韓國컴퓨터情報學會 論文誌
第14卷 第7號, 2009. 7.

2009-14-7-4-4

The method for protecting contents on a multimedia system

Seong-Ki Kim*

멀티미디어 시스템에서 콘텐츠를 보호하기 위한 방법

김 성 기*

Abstract

As a DRM is recently being removed from many sites, the content protection on a video server becomes important. However, many protection methods have their own limitations, or aren't used due to the deterioration of the streaming performance. This paper proposes a content protection method that uses both the eCryptFS and the SELinux at the same time, and measures the performance of the proposed method by using various benchmarks. Then, this paper verifies that the method doesn't significantly decrease the streaming performance although the proposed method decreases the other performances, so it can be used for the content protection in a multimedia system.

요 약

DRM이 최근에 많은 사이트에서 제거되어짐에 따라 비디오 서버 상에서 콘텐츠 보호는 중요해졌다. 그러나 기존에 존재하는 보호 방법들은 한계 들을 가지거나 스트리밍 성능을 저하시키므로 사용되어지지 않는다. 본 논문은 eCryptFS와 SELinux를 사용하는 콘텐츠 보호 방법을 제안하고 다양한 벤치마크들을 사용하여 제안한 방법의 성능을 측정한다. 그런 후 본 논문은 제안한 방법이 다른 성능 들은 저하시키지만 스트리밍 성능은 저하시키지 않아서 콘텐츠 보호를 위하여 사용할 수 있다는 것을 증명한다.

▸ Keyword : Content protection, Multimedia system, DRM, eCryptFS, SELinux

• 제1저자 : 김성기
• 투고일 : 2009. 06. 01, 심사일 : 2009. 07. 01, 게재확정일 : 2009. 07. 17.
* 삼성전자 DS부문 책임연구원

Ⅰ. Introduction

Protection against the Internet piracy on a client and an illegal access on a video server becomes important because contents are precious assets of their providers, and the entire duplications are difficult differently from a web system. However, the proliferation of both the Internet and the Peer-To-Peer (P2P) networks, and various attacks make content protection difficult. Thus, powerful protection must be applied to the multimedia system.

An attacker can try to illegally access the contents largely at a server and a client, and the DRM (Digital Rights Management) protects the contents at a client. However, the DRM tends to be removed [1][2] due to its complexity. Within this trend, any determined methods don't exist for a content protection at a server.

In order to protect a server, technologies such as a firewall, an IDS (Intrusion Detection System) and an audit trail have improved. However, these technologies have their own fundamental limitations. For instance, a firewall can't protect a system against an internal intruder. In addition, once the root privilege of a target system is gained by an intruder, there are no other ways to protect the system only with a firewall. An audit trail has the problem that it can't prevent an intrusion but only help trace the intruder by leaving messages. In addition, they can't protect themselves from being killed because they operate at a user level. These problems lead many researchers and vendors to place a focus on the protection at a kernel level.

Despite the advantages of a protection at the kernel level, they haven't been also used for a multimedia system because of the deterioration of its performance. For example, the encryption by a cryptographic file system or the permission check by an access control can decrease the streaming performance, and significantly decrease the number of concurrently served users. This paper verifies that the streaming performance isn't significantly decreased even when both of them are used at the same time.

This paper is organized as follows: Section 2 describes the related works for a content protection. Section 3 proposes the eCryptFS and the SELinux for a multimedia system, and verifies that their usages don't significantly decrease the streaming performance. Section 4 concludes all of these works.

Ⅱ. Related Works

This section describes DRM [3], cryptographic file systems such as Cryptfs [4] and eCryptFS [5], and SELinux [6].

2.1 DRM

To protect content on a client, DRM (Digital Rights Management) emerged as one of approaches. Rosenblatt and Dykstra [3] defined the DRM from two points of view. From the narrower viewpoint, DRM is defined as follows: DRM is the persistent protection of digital data. This means that DRM protects digital content by an encryption and an access control that determines a permitted user. From the broader viewpoint, DRM is defined as follows: DRM is everything that can be done to define, manage, and track the rights to digital content. This means that DRM includes all of the technologies that can be used for managing and tracking digital contents on machines connected through the Internet [3][7].

DRM protects content from an illegal usage via an encryption and an access control on a client, and tracks the content after distribution. Figure 1 shows the DRM process.

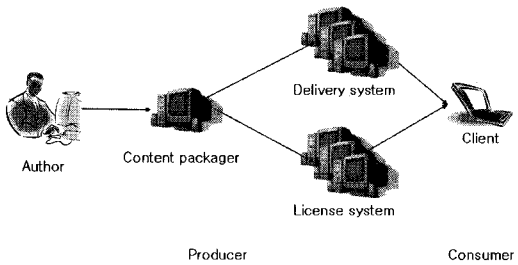

Fig 1. DRM process

In Figure 1, when an author finishes creating content that will be served, he or she sends the content to a content packager. The content packager adds some metadata to the received content by using watermarking technology for

controlling the user access when the content is downloaded, and encrypts the content. The content is published to clients when delivered to a delivery system for a VOD (Video on Demand) or a broadcasting service. License information, including the decryption key, is delivered to a license system. When a client wants to watch the content, the client verifies whether the client has an appropriate permission to watch it or not. If the client has the permission, the client downloads decoding information, including the decryption key from the license system. The client begins playing the content by using the downloaded decoding information. In a summary, DRM is an external access control protecting contents by encryption.

Through watermarked information within the metadata, DRM can place limitations on the playing counts, the playing devices, the playing users, and allow accesses to only those users who pay the appropriate value for the requested content.

However, DRM is recently viewed skeptically [8], and it recently tends to be removed from many sites [1][2] due to its complexity.

2.2 CryptFS and eCryptFS

CryptFS is a stackable cryptographic file system that can operate on top of various file systems. CryptFS supports only a single encryption algorithm, Blowfish [9], and implements only a limited key management scheme. eCryptFS [5] extends these CryptFS, and additionally supports the AES [10], CAST [11] and TwoFish [12] algorithms. Figure 2 shows the call paths of both CryptFS and eCryptFS.

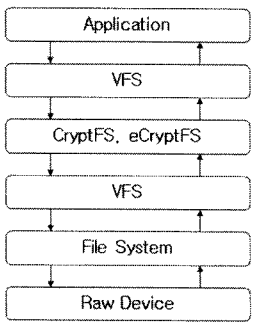

Fig 2. Call paths of CryptFS and eCryptFS

In Figure 2, whenever an application or a user makes a request to a file system, the request is translated into a vnode level call, which invokes its equivalence that encrypts or decrypts the contents, and invokes the original file system through the vnode level call [13].

Although CryptFS and eCryptFS are fast and secure because they are implemented at the kernel level, they have a common weakness that they can be controlled by an intruder who already gains the root privilege because they are implemented as a dynamically loadable kernel module, and don't have any mechanisms to control the unload system calls.

eCryptFS was integrated into a Linux kernel since a version 2.6.19.

2.3 SELinux

NSA (National Security Agency) and SCC (Secure Computing Corporation) developed SELinux (Security Enhanced Linux) that implemented a Flask architecture [6], and the goal of the architecture was to support both policy-flexible and transparent architecture for a variety of access controls. This goal was achieved by separating a security server from an object manager as shown in Figure 3.

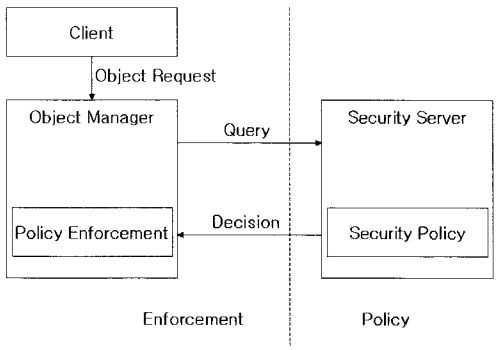

Fig 3. Flask architecture

The Flask architecture [6] separates a policy enforcement logic from a security policy logic by distinguishing a security server from an object manager. The object manager manages objects that clients want to access. The security server determines whether an access is legal or not based on a server policy.

If a client wants to access an object that is managed by the object manager, the client firstly makes a request to the object manager. The object manager determines whether the access is legal or not after querying to the security server with MAC (Mandatory Access Control) [14], RBAC (Role Based Access Control) [15] or TE (Type Enforcement). Because each security server can have a different policy, SELinux can support any security policies as far as a security server is implemented while following pre-determined interfaces and the policies.

SELinux was integrated into a Linux kernel since a version 2.6.

III. Proposal of eCryptFS and SELinux as a content protection method

This section proposes a content protection method, presents the overheads when using the SELinux or the eCryptFS, and presents that using both of them at the same time doesn't seriously decrease the streaming performance.

3.1 Description

All of the access controls have a common weakness that they are vulnerable to both a low-level access and a physical attack. If an intruder finds a way to access the data under the level of an access control, then the intruder can access all of its information. In addition, if an intruder can go outside with the protected disk, physically attach the disk to another system without any access controls, and mount the disk, then all of its information can be revealed. These low-level and physical problems can be addressed by using a decryption or an encryption when reading or writing data. In this case, even if an intruder accesses the data under the level of an access control, or steals the disk, then the intruder can't see its information due to an encryption.

In the aspect of a cryptographic file system, a cryptographic file system can't protect itself because the file system is built as a dynamically loadable kernel module or a user-level application without any self-protection mechanisms. This self-protection problem can be addressed by controlling the unload or the kill privilege of an access control. In other words, when a cryptographic file system is used with an access control, these two mechanisms can complement each other.

To complement each other, we proposed that used these two mechanisms at the same time. Among the available cryptographic file systems and the available access controls, we proposed the SELinux and the eCryptFS because they were integrated into a kernel, and they supported various algorithms.

3.2 Performance

With security enhancing mechanisms, a system has no choice but to increase overheads because of the permission check as well as both the encryption and the decryption. This subsection describes the overheads of the eCryptFS and the SELinux by using the Unixbench [16], Lmbench [17] and Bonnie++ [18]. This subsection also describes the overheads during the streaming process when both the eCryptFS and the SELinux are used at the same time.

3.2.1 Unixbench

In Unixbench, system call performs a fixed number of system calls. Pipe throughput measures the communicating ability through a pipe among processes. Pipebased context switching shows the communicating ability between a parent process and a child process. Process creation counts the number of children that a process can fork. Execl throughput replaces a process with a new one. FR, FW and FC measure the number of characters read from a file, written and copied to a file within a given time. Lastly, shell scripts execute a shell script via concurrently running processes [16].

Table 1 shows the overheads measured by Unixbench when the system has the SELinux or the eCryptFS with the written encryption algorithm. All of the numbers are the overheads by percent (%) when compared with the system without any security mechanisms such as the SELinux and the eCryptFS. These overheads are shown by using the Table because a table is more visible than the Figure due to many rows. These overheads were measured after applying the SELinux or eCryptFS with the encryption algorithm to the test directory of the Unixbench. The Unixbench couldn't measure the overheads of both the SELinux and the

eCryptFS at the same time because the Unixbench didn't work when both of them were simultaneously applied to the test directory. They didn't operate at the same time when writing to a file in the encrypted directory by the eCryptFS. The version of Unixbench was 4.0, and the overheads were measured on the Linux version 2.6.23.9-85.fc8 on Pentium 4 2.6 Ghz and 512 MB RAM.

Table 1. Overheads by Unixbench

%	SELinux	eCryptFS				
		Cast5	Aes128	Twofish	Blowfish	3des
System Call Overhead	-0.32	0.71	-0.18	-0.27	0.19	0.41
Pipe Throughput	8.88	0.48	-0.1	0.14	-0.23	-0.13
Pipe-based Context Switching	6.12	0.07	-1.24	-0.12	0.24	0
Process Creation	0.88	4.45	-1.77	-0.92	-0.81	-0.98
Execl Throughput	12.23	0.3	0.01	0.26	0.57	0.65
FR 1024 bufsize 2000 maxblocks	0.24	5.59	4.43	5.72	5.12	5.22
FW 1024 bufsize 2000 maxblocks	1.53	96.21	95.76	97.01	97.09	99.31
FC 1024 bufsize 2000 maxblocks	0.23	94.26	93.59	95.34	95.67	98.94
FR 256 bufsize 500 maxblocks	0.48	6.4	6.11	6.24	6.81	5.91
FW 256 bufsize 500 maxblocks	0.16	96.66	96.03	97.22	97.6	99.37
FC 256 bufsize 500 maxblocks	-0.16	95.14	94.24	95.87	96.12	99.07
FR 4096 bufsize 8000 maxblocks	0.23	4.98	5.52	5.09	6.33	5.16
FW 4096 bufsize 8000 maxblocks	1.49	95.29	94.54	96.2	96.28	99.12
FC 4096 bufsize 8000 maxblocks	2.05	92.42	91.34	93.79	94.02	98.53
Shell Scripts (8 concurrent)	7.8	6.01	6.69	6.41	7.88	14.09

In Table 1, the performances became better in some cases. These performance increases could be thought as a measurement error or a cache effect.

The overheads of both the system call and the process creation weren't meaningfully increased with the SELinux or the eCryptFS. However, the Pipe Throughput, the Pipe-based Context Switching and the Execl Throughput overheads were increased by average 9.07% when the SELinux protected the content. As reasons of these increases, the SELinux checks the permission. The overheads of the eCryptFS were slight in the Pipe Throughput, the Pipe-based Context Switching and the Execl Throughput cases because they weren't related with the eCryptFS.

In the FR, the FW and the FC cases, the SELinux slightly increased overheads by average 0.69% because these cases weren't related with the SELinux. However, the average 96.07% overhead occurred in the FW and the FC cases, and the 5.64% average overhead occurred in the FR case by the eCryptFS. In the case of the Shell scripts, the average overhead by the eCryptFS was similar with the SELinux and was about 8.22%.

The overheads of the FW and the FC were slightly decreased as the buffer size increased because more blocks could be read into a buffer while gathering the decryption overheads. eCryptFS with the 3des showed the worst performance in the FW and FC cases, and eCryptFS with the Aes128 showed the best performance in the FW and FC cases. We omitted the reasons why these differences occurred because the difference among encryption algorithms is out of scope of this paper.

3.2.2 Lmbench

In Lmbench, simple syscall calls a getppid system call. Simple read reads data from /dev/zero, and simple write writes data to /dev/null. Simple stat opens a file and obtains its information. Process fork+exit performs the written operation and measures the elapsed time [17].

Table 2 shows the overheads measured by Lmbench when the system has the SELinux or the eCryptFS with the written encryption algorithm. These overheads were measured after applying the SELinux or the eCryptFS with the algorithm to the file system directory of the Lmbench. The Lmbench couldn't also measure the overheads of both the SELinux and the eCryptFS at the same time because the Lmbench didn't also work when both of them were simultaneously applied to the file system directory. The version of Lmbench was 2.5, and the overheads were measured on the Linux version 2.6.23.9-85.fc8 on Pentium 4 2.6 Ghz and 512 MB RAM.

Table 2. Overheads by Lmbench

%	SELinux	eCryptFS				
		Cast5	Aes128	Twofish	Blowfish	3Des
Simple syscall	-0.21	-0.32	2.31	-0.62	-0.64	0.54
Simple read	0.71	-1.76	0.43	-1.58	-1.63	0.23
Simple write	4.7	0.16	1.44	0.68	0.2	3.27
Simple stat	18.6	12.7	11.22	11.38	11.99	12.32
Process fork+exit	1.5	0.87	0.16	1.47	-0.64	0.76

In Table 2, the performances also became better in some cases due to the reasons that could be thought as a measurement error or a cache effect.

The simple syscall, the simple read, the simple write and the process fork+exit weren't meaningfully increased with the SELinux and the eCryptFS because they weren't related the encryption or the decryption. Although the SELinux checked permissions, it didn't largely decrease these performances.

However, the 18.6% overhead additionally occurred by the SELinux and the 11.92% average overhead occurred by the eCryptFS in the simple stat case due to the file operations in the file system directory.

Also, 3des had the worst performance, and Aes128 had the best performance.

3.2.3 Bonnie++

Bonnie++ includes a lot of simple tests such as a creation, reading and deleting of small files. Bonnie++ measures the speeds of a sequential output, a sequential input and random seeks. The sequential output measures the speeds of per-character write, the block write and the rewrite, and the sequential input measures the speeds of per-character read and block read. The random seeks measures the speed that randomly seeks within a file, reads a block and writes back the 10% of the read block.

Table 3 presents the overheads measured by bonnie++ when the system has the SELinux or the eCryptFS with the written encryption algorithm. All of the numbers are also the overheads by percent (%) when compared with the general system. These overheads were measured after applying the SELinux or the eCryptFS with the encryption algorithm to the test directory of the bonnie++. The Bonnie++ couldn't also measure the overheads of both the SELinux and the eCryptFS at the same time because the Bonnie++ didn't also work when both of them were simultaneously applied to the test directory of Bonnie++. The version of bonnie++ was 1.03c, the number of files was 100, the tested file size was 2048 MByte, and the chunk size was 1024 Byte. The test was also measured on the Linux version 2.6.23.9-85.fc8 on Pentium 4 2.6 Ghz and 512 MB RAM.

Table 3. Overheads by Bonnie++

%		SELinux	eCryptFS				
			Cast5	Aes128	Twofish	Blowfish	3des
Sequential output	Per-character	-0.42	40.74	38.23	45.88	51.41	79.77
	Block	-0.65	71.11	68.53	75.97	77.2	94.33
	Rewrite	0.77	70.83	69.42	75.61	75.82	92.35
Sequential input	Per-character	1.36	68.44	68.46	72.82	71.75	85.96
	Block	-0.02	68.39	62.21	69.21	69.84	85.75
Random	Seek	2.12	10.89	12.72	14.91	19.65	19.44

In Table 3, the performance also became better in some cases due to the reasons that could be also thought as a measurement error or a cache effect.

The overheads only with the SELinux weren't large in all cases. However, the overheads with the eCryptFS were large in most of the cases. The average overhead of the SELinux was 0.53%. The average overheads of the eCryptFS were 51.21%, 77.43% and 76.81% in the per-character write case, the block write case and the rewrite case of the sequential output. The average overheads of the eCryptFS were 73.49% and 71.08% in per-character read case and block read case of the sequential input. The average overhead of the eCryptFS was 15.52% in the random seek case. The eCryptFS with the 3des algorithm also had the worst overhead in most of the cases.

As reasons why the read performance was worse than the measurement by the Unixbench, the Bonnie++ measured all performance before an optimizer realized that it was all bogus [18]. Thus, the Unixbench showed that the read

overhead was much smaller than the write overhead, but the Bonnie++ showed that the read overhead was similar with the write overhead.

3.2.4 Streaming performance

We measured the performance in order to measure the decreasing degree when using the SELinux or the eCryptFS with the written encryption algorithm. Table 4 shows the streaming overhead when the system has no protection mechanisms, only the SELinux, only the eCryptFS, and both the SELinux and eCryptFS at the same time. The server machine ran the Darwin streaming server 5.5 [19] and Linux version 2.6.23.9-85.fc8 on Pentium 4 2.6 Ghz CPU and 512 MByte RAM. The client machine ran the Quicktime player 7.3 and the Windows XP with a service pack 2 on the Pentium 4 2.0 Ghz CPU and the 256 MByte RAM.

2 files were encoded from a single movie, and had the same 320×240 size, the same 25 FPS and the same approximate 37 min length. The only difference between them was that the first file was encoded with 756.24 Kbits/S, 94.53 KByte/S, data rate and 207.24 MByte size, and the second file was encoded with 1.71 Mbits/S, 218.88 KByte/S, data rate and 461.74 MByte size. In Table 4, LQ means a low-quality file, the first file, and HQ means a high-quality file, the second file. We made a request to the high-quality file for 1 hour with repeat option and a request to the low-quality file for another 1 hour also with the repeat option between about 11:20 and about 13:30 while minimizing user or machine interventions. We measured the performance during 6 days while collecting the packets between the server machine and the client machine by the WinDump [20], and analyzed the packets for the data rate.

Table 4. Streaming performance of a high-quality file and a low-quality files to 1 client

KBytes/S		No eCryptFS	eCryptFS				
			Cast5	Aes128	Twofish	Blowfish	3des
HQ	No SELinux	209.80	210.07	209.87	209.97	210.03	210.35
	SELinux	209.89	203.72	209.76	209.78	210.06	208.68
LQ	No SELinux	92.92	92.96	92.93	92.81	93.01	92.91
	SELinux	92.92	93.01	92.98	92.91	92.89	93.05

Table 4 presents the results after dividing the amount of the received data by the time when the streaming server delivers high-quality and low-quality files to 1 client. Overall, Table 4 presents that the transferring speed isn't meaningfully decreased. The high-quality file is encoded with 218.88 KByte/S, and the average transferred speed of the high-quality file is 209.29 KByte/S even with the SELinux or the eCryptFS. When considering that the average transferred speed without any of them is 209.80 KByte/S, only the slight decrease occurred. The low-quality file was encoded by using 94.53 KByte/S bandwidth, the average transferred speed of the low-quality file was 92.94 KByte/S with the SELinux or the eCryptFS, and the average transferred speed without any of them was 92.92 KByte/S. The slight increase is within an error range. From the user's viewpoint, the two files were watchable even when protected by both the SELinux and eCryptFS at the same time, and had no large difference in the unprotected case and in the protected case.

We measured the streaming performance with the other 50 clients because we thought that enough resources could lead to the similar performance in the Table 4. Although the read speed is decreased when decrypting the files encrypted by the eCryptFS, the streaming process required only the constant speed. Thus, we could think that the streaming performance could be seriously decreased if the network became a rare resource, and more overloads were given to the server.

Table 5 shows the streaming performance when a video server has the other 50 concurrently connected users. The third machine for adding a server load ran a streaming load tool in the Darwin streaming server 5.5 and Windows Vista on an Intel Core 2 T7200 2.0 Ghz CPU and 1 GB RAM. This third machine ran the streaming load tool with setting the concurrent user to 50. At this third machine, we made requests to the high-quality file and the low-quality file with the repeat option between about 13:40 and about 15:50 while also minimizing user or machine interventions. We also collected the performance during 6 days by the WinDump [20]. In Table 5, HQ also means a high-quality file, and LQ also means a low-quality file.

Table 5. Streaming performance of a high-quality file and a low-quality files to 51 clients

KBytes/S		No eCryptFS	eCryptFS				
			Cast5	Aes128	Twofish	Blowfish	3des
HQ	No SELinux	210.10	210.05	209.80	210.01	209.94	209.91
	SELinux	210.00	210.15	210.14	209.81	210.25	209.99
LQ	No SELinux	93.29	92.82	93.24	93.23	93.25	93.2
	SELinux	93.19	93.36	93.21	93.23	93.22	93.26

Table 5 presents the results also after dividing the amount of the received data by the total time when streaming high-quality and low-quality files to 51 clients. Overall, Table 5 also presents that the streaming performance isn't significantly decreased. The high-quality file was encoded as 218.88 KByte/S, the average speed of the high-quality file is 210.00 KByte/S with the SELinux or the eCryptFS, the average transferred speed without any of them is 210.10 KByte/S. The low-quality file was encoded as 94.53 KByte/S, the average transferred speed of the low-quality file is 93.20 KByte/S with the SELinux or the eCryptFS, and the average speed without any of them is 93.29 KByte/S. From the user's viewpoint, the files were also watchable and had no significant difference in the unprotected case and the protected case with the SELinux or the eCryptFS.

We wanted to measure these performances with more 60 clients, but the QuickTime Player sometimes reported an error even without the SELinux or the eCryptFS, not enough bandwidth, thus the performance couldn't be measured with more 60 clients.

The reasons why the streaming performance wasn't decreased could be thought as follows. First, the read performance with an optimizer wasn't significantly decreased by either the SELinux or the eCryptFS as shown in Table 1. Although the read performance without an optimizer was significantly decreased by the SELinux or the eCryptFS as shown in Table 3, the file accesses by the streaming processes were continuous and the optimizer could be easily utilized. Second, the file buffering by either a file system or a streaming server could minimize the overheads. The file system and the streaming server usually used the buffering technologies that could minimize the read overheads. Third, although the decryption was used, and the read speed was decreased, the read speed could be still faster than the required speed for a client display even withe ad overheadsThe client usually required only the constant speed for a streaming process, and didn't require more speeds. Thus, the decryption didn't prevent the streaming processes only if the read and decryption speeds were faster than the streaming speeds to clients. When a system overhead wasn't significant, the overall performance at a user level wasn't decreased also in [21] due to these reasons mentioned above.

IV. Conclusion

In this paper, we proposed that both the eCryptFS and the SELinux should be used at the same time because they could complement each other for a content protection in a multimedia system. Among many available access controls and many available cryptographic file systems, we proposed the SELinux and the eCryptFS for a dedicated multimedia system because they were integrated into a Linux kernel, and supported various algorithms. In this paper, we also presented that using both of the SELinux and the eCryptFS didn't largely prevent the streaming processes to 51 clients although the other processes besides the streaming process were prevented, and their performances became worse.

Reference

[1] EMI group, EMI Music launches DRM-free superior sound quality downloads across its entire digital repertoire, *http://www.emigroup.com/Press/2007/press18.htm*, April 2007.

[2] Erin, J., Amazon To Sell DRM-Free Music, *http://www.internetnews.com/ec-news/article.php/3678181*, May 2007.

[3] Rosenblatt, B. & Dykstra, G., "Integrating content management with digital rights management: Imperatives and opportunities for digital content lifecycles", Technical report,

Giantsteps Media Technology and Dykstra Research, May 14 2003.

[4] 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.

[5] Halcrow, M., "eCryptfs: a stacked cryptographic filesystem", Linux Journal, Vol. 2007, Issue 156, April 2007.

[6] Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., & Lepreau, J., "The Flask Security Architecture: System support for diverse security policies", Proc. of the 8th USENIX Security Symposium, pp. 123-139, Washington, DC, August 1999.

[7] Arnab, A., & Hutchison, A., "Digital rights management-a current review", Departmental Technical Report No. cs04-04-00, University of Cape Town, 2004.

[8] Felten, E. W., "A skeptical view of DRM and fair use", Communications of the ACM, Vol. 46, No. 4, pp. 56-59, 2003.

[9] Schneier, B., "Applied Cryptography", 2nd Ed., John Wiley & Sons, 1995.

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

[11] Adams, C., "The CAST-128 encryption algorithm", RFC 2144, Network Working Group, May 1997.

[12] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., & Ferguson, N., "The Twofish Encryption Algorithm: A 128-Bit Block Cipher", John Wiley & Sons, 1999.

[13] Wright, C. P., Dave, J., & Zadok, E., "Cryptographic file system performance: What you don't know can hurt you", Proc. of 2003 IEEE Security In Storage Workshop, pp. 47-61, October 2003.

[14] 고영웅, "보안 운영체제를 위한 강제적 접근 제어 보호 프로파일", 한국컴퓨터정보학회 논문지, 제 10권, 제 1호, 141-148쪽, 2005년 3월.

[15] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R., & Chandramouli, R., "Proposed NIST standard for role-based access control", ACM Transactions on Information System Security, Vol. 4, No. 3, pp. 224 - 274, 2001.

[16] Niemi, D. C., Unixbench 4.1.0, *http://www.tux.org/pub/tux/niemi/unixbench*

[17] McVoy, L., & Staelin, C., lmbench 2, *http://sourceforge.net/projects/lmbench*

[18] textuality-Bonnie, *http://www.coker.com.au/bonnie++/*

[19] Darwin Streaming Server, *http://dss.macosforge.org/post/previous-releases/*

[20] WinDump: tcpdump for Windows, *http://www.mirrorservice.org/sites/ftp.wiretapped.net/pub/security/packet-capture/winpcap/windump/*

[21] 고영웅, "보안 운영체제의 오버헤드 분석", 한국 컴퓨터 정보학회 논문지, 제 10권, 제 2호, 11-19쪽, 2005년 5월.

저 자 소 개

김 성 기

2000: 건국대학교 공학사.
2009: 서울대학교 공학박사.
2009 - 현재: 삼성전자 DS 부문 책임연구원
관심분야: 병렬처리, 콘텐츠 보호, 내장형 시스템, 그래픽스.