

# Encrypt-FS: A Versatile Cryptographic File System for Linux

## Abstract

Recently, personal sensitive information faces the possibility of unauthorized access or loss of storage devices. Cryptographic technique gives a promising way to protect our files. In this paper we describe the design and implementation of Encrypt-FS, a high efficient and reliable cryptographic file system that provides dynamic data encryption and decryption at system level. Two cryptographic algorithms are used to encrypt a file: a symmetric cryptographic algorithm to cipher file's contents and a public key algorithm to encrypt the key that the symmetric cryptographic algorithm uses. File sharing and enhanced security are also included in Encrypt-FS. We evaluate its performance using Iozone and Bonnie benchmarks.

## 1. Introduction

Data sharing is playing a more and more important role in our society. At the same time, personal sensitive information faces the possibility of unauthorized access or loss of storage devices. How to trustily secure the data has become a difficult problem.

Cryptographic technique is a promising way to protect our files against unauthorized access. Nowadays people have developed so many useful cryptographic algorithms, from old DES (Data Encryption Standard) to recent IDEA (International Data Encryption Algorithm), AES (Advanced Encryption Standard) and etc. Some user-level tools (e.g. crypt program) based on these strong and speedy algorithms have come out to help users do the encryption and decryption routines, but they are not so convenient, not well integrated with the whole system and sometimes may be vulnerable to non-cryptoanalytic system level attacks.

Encrypt-FS tries to solve the above limitations by integrating cryptographic services in file system level. It is not designed to be a brand-new file system like Ext2, Ext3. Encrypt-FS works on Linux VFS layer, providing a cryptographic engine for dynamic encryption and decryption to upper processes. Therefore, user-level applications can work well without modification, and encrypted files can be stored anywhere in any directory hierarchy within any specific file system. Encrypt-FS uses two cryptographic algorithms to encrypt a file: a symmetric cryptographic algorithm like AES to cipher the file's contents, and a public key algorithm like RSA to encrypt the key that the symmetric cryptographic algorithm uses. This combinative approach could provide both reliable security and high performance.

## 2. Related work

### 2.1 For UNIX

There have existed several related projects for the UNIX operating system that offer transparent cryptographic protection for files or complete file systems, such as

Reiser4 [1], CFS [2], TCFS [3], FSFS [4] and Crypt-FS [5]. These file systems have more or less inherent limitations and security problems.

Reiser4 is a redesigned and excellent file system, which also provides encryption via file plug-ins. The main problem is that users who wish to use the cryptographic feature are confined to a specific file system.

CFS and TCFS offer encryption via a NFS client-server model. They may encounter the following issues:

- It is not so convenient to use them. In order to use an encrypted file, users have to first attach it, and then input the encryption key, and finally detach it. Attached files must be accessed from a special mount point (/crypt). Users have no choices on cryptographic algorithms as well.
- It is difficult to share an encrypted file. Only the creator can open, read and write the file. If the owner wishes to share this file among others, he has to give the encryption key to each one, since access will be denied without the key. This way of sharing sensitive files raises potential security issues. Everyone who knows the key has the same status as the legitimate owner and could enable additional users to access the protected files by just giving them the key. And if the owner has decided to remove a person from the allowing list, he has no better way but just changes the password and tells the new key to all the other users.
- There exists possible leakage of protected information. Because CFS and TCFS work above VFS layer, decrypted data in memory may leak out to swap space or temp files that reside on other unprotected directories. "Up to now, to maintain the security from a high level, it is necessary to encrypt all file systems of the machine. This makes it very difficult to use on multi-user machines where some directories are used by several users. All these users would have to share the same key making it quite useless to encrypt the data at all." [4]
- Degraded performance. Because of using a client-server model, it results in extra overheads and lowers the speed, especially for large files.

Crypt-FS (not cryptfs) has the following problems:

- Crypt-FS has an implementation error. When writing data, Crypt-FS does not judge whether the target pages have been decrypted and thus data may be messed.
- Crypt-FS does not provide the feature to share files with others either. And it also may encounters the leakage issue mentioned above.

## 2.2 For Windows

On Windows NT and above, users could encrypt their files using the built-in dynamic, efficient and powerful encryption engine, which is called EFS[6]. EFS is such a good model that we refer to its design and implementation. However EFS has two limitations:

- Only NTFS is supported.
- EFS uses DESX (an improved version of DES) and RSA combination. Users

could not specify the algorithms. And DESX may not be secure enough.

### 3. Design

#### 3.1 Design Goals

Encrypt-FS aims to protect users' private information that may be vulnerable to attack in a way that is convenient enough to use routinely. Thus, we propose the following specific goals for Encrypt-FS:

1. **Easy to use.** Users just need to decide which files should be protected, and Encrypt-FS will automatically deal the processes of creating keys, encrypting files and storing metadata. Subsequent access to those files is transparent, even without inputting keys. Of course, Encrypt-FS should also give an easy way to reverse encrypted files back.
2. **Powerful.** Encrypt-FS aims to provide enough and powerful functions. Users could choose different encryption algorithms to encrypt their files, share those files with other people, and specify how to recover files when they lost keys.
3. **Transparent access semantics.** Encrypted files behave no differently from other files. Current programs need not to be modified to access them. They should support the same access methods available on the underlying storage system too. All system calls should work normally, and it should be possible to compile and execute in a completely encrypted environment.
4. **Enhanced security.** Encrypt-FS should prepare well for the potential security issues. This involves key management, protection of contents and meta-data, exposing data unconsciously (swap, temp file, or memory dump) and etc. We must think much of security in order to protect users' information really and truly.
5. **High performance.** Though we must encrypt and decrypt contents dynamically when accessing encrypted files, the overhead should be low enough, especially to large files.

#### 3.2 Architecture

Since we will use two cryptographic algorithms to encrypt a file, 2 kinds of encryption keys exist in Encrypt-FS: FEK and UEK.

##### (1) FEK

Symmetric encryption algorithms are used to encrypt files in order to gain both security and performance. Available algorithms are DES, 3DES, AES, IDEA and etc. We prefer AES algorithm and use a 128-bit key by default. Certainly users can choose any one algorithm from the above list and specify the key length. This key that we use is called **FEK** (File Encryption Key). FEK is randomly generated by Encrypt-FS. FEK should also meet the length and other requirements of the selected algorithm. We must be careful not to generate a weak FEK.

Each encrypted file has an exclusive and different FEK. That is for security. Even if an attacker is lucky enough to obtain the FEK of one file, it is not possible for him to guess other FEKs. Therefore other encrypted files are still safe. Further more, we never automatically change the FEK during the lifetime of the encrypted file.

## **(2) UEK**

Since encrypted files may be shared among several users, it is not wise to tell the FEK to each person. A better way is to encrypt the FEK with a Public Key Algorithm, and store it with the file itself. A user will have a pair of keys: a public key and a private key, and we call them UEK (User Encryption Key). Encrypt-FS of course could generate the pair of keys for users. We prefer RSA algorithm, and use a 1024-bit key by default. When the owner wants to share the encrypted file with a person, he firstly imports the public key of that user's UEK, then encrypts the FEK, and finally inserts the encrypted FEK into the Decryption Key Ring--a special area of the file meta-data. Later on, when that user accesses the file, Encrypt-FS will firstly get the user's private key, finds the proper item in the Decryption Key Ring, decrypts the FEK, and finally uses the FEK to decrypt the contents of that file.

## **4. Implementation**

Encrypt-FS is implemented on Linux kernel 2.4. When a file is encrypted for the first time, Encrypt-FS saves related meta-data. We modifies several system calls to judge whether an opened file is encrypted, loads saved meta-data and later on provides dynamic encryption and decryption when needed.

### **4.1 Meta-data**

The meta-data describes all the encryption information associated with a file. It includes Encryption Head, Decryption Key Ring, Recovery Key Ring and UEK. All except UEK are stored as the file's extended attributes directly.

#### **(1) Encryption Head**

Encryption Head holds the basic encryption information, and it has the following items:

- MD5 hash value. It verifies the integrality.
- Version number.
- The algorithm chosen for encrypting file contents. By default it is AES.
- FEK length. By default it is 128-bit.
- The size of the Decryption Key Ring that shows how many key items are stored in this ring.
- The size of the Recovery Key Ring

#### **(2) Decryption Key Ring**

Decryption Key Ring contains an array of encryption results of the same FEK, and each item consists of:

- MD5 hash value. It verifies the integrality of this structure.
- The user's ID that tells who this encrypted FEK belongs to.
- The public key of the user's UEK. This protects wrong private key from being imported to decrypt the FEK.
- The encrypted FEK.

### **(3) Recovery Key Ring**

Recovery Key Ring also contains an array of encryption results of the same FEK. This helps the owner to recover his encrypted file by using a previously designated account when all the allowed users unfortunately could not access the file (rare but possible).

### **(4) UEK**

UEK plays a vital role in Encrypt-FS, and if the private key leaks out, the whole system breaks. We encourage saving UEK to Smartcards or other physically secure places. Currently, we encrypt the private key of UEK using user's login password, and then place it in directory `~/.encryptfs`. We also apply special permission using ACL to make sure that nobody can read the private key file. Surely it is not that safe, so we will improve it in future versions.

## **4.2 Management of encrypted files**

Only the file's owner can do things like encrypting and decrypting, adding another user in the share list, or showing status of the encrypted file and etc. We provide a command line program named `cipher` to do the work.

Here we mention how to the process of encrypting a file for the first time:

- (1) Generate a random key using built-in Kernel generator for later encryption.
- (2) Create a working temp file (`.filename.encryptfs`) for saving encrypted contents. Thus the original file will not be destroyed even if the encryption process fails.
- (3) Read data in chunks sequentially, encrypt the chunk and then write it to the temp file.
- (4) After encryption, import owner's public key of UEK to encrypt the FEK and set up the Encrypting Head and Decryption Key Ring.
- (5) Store the meta-data to the temp file as extended attributes in "system" and apply an exclusive access right using ACL that denies access from other users.
- (6) Delete the original file and rename the temp file.

## **4.3 Opening an encrypted file**

We modify normal access procedure for accessing an encrypted file. Extra steps consist of:

- (1) Read the "Encryption Head" attribute and verify its integrality.
- (2) Read the "Decryption Key Ring" and find the item for the current user. We must also verify the integrality of the encrypted key structure.
- (3) Import current user's private key of UEK and try to decrypt the FEK.

- (4) Save FEK and other information for later operations.

If any step fails, we reject the access and return an error to the user.

#### 4.4 Dynamic encryption and decryption

Compared with CPU, hard disks are too slow. Thus Linux uses page cache to speed up the reading and writing process. If the page cache stores cryptograph, the data has to be encrypted or decrypted every time when it is accessed. This may greatly slows the whole system especially for accessing large files. Thus Encrypt-FS tries to store decrypted data in page cache long as possible and do encryption only when a buffer is actually written back to disks. We will see that this approach gains considerable performance. The following explains when to encrypt and decrypt data:

- (1) Data decryption is done in function `do_generic_file_read` and `do_generic_file_write`. We first judge whether the target pages (or individual buffers) have been decrypted. If not, we decrypt those pages (or buffers) and mark them decrypted. Of course, writing a full page does not need decryption at all. From now on, subsequent access does not need decryption any more (if they are still marked decrypted).
- (2) Data encryption is done in function `generic_make_request`. `generic_make_request` sends actual requests to lower disk drivers, and thus is a good place to encrypt the buffer. Then we mark this buffer encrypted.

#### 4.5 Prevention from leakage

As mentioned above, memory pages store decrypted data nearly all the time. It is necessary to protect the sensitive data from leaking out to swap space and temp files. Because we make sure to encrypt blocks before writing to disks in function `generic_make_request`, we can solve this security issue. When they are brought back to physical memory, Encrypt-FS will decrypt them on next access.

### 5. Performance Evaluation

Iozone and Bonnie tests were run to evaluate the performance. Our test system is:

CPU	Memory	Hard Drive	OS	Kernel	Target FS
Athlon64 2800+	1G DDR	Maxtor 80G	Fedora 1(x86)	2.4.21 [7]	Ext3

In both of the tests, we consider the following three conditions:

- No Encrypt-FS support. We did not compile the Encrypt-FS into the kernel. This shows the best performance.
- Encrypt-FS + Regular. Encrypt-FS was compiled into the kernel, but a non-encrypted file is used in the tests.
- Encrypt-FS + 128bit AES. Encrypt-FS was compiled into the kernel, and test file is encrypted using 128bit AES algorithm.

## 5.1 Iozone Benchmark

Iozone is an excellent file system benchmark tool. This benchmark generates and measures a variety of file operations. Iozone has been ported to many machines and runs under many operating systems. We conducted Read/Reread, Write/Rewrite and Random Read/Random Write tests and figure 1-6 show the results:

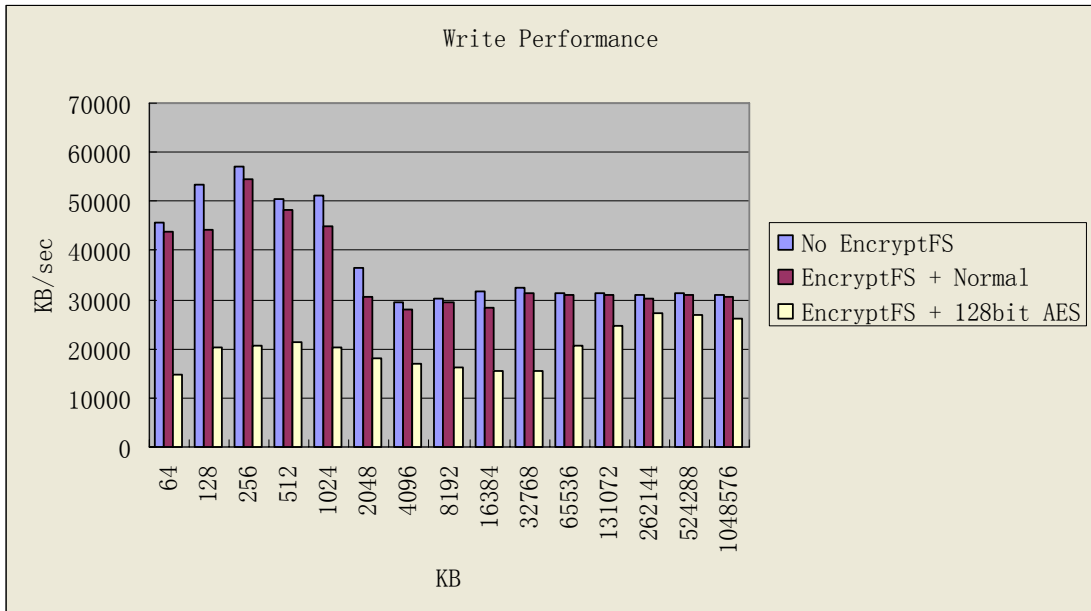


Figure 1. Write Performance

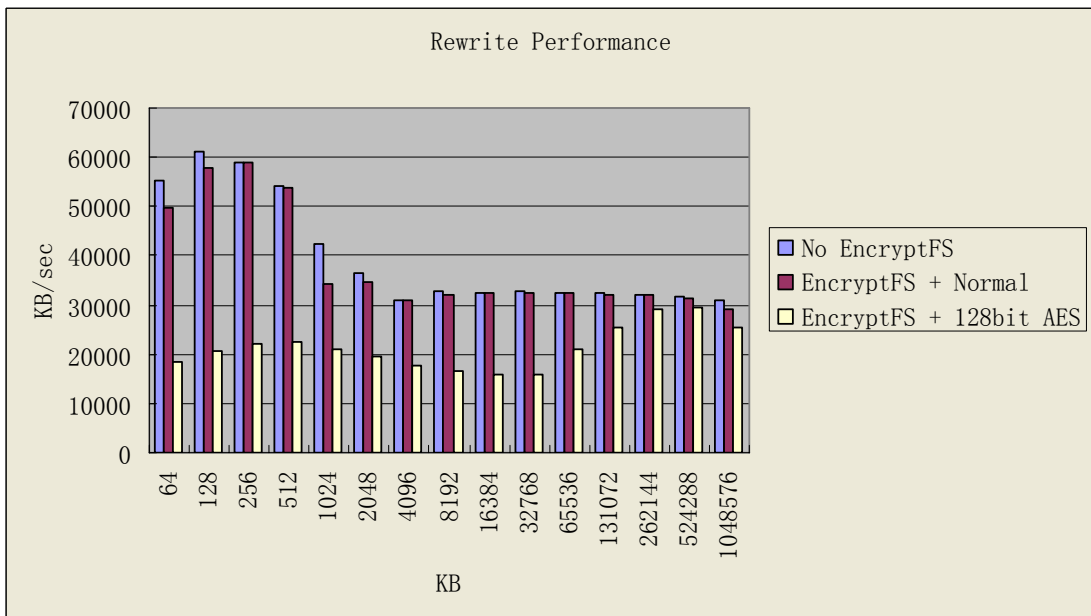


Figure 2. Rewrite Performance

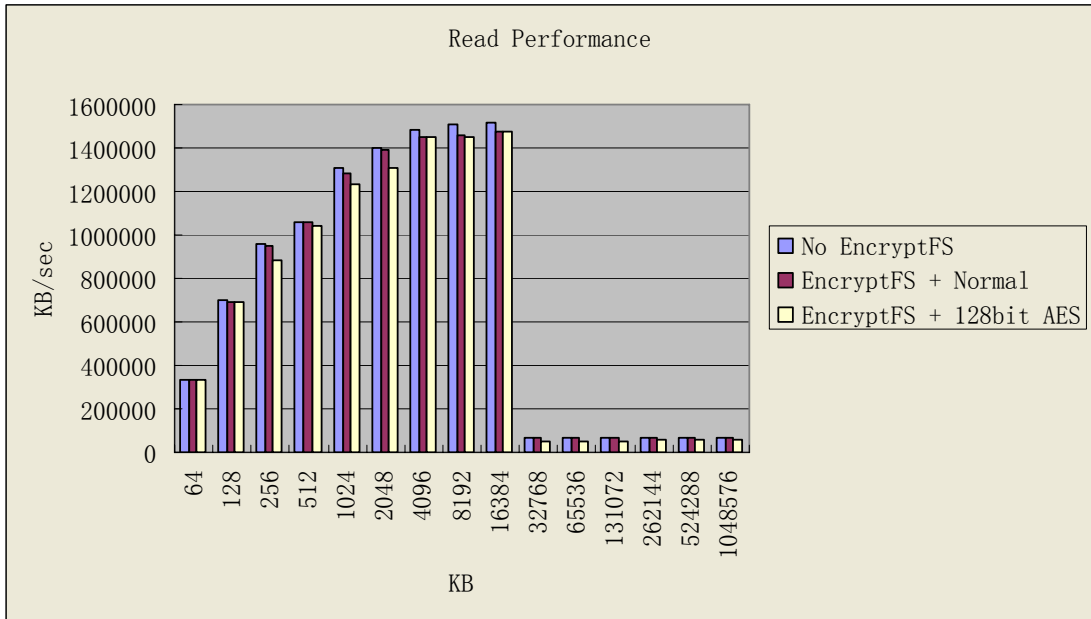


Figure 3. Read Performance

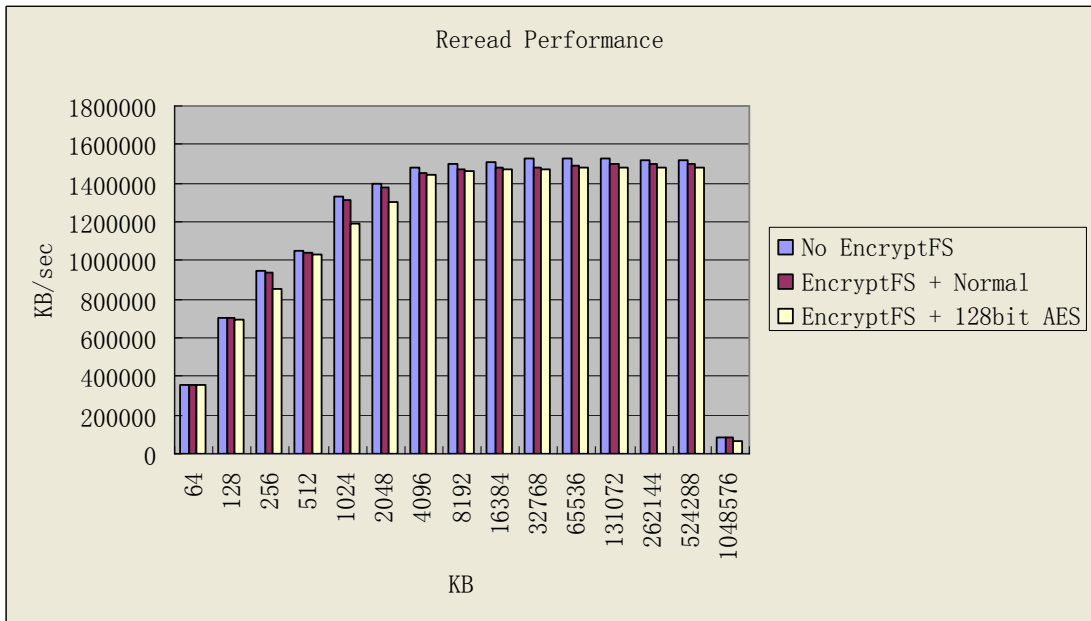


Figure 4. Reread Performance

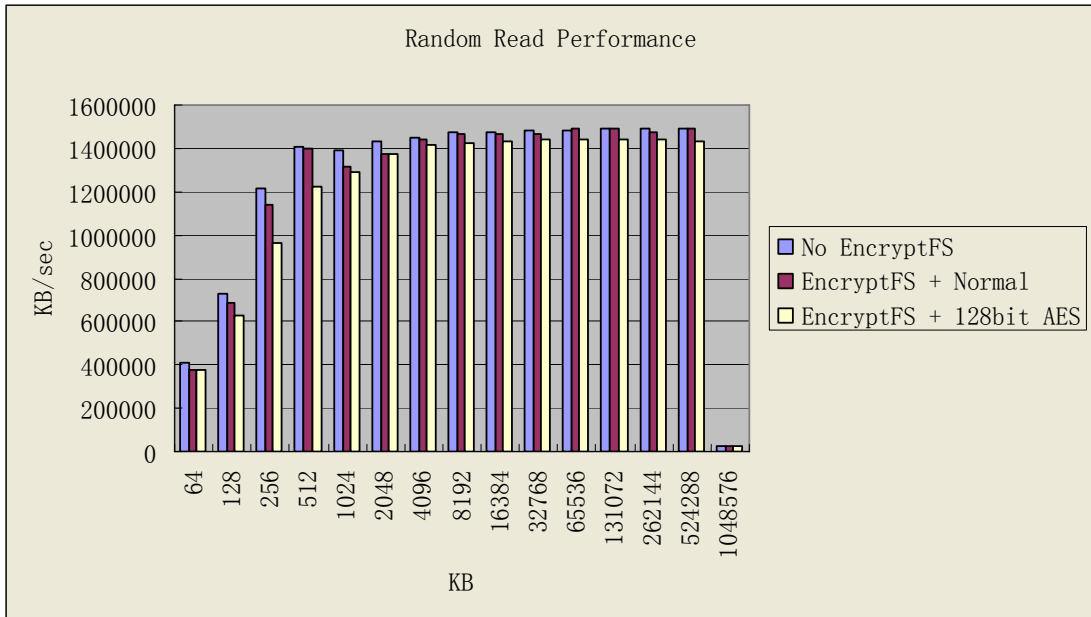


Figure 5. Random Read Performance

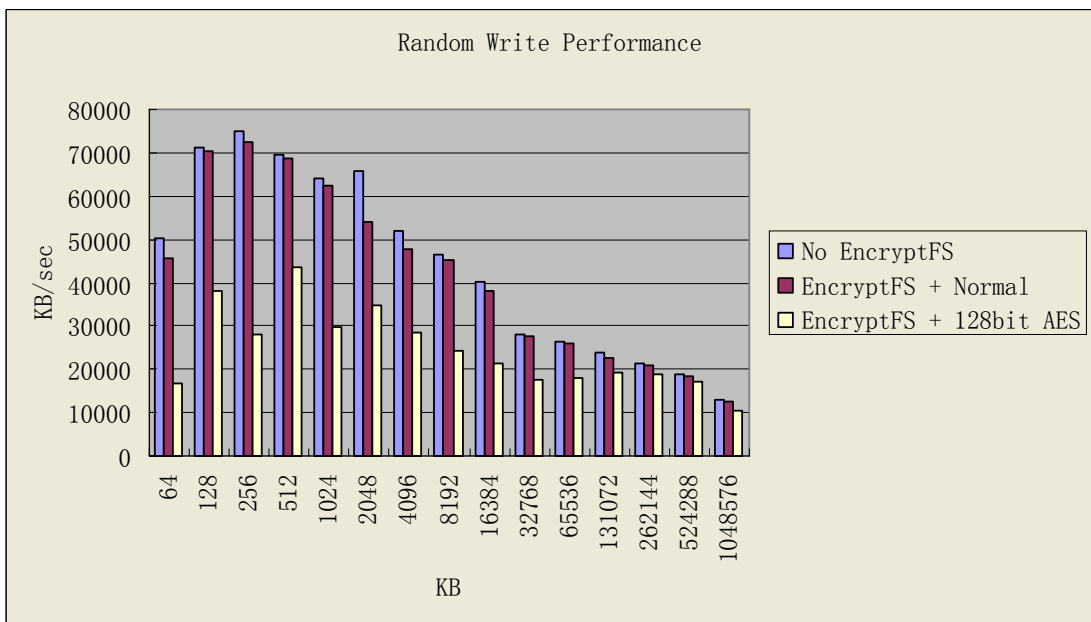


Figure 6. Random Write Performance

From the figures we see:

- Because the inherent overhead for accessing small files (< 512K) is rather low, performance degradation is remarkable; for larger files (>=512K), performance penalty is less.
- Extra overhead for non-encrypted files is low. For small files, performance degrades in the range of -5% to -15%; for larger files, performance degrades about -2%.
- Performance penalty of "Encrypt-FS + 128bit AES" is about -12% ~ -67% in "Write" test; -18% ~ -66% in "Rewrite" test; -1% ~ -27% in "Read" test; -1% ~ -23% in "Reread" test; -3% ~ -21% in "Random Read" test; -10% ~

-66% in "Rewrite" test.

- Disk cache is highly important to Encrypt-FS. After the first access, data in page cache remains decrypted as long as possible, thus the overall performance upgrades remarkably especially in "Reread" and "Rewrite" tests for large files.

## 5.2 Bonnie Benchmark

Bonnie is a file system benchmark written by Tim Bray. It is implemented as a single C program. The program takes arguments for the size of the file it will use and the path where it will be stored. Bonnie then creates the large file and performs many sequential operations on it, including writing character by character, writing block by block, rewriting, reading character by character, reading block by block, and seeking. In this test, we use a file size of 1 GB.

Figure 7 shows the performance of the different Bonnie phases:

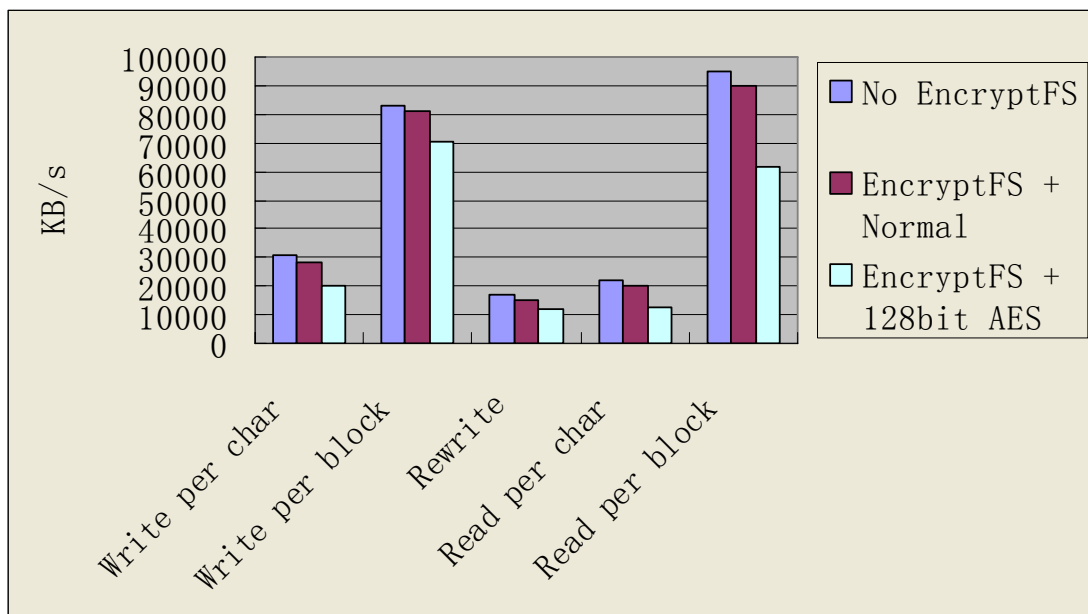


Figure 7. Bonnie results

From the figure, we see:

1. Extra overhead for non-encrypted files is less than 7%.
2. Encrypt-FS has a comparatively high performance dealing encrypted files. For "Write per block" test, performance degrades about 15%; for "Read per block" test, performance degrades about 35%.
3. For "Write per char" test, performance degrades about 33%; for "Read per char" test, performance degrades about 42%. Because manipulating characters randomly surely results in added disk operations. Thus the tests on characters perform less than those on blocks.
4. For "Rewrite" test, performance degrades about 30%. This test involves lots of random reading and writing operations which consume more extra time in dealing disk operations. At the same time disk operations bring on more

encryption and decryption.

## 6. Conclusions and future work

We have described the design and implementation of Encrypt-FS, a comprehensive cryptographic file system suitable for personal and multi-user systems. Users could store their files on any target specific file system, from Ext2, Ext3, ReiserFS to JFS and XFS, as long as the file system supports XATTR (extended attributes). Encrypt-FS offers an easy and secure way to share encrypted files with co-workers. At the same time, we try our best to enhance the security of Encrypt-FS, avoiding leaking out related keys and sensitive information in memory. Test results show that Encrypt-FS is high efficient.

In future, we plan to add more extensions to current implementation:

- (1) File name encryption. Although a file's name is not as important as its content, the name may be meaningful, and thus could expose some information for intruders.
- (2) Currently Encrypt-FS is implemented on Linux kernel 2.4, we will port it to 2.6 soon.
- (3) Some other UNIX systems have similar file system architecture with Linux, and we will attempt to port Encrypt-FS to FreeBSD.

## References

- [1] H. Reiser. [www.namesys.com](http://www.namesys.com)
- [2] M. Blaze. A Cryptographic Filesystem for Unix. Proceedings of the First ACM Conference on Computer and Communications Security, pp. 9-16, November 1993.
- [3] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic Filesystem for Unix. Unpublished Technical Report, July 1997.
- [4] S. Ludwig and W. Kalfa. File System Encryption with Integrated User Management. ACM SIGOPS Operating Systems Review, Volume 35 Issue 4, pp. 88-93, October 2001.
- [5] Cheng Ding and Yufang Sun. Crypt-FS: A Cryptographic File System Designed for Linux. Computer Engineering, pp. 111-113, November 2003.
- [6] Dava Probert and Xiangqun Chen. Principles of the Windows Operating Systems, 2<sup>nd</sup> Edition. China Machine Press, November 2004.
- [7] <ftp://ftp.redhat.com/pub/redhat/linux/updates/enterprise/3AS/en/os/SRPMS/kernel-2.4.21-27.0.2.EL.src.rpm>
- [8] N. Provos. Encrypting Virtual Memory. USENIX Security Symposium. Denver, CO, August

2000

[9] Decao Mao and Ximing Hu. Linux Kernel: Scenarios and Analyses. Zhejiang University Press, September 2001.

[10] R. Love. The Linux Kernel Development , 2nd edition. Novell Press, January 2005.

[11] D. P. Bovet and M. Cesati. Understanding the Linux Kernel, 2nd Edition. O'Reilly, December 2002.