Enhancing File Data Security in Linux Operating System by Integrating Secure File System

Rajesh Kumar Pal, Indranil Sengupta

Abstract— **In today's world securing file data is very important. The proposed 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. After SFS is integrated with the Linux operating system (OS), it enables OS to provide File Data Security as its inherent functionality. 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 is fully compatible with all underlying storage file systems. This paper describes the design and implementation of SFS for Linux which extends the operating system to provide file data security as its inherent functionality. We have discussed the motivation for the work, the proposed SFS architecture and its implementation details in the subsequent sections.**

I. Introduction

Governments, military, financial institutions, hospitals, and private businesses amass 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. If the confidential information about a businesses customers or finances or new product line fall into the hands of a competitor, then 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.

In computer systems information is stored traditionally in the 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. It is a well accepted fact that *securing file data is very important*, in modern computing environment.

Rajesh Kumar Pal is a research scholar in Dept of Computer Science and Engineering, Indian Institute of Technology, Delhi, India-110010. (email: rkpal@cse.iitd.ernet.in)

Dr. Indranil Sengupta is a professor and head Dept of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India-721302. (e-mail: isg@iitkgp.ac.in).

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 being 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 proposed Secure File System into the kernel itself.

In this paper we have introduced a new architecture of file data security in Linux, the Secure File System (SFS). SFS works on the layer of Virtual File System (VFS) and brings the encryption services into the kernel space. If a file needs to be encrypted, it will be rerouted from VFS to SFS. Operating system having SFS embedded into its kernel, provide file data security as one of its basic functionality to all applications. SFS is fully compatible with all underlying storage file systems and all applications.

II. Related Work

There 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 and the password as inputs and produce the encrypted file. This type of utility is good for limited use only, as it is very cumbersome and manual. Second approach is integrating encryption engine in application software itself, where each program that is to manipulate sensitive data has built-in cryptographic facilities. But the disadvantage here is that all application should use the same encryption engine and any change in one will require changes in all. 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 block. 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 users. 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

978-1-4244-2769-7/09/$25.00 ©2009 IEEE

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 focused on this issue and proposed SFS that solves the file data security problem. We 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, therefore we have decided to push the encryption services into the Linux kernel space mounted beneath the virtual file system.

There has been a lot of development [3, 4, 9] taken place since the time when MS DOS device driver [5] was used to encrypt the entire partition. Nowadays we have several cryptographic file systems available, which we have briefly described.

2.1 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 [2]. 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 Data Encryption Standard (DES) to encrypt file data.

The CFS prototype is implemented entirely at user level, communicating with the Unix kernel via the NFS interface. Its main disadvantage is that it runs in user mode, thus requires many context switches and data copies from user space to kernel space.

2.2 Transparent Cryptographic File System (TCFS)

TCFS works as a layer under the VFS (Virtual File system) layer, making it completely transparent to the applications [3]. The security is guaranteed by means of the DES 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 login 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.

2.3 BestCrypt

BestCrypt is designed as a loopback device driver which creates a raw block device with a single file [4]. 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.

2.4 Self-certifying File System

The Self-certifying File System addresses the issue of key management in cryptographic file systems and proposes separating key management from file system security [7, 9]. Servers have a public key and clients use the server public key to authenticate the server and establish a secure communication channel. To allow clients to authenticate servers on the spot without even having heard of them before, 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 talking to the legitimate server.

Once the client has verified the server, a secure channel is established and the actual file access takes place.

2.5 Encryption File System (EFS)

Microsoft Windows uses EFS which is based on Windows authentication methods and Windows ACLs [6]. 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.

III. PROPOSED SECURE FILE SYSTEM

3.1 Design Goals

We have designed Secure File System (SFS) with the aim 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 are 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.

- 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.
- Transparent Performance: Encrypted files should behave no different from some other files.
- Convenience: The system should be convenient to users.

3.2 Design of proposed Secure File System

The proposed Secure File System is designed to provide the above mentioned goals. Figure 1 shows the normal flow of control in standard file system. VFS shown in figure 1 has namely two main functions. 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. In our design, we have taken the control flow from VFS layer based on some condition and rerouted it to proposed SFS. The condition that is checked is the location where the file is destined to be saved. If the location is the directory starting with prefix word 'ecrypt' (e.g., ecryptdir) then we take the control flow to SFS layer. Figure 2 shows SFS layer is mounted beneath the VFS and interacts closely with it. VFS and proposed SFS functions in kernel space, therefore a user cannot access them directly. The above condition will be checked and executed by kernel.

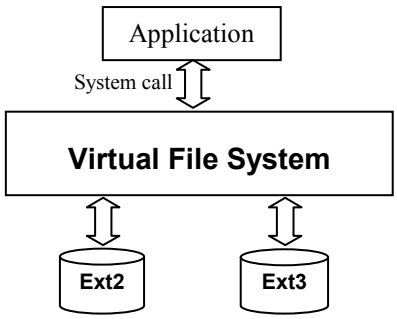

Fig 1 : Control flow in standard file system

After study of kernel filesystem structure and related works [8, 15], we have found that mounting SFS beneath VFS is the idle place because then we can efficiently use the kernel infrastructure and deviate only where it is required. In this way, it achieves one major advantage that, it provides uniform interface to 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 cryptographic techniques, it passes the data to VFS. Therefore this design decision effectively utilizes the kernel infrastructure in providing uniform interface and user convenience.

We decide to divert the control to SFS layer based on the name of directory under which the file will be saved. 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 design decision provides user convenience. This is just a mechanism we have used, which can be strengthened further by providing some utility to create directories which will contain secret data.

IV. Architecture of Secure File System

Figure 2 shows the architecture of Secure File System in detail. SFS mainly has four components:-

- Key Management Unit (KMU)
- Crypt Engine (CE)
- Access Controller (AC)
- File Header Extractor (FHE)

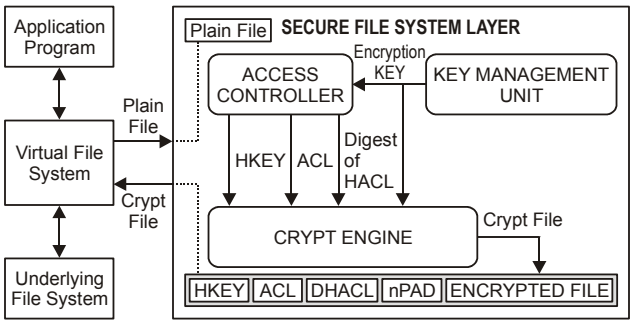

Fig 2 : Secure File System

4.1 Key Management Unit

Key management is a very crucial task and has been discussed in [1, 7]. 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 number 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. Therefore each time the file is saved; a new random symmetric key is used. So it aids in providing strong security. KMU also loads the public key of users at system startup.

4.2 Access Controller

AC is responsible for creation and management of all access related information with respect to a file. 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 the recipients of all the files stored in that directory, and thus controls the authorized recipients. 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.
- 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. Detail of file owner is stored along with file; therefore extracting its public key is not an issue.

4.3 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 a multiple of 16, we pad the file to make the file size a 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 pass it to the application. So application never comes to know about padded bytes. CE also receives the HKEY, ACL, and DHACL from the accesscontroller and attaches them as a file header to the encrypted file. Figure 2 shows the whole structure containing file header and the encrypted file, 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.

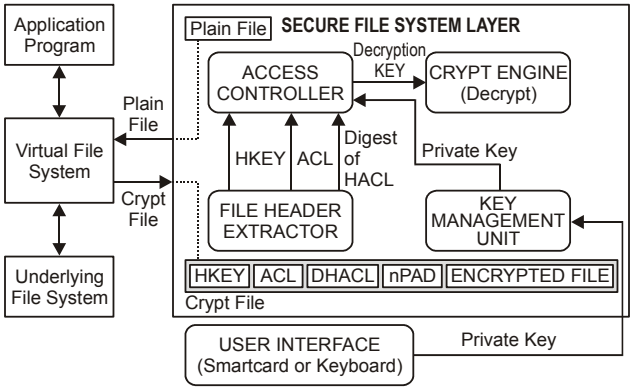

Fig 3 : File access using Secure File System

4.4 File Header Extractor

FHE comes into play when the file data is being accessed from the disk (refer figure 3). 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 either through keyboard or smartcard. It is presumed that private key is kept safely by its owner.

4.5 SFS Operation

In this section we will describe the sequence of events which takes 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.

4.5.1 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, the user may use KWrite utility to create a text file as shown in figure 4. 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 activated for all the files saved in directory starting with word 'ecrypt'. The save command issued from the application activates the system call sys_write to write the application data on the disk. It is known that all filesystem related system calls are handled by

VFS, so the control reaches to the VFS. Here the location where file is being saved will be checked, and if the name of the directory is prefixed with 'ecrypt' is true then the control flow will be transferred to the SFS layer. As shown in the figure 2, the following action will take place:-

1. Key Management Unit will generate the random symmetric key.
2. Crypt Engine will 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 a perfect multiple of 16.
3. Access Controller will generate the HKEY, ACL, and DHACL and append them at the beginning of the encrypted file, making the crypt file.
4. Crypt file is saved on the disk.

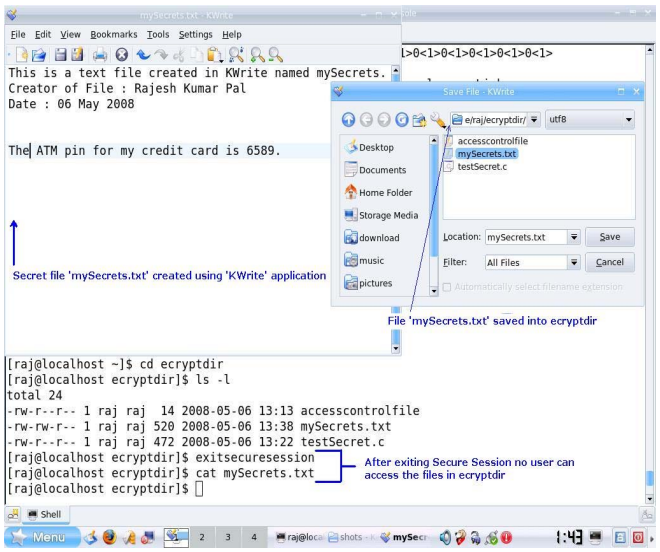

Fig 4 : Screenshot showing file creation in SFS

4.5.2 File Access

For accessing confidential data or files the user needs to enter the secure session by entering his private key either through keyboard or smartcard. 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 (i.e., access denied to unauthorized user, as shown in figure 4). The sequence of action which takes place while confidential file is accessed is shown in figure 3 and is enumerated below:-

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 it may not be. So to verify, the hash of this extracted key is generated using SHA-256 and compared with the HKEY. If both HKEYs match, 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.

V. 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. 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.

5.1 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 it is 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 owner's 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. This way SFS can handle large files needing several disk blocks consistently.

The accesscontroller function opens the 'accesscontrol' file located in 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 made available by KMU to the kernel). Now the symmetric encryption key is encrypted by public key of each user granted access rights and ACL is prepared. Figure 5 shows the structure of Access Control List. ACL is 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, i.e., 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.

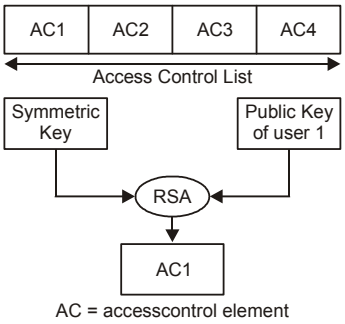

Fig 5 : Access Control List and creation of an accesscontrol element of ACL

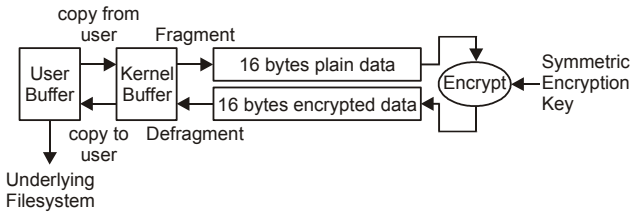

Fig 6: Encryption while file data writing

The cryptengine function is responsible for encrypting the data and its implementation is represented by the figure 6. The data to be written to the disk comes to the write system call in user buffer. The data is copied to the kernel buffer for further processing. 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 kernel buffer. Now this processed data is copied to the user buffer and the control is passed to the underlying file system in a usual way. The data is padded 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 reading or accessing the file. Similarly, while reading/accessing the file data from the disk the following operations are performed:-

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

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 user's 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.

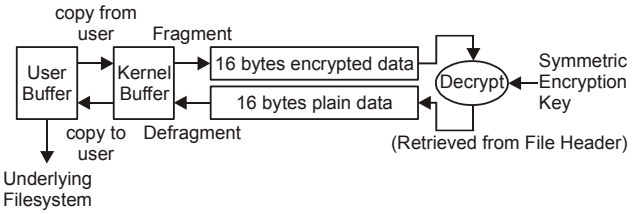

Fig 7: Decryption while reading file data

The cryptengine function decrypts the file data using the retrieved key. Figure 7 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.

5.2 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 element 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 8 shows the structure of a typical file header. The HKEY and DHACL each take 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, where nACL is the number of authorized recipients of the file.

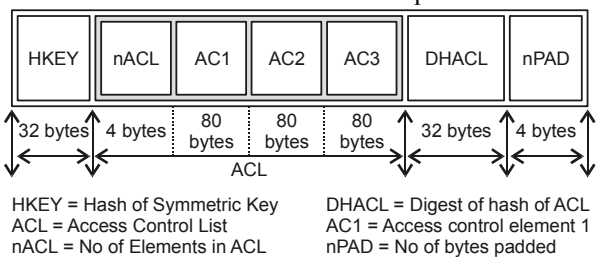

Fig 8: File header structure

VI. Evaluation and Testing

We developed the Secure File System and integrated it with the Linux kernel version 2.6.22.1 on Intel x86 architecture. The Secure File System has been thoroughly tested with different applications and evaluated. In this section 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.

6.1 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 embedded 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 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 form or sent out on network in plain form.
- No keys are stored on the disk: 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.
- 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.
- 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 where high degree of security is required. 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.

6.2 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 built 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.

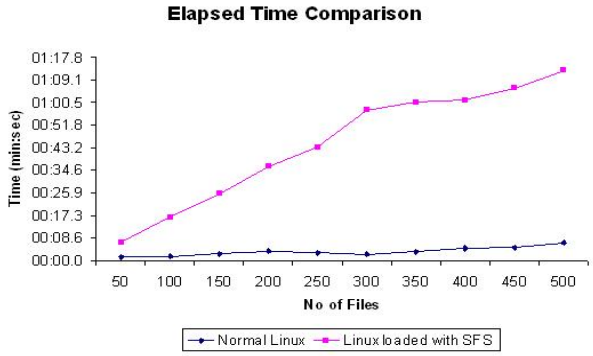

Figure 9: Elapsed time comparison between normal Linux and Linux loaded with SFS considering number of files

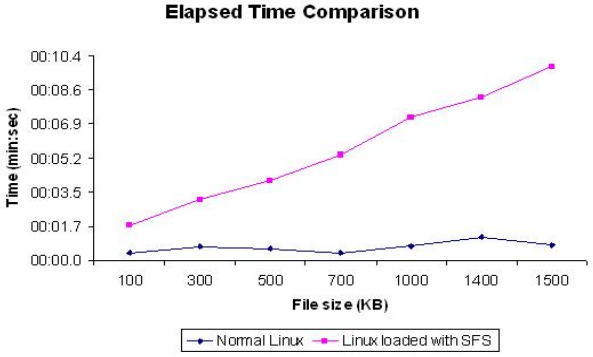

Figure 10: Elapsed time comparison between normal Linux and Linux loaded with SFS considering file size

The figure 9 shows the plot of elapsed time on normal Linux system versus the elapsed time on Linux system loaded with SFS based on number of files. 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.

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. Figure 10 shows the plot of elapsed time taken on Linux system versus elapsed time taken on Linux system loaded with SFS based on file size. It is evident that the 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 9 and Figure 10 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 a 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 their time is because of the fact that in a single access control operation (RSA & SHA) needs to be done once whereas in a set of n files, access control operation needs to be done n 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 8 shows clearly that the space overhead per file is 72+80*(number of users granted access rights).

VII. CONCLUSION

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 cryptography 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. 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 the 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.

REFERENCES

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

[2] Matt Blaze. A cryptographic file system for UNIX. In *ACM Conference on Computer and Communications Security*, pages 9-16, 1993.

[3] 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.

[4] Mick Bauer. Paranoid penguin: BestCrypt: Cross-platform filesystem encryption. Linux Journal, 98:117, June 2002.

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

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

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

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

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

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

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

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

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

[14] 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.

[15] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002), pages 15-30, Monterey, CA, January 2002.