**SCIS 2005** The 2005 Symposium on Cryptography and Information Security
Maiko Kobe, Japan, Jan.25-28, 2005
The Institute of Electronics, Information and Communication Engineers

# XExt3: The Design and Implementation of a Security Enhanced Ext3 File System

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

**Abstract**— In this paper we develop an extended Ext3(XExt3) which means security enhanced file system. It can be used in generic Linux systems or trusted operating systems (TOS) as a file system. The XExt3 file system can protect data of a computer system from physical theft by encrypting them. We concentrate on balancing security, transparency and portability while minimizing computational overheads. For security and transparency, the XExt3 supports file protection, Linux group sharing, and the minimization of interactions between users and the system. 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. Finally we implement our system as a Linux kernel module for high portability. Our experimental results show that the XExt3 is about 3 or 4 times faster than previous cryptographic file systems.

**Keywords:** Trusted Operating System, File Systems, Linux, Cryptography, Ext3 File System, File Protection

## 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 attention as promising tools in this regard. A TOS are equipped with the basic security services and mechanisms to protect, distinguish and separate classified data in a computer system.

Most research on TOS is focused on enhancing access control mechanisms. 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 as what programs executing on behalf of the users are allowed to do[1].

However, sometimes, access control can not ensure confidentiality and integrity of files when the system is stolen or an attacker bypasses the access control system. Also, when an attacker acquires the privileges of a system administrator or when a system administrator abuses of his/her privileges, the access control mechanism is helpless.

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

There have been several approaches to design secure file systems[4, 5, 6]. However, the existing methods incur heavy computational overheads in addition to inconvenience to users. To overcome these problems, we propose a secure file system that provides not only security, but also convenience to the user. Minimizing the overheads is another goal of our system. Our experimental result shows that the proposed system is about 5 times faster than existing secure file systems.

The rest of this paper is organized as follows. Section 2 surveys previous and related works. Section 3 describes the design of XExt3. We discuss current implementation state in Section 4. Section 5 shows evaluation of our prototype. Finally we conclude in Section 6.

## 2 Related Works

### 2.1 Cryptographic File System

Cryptographic File System, in short CFS, based on NFS has been developed by Matt Blaze of AT&T Bell Lab [4]. 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.

* Department of Information and Communications, Gwangju Institute of Science and Technology (GIST), 1 Oryong-dong, Buk-gu, Gwangju 500-712, Republic of Korea
† Faculty of Computer Science and Communication Engineering, Kyushu University, 6-10-1 Hakozaki, Higashi-ku, Fukuoka 812-5851, Japan

### 2.2 Transparent Cryptographic File System

Transparent Cryptographic File System[5], in short TCFS, is was developed in order to make up for CFS's defects. 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 unreasonable since it was developed for distributed environment. Therefore, 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 [6][7]. 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.

It 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 of XExt3

In this section we discuss the design issues of XExt3 file system. Firstly, we will briefly introduce our design goals: security, convenience, performance and flexibility. Next we discuss the key management and group sharing mechanisms. Finally, we will explain the operation scenarios of two cases.

### 3.1 Design Issues

Although system designer should consider issues that more various aspect of system requirements, we will restrict our discussion to design issues that focused on security and practical usability.

- **Seurity**: Security consists of confidentiality, integrity, and availability. **Confidentaility** ensures that files and resources are accessed only by authorized users. That is, only those who should have access to something will actually get that access. By "access," we mean not only reading but also viewing, printing, or simply knowing that a particular data exists. Confidentiality is sometimes called secrecy or privacy[13]. **Integrity** means that files can be modified only by authorized users or only in authorized ways. In this context, modification includes writing, changing, changing status, deleting, and creating[13]. **Availability** means that files are accessible to authorized users at appropriate times. In other words, if some person or system has legitimate access to a particular set of objects, that access should not be prevented. For this reason, availability is sometimes known by its opposite, denial of service[13].

- **Transparency(Convenience)**: Software to secure data files is not in wide use today. One of the main reasons for this is that security software is not easy to use. Securing data files can not be done conveniently and transparently. For security software to become universal, it has to be more convenient. Final goal is that the encryption and decryption is completely transparent to user and application program[6].

- **Performance**: Performance is very critical issues for using secure file system. Many vendors have been minding using secure file system since almost all secure file systems have poor performance.

- **Portability(Flexibility)**: Flexibility is another very important requirement in designing secure file system. It means the ease with which a piece of system can be "ported".

### 3.2 Key Management

A key is basically provided by a user in our system. Almost all users can input any strings as their key value. Next, the key value is converted to a new string with correct length corresponding to cryptographic algorithms using one-way hash function. Our system mainly provides DES[8], AES[9], and Blowfish[10]. DES uses 64bits long string as a key and AES uses 128bits long key. The key of blowfish ranges from 32bits to 448bits. Finally, generated key is stored on key database. Figure 1 presents the whole process of key generation.

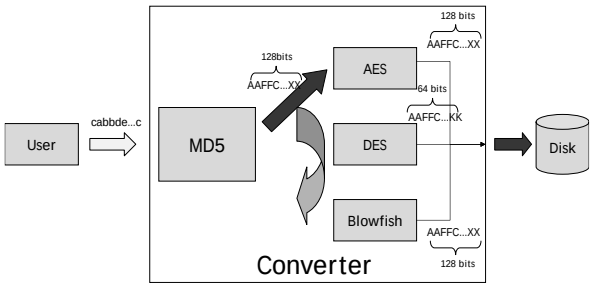


Figure 1: Key generation process

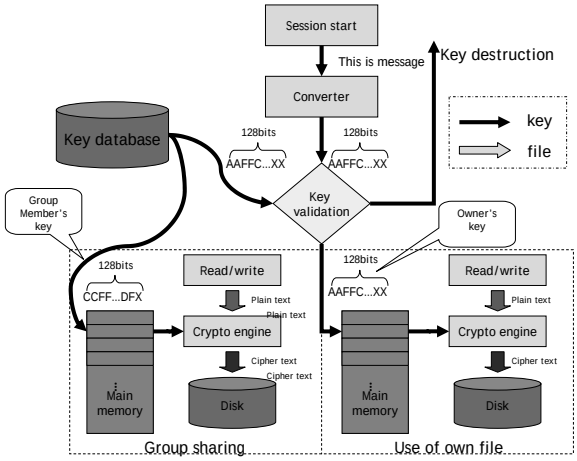


Figure 2: Session mechanisms

We use MD5 (Message-Digest Algorithm 5)[11] as a one-way hash function. In cryptography, MD5 is a widely-used cryptographic hash function that was designed by Ronald Rivest in 1991[12]. This algorithm converts any string to the 128-bit hash value. AES and Blowfish just use this hash value as a cryptographic key. We decide to fix the key length of Blowfish with 128 bits long for convenience of usage. In case of DES, we reduce the key size to 64 bits long.

### 3.3 Session Mechanisms

In our system, before the encrypted data is used, the owner must provide the key to the system. Then system validates whether the key is correct or not. Figure 2 shows two cases of session mechanisms. More detail concepts are following.

If user's key is correct the key is established on main memory while the session is opened. Keeping the key in kernel memory is more secure and faster than disk.

After using encrypted file system, user should close the session for further security. If user leaves his or her seat without closing the session or users do not use the computer for quite long time, session is automatically closed by timeout. When the session is closed, key is destroyed from the memory.

In conclusion, user can access only through session mechanism. User read not an original file but an encrypted file without active session for preserving confidentiality. Also, user do not allowed writing on protected partition without active session for preserving integrity.

Proposed system basically provides Linux group sharing. One different point is session concept. Only when the own session is open, access of the same group member's file is allowed. First, system validates the session key then group member's key established on main memory. 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. In case of writing, it is same as reading. For these mechanisms, we use permission bits of inode.

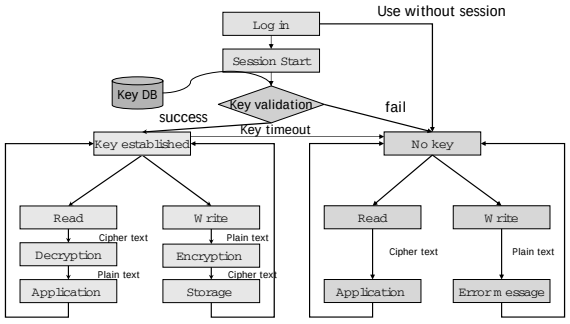


Figure 3: Flow of accessing a file by owner

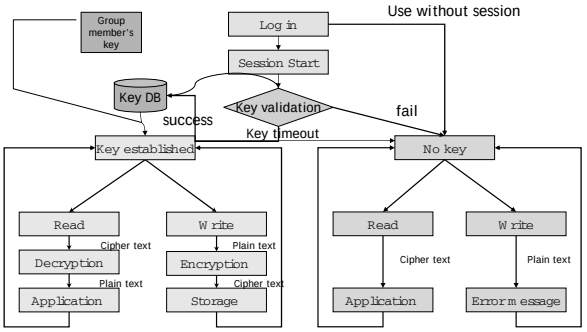


Figure 4: Flow of accessing a file by group members

### 3.4 Operation Scenarios

we present two operation scenarios in using proposed scheme. One is when an owner uses 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.

As depicted in Figure 3, 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.

Next case is when a user who may or may not in same group with the file owner uses the given files. Now we assume that A is an owner of an encrypted file and B is a user who wants to access A's file, respectively. First, user B start the session then system prompts for the key from user B. After B input own key, system validates the session key. Then group A's key established on main memory from key database. 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 4 shows an instance of group sharing.

# 4 Implementation of a Prototype

In this section, we discuss cryptographic algorithms that are used by our system. Next, we introduce overall architecture and layered architecture of our system. Finally we show encryption and decryption process in XExt3.

## 4.1 Selection of Cryptographic Algorithms

The following are the implementation issues that related to cryptographic algorithms.

- Symmetric-key algorithms such as DES, AES, and Blowfish are provided by kernel. Message digests algorithms like MD5 is provide in user mode. We use 'Scatterlist Cryptographic API' provided by Linux kernel as a cryptographic API in kernel mode. The 'Scatterlist Cryptographic API' makes a variety of cryptographic algorithms apply at the Linux kernel mode very easily. For using cryptographic algorithms in user mode we select a 'OpenSSL[15] ' API that full-strength general purpose cryptography library.
- Cryptographic process is time-consuming process. We focus on implement the encryption and decryption in the kernel to ensure that they are as fast as possible.
- The MD5 algorithm takes as input a message of arbitrary length and produces as output a 128-bit fingerprint or message digest of the input. We convert from user's key to its message digest using MD5 that provided by OpenSSL.

Our system is possible to provide various symmetric key algorithms such as DES, Triple DES, AES, and Blowfish. We decided that DES uses 64bits key and AES and Blowfish uses 128bits key in section 3.3.1. To create a key with correct length, we convert a key from the user to hash value applying for MD5. 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.

## 4.2 Overall Architecture

At the user mode, application program uses the system calls related to file operation such as open, read and write, then virtual file system calls specific file systems such as Minix, MSDOS, Ext2, Ext3, and so on. Next, each specific file system might access to disk through device drivers[14][16]. Data goes to buffer cache and goes to device drivers. Device drivers communicate with disk controllers. Our system interposes the encryption and decryption into Ext3 file system.

We implement our system using kernel module. In Linux system, primitive services such as interprocess

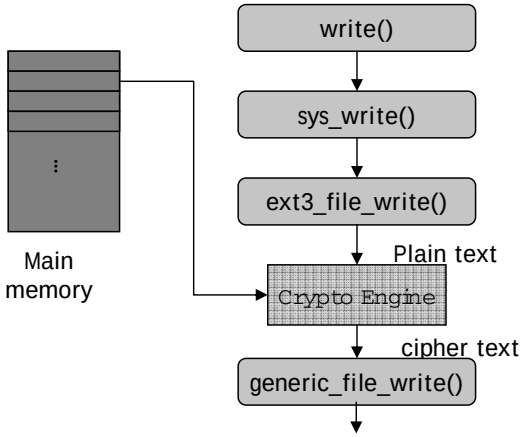


Figure 5: Encryption Process

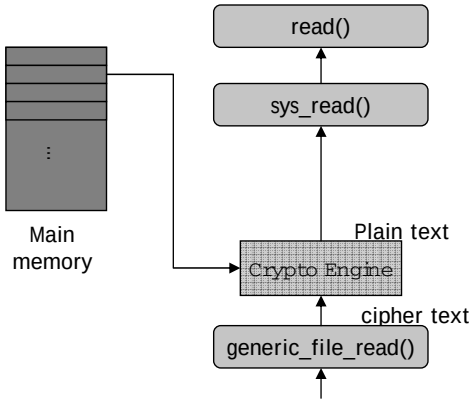


Figure 6: Decryption Process

communication, process management, systemcall handling, and memory management are provided by general kernel. On the other hands, system services such as file systems, device drivers, Crypto APIs and Networking are provided by kernel module. At this module layer we, we add a new secure file system module that is dynamically loaded and unloaded.

## 4.3 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 yet, we implemented the core parts of the proposed system, that is encryption and decryption parts.

Figure 5 illustrates with a simple diagram of our internal structure.

If the application program executes the write() instruction, then Linux system calls sys_write() system call. Next, Almost all specific file systems that deal with generic files call generic_file_write(). Especially, Ext3 file system first calls ext3_file_write() for journaling then call generic_ file_write(). Proposed file system

Table 1: Elapsed times of eah copy operation

| 1K byte X 1024 | | | | |
|---|---|---|---|---|
| Copy \ X | 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 | 0.117s | 1.255s | 1.440s | 1.385s |
| 1M byte X 8 | | | | |
| Copy \ X | EXT3 | with AES | with DES | with Blowfish |
| From X to EXT2 | 0.123s | 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 |

calls generic_file_write() after the contents of file are encrypted by predefined key. Finally, encrypted file is stored on disk by generic_file_write(). We used the symmetric key system as a cryptographic algorithm.

Figure 6 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. Then decrypted contents move to read() instruction through sys_read(). As a result, application can process a plain text data.

## 5 Performance Evaluation

In this section we analyze the encryption and decryption overheads. We experiment XExt3 on three types of copy operations. It shows the overheads of encryption and decryption processes. Next, we compare overhead of our system with previous cryptographic file systems. As a result, the XExt3 is faster than or equal to them.

### 5.1 Experimental Results

Table 1 illustrates the performance of our system. In this table, X denotes a specific file system such as Ext3, XExt3 with AES, XExt3 with DES, and XExt3 with Blowfish. XExt3 with AES means modified Ext3 by us using AES algorithm as its cryptographic algorithm. XExt3 with DES and XExt3 with Blowfish have same meaning. The test consists of 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 compared 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 compared with original Ext3.

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

We calculate the overhead of our system based on above two experimental data. We define overheads as a following equation.

$$Overheads = \frac{Time\ for\ copy\ operation\ in\ XExt3}{Time\ for\ copy\ operation\ in\ Ext3}$$

Table 2: Overheads of XExt3

| Copy \ X | with AES | with DES | with Blowfish |
|---|---|---|---|
| From X to EXT2 | 5.236548129 | 5.706882641 | 5.407372929 |
| From EXT2 to X | 3.821116279 | 4.735069767 | 4.68455814 |
| From X to X | 8.40832983 | 9.543190416 | 11.27126244 |

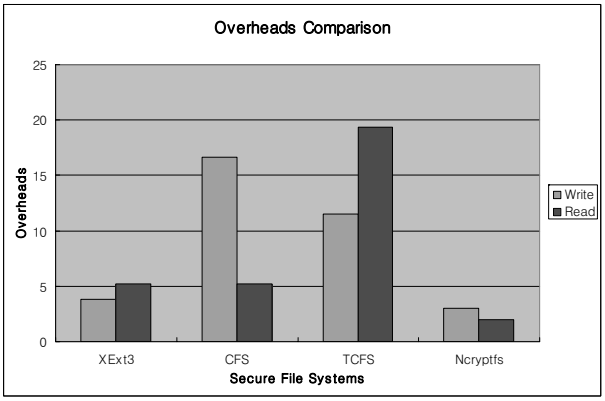


Figure 7: Comparison of overheads

Table 2 show the calculated overheads of EXext3. Our basis file system is original Ext3 file system. For example, when copy from Ext2 to original Ext3 takes 1 second, then copy from Ext2 to XExt3 file system with AES algorithms takes about 3.8 seconds.

From now on, we compare the overhead of XExt3 file system with previously developed cryptographic file systems such as CFS, TCFS, and NCryptfs. However, it is impossible that we draw a direct comparison among four secure file systems since each secure file system is executed in specific environments. For instance, TCFS is available only on system with Linux kernel version 2.2.27 or earlier. Thus, we take experimental data from original papers of each file system and then we calculate the overheads above equation. Figure 7 illustates overheads of our system compared with CFS , TCFS, and NCryptfs. In read operation, our system with AES is about 5 times slower than original Ext3 file system while TCFS is about 19 times slower than original NFS. CFS is about 5 times slower than original Ext2 file system. NCryptfs is about 2 times slower than Ext2 file system.

In write operation, our system with AES is about 3 times slower than original Ext3 file system while TCFS is about 11 times slower than original NFS. CFS is about 16 times and NCryptfs is about 3 times slower than original Ext2 file system. If TCFS is compared with Ext2 file system, it has loss of performance since NFS has overheads itself. Therefore TCFS has the heaviest overheads in compared file systems.

## 6 Conclusion and Future Work

We designed and implemented a secure file system for trusted operating system. Our system aims to balance security and transparency with minimizing performance overheads. Also, we should consider the portability issue.

We achieved security by hiring cryptographic mechanisms. Especially our system is secure from theft and cracking. Even though when an administrator wants to spy out user's private data, the system is highly secure.

By eliminating additional attach and detach commands, we achieved transparency. Also our system provides Linux group sharing while keeping security. We implemented our system as a kernel module for improving portability.

Finally, we achieved high performance by designing cryptographic mechanisms to be run in the kernel. Our system has only cryptographic overheads.

In this paper, the prototype of the XExt3 file system is implemented. We plan to implement other parts of our 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 check and recovery. Currently, our system does not check integrity of data. Recovering integrity of data is very important for a TOS. Currently we are considering digital finger printing or digital watermarking as an integrity checker.

## 7 Acknowledgement

This research was supported in part by Joint Forum for Strategic Software Research (SSR) of International Information Science Foundation, and in part by KAIST/GIST BK21 of Ministry of Education.

## References

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

[2] M. Bishop, "Computer Security: Art and Science," Addison-Wesley, 2002.

[3] C. E. Irvine, "The Reference Monitor Concept as a Unifying Principle in Computer Security Education," In proceedings of First World Conference on Information Security Education, IFIP TC11 WC 11.8, pages 27-37, 1999.

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

[5] 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.

[6] 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.

[7] 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.

[8] U.S. Department of Commerce, "Data Encryption Standard (DES)," January 22, 1988, FIPS PUB 46-1 (C13.52).

[9] J. Daemen and V. Rijmen, "The Design of Rijndael: AES - The Advanced Encryption Standard." Springer-Verlag, 2002

[10] B. Schneier, "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)," In Proceedings of Fast Software Encryption 1993, pages 191-204, 1993.

[11] http://userpages.umbc.edu/ mabzug1/cs/md5/md5.html

[12] R. L. Rivest, "The MD5 Message Digest Algorithm," Internet RFC 1321, 1992.

[13] C. P. Pfleeger, and S. L. Pfleeger, " Security in Computing,", Prentice Hall, 3rd Edition, 2002

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

[15] http://www.openssl.org/

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