DRMFS: A file system layer for transparent access semantics of DRM-protected contents ☆

Sangho Lee[a,*], Hay-Rim Lee[a], Seungkwang Lee[a], Jong Kim[b]

[a] *Department of Computer Science and Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea*

[b] *Division of IT Convergence Engineering, Pohang University of Science and Technology (POSTECH), Pohang, Republic of Korea*

Abstract

In many digital rights management (DRM) schemes, only a specialized application can decode DRM-protected contents. This restriction is harmful to users because they want to use their purchased digital contents with their preferred applications. To relax this restriction, DRM technology should provide transparent access semantics of DRM-protected contents to authorized applications. Some previous schemes achieve limited transparent access semantics but have efficiency and applicability problems. In this paper, we propose a DRM control scheme at the file system layer (DRMFS) that achieves transparent access semantics of DRM-protected contents with efficiency, applicability, and portability. Since DRMFS is working at the file system layer, any authorized application can access DRM-protected content in the same way as using general files. To implement a prototype of DRMFS, we use the Filesystem in Userspace (FUSE) library that is a well known library used to develop user level file systems. We explain details of the implementation and evaluate its performance. The evaluation results show that DRMFS has acceptable overheads.

Keywords: Digital rights management, Transparent access semantics, FUSE, Cryptographic file system

1. Introduction

Digital rights management (DRM) technology was introduced to protect the copyright of digital content from illegal users in digital environments. Nowadays, the importance of DRM is increasing due to the growth of the digital content market. Digital content users, however, dislike the current DRM technologies because technology developers give many *constraints* to users. Therefore, critics pointed out that DRM technologies actually prevent the growth of the digital content market and therefore they need to be expelled (DRM-Free activities (Engadget, 2007; Apple, 2009)). Among many constraints, DRM interoperability problems have been focused so far (Koenen et al., 2004; Taban et al., 2006; Doërr and Kalker, 2008). They consider how to unwind the constraints due to the differences between DRM content formats (DCFs) of different DRM technologies. However, even when the format interoperability is guaranteed by using their schemes, different constraints such as designated applications and content sharing still remains. Among them, we consider the constraint on the applications that can use DRM-protected content.

Users usually have a preferred application that they would like to use depending on the digital content type. However, current DRM technologies allow only their own applications or third party applications that attach their DRM modules to use their DCFs. Therefore, users cannot use their DCFs with their preferred applications when the applications are not modified to support the DRM technologies. Since there are many different DRM technologies, application developers should modify their applications to support all of these different DRM technologies. Moreover, DRM technology developers should provide DRM modules for different applications. Thus, the constraint on applications is difficult to be relaxed by using application level solutions. Therefore, we need a low level DRM scheme that provides *transparent access semantics* (Blaze, 1993) of DRM-protected contents to authorized applications.

1.1. DRM control levels and transparency

We classify DRM schemes according to the level where DRM control is performed: *application*, *middleware*, *kernel*, and *file system* (see Figure 1).

In the application level DRM, each application performs DRM operations by itself. For instance, application developers may use Windows Media SDK (Microsoft,

☆This research was supported by the MKE (The Ministry of Knowledge Economy), Korea, under the ITRC (Information Technology Research Center) support program supervised by the NIPA (National IT Industry Promotion Agency) (NIPA-2011-C1090-1131-0009) and WCU (World Class University) program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (R31-2008-000-10100-0).

*Corresponding author. Tel.: +82 54 279 2915. Fax: +82 54 279 1805

Email addresses: `sangho2@postech.ac.kr` (Sangho Lee), `hayrlee@postech.ac.kr` (Hay-Rim Lee), `skwang@postech.ac.kr` (Seungkwang Lee), `jkim@postech.ac.kr` (Jong Kim)

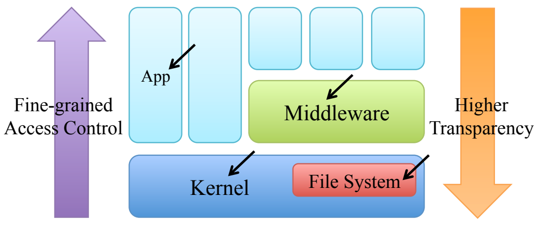


Figure 1: DRM control levels

2009) to enable their applications to support Windows Media DRM. Since each application has its own DRM and it knows all of its operations, the application level DRM allows more fine-grained access controls. However, it will increase development cost and introduce compatibility problems due to the variety of DRM schemes. Moreover, since each application needs to aware DRM schemes, its transparency level is low and thus it needs to be changed when the DRM schemes are modified.

In the middleware level DRM, all DRM operations are performed at the middleware and thus each application does not need to know whether DRM control is performed. Qtopia's Document System that natively supports DRM file type (Qtopia, 2007), Nair et al.'s scheme that extends JVM to apply the usage control model to enforce DRM policies across applications (Nair et al., 2008), and Porscha that modifies the Android middleware to support DRM (Ongtang et al., 2010) are instances of the middleware level DRM. However, it is only available on middleware-based systems. Also, each application needs to be modified to interact with the middleware; thus, its transparency level is relatively low.

In the kernel level DRM, all DRM operations are performed at the kernel and thus each application does not need to aware DRM controls as the middleware level DRM does. Moreover, application modifications are not needed. Therefore, it transparency level is high. Arnab *et al.*'s kernel level DRM scheme (Arnab et al., 2007) is composed of four stages: 1) intercept access to a DCF from applications, 2) check the corresponding licenses, 3) decrypt the DCF to temporal secure storage, and 4) redirect applications to access the temporal storage. The first and third stages of their scheme incur a significant overhead. For the first stage, it intercepts *all* system calls for file managements. Thus, it increases the overall system overheads. Moreover, in their scheme, applications attempt to access DCFs directly. Since DCFs usually have different names rather than their original names, for instance `movie.dcf` rather than `movie.mp4`, and are larger than their original size due to additional header information and padding, some applications that are sensitive to file name and size may not work with their scheme due to this *semantics mismatch problem*. This problem decreases the level of transparency.

For the third stage, their scheme decrypts a DCF *entirely* and store it in a *secure storage* temporally. Since the entire file is decrypted, the performance of their scheme depends on the size of the DCFs. For instance, if a DCF is a high-quality movie that is very large, then decrypting the DCF before using it will involve a long delay. This long delay is not desired especially when an application only needs some portions of the original file such as *metadata*. Moreover, they assume that the storage is secure to prevent access to the temporarily decrypted contents. Secure storage, however, is difficult to realize with software-based solutions. Hardware-based secure storage (Shapiro and Vingralek, 2001) can be used to enhance security but it is expensive. Therefore, this scheme is only applicable for small size contents such as music or document files.

Since most DRM operations are performed at file systems, we can apply DRM schemes for the file system level to reduce the overhead of the kernel level DRM. Cross and Leach have proposed a file system level DRM scheme in their patent (Cross and Leach, 2008). Unlike the kernel level DRM scheme, their scheme only monitors access to DRM-protected contents. Thus, its overhead is smaller than the kernel level DRM scheme. Their scheme, however, shares same problems with the kernel level DRM scheme; it does not consider the semantics mismatch problem and decrypts a DCF entirely before accessing it. Moreover, their scheme does not check whether the application that attempts to access DRM-protected contents is authorized.

1.2. Research goal

In this paper, we propose a DRM control scheme that satisfies higher level of transparent access semantics and lower system overhead. To satisfy these requirements, we implement our DRM on file systems that are referred to as DRMFS. DRMFS map DCFs to in-memory imaginary files that have the same name and size as contents without DRM protection to solve the semantics mismatch problem of the previous kernel and file system level DRM schemes. By accessing these imaginary files, authorized applications can transparently access DCFs with their real names and size. Unlike the kernel level DRM scheme, DRMFS only intercepts access to DCFs and therefore the overall performance degradation is small. Also, to reduce response time, DRMFS supports block level decryption by intercepting `read()` system call to DCFs unlike the previous kernel and file system level DRM schemes. We were able to implement DRMFS using the Filesystem in Userspace (FUSE) library (Szeredi, 2010) and the OpenSSL library (OpenSSL Project, 2010; Viega et al., 2002) on Linux. Thus, any Linux system and Mac system (with MacFUSE (Singh, 2008)) that support these two libraries can use DRMFS.

1.3. Our contributions

The contributions of this paper can be summarized as follows:

- We achieve true transparent access semantics of DCFs for applications. Because applications can see the

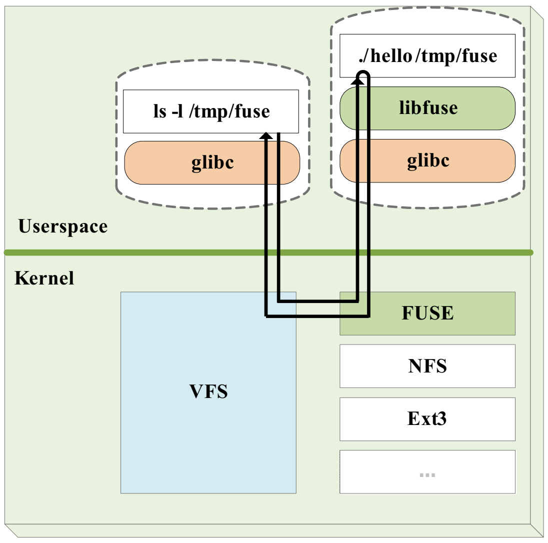


Figure 2: Flow chart diagram showing how FUSE works

real names and size of DCFs from the corresponding imaginary files, there is no difference except that additional time for decryption is needed.

- We do not induce much of a system overhead because our scheme only intercepts access to DCFs. Other activities that do not relate to DCFs are not affected.
- We reduce the response time required to access DCFs as much as possible because we use block level decryption. The response time is important especially when an application only requires some portions of the original file such as metadata. This also removes the need for secure storage to temporarily store the decrypted contents.
- We suggest a real implementation of DRMFS by using two well known libraries, FUSE and OpenSSL. These two libraries are ported to various operating systems. Therefore, its portability is good.

1.4. Paper organization

This paper is organized as follows. In Section 2 we briefly introduce the FUSE library that is the main component of DRMFS. In Section 3 we present an overview of our system. In Section 4 we explain a prototype implementation of DRMFS. In Section 5 we evaluate the performance of DRMFS. In Section 6 we introduce related work to this paper. Finally, we conclude this paper and discuss future work in Section 7.

2. Filesystem in userspace (FUSE)

The FUSE library allows users to develop high level file systems in userspace. FUSE provides an interface for userspace applications to export virtual file systems (VFSes) to the Linux kernel and also provides a secure method for non-privileged users to create and mount their own file system implementations. We explain how FUSE works using Figure 2. Initially, an application, called `hello`, mounts a FUSE file system to `/tmp/fuse`. Later, when `ls` attempts to read `/tmp/fuse`, this request is delivered to `hello` through the VFS and FUSE kernel module. Then, `hello` manipulates the data and delivers it to `ls` through FUSE and VFS.

The main advantage of FUSE is providing easy integration with other user file systems. From the user's point of view, there is no difference in the access mode between offline, local files, or files located on the distributed system. Also, it provides a common API for file systems across different platforms. Thus, the file system can be used by a wide range of users and devices. Since FUSE works in userspace, it is easy to compile and it is independent of the constant changes within the Linux kernel.

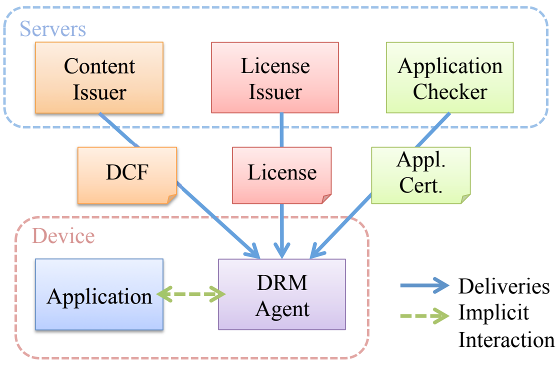


Figure 3: System model

3. System description

3.1. System model

As is common with DRM systems (Michiels et al., 2005; Open Mobile Alliance, 2008a; Rosenblatt et al., 2002), our DRM system has three basic entities: *content issuer*, *license issuer*, and *device with a DRM agent*. In addition, our DRM system has another entity, an *application checker* (see Figure 3).

A content issuer is an entity that translates (encrypts) digital content to DCF and distributes it to users. Our system allows superdistribution (Mori and Kawahara, 1990) and therefore, secret keys that are used to generate a DCF need to be securely delivered to a license issuer.

A license issuer (also known as a rights issuer) is an entity that generates and delivers licenses (also known as rights objects) to users. Each license describes who can use some DCFs along with some pre-defined constraints. Depending on agreements, a license can be issued to a device for stronger enforcement or to a user for more convenient usage. When a license is issued to a device, secret parts of a license such as the secret keys of DCFs and the user's private information will be encrypted by using the device's public key (Open Mobile Alliance, 2008a). When a license is issued to a user, the secret parts will be encrypted by using the user's public key (Koster et al., 2005). Licenses issued by a license issuer should be digitally signed by the license issuer to guarantee that they are valid. Many license definitions (ISO/IEC FDIS 21000-5:2003(E), 2003; Open Mobile Alliance, 2008c; ODRL Initiative, 2011) make use of eXtensible Markup Language (XML) due to its extensibility and flexibility.

An application checker is an entity that checks whether applications that will be executed by user devices are legitimated. Applications are authorized if they do not violate basic DRM rules such as no backdoor to extract raw (unencrypted) digital content. When an application checker confirms that an application is legitimated, it issues an application certificate for the application. In our system, applications without certificates cannot access DCFs to prevent the leakage of raw content. This application verification phase is similar to the app store models such as Apple App Store and Google Android Market. Porscha (Ongtang et al., 2010) also needs to check each application to verify whether it fulfills policy constraints.

A device is an entity that belongs to a user, for using digital contents. This device has a DRM agent to legally use DCFs. Before using a DCF, the DRM agent first checks the corresponding license. If this license is generated by a valid license issuer and has not expired, the DRM agent extracts secret keys from the license by using the device or user private key. Then, it decrypts the DCF with the extracted secret keys and then uses it. In our DRM system, the DRM agent resides in the file systems. Therefore, it can control any application that attempts to access DCFs.

Since there is a possibility of device tampering to compromise DRM systems, tamper-proof technologies such as Trusted Platform Module (TPM) (Trusted Computing Group, 2011) need to be assumed to assure the security of the DRM system. With the TPM's remote attestation technique, a remote server can verify whether the integrity of a device is crashed. When the integrity of OS or DRM agent of the device is broken, the remote server will revoke the device and will give any service to the device. In addition, the integrity of applications can be verified by using application certificates. With TPM's secure storage, a device can protect secret information such as device or user private key. Therefore, attackers will have difficulties in bypassing license checks or extracting secret information.

3.2. Data formats

We introduce the data formats of DCF, license, and application certificates. Our DCF format is binary and its header is almost the same as the DCF of OMA DRM version 2.1 (Open Mobile Alliance, 2008b). The header contains several fields including EncryptionMethod, ContentID, and RightsIssuerURL. In this research we only consider the PlaintextLength field that represents the real size of the content and the ContentType field that represents the type of content. These two fields will be used to create imaginary files. To protect DCFs while allowing random searching, we use AES in ECB mode.

Table 1: License format

Tag	Meaning
<rights>	Root element containing content information, encryption key, constraints, and signature
<content>	Containing the content information such as type and hash value
<type>	Specifying the type of the content
<hashVal>	Containing the hash value of the content
<EncryptedKey>	Containing the encryption key of the content. This key is encrypted by using the device or user public key.
<constraints>	Specifying several types of constraints such as number, time, and period of content usage
<signature>	Containing the signature of this license issued by a license issuer

Table 2: Application certificate format

Tag	Meaning
<certificate>	Root element containing application information and signature
<application>	Containing the application information such as type and hash value
<type>	Specifying the types of contents this application can use
<hashVal>	Containing the hash value of the application
<signature>	Containing the signature of this certificate issued by an application checker

Our license format is an XML format that follows the OMA DRM version 2.1 (Open Mobile Alliance, 2008c) and (Bhatt et al., 2009) (see Table 1) and the application certificate's format is similar to the license format (see Table 2). The <type> field of an application certificate is required to prevent accesses by applications to the contents that cannot be rendered by the applications. For instance, a media player does not need to open document files.

3.3. Device internals and interactions

Our device has three components: *DRMFS agent*, *FUSE kernel module*, and *application*, and has four objects: *DCF*, *license*, *application certificate*, and *imaginary file* (see Figure 4).

A DRMFS agent is a user level application that manages the overall DRMFS activities. It interacts with a

FUSE kernel module to create imaginary files and to support block level decryption of DCFs. It also checks the validity of applications, DCFs, and licenses. Currently, the DRMFS agent only supports file read operations because we aim to develop a DRM system for consumer electronics.

A FUSE kernel module is a kernel level application that can create file systems by interacting with user level applications. Its main function is the redirection of system calls that are targeted from its file systems to corresponding user level applications such as the DRMFS agent.

An application is a program that attempts to use DCFs. To use DCFs, it should have an application certificate. However, it does not need to know about the existence of its certificate.

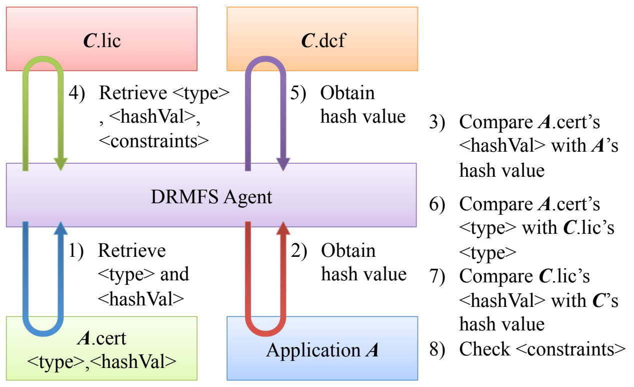
The objects have designated locations. The DCFs and their corresponding licenses are located at `/drm/dcfs`, and the application certificates are stored in locations where corresponding applications exist. Imaginary files will be displayed at `/drm/drmfs` that is where the DRMFS is mounted.

We illustrate the process that how an application reads DCFs on a device with DRMFS which is illustrated in Figure 4. When an application A requests a block of content C on `/drm/drmfs` that is imaginary, this request is redirected to the DRMFS agent by the FUSE module. The DRMFS agent first checks the validity of A's certificate, and then retrieves the <type> and <hashVal> from A's certificate. Then, it compares the retrieved hash value with A's hash value to check integrity. If A's integrity is verified, then the DRMFS agent checks the validity of C's license and retrieves <type>, <hashVal>, and <constraints> from C's license. Then, it compares <type> from A's certificate and <type> from C's license to check whether A is permitted to use C. If A is permitted, then it checks C's integrity and checks <constraints>. If <constraints> is still fulfilled (for instance, when the licensed number of content usage remains), it allows A to access C (details are in Figure 5). After checking the application certificate and license, the DRMFS agent reads blocks of DCF corresponding to C from `/drm/dcfs`. Then, it decrypts these blocks and rearranges them as blocks of C. Finally, it provides these decrypted blocks to A. To enhance the performance of the above procedures, the checking procedure can be performed only once when an application attempts to open the content.

4. Prototype implementation

4.1. Implementation goals

As mentioned in Section 3, DRMFS creates imaginary files on `/drm/drmfs` by using corresponding DCFs in `/drm/dcfs`. To achieve this, DRMFS has to intercept the applications' system calls to `/drm/drmfs` to fake those applications. The system calls that should be intercepted by DRMFS are listed in Table 3. For these system calls, DRMFS should carry out the following actions.



C.lic
C.dcf
4) Retrieve <type>, <hashVal>, <constraints>
5) Obtain hash value
3) Compare A.cert's <hashVal> with A's hash value
DRMFS Agent
6) Compare A.cert's <type> with C.lic's <type>
1) Retrieve <type> and <hashVal>
2) Obtain hash value
7) Compare C.lic's <hashVal> with C's hash value
A.cert <type>,<hashVal>
Application A
8) Check <constraints>

Figure 5: Details of application certificate and license checks

Table 3: System calls need to be intercepted

System calls	Operations
`readdir()`	Listing files and subdirectories under a specified directory
`getattr()`	Obtaining attributes such as permission, creation date, and size from specified file or directory
`open()`	Opening a specified file
`read()`	Reading some blocks of an opened file
`release()`	Closing an opened file

- When an application uses `readdir()` for a subdirectory of `/drm/drmfs`, DRMFS intercepts this call and lists the real names of DCFs in the corresponding subdirectory of `/drm/dcfs`. In this process, names of subdirectories of `/drm/dcfs` are listed without modifications and DCFs without valid licenses are not listed (see Figure 6a).

- When an application uses `getattr()` for a file in `/drm/drmfs`, DRMFS intercepts this call and returns the real size of the corresponding DCF. DRMFS sets the permissions for that file as read-only.

- When an application uses `open()` for a file in `/drm/drmfs`, DRMFS intercepts this call, and checks the validity of the certificate of the application and the license of the corresponding DCF. If this validity check succeeds, then the application can open the file. If not, the `open()` system call is denied (see Figure 6b).

- When an application uses `read()` for an opened file in `/drm/drmfs`, DRMFS intercepts this call, and decrypts the blocks of the corresponding DCF in `/drm/dcfs` and delivers decrypted blocks to the application (see Figure 6c).

- When an application uses `close()` for an opened file in `/drm/drmfs`, DRMFS intercepts this call and removes DRM-related information from the memory, and then closes the file.

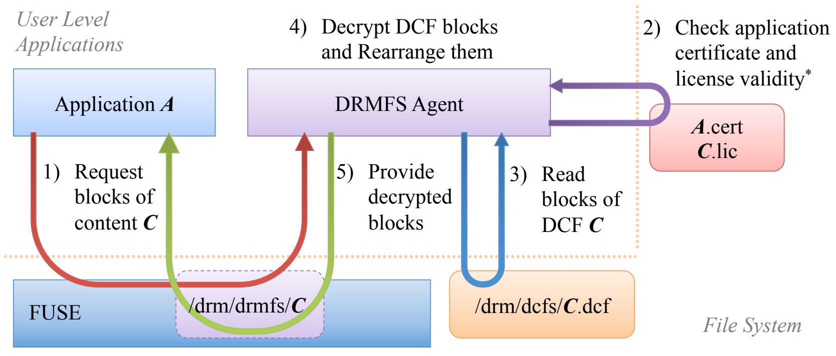

User Level Applications
4) Decrypt DCF blocks and Rearrange them
2) Check application certificate and license validity*
Application A
DRMFS Agent
A.cert
C.lic
1) Request blocks of content C
5) Provide decrypted blocks
3) Read blocks of DCF C
FUSE
/drm/drmfs/C
/drm/dcfs/C.dcf
File System

Figure 4: Device internals and interactions

4.2. Implementation details

In this subsection, we will explain about the implementation of DRMFS. We will only explain the functions relating to file system operations. Other functions such as the generator, parser, and checker of DCF, license, and application certificate will not be explained because they are of lesser importance in this study.

To replace `readdir()`, `getattr()`, `open()`, `read()`, `release()` system calls to `/drm/drmfs`, we generate functions `drmfs_readdir()`, `drmfs_getattr()`, `drmfs_open()`, `drmfs_read()`, and `drmfs_release()` according to FUSE's description. These functions can be registered by defining a C structure variable and giving it as an argument when DRMFS is initiated as below.

```
static struct fuse_operations drmfs_oper = {
     .readdir = drmfs_readdir,
     .getattr = drmfs_getattr,
     .open    = drmfs_open,
     .read    = drmfs_read,
     .release = drmfs_release,
};
```

At first, we explain `drmfs_readdir()` function. `drmfs_readdir()` receives two arguments: path name of a subdirectory in `/drm/drmfs` and buffer to store file names. From the received path name, `drmfs_readdir()` finds a corresponding subdirectory in `/drm/dcfs` and iteratively reads directories and files in the directory. When `drmfs_readdir()` reads a directory, it simply writes the directory's name to the buffer. When `drmfs_readdir()` reads a DCF with a valid license, it parses the DCF's real name and writes the name to the buffer. When `drmfs_readdir()` reads a file that is neither a directory nor a DCF with a valid license, it ignores the file to hide the file from the caller. The retrieved file names will be returned to a caller as `readdir()` does.

Next, we explain `drmfs_getattr()` function. `drmfs_getattr()` receives two arguments: path name of a file in `/drm/drmfs` and buffer to store the status information of the file. From the received path name, `drmfs_getattr()` finds a corresponding file in `/drm/dcfs` and retrieves the status information of the corresponding file. When the corresponding file is a directory, `drmfs_getattr()` simply copies its status information to the buffer. When the corresponding file is a DCF, `drmfs_getattr()` parses the DCF's real size, and then writes the read-only mode and the real size to the buffer. The retrieved status information will be returned to a caller as `getattr()` does.

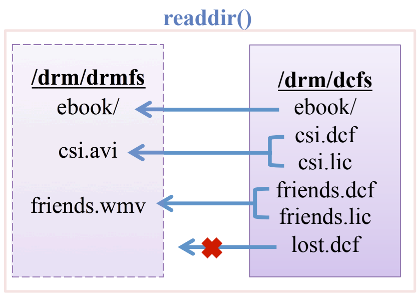
`drmfs_open()` function receives one argument: path name of a file in `/drm/drmfs`. At first, `drmfs_open()` checks the application certificate of the application that attempts to open the file. When the application certificate is valid, `drmfs_open()` finds a corresponding DCF in `/drm/drmfs`, checks whether the DCF has a valid license, and compares their types. If all checks are successful, then `drmfs_open()` loads the DCF information such as size and its secret key to memory. This memory loading is needed to enhance the performance of DRMFS. At last, a corresponding file descriptor will be returned to a caller as `open()` does.

`drmfs_read()` function receives four arguments: path name of an opened file in `/drm/drmfs`, the buffer to read data, and the size and starting position of the data that an application wants to read. At first, `drmfs_read()` reads the DCF information from the memory. Then, it reads the corresponding blocks from the DCF and then decrypts them. Because the DCF was encrypted with a block cipher, offset shifting is needed when the start position is not a multiple of an AES block size (128 bits). After the decryption, shift, and rearrangement, `drmfs_read()` returns the read buffer with the size that it actually reads as `open()` does.

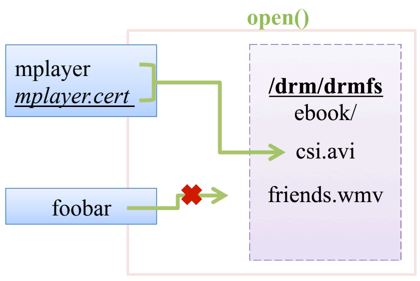
`drmfs_release()` function receives one argument: path name of an opened file in `/drm/drmfs`. It simply removes the DCF information of the file from memory and closes the corresponding DCF in `/drm/dcfs`.

5. Evaluation

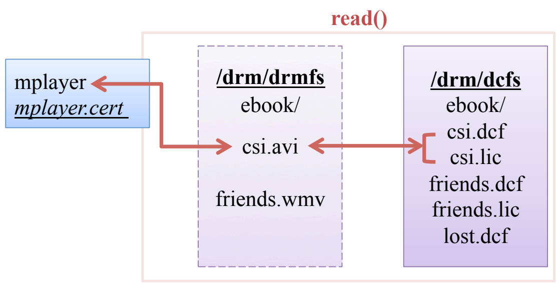
We evaluate the performance of DRMFS and compare it with previous work. Our test machine has an Intel Pentium Processor E2180 2.0 GHz CPU (dual-core), 3 GB main memory, and a 7200 RPM 320 GB hard disk. The test machine's operating system is Ubuntu 10.10 with kernel version 2.6.35 and its file system is ext4 with 4 KB



(a) Display DCFs, which have corresponding licenses, as imaginary files



(b) Allow applications which have valid certificates to open imaginary files



(c) Return plaintext blocks to applications by decrypting corresponding ciphertext blocks

Figure 6: Interactions between applications and DRMFS

block size. We use FUSE library version 2.7.4 and OpenSSL library version 0.9.8o to implement DRMFS. We also collect seven audio and video files for evaluation (see Table 4).

At first, we check the performance of the test environment and its performance with DRMFS. We compare the throughput of file read, write, decryption, and read with decryption on ext4, and read on DRMFS in sequential events. To measure the read throughput, we sequentially read all of 4 KB blocks of each of the collected file. To measure the write throughput, we copy each file to a different location and then subtract the read time. To measure the decryption throughput, we use the OpenSSL command line tool to decrypt each file, and then subtract the read and write time. Since the application level DRM schemes perform read and decryption at the same time, we also measure the throughput of read with decryption to compare it with the read throughput of DRMFS. To ignore temporal variations, all tests are performed ten times and we take their average values. In all tests, 95% confidence

Table 4: Description of the collected files. F1, F2, . . ., and F7 are acronyms of each of the collected seven files.

Name	Format	Size (MB)
F1	MP3 audio	10
F2	MKV movie	74
F3	AVI movie	183
F4	MPEG-4 movie	303
F5	AVI movie	418
F6	AVI movie	763
F7	MPEG-4 movie	1163

intervals are at most ±3.3%; therefore, variations are not much high. Between each test, we flushed file caches by using `/proc/sys/vm/drop_caches` (OpenOffice.org, 2009). We perform the tests with single-core[1] and dual-core. The results are in Figure 7.

From the evaluation results, we confirm that the performance of DRMFS is almost same as the application-level DRM schemes (read with decryption) and sometimes even better than them in sequential events. With single-core, the throughput of read with decryption is about 32.3% of the read throughput on ext4, and the read throughput of DRMFS is about 28.8% of the read throughput of ext4 (see Figure 7). Therefore, the read throughput of DRMFS is about 1.12 times lower than the application level DRM schemes that read and decrypt files by themselves. This overhead is due to the context switching between the test process and DRMFS process, and the I/O interception and manipulation; this overhead occupies about 16% of DRMFS operating time (see Figure 8). With dual-core, the throughput of read with decryption is about 37.2% of the read throughput on ext4, and the read throughput of DRMFS is about 49.3% of the read throughput on ext4 (see Figure 7). Therefore, the read throughput of DRMFS is about 1.32 times higher than the application level DRM schemes. This improved performance is due to the simultaneous execution of the DRMFS process in a different core; no context switching occurs, and read and decryption can be interleaved. Even high quality Blue-ray needs 5 MB/s. Therefore, the throughput of DRMFS is acceptable.

Next, we compare the throughput of read and read with decryption on ext4, and read on DRMFS in random events. We randomly read 4, 16, and 32 KB blocks of the file F7 until we read 4, 40, 100, 160, 200, and 300 MB portions of the file. We also flush file caches between each test. When we read 4 KB blocks and the read size exceeds 100 MB or when we read 16 KB blocks and the read size exceeds 200 MB, the read throughput of DRMFS is higher than the read throughput on ext4 (see Figure 9). This higher throughput is owing to the file caches that the FUSE library maintains. In contrast, when we read 32 KB blocks, the read throughput of DRMFS is lower than the read throughput on ext4 until the read size is less than

[1]We deactivated one core by setting a kernel option `maxcpus=1`.

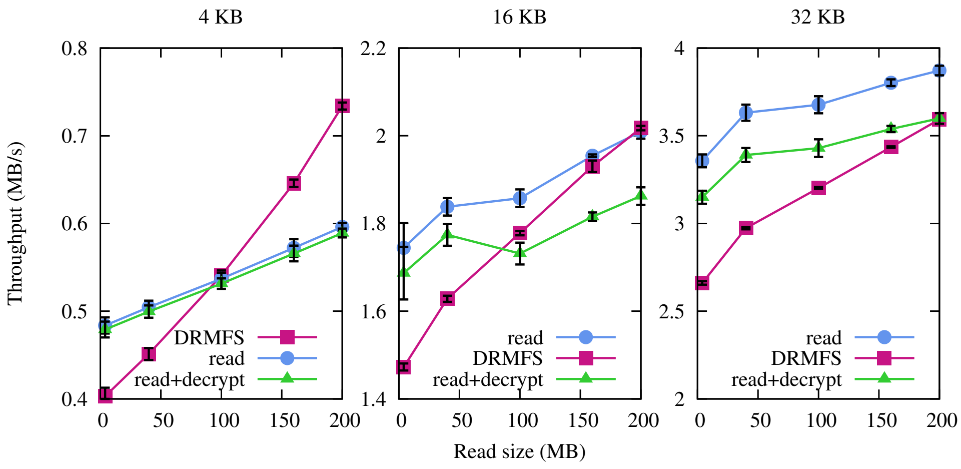


Figure 9: Comparison on the throughput of random events

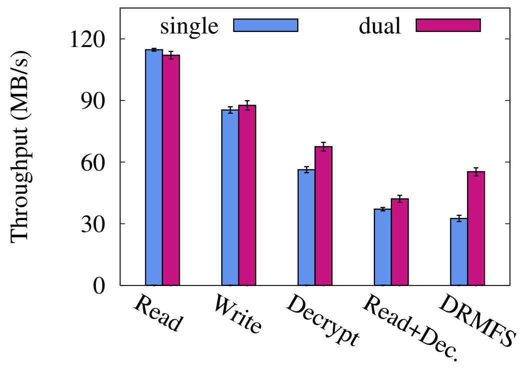


Figure 7: Comparison on the throughput of sequential events

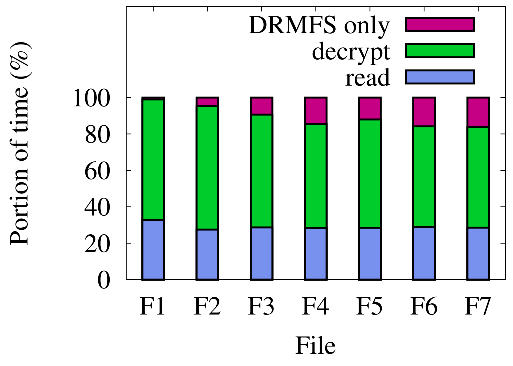


Figure 8: Portion of time of DRMFS operations

300 MB. This difference will be increased when we choose more larger size blocks. Additionally, in most cases, the read throughput of DRMFS is higher than the throughput of read with decryption on ext4. The throughput of random events increases according to the block size and the read size because the probability that reading the already read blocks, which were cached in memory, also increases according to the block size and the read size.

5.1. Comparisons with previous work

We compare the read throughput and response time of DRMFS with the previous kernel and file system level DRM schemes (Arnab et al., 2007; Cross and Leach, 2008). When we ignore the decryption and write time of the previous work for full decryption of DCFs, the read throughput of the previous work is the same as the read throughput on ext4. Therefore, the read throughput of the previous work is two to three times higher than DRMFS. The response time of the previous work, however, is always worse than DRMFS as will be shown below.

Let S is the size of a DCF, r_S is the required time to read the DCF, d_S is the required time to decrypt the DCF, and w_S is required time to write the decrypted DCF to (secure) storage. Let S' is the size of a portion of the DCF where an application needs to be initiated. The response time of the previous kernel and file system level DRM schemes is $r_S + d_S + w_S + r_{S'}$ because it needs to decrypt all of the DCF and then write the result to storage before using it. With dual-core, the response time of DRMFS is about $2r_{S'}$ (see Figure 7). Thus, when $r_S + d_S + w_S > r_{S'}$, the response time of DRMFS is better than that of the previous work. Since $r_{S'} \leq r_S$ and $d_S + w_S > 0$, the above inequality is always true. With single-core, the response time of DRMFS is about $3.5r_{S'}$. Thus, when $r_S + d_S + w_S > 2.5r_{S'}$, the response time of DRMFS is better than that of the previous work. Since $r_{S'} \leq r_S$, $d_S \approx 2r_S$, and $w_S \approx 1.3r_S$ (see Figure 7), the above inequality is always true. Therefore, the response time of DRMFS is always better than the previous work.

Response time is especially important when an application only needs some portions of DRM-protected content such as metadata. Metadata is a simple description of a file and normally located at the header of the file. Since its size is usually smaller than several kilobytes, no matter how large the file is, decrypting the entire file to retrieve the metadata is a heavy burden. We check the required

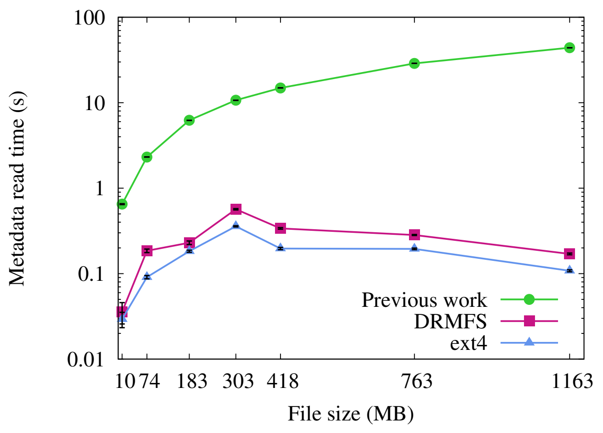


Figure 10: Comparison on metadata read time according to the size of the test files (metadata read time is in log scale)

time to retrieve the metadata of our test files by using MediaInfo (Martinez, 2010). The metadata retrieval time in ext4 and DRMFS is less than a second and does not depend on the size of the test files. In contrast, the metadata retrieval time in previous kernel and file system level DRM schemes is very long and linearly increases according to the file size due to the full decryption (see Figure 10). Therefore, DRMFS has advantages to previous work in terms of metadata retrieval time.

Moreover, the previous work requires additional storage to store the temporally decrypted DCF. The size of that additional storage is the same as for the corresponding DCF; thus, if the DCF is large, the additional storage can become a burden. Because DRMFS does not need that kind of additional storage, DRMFS is better than the previous work in terms of storage usage.

6. Related work

DRMFS can be treated as a specialized cryptographic file system. The cryptographic file system is a file system that allows the encryption and decryption of files at the file system level. The main difference between DRMFS and cryptographic file systems is that DRMFS only considers transparent decryption and it allows more delicate access controls by using licenses.

From earlier research on Cryptographic File Systems (CFS) (Blaze, 1993), many successors to the approach such as Transparent Cryptographic File System (TCFS) (Cattaneo et al., 2001), NCryptfs (Wright et al., 2003), and eCryptfs (Halcrow, 2005) have been proposed. CFS is based on the Network File System (NFS) (Sun Microsystems, Inc., 1989) that is used to mount remote storage to a local directory. CFS has a special NFS daemon cfsd to intercept requests from NFS clients. Then, cfsd encrypts data from NFS clients and store them to untrusted storage and vice versa. To use CFS, users should use special commands; attach and detach. Also, its performance is low because of frequent context switches, and data copies between the kernel and user space. TCFS is a kernel level cryptographic file system. It works transparently for applications and users and it does not need special commands from users. Nevertheless, it does have several security problems and is only available for earlier versions of Linux (before 2.2.17). NCryptfs is a cryptographic file system that is stackable (Zadok et al., 2006) for existing file systems. NCryptfs has security methods similar to CFS and therefore, its security is high compared to TCFS. Because NCryptfs works at the kernel level, its performance is good compared to CFS. Also, thanks to its stackability, NCryptfs is convenient and portable. eCryptfs is another stackable cryptographic file system that is similar to NCryptfs.

There has been a lot of research and projects done using FUSE. For instance, Cornell *et al.* proposed Wayback that is a user level versioning file system that allows users to access any previous version of their files (Cornell et al., 2004). Also, some distributed file systems such as Ceph (Weil et al., 2006) and TierStore (Demmer et al., 2008) use the FUSE system as an interface to connect applications with their distributed file systems. Many other projects such as SSHFS (Szeredi, 2008) and GmailFS (Hansen, 2010) are listed in the FUSE project homepage (Szeredi, 2010).

7. Conclusions

We have presented a file system layer DRM scheme, DRMFS, which achieves transparent access semantics of DRM-protected contents. Because DRMFS is located at the file system layer, any application can transparently access DRM-protected contents on DRMFS as general files. We have explained in depth how our system is designed and implemented. We have also evaluated our implementation and confirmed that it has an acceptable overhead.

In future we will extend DRMFS as a network file system to achieve application transparent streaming of DRM-protected contents. Projects like the SSHFS (Szeredi, 2008) will provide a good motivation for this extension. Also, we will port DRMFS to embedded systems such as the Android phone.

References

Apple, 2009. Changes Coming to the iTunes Store, `http://www.apple.com/pr/library/2009/01/06itunes.html`.

Arnab, A., Paulse, M., Bennett, D., Hutchison, A., 2007. Experiences in implementing a kernel-level DRM controller. In: Proceedings of 3rd International Conference on Automated Production of Cross Media Content for Multi-channel Distribution (AXMEDIS). pp. 39–46.

Bhatt, S., Sion, R., Carbunar, B., September 2009. A personal mobile DRM manager for smartphones. Computers & Security 28 (6), 327–340.

Blaze, M., 1993. A cryptographic file system for UNIX. In: Proceedings of 1st ACM Conference on Computer and Communications Security (CCS).

Cattaneo, G., Catuogno, L., Sorbo, A. D., Persiano, P., June 2001. The design and implementation of a transparent cryptographic filesystem for UNIX. In: Proceedings of USENIX Annual Technical Conference (ATC), FREENIX Track. pp. 245–252.

Cornell, B., Dinda, P. A., Bustamante, F. E., 2004. Wayback: A user-level versioning file system for Linux. In: Proceedings of USENIX Annual Technical Conference (ATC), FREENIX Track. pp. 19–28.

Cross, D. B., Leach, P. J., September 2008. File system operation and digital rights management (DRM). United States Patent 2008/0235807.

Demmer, M., Du, B., Brewer, E., 2008. TierStore: A distributed filesystem for challenged networks in developing regions. In: Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST).

Doërr, G., Kalker, T., 2008. Design rules for interoperable domains: Controlling content dilution and content sharing. In: Proceedings of 8th ACM Workshop on Digital Rights Management (DRM). pp. 39–50.

Engadget, 2007. Amazon launches DRM-free "Amazon MP3" music downloads, http://www.engadget.com/2007/09/25/amazon-launches-drm-free-amazon-mp3-music-downloads.

Halcrow, M. A., July 2005. eCryptfs: An enterprise-class cryptographic filesystem for Linux. In: Proceedings of Linux Symposium. pp. 201–218.

Hansen, D., 2010. GMail Filesystem over FUSE, http://sr71.net/projects/gmailfs/.

ISO/IEC FDIS 21000-5:2003(E), 2003. Information technology - multimedia framework - part 5: rights expression language.

Koenen, R. H., Lacy, J., Mackay, M., Mitchell, S., June 2004. The long march to interoperable digital rights management. Proceedings of IEEE 92 (6), 883–897.

Koster, P., Kamperman, F., Lenoir, P., Vrielink, K., 2005. Identity based DRM: Personal entertainment domain. In: Proceedings of Communications and Multimedia Security (CMS). pp. 42–54.

Martinez, J., 2010. MediaInfo, http://mediainfo.sourceforge.net.

Michiels, S., Joosen, W., Truyen, E., Verslype, K., November 2005. Digital rights management—a survey of existing technologies. Tech. rep., Department of Computer Science, Katholieke Universiteit Leuven.

Microsoft, 2009. Windows Media DRM Client Extentend APIs, http://msdn.microsoft.com/en-us/library/dd757736(v=VS.85).aspx.

Mori, R., Kawahara, M., July 1990. Superdistribution: The concept and the architecture. IEICE Transations E73-E (7), 1133–1146.

Nair, S. K., Tanenbaum, A. S., Gheorghe, G., Crispo, B., 2008. Enforcing DRM policies across applications. In: Proceedings of 8th ACM Workshop on Digital Rights Management (DRM). pp. 87–94.

ODRL Initiative, 2011. http://www.odrl.net.

Ongtang, M., Butler, K., McDaniel, P., 2010. Porscha: Policy oriented secure content handling in Android. In: Proceedings of Annual Computer Security Applications Conference (ACSAC).

Open Mobile Alliance, October 2008a. DRM Architecture - Approved Version 2.1. http://www.openmobilealliance.org.

Open Mobile Alliance, October 2008b. DRM Content Format - Approved Version 2.1. http://www.openmobilealliance.org.

Open Mobile Alliance, October 2008c. DRM Rights Expression Language - Approved Version 2.1. http://www.openmobilealliance.org.

OpenOffice.org, 2009. Cold-start-simulator – OpenOffie.org Wiki, http://wiki.services.openoffice.org/wiki/Cold-start-simulator.

OpenSSL Project, 2010. OpenSSL Project, http://www.openssl.org.

Qtopia, 2007. Document System: DRM Integration, http://doc.qt.nokia.com/qtopia4.2/docsys-drmintegr.html.

Rosenblatt, B., Trippe, B., Mooney, S., 2002. Digital Rights Management—Business and Technology. New York: M&T Books.

Shapiro, W., Vingralek, R., 2001. How to manage persistent state in DRM systems. In: Proceedings of ACM Workshop on Security and Privacy in Digital Rights Management (DRM).

Singh, A., 2008. MacFUSE, http://code.google.com/p/macfuse/.

Sun Microsystems, Inc., August 1989. NFS: Network file system protocol specification. RFC 1094.

Szeredi, M., 2008. SSH Filesystem, http://fuse.sourceforge.net/sshfs.html.

Szeredi, M., 2010. Filesystem in Userspace, http://fuse.sourceforge.net.

Taban, G., Cárdenas, A. A., Gligor, V. D., 2006. Towards a secure and interoperable DRM architecture. In: Proceedings of 6th ACM Workshop on Digital Rights Management (DRM). pp. 69–78.

Trusted Computing Group, 2011. TPM main: Part 1 design principles.

Viega, J., Messier, M., Chandra, P., 2002. Network security with OpenSSL. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., Maltzahn, C., November 2006. Ceph: A scalable, high-performance distributed file system. In: Proceedings of 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI).

Wright, C. P., Martino, M. C., Zadok, E., 2003. NCryptfs: A secure and convenient cryptographic file system. In: Proceedings of USENIX Annual Technical Conference (ATC), General Track.

Zadok, E., Iyer, R., Joukov, N., Sivathanu, G., Wright, C. P., May 2006. On incremental file system development. ACM Transactions on Storage 2 (2), 161–196.