The Design of Convenient File Protection based on EXT3 File System

Ji-Ho Cho† Dong-Hoon Yoo† Hyung-Chan Kim† R.S.Ramakrishna†
Kouichi Sakurai‡

†Department of Information and Communications,
Gwangju Institute of Science and Technology,
1 Oryong-dong, Buk-gu
Gwangju 500-712, Rep. of Korea

jhcho@gist.ac.kr

‡Faculty of Computer Science and Communication Engineering
Kyushu University
6-10-1 Hakozaki, Higashi-ku, Fukuoka
Fukuoka 812-8581, Japan

sakurai@csce.kyushu-u.ac.jp

Abstract In this paper we present a secure file system for trusted operating systems (TOS). The proposed file system can protect data of a TOS from theft by encrypting them even though the given access control in TOS is exploited. We concentrate on balancing security and convenience with minimizing performance overheads. For security and convenience, the file system supports UNIX group sharing, meta-data protection, and transparency to users and application programs. In the aspect of performance, we minimize the overheads by implementing the proposed method on a native ext3 file system in a Linux operating system. Our experimental result shows that the proposed system is about 5 times faster than existing secure file system.

1 Introduction

Recent security systems reveal their limitations according as the methods of attacks have been diversified and elaborated. Trusted operating systems, in short TOS, are beginning to attract attentions as a promising ways to solve these limitations. A TOS is equipted with the basic security services and mechanisms to protect, distinguish and separate clssified data in a computer system.

Most researches of TOS are focused on enhancing access control mechanism. The purpose of access control is to limit the operations that a legitimate user of a computer system can perform. Access control constrains what a user can do directly, as well what programs executing on behalf of the users are allowed to do[1].

However, sometimes, access control can not assure the confidentiality and the integrity of files when the system is stolen or an attacker bypass the access control system. Also when an attacker acquires the privilege of system administrator or a system administrator abuses of his or her privilege, the access control can not preserve the security of the files.

Secure file systems are designed to solve these problems. A Secure file system protects files by encrypting them. However, the encryption at the a level is very cumbersome because the user of the system has to manage whole process of encryption, decryption and key management. Therefore, file system that manages the cryptographic process at the kernel level is more attractive.

There have been several approaches to design secure file systems[2, 3, 4]. However, the existing methods cause heavy performance overheads in addition to inconvenience for users. To overcome these problems, we propose a se-

cure file system that provides not only security for preserving privacy, but also convenience for user. Minimizing the overheads of performance is another goal of our proposed system.

The rest of this paper is organized as follows. Section 2 surveys previous and related works. Section 3 describes the design of our system. Section 4 discuss the operation of this system. We discuss current implementation state in Section 5. Section 6 shows evaluation of our prototype. Finally we conclude in Section 7.

2 Related Works

2.1 Cryptographic File System(CFS)

CFS based on NFS has been developed by Matt Blaze of AT&T Bell Lab [2]. It was implemented at a user-mode NFS server. User of this system has to create a directory at the local or remote file system to store encrypted file. CFS daemon is executed in a user mode. To access encrypted data, user should use a *attach* command. The critical disadvantage of CFS is performance loss caused by too frequently occurring context switches and data exchanges between kernel and user processes. In addition, it is hard to provide transparency to users and to deal with key management since the key should be managed by each individual user for each encrypted directory.

2.2 Tramsparent Cryptographic File System (TCFS)

TCFS is was developed in order to make up for CFS's defects[3]. Thus, it implanted at a kernel-mode NFS client. TCFS provides transparency to users without using *attach and detach* command. It is possible to encrypt each file and directory.

Database is used to store user keys and group keys. It is main problem of TCFS. The stored key is very vulnerable to attacks. In addition, applying TCFS for trusted operating system is unreasonalbe since it was developed for distribed enviroment. Therfore, if we use TCFS as a standalone system, its performance is unacceptable. Finally, TCFS is available only on system with Linux kernel version 2.2.27 or earlier.

2.3 NCryptfs

NCryptfs is a secure file system created to provide convenience and high performance [4][5]. NCryptfs is stacked on top of an existing file system. Calls accessing this directory through the NCryptfs mount are intercepted by the NCryptfs daemon. The daemon then accesses the file system and retrieves the file. NCryptfs will then decrypt the file based on a key supplied by the user. This key is stored in pinned memory and when the user access a file, NCryp tfs authorizes the user by getting a password. Like many system, this makes the security provided only as strong as the user password.

NCryptfs allows the user to use attachments. NCryptfs uses attachments in the same way as CFS. The use of attach facilitates sharing files, because a user could share an attachment with a group of users who all know the key. NCryptfs also supports UNIX groups. NCryptfs uses a cipher that will take an arbitrary sized buffer and encrypts it to a set size. NCryptfs also stores files in the file systems with hashes on their actual names to prevent analysis attacks.

The one piece of information that is not protected by NCryptfs is the directory structure, which is kept as is. Also it is still inconvenient because we have to use attach command to access encrypted files.

3 Design

3.1 Basic Ideas

Our design for the secure file system focuses on three main goals: the preservation of privacy, easiness in usage, and minimization of performance overhead. In sense of privacy, our system aims to protect data of the system from theft . Also it has to be safe from crackers bypassing access controls and system administrators abusing their privileges. To achieve this goal, when the system writes a file we encrypt it then store it on a disk. In case of

reading the file, we decrypt it then pass it to an application program.

Even though the file encryption provides safe environment to users, a few people use these kinds of systems since they are very cumbersome to use. In file encryption at application level, users should have reponsibility for setting configurations of key management, encryption and decryption.

Our system provides transparency to users and application programs by inserting encryption and decryption routines into at kernel level. In this system, only an administrator has responsibility for setting the configurations and users do not need to consider it. In addition, we improve the convenience by just using a *session* command instead of using attach and detach commands. Almost all cryptographic file system could not provide file sharing. On the other hand, our system provides file sharing to group members to be consistent with UNIX group sharing.

As depicted in section 1, CFS was implemented in user mode[2]. TCFS was implemented by using RPC because it has to support distribued environment[3]. Therefore, both two systems have a problem in the aspect of performance. To achieve better performance, we minimize overheads by implementing the proposed method on a native ext3 file system in a Linux operating system.

3.2 Key Management

Our system uses symmetric key cryptographic algorithms such as DES, AES, Blowfish, and so on. Each user has his or her own key. When users access the encrypted file, user should open session for establishing key then system promp ts the key from users. At this time,the key is not stored on disk, but saved on main memory. When the session is closed, key is destroyed from the memory. If user leaves his or her seat without closing the session or user do not use the computer for quite long time, session is automatically closed by timeout.

3.3 File Sharing

Our proposed system basically provides Linux group sharing. One different point is session concept. Only when the session of the owner of file is opened, the same group members of that file are allowed for access encrypted files. In other words, only when an owner of file's key is set on main memory and the group of owner and the group of the user that wants to access owner's file are the same, encrypted file is decrypted successfully. For these mechanisms, we use permission bits of inode.

4 Operation Scenarios

In this section, we present two operation scenarios in using our proposed scheme. One is when an owner uses his or her own file and the other is when a user who may or may not in same group with the file owner uses the given files.

4.1 Case 1: An owner

As depicted in Figure 1, if a user wants to access an encrypted file, the user is required to open the session for the establishment of key. User would open the session then system is inputted the key by user. Next, it stores that key on memory. After key is established by user, in case of reading the files, encrypted files are decrypted by established key. Then user could read plaintext. In case of writing the files, before the plaintext of files are stored files are encrypted by established key.

However, if the key is not established due to timeout or open fail of session our system provides cipher text to application without decryption. Thus user can not view of correct contents of the files. User may attempt to store the file without encryption, then system print out error messages since unencrypted files are violate the our goal.

4.2 Case 2: Group members

Here we assume that A is an owner of an encrypted file and B is a user who wants to access A's file, respectively. To access A's file, it is

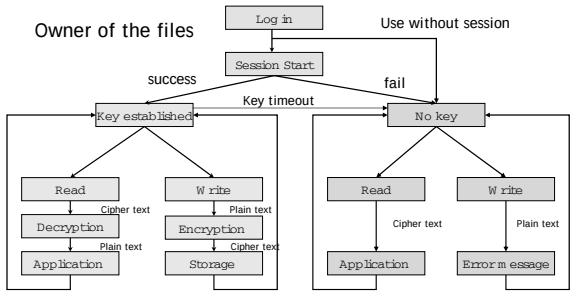

Figure 1: Flow of accessing a file by owner

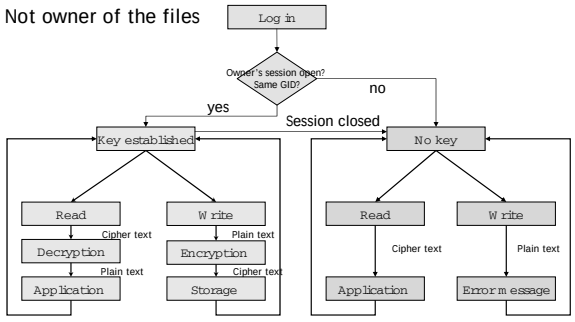

Figure 2: Flow of accessing a file by group members

required to satisfy two conditions by user B. Firstly, a session of user A has to be opened by the time of access. When the session of A is closed, the key is not available in anywhere, thus B can not encrypt the file of A. The second condition is dependent on the underlying access control in TOS. In case of generic Linux, the user A has to open the group read permission for the B under the DAC (Discretionary Access Control) scheme. The Figure 2 shows an instance of group sharing.

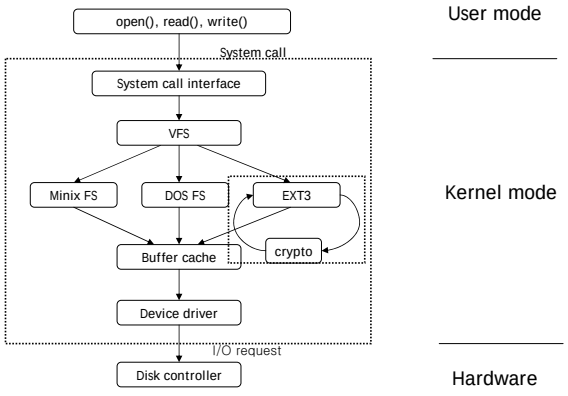

Figure 3: Overall Architecure

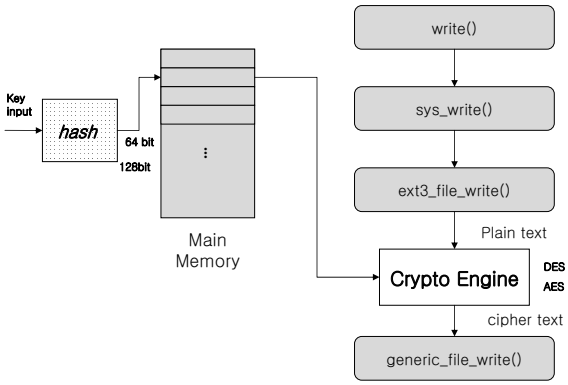

Figure 4: Encryption Process

5 Implementation

5.1 Encryption and Decryption

We implemented a prototype of our proposed system on Linux 2.4.27 modifying EXT3 file system. Although all system parts are not implemented, we implemented the core parts of the proposed system, that is encryption and decryption parts.

Overall architecture of Linux file system is presented in Figure 3. At the user mode, application program calls the file operation, then virtual file system call the system calls of specific file systems such as Minix, MSDOS, EXT2, EXT3, and so on. Next, each specific file system might access to disk through device drivers[6][8]. As you can see in Figure 3, our system interposes the encryption and decryption into EXT3 file system.

If the application program executes the wri te() instruction, then Linux system calls sys_wr ite() system call. Next, Almost all specific file systems deal with generic files call generic_file_ write(). Especially, EXT3 file system first calls ext3_file_write() for journaling then call generic_ file_write(). Figure 4 illustrates with a simple diagram of our internal structure. Our proposed file system calls generic_file_write() after the contents of file are encrypted by predefined key, . We used the symmetric key system as a cryptographic algorithm. Figure 5 is in case of reading. In this case, data flows opposite direction with writing. After reading a file from the disk using generic_file_read(), contents of file are decrypted by predefined key. As a re-

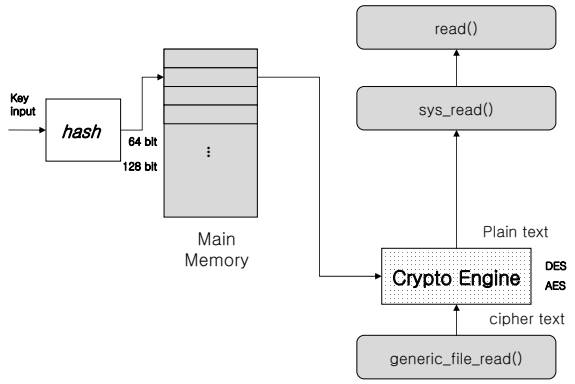

Figure 5: Decryption Process

sult, application can process a plain text data.

5.2 Cryptographic Algorithm

We use 'Scatterlist Cryptographic API' provided by Linux kernel as a our cryptographic API. This 'Scatterlist Cryptographic API' make a variety of cryptographic algorithms apply at the Linux kernel mode very easily. Our system is possible to provide various symmetric key algorithms such as DES, Triple DES, AES, and Blowfish. DES uses 64bits key and AES uses 128bits key[7]. To create a key with correct length, we convert a key from the user to hash value applying for hash function. Although it might cause a little overheads, it is safe from the cracking like a dictionary attack. Also, it provides convenient to user since users do not need to input a key with specific length.

6 Evaluation

In this chapter we analyze the encryption and decryption overheads. We compare original EXT3 file system with the proposed system using two types of data. In the first test we made 1024 different files of each 8KB size. In the second test, 8 files of 1MB size was made. Then we copy one 8KB file 1024 times, and one 1MB files 8 times, respectively.

Table 1 illustrates the performance of our system. X denotes a specific filesystem such as EXT3, EXT3 with AES, EXT3 with DES, and EXT3 with Blowfish. The test consists of

Table 1: Performance evaluation of the proposed system

1K byte X 1024				
X / Copy	EXT3	with AES	with DES	with Blowfish
From X to EXT2	0.131s	0.701s	0.755s	0.717s
From EXT2 to X	0.215s	0.673s	0.808s	0.788s
From X to X	0.017s	1.255s	1.440s	1.385s
1M byte X 8				
X / Copy	EXT3	with AES	with DES	with Blowfish
From X to EXT2	0.063s	0.630s	0.695s	0.657s
From EXT2 to X	0.125s	0.564s	0.714s	0.713s
From X to X	0.122s	0.743s	0.827s	1.306s

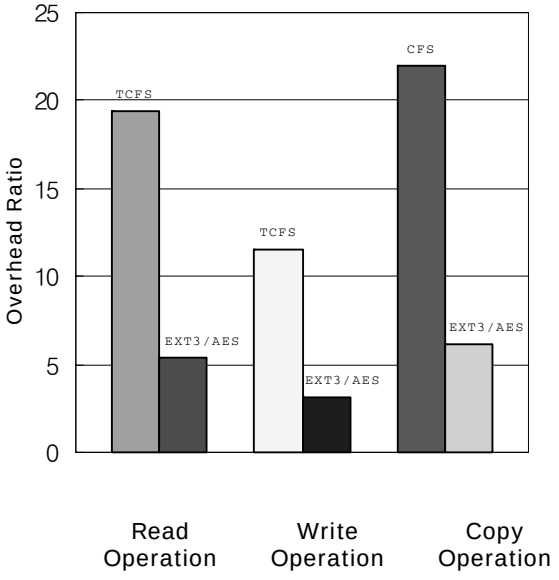

Figure 6: Comparison of overheads for each operation

three types of copy operations as follows:

1. From X to EXT2: the system copies the files from each X to EXT2. This test represents overheads of reading operation comparing with original EXT3.

2. From EXT2 to X: the system copies the files from EXT2 to each X. This test shows overheads of writing operation comparing with original EXT3.

3. From X to X: this test means overall overheads of each X.

All tests are performed on a Pentium 4 at 1.5 GHz with 512MB of main memory. As it is obvious from the experimental data, original

EXT3 is much faster than the others. The system that uses AES algorithm is fastest among three different systems with each cryptographic algorithm. As a result, we accept AES algorithm as our core cryptographic algorithm since it is very secure and fast.

Figure 6 illustates overheads ratio of our system compared with CFS and TCFS. We get experimental data from original papers of CFS[2] and TCFS[3]. In read operation, our system with AES is about 5 times slower than original EXT3fs while TCFS is about 19 times slower than original NFS. In write operation, our system with AES is about 3 times slower than original EXT3fs while TCFS is about 11 times slower than original NFS. If TCFS is compared with EXT2fs, it has heavy loss of performance since NFS has overheads itself.

In copy operation, our system with AES is about 6 times slower than original EXT3fs while CFS is about 22 times slower than original EXT2.

We believe that our system can be more improved but at this moment its overheads are still acceptable.

7 Conclusions

Our system aims to balance security and convenience with minimizing performance overheads. We achieved security by including cryptographic mechanisms. Especially our system is secure from theft and cracking. Even though system when an administrator wants to spy out user's private data, the system is highly secure.

Eliminating additional attach and detach command, we achieved convenience. Also our system provides Linux group sharing keeping security. Finally, we achieved high performance by designing cryptographic mechanisms to be run in the kernel. Our system has only cryptographic overheads.

We plan to implement other parts of our proposed system. Also we will encrypt important meta data that can give a clue to attackers for guessing original data.

Another possible feature of our system is integrity assurance. Currently, our system does not preserve integrity of data. Preserving integrity of data is very important for the system. Currently we consider digital finger printing or digital watermarking as an integrity checker.

References

[1] R. S. Sandhu and P. Samaratiy, "Access Control: Principles and Practice," IEEE Communications, 1993.

[2] M.Blaze, "A Crypographic File System for Unix.," In proceedings of the first ACM Conference on Computer and Communications Security, 1993.

[3] G. Cattaneo, L. Catuogno, A. Del Sorbo, and P. Persiano, "The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX.," In Proceedings of the Annual USENIX Technical Conference, FREENIX Track, pages 245.252, June 2001.

[4] C. P. Wright, M. Martino, and E. Zadok, "NCryptfs: A Secure and Convenient Cryptographic File System.," In Proceedings of the Annual USENIX Technical Conference, pages 197-210, June 2003.

[5] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A stackable vnode level encryption file system," Technical Report CUCS-021-98, Computer Science Department, Columbia University, June 1998.

[6] D. P. Bovet, and M. Cesati, "Understanding the LINUX Kernel," O'Reilly, 2nd Edition, 2003

[7] B. Schneie, "Applied Cryptography," JohnWiley & Sons, 2 edition, October 1995.

[8] R. Love, "Linux Kernel Development," Developer's Library, 2004