Buffer Cache Level Encryption for Embedded Secure Operating System*

Jaeheung Lee[1], Junyoung Heo[1], Jaemin Park[1], Yookun Cho[1], Jiman Hong[2], and Minkyu Park[3]**

[1] Seoul National University
{jhlee, jyheo, jmpark, cho}@ssrnet.snu.ac.kr
[2] Soongsil University
jiman@ssu.ac.kr
[3] Konkuk University
minkyup@kku.ac.kr

Abstract. A cryptographic file system is the representative way of assuring confidentiality of files in operating systems. For secure embedded operating systems, the cryptographic file system could be a practical technique. In general, cryptographic file systems are implemented using a stackable file system or a device driver. These two mechanisms can provide user transparent encryption/decryption of cryptographic file systems. But these mechanisms sometimes encrypt or decrypt data redundantly or unnecessarily. Embedded systems with a low speed CPU and flash storage are more affected by the problems than general systems. We addressed the above mentioned problems by applying an encryption algorithm on buffer caches and enabling one buffer cache to have both encrypted and decrypted data together. Experimental results show that the proposed mechanisms reduce the redundant or unnecessary operations and it can improve the performance of cryptographic file systems.

KEY WORDS
Security, Cryptographic File System, Embedded Operating System, Buffer Cache, Linux

1 Introduction

Secure data technique is more attractive as the value of information increases and threats to the information increase[1]. Cryptographic file systems are the most practical technique for secure operating systems. Even though the files are stolen by physical or network attacks, it can protect files that contain the

* This research was supported in part by the Brain Korea 21 project and MIC & IITA through IT Leading R&D Support Project. The ICT at Seoul National University provides research facilities for this study.

** corresponding author

valuable information by encrypting the information. Attackers cannot get secure information of files unless they get the key of the cryptographic algorithm.

Considering the growing interest in mobile embedded systems, the use of cryptographic file systems in embedded systems will be more important. The securing data in mobile embedded systems is more important than existing general systems because the mobility causes a new threat, that is the lost of the system itself[2].

The performance of a file system is sure to decrease after applying a cryptographic file system. Because cryptographic algorithms require a lot of CPU instruction for encryption and decryption. From the user's point of view, the degradation of the performance must be tolerable. So the reasonable performance is required for cryptographic file systems. Especially, embedded systems are more affected by the degradation of the performance than other systems because they are composed of a low speed CPU, a small memory and batteries.

So far, many cryptographic file systems have been introduced. In this study, we consider cryptographic file systems only in kernel-level [3–9]. These are categorized by the implementation method: using a stackable file system and using a device driver.

A stackable file system is an easy and efficient way to add new features into existing file systems of a kernel[10, 7]. With a stackable file system, you can easily add encryption or compression algorithms into a file system. Kernel modification is a very hard job due to the complexity of a kernel and the difficulty in debugging it. The stackable file system allows a new feature to be integrated into a kernel without influencing another part of a kernel. In addition, it is faster than user-level library because the stackable layer is in a kernel. Therefore, many cryptographic file systems using a stackable file system have been developed.

However, the stackable file system cannot exploit the advantage of buffer caches[11]. If a user process accesses some part of a file twice, encryption or decryption should be repeated twice in a stackable layer. Therefore, redundant encryption or decryption may occur in a cryptographic file system using a stackable file system.

Another popular method to implement cryptographic file systems is a device driver[3–6]. Strictly speaking, there is no relation between using a device driver and the file systems. Because it locates a cryptographic algorithm in a device driver, a file system does not know the existence of a cryptographic algorithm. As a result, all data read from a disk is always decrypted before being stored in buffer caches. Inversely, all data stored in buffer caches is always encrypted before being stored in a disk. Therefore, unnecessary encryption or decryption may occur in a device driver.

To improve the performance by reducing those redundant and unnecessary encryption/decryption, we propose the mechanism, support in a buffer cache level. Basically, our mechanism locates a cryptographic algorithm in buffer caches and enables a buffer cache to have mixed plaintext and ciphertext.

In the request of a read operation from the VFS(or user application), encrypted data is decrypted before copying the data from a buffer cache to the

VFS. The amount of decryption is strictly limited to the size of requested data. If data in a buffer cache is decrypted once, the decrypted data is replaced with an encrypted one. If the request of a read operation at the same position in a file occurs again, no more decryption is required. This is helpful to remove redundant encryption/decryption of a cryptographic file system using a stackable file system.

By limiting the amount of decryption to the size of requested data, the problem of unnecessary decryption is resolved. To this end, we designed a new buffer cache with a bitmap to indicate whether data is encrypted or decrypted. Therefore, a buffer cache of our mechanism is able to have plaintext and ciphertext together.

The rest of this paper is organized as follows. Section 2 surveys related works. Section 3 describes our mechanism. Section 4 presents our experimentation and results. Section 5 presents our concluding remarks.

2 Related Work

We survey some related work in this section. There are many techniques related to securing files: encryption of a storage device, encryption using a user-level library, encryption in a device driver and encryption using a stackable file system.

Encryption of a storage device can be operated without operating systems. It is integrated with a storage hardware. Therefore, its performance is better than that of other software techniques. DataTravler Elite [12] and SecureIDE [13] fall under this category. Encryption using a user-level library such as crypt(3) and GNU PG[14] are also available. CFS[15] and TCFS[16] are examples of user-level mechanisms. They use a NFS server for applying cryptographic algorithms. These user-level mechanisms can be easily implemented, but it has many problems with respect to key management, consistency, performance and so on. We will not mention these techniques any more because our study focuses on the kernel mechanism.

We focus on two kernel mechanism: encryption in a device driver and a stackable file system. Strictly speaking, encryption in a device driver is not a cryptographic file system because a cryptographic file system is a file system to manage files on a disk securely. However, to simplify the terms, we will refer to both of them as a kind of cryptographic file systems.

In a cryptographic file system using a device driver, encryption is carried out while an I/O operation is being performed. It can be used for the applications that accesses to a storage directly such as a swap device and a database. It can also be used for the general applications that require file systems. It can provide user transparency and good performance, but it cannot encrypt or decrypt a unit of a file. Cryptoloop [3], CryptoGraphic Disk Driver (CGD) [4], SFS [6] and BestCrypt [5] fall into this category.

A cryptographic file system using a device driver outperforms the stackable mechanism. However, the stackable mechanism can provide file encryption. It also provides user transparency like the device driver mechanism.

Cryptfs [7] and Ncryptfs [8] use FiST [17] as a stackable file system. Ncryptfs is an improved version of Cryptfs. Ncryptfs can authenticate several users simultaneously and apply cryptographic algorithms dynamically. It also provides challenge-response authentication. It uses block cipher and applies CFB (Cipher FeedBack) mode for inode blocks, ECB (Electronic CodeBook) for data blocks.

EFS (Encryption File System) is a cryptographic file system based on MS Windows NT[9]. It exists in a kernel, but it requires user DLL for encryption and user authentication.

3 Buffer Cache Level Support

The main objective of our mechanism is to exploit the advantage of buffer caches. In addition, our mechanism enables a buffer cache to be encrypted or decrypted partially. In this study, we use a block cipher as a cryptographic algorithm and an ECB(Electronic CodeBook) mode as a block cipher mode. With this combination, random access and equal length of plaintext and ciphertext can be achieved.

Fig. 1 shows how decryption is performed in a read operation. When a user requests an operation that reads data in the block 2 of a file, the block is loaded to a buffer cache and copied to a user area. In the system using a stackable file system, Fig. 1(a), decryption is performed right before copying data from a buffer cache to a user area. If another read operation that requests data which is already requested in a previous read operation is performed, decryption will be repeated. Such redundant operation can occur because a stackable file system does not care whether a buffer cache exists or not.

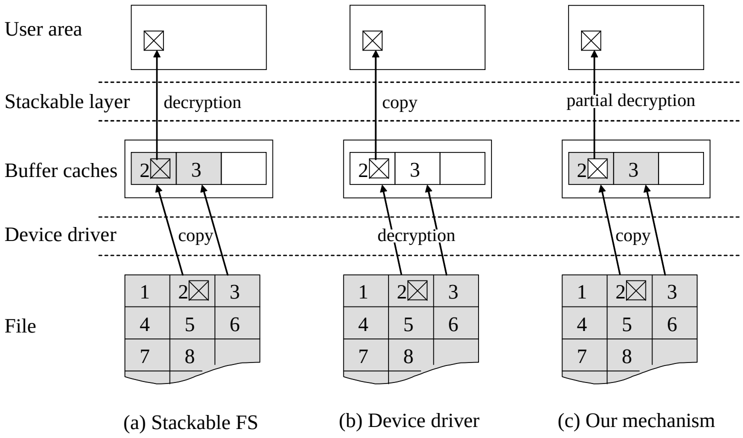


Fig. 1. Decryption in Read Operation

In case of using a device driver, Fig. 1(b), decryption is performed right before loading data from a file to a buffer cache. Buffer caches always have decrypted data. Therefore, redundant decryption does not occur. However, decryption in a device driver level may cause unnecessary decryption. As a device driver does not have any information about the read operation, the entire block 2 is read and decrypted even though the part of the block 2 is required.

Another case of unnecessary decryption in a device driver level may occur because of a read-ahead technique. Operating systems sometimes read the next data of current read in advance. A read-ahead assumes that disk accesses are sequential[18]. However, a read-ahead algorithm does not always succeed as predicted.

In our mechanism, Fig. 1(c), the block 2 of a file is copied to buffer caches but not decrypted. Only the requested area of the block 2 is decrypted and stored back in buffer caches. After this partial decryption, the requested data is copied to a user area. Because unused area of the block 2 is not decrypted, this partial decryption can remove the unnecessary decryption. We can also expect that the redundant decryption will be removed by the buffer caches because the decrypted result is stored in buffer caches.

In a write operation, redundant encryption can occur as in a read operation. We consider only overwrite operations that write data in the existing part of a file. In case of a append operation, three mechanisms are similar. Fig. 2 shows the case of a write operation. In a repeated write operation, the stackable layer encrypts the data redundantly. In case of using a device driver and our mechanism, this redundancy does not occur.

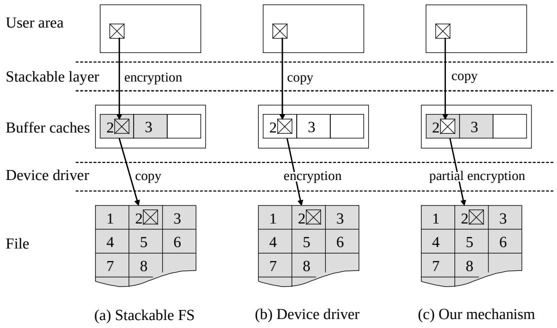


Fig. 2. Encryption in Write Operation

We add new information in a buffer cache to support partial encryption/ decryption of a buffer cache. This information enables a buffer cache to distinguish which part of a buffer cache is encrypted or decrypted. The information, BITMAP is shown in Fig. 3. The DATA in a buffer cache is divided into cipher blocks. If the bit is 1, the corresponding cipher block is encrypted. Otherwise, the cipher block is decrypted. Block cipher algorithms use a cipher block as a basic encryption or decryption unit. In case of AES, the size of a cipher block is 16 bytes. If the size of a buffer cache(DATA) is 4096 bytes, the size of BITMAP is 32 bytes. That is, additional 32 bytes per each buffer cache is necessary when AES is used as a cryptographic algorithm.

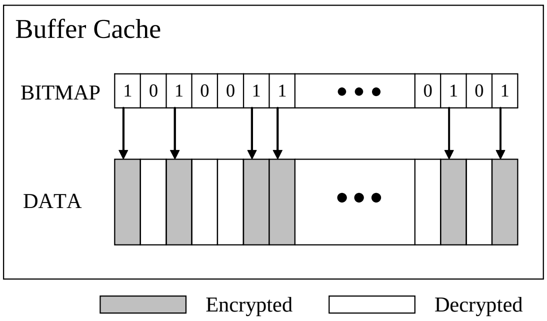


Fig. 3. Information for Partial Encryption/Decryption in Buffer Cache

4 Performance Evaluation

We implemented our buffer cache level support on Linux kernel version 2.6.11. In addition, we made other mechanisms because we want to exclude other factors which affect the performance except buffer caches.

Table 1 shows the experimental setup. We selected the embedded system that has a lower CPU than a desktop and a NAND flash for a storage. The reason is that the performance improvement is more important to the mobile embedded system than other general systems. In that respect, our mechanism is more suitable for the mobile embedded systems.

Table 1. Experimental setup

CPU	Intel Xscale PXA270 520MHz
Main memory	SDRAM 64Mbytes
Storage	NAND Flash - page size 512+16 bytes - erase block size 16K+512 bytes
Operating system	Linux kernel version 2.6.11
File system	YAFFS[19]
Size of buffer cache	4Kbytes
Cryptographic algorithm	AES(ECB mode) with 128bits key

In our experiment, AES was used for encryption. The key length was 128 bits and ECB(Electronic CodeBook) mode was used for block encryption. The throughput of AES was about 240 Kbytes/s.

General file system benchmarks are not suitable to measure the effect of a cache. We made two type of synthesized workload to compare our mechanism with other cryptographic file systems in terms of buffer cache hit ratio.

- Workloads with requests of a read operation with cache hit ratio 0, 0.25, 0.50, 0.75 and 1.0
- Workloads with requests of a write operation with cache hit ratio 0, 0.25, 0.50, 0.75 and 1.0

We applied these workloads for various sizes of data: 64bytes, 128bytes, 256bytes, 512bytes, 1Kbytes, 2Kbytes and 4Kbytes. We got the average throughput by repeating the experiments 1000 times.

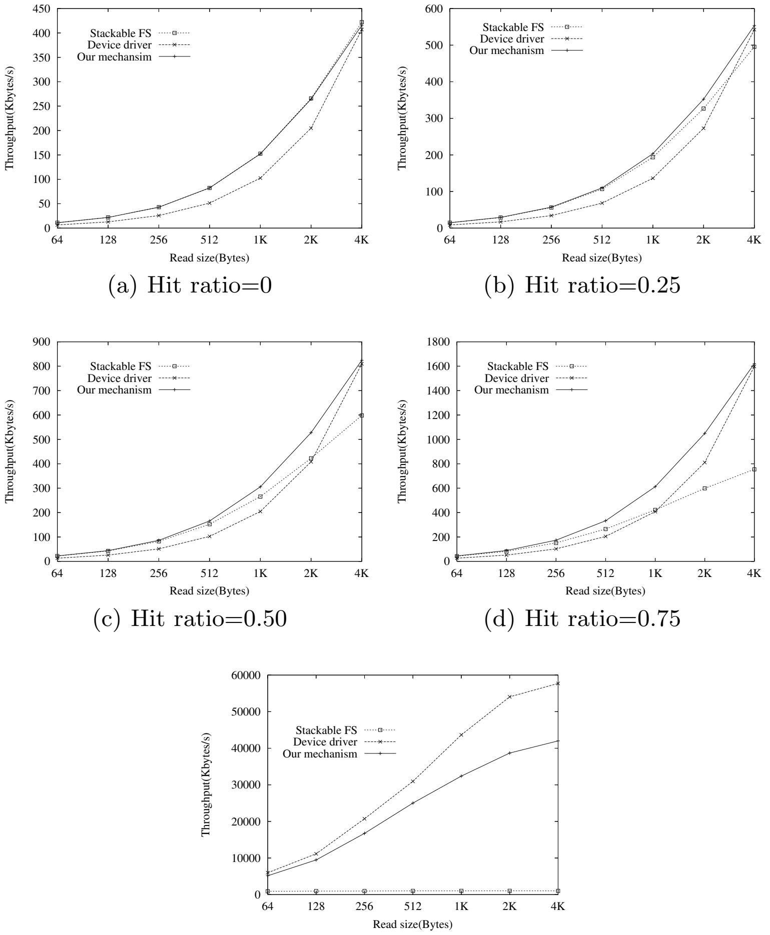
Fig. 4 shows the results of read operations with the workloads. As the read size increases, the throughput also increases. Before copying data to a user area, the systems read 4Kbytes data from a file regardless of the actual read size. 4Kbytes is the size of a buffer cache. A read operation of 64bytes data results in reading unnecessary 4Kbytes - 64bytes. Therefore, the larger read size makes the larger throughput.

Except Fig. 4(e), hit ratio=1.0, our mechanism performs better than others. In case of hit ratio=1.0, the read operations request the data already loaded in a buffer cache. Therefore, unnecessary decryption is never occurred in case of using a device driver. In other graphs of Fig. 4, the performance of the system using a device driver is less than that of our mechanism due to the unnecessary decryption.

In Fig. 4(a), hit ratio=0, the throughput of the system using a stackable file system is similar to that of our mechanism. Our mechanism cannot exploit the buffer cache when hit ratio is 0. In case of using a device driver, it always decrypts entire buffer cache, even though less data is requested. This results in the degradation of the performance in case of using a device driver.

As the hit ratio increases, the relative performance of the system using a stackable file system decreases and the relative performance of the system using a device driver increases. However, the relative performance of our mechanism is not affected largely by the hit ratio because our mechanism can fully exploit the buffer caches. The decrease of the relative performance of the system using a stackable file system results from the redundant decryption.

Fig. 5 shows the results of write operations with the workloads. Like the read operations, the throughput is increases as the write size increases. Except Fig. 5(e), hit ratio=1.0, our mechanism outperforms the case of using a device driver. The system using a device driver performs unnecessary encryption while committing a buffer cache into a disk. This results in the performance degradation. The throughput of the system using a stackable file system is equal to ours because the repeated write is not considered in this experiment. In case of 4Kbytes, other mechanisms must encrypt entire buffer cache like the system



(a) Hit ratio=0

(b) Hit ratio=0.25

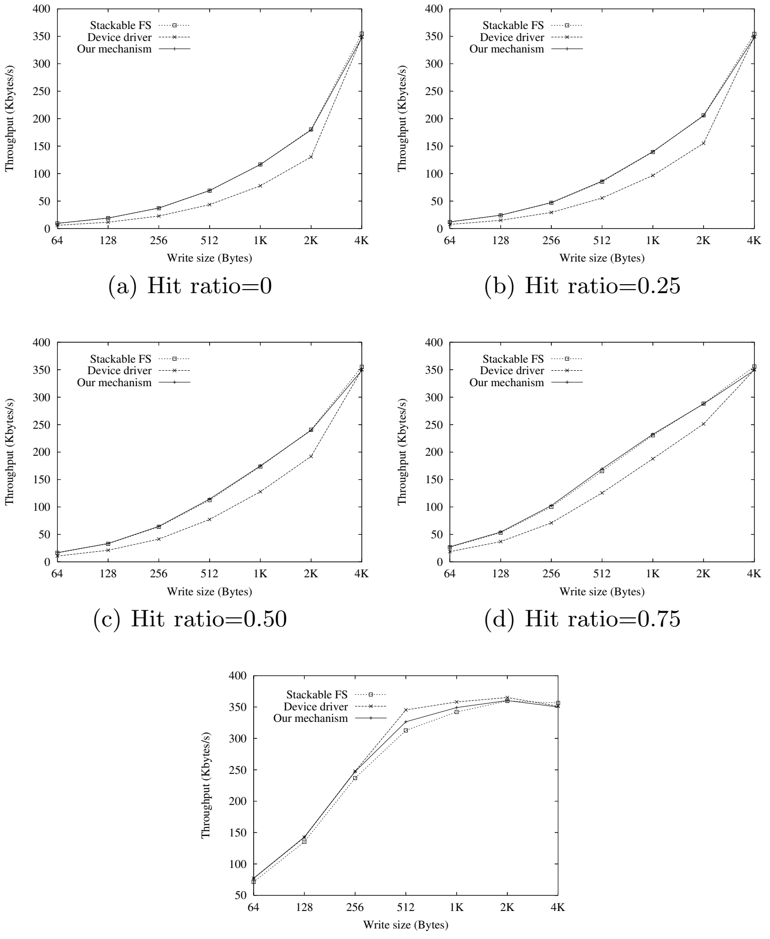
(c) Hit ratio=0.50

(d) Hit ratio=0.75

(e) Hit ratio=1.0

Fig. 4. Read with cache hit ratio=0, 0.25, 0.50, 0.75 and 1.0

using a device driver. Therefore, when the write size is 4Kbytes, the throughput of the system using a device driver is equal to that of others.



(a) Hit ratio=0

(b) Hit ratio=0.25

(c) Hit ratio=0.50

(d) Hit ratio=0.75

(e) Hit ratio=1.0

Fig. 5. Write with cache hit ratio=0, 0.25, 0.50, 0.75 and 1.0

From the Fig. 4 and Fig. 5, we can know that the redundant and unnecessary encryption/decryption can be reduced by our mechanism. The system using a device driver outperforms others sometimes. However, this is occurred in a special case, hit ratio=1.0 and this special case does not almost happen in real computer systems.

5 Conclusions and Future Work

Many cryptographic file systems have been developed and used in real systems. In general, kernel-level techniques are preferred because they are more cost-effective than hardware techniques and outperform user-level techniques. However, existing kernel-level cryptographic file systems have some drawbacks. The system using a stackable file system overlooks the effect of buffer caches and the system using a device driver causes unnecessary decryption of data. We modified buffer caches of a kernel to encrypt or decrypt data on a buffer cache partially. This modification is helpful to eliminate unnecessary or redundant operations of encryption/decryption and improve the performance.

References

1. Hasan, R., Myagmar, S., Lee, A., Yurcik, W.: Toward a threat model for storage systems. In: Proceedings of International Workshop on Storage Security and Survivability(StorageSS). (2005)
2. Ravi, S., Raghunathan, A., Kocher, P., Hattangady, S.: Security in embedded systems: Design challenges. ACM Transactions on Embedded Computing Systems **3** (2004) 461–491
3. GNU: The GNU/Linux CryptoAPI (2003)
4. Dowdeswell, R., Ioannidis, J.: The cryptographic disk driver. In: Proceedings of the Annual USENIX Technical Conference, FREENIX Track. (2003)
5. Jetico Inc.: Bestcrypt corporate edition (2001)
6. Gutmann, P.C.: Secure file system(SFS) for DOS/Windows (1994)
7. Zadok, E., Badulescu, I., Shender, A.: Cryptfs: A stackable vnode level encryption file system. Technical Report CUCS-021-98, Computer Science Department, Columbia University (1998)
8. Wright, C., Martino, M., Zadok, E.: Ncryptfs: A secure and convenient cryptographic file system. In: Proceedings of the Annual USENIX Technical Conference. (2003) 197–210
9. Microsoft Corporation: Encrypting file system for Windows 2000 (1999)
10. Zadok, E., Badulescu, I.: A stackable file system interface for Linux. In: Proceedings of the 5th Annual Linux Expo. (1999) 141–151
11. Wright, C., Dave, J., Zadok, E.: Cryptographic file systems performance: What you don't know can hurt you. In: Proceedings of the Second IEEE International Security In Storage Workshop. (2003) 47–62
12. Kingston Technology company: DataTraveler Elite (2006)
13. ABIT Computer corporation: Secure IDE (2003)
14. GNU: GNU Privacy Guard (1999)
15. Blaze, M.: A cryptographic file system for UNIX. In: CCS '93: Proceedings of the 1st ACM conference on Computer and communications security. (1993) 9–16
16. Cattaneo, G., Catuogno, L., Sorbo, A.D., Persiano, P.: The design and implementation of a transparent cryptographic file system for UNIX. In: Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. (2001) 199–212
17. Zadok, E., Nieh, J.: FiST: A language for stackable file systems. In: Proceedings of the Annual USENIX Technical Conference. (2000) 55–70
18. Bovet, D.P., Cesati, M.: Understanding the Linux Kernel. O'Reilly (2006)
19. Aleph One: YAFFS: the NAND-specific flash file system (2002)