情報保護學會論文誌
第16卷 第3號, 2006. 6

사용자 투명성을 갖는 커널 수준의 파일 암호화 메카니즘

김 재 환[1†], 박 태 규[2], 조 기 환[1‡]
[1]전북대학교, [2]한서대학교

User Transparent File Encryption Mechanisms at Kernel Level

Jae-Hwan Kim[1†], Tae-Kyou Park[2], Gi-Hwan Cho[1‡]
[1]Chonbuk National University, [2]Hanseo University

요 약

기존의 운영체제에서 암호화 및 복호화는 응용 프로그램 수준에서 사용자가 지정한 파일을 암호화 키 및 복호화 키를 입력하여, 파일 단위로 일괄적으로 전체를 암호화하고 복호화하는 방식을 사용하였다. 이러한 방식에서는 성능상의 오버헤드가 매우 크다. 본 논문에서 제안하는 메커니즘은 리눅스 커널 수준에서, 일정한 보안 등급을 보유하는 사용자 프로세스가 비밀 수준의 파일을 쓰고(write)자 할 때 사용자에게 투명하게 자동으로 효율적으로 파일을 암호화하여 저장한다. 또한 암호화된 파일을 수정할 때는 필요한 만큼만의 데이터를 자동으로 부분 복호화 및 암호화를 하고, 읽을(read) 때에도 필요한 만큼만을 부분 복호화한다. 따라서 본 메커니즘은 종래의 응용 수준에서의 전체 암호화 및 복호화로 인한 처리 시간을 크게 감소시킨 효율적인 커널 수준의 자동 부분 암호화 수단을 제공한다.

ABSTRACT

Encipherment in existing OS(Operating Systems) has typically used the techniques which encrypt and decrypt entirely a secret file at the application level with keys chosen by user. In this mechanism it causes much overhead on the performance. However when a security-classified user-process writes a secret file, our proposed mechanism encrypts and stores automatically and efficiently the file by providing transparency to the user at the kernel level of Linux. Also when the user modifies the encrypted secret file, this mechanism decrypts partially the file and encrypts partially the file for restoring. When user reads only the part of the encrypted file, this mechanism decrypts automatically and partially the file. Therefore our proposed mechanism provides user much faster enciphering speed than that of the existing techniques at the application level.

Keywords : *Encryption, Kernel level*

Ⅰ. 서 론

데이터 암호화는 일상의 작업 시마다 중요성이 점차 증가되는 한 요소가 되고 있다. 사용자는 자신을 최대한 편하게 하고, 자신들만의 추가적인 요구는 최소한으로 하면서 자신들의 데이터를 안전하게 취급하고자 하는 방법을 찾고 있다. 사용자는 특정 응용 시스템에 종속된 암호화 방법에 구애받지 않고, 어떤 응용 시스템에서든 사용되는 모든 파일을 보호할 수

접수일: 2005년 12월 19일; 채택일: 2006년 4월 18일
† 주저자, kajah@kwater.or.kr
‡ 교신저자, ghcho@chonbuk.ac.kr

있는 보안 시스템을 원한다. 암호화는 처리 시간이 꽤 소요되는 작업이기 때문에 이때의 성능은 사용자에게 있어서 중요한 요인이다. 최근 들어, 정보통신의 비약적 발달로 비밀 수준의 정보들이 컴퓨터에 저장되어 통신망 혹은 저장매체를 통하여 불법적으로 탈취, 변조 또는 소유자의 실수에 의하여 비밀 정보가 누출되는 등의 보안문제가 심각해지고 있다. 따라서 컴퓨터 보안을 위하여 운영체제 수준에서 강화된 접근통제 기법과 함께 암호화 기법은 매우 중요한 역할을 수행한다.[1,3] 운영체제 제작자는 비용이 많이 소요되는 새로운 파일시스템을 개발하지 않고도 이러한 기능을 컴퓨터 시스템에서 사용자에게 제공하고 싶어 한다.

일반적으로 컴퓨터 운영체제 상에서 암호화 기법을 이용하여 시스템 관리자(또는 보안 관리자)나 사용자 측면에서 중요한 파일의 비밀성을 유지하는 방법은 두 가지 방법으로 나뉜다. 첫째는 응용 프로그램 수준에서 명령어나 응용 프로그램을 통하여 사용자가 중요한 파일을 선택적으로 지정하여 전체 파일 데이터 또는 디렉터리를 암호화 및 복호화하는 방법이다. 이때 사용자의 번거로움을 제거하기 위하여 데이터의 암호화 기능을 응용 프로그램 내에서 통합하여 실행하는 방법도 있을 수 있다. 둘째는 운영체제 수준에서 파일시스템을 통하여 파일 단위로 암호화 및 복호화를 실행하는 방법이 있다.[3-5] 이 방법은 응용 프로그램에서 필수적으로 사용하는 로컬 또는 NFS(Network File System)와 같은 네트워크 파일시스템 자체에 암호화 및 복호화 기능을 내장하여 중요 파일에 대하여 강제적이며 자동적으로 암호화 및 복호화를 수행토록 하는 방법이다.

지금까지 널리 사용되는 응용 수준 및 운영체제 수준의 암호화·복호화 도구나 파일시스템들은 모두, 암호화 및 복호화 시에 파일 또는 데이터 스트림 전체를 일괄적으로 암호화 및 복호화를 실행한다. 다시 말해서 암호화되어 저장된 파일의 내용을 일부라도 읽거나, 수정하기 위해서는 암호화된 파일의 전체를 복호화하여야 하며, 복호화된 파일의 평문 내용을 부분적으로 한 글자만이라도 수정해도 파일 내용 전체를 암호화하여 다시 저장하여야만 한다. 이로써 야기되는 성능상의 오버헤드는 매우 크다. 특히 크기에 있어 대형화되는 추세인 시스템 파일 및 일반 사용자 파일, 중요 보고서 파일이나 동영상 파일, 공유 데이터 파일 등에 있어서 약간의 부분적인 수정에도 불구하고 전체 파일을 모두 암호화 및 복호화함은 매우 큰 성능상의 비용을 치러야만 한다. 이러한 성능상의 문제로 암호화를 위한 전용 보조 프로세서의 필요성이 제기되기도 하나 구현의 어려움과 비용 과다의 문제가 있다.

따라서 본 논문에서는 커널로 존재하는 파일시스템 수준에서 사용자에게 투명성과 자동성을 제공하는 것 외에도, 부분적인 암호화 및 복호화 기법을 통하여 효율적으로 빠르게 처리하는 부분적 암호화 메커니즘을 제안한다. 다시 말해서 이미 암호화되어 저장된 파일에 대하여 사용자에 의해 변경이 요구되는 부분(블록)만을 투명하게 다시 암호화하여 저장하고, 변경이 요구되지 않은 다른 부분들은 새로운 프로세스를 거치지 않고 종전의 암호화 내용을 그대로 사용함으로써 빠르게 암호화 처리할 수 있는 암호화 메커니즘을 제시한다.

본 논문의 구성은 제 2장에서 응용 프로그램 수준 또는 커널 수준에서 파일시스템을 통하여 파일을 암호화하는 방법을 채택하는 주요 암호화 파일시스템인 CFS(Cryptographic File System), CryptFS, TCFS(Transparent Cryptographic File System) 등의 방법을 알아보고 이 도구들을 분석한다. 제 3장에서는 제안하는 OS의 커널 모드에서 실행되는 부분 파일 암호화 메커니즘의 설계와 동작 예시, 제 4장에서는 이 메커니즘을 리눅스 커널의 시스템 호출에 구현한 내용과, 제 5장에서는 평가 분석을 통하여 처리 성능이 응용 수준에서의 처리보다 효율적임을 밝히고, 이 방법에 따라 추가로 소요되는 기억 공간을 분석한다. 아울러 제공 가능한 보안성을 분석한다.

II. 관련 연구

본 장에서는 기존 응용수준 및 운영체제 수준의 암복호화 방법을 소개하고 그 구현방법에 대해서 다룬다. 제2.1절에서는 crypt 프로그램을, 제2.2절에서는 운영체제수준에서 널리 사용되는 CFS의 암호화 동작을 설명한다. 제2.3절에서는 Vnode 수준에서 구현된 CryptFS를 다루고, 제2.4절에서는 암호화 분산 파일시스템인 TCFS를 설명한다. 제2.5절에서는 관련 연구의 분석 결과를 논한다.

2.1 crypt 프로그램[7]

응용 수준에서 데이터 파일을 사용자가 선택적으

로 암호화 및 복호화를 실행하는 방법으로 표준 Unix 계열에는 DES 암호화 알고리즘을 사용하고 있는 crypt 프로그램이 있다. 이 도구는 사용자가 부여하는 키로 파일 단위 또는 데이터 스트림 단위로 전체를 일괄적으로 암호화 또는 복호화 한다. 이 때 모든 암호화와 복호화 과정은 사용자의 직접적인 통제 하에 이루어진다. 따라서 이런 사용자의 번거로움을 없애기 위해서 중요한 데이터의 암호화 및 복호화 기능을 응용 프로그램에 내장하여 응용 프로그램 내에서 암호화 기능과 응용 기능을 동시에 통합하여 실행하도록 하는 경우가 있다. 예로 vi와 같은 문서 작성기에서는 파일을 열고, 읽거나, 쓸 때 키를 요구하도록 하여 자동적으로 작성 데이터를 암호화하여 저장토록 하게 하거나, 자동적으로 복호화하여 보여주거나 편집할 수 있도록 해준다.

2.2 CFS[4]

Matt Blaze의 CFS 암호화 파일시스템은 현재 널리 사용되는 운영체제 수준에서의 암호화 방법을 사용하며, 이식성이 좋고 파일 단위에서 사용자에게 투명한 암호화 파일시스템이다. CFS은 그림 1과 같이 구성되며 파일 시스템 자체에서 CFS Daemon을 통하여 암호화 서비스를 제공한다. CFS는 시스템 수준에서 표준 Unix 파일 시스템 인터페이스를 통하여 암호화된 파일에 대하여 안전하게 저장할 수 있는 기능을 지원한다.

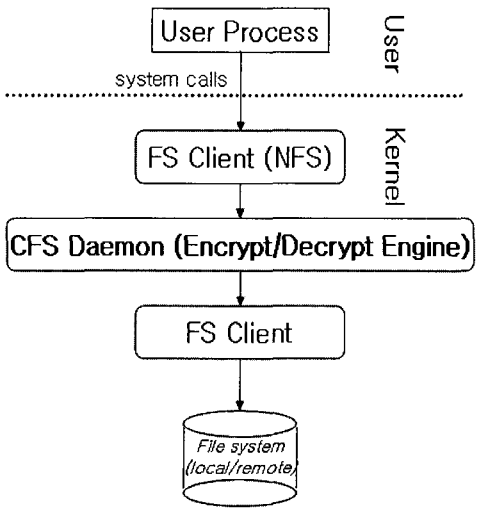

그림 1. CFS의 데이터 흐름도

사용자는 보호하고자 하는 디렉터리에 관련된 암호화 키에 대해서만 관여하며, 그 디렉터리에 존재하는 파일들은 경로 명과 함께 사용자의 부가적인 간섭 없이 지정하는 키에 의해서 투명하게 암호화되고 복호화 된다. 이때 평문 내용은 디스크에 저장되지 않으며, 원격 파일 서버에 전송되지도 않는다. CFS는 NFS와 같은 원격 파일 서버를 포함하여 가용한 어떠한 파일 시스템을 수정하지 않고도 이러한 안전한 파일 저장 기능을 사용할 수 있게 된다. 파일 백업과 같은 시스템 관리 기능은 키를 알지 못해도 정상적인 방법으로 동작된다. 구조적인 측면에서는 2.4절의 TCFS와 가장 유사하다.

2.3 CryptFS[5]

CryptFS는 스태킹이 가능한(stackable) 파일시스템의 추상화를 사용하여 가상 inode 수준에서 구현된 암호화 파일시스템으로, 지역 또는 원격 파일시스템 상에서 사용될 수 있다. 스태킹이 가능한 Vnode 계층에서 LKM(Loadable Kernel Module)으로 개발되었으며 사용자에게 투명하게 동작한다.[8] 즉 CryptFS는 사용자에게 투명하면서 암호화 계층을 갖는 클라이언트 파일시스템을 캡슐화함으로써 동작한다. 커널에 상주함으로써 CryptFS는 사용자 수준에서 보다 좋은 성능을 보여주며, CFS[4], TCFS[6]와 같은 NFS 기반의 파일 서버보다도 더 좋은 성능을 보여준다. CryptFS에서는 사용자 ID 뿐만 아니라 프로세스의 세션 ID에 근거하여 키를 생성하여 사용하기 때문에 더 강한 보안성을 제공하며, 커널 메모리에 접근하기가 더 어렵기 때문에 더 좋은 보안성을 갖는다고 할 수 있다. Vnode 상에서 동작하기 때문에 CryptFS는 디스크나 네트워크와 같은 원래의 저장 매체에서 직접 작업하는 파일 시스템에 비하여 더욱 이식성이 좋다.

CryptFS는 UFS/FFS, NFS와 같은 시스템 본래의 어떤 파일시스템 상에서도 동작이 될 수 있다. 결국 CryptFS는 원격 서버나 클라이언트 파일시스템에 대한 변경이 필요치 않다. 그림 2는 단일 수준의 스태킹이 가능한 암호화 파일시스템의 구조로서 사용자 프로세스에서의 시스템호출은 Vnode 수준의 호출로, 이 Vnode 호출은 CryptFS 수준의 호출로 바뀌며, 다시 일반 Vnode 연산을 호출하고 계속해서 UFS와 같은 하위 수준의 특정 파일시스템을 부르게 된다. CryptFS는 TCFS와 같이 일반적으로

4K 또는 8K 크기인 하나의 블록 내에서 암호 블록 체인이라는 암호화 방법을 사용하며, 암호화 알고리즘으로써 Blowfish 등을 제공하고 있다.

2.4 TCFS[6,10]

TCFS는 그림 3과 같이 구성되는 암호화 분산 파일시스템이다. 이것은 사용자로 하여금 안전한 방법으로 원격 서버에 저장된 중요한 파일에 접근할 수 있도록 해준다. 이것은 서버와 네트워크 상에서 암호화와 메시지 요약(digest)을 사용하여 위장 공격과 데이터 변조를 방어할 수 있게 한다. 응용 시스템은 정상적인 시스템 호출을 통하여 TCFS 파일시스템 상에서 데이터에 접근할 수 있기 때문에 사용자에게 완전한 투명성을 제공한다. TCFS는 가상 파일시스템(VFS)과 저장 파일시스템(ext2fs, ufs, nfs 등) 사이에 존재하도록 한 하나의 중간 계층으로서 커널 공간에서 동작하도록 설계되었다.

이런 방법으로 사용자 응용 프로그램은 프로그램을 재작성하거나 재 컴파일을 하지 않고 시스템 호출 인터페이스를 통해서 모든 일반적인 파일 연산을 수행할 수 있다. 사용자는 mount와 ioctl과 같은 시스템 호출을 통해서 TCFS 계층과 대화식으로 처리하는 유틸리티로 보호 및 암호화 연산을 수행할 수 있다. TCFS는 단지 응용 데이터만을 처리하며, inode, 디렉터리 구조 등의 파일 시스템의 논리적인 구조는 처리하지 않는다. 그러므로 모든 디스크 관리 작업(check, backup, recovery)은 종전과 같이 처리된다. 모든 키 관리 기능은 사용자 수준에서 구현되었다. TCFS는 Linux, NetBSD, OpenBSD에 구현되었으며, 원시 코드가 공개적으로 제공된다. 리눅스 버전은 NFS 클라이언트를 암호화 기능을 갖도록 확장하였으며, 원격 NFS 서버는 저장 파일 시스템으로서 동작한다. 클라이언트와 서버는 이 NFS 프로토콜을 통해서 통신하며, 사용자가 TCFS에게 파일의 암호화 및 복호화를 위한 키를 제공한다. 사용자 수준의 응용은 파일 시스템을 mount하는 시점에 키를 전달하고 ioctl 시스템 호출을 통하여 TCFS에게 지시를 하게 된다. 버전 2.0이후에서는 리눅스 커널 모듈(LKM)로 개발되어 암호화 서비스가 동적인 커널 모듈로 적재되고 실행된다. 사용자는 파일이나 디렉터리 마다 다른 암호 알고리즘 모듈(3DES, IDEA, Blowfish, RC5 등)을 선택할 수 있다.

2.5 관련 연구 분석

Unix의 crypt 프로그램과 같은 응용 수준에서의 암호화 도구는 조직에서 어떤 파일이 중요하다 할지라도 사용자가 선택적으로 지정을 해야만 그 파일에 대하여 암호화 및 복호화를 실행하게 된다. 더구나 사용자 수준에서 실행하는 파일 데이터의 암호화 도구는 사용자가 사용하기에 귀찮은 일이며, 암호화 및 복호화 처리 속도의 저하로 오버헤드가 크고, 보안의 측면에서도 취약점을 내포하고 있다. 응용 수준에서 내장된 암호화 기능을 갖는 응용 프로그램 내에서 통합하여 암호화 및 복호화를 수행한다 할지라도 파일을 열고 읽거나 쓸 때마다 키를 요구할 때 주입을 시켜주어야만 하며, 중요한 파일이라 할지라도 강제적으로 자동적인 암호화가 이루어 지지 않는다.

한편 운영체제의 시스템 수준에서의 암호화 방법 중의 하나인 CFS는 데이터가 신뢰성이 없는 장치를 통과하기 전에 그 데이터를 암호화하며, 신뢰성 장치에 들어가자마자 그 데이터를 복호화하여 평문으로 존재하게 된다. CFS 사용자는 키를 가지고 보호하

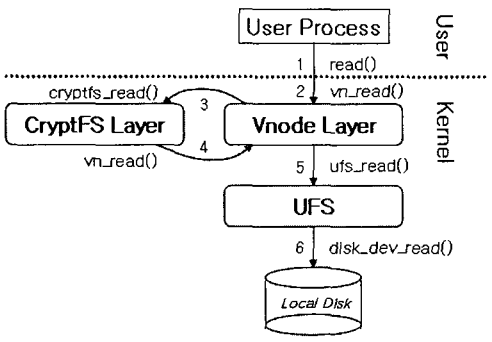

그림 2. Vnode에서 스태킹이 가능한 파일시스템

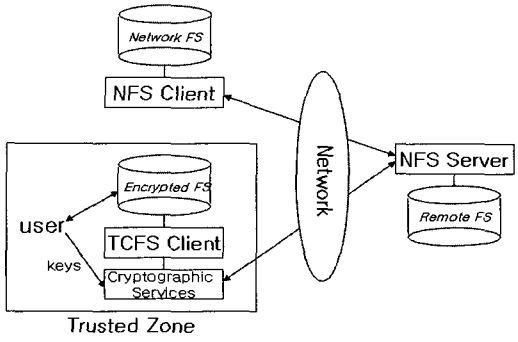

그림 3. TCFS 시스템 구성도

고자 하는 디렉터리를 만들며, 그 보호 디렉터리 내에 있는 각각의 파일들은 자동으로 암호화된다. 그러나 CFS의 주요 문제점은 다음과 같다. 첫째, CFS는 사용자에게 완전히 투명하지 않다. 암호화된 디렉터리는 사용자에 의해서 접근될 수 있는 특정한 디렉터리에 명시적으로 연결되어야만 한다. 둘째, 암호화 단위는 디렉터리 수준이다. 이것은 암호화된 디렉터리를 소유하고 있는 사용자는 반드시 그 디렉터리의 키를 기억하고 있어야만 함을 의미한다. 더욱이 암호화된 디렉터리 내의 모든 파일이 암호화가 이루어진다. 셋째, CFS는 사용자 응용 프로그램으로서 구현되었다. 긍정적 측면에서 이 접근 방법은 다른 OS에 이식하기에 용이하다. 부정적인 측면에서는 이 방법은 클라이언트 시스템에 대하여 공격에 대한 취약성을 증가시키게 되며, 성능을 저하시키게 된다. 넷째, CFS는 보호 대상 자원에 대한 그룹의 공유를 허락하지 않으며, 데이터 인증을 제공하지도 않는다. 그리고 CryptFS는 데이터의 무결성을 보장하지 못하며, 암호화 파일 시스템에서 비 암호화 파일을 허락하지 않는다. 이것은 성능에 적지 않은 영향을 미친다. 예로서 CryptFS는 그 파일이 암호화될 것인지 아닌지에 대한 선택이 불가하며, read 시 블록의 무결성에 대한 확인을 필요로 하지 않는다. 특히 CryptFS의 암호화 파일에 대한 그룹 간의 공유에 대한 제약은 큰 문제로 지적될 수 있다. 다섯째, TCFS는 사용자가 어떤 파일을 암호화할 것인지, 어떤 파일은 평문으로 그대로 놔둘 것인지를 사용자가 선택할 수 있도록 하며, 리눅스 TCFS 구현은 클라이언트와 서버가 동일한 호스트에서 동작하도록 되어 있어 NFS 프로토콜을 통하여 통신을 할 때 속도가 떨어지는 문제점을 갖고 있다.

이제까지 살펴본 응용 수준 및 시스템 수준에서의 모든 파일 암호화시스템들은 공통적으로 파일 또는 데이터 스트림 전체에 대하여 일괄적으로 암호화 및 복호화를 수행함에 따라서 많은 성능상의 오버헤드가 불가피하게 존재하게 된다. 위와 같은 분석에 의해서 본 논문에서는 불필요한 암복호화 프로세스를 줄임으로써 그 성능을 현저하게 개선할 수 있는 커널수준에서 부분적인 암복호화가 자동으로 수행되는 특성을 갖는 방안을 제안한다.

Ⅲ. 커널 수준 파일 암호화 메커니즘 설계

커널 모드에서 암호화 기능을 제공하기 위해서, 본 논문에서는 그림 4와 같이 운영체제 커널에서 동작하는 시스템 호출에서 중요 파일의 암호화 및 복호화가 자동적으로 행해지도록 설계하고, 컴퓨터 파일 시스템 내에 존재하는 어떤 비밀문서가 포함되어 있는 파일 중에서 보안등급을 나타내는 특정 필드의 값이 일정 보안등급 이상의 레이블 값을 갖게 되면 이 파일은 비밀로 간주하여 디스크 상에 저장할 때는 반드시 자동으로 암호화가 이루어지도록 하였다.

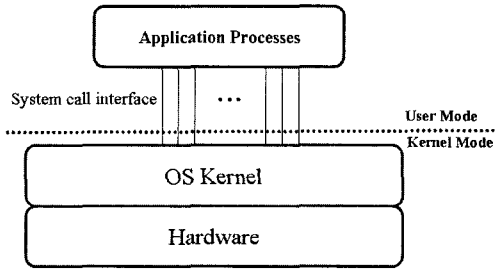

그림 4. 시스템호출과 OS Kernel

3.1 파일 암호화 기본 구조

운영체제 혹은 응용 프로그램 수준에서 이 비밀 파일을 읽고, 쓰고, 파일의 포인터를 이동할 때마다 커널 모드에서 사용자에게는 투명하게 자동적 암호화, 복호화가 부분적으로 필요한 만큼만 이루어지며, 키 관리 또한 운영체제의 커널 수준에서 투명하게 이루어짐으로써 매번 사용자가 키를 주입하는 것과 같은 사용자의 개입에 따른 불편함을 제거하도록 설계하였다.

그림 5는 커널에서 쓰기 혹은 읽기 시스템 호출을 수행할 때 부분적 데이터 블록만을 암호화 혹은 복호화를 실행하도록 하기 위해서, 주 기억장치 내에서 구성되는 데이터 블록 구조도로서, 각 블록 간의 연결은 처리 효율성을 위해 이중 연결리스트(Doubly Linked List) 구조로 구성하였다. 그림 5에서 나타내는 구성 요소 중 위치는 파일 포인터가 현재 위치한 곳, Data 수는 해당 블록 내의 데이터 바이트의 수, Read/Write 수는 읽거나 쓰기위해서 블록 내에 구성한 바이트 수, Previous block은 앞의 블록을 가리키는 포인터, Next block은 다음 블록의 포인터를 각각 나타낸다. 또한 각 데이터 블록의 크기는 본 논문에서 사용하는 암호화 알고리즘의 한 단위 블록 크기인 16바이트(128비트) 크기로 구성하였다. 한 블록을 16바이트(128비트)로 설계한 이유는, 암호화/복호

화의 기본 단위인 128비트 방식의 SEED 알고리즘에 가장 적합하도록 하기 위함이다. 따라서 3DES, AES 등 다른 암호화 알고리즘에 대하여 적용하고자 할 때는 해당 암호화 알고리즘의 암호화 기본 단위에 맞도록 최적화 블록 크기로 설계되어야 할 것이다.

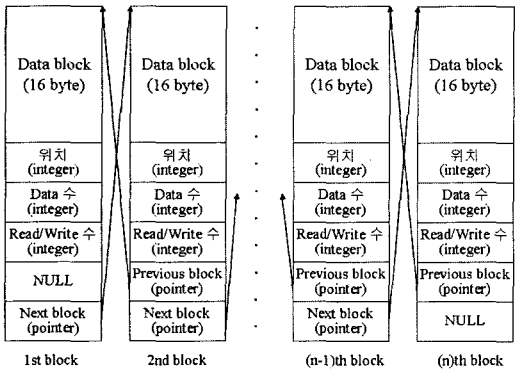

그림 5. 주 기억장치 내의 데이터 블록 구조도

3.2 파일 암호화 -Write

본 암호화 메커니즘에 의하여 커널에서 행해지는 write 시스템 호출(sys_write)의 동작을 예를 들어 설명하자면 다음과 같다. 사용자 프로세스로부터 sys_write(fd, buf, 40) 시스템호출로 40바이트 데이터를 암호화하여 파일에 기록한 파일의 예가 그림 6과 같다고 하자. 편의상 파일에 암호화된 내용은 평문으로 표시하였다. 이때 제1블록(A), 2블록(B) 및 3블록(C)의 각 마지막 16번째 바이트에는 15, 15, 10이 기록되어 있는데 이는, 각 블록의 1번째~15번째 바이트 사이의 암호화된 유효 바이트 수를 나타낸다. 그리고 3블록의 11~15번째 바이트 자리는 NULL로서, 이후에 사용자 프로세스로부터 추가 write 시스템 호출이 있을 경우, NULL의 수를 초과하는 요청의 경우에는 그 블록 내에 데이터가 수록될 수 없으며, 다른 블록을 생성하여 수록하게 된다. 예로서 3블록의 유효 데이터 수가 10이므로, 5바이트를 초과하는 추가 쓰기 요청 시의 데이터 수록은 3블록에서 허용되지 않고 4블록을 생성하여 수록하게 된다. 즉, 마지막 블록의 남은 공간에 저장해야 하는 데이터가 남은 공간보다 큰 경우에는 저장하지 못하고 빈 상태로 남겨 놓게 된다. 이는 초기에 암호화되어 저장되는 초기 암호화 블록들은 마지막 블록을 제외하고는 빈 공간이 없이 모두 채워지나 마지막 블록 16Byte에는 평균적으로 50%인 8Byte가 공백으로 남을 수 있다. 이후에 파일의 중간부분에 대하여 잦은 부분 수정(삭제, 추가)이 이루어짐에 따라 중간 부분에 공간이 생길 수 있으나 이 공간의 양은 무시할 수 있을 만큼 미미하며, 일반적인 기존의 암/복호화 방법처럼 암호화된 파일 전체를 복호화하고 다시 전체를 암호화하는 시간 비용에 비하면 감수할 수 있을 것이다.

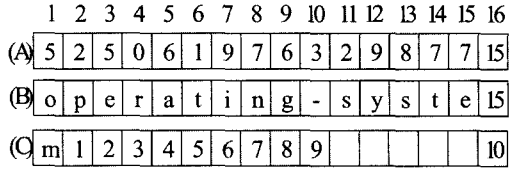

그림 6. 블록 암호화 파일의 예

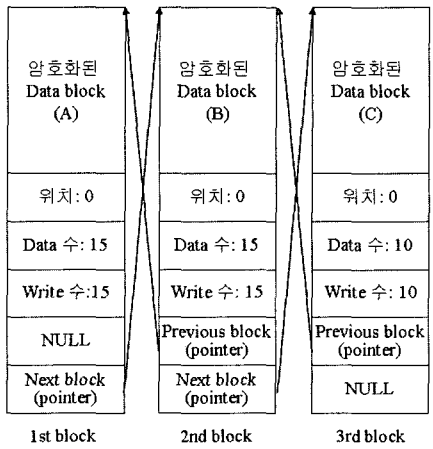

그림 7. 주 기억장치 내 암호화 블록 구조 예

그림 7은 디스크 파일에 쓰기 직전의 주 기억장치 내의 이중 연결리스트 구조로서 40바이트가 암호화되어 3개의 블록 구조로 구성됨을 보여주며, 그림 8에서는 sys_write(fd, buf, count)에 대한 동작 알고리즘을 보여준다.

3.3 파일 암호화 -Read

사용자 프로세스로부터 read, fread, fscanf, getw, getc 등 라이브러리를 통하여 시스템호출을 하게 되면 커널은 사용자 프로세스가 요청한 작업을 수행하는데, 예로 read(fd, buf, 20)는 sys_read (fd, buf, 20) 시스템호출로 바뀌며, 파일 디스크립터 fd가 지정하는 파일로부터 20바이트만큼을 읽어

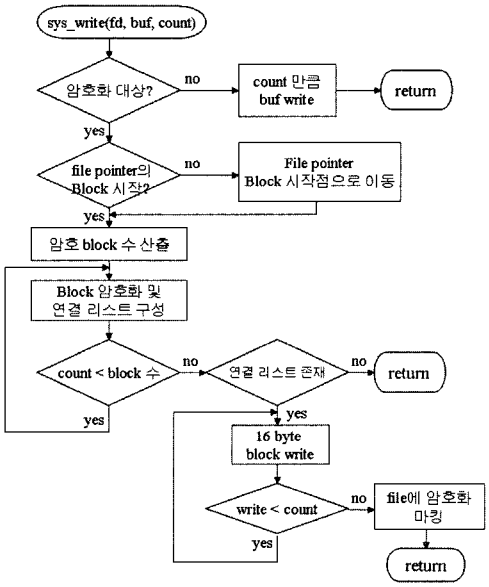

그림 8. 쓰기 시스템 호출의 처리 과정

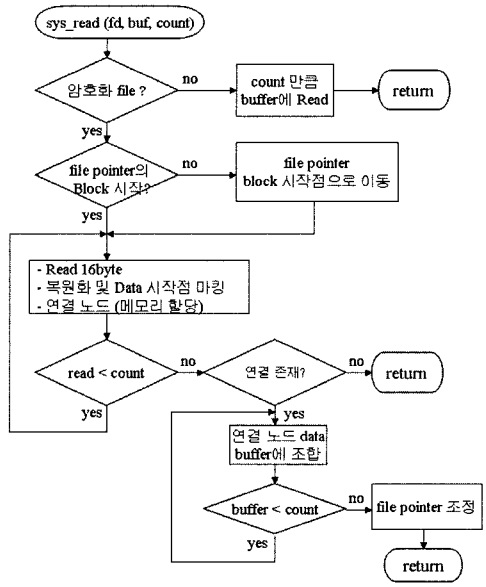

그림 9. 읽기 시스템 호출의 처리 과정

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	b	c	d	e	f	g									7
h	i	j	k	l	m	n	o	p	q	r	s	t			13

그림 10. 48바이트(16바이트x3블록)의 파일을 복호화하여 표시

(A)	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
(B)	a	b	c	d	e	f	g									7
(C)	h	i	j	k	l	m	n	o	p	q	r	s	t			13

그림 11. 12위치에서 20바이트 읽기요청 시의 유효 데이터를 표시

서 기억장치의 buffer에 기억시키라는 read 명령이다. 그림 9에서 보는바와 같이 읽고자 하는 파일이 암호화된 파일인지의 여부는 전달된 파일 디스크립터를 이용하여 파일의 i-node를 구한 후, i-node가 가지고 있는 보안 등급 필드의 MSB를 1로 비트 AND를 함으로써 판단할 수 있다.

암호화된 파일이 아니면 일반 문서로 분류하여 현재의 파일 포인터를 기준으로 읽고자 하는 부분을 lock을 하고, 요구된 수만큼 디스크에서 읽은 후 그 내용을 전달된 buffer에 저장한다. 암호 알고리즘이 블록 단위로 복호화를 하기 때문에 현재의 파일포인터가 블록의 시작점에 위치하는지를 점검한다. 점검은 현재 파일포인터를 16으로 나눠서 나머지가 있으면 블록의 시작점이 아닌 것으로 판단한다. 나머지가 있으면 블록의 시작점으로 파일포인터를 이동시킨다. 그림 10은 암호화된 파일을 48바이트(16바이트 x 3 블록)만큼 복호화 하여 표시 한 것인데 현재의 파일 포인터가 12의 위치에 있다면 파일포인터를 0의 위치로 조정을 하고 변수에 이동 거리를 저장한다. 복호화된 파일의 구조는 앞에서 설명한바와 같이 16 바이트 단위로 되어 있는데 각 블록의 마지막 바이트 자리에 해당 블록의 유효한 데이터 수가 기록되어 있기 때문에 순차적으로 블록 단위로 읽어서 복호화를 한 후 16번째 바이트를 참조해서 유효한 data를 메모리에 이중 연결리스트로 구성하여 저장하고 유효한 데이터 수를 누적시킨다.

12	13	14	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q

그림 13. 사용자 프로세스로 return될 buffer의 내용

0 ②	1	2	3	4	5	6	7	8	9	10	11	12 ①	13	14	15
a ③	b	c	d	e	f	g									7
h ④	i	j	k	l	m	n	o	p	q	r ⑥	s	t			13
⑤															

그림 14. 읽기 시스템 호출 이후의 파일 포인터 이동 경로

그림 11은 파일을 사용자 모드의 프로세스로부터 20바이트 읽기 시스템호출이 있을 때 순차적으로 블록 단위 읽기와 복호화를 한 후 유효한 데이터에 대해서 음영으로 표시를 하였으며, 이때의 메모리에 구성될 연결리스트의 모습을 그림 12에 나타내었다.

메모리에 이중 연결리스트를 한 블록씩 구성을 하면서 바이트 수를 누적하고 이 수가 사용자 프로세스로부터 요청된 count보다 작다면, 위의 과정을 반복하여 구성한다. 앞의 작업 결과에서 첫 연결리스트가 NULL인지를 비교하며, 비교 값이 NULL이라면 사용자 프로세스로 return한다. 메모리에 구성된 이중 연결리스트 (A)(B)(C)의 유효한 데이터를 사용자의 프로세스로부터 전달받은 buf에 순차적으로 복사한다.

그림 13은 작업 후의 buffer 내용이다. 그림 14는 앞에서의 과정에 대한 파일 포인터의 이동과정을 보여준다. 사용자 모드 프로세서로 복귀하기 전에 파일 포인터를 ⑥ 위치로 옮기며, 모든 할당된 자원을 해제한 후 사용자 프로세서로 복귀한다.

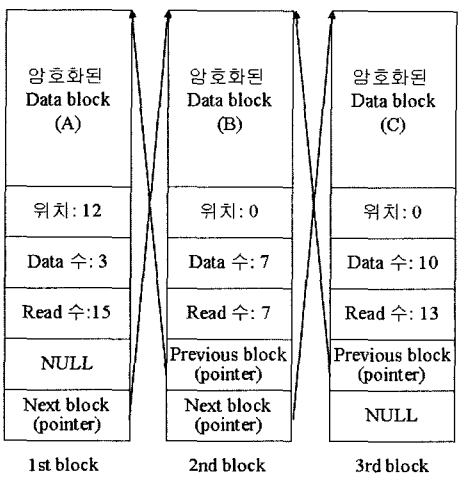

그림 12. 20바이트 복호화 블록 기억장치 구성

3.4 암호화 동작 -파일 포인터 이동

커널 모드의 파일 포인터 이동 시스템 호출인 sys_lseek는 사용자 모드의 프로세스가 커널에 파일 포인터 이동 서비스를 요청할 때 사용되는 시스템 호출이다. [그림 15]는 sys_lseek의 처리 과정을 나타낸 흐름도이며, 사용자 프로세스에서 fseek, lseek 라이브러리 함수를 통하여 시스템 호출을 하면 커널은 사용자의 프로세스가 요청한 작업을 수행한다.

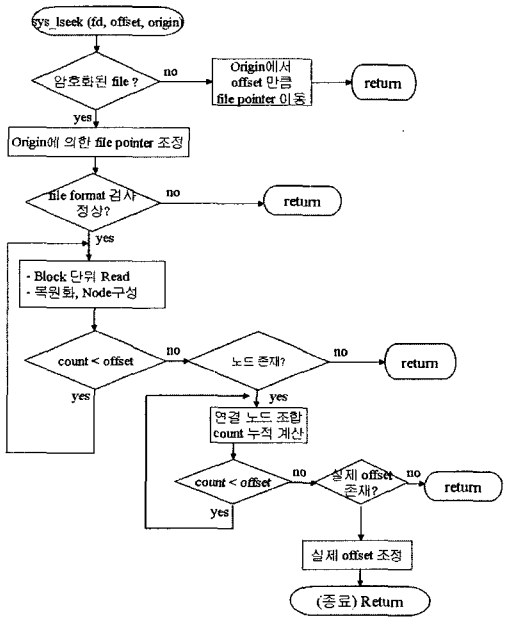

그림 15. 파일포인터 이동 시스템호출의 처리 과정

예컨대 lseek(fd, offset, origin)는 파일 디스크립터 fd가 지정하는 파일의 원래의 파일 포인터 위치인 origin으로부터 offset만큼의 바이트만큼을 이

0 ②	1	2	3	4	5	6	7	8	9	10	11	12 ①	13	14	15
a ③	b	c	d	e	f	g									7
h ④	i	j	k	l	m	n	o	p	q	r	s	t			13
⑤															

그림 16. +offset의 경우 파일 포인터 이동의 예

0 ⑥	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a ④⑦	b	c	d	e	f	g									7
h ②⑤	i	j	k	l	m	n	o	p	q	r ①	s	t			13
③															

그림 17. -offset의 경우 파일 포인터 이동의 예

동을 실행한다. 본 논문에서 수정된 파일 포인터이동 시스템 호출에서는 읽고자 하는 파일이 암호화된 파일인지의 여부는 전달된 파일 디스크립터를 이용하여, 파일의 i-node를 구하고 그 i-node가 가지고 있는 보안 등급 필드의 MSB를 1로 비트 AND를 함으로써 판단할 수 있다. 암호화된 파일이 아니면 일반 문서로 분류하여 사용자 프로세스로부터 받은 origin을 기준으로 offset만큼 파일포인터를 이동한다. 사용자 프로세스로부터 전달된 origin이 2이면 파일의 끝으로, 0이면 파일의 처음으로 파일 포인터를 옮긴다. 암호화된 파일은 블록 단위로 읽어서 블록의 마지막 바이트를 참조해야 유효한 데이터를 알 수 있는 특성이 있으므로 읽기 위한 준비 단계로 조정된 파일 포인터가 파일의 끝이면 -16을 한다. 파일 크기를 16으로 나누어서 나머지가 있는지를 점검한다. 나머지가 있으면 파일이 정상이 아니므로 사용자 프로세스로 return한다. 암호화된 파일의 구조는 16 바이트 단위의 블록들로 구성이 되어 있는데 각 블록의 마지막 바이트 자리에 해당 블록의 유효한 데이터 수가 기록되어 있기 때문에 순차적으로 블록 단위로 읽어서 복호화를 한 후 16번째 바이트를 참조해서 유효한 데이터를 메모리에 연결리스트로 구성하여 저장하고 유효한 데이터 수를 누적시킨다. 여기에서 사용자 프로세스로부터 전달받은 offset에 따라서 읽는 방법이 틀린데 offset이 양의 숫자라면 16 바이트 씩 읽어 가면 되지만 음의 숫자라면 16 바이트를 읽고 32 바이트의 마이너스 포인터 조정이 필요하다.

그림 16은 +offset의 경우, 파일 포인터의 이동 예를 보여주며, 그림 17은 -offset인 경우의 파일 포인터 이동 예를 보여준다. 그림 16에서 파일 포인터 ①의 위치에서 20 바이트를 파일 끝 쪽으로 옮기는 요청이 있다면, 작업 순서는 먼저 ②로 파일 포인터를 옮긴 후, 16 바이트씩 읽기 때문에 파일 포인터 이동 과정은 ①,②,③,④,⑤와 같다. 그림 17에서는 파일 포인터 ①의 위치에서 20 바이트를 파일 앞쪽으로 옮기는 요청이 있다면, 작업 순서는 먼저 ②로 옮긴 후, 16 바이트를 읽고 다시 32 바이트 앞쪽으로 옮겨서 16 바이트를 읽는 과정이 반복되기 때문에 파일 포인터의 이동 과정은 ①,②,③,④,⑤,⑥,⑦,⑧순으로 진행된다. 그림 18은 위와 같은 방법으로 블록 단위 읽기와 복호화를 한 후 메모리에 구성될 연결 리스트의 모습을 보여준다. 메모리에 연결리

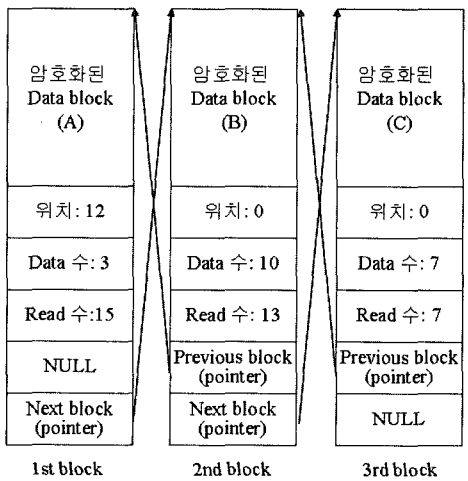

그림 18. ±offset의 경우 파일포인터의 이동을 통한 블록 단위 읽기와 복호화 후의 메모리에 구성한 이중 연결리스트

스트를 한 블록씩 구성을 하면서 유효한 바이트 수를 누적하고 이 수가 사용자 프로세스로부터 요청된 offset보다 작다면, 위의 과정을 반복 작업하여 (A)(B)(C)를 구성한다. 앞의 작업 결과에서 첫 연결리스트가 NULL인지를 비교하여 비교 값이 NULL이라면 사용자 프로세스로 return한다. 메모리에 구성된 연결리스트 (A)(B)(C)의 유효한 바이트 수를 누적 계산하여 사용자의 프로세스로부터 전달받은 offset과 같은지 점검한다. 여기서 구한 offset만큼 파일포인터를 이동시키며, offset이 NULL이면 사용자 프로세스로 복귀한다. 이때는 할당된 자원을 해제하고 사용자 프로세스로 복귀한다.

Ⅳ. 구 현

응용 프로그램(보조 기억장치 내에서 파일에 대하여 읽기, 쓰기를 행하는 모든 프로그램)에서 파일을 새로 작성하거나, 기존 파일에 대하여 데이터를 수정하거나 읽고자할 때, 시스템 라이브러리를 통하여 궁극적으로 운영체제의 시스템호출인 sys_write(), sys_read(), sys_lseek()를 수행하게 된다. 따라서 본 논문에서는 리눅스 kernel 2.2.14 버전에서 이러한 3개의 시스템호출을 수정함으로써 앞서 설계한 암호화 메커니즘에 따라서 128비트 SEED 대칭 키 블록 암호 알고리즘[10]을 이용하여 자동으로 암호화가 수행되도록 구현하였다.

4.1 커널 시스템 호출 수정

커널 모드에서 암호화 기능을 자동으로 제공하기 위해서, 본 논문에서는 그림 19와 같이 리눅스 커널의 VFS(Virtual File System)에서 동작하는 3개의 시스템호출에서 컴퓨터 파일시스템 내에 존재하는 비밀문서가 포함되어 있는 파일의 i-node 내의 필드 중에서 보안등급을 나타내는 특정 필드의 값이 일정 보안등급 이상의 레이블 값을 갖게 되면 이 파일은 비밀로 간주하여 디스크 상에 저장할 때는 반드시 자동으로 암호화가 이루어지도록 구현하였다. 운영체제 혹은 응용 프로그램 수준에서 이 비밀 파일을 읽고, 쓰고, 파일의 포인터를 이동할 때마다 커널 모드에서 사용자에게는 투명하게 자동적 암호화, 복호화가 부분적으로 필요한 만큼만 이루어진다. 일반 사용자 응용 프로그램의 프로세스에서 파일을 새로 작성하거나, 기존에 존재하는 파일에 대하여 데이터를 수정하고자 할 때, 시스템 라이브러리를 경유하여, VFS로 지원되는 ext2, msdos, minix, proc 등의 파일 시스템을 통하여, sys_write, sys_read, sys_lseek를 수행하게 된다. 이러한 시스템 호출들은 파일 시스템 내의 버퍼 캐쉬와 장치 드라이버를 통하여 보조기억장치에 데이터를 쓰거나, 읽거나, 포인터의 위치를 이동하게 된다. 따라서 사용자에게 투명하게, 즉, 암호화의 수행여부를 느끼지 못하게 자동으로 암호화가 진행된다.

본 논문에서 구현한 SEED 블록 암호화 알고리즘은 암호화 및 복호화 블록의 기본 크기 단위를 16(128비트)바이트로 하고, 128비트 대칭 키를 사용하므로 본 논문에서 설계한 16바이트 블록처리에 효과적이다.

4.2 자동 암호화 키 관리

데이터의 블록 암호화 시 사용되는 암호화 키 관리는 비밀 파일에 대한 비밀성을 강화하기 위해 중요하다. 본 구현에 사용된 SEED 암호화 키는 그림

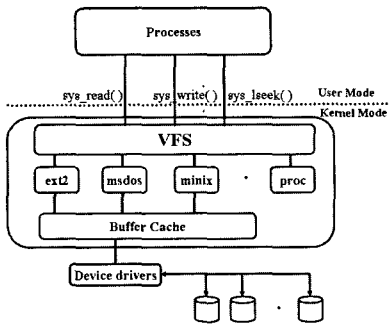

그림 19. 구현된 리눅스 VFS 파일시스템 구조도

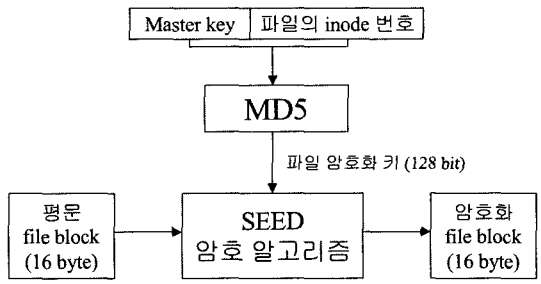

그림 20. 자동 암호화 알고리즘과 키 관리

20에서 보는바와 같이 리눅스 커널 컴파일 시에 보안관리자에 의해서만 주입되는 마스터 키에 추가하여 암호화 대상 파일마다 서로 다른 고유의 암호화 키를 결합하는 방식으로 대상 파일마다 다르게 생성하여 사용한다. 이를 위해, 보안등급이 부여된 파일에 데이터를 암호화하여 쓸 때에는 각 파일의 i-node를 참조하여 각 파일마다 유일한 값을 가지는 데이터, 예컨대 파일 최초 생성 시간 또는 파일마다 주어지는 파일의 고유 번호 등을 추출한다. 그리고 보안관리자가 주입한 커널 메모리 상에만 존재하는 마스터 키와 i-node에서 추출한 고유번호를 결합하여 MD5 해시함수를 통과시킴으로써 128비트의 파일별 개별 암호화 키를 생성하게 된다. 이처럼 보안등급이 부여된 매 파일마다 서로 다른 암호화 키를 사용하게 됨으로써 파일의 보안을 더욱 강화시킬 수 있다.

V. 성능 평가

일반적으로 응용 프로그램에서 파일의 내용을 읽고자하거나, 쓰고자 할 때는 반드시 sys_read, sys_write, sys_lseek와 같은 시스템 호출을 공통적으로 함께 호출하여 사용하게 된다. 따라서 앞서 설명한바와 같이 3개의 시스템 호출(sys_write, sys_read, sys_lseek)을 수정하여 구현한 리눅스 Kernel 2.2.14를 탑재한 Intel i386(1-CPU) 650MHz (256MRAM, HDD: 20GB) Notebook PC 상에서 다음의 3가지 응용 경우에 대하여 처리 성능을 μs 단위까지 측정하여, 3개의 시스템 호출에 대하여 전체적인 효과를 보였다. 표 1은 메모리 상에서 데이터를 임의로 다양한 크기로 발생하게 하여 전체 데이터를 일괄 처리(비암호화, 응용 수준 암호화, 본 커널 수준 암호화 등 3가지 처리 방법으로) 후 디스크에 쓰기를 한 응용 경우의 측정 시간이다.

표 2는 다양한 크기로 암호화된 디스크 파일에서 3가지 처리 방법으로 전체 데이터를 일괄 복호화 후 메모리로 읽기를 한 응용 경우의 측정 시간이다. 표 3은 암호화된 다양한 크기의 디스크 파일에서 임의의 부분 데이터를 읽은 후, 부분 수정하여, 부분 암호화를 수행한 후 저장한 응용 경우의 측정 시간이다. 이 때 처리한 파일의 크기는 최소 1KB부터 최대 1GB로 설정하였는데, 최대 크기는 시험 컴퓨터의 물리적 메모리 및 디스크의 한계로 말미암아 제한하였다. 각 경우의 처리 방법은 암호화 과정이 전혀 없는 비 암호화, 응용 수준에서의 SEED 암호화, 그리고 본 구현의 커널 수준에서의 SEED 암호화 등 3가지로 한정하였다.

표 1에서 보는 바와 같이, 예컨대 메모리 상에서 2KB의 데이터를 생성 후, 암호화를 하지 않고 디스크에 저장할 때는 22μs가 소요되었고, 응용 프로그램 수준에서 일괄적으로 SEED 알고리즘으로 암호화하여 저장 시에는 731μs가 소요되었다. 2KB 크기의 파일에 대하여 본 구현 부분 암호화 방법으로는 309μs가 소요되어, 본 논문의 커널 모드에서 암호화 처리한 방법이 응용 수준에서 암호화 처리 시간의 42.3% 수준이다. 평균적으로 응용 수준 대비 44.7% 수준으로서 2배 이상 빠르게 처리됨을 알 수 있다. 이는 본 논문에서 제안한 부분 암호화 구조 자체의 효율성보다는 디스크 I/O 시 입출력 블록의 단위가 응용 수준에서는 16바이트 단위인 반면에, 본 커널 수준에서는 1024바이트 입출력 블록 단위에 기인한다.

표 1. 메모리 상에서 일괄 처리 후 디스크에 쓰기 한 경우의 처리 시간 (단위: second)

파일크기 / 처리방법	1KB	2KB	10KB	100KB	1MB	10MB	100MB	1GB	평균
비암호화	0.000038	0.000022	0.000053	0.000454	0.007224	0.079585	2.762418	48.467243	(커널수준 ÷ 응용수준) *100
응용수준 암호화	0.000392	0.000731	0.003635	0.036419	0.374791	3.803668	38.731242	389.061074	
커널수준 암호화	0.000200	0.000309	0.001482	0.015464	0.160451	1.706875	17.963369	182.005402	
커널수준 /응용수준	51.0%	42.3%	40.8%	42.5%	42.8%	44.9%	46.4%	46.8%	44.7%

표 2. 암호화된 파일을 일괄 복호화 후 메모리로 읽기 한 경우의 처리 시간 (단위: second)

파일크기 / 처리방법	1KB	2KB	10KB	100KB	1MB	10MB	100MB	1GB	평균
비암호화	0.000024	0.000010	0.000034	0.000482	0.008794	0.091215	3.089664	52.366774	(커널수준÷응용수준) * 100
응용수준 암호화	0.000178	0.000352	0.001768	0.017998	0.181175	1.805478	18.589823	185.314205	
커널수준 암호화	0.000133	0.000258	0.001308	0.013185	0.134306	1.342944	13.639330	137.363490	
커널수준 /응용수준	74.7%	73.3%	74.0%	73.3%	74.1%	74.4%	73.4%	74.1%	73.9%

표 3. 암호화된 파일을 부분 읽기, 부분 수정, 부분 암호화 저장 시의 처리 시간 (단위: second)

파일크기 / 처리방법	1KB	2KB	10KB	100KB	1MB	10MB	100MB	1GB	평균
비암호화	0.000010	0.000005	0.000004	0.000012	0.000021	0.000026	0.000031	0.383934	(커널수준 ÷ 응용수준) * 100
응용수준 암호화	0.000571	0.001063	0.005286	0.056677	0.573699	5.933652	84.055220	882.812277	
커널수준 암호화	0.000030	0.000030	0.000032	0.000078	0.000224	0.001530	0.015946	0.180134	
커널수준 /응용수준	5.25	2.82	0.61	0.14	0.04	0.03	0.02	0.02	1.12%

표 2에서 보는바와 같이 암호화된 파일을 일괄 복호화 후 메모리로 읽기를 한 경우는, 평균적으로 응용 수준 대비 73.9% 수준으로 약간 빠르게 처리되었음을 알 수 있다. 표 1의 44.7%인 점에 비하여 처리 효율이 상대적으로 낮은 이유는, 읽기의 경우엔 쓰기보다 i-node의 정보를 통하여 비밀 파일 여부 등 부수적인 체크 루틴을 더 많이 수행하기 때문이다. 표 3의 경우는 크기별 암호화된 파일에서 부분 데이터만을 읽고, 부분 데이터만을 수정한 후, 그 부분 블록만을 다시 암호화하여 저장하는 경우로서, 기존 응용 수준에서의 전체 파일 데이터를 복호화한 후, 전체 데이터 파일을 암호화하여 저장하는 처리 방법에 비하여 본 제안 메커니즘이 매우 효과적임을 알 수 있다. 이때 선택한 부분 데이터의 위치는 각 크기 파일 크기 ÷ 1,000을 한 후 지점이며, 이 바이트로부터 수 바이트를 수정하는 방식으로 측정하였으며, 측정 결과는 표 3에서 보는바와 같이 1GB 크기의 경우 응용 수준의 0.02%에 불과하며, 평균적으로 응용 수준 대비 1.12%에 불과함을 알 수 있다. 이는 당초 본 논문에서 의도하고 예상했던 대로 파일이 크면 클수록 그 효율은 크게 높아져, 본 메커니즘이 큰 암호화 파일의 부분 편집이나 부분 읽기의 경우에 특히 효과가 있음을 알 수 있다. 그리고 이러한 효과를 얻기 위해서는 추가적 관리 공간(주 기억 장치 및 디스크)이 불가피하게 소요된다. 즉, 부분 암호화/복호화 방법을 위해서 블록체인 형태의 구성 시 위치(2Byte), 블록 내의 데이터 바이트 수(2Byte), Read/Write 수(2Byte), Previous block 포인터(4Byte) 및 Next block 포인터(4Byte) 정보 등의 관리 데이터(Meta Data)가 필요하다. 따라서 각 블록 당 관리정보의 양은 14Byte이다. 예를 들어 1KB 데이터 파일의 경우, 구성되는 블록의 수는 69개(1024Byte/15Byte)로서 이때의 관리정보는 966Byte(14Byte x 69개 블록)로 파일의 94%(966Byte / 1024Byte x 100) 만큼의 부가적인 주 기억 장치 및 디스크 공간이 필요하게 된다. 이 부가적 정보의 양은 보안을 위해서 추가적으로 지불해야 하는 비용으로서, 이

오버헤드는 보안성과 처리 속도의 효과에 비하여 지불을 감수할 수 있을 것으로 판단된다. 그 이유는 최근 메모리 단가가 급격히 줄고, 보안에 대한 투자가 급증하는 추세이기 때문이다.

한편 본 방법에 대하여 가능한 형태의 공격과 이에 대한 보장 보안성을 분석해 보면 다음과 같다. 본 방식에서 존재할 수 있는 보안 위협은, 첫째로 일반적인 암호 시스템에서와 같이 암호화된 파일에 대한 제반 공격(비밀 키 탈취 등)이 있을 수 있으며, 둘째로 OS의 접근제어 공격을 통한 파일 삭제 등을 들 수 있다. 암호화 파일에 대한 공격에 대해서는 이미 SEED가 국내외에서 안전한 표준 암호화 알고리즘으로 검증되고 있는 시점이며, 컴퓨터 시스템 내에서 비밀 키 탈취에 대한 공격에 대해서는 본 구현이 다중등급 보안 운영체제 환경에서 구현된 만큼 시스템 관리자 및 보안 관리자 인증을 거친 경우에만 마스터 키에 접근할 수가 있으며, 파일 비밀 키 또한 이 마스터 키와 해당 비밀 파일의 i-node의 조합을 통하여 필요시마다 MD5 Digest를 통하여 128비트 비밀 키가 생성되어 사용되며, 컴퓨터 시스템 내의 어느 곳에도 저장되지 않는다. 아울러 이 환경에서는 특정 비밀 등급(I, II, III, 대외비 등)을 갖는 비밀 파일만이 본 구현을 통하여 암호화되고 접근이 인가된 사용자별 보안등급 및 보호범주(category)에 대해서만 접근이 가능하며, 이 비밀 파일에 접근 시에는 상세한 로그가 기록되어 관리되므로 접근제어를 통한 암호화 파일에 대한 읽기, 쓰기, 삭제 등에 대한 거부는 보장될 것이다.

Ⅵ. 결 론

매우 큰 암호화된 파일에 대하여 일부만을 수정하고자 할 때는 일반적으로 전체 파일을 복호화하여 수정한 다음에 전체를 다시 암호화하여 저장하게 되는데, 이때의 일괄 처리로 인한 성능 비용은 상당하다. 본 논문에서 제안하는 파일의 부분적 암호화 및 복호화 기능은 기존 연구와는 다르게 사용자에게 투명하며, 응용 프로그램 수준에서 중요한 파일에 대하여 쓰거나, 읽고자 할 때, 혹은 파일 포인터 이동 시에 부분 암호화 및 복호화를 통하여 처리 성능을 크게 향상시킬 수 있는 커널 모드 부분 암호화 메커니즘을 제시하였다. 이 메커니즘을 리눅스 커널 2.2.14에 3개의 시스템 호출인 sys_write(), sys_read(), sys_lseek()를 수정하는 방식으로 구현하여 처리 성능을 측정하고 기존의 응용 수준에서의 처리 성능과 비교하였다. 성능 평가를 통하여 응용 수준 대비 본 제안 방법에 의한 커널 수준에서의 처리 시간은, 일괄 쓰기 경우 평균 44.7%, 일괄 읽기의 경우 평균 73.9%, 부분 수정의 경우 평균 1.12%에 불과함을 알 수 있으며, 이는 본 메커니즘이 매우 효율적임을 알 수 있다. 아울러 본 논문에서 설계하고 구현하여 제시한 커널 모드 부분 암호화 메커니즘은 사용자의 개입 없이도 투명하며 안전하게 각 파일의 비밀성을 유지할 수 있으므로, 비밀 수준의 데이터, 텍스트 문서 파일 암호화에 적합하며, 특히 수시로 수정이 발생하는 큰 비밀문서 파일 보호에 효과적으로 적용이 가능하다. 아울러 보안에 중요한 영향을 미치는 제반 컴퓨터 파일에 모두 확장 적용이 가능하다. 특히 노트북 컴퓨터와 같이 이동 가능한 컴퓨터나 디스크의 분실 혹은 탈취 시에 불법적으로 디스크의 내용이 덤프(Dump)를 통하여 노출된다 하여도 정보 노출을 예방할 수 있다. 따라서 비밀 정보의 누출 위험이 존재하는 어떤 환경에서도 중요 파일의 정보 누출을 막을 수 있는 효과적인 방법을 제공할 수 있다. 또한 하드웨어 장치에 의한 고정된 암호화 알고리즘 사용으로 인한 경직성과 달리 소프트웨어적으로 처리되므로 가변적으로 암호화 알고리즘과 키를 필요에 따라 변경할 수 있는 장점이 있다. 그리고 향후 연구로서, 블록체인의 경우 마지막 블록은 평균적으로 50%인 8Byte가 공백으로 남을 수 있는데, 이를 굳이 해결하자면 일괄처리 형태의 별도 처리 방법으로 주기적으로 파일의 빈 공간을 압축(Garbage Collection)하는 방법을 구현할 수 있으나, 이는 더 큰 오버헤드를 유발할 수 있을 것이므로 더 효과적인 방법 연구가 필요할 것이다.

본 논문에서는 국내에서 개발된 128비트 블록 암호화 알고리즘인 SEED 만을 대상으로 하였으나, 향후 계속적인 연구를 통하여 3DES, IDEA, AES 등 다양한 암호화 알고리즘에 선택적으로 적용하거나, 여러 암호화 알고리즘에 공통적으로 적용할 수 있는 최적의 구조와 메커니즘에 관한 연구가 더 필요하다.

참 고 문 헌

[1] 박태규, "보안 리눅스(Secure Linux) 연구 개발 동향," 한국정보보호학회지, 2003. 8.

[2] 한국정보통신표준협회,128-bit Symmetric Block

Cipher(SEED), 1999. 4.
[3] NSA, Security Enhanced Linux, http://www.nsa.gov/selinux/
[4] M.Blaze, "A Cryptographic File System for Unix," In Proc. 1st ACM Conference on Communica -tion & Computing Security, Nov. 1993.
[5] E.Zadok, I. Badulescu and A. Shender, "CryptFS : A Stackable Vnode Level Encryption File System," 1998.
[6] G. Cattaneo and G. Persiano, "The Design and Implementation of a Transparent Cryptographic Filesystem for UNIX," In Proc. of 2001 USENIX Annual Technical Conference, Jun. 2001.
[7] M. Beck, et al, Linux Kernel Internal, Addison-Wesley, 1996.
[8] O. Pomerantz, " Linux Kernel Module Programming Guide," GPL Licensed book, 1999.
[9] E. Zadok, "FiST: A File System Component Compiler," PhD Thesis, published as Technical Report CUCS-033-97,Computer Science Dept.,Columbia University., APR. 1997.
[10] http://www.tcfs.it, Transparent Cryp -tographic Filesystem

〈著者紹介〉

김 재 환 (Jae-Hwan Kim) 정회원
1983년 2월 서울시립대학교 전자공학과 졸업
1985년 2월 아주대학교 전자공학과 석사
1987년 7월 ~ 현재 한국수자원공사
2004년 2월 ~ 전북대학교 정보보호공학과 박사과정
〈관심분야〉 네트워크 보안, 유비쿼터스센서네트워크 보안

박 태 규 (Tae-Kyou Park) 종신회원
1980년 10월: 경북대학교 전자계산기공학과 졸업
1989년 8월: 충남대학교 전산학과 석사
1996년 2월: 성균관대학교 정보공학과 박사
1981년 2월~1982년 12월 한국국방연구원 연구원
1982년12월~1992년 2월 한국전자통신연구원 선임연구원(부호5실장)
1997년 1월~1998년 1월: Univ of Western Sydney, Post-doc.
1992년 3월~ 현재 한서대학교 교수
〈관심분야〉 보안 운영체제, 컴퓨터 보안

조 기 환 (Gi-Hwan Cho) 정회원
1985년 전남대학교 계산통계학과 졸업
1987년 서울대학교 계산통계학과 석사
1996년 영국 Newcastle 대학교 전산학과 박사
1987년~1997년 한국전자통신연구원 선임연구원
1997년~1999년 목포대학교 컴퓨터과학과 전임강사
1999년~현재 전북대학교 전자정보공학부 부교수
〈관심분야〉 이동컴퓨팅, 네트워크 보안, 무선인터넷, 컴퓨터통신