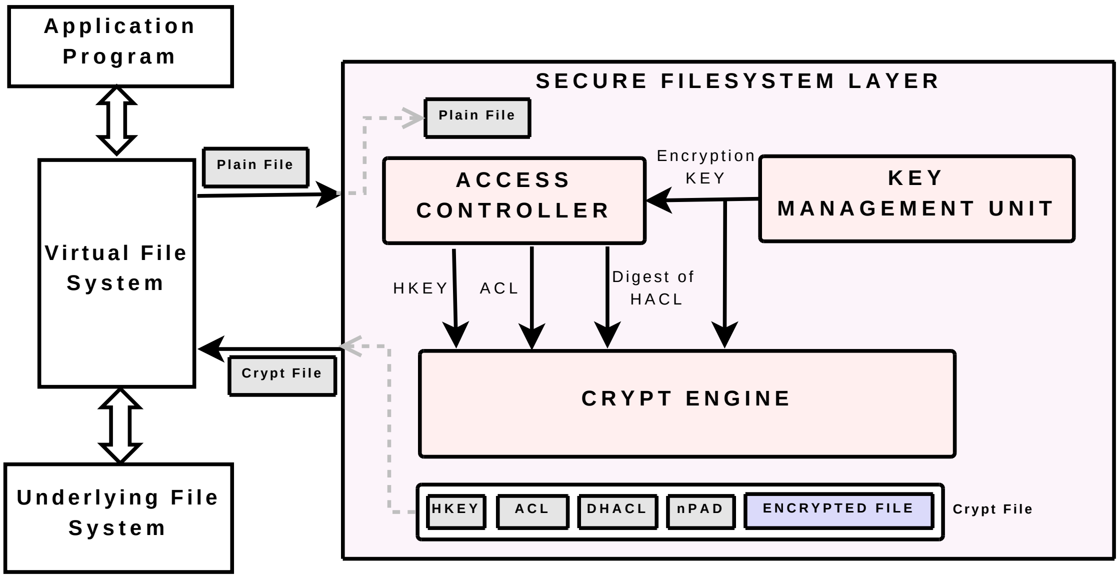


Figure 3.3: Secure File System

3.2.1.1 Key Management Unit

Key management is very crucial task and have been discussed in [3, 16, 21, 7, 10]. KMU is responsible for generating the random symmetric key used for encrypting the file data and providing it to AC and CE. We have used the kernel provided random generation function. The Linux random number generator is part of the kernel of all Linux distributions and is based on generating randomness from entropy of operating system events. The output of this generator is used for almost every security protocol, including TLS/SSL key generation, choosing TCP sequence numbers, and file system and email encryption. Entropy bits are added to the primary pool from external sources like mouse and keyboard activity, disk I/O operations, and specific interrupts. KMU generates the random symmetric key each time a file needs to be encrypted on the fly. Therefore each time the file is saved, a new random symmetric key is used. So it aids in providing strong security.

Public Key Table		
User id	**Name**	**Public Key**
11	Rajesh	A1023BF4839FDC78
22	Dipanjan	78FA73B34FFCD823
33	Alpha	8FE345CD09303ABB
44	Bravo	B7382FFF900ABC46
55	Charlie	CFDEA77635BDAB45
66	Delta	1234DDAFECC6783A

Table 3.1: Public Key table

KMU also loads the 'pubkey' table containing userid of the users, their name and their public key into the system. Table 3.1 shows the details of the 'pubkey' table. Once the table is loaded the public key of users is available to the kernel. This is a one time process and executed at system boot time. We have presumed that public key infrastructure (PKI) is available for generating public-private key pairs. Once PKI generates the pair, the public key part is populated in the table and system is made aware of the fact. For our system, we can have a server running the PKI system and any host requiring information regarding public key can query with the server.

3.2.1.2 Access Controller

AC is responsible for creation and management of all access related information with respect to a file [5, 15]. It has been mentioned that a user needs to create a directory prefixed with word 'ecrypt' which will contain the encrypted data. This 'ecrypt' prefix distinguishes the directory from the normal directories. All the directories having 'ecrypt' as a prefix will only contain encrypted files. Each such directory also contains a special file called 'accesscontrol' file which keeps the access-control related information common for all files in that directory. The creator of the directory exclusively sets and thus controls the recipients of all the files stored in that directory. SFS provides a utility named 'setaccesscontrol' using which a user needs to set the recipients of files stored in these directories. Each such directory will have only one 'accesscontrol' file. Access Controller performs the following tasks in order to control the access of a file:-

- **Preparation of Hash of symmetric key (HKEY)**
  AC creates hash of the random symmetric key (HKEY), which is used to verify the authenticity of the key.

- **Preparation of Access Control List (ACL)**
  AC reads the 'accesscontrol' file and extracts the public key of the recipients of the file. The public key of all users is available with the system after KMU loads it. AC then encrypts the random symmetric key with the public key of each recipient of the file using RSA algorithm one by one and prepares a list which we will call Access control List (ACL). ACL is basically a list of elements where each element represents the encrypted key obtained on encrypting random symmetric key with the public key of a recipient. The purpose of the ACL is to ensure that file is accessed by authorized users only. A file can be accessed by only those users whose authorization is contained in ACL in the form of an element of ACL. The figure 4.2 shows how ACL is created.

- **Preparation of HACL**
  The hash of the ACL (called as HACL) is created using SHA-256. Here we are using a secret key to generate a cryptographic checksum. We are selecting this secret key from the ACL. The purpose of HACL is to cater for integrity of ACL. The logic for this design decision is that, if we can ensure the integrity of ACL which controls the access of the file then in turn integrity of file data is also ensured.

- **Preparation of DHACL** The digest of HACL (called as DHACL) is created by using private key of the creator/owner of the file. DHACL will be used to ensure that file access related information is not tempered. Before opening (or reading) of the encrypted file, the HACL is extracted from DHACL using public key of the owner of the file. File's owners detail is stored along with file, thertfore extracting its public key is not an issue.

3.2.1.3 Crypt Engine

When the file is to be written on disk, CE receives mainly two inputs, the plain data file and the random symmetric key. It uses AES algorithm to encrypt the file and produces the encrypted file. AES is a block cipher which works on a block of 16 bytes. Therefore here we need to check the file size and if file size is not multiple of 16, we pad the file to make the file size perfect multiple of 16. This padding of file is done while it is being written. When the encrypted file is read from the disk, we first decrypt it using AES and remove the padded bytes and then passes it to the application. So application never comes to know about padded bytes.

CE also receives the HKEY, ACL, DHACL from the accesscontroller and attaches them as a file header to the encrypted file. This whole structure shown in figure 3.3, containing file header and the encrypted file will be called as *crypt file*. This crypt file is passed to the underlying file system. So to summarize, basically Crypt Engine encrypts the file data on the fly and passes it to the low level file system while writing and while reading it receives the encrypted file and decrypts the file data on the fly.

3.2.1.4 File Header Extractor (FHE)

FHE comes into play when the file data is being accessed from the disk. FHE extracts the HKEY, ACL and DHACL from the file stored on the disk. It then uses the private key of the person accessing the file to decipher the key. The private key is made available by the person holding it. It is presumed that private key is kept safely by its owner.

3.2.2 System Operation

In this section we will describe the sequence of events which take place while the file is being created or written to the disk and also while the file is being modified or read from the disk.

3.2.2.1 File creation

For working with confidential data or files the user needs to enter the secure session by entering his private key. Then he needs to create a directory with prefix 'ecrypt' e.g. ecryptdir, which will house all the files containing confidential data. After these preliminary requirements are met, the user is free to use any application to create his file. For example, say the user uses KWrite utility to create a text file. When the user is finished with the file, he needs to save the file in his newly created directory (ecryptdir) which will be housing all such files. This is an important step because SFS will get activate for all the files saved in directory starting with word 'ecrypt'. Now what happens is the save command from the application activates the system call sys_write to write the application data on the disk. As we know all filesystem related system calls will be handled by VFS, so the control reaches to the VFS. Here the location where file is being saved will be checked, and if the condition of ecryptdir (or directory with prefix 'ecrypt') is correct the control flow will be transferred to the SFS layer. As shown in the figure 3.3, the following action will take place:-

1. **Key Management Unit** will generate the Random symmetric key.
2. **Crypt Engine** will be encrypt the file data with the symmetric key (generated in step 1) using AES algorithm. It also appends the number of bytes padded in the file to make it perfect multiple of 16.
3. **Access Controller** will generate the ACL, DHACL, HKEY and append them at the begining of the encrypted file, making the crypt file.
4. **Crypt file** is saved on the disk.

3.2.2.2 File Access

For working with confidential data or files the user needs to enter the secure session by entering his private key. Now user can open the file with some application, for example

KWrite for opening or accessing the text file. The file data will be displayed to the user in plain form if he was given authorization by the owner at the time of file creation else, the file contents will not be displayed (ie, access denied). As shown in the Figure 3.4, the

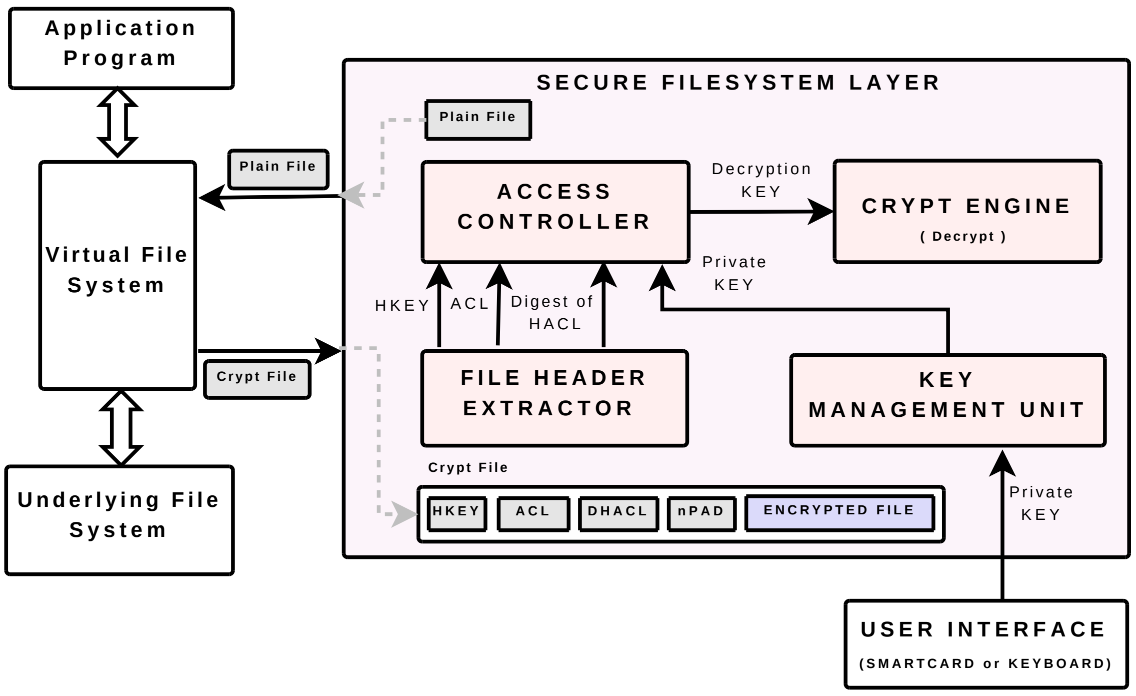


Figure 3.4: File access using Secure File System

following action takes place for file access:-

1. **File Header Extractor** extracts the file headers like HKEY, ACL, DHACL, npad and made them available to the access controller.

2. **Access Controller** will generate the HACL from ACL by using SHA-256. It will also extract HACL from DHACL by using the public key of the owner by applying RSA algorithm. Now these two HACLs will be compared, if both matches it means the ACL or access related information has not been tampered with. If they do not match then access is denied and error reported. Further each element of ACL is taken and it is decrypted with the private key of the user accessing file by applying RSA algorithm. The key obtained with it, may be the genuine key or may not.

So to verify this the hash of this extracted key is generated using SHA-256 and compared with the HKEY. If both HKEYs matches, it means we have found the right symmetric key, which was used for encrypting the file data. Now this key is passed to crypt engine.

3. **Crypt Engine** will decrypt the crypt file using key provided by access controller by applying AES. The plain file obtained is now passed on to the application.

3.3 System Usage

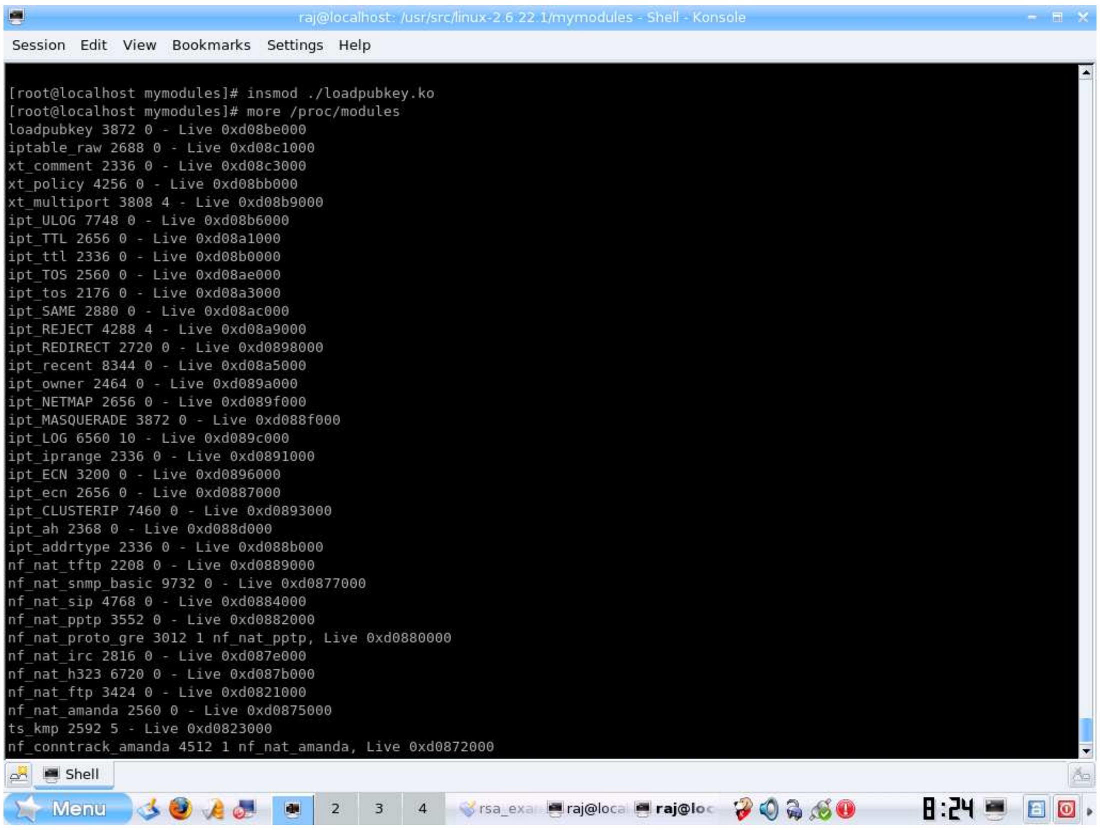
In this section we describe a typical scenario for Secure File System usage. First, the Linux kernel integrated with secure file system layer loads or boots up. The loadable module 'loadpubkey' inserts the public key table containing user-id, name of user and user's public key into the kernel. Figure 3.5 shows that 'loadpubkey' module is manually inserted and verified by checking the list of loaded modules.

Figure 3.6 shows the data loaded into the kernel by using dmesg. 'dmesg' utility is used to diplay the kernel buffer on standard output.

Next say, user Alpha enters a secure session using the *entersecuresession* utility by providing his private key.

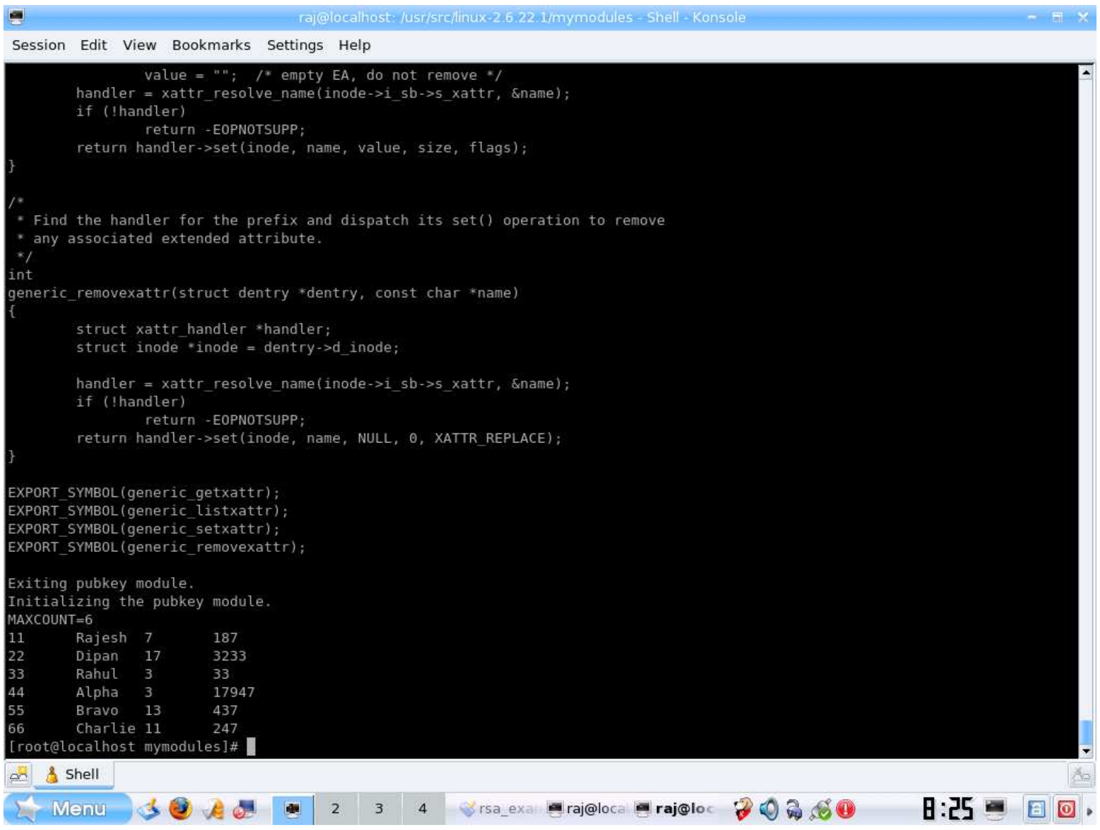
Then User Alpha creates a directory say ecryptdir (directory needs to be prefixed with word *ecrypt*) and enters into that directory. User uses the utility *setaccesscontrol* to set the access control of all the files in ecryptAlpha directory. The user needs to provide the user id of the users he wants to authorize as recipient of the files stored in this directory. This list can be modified later by the creator using the same utility.

User creates a file using any application and saves it in this directory (ecryptdir). Say user opens KWrite, create a document and save it in ecryptdir of as shown in figure 3.9 user can copy some file into ecryptdir. The file data will get transparently encrypted and its access related information will be processed and finally saved to the disk.


raj@localhost: /usr/src/linux-2.6.22.1/mymodules - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Menu

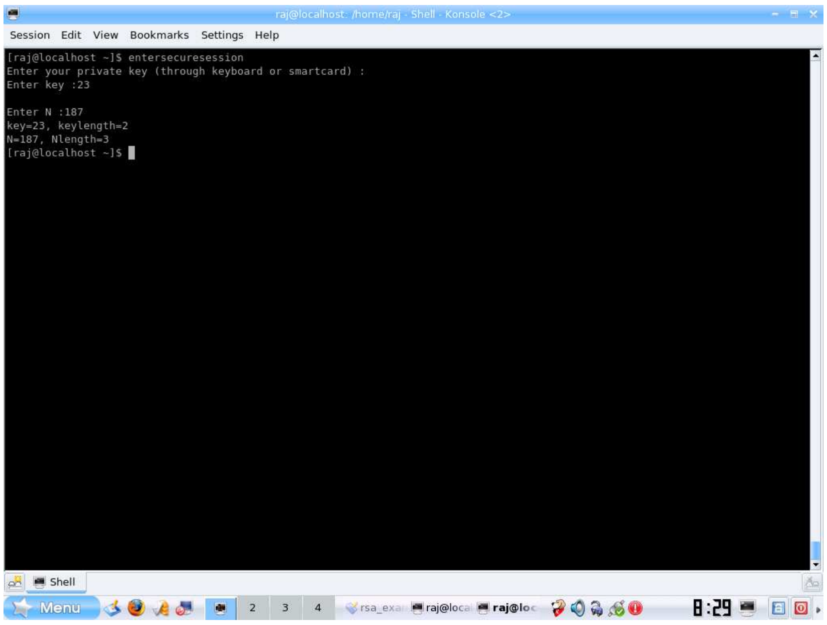
```
[root@localhost mymodules]# insmod ./loadpubkey.ko
[root@localhost mymodules]# more /proc/modules
loadpubkey 3872 0 - Live 0xd08be000
iptable_raw 2688 0 - Live 0xd08c1000
xt_comment 2336 0 - Live 0xd08c3000
xt_policy 4256 0 - Live 0xd08bb000
xt_multiport 3808 4 - Live 0xd08b9000
ipt_ULOG 7748 0 - Live 0xd08b6000
ipt_TTL 2656 0 - Live 0xd08a1000
ipt_ttl 2336 0 - Live 0xd08b0000
ipt_TOS 2560 0 - Live 0xd08ae000
ipt_tos 2176 0 - Live 0xd08a3000
ipt_SAME 2880 0 - Live 0xd08ac000
ipt_REJECT 4288 4 - Live 0xd08a9000
ipt_REDIRECT 2720 0 - Live 0xd0898000
ipt_recent 8344 0 - Live 0xd08a5000
ipt_owner 2464 0 - Live 0xd089a000
ipt_NETMAP 2656 0 - Live 0xd089f000
ipt_MASQUERADE 3872 0 - Live 0xd088f000
ipt_LOG 6560 10 - Live 0xd089c000
ipt_iprange 2336 0 - Live 0xd0891000
ipt_ECN 3200 0 - Live 0xd0896000
ipt_ecn 2656 0 - Live 0xd0887000
ipt_CLUSTERIP 7460 0 - Live 0xd0893000
ipt_ah 2368 0 - Live 0xd088d000
ipt_addrtype 2336 0 - Live 0xd088b000
nf_nat_tftp 2208 0 - Live 0xd0889000
nf_nat_snmp_basic 9732 0 - Live 0xd0877000
nf_nat_sip 4768 0 - Live 0xd0884000
nf_nat_pptp 3552 0 - Live 0xd0882000
nf_nat_proto_gre 3012 1 nf_nat_pptp, Live 0xd0880000
nf_nat_irc 2816 0 - Live 0xd087e000
nf_nat_h323 6720 0 - Live 0xd087b000
nf_nat_ftp 3424 0 - Live 0xd0821000
nf_nat_amanda 2560 0 - Live 0xd0875000
ts_kmp 2592 5 - Live 0xd0823000
nf_conntrack_amanda 4512 1 nf_nat_amanda, Live 0xd0872000
```

Figure 3.5: Load Public key table


raj@localhost: /usr/src/linux-2.6.22.1/mymodules - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Menu

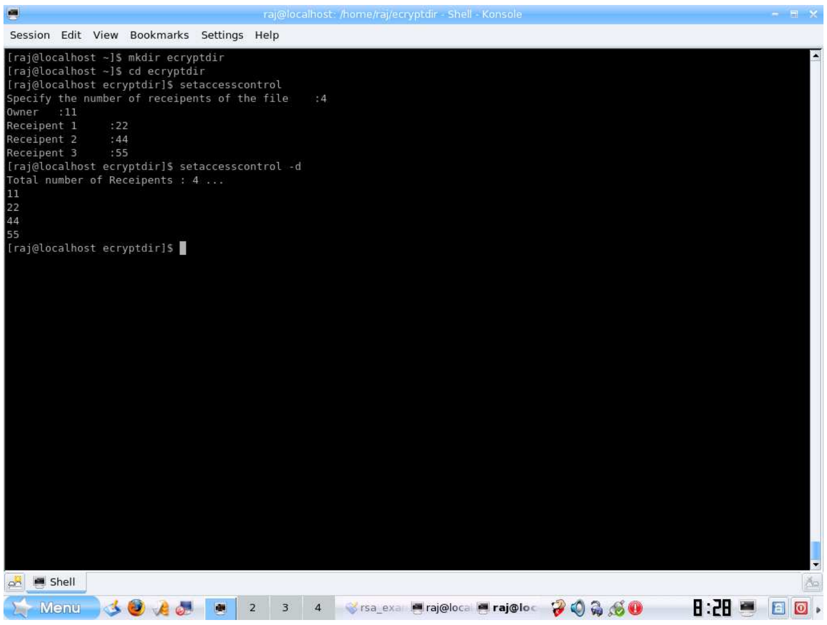
```
                value = "";  /* empty EA, do not remove */
        handler = xattr_resolve_name(inode->i_sb->s_xattr, &name);
        if (!handler)
                return -EOPNOTSUPP;
        return handler->set(inode, name, value, size, flags);
}

/*
 * Find the handler for the prefix and dispatch its set() operation to remove
 * any associated extended attribute.
 */
int
generic_removexattr(struct dentry *dentry, const char *name)
{
        struct xattr_handler *handler;
        struct inode *inode = dentry->d_inode;

        handler = xattr_resolve_name(inode->i_sb->s_xattr, &name);
        if (!handler)
                return -EOPNOTSUPP;
        return handler->set(inode, name, NULL, 0, XATTR_REPLACE);
}

EXPORT_SYMBOL(generic_getxattr);
EXPORT_SYMBOL(generic_listxattr);
EXPORT_SYMBOL(generic_setxattr);
EXPORT_SYMBOL(generic_removexattr);

Exiting pubkey module.
Initializing the pubkey module.
MAXCOUNT=6
11      Rajesh  7       187
22      Dipan   17      3233
33      Rahul   3       33
44      Alpha   3       17947
55      Bravo   13      437
66      Charlie 11      247
[root@localhost mymodules]#
```

Figure 3.6: Contents of Public key table


raj@localhost: /home/raj - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Menu

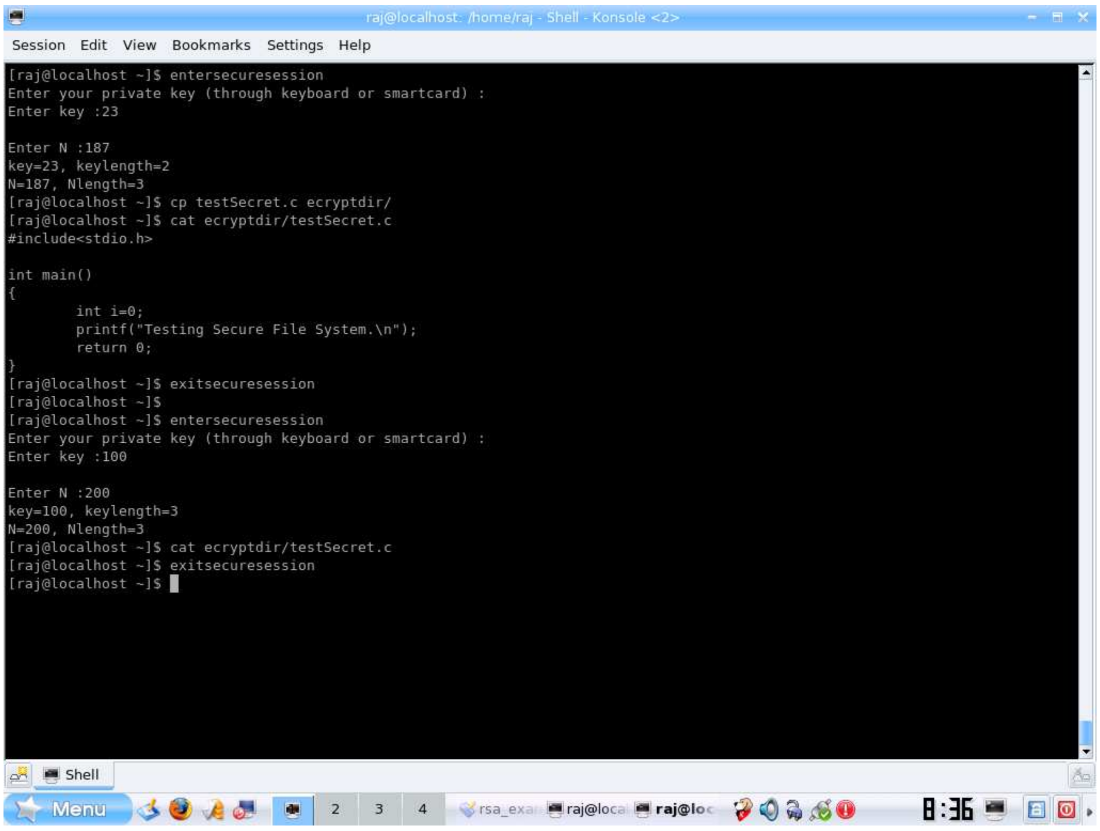
```
[raj@localhost ~]$ entersecuresession
Enter your private key (through keyboard or smartcard) :
Enter key :23

Enter N :187
key=23, keylength=2
N=187, Nlength=3
[raj@localhost ~]$
```

Figure 3.7: The user enters the secure session for performing operations on secret files


raj@localhost: /home/raj/ecryptdir - Shell - Konsole
Session Edit View Bookmarks Settings Help
Shell
Menu

```
[raj@localhost ~]$ mkdir ecryptdir
[raj@localhost ~]$ cd ecryptdir
[raj@localhost ecryptdir]$ setaccesscontrol
Specify the number of receipents of the file    :4
Owner   :11
Receipent 1     :22
Receipent 2     :44
Receipent 3     :55
[raj@localhost ecryptdir]$ setaccesscontrol -d
Total number of Receipents : 4 ...
11
22
44
55
[raj@localhost ecryptdir]$
```

Figure 3.8: The user grants the access rights by using setaccesscontrol utility



```
[raj@localhost ~]$ entersecuresession
Enter your private key (through keyboard or smartcard) :
Enter key :23

Enter N :187
key=23, keylength=2
N=187, Nlength=3
[raj@localhost ~]$ cp testSecret.c ecryptdir/
[raj@localhost ~]$ cat ecryptdir/testSecret.c
#include<stdio.h>

int main()
{
        int i=0;
        printf("Testing Secure File System.\n");
        return 0;
}
[raj@localhost ~]$ exitsecuresession
[raj@localhost ~]$
[raj@localhost ~]$ entersecuresession
Enter your private key (through keyboard or smartcard) :
Enter key :100

Enter N :200
key=100, keylength=3
N=200, Nlength=3
[raj@localhost ~]$ cat ecryptdir/testSecret.c
[raj@localhost ~]$ exitsecuresession
[raj@localhost ~]$
```

raj@localhost: /home/raj - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Menu
2 3 4
rsa_exa
raj@loca
raj@loc
8:36

Figure 3.9: The user creates a file using any application and saves it into *ecrypt*dir

When the user is finished, he can use *exitsecuresession* utility to exit the secure mode of operation. Figure 3.9 also depicts that if some unauthorized user (or somebody enters wrong private key, other than the authenticated one) enters the system, he cannot access the data. The system will display nothing in the application.

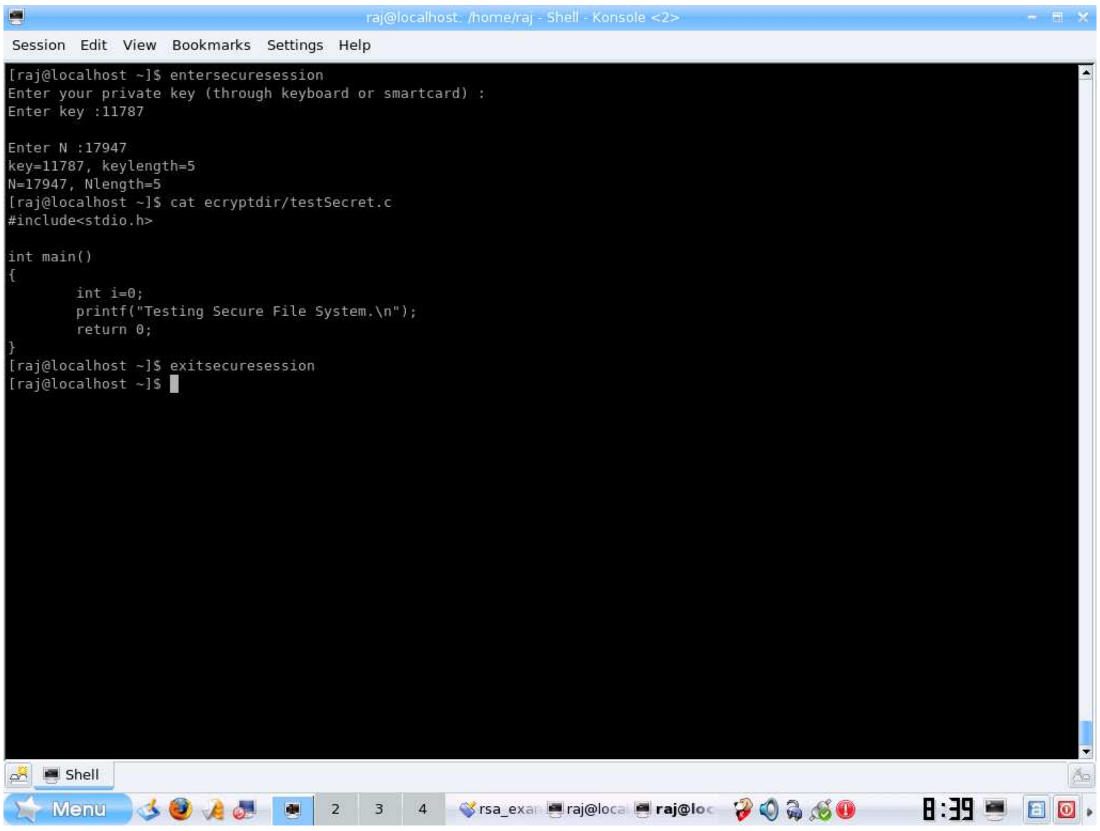
Now, assuming that the file is transferred from user alpha to user Bravo (*authorized* user ie, he was granted access rights by the owner or creator of file) and user Bravo saves that file in /home/bravo/ecryptbravo. For user Bravo to access the file he needs to have the same kernel integrated with secure file system running at his end, and follow the following sequence:-

- Use *entersecuresession* utility to access the secret data
- Open the file in desired application

Now, user can go to the application directly and open up the file as if it is a normal file. Say Bravo uses KWrite application to open and see the content or 'cat' utility to display the file content on the screen. When the application tries to read the encrypted file data, the private key of Bravo is used to check whether he has been granted access for the file or not. If user Bravo is not given access, the file access will be denied, otherwise the file will be decrypted transparently on-the-fly and its clear-text form will be displayed. Once the user Bravo is finished he can use *exitsecuresession* to exit the secure session.

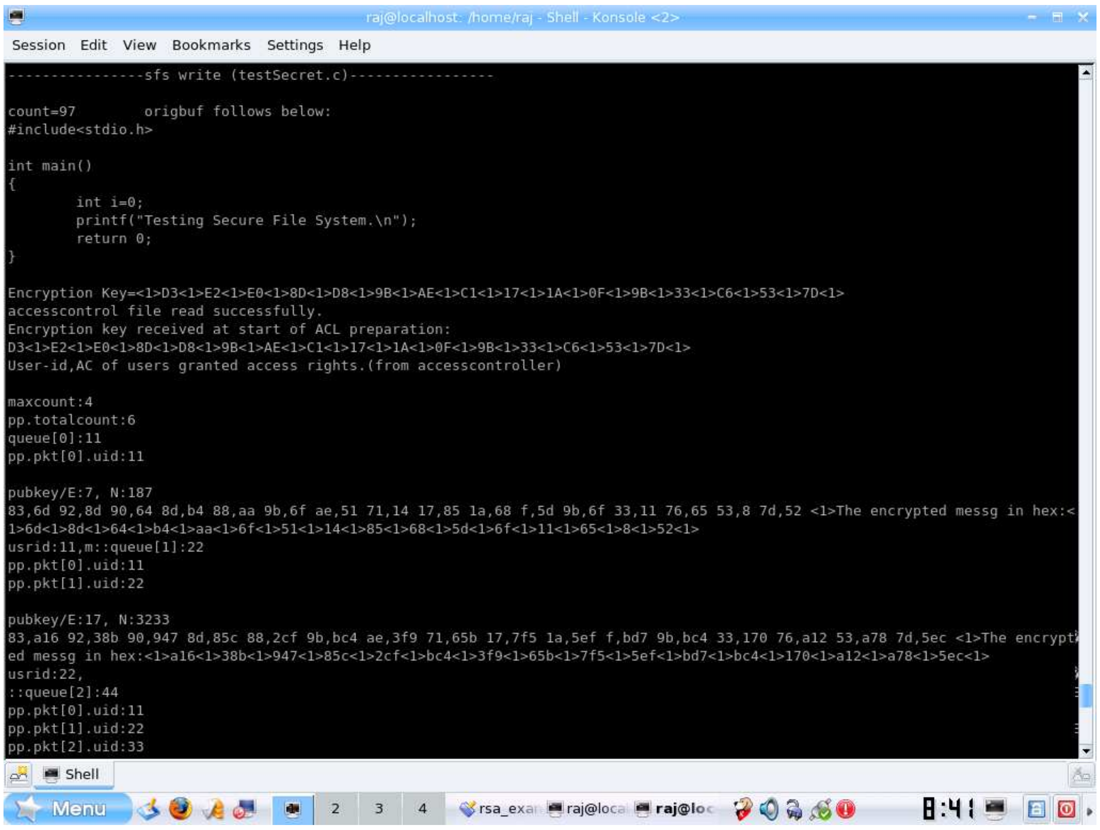
The size of the encrypted files will be increased as it is accommodating several file headers and padding of file data which is required sometime to meet the encryption block size. The file names are currently not encrypted to facilitate better readability and convenience.

Figure 3.11 shows the contents of a file 'testSecret.c' undergoing through SFS layer and data from different stages are printed on screen. The figure also shows the no of bytes of the file and the original buffer's content containing file data. Figure 3.12 shows the cryptbuf buffer's content after file passes through SFS. cryptbuf's content are encrypted


raj@localhost: /home/raj - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Menu

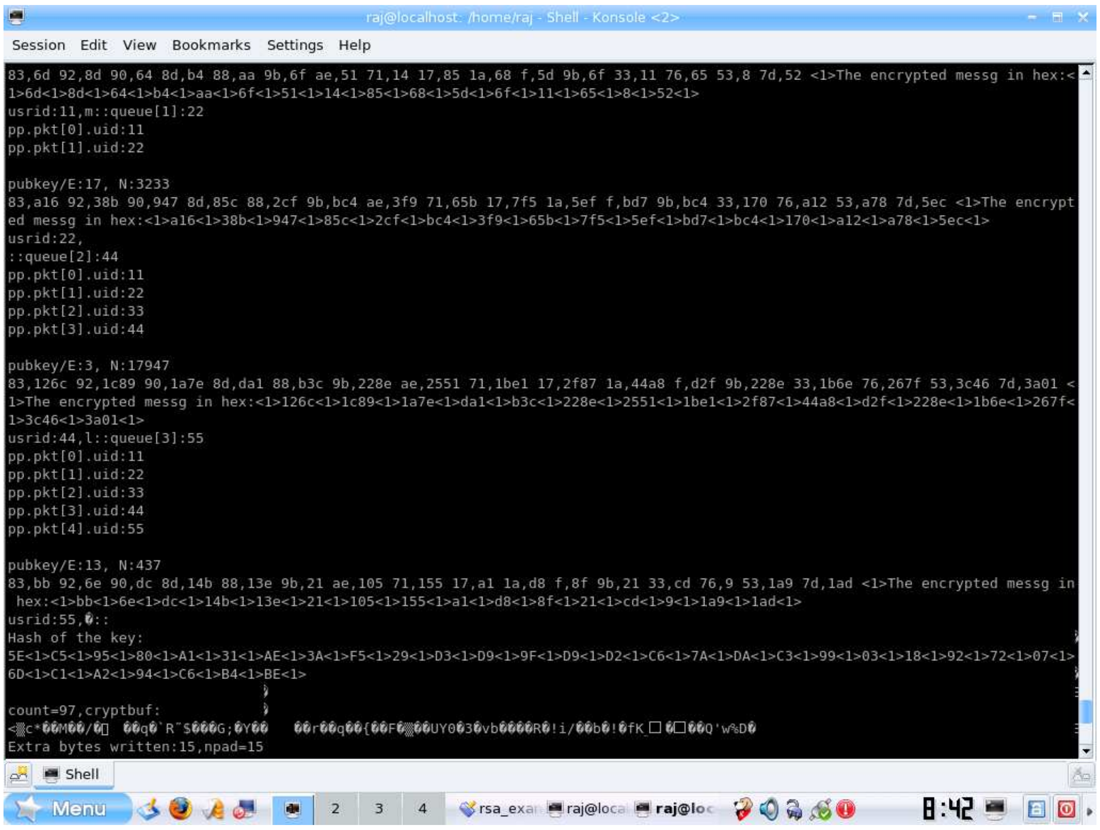
```
[raj@localhost ~]$ entersecuresession
Enter your private key (through keyboard or smartcard) :
Enter key :11787

Enter N :17947
key=11787, keylength=5
N=17947, Nlength=5
[raj@localhost ~]$ cat ecryptdir/testSecret.c
#include<stdio.h>

int main()
{
        int i=0;
        printf("Testing Secure File System.\n");
        return 0;
}
[raj@localhost ~]$ exitsecuresession
[raj@localhost ~]$
```

Figure 3.10: File access by an authorized user


raj@localhost: /home/raj - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
Shell
Menu

```
----------------sfs write (testSecret.c)-----------------

count=97        origbuf follows below:
#include<stdio.h>

int main()
{
        int i=0;
        printf("Testing Secure File System.\n");
        return 0;
}

Encryption Key=<1>D3<1>E2<1>E0<1>8D<1>D8<1>9B<1>AE<1>C1<1>17<1>1A<1>0F<1>9B<1>33<1>C6<1>53<1>7D<1>
accesscontrol file read successfully.
Encryption key received at start of ACL preparation:
D3<1>E2<1>E0<1>8D<1>D8<1>9B<1>AE<1>C1<1>17<1>1A<1>0F<1>9B<1>33<1>C6<1>53<1>7D<1>
User-id,AC of users granted access rights.(from accesscontroller)

maxcount:4
pp.totalcount:6
queue[0]:11
pp.pkt[0].uid:11

pubkey/E:7, N:187
83,6d 92,8d 90,64 8d,b4 88,aa 9b,6f ae,51 71,14 17,85 1a,68 f,5d 9b,6f 33,11 76,65 53,8 7d,52 <1>The encrypted messg in hex:<
1>6d<1>8d<1>64<1>b4<1>aa<1>6f<1>51<1>14<1>85<1>68<1>5d<1>6f<1>11<1>65<1>8<1>52<1>
usrid:11,m::queue[1]:22
pp.pkt[0].uid:11
pp.pkt[1].uid:22

pubkey/E:17, N:3233
83,a16 92,38b 90,947 8d,85c 88,2cf 9b,bc4 ae,3f9 71,65b 17,7f5 1a,5ef f,bd7 9b,bc4 33,170 76,a12 53,a78 7d,5ec <1>The encrypt
ed messg in hex:<1>a16<1>38b<1>947<1>85c<1>2cf<1>bc4<1>3f9<1>65b<1>7f5<1>5ef<1>bd7<1>bc4<1>170<1>a12<1>a78<1>5ec<1>
usrid:22,
::queue[2]:44
pp.pkt[0].uid:11
pp.pkt[1].uid:22
pp.pkt[2].uid:33
```

Figure 3.11: Contents of file while passing through SFS



```
83,6d 92,8d 90,64 8d,b4 88,aa 9b,6f ae,51 71,14 17,85 1a,68 f,5d 9b,6f 33,11 76,65 53,8 7d,52 <1>The encrypted messg in hex:<
1>6d<1>8d<1>64<1>b4<1>aa<1>6f<1>51<1>14<1>85<1>68<1>5d<1>6f<1>11<1>65<1>8<1>52<1>
usrid:11,m::queue[1]:22
pp.pkt[0].uid:11
pp.pkt[1].uid:22

pubkey/E:17, N:3233
83,a16 92,38b 90,947 8d,85c 88,2cf 9b,bc4 ae,3f9 71,65b 17,7f5 1a,5ef f,bd7 9b,bc4 33,170 76,a12 53,a78 7d,5ec <1>The encrypt
ed messg in hex:<1>a16<1>38b<1>947<1>85c<1>2cf<1>bc4<1>3f9<1>65b<1>7f5<1>5ef<1>bd7<1>bc4<1>170<1>a12<1>a78<1>5ec<1>
usrid:22,
::queue[2]:44
pp.pkt[0].uid:11
pp.pkt[1].uid:22
pp.pkt[2].uid:33
pp.pkt[3].uid:44

pubkey/E:3, N:17947
83,126c 92,1c89 90,1a7e 8d,da1 88,b3c 9b,228e ae,2551 71,1be1 17,2f87 1a,44a8 f,d2f 9b,228e 33,1b6e 76,267f 53,3c46 7d,3a01 <
1>The encrypted messg in hex:<1>126c<1>1c89<1>1a7e<1>da1<1>b3c<1>228e<1>2551<1>1be1<1>2f87<1>44a8<1>d2f<1>228e<1>1b6e<1>267f<
1>3c46<1>3a01<1>
usrid:44,l::queue[3]:55
pp.pkt[0].uid:11
pp.pkt[1].uid:22
pp.pkt[2].uid:33
pp.pkt[3].uid:44
pp.pkt[4].uid:55

pubkey/E:13, N:437
83,bb 92,6e 90,dc 8d,14b 88,13e 9b,21 ae,105 71,155 17,a1 1a,d8 f,8f 9b,21 33,cd 76,9 53,1a9 7d,1ad <1>The encrypted messg in
 hex:<1>bb<1>6e<1>dc<1>14b<1>13e<1>21<1>105<1>155<1>a1<1>d8<1>8f<1>21<1>cd<1>9<1>1a9<1>1ad<1>
usrid:55,�::
Hash of the key:
5E<1>C5<1>95<1>80<1>A1<1>31<1>AE<1>3A<1>F5<1>29<1>D3<1>D9<1>9F<1>D9<1>D2<1>C6<1>7A<1>DA<1>C3<1>99<1>03<1>18<1>92<1>72<1>07<1>
6D<1>C1<1>A2<1>94<1>C6<1>B4<1>BE<1>

count=97,cryptbuf:
<▒c*��M��/�▯  ��q�`R˜$���G;�Y��    ��r��q��{��F�▒��UY0�3�vb����R�!i/��b�!�fK_□ �□��Q'w%D�
Extra bytes written:15,npad=15
```

Figure 3.12: Contents of crypt file while passing through SFS

therefore not readable (see second last line of figure). The figure also displays the number of bytes padded and removed.

So, we can clearly see that usage of system is very easy and user friendly. The cryptographic services are provided by the OS transparently and user is not bothered of any other issue, except he needs to save the confidential data in the directory prefixed with 'ecrypt'.

Chapter 4

Implementation of Secure File System

SFS has been implemented in kernel space. We have taken the Linux source tree version 2.6.22.1 from kernel.org and worked on it. We have mainly dealt with filesystem and virtual file system of Linux. Therefore we have first studied the code of these areas in kernel and identified the place where our SFS needs to be implemented and integrated with VFS. Our work basically deals with the implementation of SFS in Linux kernel on Intel i386 architecture, however the concept can easily be extended to other architectures.

4.1 Understanding read system call

We strongly feel that to understand our work, clear understanding of atleast read or write system call is must. Therefore we are briefly describing the read system call form systems perspective. Whenever any application reads data from a file (in turn from a disk), the application basically calls read(). When a user level program calls read(), Linux translates this to a system call, sys_read(). The code of sys_read system call, which follows below, is taken from fs\read_write.c:

```
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf,
   size_t count)
{
    struct file *file;
    ssize_t ret = -EBADF;
    int fput_needed;

    file = fget_light(fd, &fput_needed);
    if (file) {
            loff_t pos = file_pos_read(file);
            ret = vfs_read(file, buf, count, &pos);
            file_pos_write(file, pos);
            fput_light(file, fput_needed);
    }

    return ret;
}
```

sys_read() takes a file descriptor, a user-space buffer pointer, and a number of bytes to read from the file into the buffer. A file lookup is done to translate the file descriptor to a file pointer with `fget_light()`. Then `vfs_read()` is called, which does all the main work. Each `fget_light()` needs to be paired with `fput_light()`, so that is done after `vfs_read()` finishes.

The system call, `sys_read()`, has passed control to `vfs_read()`. The `vfs_read()` kernel function does following:

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count,
   loff_t *pos)
{
    ssize_t ret;
```

```

        if (!(file->f_mode & FMODE_READ))
                return -EBADF;
        if (!file->f_op || (!file->f_op->read && !file->f_op->aio_read))
                return -EINVAL;
        if (unlikely(!access_ok(VERIFY_WRITE, buf, count)))
                return -EFAULT;

        ret = rw_verify_area(READ, file, pos, count);
        if (ret >= 0) {
                count = ret;
                ret = security_file_permission (file, MAY_READ);
                if (!ret) {
                        if (file->f_op->read)
                                ret = file->f_op->read(file, buf, count, pos);
                        else
                                ret = do_sync_read(file, buf, count, pos);
                        if (ret > 0) {
                                fsnotify_access(file->f_path.dentry);
                                add_rchar(current, ret);
                        }
                        inc_syscr(current);
                }
        }

        return ret;
}
```

The first three parameters are passed via, or are translations from, the original `sys_read()` parameters. The fourth parameter is offset within **file**, where the read should start. This is could be non-zero if `vfs_read()` is called explicitly because it could be called from within the kernel. In line 259-264, basic checking is done on the file operations structure to ensure that read or asynchronous read operations have been defined. If no read operation is defined, or if the operation table is missing, the function returns EINAV error at this point. This indicates that the file descriptor is attached to a structure that cannot be used for reading.

Further in line 266-271, we verify that the area to be read is not locked and that the file is authorized to be read. If it is not, we notify the parent of the file(line 275-277). Line 272-274, shows the main work done by `vfs_read()`. If the read file operation has been defined, we call it; otherwise, we call `do_sync_read()`. Line 272 shows that the actual read file operation of the actual file system is called. ie, This corresponds to the read file operation of ext2 if the file is on the mounted file system ext2. We will restrict ourself upto explanation of vfs_read as it is important with relation to our work. However to make the control flow very clear we have used the Figure 4.1 to show the top down traversal of `read()`.

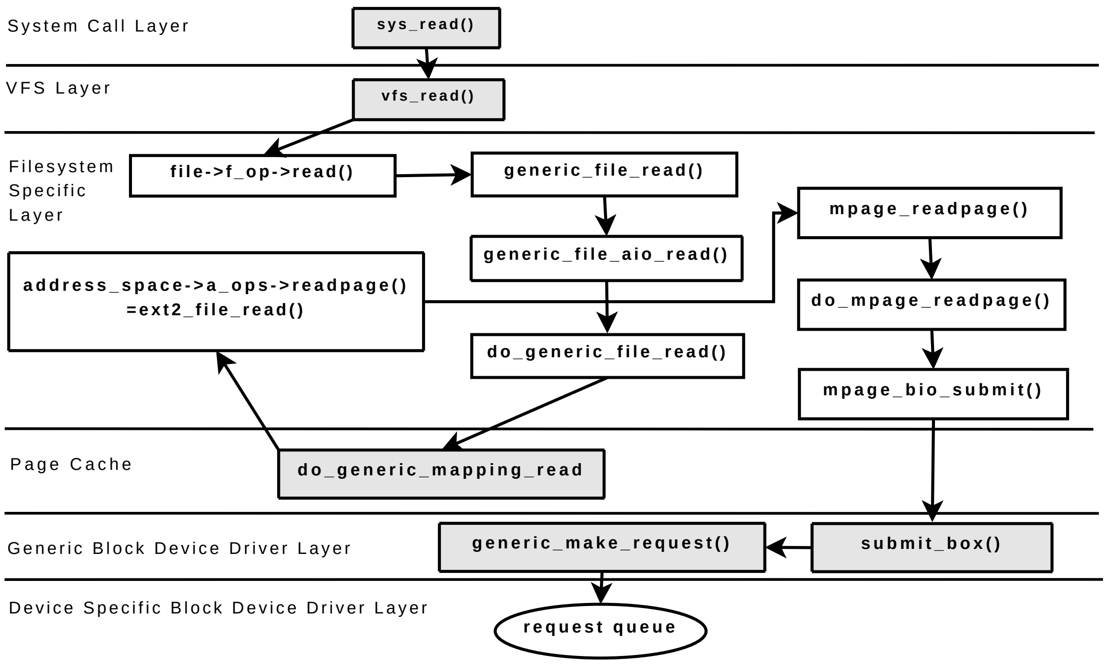


Figure 4.1: read() Top-Down Traversal

Figure 4.1 shows that the for file system related system calls, the control flows from system call layer to the VFS layer and then VFS layer switches this to a specific filesystem layer based on which filesystem the file is located. This detail is contained in dentry object of the file hence easily accessible. From the specific filesystem layer the control flows to block device driver layer via Page Cache. Once the control reaches the specific block device driver layer the request is serviced by the device, and the action completes.

4.2 Internals of SFS

SFS layer interacts closely with the virtual file system (VFS) layer. VFS handles all system calls related to file system. Whenever any file-system related system call comes to VFS, we check the location of file being saved. If the file saved, is in a directory which is prefixed by the word 'ecrypt' (meaning data to be encrypted) the data of the file will be passed through the SFS layer. We transfer the control from VFS to our SFS. The algorithm for which is shown below:

```
vfs_write{
    if (datatobeEncrypted == TRUE){
        sfs_write{
            call genrandomkey(); /* Generate random symmetric key */

            call accesscontroller(); /* Prepare ACL, HKEY and  DHACL */

            call cryptengine(); /* Encrypt the file data */

            file->f_op->write(); /* write function of underlying file system */
        }
    }
}
```

sfs_write calls the get_random_bytes function to generate a random symmetric key to be used as cryptographic key for encrypting the data. The accesscontroller function prepares the Access control list (ACL), HKEY and DHACL and all this data is appended at the beginning of the file. The cryptengine function encrypts the file data.

It is important to note here that, the data to be written in a file is passed to kernel in a user space buffer. If the file size is large the file data is split into a number of parts by the application and each part of the file is passed in the buffer, one after other. We have used a unique random generated symmetric key for encrypting the file data of per file. Therefore its important to detect the first block of a file, so that the key is generated for first block and the same key is used for subsequent blocks. Thus after generation of the key in first block, we encrypt the key using owners public key and save this in the file header. The hash of this key is also created and stored in the file header. In subsequent blocks (except the first block) we read the file header, retrieve the key and use it for the subsequent blocks.

The accesscontroller, opens the accesscontrol file located in the the directory where the file is being saved. If the accesscontrol file is not available, SFS fails and exits. The accesscontrol file containing user id of users granted access rights are read and public key corresponding to the user ids is searched from the pubkey table (this table is available with the kernel). Now the symmetric encryption key is encrypted by public key of each user granted access rights and ACL is prepared. Figure 4.2 shows the structure of Access Control List. ACL basically consist of a queue of accesscontrol elements. Each accesscontrol element contains the access control information pertaining to a particular user. The figure also demonstrates how an accesscontrol element is created, ie, AC1 is created when symmetric key is encrypted with public key of user 1. Similarly AC2, AC3 is created when symmetric key is encrypted with the public key of user 2, 3 respectively.

The cryptengine is responsible for encrypting the data and its implementation is represented by the figure 4.3.

The data to be written to the disk comes to the write system call in user buffer marked as buf in figure 4.3. The data is copied to the kernel buffer (kbuf) for further processing.

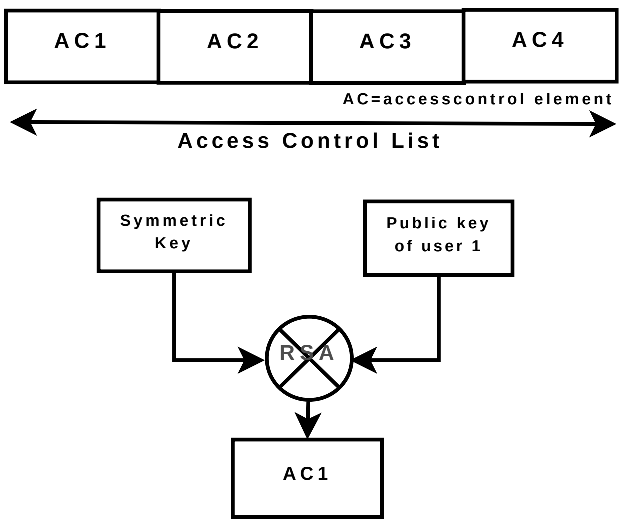


Figure 4.2: Access Control List and creation of an accesscontrol element of ACL

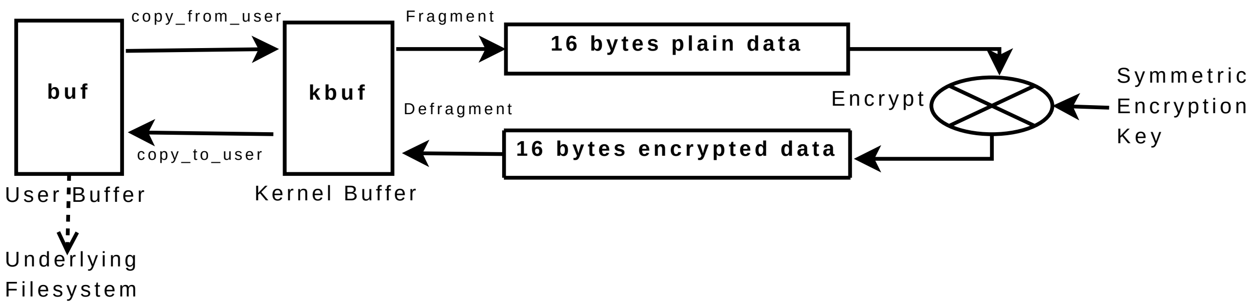


Figure 4.3: Encryption while file data writing

The data from the kernel buffer is fragmented into size of 16 bytes and is then encrypted using AES algorithm. The encrypted data is defragmented i.e, put to its original place in kbuf. Now this processed data is copied to the buf and the control is passed to the underlying file system in a usual way. Here we also pad the data with extra bytes if the block is not perfect multiple of 16. This information called as nPad is also saved in the file header and used at the time of read.

Similarly, while reading/accessing the file data from the disk the following operations are performed:-

```
vfs_read{
    if (datatobeDecrypted == TRUE){
        sfs_read{
            call fileheaderextractor(); /* Extract the file header
                              and provide it to accesscontroller */

            call accesscontroller(); /* Retrieve KEY using ACL, HKEY and DHACL */

            call cryptengine(); /* Decrypt the cryptfile data */

            file->f_op->read(); /* read function of underlying file system */
        }
    }
}
```

sfs_read calls the fileheaderextractor function to extract the file headers like ACL, HKEY and DHACL and passes it to the accesscontroller function. The accesscontroller function checks if access is granted to the user by utilizing his private key and manipulation on ACL and HKEY. If access is authorized the symmetric key is retrieved back by RSA algorithm. Actually, each accesscontrol element of ACL is decrypted with the private key of the user accessing the file. The hash of the result obtained is created and compared with the HKEY (hash of symmetric key). If it matches means the user has the access rights and the file is decrypted else access to the file is denied.

The cryptengine function decrypts the file data using the retrieved key.

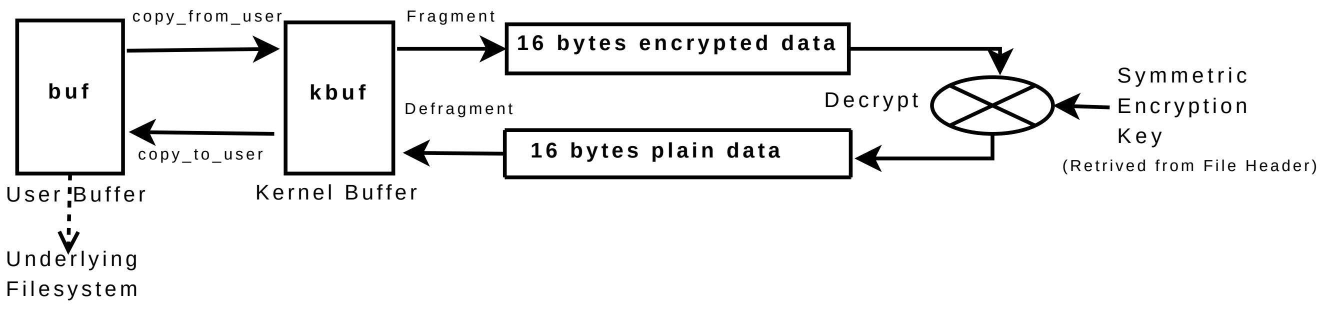


Figure 4.4: Decryption while reading file data

Figure 4.4 shows the sequence of events taking place when the data is being decrypted by cryptengine. After file decryption the padded data are removed and then the actual data is passed to the application. This whole process is done without user intervention and on the fly; therefore it provides transparency and convenience.

4.3 File Header Structure

The file header is added in front of each file by the SFS for control and providing encryption services. It is used to keep the important cryptographic accesscontrol elements which controls the access of any file in the OS having SFS as its integral part. The size of header is not fixed, its size depends on the no of user granted the access rights. Figure 4.5 shows the structure of a typical file header.

The HKEY and DHACL each takes 32 bytes as SHA-256 is used. nACL and nPAD each is of 4 bytes length, to store basically integers. Each element of ACL takes 80 bytes after encryption using RSA. The size of file header is directly proportional to the number of users granted access rights. Therefore the space overhead per file is roughly 72+nACL*80 bytes.

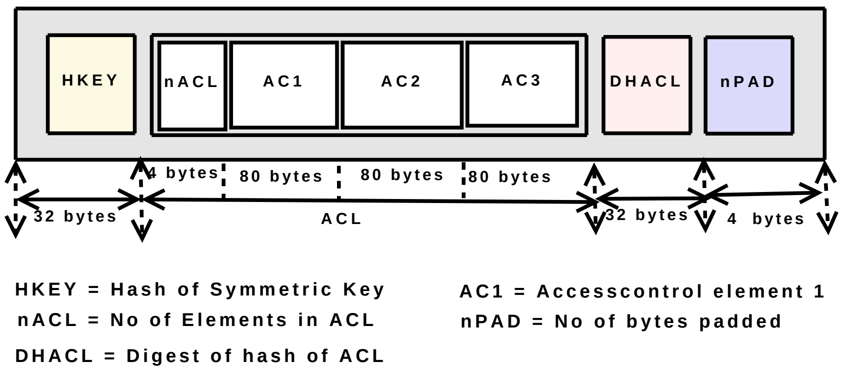

HKEY
nACL
AC1
AC2
AC3
DHACL
nPAD
32 bytes
4 bytes
80 bytes
80 bytes
80 bytes
ACL
32 bytes
4 bytes
HKEY = Hash of Symmetric Key
nACL = No of Elements in ACL
DHACL = Digest of hash of ACL
AC1 = Accesscontrol element 1
nPAD = No of bytes padded

Figure 4.5: File Header structure

4.4 Newly implemented system calls, modules, utilities

The newly implemented system calls, modules and utilities are described in this section:-

4.4.1 System Calls

We have implemented two new system calls required for SFS. These new system calls are described below:-

- **accesscontrol**
  accesscontrol system call reads the access control file and finds out the public key of the recipient of this file. The prototype of accesscontrol is as follows:

```
asmlinkage long sys_accesscontrol(unsigned int fd, int __user * buf,
                size_t count1, int __user * access_queue, size_t count2)
```

The system call receives the access_queue containing user-id of users granted access, and count2 contains number of elements in access_queue. The system call uses this data provided to update the kernel data structure. This data structure is exported to the kernel, so that its available while SFS is running in kernel mode. Exporting the symbol makes a entry in the global symbol table of kernel, therefore these variables will be available to other kernel modules.

- **storeprivatekey**
  storeprivatekey system call receives the private key of the user and sets or resets it to the privatekey kernel-variable. This kernel-variable needs to be in a secure kernel memory area and must be well protected and have very strict limited access. The prototype of storeprivatekey is as follows:

```
asmlinkage long sys_storeprivatekey(unsigned char __user * pvtkey,
size_t keycount, int flag)
```

  The system call receives the pointer to the pvtkey containing private key of the user, keycount contains the length of the private key and flag is used to denote whether kernel privatekey variable needs to be set or reset.

4.4.2 Kernel Modules

We have implemented a loadable kernel module named as loadpubkey, which will load the public-key related data into kernel. A static file containing the public key of all the user remains available on the disk. This loadable kernel module reads this file and populates the kernel data structure. This data structure contains public key of all users and once exported it will be available to the kernel.

4.4.3 Utilities

We have implemented the following utilities for providing user convenience.

- **setaccesscontrol**
  setaccesscontrol utility creates the accesscontrol file in the ecryptdir directory(ie, the directory that stores the encrypted files). The user uses this utility to set the access rights for all the files located in a particular directory. This utility needs to be run for each directory atleast once.

- **entersecuresession**
  This utility basically implements the secure session code. The utility takes private key as input, then calls setprivatekey system call which stores it in a secret kernel memory area, from where it will be used for decrypting the Access Control list (ACL). Its usage is either from a command prompt or smartcard to enable users to give their private key.

- **exitsecuresession**
  This utility basically implements the closing (or exiting from) secure session.

Chapter 5

Evaluation and Testing of SFS

We developed the Secure File System and integrated it with the Linux kernel version 2.6.22.1 for Intel X86 architecture. The Secure File System has been thoroughly tested with different applications and evaluated. In this chapter we will be covering the evaluation and test results obtained with SFS. The evaluation part has been divided into functional evaluation and performance evaluation. We have carried out few tests to ascertain the overhead SFS is introducing in respect of space and time.

5.1 Functional Evaluation

The SFS has been tested with various applications and found that it works smoothly. It has been checked with different formats of data like text or ascii files, image files, music and video files, and web files. There has been no incident of data corruption was noticed. SFS has also been tested with different types of underlying file system like ext2, ext3, FAT etc. We have identified the following functionality (or features) of Secure File System:-

- **End-to-End protection**: The file is encrypted and the access control information is embed with the file since its creation, so wherever file moves the access control and protection is ensured. Thus it provides end-to-end security, from the point of its creation till its delivery to authorized users.

- **Provides transparency and convenience**: SFS is implemented in kernel space and once integrated with OS, provides data security as its inherent feature. It hides the cryptographic services from user and needs very minimal basic settings. SFS provides cryptographic services on-the-fly, so data is never stored in plain from or sent out on network in plain form.
- **No keys are stored on the disk**: We believe that if keys are stored persistently, then encryption adds little security. SFS does not store keys on disk. The private key of the user is kept securely on a smart card or else he needs to memorize it and use it through keyboard. We have willingly separated the private key from the login password, to facilitate independence of usage and to provide an additional layer of security.
- **Keys protected from swap devices**: If memory becomes scarce, memory that could contain keys may be swapped. SFS prevents this by pinning keys in physical core kernel memory. But cleartext process data can still be written to swap.
- The current implementation supports only **single user**. However it can easily be extended for concurrent usage by multiple users.
- **Strong access control**: By using *public cryptography* and keeping the access-related information with the file itself it provides strong access control of files.
- **Users do not need root intervention**: Even the root himself cannot access files for which root has not been given access rights by the owner of the file.
- **Reveals directory structure information**: The present implementation reveals the directory structure as file-meta data information is not encrypted. This may need to be modified in order to enhance security attacks against cryptanalysis.
- **Single Cipher support**: SFS uses AES for data encryption. It uses the modularity concepts, so it can be extended to support multiple ciphers.

- **Implementation technique**: SFS is implemented in kernel space, so portability is an issue. Actually portability has not been an immediate design goal, as SFS is designed for providing file data security to a small set of users. The system becomes bit inefficient as the data from user space is copied to kernel space for manipulation and after manipulation again copied back to user space. Also For the scheme to work correctly, the kernel integrated with secure file system needs to be loaded and running on all machines in the network.

5.2 Performance Evaluation

We ran all our tests on a 1.7GHz Pentium machine with 128MB of RAM. The machine was installed with Mandriva Linux (version 2.6.22.1). We have build our kernel containing SFS on this system and gave option to run it at the time of computer start up. We ran all tests several times, and our computed standard deviations were less than 5%. We recorded elapsed, system, and user times for all tests.

We have carried out the testing in two configurations. In first configuration, we have taken a set of 50 files containing 6000 lines of C code. Our test consists of once writing and then reading the files. We measured the elapsed, system and user time by performing the test on Linux system and on Linux system containing SFS. The number of files in the set was increased by adding 50 files in each step. The readings obtained for each step are given in the Table 5.1.

The figure 5.1 shows the plot of system time taken on normal Linux system versus the system time taken on Linux system loaded with SFS. We can see both lines are linearly proportional to each other, it means the overhead due to cryptographic services (AES, RSA and SHA) grows linearly. The gap between the two lines is the overhead added in terms of time by the SFS system.

The figure 5.2 shows the plot of elapsed time on normal Linux system versus the elapsed time on Linux system loaded with SFS. It can be seen that the elapsed time for the Linux loaded with SFS is increasing at a faster speed than the normal Linux system.

Test Results				
	Normal Linux		Linux loaded with SFS	
No of Files	**System Time** in sec	**Elapsed Time** in min:sec	**System Time** in sec	**Elapsed Time** in min:sec
50	0.11	0:01.32	2.99	0:07.15
100	0.16	0:01.62	5.90	0:16.64
150	0.21	0:02.73	8.93	0:25.75
200	0.32	0:03.85	11.82	0:36.07
250	0.36	0:03.16	14.82	0:43.56
300	0.43	0:02.22	17.97	0:57.63
350	0.41	0:03.38	20.74	1:00.73
400	0.52	0:04.85	23.74	1:01.16
450	0.56	0:05.12	26.75	1:05.84
500	0.65	0:06.90	29.73	1:12.85

Table 5.1: System and Elapsed Time considering number of files

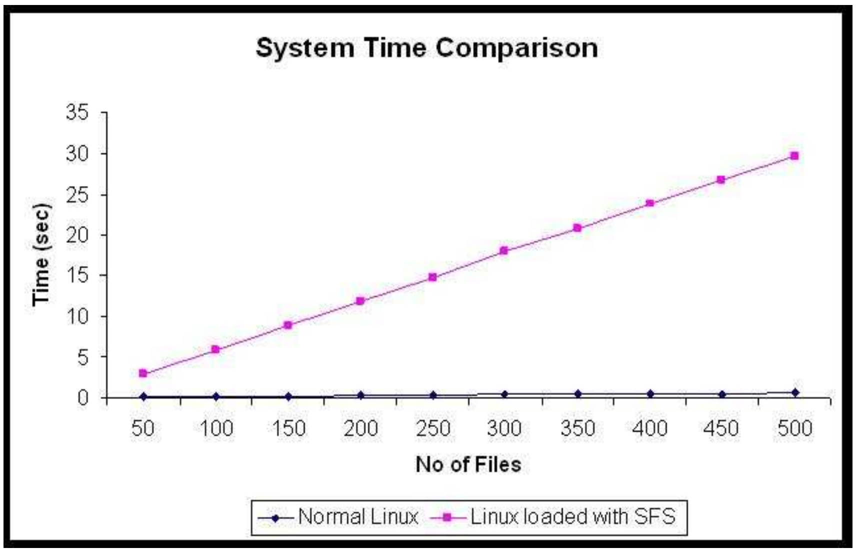


Figure 5.1: System Time comparison between normal Linux and Linux loaded with SFS considering number of files

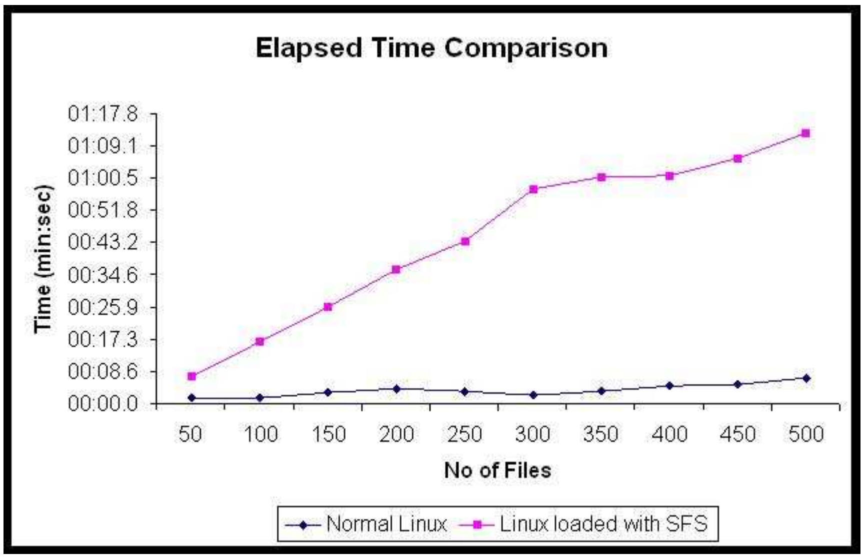


Figure 5.2: Elapsed Time comparison between normal Linux and Linux loaded with SFS considering number of files

In second configuration, the test consists of file operation on a single file. The file operation basically constitute once writing and then reading the single file. The test was repeatedly performed by increasing the file size of the single file. The reading obtained in each step is given in Table 5.2.

Figure 5.3 shows the plot of system time taken on Linux system versus system time taken on Linux system loaded with SFS. It is evident that the system time and elapsed time taken by Linux loaded with SFS, increases linearly, due to overhead of cryptographic services (mainly due to AES) incorporated into SFS. The gap between the two lines is the overhead added in terms of time by the SFS system.

On comparison of Figure 5.2 and Figure 5.4 we have noticed that as the number of files increases the elapsed time taken for file operations also increases because the access control functionality needs to be done per file basis. We have found that single file of size X and a set of n files of total size X, takes different time for same file operation. The difference in there time is because of the fact that if its a single file of large size then access control operation (RSA & SHA) needs to be done once whereas if it is a set of n

Test Results				
	Normal Linux		Linux loaded with SFS	
File Size in KB	**System Time** in sec	**Elapsed Time** in min:sec	**System Time** in sec	**Elapsed Time** in min:sec
100	0.02	0:00.35	0.44	0:01.78
300	0.04	0:00.68	1.24	0:03.07
500	0.05	0:00.58	1.74	0:04.04
700	0.06	0:00.36	2.36	0:05.32
1000	0.09	0:00.72	3.49	0:07.25
1400	0.09	0:01.19	4.62	0:08.28
1500	0.08	0:00.78	5.02	0:09.82

Table 5.2: System and Elapsed Time considering file size

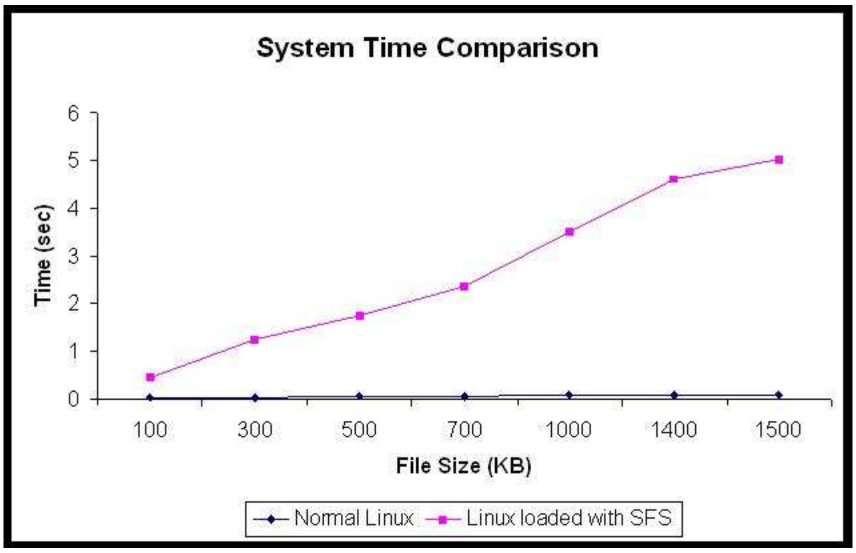


Figure 5.3: System Time comparison between normal Linux and Linux loaded with SFS considering file size

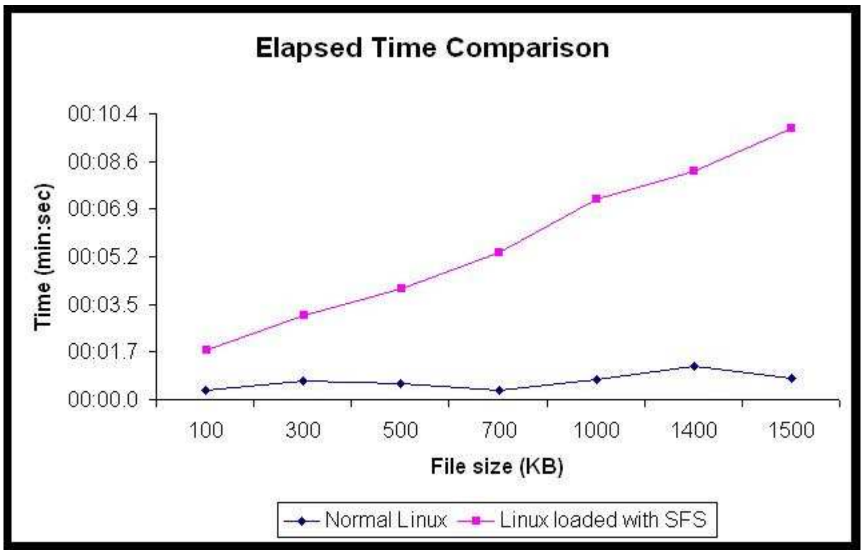


Figure 5.4: Elapsed Time comparison between normal Linux and Linux loaded with SFS considering file size

files, then access control operation needs to be done n number of times and thus it takes more time.

SFS also introduces space overhead, as it saves essential information in form of file header to carry out the cryptographic functions. Figure 4.5 in section 4.3 shows clearly that the space overhead per file is 72+80*(number of users granted access rights).

Chapter 6

Conclusions

Our main contribution is in designing and building a Secure File System that was developed with the express goal of enhancing file data security in Linux kernel. The main objective is to provide data security with user convenience. This has been done by implementing SFS in kernel space and enabling encryption and decryption of the files on-the-fly and in a transparent way.

We have seen that implementing SFS in kernel enables the operating system to provide file data security as one of its inherent functionality. SFS is cryptographically enforced and uses public-private pair key to control the access of a file. Using public cryptographic makes it more reliable, secure and in a way provides user authentication also. SFS is very convenient to user as it performs the encryption and decryption transparently and even all system administration tasks like backup, etc are having the same common interface. The scheme guarantees an end-to-end protection leading to a secure computing environment. SFS overcomes one of the major drawbacks of the TCFS, which uses one key per user for encrypting all files; moreover this key is encrypted with the user login password and saved in a database. We believe that storing data and the key on the same disk and using login password to encrypt the key has several vulnerabilities. SFS uses randomly generated key for a file and each file is encrypted with a different key. SFS stores the private key on a smart card thus separating it from data it protects. But this forces this scheme to be used on a system which has a card reader.

We achieved high security by including support for AES, designing a strong access control mechanism using public cryptography and session entry for accessing confidential data. We achieved high performance by designing SFS to run in kernel. We achieved ease of use by providing encryption and authentication that is transparent to users and process.

Bibliography

[1] Albert D. Alexandrov, Maximiliam Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user lever: the UFO global file system. In *Proc. 1997 Usenix Technical Conference*, Anaheim, CA, January 1997. Usenix Association. 1.3

[2] Mick Bauer. Paranoid penguin: BestCrypt: Cross-platform filesystem encryption. *Linux Journal*, 98:117–135, June 2002. 2.1.4

[3] M. Blaze. Key management in an encrypting file system. In *USENIX Summer 1994 Technical Conference*, Boston, MA, June 1994. 3.2.1.1

[4] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9–16, 1993. 1.3, 2.1, 2.1.2

[5] D. (Daniele) Bovet and Marco Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, Inc., pub-ORA:adr, 2000. 2.2.1, 3.2.1.2

[6] Giuseppe Cattaneo, Luigi Catuogno, Aniello Del Sorbo, and Pino Persiano. The design and implementation of a transparent cryptographic file system for UNIX. In Clem Cole, editor, *USENIX Annual Technical Conference, FREENIX Track*, pages 199–212. USENIX, 2001. 1.3, 2.1.3

[7] Paul G. Comba. Approaches to cryptographic key management. 1986. 3.2.1.1

[8] Roland C. Dowdeswell and John Ioannidis. The cryptographic disk driver. In *USENIX Annual Technical Conference, FREENIX Track*, pages 179–186. USENIX, 2003. 2.1

[9] David Mazi Eres, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security, October 26 1999. 2.1.5

[10] Dahl A. Gerberick. Cryptographic key management or strong network security management. *ACM SIGSAC*, 8(2):12–23, Summer 1990. no refs. 3.2.1.1

[11] James P. Hughes and Christopher J. Feist. Architecture of the secure file system, June 26 2001. 3.2

[12] Naomaru Itoi. SC-CFS: Smartcard secured cryptographic file system. In USENIX, editor, *Proceedings of the Tenth USENIX Security Symposium, August 13–17, 2001, Washington, DC, USA*, pages 121–129, pub-USENIX:adr, 2001. USENIX. 2.1.2

[13] M. Frans Kaashoek. Self-certifying file system implementation for windows, August 30 2002. 2.1.5

[14] Stephen T. Kent. Some cryptographic techniques for file protection. In *CRYPTO*, page 80, 1981. 2.1, 3.2

[15] SeongKi Kim, WanJin Park, SeokKyoo Kim, Sunil Ahn, and Sangyong Han. Integration of a cryptographic file system and access control. In Hsinchun Chen, Fei-Yue Wang, Christopher C. Yang, Daniel Dajun Zeng, Michael Chau, and Kuiyu Chang, editors, *WISI*, volume 3917 of *Lecture Notes in Computer Science*, pages 139–151. Springer, 2006. 3.2.1.2

[16] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. *Operating Systems Review*, 33(5):124–139, December 1999. 3.2.1.1

[17] Andrew McDonald and Markus Kuhn. StegFS: A steganographic file system for Linux. In Andreas Pfitzmann, editor, *Information Hiding —3rd International Workshop, IH'99*, volume 1768 of *Lecture Notes in Computer Science*, pages 463–477, Dresden, Germany, October 2000. Springer-Verlag, Berlin Germany. 2.1.7

[18] Matthew Scott Rimer and M. Frans Kaashoek. The secure file system under windows NT, December 19 1999. 2.1.6

[19] Alessandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., pub-ORA:adr, second edition, 2001. 2.2

[20] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, 1994. 1.1.2

[21] S. Thyregod. Key management in cryptographic access control. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2006. Supervised by Assoc. Prof. Christian D. Jensen, IMM. 3.2.1.1

[22] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux Security Modules: General security support for the Linux kernel. In *Proceedings of the 11th USENIX Security Symposium*. USENIX, August 2002. 3.2

[23] Joonsuk Yu, Jaedeok Lim, and Jeongnyeo Kim. A cryptographic file system supporting multi-level security. WSEAS Int. Conf. on e-activities Singapore, December 9-12, 2002, December 9-12 2002. WSEAS. 2.1