Experiences in Implementing a Kernel-Level DRM Controller

Alapan Arnab, Marlon Paulse, Duncan Bennett and Andrew Hutchison
{aarnab, mpaulse, dbennett, hutch}@cs.uct.ac.za
Data Network Architectures Group
Department of Computer Science
University of Cape Town
Rondebosch, 7701
South Africa

Technical Report Number: CS07-01-00

Abstract

The enforcement of DRM licenses is performed by a DRM controller, and it can be implemented at the application level, the operating system level and at a hardware level. In this paper we discuss our experiences in implementing an operating system level DRM controller based on the GNU-Linux kernel. This paper investigates the feasibility of creating a transparent, application independent DRM controller and the performance implications thereof. Our investigation has revealed, that while a number of access control rules can be enforced transparently at the operating system level, there are also a number of rules that require application level enforcement. Thus, we recommend separation of rights to two levels of enforcement to take advantage of transparent enforcement at the kernel level. Our performance analysis shows promise with minimal user observable time impact for small files less than 25 MB in size. However, there is still a significant performance impact and a very noticeable user observable time performance impact for larger files. Thus improvements are necessary if DRM controllers are to be deployed in multi-user, high-load environments like file servers.

1 Introduction

Digital Rights Management (DRM) has been attacked by many for reducing consumer choices [2], for impeding rights granted by copyright [15] and has been labelled by the editor of a leading technology news website as "*a load of C.R.A.P*" [8]. While DRM's use as a mechanism to combat media piracy is well known; use of DRM to provide "*persistent access control*" [22] for any data type is often ignored. In fact, DRM is often relabelled when deployed for use in enterprises, even though they share many common features and functionalities.

Many of the current problems with DRM systems stem from the fact that access control is exercised by "DRM-enabled" applications, while traditional access control is often administered by the operating system. These problems include vendor lock-in and limited application usability for consumers; while rights holders have less control over how well the protection is actually enforced. For this reason, the usefulness of DRM as a general security solution will depend on the ease of supporting and integrating with existing computer systems, which in turn depends on the implementation of the system that interprets and implements the DRM access control rules – the DRM controller.

1.1 Differences with existing Access Control Models

In [19], Reid et al. argue that mainstream operating systems are inappropriate for use as DRM clients. Mainstream operating systems use Discretionary Access Control (DAC) security policies rather than Mandatory Access Control (MAC) security policies. A DAC system allows ordinary users of a system to define their own security. By granting ordinary users this ability, a user could reconfigure the security policy of the system to subvert the DRM protection.

The authors also point out the inability of mainstream operating systems to support the principle of least privilege. Since system privileges are based on the users' identity, any program executing on behalf of a user is granted the same access control privileges as the user. There are no efficient mechanisms for restricting users' access control rights.

In MAC based systems, and the associated Multi-Level security (MLS) systems proposed by Bell and LaPadula [7, 6], access control is assured through a central security administrator, and thus ordinary users of the system are prevented from reconfiguring the computer's security policy [20]. However, for the purposes of DRM, the rights holders (or the owners of the data) are not guaranteed any control over the consuming device.

In MAC based systems, access control is based on the user's credentials, with users classified under a hierarchical structure, discussed in the Bell-LaPadula model [6, 7]. The hierarchical structure allows greater rights for some users while allowing lesser rights for other users. Protected objects are classified under this structure, and the object does not determine the level of access for the user. This creates a problem for DRM systems, where it is sometimes necessary to determine access according to the nature of the object as opposed to the classification of the user. For example, the author of an article can be classified as a rights-holder and a reader. However, the classification of the author as a rights-holder means that he has access to other works that are not necessarily his own.

The third and newest, popular access control model is Role-Based Access Control (RBAC), first described by Ferraiolo and Kuhn in [11], and subsequently detailed further by Ferraiolo et al. in [10], as well as Sandhu et al. in [23] and von Solms and van der Merwe in [24]. Ferraiolo and Kuhn argued that access to data should be determined by the function, and these functions of the users, which are usually defined by roles users play in an organisation [11]. For this reason, a role based access control model is more suitable than the DAC or MAC based approaches that were available at the time. von Solms and van der Merwe further argued that the role based approach is a combination of the resource based approach (as found in DAC) and the user based approach (as found in MAC) [24], a property that is desirable for DRM systems.

A pure role based approach is however not suitable for DRM, for two reasons. Firstly, a pure role based approach may not be able to distinguish access depending on the function of the object (as opposed to the function of the user), which would create a problem similar to the author-user problem discussed earlier. Secondly, the definition of roles are determined by individual organisations, and these roles vary from organisation to organisation; and sometimes differ within organisations. This problem could be solved by defining access roles with respect to the organisation (or department); but this would severely restrict portability of sensitive data between organisations.

The main difference between DRM and traditional access control however remains the boundary of control. Traditional access control models operate on an object within a defined boundary: either a system or organisation. DRM however aims to operate on objects that do not have any defined boundaries, and thus across different systems and organisations.

1.2 Our Contribution

In [4], we have introduced a formal access control model for DRM. In this paper, we explore the design and implementation of a scaled down version a DRM controller based on the model in the kernel of an operating system; instead of implementing it at the application layer.

DRM controllers at the operating system level, potentially provide some great benefits. The main advantage is that it should be possible for any application to access protected works, as the main underlying protection is provided by the operating system and not the individual applications. Furthermore, it also means that DRM protection can be offered to any data types, not just multimedia. This means that DRM could be used as a privacy enhancing mechanism for ordinary users, who can determine the exact access control rules for their own private data.

A different concern raised on the viability of operating system level implementations regards to the overall performance of the system. In [21], Rosenblatt argues that, since an operating system level DRM controller must intercept system call requests to enforce digital rights, it would need to intercept all such requests. It does not discriminate between system call requests that need protection and those that do not need protection. This unnecessary interception of system call requests can impose a large overhead on the system.

We believe that operating system and hardware level DRM controllers share similar constraints and capabilities. We also use our results to project the potential design considerations for hardware level DRM controllers and potential pitfalls. This is particularly important as new developments in processors have seen the first potential building blocks for a DRM controller at the hardware level [9].

1.3 Layout

The paper is structured as follows. In section 2 we discuss some related work in DRM controllers at the operating system level. Following this, we discuss our design and motivations for our design in section 3. We then discuss our experiences in implementing the controller in the GNU-Linux kernel in section 4. We then discuss our experimental evaluation of our system in section 5 followed by the analysis of our results in section 6 before drawing our conclusions in section 7.

2 Related Work

The majority of current DRM systems are implemented at the application level, and as discussed previously, only one system – Microsoft's Rights Management Services (RMS) – features a DRM controller in the operating system kernel.

Microsoft's RMS controller does not provide transparent DRM protection. Instead, it requires applications to be "*RMS enabled*" before they may interact with DRM protected files [14]. DRM Protected files in a non-RMS enabled kernel are seen as encrypted files and no actions can be performed on them. Applications which are not RMS-enabled cannot perform simple functions such as opening a file, even if the application is running in a RMS enabled kernel [14].

Because Microsoft RMS is a proprietary system, not much has been disclosed about its design, how its DRM controller interacts with the Windows NT kernel and the performance impacts its DRM controller has in relation to a normal Windows NT kernel. Since the enforcement of rights is through the RMS enabled application, it is difficult to do comparative analysis with our own operating system DRM model.

3 System Design

In this section, we detail an implementation agnostic design for an operating system DRM controller, and the motivations behind our design.

In [17], Park et al. categorised a variety of DRM systems. These categories depended on two factors – (1) the type of "*control set*" or use license used and (2) the distribution mechanism. Since DRM is a form of persistent access control, the distribution mechanism should not impact the security of the data. Hence, we did not consider the distribution as a factor. Park et al. discussed three types of control sets – the *fixed control set*, where the DRM policies are set with the DRM controller, the *embedded control set*, where the DRM policies are combined with the protected data and the *external control set* where the DRM policies are separate to the protected data providing maximum flexibility. These distinctions are also present in the OMA DRM 1.0 specifications and are called Forward Lock, Combined Delivery and Separate Delivery respectively [16]. Although our design could make use of embedded control sets, we only implemented and tested protected data with external control sets.

3.1 Overview

One of the core ideas in our model is the separation of a DRM controller into two distinct components: (1) a digital rights enforcement engine, which is embedded within the operating system kernel and is responsible for interpreting and enforcing access controls; and (2) a management daemon, which is responsible for the acquisition and management of all the data required by the enforcement engine when enforcing digital rights.

This approach was taken for three reasons. Firstly, by separating the DRM controller into two components, the responsibility for acquiring and managing DRM use licenses is removed from the the enforcement engine. This results in a reduced workload on the enforcement engine. Also, the enforcement engine is not required to be connected to any network (to acquire a license for example) or provide any user interfaces for managing use licenses. This is not a

security risk, as the enforcement engine is still responsible for checking validity and deciding on access requests.

The second reason for separating the enforcement engine and the management daemon is to allow for different implementation mechanisms for managing use licenses etc. For example, it would be easy to extend our design to incorporate a central management service, that serves a number of DRM controllers – a feature that would be very useful in home networks or corporate environments.

Finally, separation of the interface and the enforcement engine allows for easier portability. Many operating systems can be modified to run on different platforms and devices; all of which have different user interfaces. Very little changes are required at the kernel level for the enforcement engine; but the management daemon will need to take account of different storage systems and user interfaces.

In the remainder of this section, we will outline the responsibilities of each component, the interaction between the components and other related design issues such as file formats and rights expression languages.

3.2 The Digital Rights Enforcement Engine

The enforcement engine is responsible for making decisions on whether the user should be allowed a certain operation on a protected file. Thus, the enforcement engine needs to first identify the user requesting access to the protected file. It also needs the user to have a valid use license (a license that is not expired, and issued by a trusted authority) before it can decide if the use license allows the user access to the requested operation.

To make it's decision, the enforcement engine requires two pieces of information: a mechanism to verify the user's identity and the use licenses. The management daemon is responsible for providing information to allow for the first, as well as acquiring and managing such data.

3.3 The Management Daemon

3.3.1 User Management

Interoperable user identity management systems is one of the major problems with current DRM systems. User identity was also a major criticism levelled by by Reid et al. [19] with regards to the principle of least privilege. We use a Kerberos like, authentication ticket identity system, which we have previously discussed in detail in [3]. In this system, the user authenticates themselves to the authentication server, which provides him/her with a signed ticket identifying the user as authenticated for a specified amount of time on a specific machine. The daemon then presents this ticket (as long as it is still valid) to the DRM controller when required. Limiting the validity period reduces the scope of using replicated tickets as they last a short period of time and are bound to a specific machine. This approach also removes the need for user interaction (like passwords) at the time of use. If the ticket is not present, or is expired, the user is denied access.

Tickets are acquired and administered by the management daemon but it is the enforcement engine that ascertains the validity of the ticket, and thus a compromised ticket by the user would require the user to get the ticket issuer's private key. Our system allows any user to use a DRM protected work, as long as they have a valid license for that work, allowing for a high degree of portability.

3.3.2 Acquisition and Management of Use Licenses

If a license is not available in the license store, the management daemon initiates a negotiation for a new license with a content distributor's license server. The daemon provides a user interface to interact with the license server, and to communicate the necessary details required for the issuing of a use license. The management daemon should also be able to acquire use licenses from other sources. The daemon is also responsible for storing (if required) and indexing use licenses as well as removing revoked and expired use licenses.

3.4 Enforcement Engine – Management Daemon Interaction

Figure 1 gives an overall view of the process involved in accessing a DRM protected file in our system. The following steps describe the interaction between the enforcement engine and the management daemon.

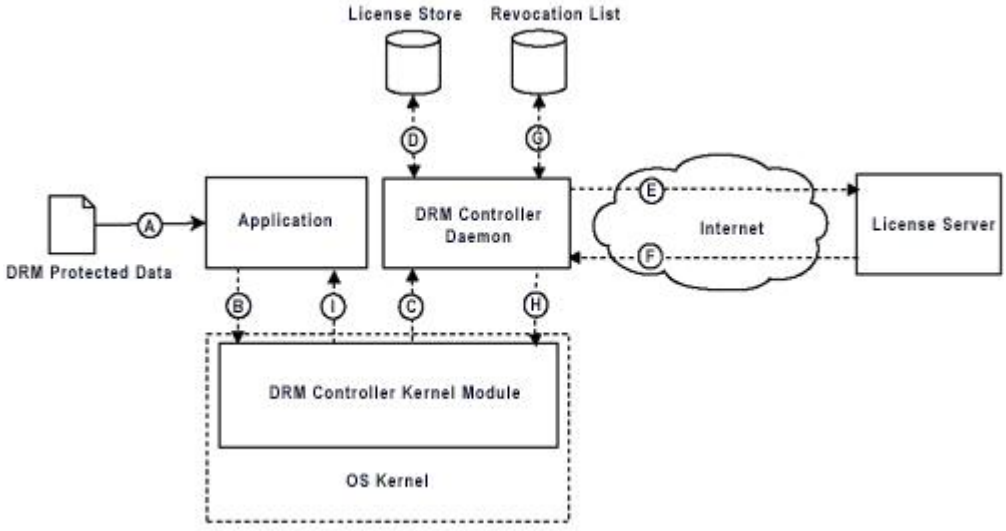

Figure 1: The DRM Controller Architecture and Communications

Step A: The application receives as input a DRM protected file.

Step B: The application requests access to the file. The enforcement engine intercepts this request.

Step C: The enforcement engine sends a request for the DRM use license details to the daemon.

Step D: The daemon checks the license store for a license. If a license exists, the daemon proceeds to step G. Otherwise, it proceeds to step E.

Step E: The daemon connects to a license server enabling the negotiation of a license download.

Step F: If a license is successfully negotiated, the daemon proceeds to step G. Otherwise, a message is sent to the enforcement engine to deny file access.

Step G: The validity of the license is checked against a revocation list. If a license is invalid, the daemon may return to step E to negotiate a new license.

Step H: The daemon sends the use license and the authentication ticket for the user, to the enforcement engine.

Step I: The enforcement engine performs a final check on the access request. The end-user and the request are referenced against the relevant fields in the use license. If these details are valid, the application is granted access to the requested file.

The daemon also manages the authentication tickets, and the steps to acquire and manage authentication tickets is similar to steps D to G described above.

3.5 DRM File Format

One of the most often mentioned problems with DRM system interoperability is the file formats used. We make use of a layered file format that can accommodate different requirements, and any data type. Our approach is shown in figure 2. The layers represent data that is added or a process that is applied to a previous layer. For example, metadata

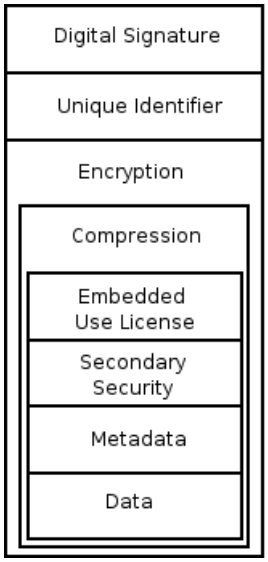

Figure 2: Layered approach to DRM file formats

is applied to the original data while compression is performed to all the data combined in lower layers. The layers can be used in either directions (i.e. to create or to use the protected data). We have also catered for other security mechanisms that can be applied to data, usually before encryption, like watermarking.

4 Implementation Experience

4.1 Setup

The system implements a DRM controller responsible for enforcing rights protection on an end user machine. The controller represents part of the user component in the architecture discussed above. As discussed earlier the aim of the implementation is to create a controller that supports multiple file formats and is transparent to the applications that access the files. Due to the wide availability of literature on Linux kernels and its open source nature, we decided to implement the system on a Linux-based operating system with a *vanilla* 2.6.15 version Linux kernel.

The controller consists of two core modules: an operating system kernel module and a user-space daemon. The kernel module is responsible for enforcing the access control rules specified in DRM use licenses, while the daemon retrieves these use licenses from the content publisher's license servers and manages them in its local license store. When an application requests access to a DRM protected file, the daemon retrieves the DRM use license and sends it to the kernel module where the rights enforcement will occur. A more detailed description of this communication is given in section 3.4.

The communication between the kernel module and the daemon is performed via a character device file in the */dev* directory. Ideally, this communication channel would need to be secure, tamper-proof and only accessible to the kernel module and the daemon. However, we were unaware of how to implement secure userspace-to-kernel communication in Linux. The management daemon does not modify any data it manages, and thus all data is signed by the original services. Thus, the confidentiality of the communication is not breached by our approach.

4.2 File Format

The file format used by our DRM controller implementation is presented on the right in figure 3. As shown, we closely followed the layering architecture, except for an additional layer at the top representing the magic number of the DRM protected file. The magic number gives the file format an unique MIME type identifier allowing the kernel module to quickly identify a DRM protected file and to ignore non-DRM protected files, and as far as we are aware, the number

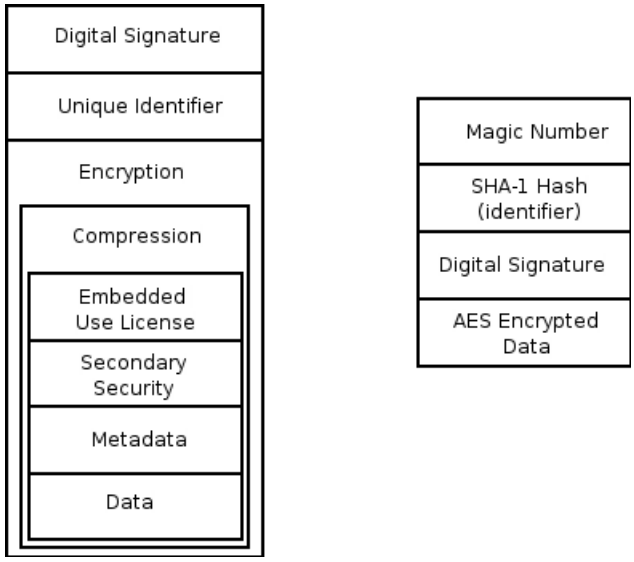

Figure 3: Layered approach to DRM file formats (left) and the implemented file format (right)

we have chosen is not being used by any other MIME type.

At the moment we use a SHA-1 hash of the unencrypted content as the identifier. We admit that this approach is not ideal, as there is a chance that two different objects can have the same message digest, and a more complete identifier system, is necessary in a full implementation. This approach however is better than a simple labelling scheme as it provides a primitive identity verification service, that is not provided by common identifier schemes. The protected file is encrypted using AES (128bit).

4.3 Rights Expression Language (REL)

We wanted to reduce the complexities involved in our test implementation and thus used a flat file representation for the use license instead of XML based license like ODRL or XrML. The format we used is shown in figure 4, and our format follows the core set of elements required for license enforcement as discussed by Guth et al. in [12]. Each license was signed and also contained the the user identifier. This flat file model is also compatible to the formal model we have discussed in [4].

```
License         ::= ( Permission )+ End
Permission      ::= PermissionStart Type ( Constraint )* End
Constraint      ::= ConstraintStart Type ( Argument )* ( Constraint )* End
Argument        ::= ArgumentStart Type ArgumentValue End
ArgumentValue   ::= ( 1-9a-zA-Z )+
Type            ::= ( 1-9 )+
End             ::= ``;''
PermissionStart ::= ``!``
ConstraintStart ::= ''&''
ArgumentStart   ::= ``@``
```

Figure 4: The BNF grammer for the kernel license

4.4 Access Controls Implemented

In ODRL, rights are known as *permissions* and limitations to these permissions are known as *constraints*. Thus a permission like **display** can have a constraint of **count** with a value of 5 to restrict the end user to viewing a file for a maximum of 5 times. For the implementation, we focused on the following permissions which we extracted from the ODRL 1.1 core data dictionary. These permissions reflect a fair percentage of the total permissions expressible in the standard ODRL data dictionary.

1. **DELETE**
2. **DISPLAY**
3. **EXECUTE**
4. **MODIFY**
5. **SAVE**
6. **MOVE**

We also implemented the following two ODRL constraints:

1. **COUNT**
2. **DATETIME**

4.5 The Enforcement Engine: The Operating System Kernel Module

The kernel module consists of five components:

1. **Enforcement Component (EC)** , which enforces the rules specified in DRM use licenses
2. **Decision Component (DC)** , which decides whether an application may access a DRM protected file in a certain manner, based on the permissions and constraints specified in the DRM use license.
3. **Access Control Rules Manager (ACRM)** , which parses the kernel licenses (shown in figure 4) received from the user-space daemon into an internal data structure that allows for more efficient in-memory storage and look-ups. The ACRM also serialises the use license data structures back into the kernel license format so that any modifications made to the license constraints by the kernel can be sent back to the daemon for storage in its license store.
4. **Data Handling Component (DHC)** , which is used to determine whether files being accessed are DRM protected, perform digital signature verification and to decrypt DRM files. This component is also responsible for storing the state of all open DRM protected files in the system.
5. **Communications Interface (CI)** , which implements the character device driver that provides the communication mechanism between the kernel module and the daemon.

These components are shown in figure 5. The kernel module operates as follows:

Step A: The kernel module receives a file access request from an end-user application.

Step B: The EC uses the DHC to determine whether the file is DRM protected. If it is, the DCH verifies the digital signature embedded in the DRM protected file. If the digital signatures is successfully verified, the DRM protected file is decrypted.

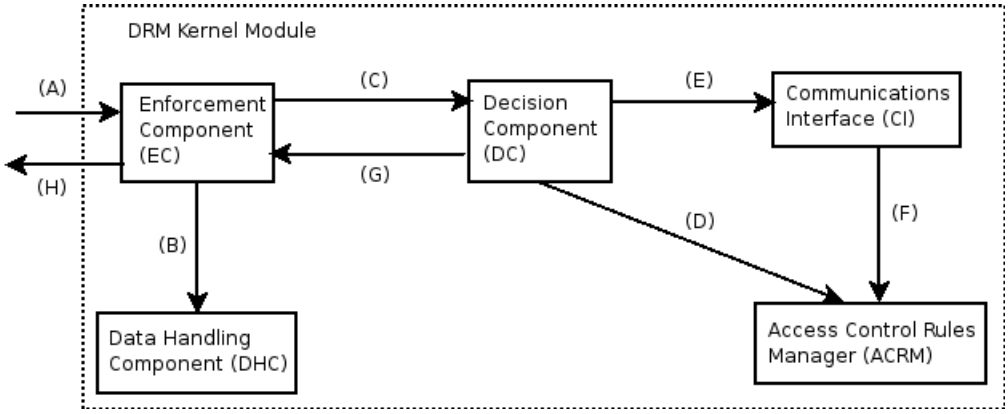

Figure 5: The interaction of the various components withing the kernel module

Step C: The EC checks with the DC whether the application can access the file in the manner it requested.

Step D: The DC requests the DRM use license in the kernel module's in-memory license cache from the ACRM.

Step E: If the DRM use license is not available in the kernel module's in-memory license cache, the DC retrieves the license using the CI

Step F: The CI communicates with the daemon to retrieve the license from the daemon's local license store.

Step G: The license from the daemon is sent to the ACRM so that it can be parsed into a data structure that the DC can use to efficiently look-up permissions and constraints. The license is also saved in the kernel module's in-memory license cache, in case the end-user application performs another file accesxs request in future.

Step H: The DH checks if the application may be granted access by looking-up the permissions in the license data structure constructed by the ACRM and verifies that all the license constraints are satisfied. The DH notifies the EC whether access may be granted to the application

Step I: Based on the response by the DH, the EC grants or denies the application access to the file.

It should be noted that the digital signature verification and file decryption processes only occur once: when the application opens the file. Subsequent file access requests, such as DISPLAY, do not incur this overhead. Also, for simplicity, decrypted DRM protected files are stored in the */tmp* directory. Temporary files are required because all the application's file requests are redirected from the encrypted file to the temporary, unencrypted file, and performing the operation in memory proved to be too expensive. We recognise the security risk introduced by this approach, and we discuss this in more detail in section 4.7 . To lower the security risk, the temporary files were given random names, and the redirection does not appear under normal process listing (such as ps). Once the application closes the file, the these temporary files are deleted.

In order to enforce the DRM use license permissions listed in section 4.3, we define mappings between permission names and system call routines. These mappings are defined as follows:

1. **DELETE** : unlink
2. **DISPLAY** : read
3. **EXECUTE** : execve
4. **MODIFY** : write
5. **SAVE** : write

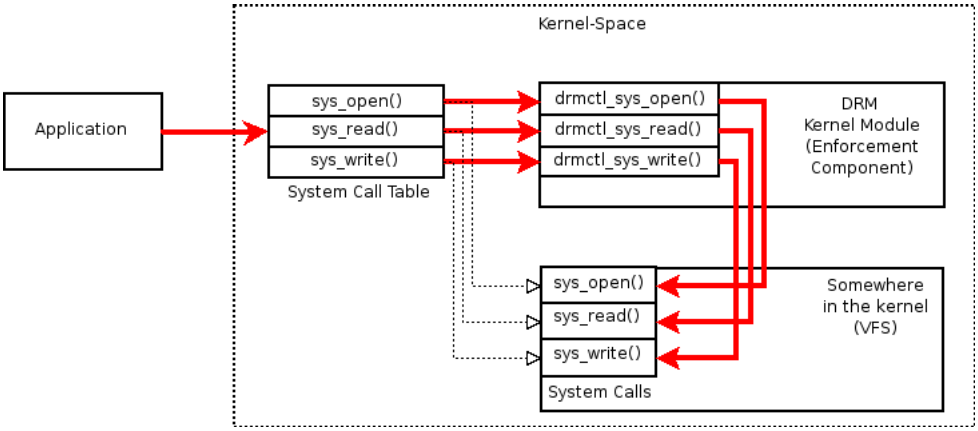

Figure 6: Intercepting system calls and redirecting file access requests in the kernel

6. **MOVE** : rename

Therefore, whenever an application makes a request for a permission, all the kernel module has to do when access is granted is to execute the corresponding system call routine. If permission is denied, the kernel module simply needs to return from the system call routine with an error message. In order to implement this procedure, the EC replaces the I/O system calls in the Linux kernel with its own set of system calls. This means that whenever an I/O request is made, the EC's system calls are called instead of the original Linux kernel system calls, thus allowing the DRM kernel module to perform the DRM access control verification as described above. This is accomplished by replacing the addresses in the system call table that point to the original I/O system call routines with the addresses of the kernel modules I/O system calls. Figure 6 illustrates this process. The dotted arrows show the normal flow of control from an application request (invoking a system call) to the original system call routines that service the request. By changing the values in the system call table, control instead passes to the kernel modules system call routines, as indicated by the bold arrows. If an end-user application is denied access to the DRM protected file, the EC simply exits its custom system call routines, returning an error code (*-EACCES*) to the end-user application. If access is granted, the EC calls the original system calls from within its custom system call routines, enabling the end-user application to carry out the action associated with the access request.

4.6 The User-Space Daemon

The user-space daemon has several responsibilities which are described in more detail below.

4.6.1 Management of a License Store

In order to retrieve DRM use licenses for the kernel module, the daemon stores licenses in a local license store. The daemon is responsible for acquisition of license (over the Internet), storing and indexing retrieved licenses and removing expired and revoked licenses.

4.6.2 Negotiate Licenses with a License Server

If a license is not available in the license store, the daemon initiates a negotiation for a new license with a content distributor's license server. The daemon activates a user interface, and a child process is started to await a response from the user interface. This frees the main daemon process to continue communicating with the kernel. Once the negotiation is complete, a message is sent to the child process. If the message contains a license, the license is added to the license store. The user can then try to attempt to access the file once more.

4.6.3 Management of authentication tickets

As discussed in section 3.3.1, we make use of authentication tickets for user authentication. The daemon is responsible for acquiring authentication tickets (over the Internet) and the storing and indexing of authentication tickets, removing

expired authentication tickets.

4.6.4 Communications with the Kernel

When requested by the kernel, the daemon finds the appropriate use license and associated authentication ticket and then sends them to the kernel.

4.7 Motivation for our approach

The use of a temporary decryption file is not ideal. Firstly, it requires that the entire DRM protected file be decrypted before the end-user application may access it, incurring a big performance penalty on large files. A better approach would be to implement on-the-fly decryption where the file decryption and application access may occur simultaneously. Also, using a temporary file complicated the updating of DRM protected files when end-user application have MODIFY permissions. With our approach, if an update should occur, the kernel module would need to re-encrypt the temporary file and move replace the original DRM protected file with the newly encrypted file. Not only does this incur further performance losses, but ensuring consistency between the original DRM protected file and the newly created protected file in multi- threaded or parallel processing environments would be very difficult. Lastly, the there may also be not enough space on the computer on which the DRM protected file is being accessed. This is especially a problem when the DRM protected file is large. However, despite these drawbacks, we still opted for this approach, as it is simple to implement and will provide a reasonable indication of the performance losses involved in decrypting DRM protected files in the kernel. Furthermore, using temporary files also gets around memory limitations that are experienced if the unencrypted file is stored in memory.

5 Experimental Evaluation

In this section, we describe the experiment that was conducted to determine the performance cost imposed on the system by DRM controller implementation, and present and analyse the results thereof.

5.1 Experiment Setup

The following two system calls were used during this experiment: read() and rename(). These system calls correspond to the DRM use license permissions, DISPLAY and MOVE, respectively. They were chosen for this experiment because they represent the two types of access operations which can be performed by the file system of an operating system on the data stored on the disk. The read() system call performs sequential accesses on each byte of the data, whereas the rename() system call only operates on the entire chunk of data - a file - as a whole.

The experiment involved the following three tests. First, we measured the duration of the read() and rename() system calls in a standard Linux kernel when accessing non-DRM protected files of various sizes. Then, we determined the system call overhead of the two system calls when the DRM controller kernel module was enabled. As in the first test, all the files used in this test were non-DRM protected. Finally, we repeated the second test, but this time, used files which were DRM protected.

Two sets of files were used in our experiment. The first set contained regular non-DRM protected files, while the second set contained DRM-protected copies of the files in the first set (encrypted with AES in ECB mode with key length of 128 bits). Each set consisted of six files of various types and sizes as shown in table 1. We believe that these files represent a good selection of real world digital works that could be protected using DRM.

For each test-run, the system calls were invoked 100 times per file. The duration of the system call was then determined by taking the average of the 100 measurements. These results are presented in tables 2 and 3.

5.2 Test Environment

The experiment was conducted on an Intel Celeron computer with a CPU clock speed of 1.7GHz, 512 MB of RAM and a 40Gb 7200RPM PATA hard drive. In most respects, these hardware specifications can be considered as representative of an average end-user machine. The computer was loaded with a typical desktop installation of Linux running a

File type	File size
Text file	98 B
PDF document	38.517 KB
JPEG image	1757.283 KB
MP3 audio file	4.670 MB
MPG video file	23.848 MB
GZIP tarball	102.401 MB

Table 1: The types and sizes of the files used during the performance evaluation experiment.

vanilla 2.6.15 version kernel.

5.3 Results

Tables 2 and 3 show the results of the three tests for the read() and rename() system calls respectively. In each table, the performance costs incurred by the DRM controller are shown.

File size (B)	**Std kernel Non-DRM data access time** (μs)	**Std kernel + DRM ctl. Non-DRM data access time** (μs)	**Std kernel + DRM ctl. DRM data access time** (μs)	**Non-DRM data overhead** (%)	**DRM data overhead** (%)
98	23.476	30.758	55275.004	31.019	235353.246
39441	71.409	84.297	55303.169	18.048	77345.657
775458	835.512	905.349	83927.375	8.358	9945.023
4896677	4840.858	5041.487	236563.284	4.144	4786.805
25006182	24695.321	25528.383	1119149.073	3.373	4431.826
107375252	104913.480	205856.577	7664316.603	96.216	7205.369

Table 2: Comparing the duration of the read() system call when handling DRM protected and non-DRM protected data on a DRM-enabled and DRM-free system.

File size (B)	**Std kernel Non-DRM data access time** (μs)	**Std kernel + DRM ctl. Non-DRM data access time** (μs)	**Std kernel + DRM ctl. DRM data access time** (μs)	**Non-DRM data overhead** (%)	**DRM data overhead** (%)
98	49.695	62.341	73.251	25.447	47.401
39441	48.963	58.061	69.188	18.581	41.307
775458	50.041	60.247	70.498	20.395	40.880
4896677	50.434	60.779	71.460	20.512	41.690
25006182	49.641	67.955	71.009	36.893	43.045
107375252	50.667	62.769	76.860	23.885	51.696

Table 3: Comparing the duration of the rename() system call when handling DRM protected and non-DRM protected data on a DRM-enabled and DRM-free system.

5.4 Analysis

There are three areas in the DRM controller which contribute to a performance overhead:

1. Intercepting the access request and detecting DRM protected data.

2. Communicating with the daemon.

3. Parsing the use license and enforcing the rights specified in the license.

The cost of intercepting the access request and detecting whether the request applies to DRM protected data occurs regardless whether the data is DRM protected or not. This is the stage where the DRM controller distinguishes between access requests that need DRM protection and access requests that do not. Assuming that the daemon communications and rights enforcement cost is negligible, this cost will be the best-case performance cost that will be incurred by the DRM controller.

Table 2 shows the result of the three tests for the read() system call. We see that the performance costs imposed by the DRM controller when accessing non-DRM protected information, there is an increase in the overhead by 31% for a 98B file, decreasing until it reaches a near 3.4% performance overhead cost for a 23.8MB file. This suggests that the overhead from intercepting access requests and detecting DRM data is so small compared to the overhead of communicating with the daemon and enforcing the license rights, that it is negligible. The large jump in the overhead for large files however does suggest that there might be other factors that have influence, other than the daemon-kernel communication.

Looking at table 3, we see the access times when non-DRM protected content in a standard kernel with the DRM controller enabled remain almost the same. This is as expected, as the rename() system call performs only one access on the data, regardless on the size of the data. We attribute the discrepancies in the access times to the arrangement of files on disc and possible variances in system load.

When accessing DRM-protected content on a DRM-enabled system using the read() system call, we observed a similar trend as in the non-DRM protected case. Although the access times increase as the file sizes increase, the performance overhead decreases. Initially, we find a 235353% increase in performance cost. This high cost increase is due to the large overhead involved when communicating with the daemon, parsing the license, and traversing the in-memory license structure to enforce digital rights. As more read requests are performed this cost becomes less noticeable, and drops to approximately 4431% for a 23.8MB file.

Table 3 shows the performance results for rights enforcement where decryption is not required. In this case, the overheads are introduced in the daemon-kernel communication and the interpretation of the licenses. As can be expected, the overheads involved are almost constant, and have no user obeservable time performance effect.

In both the read() and rename() cases, the overhead due to the daemon communications and the rights enforcement far outweigh the cost of intercepting access requests. This, of course, raises questions regarding the infrastructure of the DRM controller. If a file found to be DRM protected by the kernel, it must start a expensive communication with the daemon. This costly process might best be avoided by introducing a hardware implementation of a license store, instead of file-system based one which is managed by a user-space application. If the license store was managed by kernel, the need to request licenses from the daemon would be removed. However, there might still be a cost, as the the kernel still needs to establish communication with the daemon to allow it to retrieve licenses from remote license servers.

Even though there are high costs incurred, the net effect on time is minimal and will not be noticeable by the end user for single user machines. This analysis will however not hold for multi-user or high performance machines and thus improvements are necessary before it can be considered for deployment.

We are also need to consider the frequency at which accesses to DRM protected content are made. If access to protected content is relatively infrequent, then the huge cost might still be judged to be negligible, since the performance cost is still sufficiently small that a human user would not notice it. However such a judgement cannot be reached without further investigation of file access patterns, such as those created by web servers, or a multi-user system where the majority of files are protected.

6 Analysis of our Approach

The current system has been built in order to test the feasibility of an operating system level DRM controller. We have tried to make it as complete as possible, but our implementation is not a complete solution for DRM. In this section, we discuss how well our approach works in achieving its goals, as well as detailing some issues that we feel need to be addressed for a more complete system.

6.1 Application Level Transparency

Our design allows for any application to access any DRM protected file, and application behaviour is not affected, other than the ability to modify a file. We have tested on a wide variety of GNU-Linux applications, including various PDF readers (examples: Ghostview, xpdf, Gnome PDF Reader), different media players (examples: mplayer, xmms, mpg123) and text editors (examples: vim, gvim, kwrite).

6.2 Wide range of rights

We have implemented most of the rights that can be enforced at an operating system level, and we feel that there certain rights (like printing) that can only be enforced at the application level. This is discussed in more detail in section 6.4.

6.3 Performance

In [18], Raskin discusses how humans can easily pick up changes in application behaviour once they become used to the applications. He gives a small boundary of a couple of seconds, before changes like longer loading times, or sluggish operation will become noticable to the average user. In [5], Arnab and Nunez, determined experimentally that users are willing to wait, on average, 5.5 seconds longer if they are aware that there is a security operation that needs to be performed before they are able to access the data. We have therefore chosen to use this value as the threshold for performance degradation.

As detailed in table 2, for files up to the size of 23.8 MB, the performance degradation was quite low, with the largest file having a degradation of 1.1 seconds for a DRM enabled file. However, for a large compressed data file with a size over 102.4 MB, the performance degradation is quite significant at 7.7 seconds for a DRM enabled file. However, the performance degradation for non DRM enabled files remained negligible.

Thus, while our system is quite suitable for small data files like music and PDF documents, it does not meet the performance requirements for larger files, like movies.

6.4 Interpreting Rights Expressions

Some rights are harder to enforce since they cannot easily be identified at the kernel level. For example, ODRL defines limitations to be placed upon the number of pages that an end user may print. Within the GNU-Linux kernel there is no concept of a document page because they only exist within applications such as word processors that need to support such an entity. Similar problems will exist with hardware implementations of DRM controllers.

Thus, we believe that there needs to be two levels of rights in a DRM system if an operating system or hardware level implementation is to be successful.

1. **Level 1:** These rights (or permissions in ODRL) are common to all operating systems and devices. They would include the permissions we have implemented in our prototype, such as restrictions on reading a file, and will in effect be persistent extensions of existing access control rules. Thus these restrictions will be enforced at the operating system or hardware level.
2. **Level 2:** These rights are only enforceable at the application layer because only certain applications will be able to make sense of the rights. Rights like the permission to print a certain number of pages or the portions of a document that can be excerpted only make sense to the application handling the file.

With the above categorisation, it will no longer be necessary to require application support if the DRM protection is

restricted to Level 1 rights. It should be possible to specify in Level 1, the right to access a file only with a specific application thus preventing bypassing of Level 2 restrictions. Our approach of using a management daemon becomes particularly important, as this daemon can be used by DRM enabled applications for the same functions. In table 4, we categorise all the rights and permissions defined in the core ODRL 1.1 [13] and XrML [1] (the base for MPEG-REL) RELs. Some rights, such as Embed and Extract can be implemented at the operating system layer, but would require application support for maximum effect.

Level 1 Rights	**Level 2 Rights**
1. **Usage Rules:** Display, Execute, Play, Read 2. **Reuse Rules:** Aggregate, Edit, Embed, Excerpt, Extract, Modify, Write 3. **User Management:** Give, Lend, Lease, Load, Sell, Transfer 4. **Asset Management:** Backup, Copy, Delete, Install, Export, Restore, Save, Uninstall, Verify	1. **Usage Rules:** Print 2. **Reuse Rules:** Aggregate, Annotate, Edit, Extract, Excerpt, Embed, Modify, Write

Table 4: Classification of Level 1 and Level 2 Rights

6.5 Modification of Protected Files

Allowing and disallowing read only functionalities is easily accommodated, but the major problems occur when trying to cater for modification of protected data. To cater for modification of data, functionality to re-identify and repackage the data needs to be provided at the kernel level. If modification of the the protected file is frequent (capturing event data or even traditional office file), these operations will severely slow down saving of an application. In our solution, we did not provide these functionalities, and only provided for the outright prevention of modification of data. In our opinion, some level of application support is required before modification of data is seamless.

6.6 Correct Identification of Accesses

More complex applications may break a single user level access into several smaller accesses. For example, playing a music file may require multiple read attempts although only a single play permission is exercised. The DRM controller must be able to correctly identify the purpose of these calls. If it does not, a user's access rights may expire prematurely, Consider the example of a "play" permission limited by a count constraint. The count must be decremented only when the media starts to play and not for each read access.

6.7 Stream Encryption

Our solution currently does not handle stream encryption even though, in theory, it would seem to be faster and more secure solution. However, most applications tend to load files in their entirety instead of a portion of a file due to a variety of reasons including compression techniques and metadata storage. For this reason, stream ciphers would be impractical without application level support, and would make sense for only certain file types.

6.8 Compensating for Application Behaviour

Some applications behave unexpectedly. For instance, multiple access attempts may be made before an application determines that a file cannot be accessed. The daemon module must compensate and distinguish genuine requests from repeat requests. This is to avoid initiating multiple license negotiations for the same asset.

6.9 Implications for hardware based DRM systems

We think that there would have been a better performance from our system if the license and authentication tickets were stored in hardware. However, such a store would have restricted memory, and this could affect the overall system. Memory is also the main factor to consider for hardware DRM controllers. As we have discussed, stream based encryption is unpractical for the general case, and this implies that the hardware DRM controller will need to store the decrypted DRM file somewhere while it is being used. Making use of a dedicated memory store for the controller would be the most secure approach, but this would limit the number and size of secure data files that can be accessed simultaneously to available memory. Modification of files would remain a problem, although the process of repackaging should be faster. However, application level support will still be necessary to make the process seamless.

We think that hardware based DRM will ultimately offer the advantages that is offered by kernel level DRM controllers, at a better performance. Furthermore, there should be no reason why open source software cannot make use of the hardware based DRM to provide persistent access control, as long as the relevant drivers are available.

7 Conclusions

In this paper, we presented a prototype DRM controller, which enforces rights to digital content at the operating system level of a computer. It offers DRM protection for any file format, transparently to any user-space application trying to access the DRM protected content. We described its implementation, and discussed an experiment which was conducted to evaluate its performance. We also looked at how effectively it is able to enforce digital rights and examined the viability of the approach.

The existing prototype does not address all the problems of implementing a DRM controller. Our system does have a security flaw that the unencrypted data needs to be stored because the use of stream ciphers is not possible in the general case. However, if a secure temporary storage solution could be implemented, the system does demonstrate that an operating system level DRM implementation is possible, supporting multiple file formats and remaining transparent to applications.

However, not all access control rules can be enforced at the application layer and indeed any future hardware layer implementations. This leads to a separation of types of permissions into two levels – permissions can be easily enforced by the operating system (or future hardware DRM controllers) and permissions that need to be enforced by applications. Thus, there is a need to categorise permissions according to these levels, and we have categorised existing rights as defined by two popular RELs into these categories.

In addition, the performance overhead for unprotected data should not increase in a fully implemented system, and while operations on protected data should incur a performance degradation, the degradation should be minimal. In the later case, it is sufficient to ensure that the user remains unaware of performance overhead, and 5.5 seconds was discussed as the maximum user observable time degradation.

However, for large files, the degradation is more than the threshold, at 7.7 seconds for a 102.4 MB data file for some operations. However the user observable time performance degradation is acceptable for files, with a 23.8 MB file requiring only 1.1 seconds longer to read. Thus, for current DRM requirements, such as protection of music, the performance is acceptable. In computational time, the performance degradation is very high for certain file operations and needs to be improved before real world implementations can be considered. However, for non protected data, the performance degradation was well within the set boundaries.

8 Acknowledgements

This work is partially supported through grants from the University of Cape Town (UCT) Council and the National Research Foundation (NRF) of South Africa. Any opinions, findings, and conclusions or recommendations expressed in this paper/report are those of the author(s) and do not necessarily reflect the views of UCT, the NRF or the trustees of the UCT Council.

References

[1] *eXtensible rights Markup Language (XrML) 2.0 Specification*, 2001.

[2] DRM From the Viewpoint of the Electronic Industry. *Slashdot* (2003).
URL: http://slashdot.org/article.pl?sid=03/11/25/1821218.

[3] ARNAB, A., AND HUTCHISON, A. Ticket based identity system for drm. In *Proceedings of Information Security South Africa (ISSA) Conference 2006* (2006).

[4] ARNAB, A., AND HUTCHISON, A. Persistent access control: A formal model for drm. Submitted for consideration to ACM SACMAT 2007.

[5] ARNAB, A., AND NUNEZ, D. Loading ... a short study into security delay frustration. Tech. Rep. CS06-03-00, University of Cape Town, 2006.

[6] BELL, D. E., AND LAPADULA, L. J. Secure computer system: Unified exposition and multics interpretation. Mtr-2997 rev. 1, The MITRE Corporation. Online, last accessed: 2006-05-06.
URL: http://csrc.nist.gov/publications/history/bell76.pdf.

[7] BELL, D. E., AND LAPADULA, L. J. Secure computer systems: A mathematical model. *Journal of Computer Security 4*, 2/3 (1996), 229 – 263. Reprint of 1973 technical report M74 244, MITRE Corp.

[8] BERLIND, D. A load of C.R.A.P. *ZDNet.com* (2005).
URL: http://news.zdnet.com/html/z/wb/6035707.html Last Accessed: 2007-01-06.

[9] EVERS, J. Ibm looks to hardwired DRM. *ZDNet-UK*. Online, last accessed 2006-05-05
URL: http://news.zdnet.co.uk/business/legal/0,39020651,39262333,00.htm.

[10] FERRAIOLO, D. F., CUGINI, J. A., AND KUHN, D. R. Role-based access control (RBAC): Features and motivations. In *Annual Computer Security Applications Conference* (1995), IEEE Computer Society Press.
Available online: http://csrc.nist.gov/rbac/ferraiolo-cugini-kuhn-95.pdf.

[11] FERRAIOLO, D. F., AND KUHN, D. R. Role-based access control. In *Proceedings of the 15th NIST-NSA National Computer Security Conference* (1992).
Available online: http://csrc.nist.gov/rbac/ferraiolo-kuhn-92.pdf.

[12] GUTH, S., NEUMANN, G., AND STREMBECK, M. Experiences with the enforcement of access rights extracted from ODRL-based digital contracts. In *Proceedings of the 2003 ACM workshop on Digital Rights Management* (2003), ACM, pp. 90–102.

[13] IANNELLA, R., Ed. *Open Digital Rights Language (ODRL) 1.1*. IPR Systems Pty Ltd., 2002.
URL: http://odrl.net/1.1/ODRL-11.pdf.

[14] MICROSOFT. Technical overview of windows rights management services for windows server 2003. White paper, 2003.

[15] MULLIGAN, D., HAN, J., AND BURSTEIN, A. How DRM Based Content Delivery Systems Disrupt Expectations of "Personal Use". In *Proceedings of the 2003 ACM workshop on Digital Rights Management* (2003), ACM, pp. 77–89.
URL: http://doi.acm.org/10.1145/947380.947391.

[16] OPEN MOBILE ALLIANCE (OMA). Oma digital rights management v1.0. Approved version, 2004-06-25. URL: http://www.openmobilealliance.org/release_program/drm_v1_0.html.

[17] PARK, J., SANDHU, R., AND SCHIFALACQUA, J. Security architectures for controlled digital information dissemination. In *Proceedings of the 16th Annual Computer Security Applications Conference* (2000).

[18] RASKIN, J. *The Human Interface – New directions for designing interactive systems*. Addison-Wesley, 2000.

[19] REID, J. F., AND CAELLI, W. J. DRM, Trusted Computing and Operating System Architecture. In *Conferences in Research and Practice in Information Techology* (Newcastle, Australia, 2005), vol. 44, Australian Computer Society, Inc., pp. 127 – 136.

[20] RHODES, T. Chapter 15 – Mandatory Access Control. FreeBSD Handbook, FreeBSD.org. Online, last accessed: 2006-05-06.
URL: http://www.freebsd.org/doc/handbook/mac.html.

[21] ROSENBLATT, B. DRM for the Enterprise, 2004. Jupiter Media Webinar.

[22] ROSENBLATT, B., AND DYKSTRA, G. Integrating content management with digital rights management - imperatives and opportunities for digital content lifecycles. White paper, Giantsteps Media Technology Strategies, 2003.
URL: http://www.giantstepsmts.com/drm-cm_white_paper.htm.

[23] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *IEEE Computer 29*, 2 (1996), 38 – 47.

[24] VON SOLMS, S. H., AND VAN DER MERWE, I. The management of computer security profiles using a role-oriented approach. *Computers and Security 13*, 8 (1994), 673 – 680.