CHAPTER 5

ECFS: AN ENTERPRISE-CLASS CRYPTOGRAPHIC FILE SYSTEM FOR LINUX

This chapter proposes a secure and efficient approach for designing and implementing an Enterprise-class Cryptographic File System for Linux (ECFS) in kernel-space. It uses stackable file system interface to introduce a layer for encrypting files using symmetric keys, and public-key cryptography for user authentication and file sharing, like other existing enterprise-class cryptographic file systems. It differs itself from existing systems by including all public-key cryptographic operations and Public Key Infrastructure (PKI) support in kernel-space that protects it from attacks that may take place with a user-space PKI support. It thus has a narrow domain of trust than existing systems. It uses XTS mode of AES algorithm for file encryption for providing better protection and performance. It also uses kernel-keyring service for improving performance. It stores the cryptographic metadata in file's Access Control List (ACL) as extended attributes to ease the task of file sharing.

5.1 INTRODUCTION

Kernel-space cryptographic file systems like NCryptfs [Wright et. al. (2003a)], eCryptfs [Halcrow (2005), Kirkland (2011)], TransCryptDFS [Modi et al., (2010)], uses stackable file system interface approach [Zadok and Badulescu (1999), Zadok and Nieh (2000), Zadok et al. (1999)] to introduce a layer of encryption that can fit over any underlying file system. These file systems are more efficient than user-space cryptographic file systems. eCryptfs and TransCryptDFS also supports for file sharing among multiple users using public-key cryptography, as explained in section 2.3.3.2. Public key certificate verification in eCryptfs is performed in user-space, while in TransCryptDFS, it is performed in kernel-space. In eCryptfs, a PGP inspired file header format stores the cryptographic metadata associated with each file. Including metadata in file contents as a header in eCryptfs reduces transparency for end-use and requires separate tools to manage file sharing. TransCryptDFS stores cryptographic metadata as extended attributes in file's Access Control List (ACL) [Grunbacher (2003)] to ease file sharing task.

Enterprise-class Cryptographic File System for Linux (ECFS) [Rawat and Kumar (2012a)], is based on eCryptfs, but it stores cryptographic metadata in file ACL as extended attributes to ease the task of file sharing, like TransCryptDFS. The other key differences from eCryptfs are mentioned below.

Public-key management in eCryptfs, for user authentication and file sharing, is done by a user-space daemon, named *eCryptfsd*, which can be easily spoofed by user-space processes having superuser privileges, to provide the kernel with the wrong public-key and hence cannot be trusted. ECFS includes whole public-key infrastructure (PKI) support in Linux kernel to exclude privileged user-space processes from domain of trust.

eCryptfs uses CBC mode for file encryption with keyed hashes, like HMAC, for file integrity. ECFS uses XTS (**X**EX-based **T**weaked codebook mode with ciphertext **S**tealing) mode of the AES algorithm [IEEE (2008), Dworkin (2009)] for file encryption to get better performance. XTS-AES provides more protection than the other approved confidentiality-only modes against unauthorized manipulation of the encrypted data, thus does not require separate integrity support, as explained in section 2.2.4. It also provides random access to encrypted data.

Subsequent sections explain overall ECFS architecture, key management scheme, various cryptographic operations and implementation details. In last section, performance and security evaluation of proposed system is presented, along with conclusion.

5.2 ECFS ARCHITECTURE AND CRYPTOGRAPHIC OPERATIONS

Overall ECFS architecture is shown in Figure 5.1. ECFS layer uses stackable file system interface [Zadok and Badulescu (1999)] for performing various symmetric and asymmetric cryptographic operations. It uses File System Translator (FiST) API that provides a set of stackable file system templates [Zadok and Nieh (2000), Zadok et al. (1999)] and a high-level language that can describe stackable file systems in a cross-platform portable fashion. FiST's code generation tool, fistgen (http://www.filesystems.org), compiles a single file system description into loadable kernel modules for several operating systems (currently Solaris, Linux, and FreeBSD). So, ECFS can fit over any underlying file system. ECFS components along with cryptographic operations are described in this section.

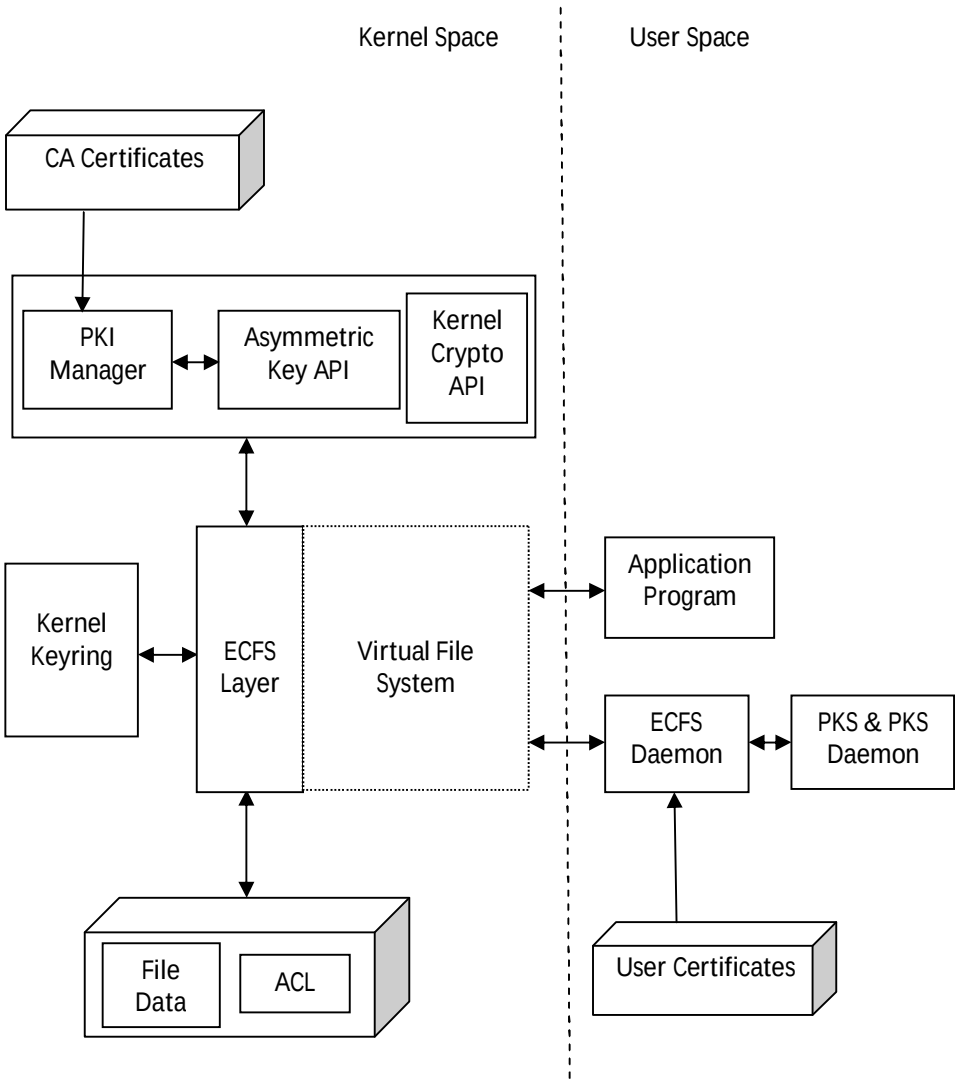

Figure 5.1: Overall ECFS Architecture

5.2.1 FILE ENCRYPTION

Using kernel crypto API, ECFS generates a random per-file File Encryption Key (FEK) and random 64-bit per-file tweak (TWK), for each file created that is used to carry out encryption of file contents using XTS-AES algorithm. XTS-AES allows for random access and encrypts file to the same length as their original size. XTS tweak of 128-bit is formed by concatenating 64-bit per-file tweak value (TWK) with 64-bit file offset. The idea behind using a tweak is to get unique, per-file ciphertext, thus protects from chosen ciphertext attack and copy-paste attack. If file size is not an integer multiple of the cipher block size, the XTS construction uses ciphertext stealing. Ciphertext stealing is a technique that can be used to encrypt data that

does not comprise an integer multiple of the cipher's block size. It does this by combining the last two blocks of ciphertext. Sparse files are detected by examining if all 4096 bytes in a sector are zero before decryption, in which case they will not be decrypted rather zero-filled sector is returned.

5.2.2 User Authentication and File Sharing

A public-key and private-key pair is associated with every user for authentication and file sharing. Per-file FEK and per-file tweak are encrypted with the public keys of all users who have access to the file. When a file is created, such metadata entries are created and stored in ACL, for the owner and those users with default access to the file. Later, such metadata entries are also created for other users when they are granted access to a file.

ECFS generates public and private key-pair for every user of the authentication domain using *openssl*. It uses Base64-encoded X.509 certificate in PEM format signed by a certificate authority trusted by the organization. It does not use public-private key pairs based on passphrase, like pkcs#5, due to vulnerabilities of passphrase based cryptography [Boklan (2009)]. These user certificates are kept in certificate repository at a publicly known network location.

ECFS takes verified public-keys from kernel-space PKI manager, discussed in subsequent subsection. Then, it encrypts FEK || TWK with public-keys of authorized users using RSA based asymmetric key API, and store in the file ACL as a token along with other user information. An ACL entry in ECFS looks like as:

UID: username: permissions: token

Here *token* is per-file encryption key concatenated with per-file random tweak, encrypted with the user's public-key i.e. E^{p}_{user} *(FEK|| TWK)*.

5.2.3 File Operations

When user opens a file, ECFS looks in ACL for the corresponding entry of the user, retrieves the token i.e. encrypted FEK and TWK, and sends it to user's Private Key Store (PKS) via ECFS daemon. PKS daemon decrypts the token by using private-key of the user, sends FEK

and TWK, to the kernel via ECFS daemon. It implies that ECFS daemon acts as a channel between the kernel and user's PKS. Then, FEK and TWK will be stored in kernel keyring.

During file read and write operations, ECFS retrieves FEK and TWK from kernel keyring. It uses inode number of file to index FEK in the keyring.

File based PKS with private-key kept on a disk file or USB drive has been used in proposed approach, but it can also be kept on a smart card in secure manner.

5.2.4 KERNEL-SPACE PKI SUPPORT

All public-key cryptographic operations and PKI functions in eCryptfs [Halcrow (2005), Kirkland (2011)] are performed in user-space by a user-space daemon. The daemon does certificate acquisition, certificate verification, extracts public-key from certificate and then use it to encrypt per file encryption key. The user-space daemon is prone to attacks by user-space processes having root privileges. It can be easily spoofed to provide the kernel with the wrong public-key and hence cannot be trusted.

Considering the above issue, ECFS does all asymmetric cryptographic operations and PKI functions in kernel-space. PKI manager module maintains a list trusted certificate authority's certificates which are used to verify the X.509 formatted user certificates. ECFS retrieves the user certificate from ECFS daemon and then, it will be parsed and verified in kernel-space by PKI manager. ECFS requires user public-keys for creating token at the time of file creation and ACL manipulation, as discussed in the subsection, User authentication and file sharing. ECFS caches verified user public-keys in kernel keyring [Howells (2004)] for improving performance, as it need not require contacting ECFS daemon for user certificate and PKI manager for verification, if public-key is present in keyring.

5.3 ECFS IMPLEMENTATION

ECFS has been implemented for 2.6.34.10 Linux kernel stable release. Changes have been made in kernel-space as outlined in the previous section. ECFS daemon and PKS daemon have been implemented in user-space.

5.3.1 KERNEL-SPACE CHANGES

ECFS layer has been implemented using stackable file system templates (http://www.filesystems.org) to perform symmetric and asymmetric cryptographic operations. It uses kernel crypto API for per-file random FEK & TWK generation, and symmetric encryption/decryption of file contents using XTS-AES algorithm. It also uses asymmetric key API for user's public-key validation and for encryption/decryption of FEK||TWK. The following four functions provided by FiST API has been use for encrypting and decrypting file contents and file names:

- *encode_data* function takes input argument as a buffer of size 4KB or 8KB (typically page size) from user space, and returns encoded buffer of same size to lower level file system. ECFS uses this function to encrypt the file contents. This function also returns a status code indicating any possible error (negative integer) or the number of bytes successfully encoded.

- *decode_data* function is the inverse function of *encode_data* function. Input argument to the function is a buffer read from lower level file system and returns decoded buffer of same size to the user space. ECFS uses this function to decrypt a block of data.

- *encode_filename* function takes a file name string as input from user system call and returns a newly allocated and encoded file name to lower level file system. It also returns a status code indicating either an error (negative integer) or the number of bytes in the new string. This function has been used to encrypt filenames.

- *decode_filename* function is the inverse function of *encode_filename* and otherwise has the same behaviour. It has been used to decrypt filenames.

RSA based asymmetric key cryptography API has been implemented using RSA algorithm patch [Parisinos (2007)]. It offers an API for faster modular exponentiation and multi precision arithmetic operations. The asymmetric key API implemented provides an interface similar to the existing kernel Crypto API. It provides the *setkey, freekey, encrypt, decrypt, sign* and *verify* functions.

PKI manager has been implemented using PolarSSL library (http://polarssl.org) and ported to the Linux kernel. It is used to parse and verify user certificates.

Posix ACL's are implemented using extended attributes [Grunbacher (2003)] inside the kernel data structures. The posix*_acl_entry* and *xattr_ acl_entry* structures, that define the kernel's representation of ACL entries, have been augmented with the 'token' field. On-disk format of *ext4* ACL entries, defined in the *ext4_acl_ entry* and *ext4_ acl_ entry_ short* structures have also been augmented with 'token' field. Suitable changes have also been made to various ACL manipulation functions.

ECFS uses kernel keyring service [Howells (2004)] to store and retrieve FEK and TWK at the time of open/creat and read/write system calls respectively. It uses session specific keyring and calls the *key_put* and *request_key* functions respectively to store and retrieve FEK||TWK. In *request_key* function, inode pointer has been used as the *description* which will be used in *match* function for searching a particular FEK||TWK. It also stores verified public-keys in keyring, which will be retrieved at the time of file creation and ACL manipulation. *User id* has been used as the *description* for searching a particular public-key.

5.3.2 USER-SPACE ECFS DAEMON

ECFS daemon acts as the channel for transfer of packets between the kernel and user's private-key store for decrypting the token. It also provides the kernel with user's certificate to create the token. ECFS daemon has used netlink sockets for communicating with the kernel and sockets API to communicate with the PKS daemon.

5.3.3 USING ECFS

For using ECFS, kernel-space and user-space modules of ECFS have to be compiled and installed, and then ECFS has to be mounted to use it.

The ECFS kernel should be compiled with ECFS option 'on', while kernel is being configured. This will also enable several cryptographic libraries, RSA crypto API, X.509 module and libraries used by them. Then the kernel should be compiled in the usual way and setup, for being run at boot.

Then, ECFS daemon should be compiled and setup for running from */etc/inittab* with the *respawn* flag.

Finally, ECFS has to be mounted using mount utility with *acl* and *user_attr* options, to enable access control list and user extended attributes respectively. Other options have to be specified for the private-key store, which may be a disk file, USB drive or smart card. In proposed approach, *.pem* files have been used as private-key store which can be stored on disk or USB drive. So, mount command can be executed as given below:

#mount -t ecfs -o acl,user_attr,key=openssl:keyfile=/usb-drive/mykey.pem /secret/mountpt.

ECFS now writes encrypted files in */secret* directory.

5.4 EVALUATION

In this section, performance and security evaluation of ECFS [Rawat and Kumar (2012a)] with CFS_Unix [Blaze (1993), Blaze (1997)], EncFS [Ozen (2007), Gough (2011)], extended CFS_Unix and extended EncFS [Rawat and Kumar (2012b)], eCryptfs [Halcrow (2005), Kirkland (2011)] crytptographic file systems, is presented.

5.4.1 PERFORMANCE

Performance of ECFS and ECFS mounted over NFS has been evaluated by running IOZone v3 [Norcott (2003)] like other cryptographic file systems in chapter 3 and chapter 4. IOZone is a popular file system benchmarking tool that performs synthetic read/write tests to determine the throughput of the system. Tests have been conducted on a 3 GHz Intel core i-3 machine with 2GB RAM running Linux kernel 2.6.34. For ECFS mounted over NFS, server and client machines of above mentioned configuration are connected over an isolated Fast Ethernet LAN of 100 MBPS.

Table 5.1 reports the throughput obtained in Kbytes/sec for write and read operation in ECFS and ECFS mounted over NFS. The *iozone* utility has been run on file system mount point with *–a* option (auto mode), for obtaining the throughput with file sizes varying from 64 Kbyte to 512 Mbyte.

 # iozone -a -i 0 -i 1 -b /home/output.xls

where *–i* option specifies the read and write tests (0 and 1 respectively); and *–b* option specifies the output file in which obtained throughput will be stored after successful execution of the command.

Table 5.1: Throughput obtained (in KBPS) for write and read operation in ECFS and ECFS mounted over NFS

File Size	Write Throughput	Read Throughput	Write Throughput over NFS	Read Throughput over NFS
64k	719961	496932	683963	472085
128k	524839	683495	498597	649320
256k	1288015	924338	1223614	878121
512k	663411	2921076	630240	2628968
1m	680512	3648606	612461	3283746
2m	680179	1820286	612161	1638258
4m	652367	1787599	587130	1519460
8m	795933	4997758	676543	4248095
16m	710046	2156821	603539	1725457
32m	930711	2316806	791105	1853445
64m	1380075	3091747	1104060	2318810
128m	1104523	3567816	331357	891954
256m	173012	4322349	51904	1080587
512m	91776	4239060	27533	1059765

Performance overhead for write and read operation is calculated from obtained throughput (Table 5.1 and Table 3.1a, 3.1b) for ECFS cryptographic file system with respect to unencrypted Ext4 file system, and is recorded in Table 5.2.

Performance overhead in obtained throughput for ECFS has been compared with other cryptographic file systems in Figure 5.2a and Figure 5.2b for write and read operation respectively. Performance overhead for other cryptographic file systems has been taken from Table 3.2a and Table 3.2b for write and read operation respectively.

Table 5.2: Percentage write and read overhead in ECFS and ECFS over NFS with respect to Ext4

File Size	Write Overhead	Read Overhead	Write Overhead over NFS	Read Overhead over NFS
64k	10	13	15	17
128k	11	13	15	17
256k	13	17	17	21
512k	15	16	19	24
1m	16	17	24	25
2m	17	16	25	24
4m	20	17	28	29
8m	21	21	33	33
16m	21	27	33	41
32m	21	28	33	42
64m	21	28	37	46
128m	27	32	78	83
256m	31	30	79	82
512m	33	32	80	83

The overhead is more in user-space cryptographic file systems (CFS_Unix, extended CFS_Unix, EncFS, extended EncFS) as compared with kernel-resident file systems (eCryptfs and ECFS), as they have to perform many context switches and data copies between kernel-space and user-space as mentioned in section 2.3. Performance of ECFS is somewhat better than eCryptfs. This may be attributed to the following reason that eCryptfs uses keyed hashes, like HMAC, for file integrity, rather ECFS does not use any separate mechanism for integrity. It uses XTS mode that itself provides it better protection than other confidentiality-only modes. Performance gain of around 5% for file sizes upto 2 MB and about 13% for large file sizes in obtained throughput for write, and around 10 % for obtained throughput for read has been observed in ECFS as compared with eCryptfs, shown in Figure 5.3.

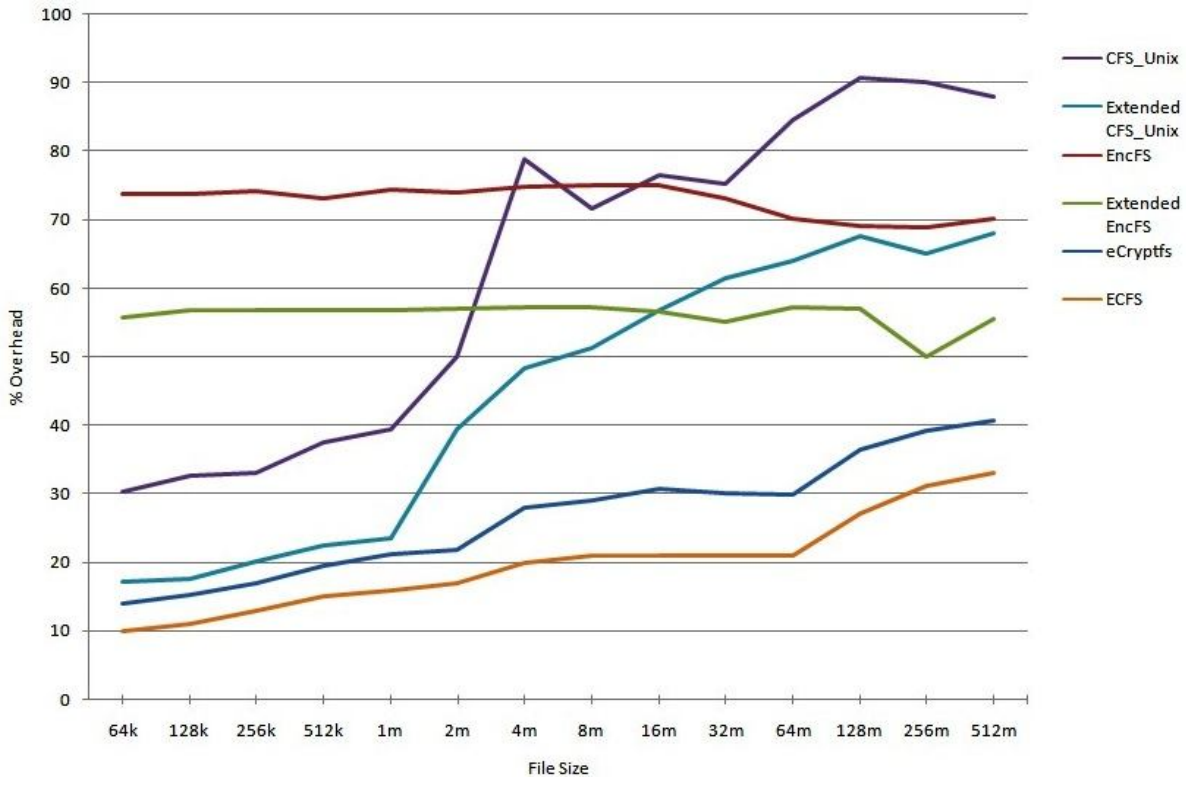

Figure 5.2a: Write overhead comparison of ECFS with other cryptographic file systems

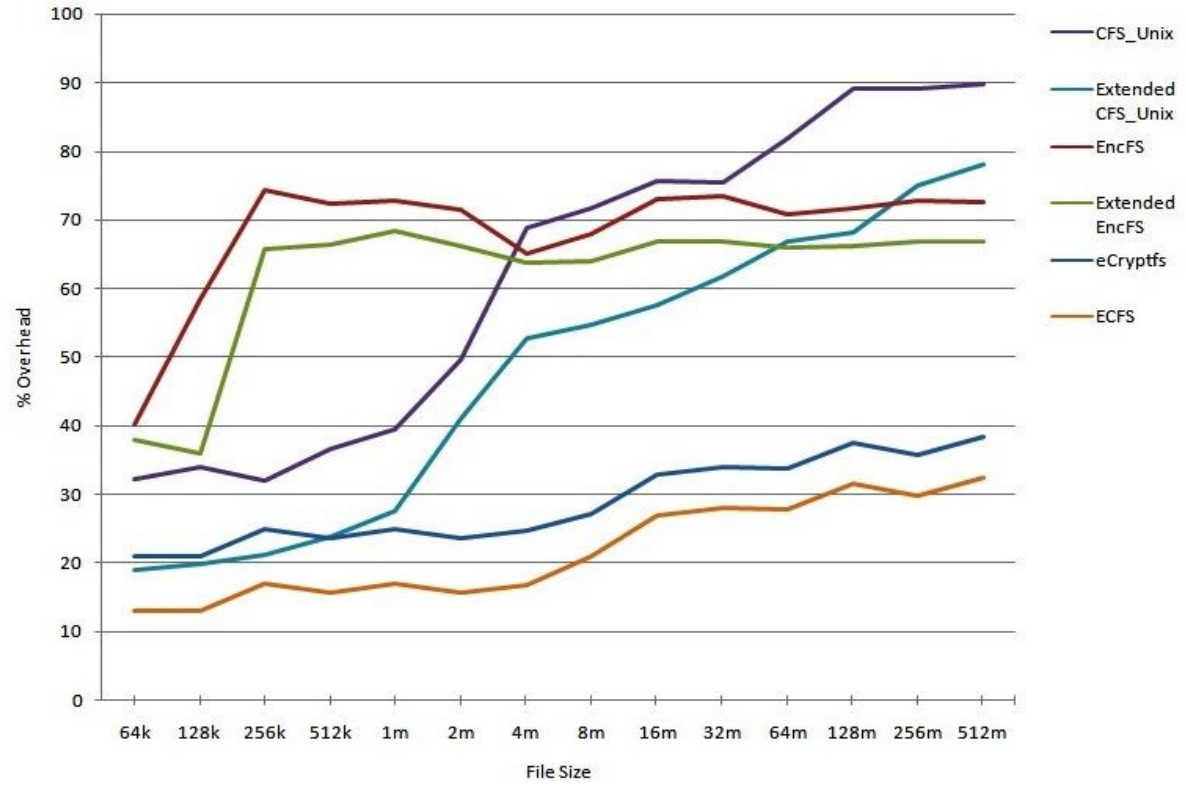

Figure 5.2b: Read overhead comparison of ECFS with other cryptographic file systems

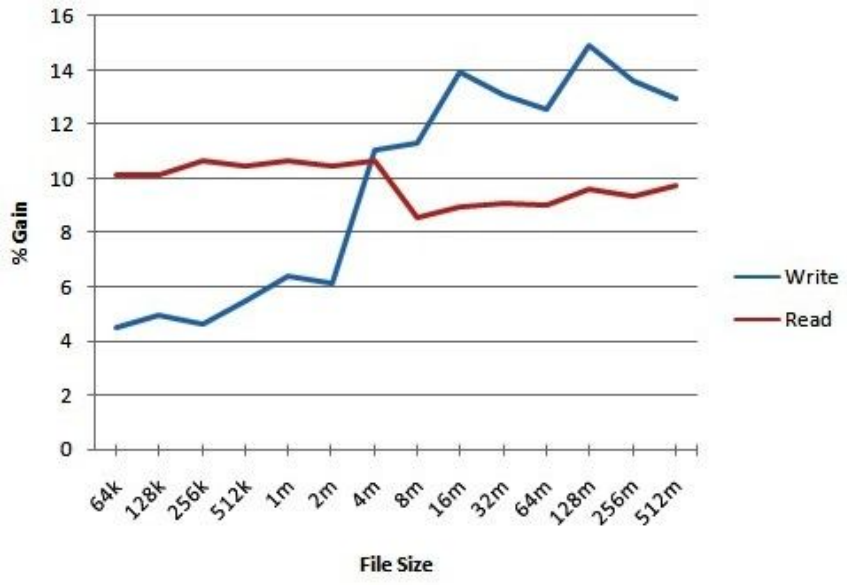

Figure 5.3: Gain in obtained throughput for write and read operation in ECFS as compared with eCryptfs

Performance overhead over NFS in ECFS has also been compared with extended CFS_Unix, extended EncFS and eCryptfs cryptographic file systems mounted over NFS in Figure 5.4a and 5.4b for write and read operation respectively. Performance overhead for extended CFS_Unix, extended EncFS and eCryptfs mounted over NFS has been taken from Table 4.3a and Table 4.3b for write and read operation respectively.

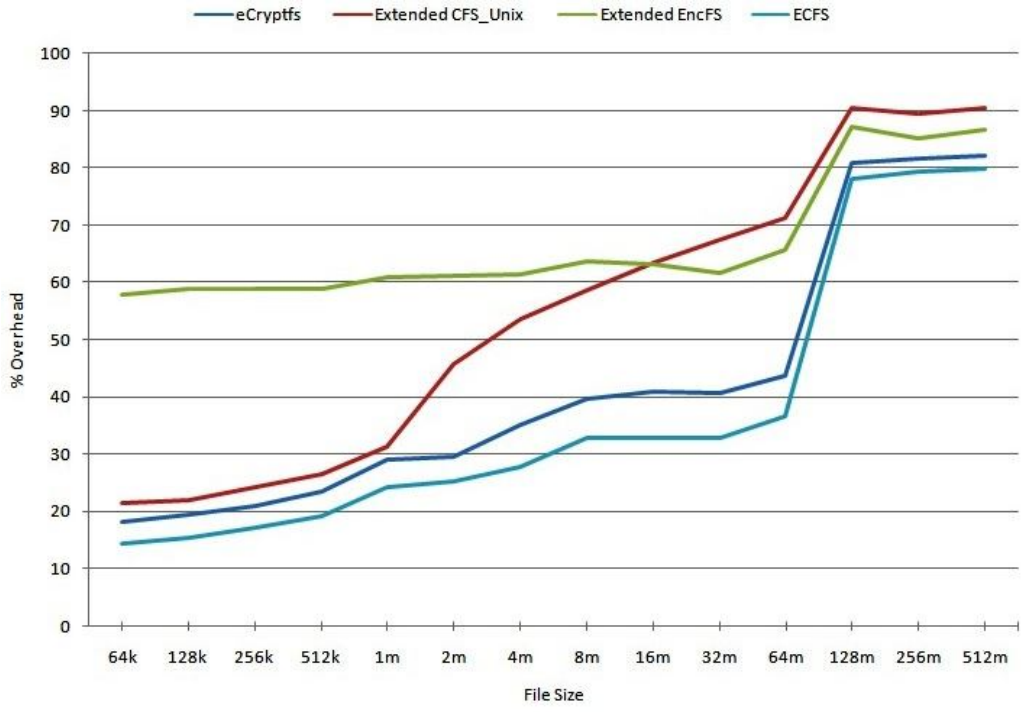

Figure 5.4a: Write overhead comparison of ECFS over NFS with other cryptographic file systems over NFS

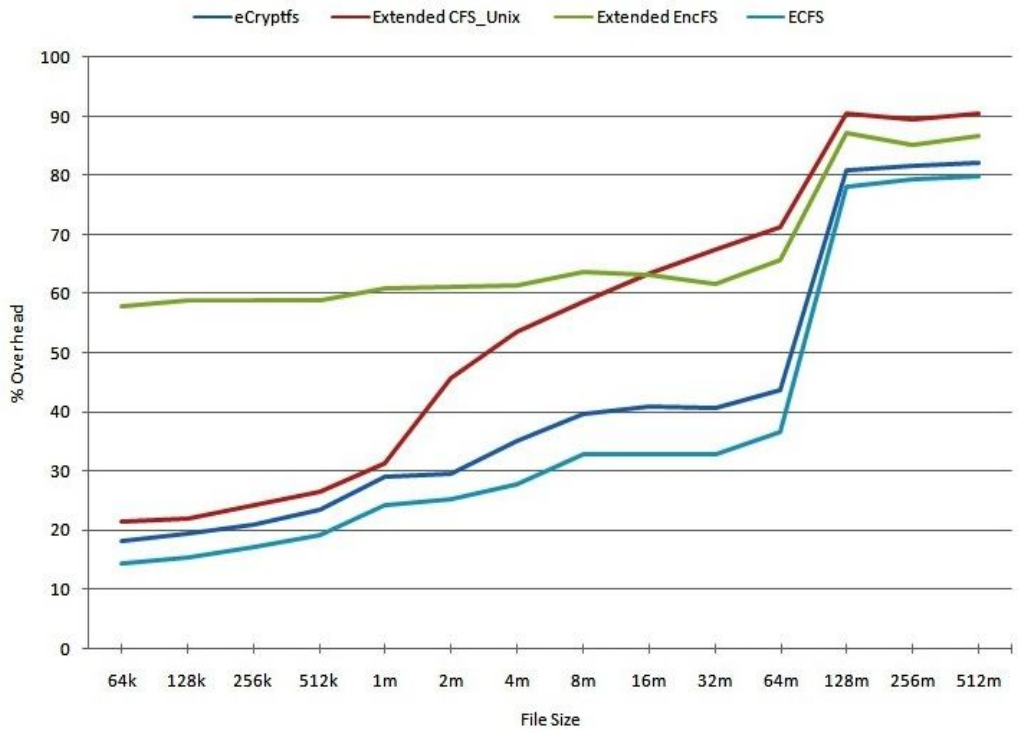

Figure 5.4b: Read overhead comparison of ECFS over NFS with other cryptographic file systems over NFS

5.4.2 SECURITY

Kernel-space cryptographic file system implementations of ECFS [Rawat and Kumar (2012a)] and eCryptfs [Halcrow (2005), Kirkland (2011)] can offer better security than user-space cryptographic file systems, CFS_Unix [Blaze (1993), Blaze (1997)], EncFS [Ozen (2007), Gough (2011)], extended CFS_Unix and extended EncFS [Rawat and Kumar (2012b)], because it is harder to access kernel resident information. Further, ECFS uses whole PKI support in kernel to protect from attacks that can take place with user-space PKI support module, as in case of eCryptfs.

But, in all the above file systems, plaintext keys and file data exist in the memory of user processes or in kernel memory. Thus, an attacker with root privileges and having source access could read kernel memory through */dev/kmem* and get acess to plaintext keys and file data by following data structures representing processes, users, filesystems, and vnodes.

5.5 SUMMARY

ECFS has been designed and implemented supporting the requirements of an enterprise-class cryptographic file system. Kernel-space PKI support protects ECFS from attacks that can take place with user-space key management daemons. ECFS simplifies key management process by keeping cryptographic metadata in file ACL. It uses kernel keyring service to cache cryptographic keys for improving performance. XTS-AES algorithm for file encryption benefits ECFS with better performance. Performance gain of around 5% for file sizes upto 2 MB and about 13% for large file sizes in obtained throughput for write, and around 10 % for obtained throughput for read has been observed in ECFS as compared with eCryptfs.