The Implications of an Operating System Level DRM Controller

Duncan Bennett, Marlon Paulse, Alapan Arnab and Andrew Hutchison

DATA NETWORKS ARCHITECTURE GROUP
Department of Computer Science,
University of Cape Town,
Rondebosch 7701, South Africa
{aarnab, dbennett, hutch, mpaulse}@cs.uct.ac.za

ABSTRACT
Digital Rights Management (DRM) is the persistent access control of digital content. This paper examines the implications of enforcing access control rules at the operating system kernel level. We describe our design for a prototype, operating system kernel level DRM Controller. Initial performance benchmarks measuring access time have yielded promising results. Only negligible overhead was measured unprotected data, while the overhead incurred for protected data is unnoticeable to a human user. Lastly, we discuss implications of operating system level DRM.

1. INTRODUCTION
Traditional security mechanisms, such as encryption, operating system file access permissions, database access controls and firewalls are widely used to protect confidential and sensitive data from unauthorised use. These security mechanisms offer sufficient levels of security and access control to the data. However, effective protection can only be guaranteed as long as the data reside within a secure environment, such the content creator's own computer system. Once the data are taken out of the secure environment, the protection offered by firewalls and data access controls are completely lost. Also, if an end-user obtains access to an encrypted file and knows how to decrypt it, there is nothing stopping that end-user from passing the data in its decrypted form onto a malicious end-user.

In contrast, digital rights management (DRM) provides persistent access control. It allows creators of digital content to specify exactly which end-users are allowed to access the content, how they may access it, and under which circumstances it may be accessed. Even if the data is transported to another computer in an insecure computing environment, DRM protection will prevent any malicious user from using the data in an unauthorised manner. DRM provides mechanisms to manage and enforce rights to digital content, and thus, ensures the persistent access control of digital content [5].

Arnab and Hutchison [2] defined a specification for a DRM framework, consisting of a set of distributed components that create, administer and distribute DRM protected content. We implemented a prototype operating system level DRM controller for this framework, that enforces digital rights specified in DRM use licenses and offers DRM protection for any file format. Our aim was to evaluate the performance of this DRM controller in order to assess the viability of DRM at the operating system. Additionally, we wished to see how effectively digital rights could be enforced.

We defined two hypotheses for our investigation:

1. The prototype DRM controller can enforce a wide range of digital rights, transparently to any end-user applications trying to access the DRM protected content.

2. The performance cost imposed on the system by the prototype DRM controller is negligible.

This paper is structured as follows. In section 2, we present some background on existing DRM implementations. We then give an overview of the DRM controller implementation in section 3. We describe an experiment that was conducted to evaluate the performance of the DRM controller, followed by a detailed analysis of the results of this experiment The implications of operating system DRM with respect to the enforcement of digital rights is discussed in section 5. We then conclude by assessing the viability of operating system DRM.

2. BACKGROUND AND MOTIVATION
A DRM controller may be implemented at one of three levels in a computer:

1. the application-level,

2. the operating system-level, or

3. the hardware-level.

Many existing DRM solutions are application level solutions often designed to support one file format or product line. Apple's iTunes music store provides an example of such a system. iTunes enables the sale of digital content online through the use of a proprietary DRM called Fairplay [3]. The strength of the iTunes music store lies in the fact that Fairplay is able to support many of the users' rights under traditional copyright law [1]. At the same time, it is powerful enough to prevent piracy and effectively enforce the rights of the copyright holder [1]. Without balance between the rights of the copyright holder and the end-user, iTune's would arguably have experienced less success with their online music sales.

While Fairplay represents a significant advance in the protection and distribution of digital media, it has some associated problems. Firstly, Fairplay is proprietary and Apple will not license the DRM [3]. This creates artificial incompatibilities between applications because without a license, it is not legal for another company to support Fairplay [3]. A major concern here is fair trade: if no other company can support Fairplay in their software or hardware, this may limit their ability to compete with iTunes. This ultimately inconveniences the end-user who gets locked into a single software solution.

Another problem with an application level DRM such as Fairplay is that it is format and application specific. For instance, Fairplay only protects a particular music format. This lack of generality at the application level often leads to multiple, incompatible DRM implementations [1]. There is thus no single, stable DRM standard, representing a more significant problem for organisations attempting to maintain large archives or work. DRM enforced at the application level is also less secure since operating system commands such as Print Screen cannot be disabled [1].

Microsoft Right Management Services (RMS) is an example of an operating system level DRM. Implementing DRM at this level provides better security since operating system commands can be intercepted. An operating system level DRM such as RMS can also support a wider range of media since it is not geared toward a particular type of application. However, RMS is not transparent at the application level. Support for RMS must be built into applications requiring DRM support using a software development kit for RMS. This allows the application to interact with a client side DRM controller and a server module. The controller module is responsible for ensuring that only legal transactions take place while the server module is responsible for administering DRM enabled content. This has been discussed in a paper by Arnab and Hutchison's [1]. Since any content format that is RMS enabled can be read using an appropriate application, RMS achieves many of the benefits of an operating system level DRM. Nonetheless, it does not free the application level from having to deal with the underlying DRM.

Operating system level implementations may degrade the overall performance of a system. Rosenblatt [4] stated 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.

Another alternative is to implement DRM at the hardware level. Rosenblatt [4] believes that a hardware controller is ultimately the best choice for a DRM implementation. However, if the underlying hardware implementation is insecure, a breach of security is harder to address. An example of this problem is the rights protection build into DVD's. In this case, a flawed encryption algorithm resulted in a security breach [1]. Because the DRM was built into hardware, which could not be upgraded, it was not possible to fix the problem. At the present time, there are a number of problems that must be addressed before DRM can be properly implemented in hardware.

To summarise, a DRM controller at the application-level is relatively easy to implement, but offers the least security. The DRM protection can easily be bypassed by simply modifying the executable application code. A hardware-level implementation, on the other hand, offers the best security. However, this could lead to increased computer hardware costs, and may be expensive and difficult to upgrade. A DRM controller at the operating system level thus seems like a good balance between monetary expense and level of security. However, Rosenblatt's concerns about system performance may be valid. It is therefore to necessary investigate such performance issues in order to determine whether operating system DRM is indeed a viable option.

3. SYSTEM DESIGN AND IMPLEMENTATION

3.1 System Overview

The prototype DRM controller was developed on 2.6 Linux-based operating system. It consists of two core modules: an operating system kernel module and a user-space daemon module. The daemon module is responsible for the management and retrieval of DRM use licenses from the DRM content distributors' remote license servers, while kernel module enforces the access control rules specified in these use licenses.

Figure 1 gives an overall view of the DRM system architecture. The following steps describe the interaction between the kernel module and the daemon:

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

Step B: The application requests access to the file. The kernel module intercepts this request.

Step C: The kernel module sends a request for 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.

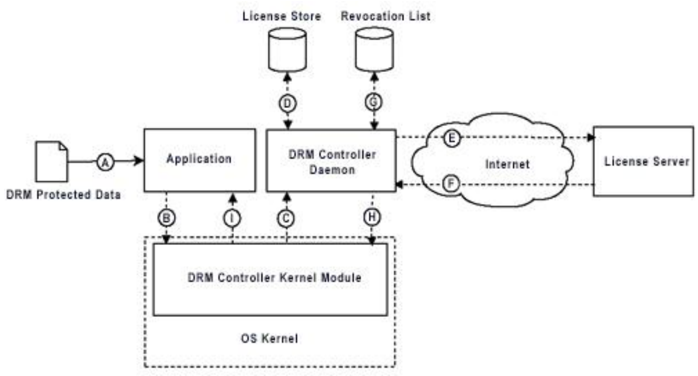

Figure 1: Figure 1. The DRM Controller Architecture and Communications

Step F: If a license is successfully negotiated, the daemon proceeds to step G. Otherwise, a message is sent to the kernel module 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 returns the license to the kernel module in a simplified (non-XML), common format. This will contain all relevant information form the original license.

Step I: The kernel module 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.

3.2 The Daemon

The daemon module implements several components. The responsibilities of the daemon module are described below.

3.2.1 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 manages these licenses, having full rights to add, revoke or modify license content.

3.2.2 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.

3.3 The Kernel Module

The kernel module contains two components: the Access Enforcement Component (AEC) and the Access Decision Component (ADC). Together these two components act as a reference monitor. All access requests to DRM protected content must pass through these two components. The ADC makes *decisions* on whether to allow access to the DRM content or not, while the AEC is responsible for *enforcing* the decisions made by the ADC.

In order to enforce digital rights, the AEC defines a set system calls which replace the some of the original system calls in the kernel. Whenever a request is made, the AEC's system calls are called instead of the original system calls. Within each of its own system calls, the AEC, checks with the ADC if access can be granted, and depending on the response from the ADC, access is either granted or denied. In order to deny access, the AEC simply returns an error message to the application requesting access. In order to grant access, the AEC calls the original system call to serve the access request.

4. PERFORMANCE ANALYSIS

This section describes the experiment that was conducted to determine the performance cost imposed on the system by the DRM controller. The aim in this experiment was to prove that hypothesis 2, discussed in section 1, is correct.

4.1 Experiment

4.1.1 Method

The following two system calls were used during this experiment:

1. read(), and
2. 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.

1. 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.
2. We then determined the system call overhead of the two system calls when the DRM controller kernel module was enabled. As in test 1, all the files used in this test were non-DRM protected.
3. Finally, we repeated test 2, but this time, we used files which were DRM protected.

Two sets of files, both consisting of six files each, were used. The files in the one set were DRM protected, while those in the other set were regular non-DRM protected files. Each set was comprised of the files with the following file sizes:

1. 1KB,

2. 32KB,
3. 128KB,
4. 1MB,
5. 32MB, and
6. 128MB.

For each test-run, the system calls were invoked 10 times per file. The duration of the system call was then determined by taking the average of the 10 measurements. These results were tabulated and are presented in tables 1 and 2.

4.1.2 Test Environment

The experiment was conducted on an Intel Celeron desktop computer with a 1.7GHz CPU clock speed and 512 MB of RAM. We used a standard 2.6.13.4 Linux kernel compiled with preemptive support.

4.2 Results

Tables 1 and 2 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 (KB)	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 (%)
1	39.237	47.902	4939.474	22.084	12488.817
32	80.437	86.257	6557.291	7.235	8052.083
128	245.132	248.649	5113.205	1.435	1985.899
1024	1602.570	1581.591	6587.491	1.309	311.058
32768	47366.808	46725.513	57306.870	1.354	20.985
131072	188026.205	185969.232	214434.032	1.094	14.045

Table 1: 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 (KB)	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 (%)
1	15.0425	25.337	6804.710	68.436	45136.563
32	12.868	22.066	6272.514	71.480	48645.058
128	11.301	19.802	6561.940	75.223	57965.127
1024	11.238	21.228	6486.350	88.895	57618.010
32768	11.031	21.392	7052.769	93.926	63835.899
131072	11.115	23.912	7422.106	115.133	66675.583

Table 2: 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.

4.3 Analysis

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

1. Intercepting the access request and the 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 1 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 cost is 22% for a 1K file and decreases until it reaches near 1% performance costs for a 128MB 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. If we had measured access times for file sizes beyond 128MB, we are certain that overhead would decrease even further.

Looking at table 2, 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 experimental error and varying system load. We also notice that performance overhead increases as the file size increases. However, this is due to experimental error as well.

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 12489% increase in performance cost. This high cost increase is due to the large overhead involved when communicating 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 14% for a 128 MB file.

Table 2 again shows similar access times when accessing DRM-protected content, regardless of the size of the file. This time, the performance cost ranges between 45136.6% to 66675.6%, which is very high, compared to the performance cost imposed when accessing non-DRM protected data in a DRM-enabled system. the large overhead remains, even on large files.

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.

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.

5. THE EFFECTIVENESS OF ENFORCING RIGHTS IN AN OPERATING SYSTEM LEVEL DRM

The current system has been built in order to test the feasibility of an operating system level DRM controller. It does not seek to provide a complete solution to the problem. It has been considered sufficient to implement only the core functionality of the system. There are a number of problems which have been encountered that fall outside the scope of the current implementation. This section describes the issues would need to be addressed in a complete system implementation. We conclude this section by re-examining our first hypothesis, described in section 1.

5.1 Interpreting Rights Expressions

Some rights are harder to enforce since they cannot easily be identified at the Kernel level. For example, the ODRL language allows a limitation to be placed upon the number of pages that an end user may print. Within the Kernel there is no concept of a page. Pages only exist within applications such as word processors that need to support such an entity.

5.2 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 Kernel 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.

5.3 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.

5.4 General Implications of Operating System level DRM

Our first hypothesis states that application level transparency can be achieved. The initial prototype has demonstrated that a basic mechanism can be put in place to handle access attempts at the Kernel level. Furthermore, application level support was not required. However, there are still a number of problems that must be solved before access control can be successfully enforced at the Kernel level. Simple file formats and applications can be supported with relatively little work. However other applications behave in a more complex manner. This requires the presence of additional logic within the Kernel Module.

6. CONCLUSIONS

In this paper, we presented a prototype DRM controller, which enforces rights to digital content at the operating system level of a computer. We defined the following hypotheses in order to assess the viability of such a controller in terms of performance and how effective it is at enforcing rights to digital content.

1. The prototype DRM controller can enforce a wide range of digital rights, transparently to any end-user applications trying to access the DRM protected content.
2. The performance cost imposed on the system by the prototype DRM controller is negligible.

With regard to the first hypothesis, the current prototype does achieve transparent enforcement of access control rules for multiple file formats. This proves the hypothesis to be correct. However further work would be required in a full implementation in order to ensure that all methods of files access are handled correctly.

With regard to the second hypothesis, we found that DRM Controller incurred performance costs between 1% and 116% for unprotected data. On a single-user system where accesses to DRM content is relatively infrequent, this cost is sufficiently small for a user not to notice the overhead. However, it is still too high for multi-user systems with heavy load. On the other hand, the performance cost incurred when accessing DRM protected data were sometimes as high as 12489%. This is far beyond what we consider acceptable. Therefore, the second hypothesis is incorrect. However, considering that our prototype was not an optimised implementation, we do consider these performance measurements as promising.

7. REFERENCES

[1] A. Arnab and A. Hutchison. Digital Rights Management - An Overview of Current Challenges and Solutions. In *Proceedings Information Security South Africa (ISSA)*, Midrand, South Africa, 2004.

[2] A. Arnab and A. Hutchison. Requirement Analysis of Enterprise DRM Systems. In *Proceedings nformation Security South Africa (ISSA)*, 2005.

[3] W. W. Fisher and U. Gasser. iTunes: How Copyright, Contract and Technology Shape The Business of Digital Media., 2004.

[4] B. Rosenblatt. DRM for the Enterprise. Jupiter Webinar, Jupiter Media Corporation, Inc., February 2004.

[5] B. Rosenblatt and G. Dykstra. 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.