CHAPTER 2

LITERATURE REVIEW

This chapter presents an in-depth literature survey of existing cryptographic file systems. The chapter starts with a description of various design goals and design parameters that should be considered while designing a cryptographic file system. A brief description of various ciphers and modes of operations used by existing cryptographic file systems has been provided along with a detailed description of **X**EX-based **T**weaked codebook mode with ciphertext **S**tealing (XTS) [IEEE (2008), Dworkin (2009)] that can be used by cryptographic file systems for better performance. Then, existing cryptographic file systems at the block device level and at file system level in user-space and in kernel space are presented with their advantages and limitations. Further, a brief review of trusted computing technologies and benefits of using them for key management in cryptographic file systems has been described. Finally, Summary of the properties of existing cryptographic file systems has been presented along with the problems identified for carrying out research work.

2.1 CRYPTOGRAPHIC FILE SYSTEMS DESIGN GOALS

A cryptographic file system (CFS) design may take several approaches concerning the placement of encryption layer, as explained in section 1.3. The design of cryptographic file system needs to trade between flexibility, efficiency and security. Various design goals and design parameters that should be considered [Balze (1993)] while designing a cryptographic file system are mentioned below:

- **Encryption Layer.** This decides where the actual encryption/decryption operations are performed on the contents of a file during the write/read process from the disk to memory. This would decide where data remain in plain text during various stages on buffer cache or page cache. The criterion is important because some of these buffers are per-process and others are per-system.
- **Transparent access semantics.** Encrypted files should support the same access methods available on the underlying file system. All system calls should work in the same way, and it should be possible to compile and execute in a completely encrypted environment.

- **Transparent performance. C**ryptographic algorithms incur computational overhead, so the performance penalty associated with cryptographic file systems should not be so high that it discourages their use. In particular, interactive response time should not be noticeably degraded.
- **Key Granularity and Encryption Target.** This refers to the smallest unit which uses the same encryption key. This may be whole file system, a directory or a file. It is also necessary to decide what elements are stored encrypted on the disk, i.e., file data, metadata, file system based information etc.
- **Concurrent access.** It should be possible for several users (or processes) to have access to the encrypted files simultaneously. Sharing semantics should be similar to those of the underlying file system.
- **Protection of network connections.** Various attacks like masquerade, interception, replay etc., that may occur to obtain sensitive file data in a networked environment, should be considered.
- **Compatibility with underlying system services.** Administrators should be able to backup and restore individual encrypted files without the use of special tools and without knowing the key.
- **Portability.** A CFS should exploit existing system interfaces for their implementation. Encrypted files should be portable to different operating systems; they should be usable wherever the key is supplied.
- **Scale.** The encryption engine should not place an unusual load on any shared component of the system. File servers in particular should not be required to perform any special additional processing for clients who require cryptographic protection.
- **Limited trust.** In general, the user should be required to trust only those components under his or her direct control and whose integrity can be independently verified. A user should not be required to trust the file servers in case of remote file access.
- **Compatibility with future technology**. Several emerging technologies have potential applicability for protecting data. In particular, keys may be contained in or managed by smart cards or Trusted Platform Module (TPM) [TCG (2011)]. Cryptographic file systems should support novel hardware of this sort.

Integrity is another important criterion for a security-conscious user, but is generally not essential for a cryptographic file system design and hence is not present in the above list.

2.2 CIPHERS AND MODES OF OPERATION

Existing cryptographic file systems uses symmetric block ciphers for encrypting file contents as they are more efficient than asymmetric ciphers. Asymmetric ciphers like RSA is used for providing fine grained file sharing support in few cryptographic file systems, for encrypting the symmetric file encryption key, explained in section 2.3.3.2.

A block cipher takes a fixed-length block of text of length *b*-bit and a key as input and produces a *b*-bit of ciphertext. If the amount of plaintext to be encrypted is greater than *b* bits, then the block cipher can still be used by breaking the plaintext up into *b*-bit blocks. To apply a block cipher in a variety of applications, five ‘‘modes of operation’’ have been defined by the National Institute of Standards and Technology (NIST) in special publication SP 800-38A [Dworkin (2001)], namely Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feed Back (CFB), Output Feed Back (OFB) and Counter (CTR) mode. There detailed explanation is provided in Appendix A. A mode of operation is a technique for enhancing the effect of a cryptographic algorithm or adapting the algorithm for an application, such as applying a block cipher to a sequence of data blocks or a data stream [Wright et. al. (2003b)]. The five modes are intended to cover a wide variety of applications of encryption, but none of these modes are completely suitable for cryptographic file systems. In the simplest ECB mode, each 64-bit block of a file is independently encrypted with the given key. Encryption and decryption can be performed randomly on any block boundary. Although this protects the file contents, it can reveal information about the file structure, repeated blocks can be easily identified by an opponent. Other modes of DES operation include various chaining ciphers that base the encryption of a block of the data that preceded it. These modes defeat the kinds of structural analysis possible with ECB mode, but make it difficult to randomly read or write in constant time. For example, a write to the middle of a file could require reading the data that preceded it, and again encrypting and rewriting the data that follow it. Existing cryptographic file systems uses these modes in such a manner, so that they can provide for random access, also defeating structural analysis and other possible attacks, explained in detail in section 2.3. Another important mode, **X**EX-based **T**weaked codebook mode with ciphertext **S**tealing (XTS) [IEEE (2008), Dworkin (2009)] can be used by cryptographic file systems for better performance. The standard describes a method of encryption for data stored in sector-based devices where the threat model includes possible access to stored data by the adversary.

This section now provides a discussion of the Data Encryption Standard (DES) variants, Blowfish, and Advanced Encryption Standard (AES) because they are often used for file encryption in popular cryptographic file systems, and are believed to be secure. Other block ciphers like CAST, GOST, IDEA, MARS, Serpent, RC5, RC6, and TwoFish have similar characteristics with varying block and key sizes. XTS-AES mode has been described in section 2.2.4.

2.2.1 DATA ENCRYPTION STANDARD (DES)

DES is a block cipher designed by IBM researchers with assistance from the National Security Agency (NSA) in the 1970s [Schneier (1995), Stallings (2011)]. It was the first encryption scheme that was adopted as a standard by National Institute of Standards and Technology (NIST) as Federal Information Processing Standard (FIPS) publication PUB 46 [NIST (1977)]. DES uses a 56-bit key, a 64-bit block size, and can be implemented efficiently in hardware.

The overall scheme for DES encryption is illustrated in Figure 2.1. Left-hand side of the Figure shows the processing of the plaintext in three phases. First, the 64-bit plaintext passes through an initial permutation (IP) that rearranges the bits to produce the permuted input. This is followed by a phase consisting of 16 rounds of the same function, which involves both permutation and substitution functions. The output of the last (sixteenth) round consists of 64 bits that are a function of the input plaintext and the key. The left and right halves of the output are swapped to produce the preoutput. Finally, the preoutput is passed through a permutation (IP^{-1}) that is the inverse of the initial permutation function, to produce the 64-bit ciphertext. With the exception of the initial and final permutations, DES has the exact structure of a Feistel cipher [Feistel (1973)], shown in Figure 2.2.

The right-hand portion of Figure 2.1 shows the way in which the 56-bit key is used. Initially, the key is passed through a permutation function. Then, for each of the 16 rounds, a *subkey* (K_i) is produced by the combination of a left circular shift and a permutation. The permutation function is the same for each round, but a different subkey is produced because of the repeated shifts of the key bits.

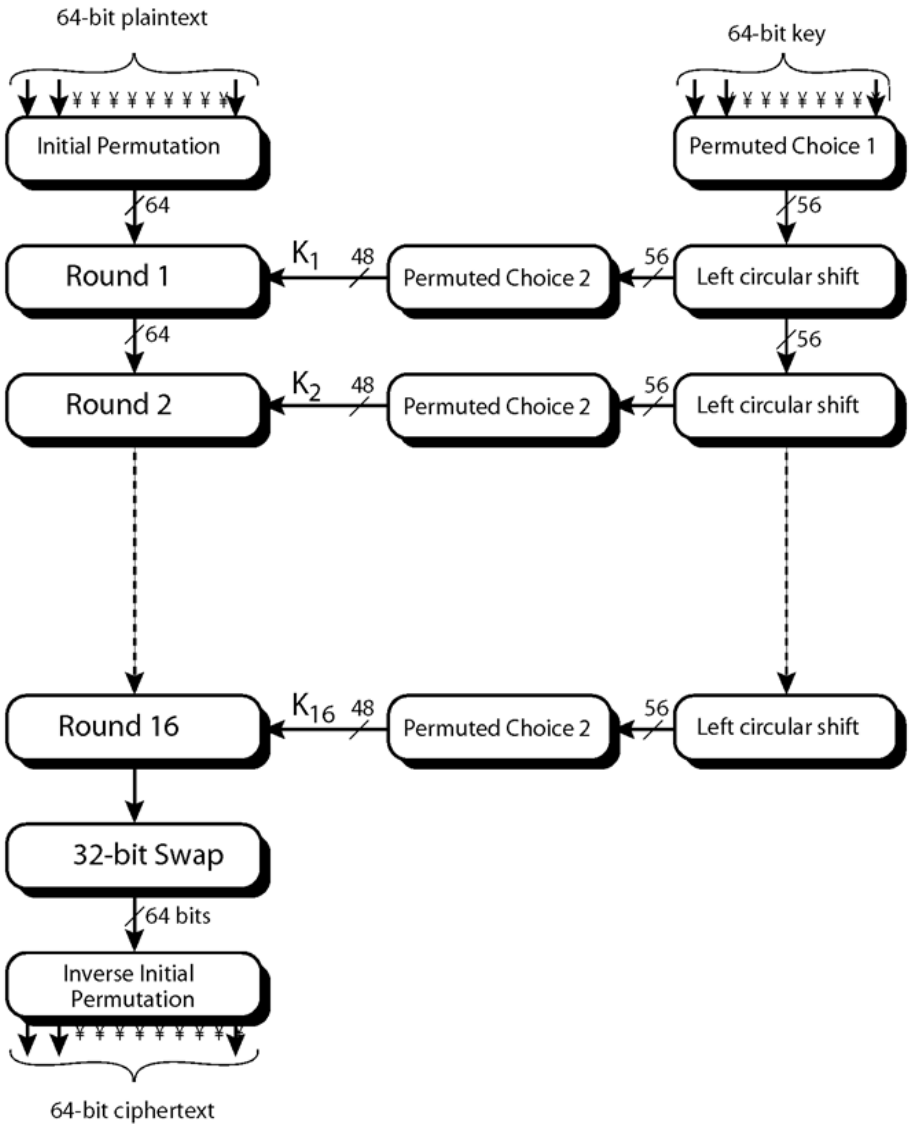

Figure 2.1: DES Encryption

DES is no longer considered to be secure due to smaller key size; the brute force attack can break DES in a few hours. There are several more secure variants of DES, like triple DES. Triple DES FIPS PUB 46 [NIST (1999)] uses three separate DES encryptions with three different keys, increasing the total key length to 168 bits. DESX is a variant designed by RSA Data Security that uses a second 64-bit key for whitening the data before the first round and after the last round of DES, thereby reducing its vulnerability to brute-force attacks, as well as differential and linear cryptanalysis [Schneier (1995)].

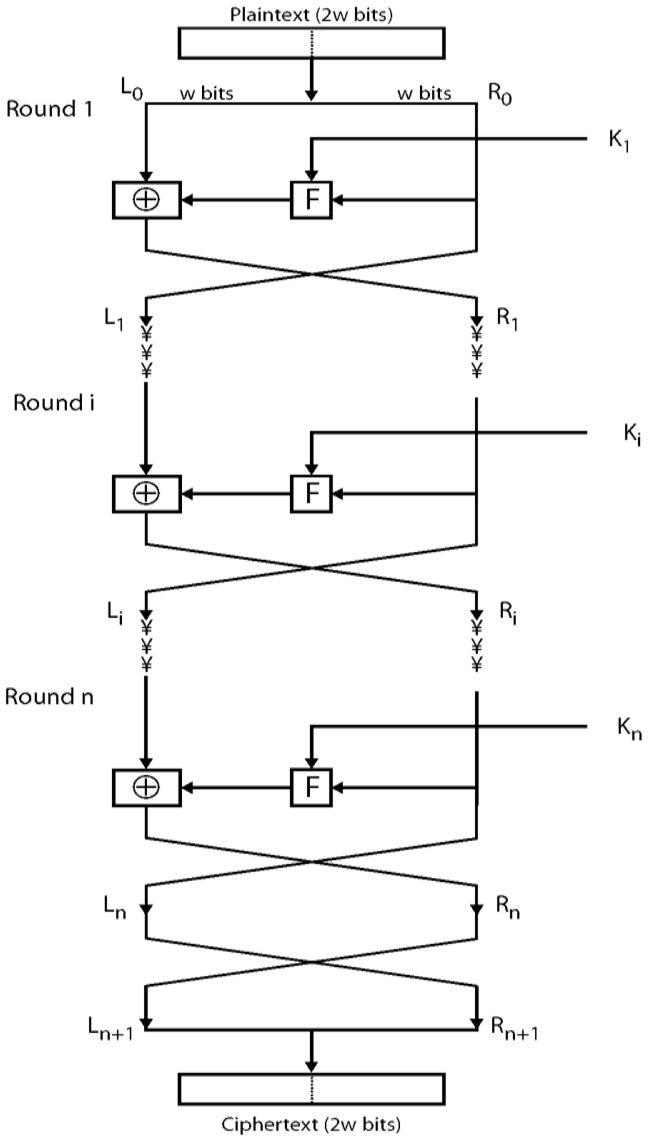

Figure 2.2: Feistel Cipher Structure

2.2.2 BLOWFISH

Blowfish is a block cipher designed by Bruce Schneier with a 64-bit block size and key sizes up to 448 bits [Schneier (1995)]. Blowfish had been designed considering four design criteria: speed, compact memory usage, simple operations, and variable security. The algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a variable-length key of at most 56 bytes (448 bits) into several subkey arrays (P-array and S-boxes) with a total of 4168 bytes.

The P-array consists of 18 32-bit subkeys: P_1, P_2,..., P_{18}

There are four 32-bit S-boxes with 256 entries each:

$S_{1,0}$, $S_{1,1}$,..., $S_{1,255}$;

$S_{2,0}$, $S_{2,1}$,..,, $S_{2,255}$;

$S_{3,0}$, $S_{3,1}$,..., $S_{3,255}$;

$S_{4,0}$, $S_{4,1}$,..,, $S_{4,255}$.

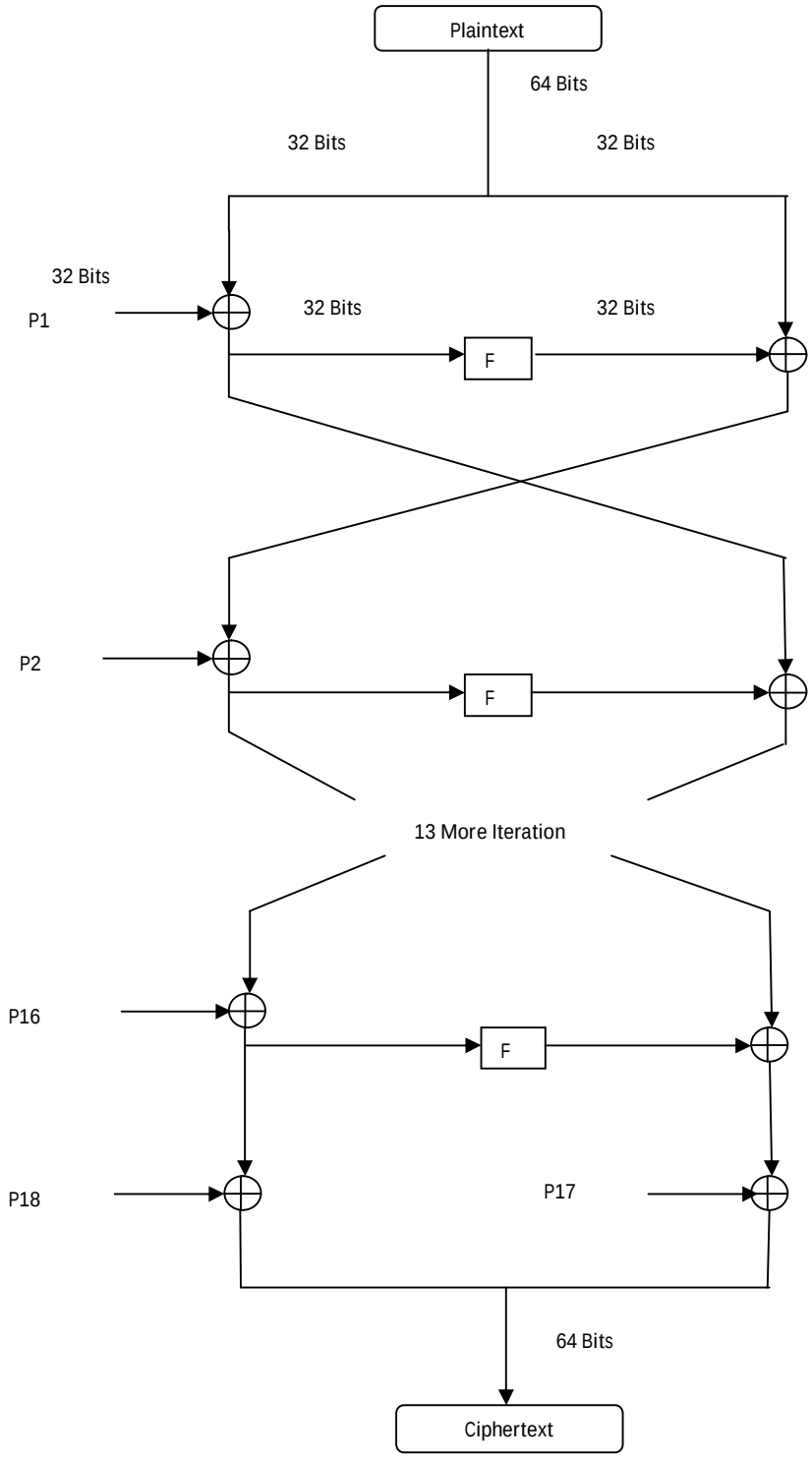

Figure 2.3: Blowfish Encryption

Data encryption consists of a simple function iterated 16 times shown in Figure 2.3. The input 64-bit plaintext is divided into two 32-bit halves. The overall encryption process can be seen as:

$$L_i = F(L_{i-1} \oplus P_{i-1}) \oplus R_{i-1}$$
$$R_i = L_{i-1} \oplus P_{i-1}$$
$$L_{17} = L_{16} \oplus P_{18}$$
$$R_{17} = R_{16} \oplus P_{17}$$

The function F is shown in Figure 2.4. Divide the 32-bit input *X* to the function F into four 8-bit quarters and apply F as follows:

$$F(X_{31-0}) = ((S1[X_{31-24}] + S2[X_{23-16}]) \oplus S3[X_{15-8}]) + S4[X_{7-0}]$$

Here addition operation is module 2^{32} addition.

In essence, each round consists of a key-dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words.

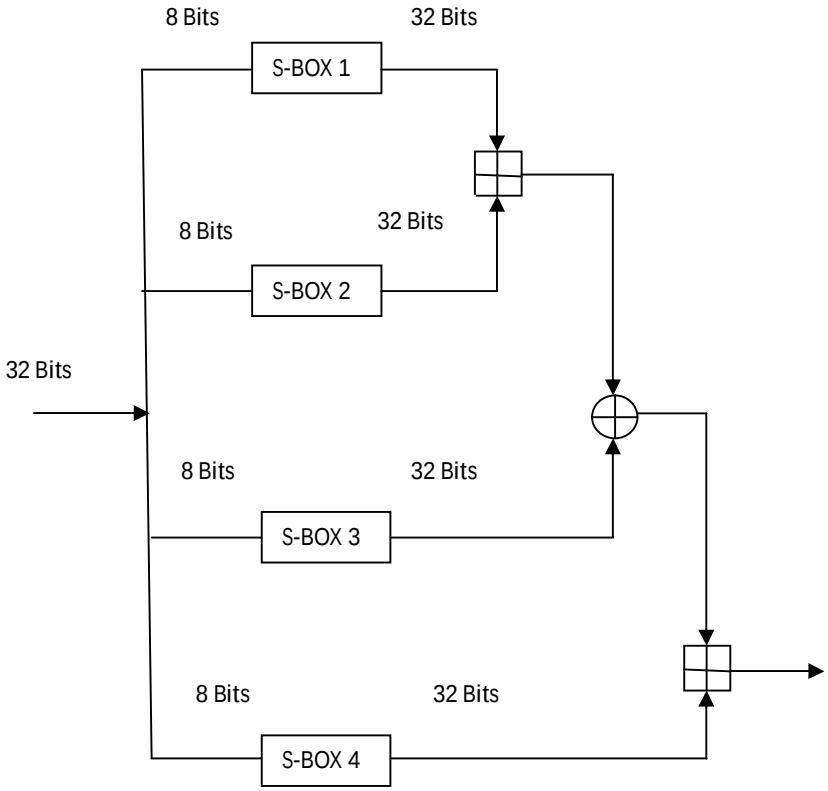

Figure 2.5: Function F in Blowfish Encryption

2.2.3 ADVANCED ENCRYPTION STANDARD (AES)

AES was published as FIPS PUB 197 [NIST (2001)], intended to replace the DES for commercial applications. AES was selected by a public competition. Though all of the six finalists were judged to be sufficiently secure for AES, the final choice for AES was Rijndael based on the three selection criteria: security, cost, and algorithm characteristics. Rijndael [Daemen (2002)] uses a 128-bit block size and supports 128, 192, and 256 bit keys. It is not based on feistel cipher structure like DES, Blowfish and other symmetric block ciphers, rather based on the Square cipher that uses S-boxes (substitution), shifting, and XOR operations.

2.2.4 XEX-BASED TWEAKED CODEBOOK MODE WITH CIPHERTEXT STEALING (XTS)

XEX-based **T**weaked codebook mode with ciphertext **S**tealing (XTS) with AES cipher [IEEE (2008), Dworkin (2009)] provides more protection than the other approved confidentiality-only modes against unauthorized manipulation of the encrypted data. It protects storage devices from chosen ciphertext attack (CCA) and copy & paste attacks [Ball et. al. (2012), Liskov and Minematsu (2008)]. Thus, XTS mode is suitable choice for encrypting data stored on hard disks where there is not additional space for an integrity field. It also provides random access to encrypted data. It can also be implemented as parallel algorithm. Parallel implementation of XTS-AES algorithm is 90% more efficient than the serial algorithm [Alomari (2009)].

The requirement of encrypting data stored on disk is different from transmitting encrypted data. The IEEE P1619 standard was designed to fulfil the aforementioned requirements with the following characteristics:

- The encrypted data is available for an attacker. There are some circumstances that may lead to this situation like:
  - There is a group of authorized users who have access to a database. They store their records in the database in which some of the records are encrypted so that only specific users can read/write them. Other users only have access to encrypted records but unable to read and modify with a specific key.
  - Somehow an adversary gains access to those encrypted records.
- File size does not change on the disk and in transit. The plaintext and the encrypted data must be of the same size.

- Each data block of fixed size is accessed independently. That is, an authorized user can access data blocks in any manner.
- Each block of size 16-byte is encrypted independently from other blocks (except the last two blocks of a sector if its size is not a multiple of 16 bytes).
- No other metadata is used, except the data blocks location.
- The same plaintext gives different cipher, when encrypted at a different location, but it gives same ciphertext when written in the same place again.

Before describing P1619, consider some of the existing modes of encryption. Consider CBC mode of encryption in which same plaintext is encrypted to different ciphertext at different locations. In this mode of operation, IV can be derived from the sector number. Each sector consists of N number of blocks. An unauthorized user with read/write permission to the storage media where encrypted data is stored can copy an encrypted sector from one position to another. It will produce the same plaintext when decryption is applied to the cipher text. An adversary can get plain text by using chosen cipher text attack. There is another flaw of this mode of operation is that an adversary user with read/write permission can modify the cipher text by flipping corresponding bits in the previous block.

Counter (CTR) mode of encryption where nonce and the counter is used for encrypting the each block, an unauthorized user with read/write permission can modify the plaintext by flipping the corresponding ciphertext bit.

2.2.4.1 XTS-AES Operation on a Single Block

The XTS-AES mode is based on the concept of a tweakable block cipher, introduced in [Liskov (2002)]. The form of this concept used in XTS-AES was first described in [Rogaway (2004)]. Figure 2.5 and 2.6 show the encryption and decryption of a single block. The encryption and decryption of a single block involve two instances of the AES algorithm with two different keys. The associated parameters are listed below.

Key: This key can be of 256 or 512 bit XTS-AES key. This key is the concatenation of two different keys of equal size, i.e. $key = key_1 \,||\, key_2$.

P_j: Here P stands for plaintext and subscript j stands for *jth* block in the plaintext. There will be a sequence of plaintext blocks on the disk like P_1, P_2, ... P_m. The size of each plaintext block will be 128 bits except the last block when file size is not multiple of 16 bytes.

C_j: Here C_j stands for ciphertext corresponds to P_j where j stands for *jth* block of plaintext. The size of each ciphertext block will be 128 bits except the last block when file size is not multiple of 16 bytes.

j: This is 128 bit long sequential number inside the data sector.

i: This is 128 bit nonnegative integer value used as tweak in the encryption. There is a different tweak value for each data unit or sector. This value starts from nonnegative random integer and assign consecutively to each data unit.

α: Corresponding to polynomial *x*, it is a primitive root of Galois Field (2^{128}).

α^j : α is multiplied *j* times by itself in GF(2^{128}).

The parameter *j* is used here same as counter in CTR mode. It guarantees that, if two same plaintext data block occurs at a different position in the same data unit, it will encrypt them into two different ciphertext. Here *i* act as the nonce for data unit. It guarantees that, if two same plaintext data block occurs at the same position in the different data units, it will encrypt them into two different ciphertext. It will also encrypt same data unit into a different ciphertext data unit at different position.

The encryption and decryption of a single block can be described as:

XTS-AES block operation	$T = \mathrm{E}(K_2, i) \otimes \alpha^j$ $PP = P \oplus T$ $CC = \mathrm{E}(K_1, PP)$ $C = CC \oplus T$	$T = \mathrm{E}(K_2, i) \otimes \alpha^j$ $CC = C \oplus T$ $PP = \mathrm{D}(K_1, CC)$ $P = PP \oplus T$

To check that, it is reversible operation, let us expand the last line of both encryption and decryption. For encryption, we have

$$C = CC \oplus T = E(K_1, PP) \oplus T = E(K1, P \oplus T) \oplus T$$

And for decryption, we have

$$P = PP \oplus T = D(K_1, CC) \oplus T = D(K_1, C \oplus T) \oplus T$$

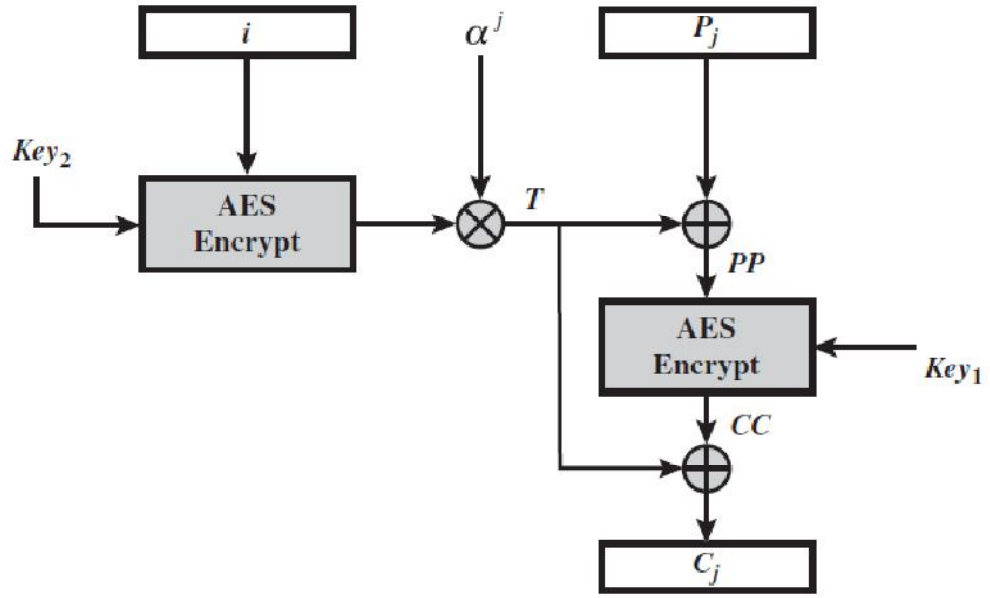

Figure 2.5: XTS-AES Encryption Operation on single block

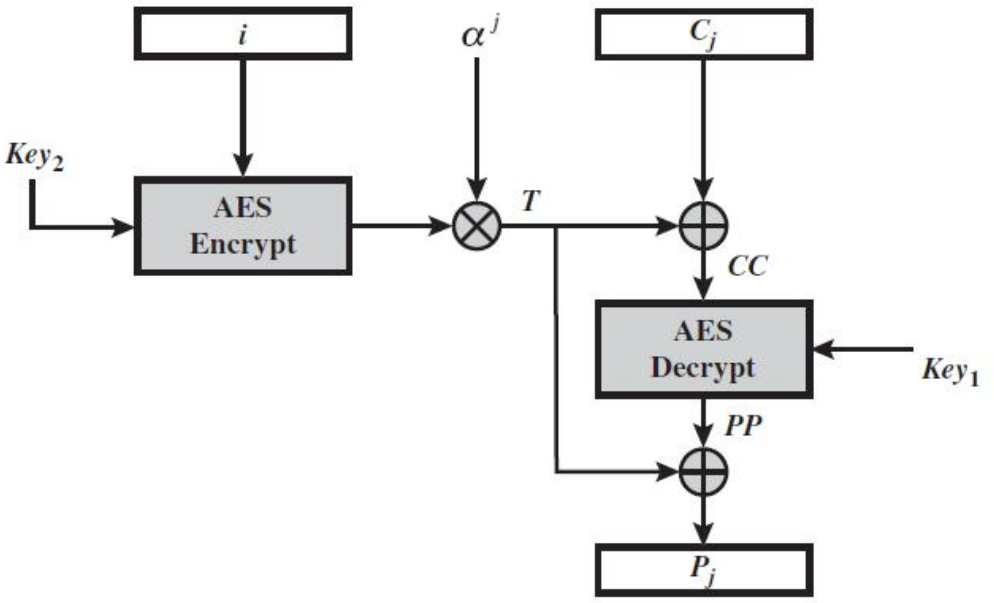

Figure 2.6: XTS-AES Decryption Operation on single block

Now, we substitute for C:

$$P = D(K_1, C \oplus T) \oplus T$$

$$= D(K_1, [E(K_1, P \oplus T) \oplus T] \oplus T) \oplus T$$

$$= D(K_1, E(K_1, P \oplus T)) \oplus T$$

$$= (P \oplus T) \oplus T = P$$

2.2.4.2 XTS-AES Operation on a Sector

A sector or data unit of plaintext is divided into blocks of 128 bits. Blocks are named as P_1, P_2, P_3… P_m. The size of the last block may vary from null to 127 bits. It means that, the input to the XTS-AES algorithm may contain m 16 byte block and last block as 1 to 127 bits.

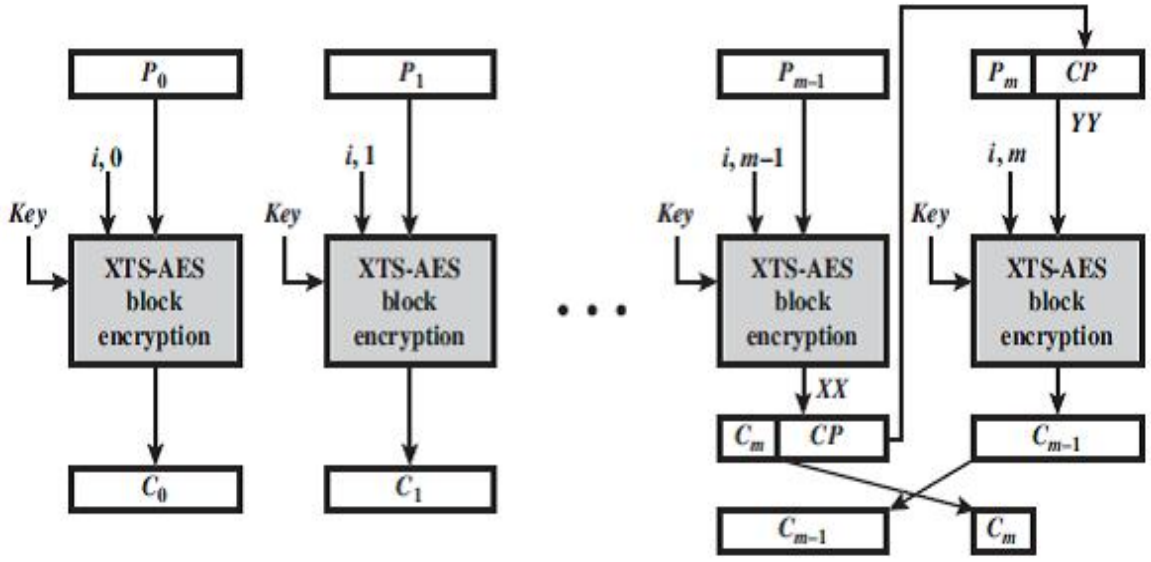

Figure 2.7: XTS-AES Encryption Mode

For encryption and decryption, each block of 128 bits is individually and independently encrypted or decrypted as you can see in Figure 2.7 & 2.8. There is only one exception when the last block is less than 128 bits. Here it uses ciphertext-stealing technique for encrypting/decrypting the last two blocks instead of padding. P_m and P_{m-1} are last and second last block of the data unit where P_m block contain $1 \leq s \leq 127$ bits. S is the number of bits presents in the last block. C_{m-1} is 128 bit ciphertext block, and C_m is the last ciphertext block, which contain s bits.

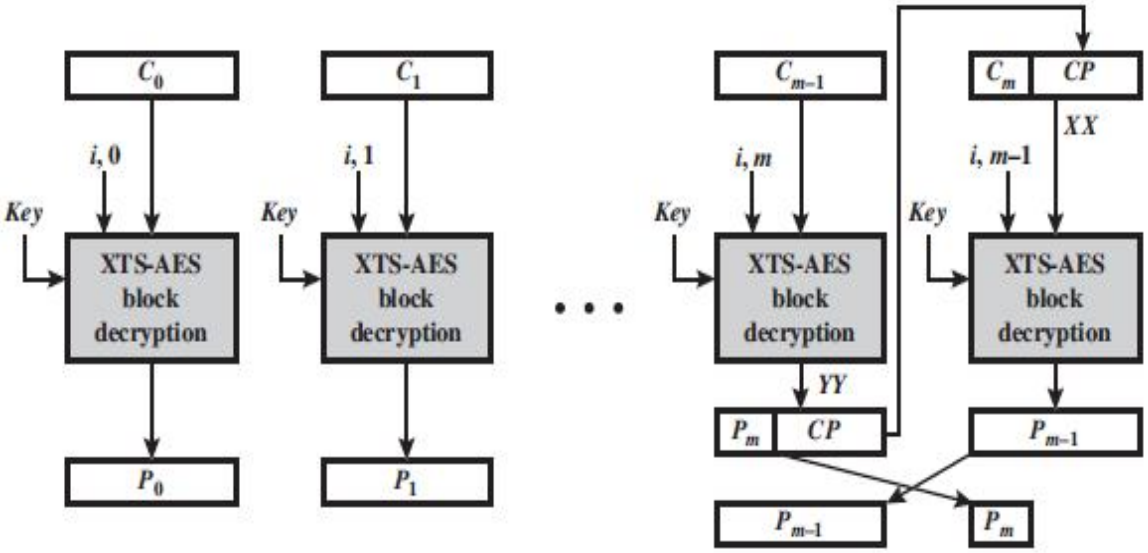

Figure 2.8: XTS-AES Decryption Mode

Let us named the block encryption and decryption as

Encryption Block: XTS-AES-blockEnc(K, P_j, i, j)

Decryption Block: XTS-AES-blockDec(K, C_j, i, j)

Then, if the last block is null, XTS-AES mode is defined as follows:

XTS-AES mode with null final block	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j) \quad j = 0, \ldots, m-1$
	$P_j = \text{XTS-AES-blockEnc}(K, C_j, i, j) \quad j = 0, \ldots, m-1$
XTS-AES mode with final block containing s bits	$C_j = \text{XTS-AES-blockEnc}(K, P_j, i, j) \quad j = 0, \ldots, m-2$ $XX = \text{XTS-AES-blockEnc}(K, P_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(XX)$ $YY = P_m \Vert CP$ $C_{m-1} = \text{XTS-AES-blockEnc}(K, YY, i, m)$ $C_m = \text{MSB}_s(XX)$
	$P_j = \text{XTS-AES-blockDec}(K, C_j, i, j) \quad j = 0, \ldots, m-2$ $YY = \text{XTS-AES-blockDec}(K, C_{m-1}, i, m-1)$ $CP = \text{LSB}_{128-s}(YY)$ $XX = C_m \Vert \text{CP}$ $P_{m-1} = \text{XTS-AES-blockDec}(K, XX, i, m)$ $P_m = \text{MSB}_s(YY)$

As it can be observed that, like CTR mode of encryption, XTS-AES mode can be used in parallel operation. Because encryption is done at the individual level independently so it can

encrypt and decrypt multiple blocks simultaneously. But XTS-AES is more effective because it uses one mode parameter *i* as nonce and *j* as a counter.

2.3 SURVEY OF EXISTING CRYPTOGRAPHIC FILE SYSTEMS

This section describes existing cryptographic file systems at the block device level and at file system level in user-space and kernel space.

2.3.1 CRYPTOGRAPHIC FILE SYSTEMS AT BLOCK DEVICE LEVEL

Block-based cryptographic file system operates below the file system level, encrypting one disk block at a time. They provide better performance and do not require knowledge of the file system that resides on top of them, and can even be used for swap partitions or applications that require access to raw partitions (such as database servers). They also do not reveal information about individual files (such as sizes and owners) or directory structure. These systems perform encryption with a single key on the entire block device, due to which they falter on various flexibility and security aspects. Also, they cannot be mounted by non-privileged users and cannot be used securely over the Network File System (NFS).

2.3.1.1 LOOPBACK CRYPTOGRAPHIC FILE SYSTEM (CRYPTOLOOP)

Loopback Cryptographic File System (Cryptoloop) [Hoelzer (2004)] is most well-known file system that performs encryption at the block device level. It is part of the 2.6 Linux kernel and uses the Linux kernel cryptographic framework, CryptoAPI [Cooke and Bryson (2003)] that exports a uniform interface for all ciphers and hashes. The Linux loopback device driver presents a file as a block device, optionally transforming the data before it is written and after it is read from the native file, to provide encryption/decryption. With Cryptoloop, the administrator can choose any cipher provided by CryptoAPI for file system encryption. The mount package on Linux distributions contains the losetup utility, which can be used to set up the Cryptoloop.

root# modprobe cryptoloop

root# modprobe blowfish

root# dd if=/dev/urandom of= /home/secret_dir bs=4k count=1000

root# losetup -e blowfish /dev/loop0 /home/secret_dir

root# mkfs.ext3 /dev/loop0

root# mkdir /mnt/unencrypted-view

root# mount /dev/loop0 /mnt/unencrypted-view

Cryptoloop uses the loop device as a pseudo device that allows each file system calls to be intercepted for encryption/decryption as shown in Figure 2.9. This implies that all system and process buffers remain encrypted. Cryptoloop encrypts the entire file system using a common mount-time passphrase due to which it has following disadvantages:

- The authentication is solely based on the passphrase and very susceptible to dictionary attacks.
- A single system wide key implies re-encryption of entire file system if one needs to change the key.

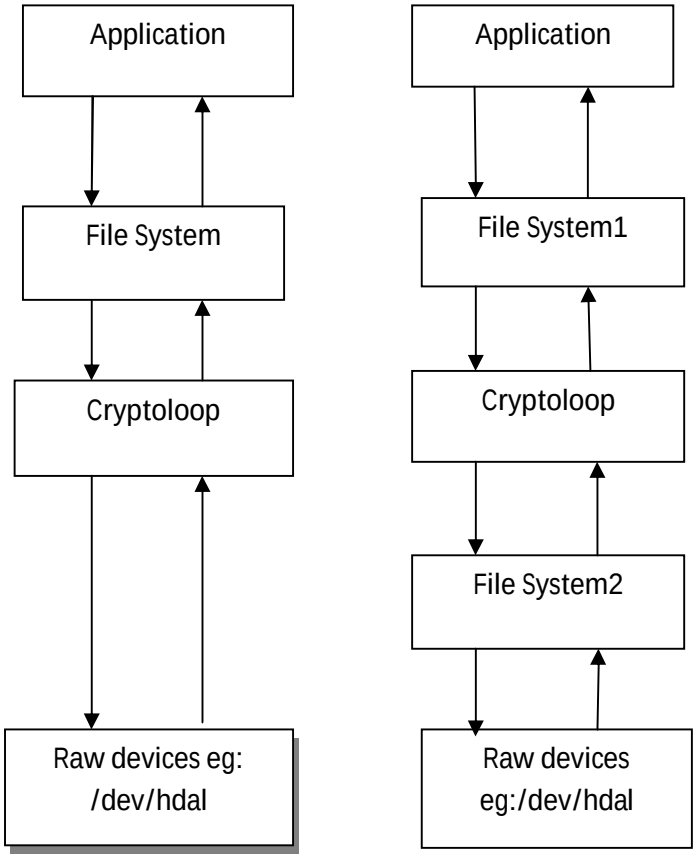

Figure 2.9: Cryptoloop stacked on top of a raw device (left) and a file (right)

- It is impossible for most standard backup utilities to perform incremental backups on the sets of encrypted files without being given access to the unencrypted files.
- No recovery is possible if the user forgets the password as there is no specific recovery agent.
- Sharing a set of encrypted files between different users is not possible.

The remote users will need to use IPSec or some other network encryption layer when accessing the files, which must be exported from the unencrypted mount point on the server. Cryptoloop is, however, the best performing cryptographic file system that is freely available and integrated with most GNU/Linux distributions.

2.3.1.2 DEVICE-MAPPER CRYPTO TARGET (DMCRYPT)

Device-Mapper Crypto Target (DMCrypt) [Fruhwirth (2005)] is the successor of Cryptoloop. DMCrypt is now the preferred method of encrypting at the block device level in the 2.6 kernel. Compared to Cryptoloop, DMCrypt provides several additional features like setting up initialization vectors and chaining modes. It provides greater compatibility with file systems, and decouples itself from the Logical Volume Manager (LVM) [Timme 2007]. It uses the same base architecture as Cryptoloop, so the advantages and disadvantages discussed with Cryptoloop are also applicable with dm-crypt.

Device mapper infrastructure in Linux kernel provides a generic way to create virtual layers of block devices. DMCrypt uses this to implement cryptographic operations using kernel Crypto API [Cooke and Bryson (2003)]. The user specifies the symmetric cipher, an encryption mode, a key and an initialization vector, and then creates a new block device in */dev* using dmsetup utility. The user then mounts a filesystem on a new block device created. Now all the writes to this device will be encrypted and reads decrypted. Using dmsetup utility is complicated, because the user has to remember all the parameters passed to the utility. The cryptsetup utility is created as a wrapper around dm-setup to make management tasks simpler as part of LUKS (Linux Unified Key Setup) project [Timme 2007]. LUKS for decrypt is an enhanced version of dmCrypt. It stores all necessary setup information in the partition header, enabling the user to transport or migrate his data seamlessly.

2.3.1.3 ENCRYPTED DISC DEVICE DRIVER (PPDD)

PPDD [Latham (2002)] is a block device driver for Linux that encrypts and decrypts data as it goes to and comes from another block device. It allows creating a device which looks like a disc partition. One can create an ext2, ext3 or ext4 filesystem or even a swap partition on this device. It uses blowfish cipher for encryption. In the 1.2 version of PPDD, which works with the 2.0 and 2.2 series of Linux kernels, the device driver is specific to PPDD. In the later versions which work with the 2.4 series of Linux kernels PPDD makes use of the loop device driver, like Cryptoloop [Hoelzer (2004)]. PPDD has not been ported to the 2.6 kernel.

2.3.2 USER-SPACE CRYPTOGRAPHIC FILE SYSTEMS AT FILE SYSTEM LEVEL

Cryptographic File System for Unix (CFS_Unix) [Blaze (1993), Blaze (1997)] and Encrypted File System (EncFS) [Ozen (2007), Gough (2011)] are two popular user-space cryptographic file systems at file system level. CFS_Unix is implemented as modified Network File System (NFS) server and EncFS by using the File System in User-space (FUSE) API [Szeredi (2004)]. They can be mounted by any user on the system and does not require any modifications to the kernel so can be easily portable. The limitation of these systems is their poor performance due to frequent context switches and data copies between user-space and kernel-space. They perform encryption with a single key on entire directory, so sharing of individual files is not possible among different users.

2.3.2.1 CRYPTOGRAPHIC FILE SYSTEM FOR UNIX (CFS_UNIX)

CFS_Unix **[**Blaze (1993), Blaze (1997)] is implemented entirely in user-space as a modified NFS server as shown in Figure 2.10. A userspace daemon, *cfsd*, acts as an NFS server running over the loopback network interface, lo and NFS client in the kernel makes RPC calls to the daemon. The CFS daemon performs transparent encryption/decryption of file contents during write and read operations. CFS_Unix presents a virtual file system, typically mounted on */crypt*, through which users access their encrypted files.

CFS_Unix provides a set of commands to the user for creating and accessing the encrypted directories. The *cmkdir* command is used to create encrypted directories and assign their keys. Its operation is similar to that of the Unix *mkdir* command with the addition that it asks for a key.

$ cmkdir /usr/cse/secret

Key: (user enters passphrase, which does not echo)

Again: (same phrase entered again to prevent errors)

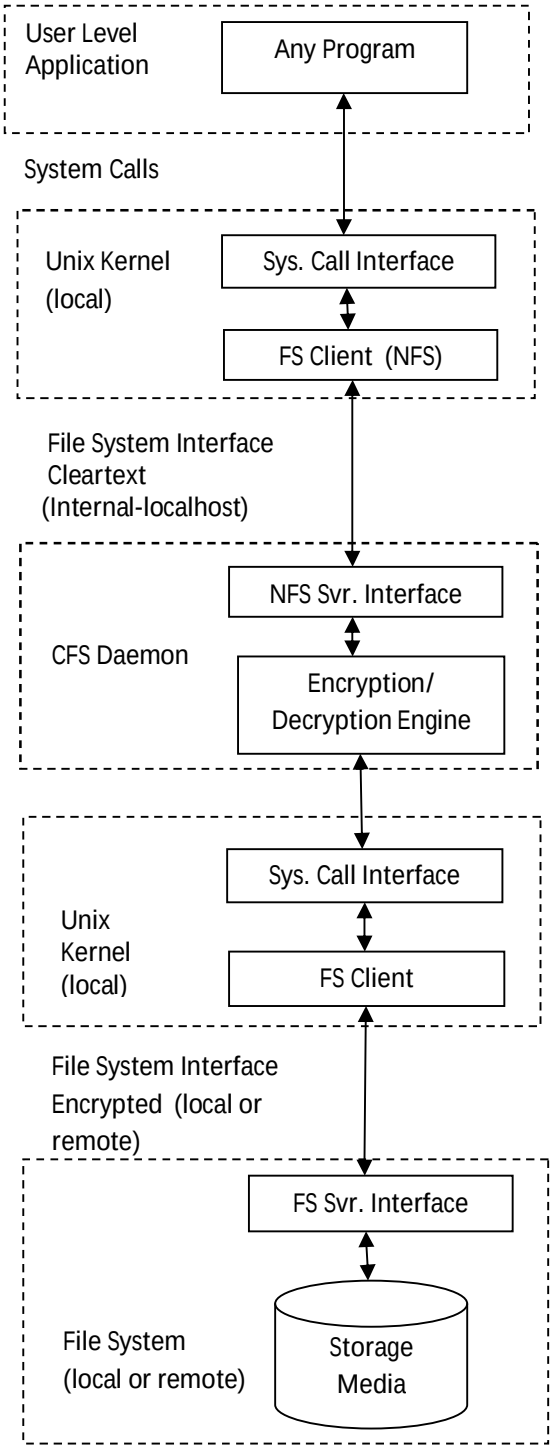

Figure 2.10: Data flow in CFS_Unix

To use an encrypted directory, its key must be supplied to CFS_Unix with the *cattach* command. *cattach* takes three parameters: an encryption key (which is prompted for), the name of a directory previously created with *cmkdir*, and a name that will be used to access the directory under the CFS mount point, that is */crypt*.

$ cattach /usr/cse/secret unencrypted

Key: (same key used in the *cmkdir* command)

The user can now switch to */crypt/unencrypted* and perform all standard file and directory operations (creating, reading, writing, compiling, executing, change directory etc.). The actual files are stored under */usr/cse/secret* in encrypted form. Note that any user on the system can make a new encrypted directory and attach it. It is not necessary to initialize and mount an entire block device, as in case of block device level cryptographic file systems explained in section 2.3.1.

CFS_Unix is capable of acting as a remote NFS server, but security issues with plaintext passwords and unencrypted data being transmitted over the network have to be considered [Halcrow (2004)]. CFS_Unix does not require any modifications to the kernel so can be easily portable. File system services such as backup, restore, usage accounting, and archival work normally on encrypted files and directories without the key. CFS_Unix ensures that cleartext file contents and name data are never stored on a disk or transmitted over a network. The limitation of CFS_Unix is its poor performance due to frequent context switches and data copies between user-kernel boundary and through the network stack. Also, it uses the DES algorithm for file encryption/ decryption, which further degrades its performance.

CFS_Unix uses the DES algorithm in OFB+ECB mode to allow random access to files but still discourage structural analysis. The passphrase provided at *cattach* command is used to derive two separate 56 bit DES keys. The first key is used to pre-compute a long (half megabyte) pseudo-random bit mask with DES's OFB mode. This mask is stored until user does not execute *dattach* command to unmount the */crypt/unencrypted* directory. When a file block is to be written, it is first XOR with the part of the mask corresponding to its byte offset in the file modulo the precomputed mask length. The result is then encrypted with the second key using standard ECB mode. When reading, the cipher is reversed in the obvious manner: first decrypt in ECB mode, then XOR with the positional mask.

2.3.2.2 ENCRYPTED FILE SYSTEM (ENCFS)

EncFS [Ozen (2007), Gough (2011)] is another popular user-space cryptographic file system for Linux, written using FUSE [Szeredi (2004)] library. FUSE has been integrated into the Linux kernel tree and provides a good way to write virtual file systems. FUSE exports all file system calls within the kernel to the user-space through a simple application programming interface (API), *libfuse,* by connecting to a daemon that is running in the user-space. In EncFS, this user-space daemon has been modified to perform transparent encryption and decryption of file contents during write and read system calls respectively as shown in Figure 2.11.

Once an EncFS volume is mounted, all file system calls targeting the mount point are forwarded to the FUSE kernel module, which passes it to user-space EncFS daemon. The user space file system functionality is implemented as a set of callback functions in the *libencfs* routines. Assume the directory */decrypted* in the underlying file system is chosen as the EncFS mount point. When an application issues a *read()* system call for the file */encrypted/file,* the VFS invokes the appropriate handler in EncFS. If the requested data is found in the page cache, it is returned immediately. Otherwise, the system call is forwarded over a character device, */dev/fuse,* to the *libfuse* library, which in turn invokes the callback defined in EncFS for the *read()* operation. The callback routine requests the data from the underlying file system, and then decrypt the read data. Finally, the result is propagated back by *libfuse*, through the kernel, and to the application that issued the *read()* system call. Other file system operations work in a similar manner.

A helper utility, *fusermount,* is provided to allow non-privileged users to mount file systems. The *fusermount* utility allows several parameters to be customized at the time of mounting the file system.

EncFS is portable as FUSE has ports available for other major operating systems. EncFS also has provisions to permit non-privileged users to mount the file system. FUSE provides an efficient userspace-kernel interface, so the performance of EncFS is somewhat better than CFS. EncFS can be used remotely, mounted on top of NFS. It also provides support for file integrity using keyed hashes. Both CFS and EncFS perform encryption with a single key on entire directory, so sharing of files is not possible among different users.

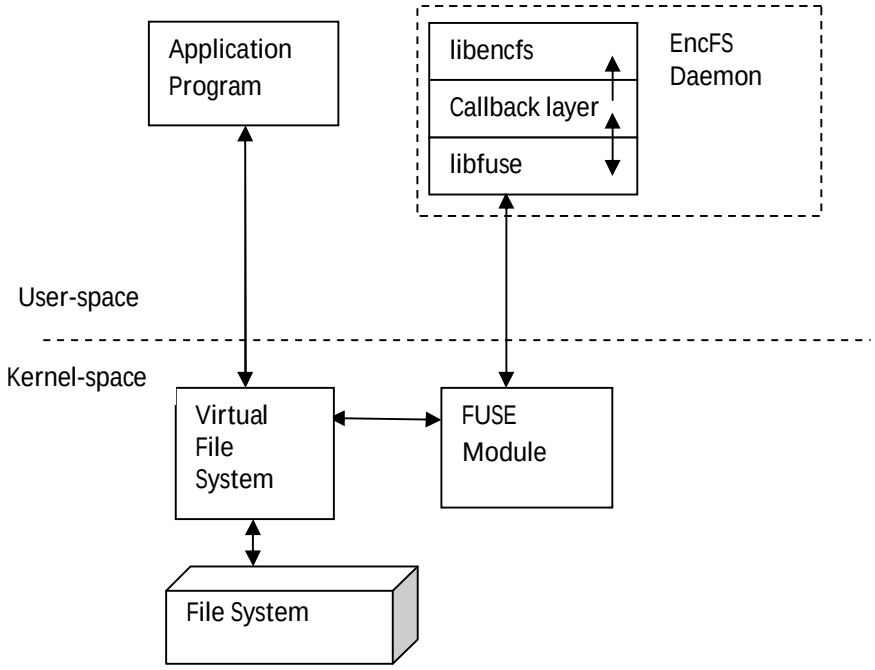

Figure 2.11: EncFS file system

Encfs uses algorithms from third-party libraries (OpenSSL is the default) to encrypt data and filenames. A user supplied password is used to decrypt a volume key, and the volume key is used for encrypting all file names and contents. This makes it possible to change the password without re-encrypting all files. EncFS uses both CFB stream mode and CBC block mode for encryption. The CFB stream mode is used for encrypting file names and partial blocks at the end of files. Fixed size file system blocks are encrypted using CBC mode. Each block has a deterministic initialization vector which allows for simple random access to blocks within a file. Data blocks are handled in fixed size blocks (64 byte blocks for Encfs versions 0.2 to 0.6, and user specified size in the newer versions of Encfs, default is 512 byte block). The block size is set during creation of the filesystem and is constant thereafter.

Message Authentication Codes can be optionally added to each block that will require 8 bytes per block and a large performance penalty, but makes it possible detect any modification within a block. Also during file system creation, one can enable per-file initialization vectors. This causes a header with a random initialization vector to be maintained with each file. Each file then has its own 64 bit initialization vector which is augmented by the block number, so that each block within a file has a unique initialization vector. This makes it infeasible to copy a whole block from one file to another.

2.3.3 KERNEL-SPACE CRYPTOGRAPHIC FILE SYSTEMS AT FILE SYSTEM LEVEL

Kernel-space cryptographic file system at file system level 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 systems are implemented using File System Translator (FiST) [Zadok and Nieh (2000), Zadok et al. (1999)], a tool that can be used to develop stackable file systems using template code. They are more efficient than existing user-space cryptographic file system at file system level, explained in section 2.3.2. They can be used remotely on top of networked file systems in a secure manner. These systems have a limitation that they cannot be ported across different platforms and cannot be mounted without administrator intervention.

2.3.3.1 STACKABLE VNODE LEVEL ENCRYPTION FILE SYSTEM (CRYPTFS)

Cryptfs [Zadok et. al. (1998)] is the stackable, cryptographic file system and part of the FiST [Zadok et al. (1999)] toolkit. Cryptfs was never designed to be a secure file system, but rather a proof-of-concept application of FiST [Wright et. al. (2003)]. Cryptfs supports only Blowfish cipher with CBC mode and does not use asymmetric keys. Cryptfs serves as the basis for several cryptographic file systems like Secure and Convenient Cryptographic File System (NCryptfs) [Wright et. al. (2003a)]; Secure and Transparent Encrypting File System for Enterprises (TransCrypt) [Sharma et. al. (2006)]; Enterprise-class Cryptographic Filesystem for Linux (eCryptfs) [Halcrow (2005), Kirkland (2011)], Hybrid Cryptographic File System (HCFS) [Kumar and Rawat (2011)]. Figure 2.12 shows Cryptfs operation. A user application invokes a system call through the Virtual File System (VFS). The VFS calls the stackable file system, which again invokes the VFS after encrypting or decrypting the data. The VFS calls the lower level file system, which writes the data to its backing store.

Encryption in Cryptfs is done by calculating the pages being affected by a write. This leads to encryption of more blocks than that are affected. Encryption layer is at page cache.

It encrypts at a file granularity. However, multiple files may use the same key. Cryptfs decouples encryption with ownership. The keys used are per-session. This means the users sharing the same keys have to use the same session. The keys are derived from a 16-character pass-phrase prompted when a user starts a new session for reading files.

Cryptfs can only be mounted by administrator and it has no recovery module specified.

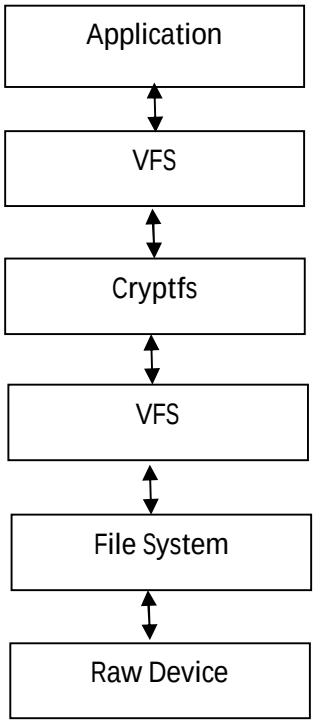

Figure 2.12: Data flow in Cryptfs

2.3.3.2 eCryptfs and TransCrypt

eCryptfs [Halcrow (2005), Kirkland (2011)] and TransCrypt [Sharma et. al. (2006)] are enterprise class cryptographic file systems based on Cryptfs. They use stackable file system interface approach to introduce a layer of encryption that can fit over any underlying file system, like Cryptfs. They support for file sharing among multiple users using public-key cryptography. They use per-file symmetric keys for encryption and user-specific public private-key pairs for user authentication, which enable fine-grained sharing. They encrypt each file with a symmetric encryption key and that key is encrypted with the public-key of the users who have authorized access to the file. eCryptfs has been integrated with the Linux kernel since 2.6.19. The overall architecture of eCryptfs is shown in Figure 2.13.

Public key certificate verification in eCryptfs is performed in user-space, while in TransCrypt, 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. The storage format makes it incompatible for sparse files.

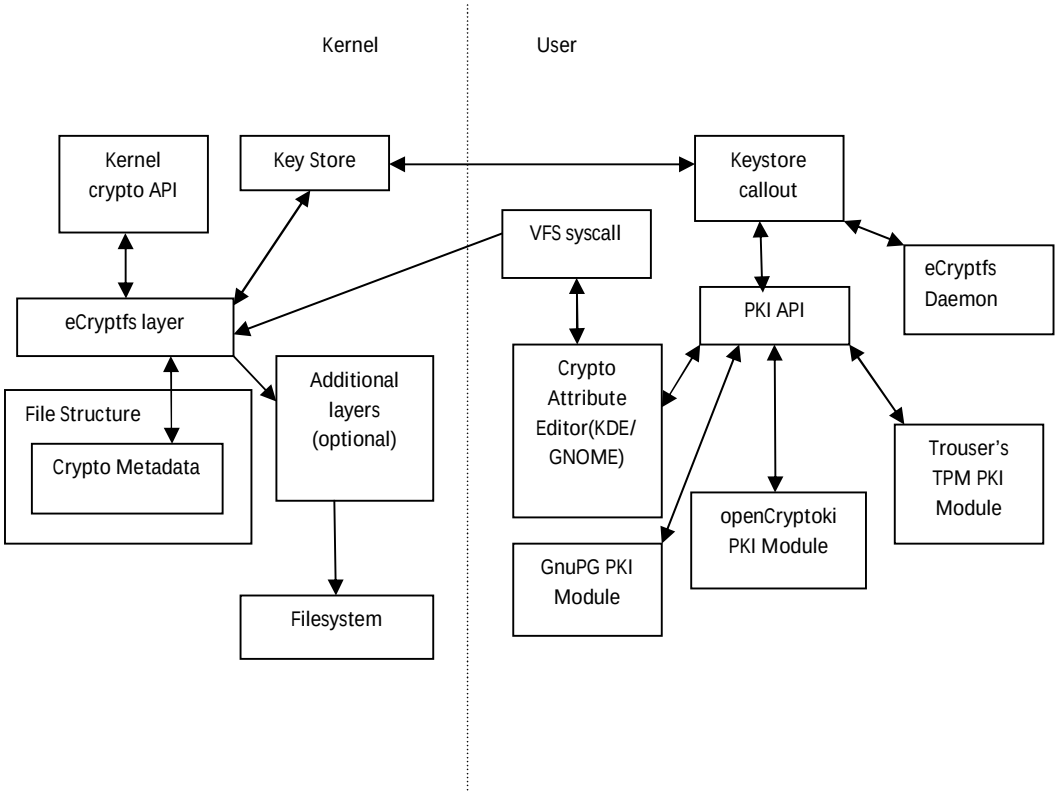

Figure 2.13: Overview of eCryptfs Architecture

TransCrypt stores cryptographic metadata as extended attributes in the file's access control list (ACL) [Grunbacher, (2003)] to ease file sharing task. eCryptfs can be used securely mounted on top of NFS, while in case of Transcrypt, secure protocol is required for its secure use over NFS [Modi et al. (2010)].

eCryptfs and Transcrypt uses CBC mode for file encryption with keyed hashes, like HMAC, for file integrity. They use kernel Crypto API for performing various cryptographic operations.

Since the data is encrypted using block cipher in CBC mode, random access would not be possible. For reading the very last byte of a file would require decrypting the entire file contents up to that point. Likewise, writing the very first byte of the file would require re-encrypting the entire file contents from that point.

To compensate for this particular issue while maintaining the security afforded by a cipher operating in block chaining mode, eCryptfs breaks the file into extents as shown in Figure 2.14. These extents, by default, span the page size. Data is dealt with on a per-extent basis;

any data read from the middle of an extent causes that entire extent to be decrypted, and any data written to that extent causes that entire extent to be encrypted.

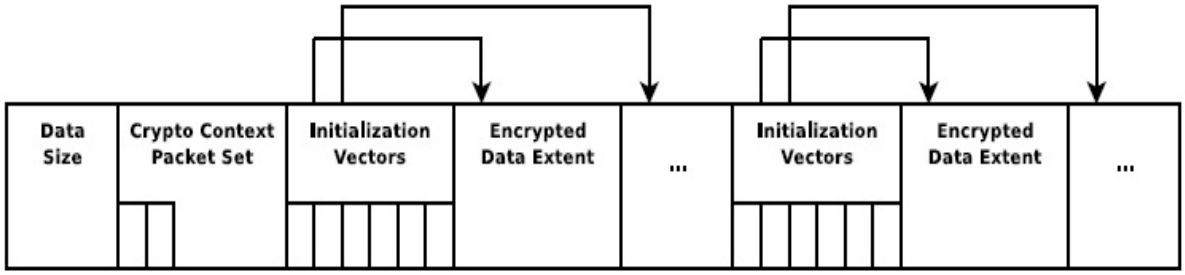

Figure 2.14: Underlying File Format of eCryptfs

Each extent has a unique initialization vector (IV) associated with it. One extent containing IV's precedes a group of the extents to which those IV's apply. Whenever data is written to an extent, its associated IV is rotated and rewritten to the IV extent before the associated data extent is encrypted. The extents are encrypted with the block cipher selected by policy for that file and employ CBC mode to chain the blocks.

2.4 TRUSTED COMPUTING IN CRYPTOGRAPHIC FILE SYSTEMS

Key management is a fundamental issue in the cryptographic file systems explained in the previous section. The majority of these cryptographic file systems employ only basic password protection schemes, disregarding the best practices of multi-factor authentication. Most passwords that users can reasonably expect to memorize can be successfully attacked with straightforward algorithms running on computing devices in present scenario. Token devices such as smart cards can be used for storing keys in some existing cryptographic file systems; however, the deployment of smart cards has been often prohibitively expensive, cumbersome, and error-prone. Security technology, such as trusted computing can be used for providing multi-factor authentication in a CFS without incurring additional cost.

The Trusted Computing Group (TCG) has proposed a Trusted Computing Platform (TCP) based on the Trusted Platform Module (TPM) cryptographic microcontroller system. The TCG is a not-for-profit organization that was formed to develop, define, and promote open standards for hardware-enabled trusted computing and security technologies. The TCP includes both hardware and software components. The functions provided by hardware

components are called TPM functions [TCG (2011)]; those provided by the software are called Trusted Software Stack (TSS) [TCG (2007)] functions. The TPM is a hardware chip that enables computers to achieve greater levels of security than was possible earlier. Since the year 2002, most computer manufacturers, including IBM, HP, Dell, Apple, have embedded a TPM chip in the hardware that regulates access to keys based on the state of the machine. The TPM therefore provides a hardware-based root of trust and contains the cryptographic functionality to generate, store, and manage cryptographic keys in hardware. The TPM works by measuring the bootloader, kernel, and other critical components of the machine, restricting keys to a strictly defined system state and releasing them only if the machine is booted into its trusted configuration. This approach helps to mitigate attacks that involve booting the machine from untrusted media [Ryan (2009)].

The TSS is the software required to manage the TPM chip. TrouSerS (http://trousers.sourceforge.net) and Trusted Computing for the Java(tm) Platform (http://trustedjava.sourceforge.net) are the most popular open source TSS packages available. TPM support for key management is provided by dmCrypt and recent versions of the eCryptfs cryptographic file system.

2.5 Summary of Cryptographic File Systems and Statement of Problem

Table 2.1 summarizes the properties of the various existing cryptographic file systems explained in previous sections. Block based cryptographic file systems have only performance advantage, so they are suitable only for single user environment. The primary focus of our research work will be on cryptographic file systems at file system level. As mentioned in Table 2.1, performance, file sharing, portability and availability to non-privileged users, all cannot be achieved together. Existing user-space cryptographic file systems at file system level have performance limitations and does not provide support for file sharing; and kernel-space cryptographic file systems are not portable and cannot be mounted by non-privileged users.

CFS_Unix is capable of acting as a remote NFS server, so it can be accessed remotely without requiring an additional NFS mount. This is, however, not recommended due to security issues

with plaintext passwords and unencrypted data being transmitted over the network, and also due to poor performance of CFS.

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.

Table 2.1: Summary of properties of cryptographic file systems

Features	**Block-based CFS (Cryptoloop, dmCrypt)**	**User-space CFS at file system level (CFS, Encfs)**	**Kernel-space CFS at file system level (eCryptfs, Transcrypt)**
Performance	Good	Poor	Good
Key Granularity	Common mount-wide key for whole file system	Common mount-wide key for a directory	per-file keys
Authentication using public keys	No	No	Yes
File sharing support	No	No	Yes
Cipher	Crypto API	DES in CFS, OpenSSL in Encfs	Crypto API
Encryption mode	CBC	ECB+OFB in CFS, CBC in Encfs	CBC
Integrity Support	No	Not in CFS, Supported in Encfs using keyed hash	Supported using keyed hash
Administrator Intervention	Required	Not required	Required
Portability	Not portable	Portable on any operating system	Portable on Linux based systems
Secure use over NFS	No	Not in CFS, possible in Encfs	Not in TransCrypt, possible in eCryptfs
TPM Support	Provided in dmCrypt	No	Provided in eCryptfs, Not in TransCrypt
Compatibility with underlying system services	Not Compatible	Compatible	Compatible

XTS-AES mode has not been implemented in any of the existing user-space or kernel-space cryptographic file systems at file system level.

The following work has been identified, after a critical analysis of existing cryptographic file systems and their properties:

- Design and implementation of user-space CFS, extending CFS_Unix [Blaze (1993), Blaze (1997)] and EncFS [Ozen (2007), Gough (2011)] cryptographic file systems, with performance improvements using faster ciphers and file sharing support.

- Design and implementation of secure protocol for CFS_Unix, using cryptographic methods such as mutual authentication and session establishment, which enables its secure use remotely.

- Design and implementation of kernel-space CFS, based on eCryptfs [Halcrow (2005), Kirkland (2011)], with improved performance using XTS-AES and inclusion of whole PKI support in the Linux kernel to exclude privileged user-space processes from domain of trust.

- Use of trusted computing technologies for key management in kernel-space CFS.