

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/315081462>


Digital Rights Management for Streaming Media

Chapter · January 2009
DOI: 10.4018/978-1-60566-262-6.ch002

CITATIONS
0

READS
41

4 authors, including:




Shiguo Lian

CloudMinds

244 PUBLICATIONS 3,773 CITATIONS

SEE PROFILE



Mark Stamp

San Jose State University

156 PUBLICATIONS 2,236 CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

- Project

Multimedia Content Indexing and Retrieval [View project](#)
- Project

neural networks, chaos, soft computing and their applications [View project](#)

Chapter II

Digital Rights Management for Streaming Media

Deepali Brahmbhatt

San Jose State University, USA

Mark Stamp

San Jose State University, USA

ABSTRACT

This chapter presents a digital rights management (DRM) system designed for streaming media. A brief, general introduction to DRM is also provided, along with a discussion of the some specific issues that arise in the context of streaming media. The DRM system proposed here has been implemented and some details related to the implementation are discussed.

INTRODUCTION

For the purposes of this paper, digital rights management (DRM) can be viewed as an attempt to provide “remote control” over digital content. That is, DRM is supposed to make it possible to securely deliver digital content and to restrict the actions of the recipient after the data has been delivered.

Consider, for example, a digital book. Certainly, the publisher would like to deliver such a book over the Internet, since there is an enormous

potential market and the costs of reproduction and delivery are negligible.

However, if an attacker can redistribute a perfect digital copy of the book, then the rate of piracy would almost certainly be intolerable—if not virtually 100%. An ideal DRM system (from the perspective of the copyright holder) would prevent the redistribution of the book in an unprotected form, and perhaps also enforce other restrictions, such as “no printing.” Such an ideal DRM system is impossible, as noted in the indispensable paper by Biddle, et. al. (2002) and

in Stamp (2002). The interesting question then is, what is the best practical DRM system? What is “best” depends on many factors, not all of which are strictly technical in nature. For example, the value of the content being protected, the technical sophistication of typical users, the overall business model, and the credibility of threats of legal reprisals greatly effect the utility of any DRM system; see Stamp (2003b) for a discussion of these and other non-technical DRM issues.

It is important to note a few salient issues concerning DRM. First, the fundamental requirement of a DRM system is that restrictions must be enforced after the content is delivered to the intended recipient. Since these restrictions must stay with the data wherever it goes, the buzzword for this DRM requirement is “persistent protection.” This is in contrast to the usual cryptographic scenario, where the goal is simply to securely deliver the bits.

Second, the legitimate recipient is a potential attacker. This is also in stark contrast to the usual cryptographic situation, where the intended recipient (i.e., the entity that holds the corresponding decryption key) is a “good guy”, not an attacker.

Third, cryptography is necessary, but far from sufficient for useful DRM. It is necessary to encrypt digital content in order to securely deliver the bits, and this is a part of any DRM system. However, securely delivering the bits is the easy part of DRM.

Cryptography alone is insufficient for effective DRM—to render the content, the recipient must access to the key, and the recipient is a potential attacker. Cryptography was not designed to solve this problem, where, in effect, we must give the attacker the key. The essence of DRM security can therefore be reduced to playing “hide and seek” with cryptographic keys, although this fundamental fact is not always clear from the descriptions of fielded or proposed DRM systems.

It has been shown that it is not trivial to effectively hide a key in data (Shamir and van

Someren 1999) and that software obfuscation techniques are not sufficient to hide a key in software (Jacob, Boneh, and Felten 2003). In any event, this game of hide and seek is at the core of any DRM system. This topic is explored further in the next section.

Finally, it is important to note that there is an absolute limit on the utility of any DRM system. The digital content must ultimately be rendered, and at that point it is subject to capture in analog form. Even if a perfect DRM system were available, digital music, for example, could be recorded using a microphone when it is rendered. In our digital book example, an attacker could take digital snapshots of the pages of the book when it is displayed on a computer screen. This is the so-called “analog hole” (Doctorow 2002), which is obviously present in any DRM system. These sorts of analog attacks are beyond the scope of the DRM system. But since such attacks likely result in a loss of fidelity as compared to the original digital content, they are, perhaps, not as serious of a concern as a successful attack on the DRM system itself. Consequently, the goal of a DRM system is to prevent the attacker from obtaining an unprotected and high quality digital copy of the original.

In the next section we give a brief description of some of the techniques commonly employed in DRM systems. Our focus is on software-based systems, but we do mention some of the inherent advantages and disadvantages of hardware-based protection. Then we turn our attention to our proposed software-based DRM system for streaming media. A prototype of this proposed system has been implemented and we briefly discuss some details of the implementation. We then consider likely attacks on our proposed system as well as open problems in the field of DRM.

DRM TECHNIQUES

As mentioned in the previous section, the fundamental DRM problem is the protection of a

secret, which, typically, is a cryptographic key or the equivalent thereof. The difficulty lies in the fact that the secret must be protected from the legitimate recipient, who, ultimately, must have access to the secret in order to render the digital content.

When DRM is implemented in software, the software must somehow protect itself from attack. Since the attacker is likely to have full administrative privilege, it is nontrivial to provide any meaningful software self-defense mechanisms.

Consider DRM client software executing under an attacker's control. Of course, we would like to make the attacker's job as difficult as possible. Ideally, we would like to have the challenge confronting the attacker be comparable to, say, breaking a secure cryptosystem. But, even in principle, we cannot hide a secret from a user with full administrative privileges. Furthermore, fielded DRM systems have failed so miserably and so consistently that there is now overwhelming empirical evidence that even a more modest "practical" level of protection is extremely difficult to achieve (CDFreaks 2002, Guignard 2002, "Beale Screamer" 2001, Ackerman 2003).

Today it is clearly the case that no software protection technique—or combination of such techniques—can make the attacker's job sufficiently difficult to ensure that the DRM client software cannot be broken. While unconditionally secure software-based DRM protection is unobtainable, there are techniques available that can make the attacker's job far more challenging. But since a persistent attacker will ultimately win, it may appear that software-based DRM is hopeless. Next, we argue that this is not the case, that is, software-based DRM can be of practical benefit in certain situations.

It is important to distinguish between an individual instance of DRM client software and the DRM system as a whole. We need to make it sufficiently challenging for an attacker to break a particular instance of the DRM software. We also need to be cognizant of the tradeoffs between

security, usability, development, maintenance, complexity, etc. In any case, we must recognize that a persistent and dedicated attacker can, with sufficient effort, break an instance of the DRM client software. With this in mind, our goal will be to design a DRM system that remains useful even after an instance (or several instances) of the client software has (have) been broken. That is, the system as a whole should degrade gracefully under attack, instead of collapsing at the first successful attack on an instance of the client software. This system-wide DRM security is often known as "break once, break everywhere" resistance, or BOBE-resistance (Biddle, et. al., 2002).

Next, we discuss various techniques that can be used to "harden" each instance of DRM client software. Then we discuss some of the techniques that can be used to make a system more BOBE-resistant. Finally, we consider the potential benefits of hardware-assisted DRM, that is, we consider the benefits of using tamper resistant hardware as part of a DRM system.

Software Self-Defense

Software-based DRM systems can employ a wide variety of software self-defense techniques. Most of the techniques mentioned here are discussed in more detail in (Stamp 2003a); an overview of these topics can be found in (Stamp 2005).

From the DRM perspective, the fundamental weakness of software is that it is susceptible to reverse engineering. That is, even if we only provide the executable code (i.e., no source code), an attacker can analyze the software in great detail; see the excellent book (Eilam 2005) for more information on software reverse engineering. Not surprisingly, DRM protection mechanisms are squarely aimed at making reverse engineering more difficult; see (Cerven, 2002) for a discussion of several anti-reverse engineering techniques.

The essential tools of the software reverse engineer are a disassembler and a debugger. The disassembler provides a static view of the code,

while a debugger provides a dynamic view. Various anti-disassembly techniques are known. For example, encrypting the executable code makes it impossible to disassemble. However, the code must be decrypted before it can be executed and an attacker may simply let the code decrypt itself before attempting to analyze it or the attacker may be able to reverse engineer the decryption routine and use it to decrypt the code.

Various “false disassembly” techniques are also used to confuse a disassembler.

Debuggers can also be confused by carefully constructed code. But, as with anti-disassembly techniques, a patient and skilled attacker can eventually remove these roadblocks.

Some fairly intricate combinations of these methods are possible. For example, Stamp (2003a) describes a method where an anti-disassembly method and an anti-debugging technique are combined in such a way that each protects the other to some extent.

Eventually, the persistent attacker will disassemble the code and run a debugger on the interesting parts of it. At this point we would like to make the code as difficult as possible to understand, thereby further delaying the attacker. Collberg, Thomborson, and Low (1997) have developed sophisticated methods of software obfuscation, which make the software difficult for an attacker to comprehend; see Eilam (2005) for a discussion of the effectiveness of such methods from a reverse engineer’s perspective. The use of such obfuscation techniques is standard practice in “serious” DRM systems.

The use of proprietary algorithms is also sometimes suggested as a way to make reverse engineering more difficult (Stamp 2003a). Standard cryptographic algorithms, for example, are easy to detect and once detected, they do not need to be reverse engineered—only the key is required. On the other hand, non-standard cryptographic algorithms need to be analyzed, modified, or reverse engineered, thereby increasing the work factor for an attacker. However, Kerckhoffs’ Prin-

ciple states that we can only trust cryptographic algorithms that have been publicly scrutinized by experts and such algorithms are, inevitably, well known. One solution—as suggested in Stamp (2003a)—is to use both a standard, secure cryptographic algorithm and a proprietary “scrambling” algorithm. This does not violate Kerckhoffs’ Principle and it should significantly increase the work for an attacker.

Another software protection technique that has been widely discussed in the literature is the use of so-called guards (Chang and Atallah, 2001; Horne, et. al., 2001). That is, parts of the code (the guards) tamper-check other parts of the code. Note that by itself this technique does not prevent an attacker from analyzing the code. Instead, guards are designed to prevent the attacker from being able to modify, or “patch” the code so that it performs a different function than the developer intended. In effect, this technique is designed to make the code fragile, which is useful for limiting attacks that make changes to the code, such as an attack that bypasses authentication.

Of course, combinations of most of these methods can be used. For example, the DRM system discussed in Stamp (2003a) uses virtually all of these techniques to some degree.

BOBE-Resistance

The goal of BOBE-resistance is to prevent one successful attack on client DRM software from breaking the entire DRM system. One step in the direction of BOBE-resistance is to “personalize” each copy of the DRM client software. That is, each instance of the software has some user-specific information embedded in it. An attacker must then determine this user-specific information in order to break a specific copy of the DRM client software.

However, personalization in this form is often very weak. For example, it may consist of nothing more than embedding a user-specific key in the software. This level of defense is unlikely

to prevent an attacker from parlaying a single successful attack on a single DRM client into a system breaking attack.

For a more robust form of BOBE-resistance we turn to virus writers. Virus and worm developers have recently become infatuated with unique, or metamorphic software. In the standard software development model, each instance of a particular piece of software is identical, that is, the software is cloned from a single master copy. Virus and worm writers have recently begun to develop metamorphic software, where each copy is functionally equivalent, but the internal structure is significantly different (Stamp 2005). Malware writers are interested in metamorphism as a way to foil signature-based detection tools. In the field of DRM, metamorphism can be used so that each copy of DRM client software has a different internal structure. As discussed in (Stamp 2004), the potential benefit of such an approach is that a successful attack on one instance of the software will not necessarily succeed on any other instance of the software. In the best case, the work factor to attack N instances of the software would be N times greater than the work required to attack one instance. This would appear to be optimal with respect to BOBE-resistance.

Other approaches designed to achieve something similar to BOBE-resistance have been suggested. For example, it is possible to embed multiple security features in the software with many of these features initially dormant. Then if the DRM protection is compromised, one of the inactive security features can be activated, essentially creating a new protection scheme “on the fly.” Or there may be other ways to update the software in real time. We mention one such method in the discussion of our proposed system, below.

Hardware-Assisted DRM

The discussion so far has focused on software-based protection. The fundamental problem with

any such approach is that we cannot securely hide a secret in software. On the other hand, a secret hidden in tamper resistant hardware will almost certainly present a more significant challenge to an attacker (Anderson, 2001). The benefits of a “hardware solution” can be seen in game consoles, which do a relatively good job of protecting their secrets, namely, the copyrighted software running on the system.

The potential advantages of hardware-based protection have not been lost on DRM advocates. The Trusted Computing Group (2007) is a consortium of manufacturers that produce computer chips containing special-purpose hardware that can store a secret, such as a key. This key is not directly available to a user, even if the user has full administrative privileges.

In coordination with the TCG hardware, Microsoft is developing a trusted operating system known as the Next Generation Secure Computing Base (2007). Although TCG/NGSCB is marketed today as a general-purpose security enhancing technology, the original motivation was DRM. The concern within the personal computing industry is that without a relatively strong form of DRM, the PC will lose out to closed systems, particularly in the realm of audio and video. In effect, TCG/NGSCB attempts to provide the security benefits of closed systems (e.g., game consoles) while maintaining the multitude of benefits that derive from an open platform (i.e., PCs). The details of TCG/NGSCB are beyond the scope of this paper; see Stamp (2005) for a high-level description.

DRM FOR STREAMING MEDIA

With the extensive background above, we are now ready to discuss a specific DRM system for streaming media. This system is software-based and is designed to provide simple but reasonably robust DRM protection with strong BOBE-resis-

tance. Some aspects of this proposed system are outlined in Holankar and Stamp (2004).

First, we describe a generic streaming media application and the potential attacks on such a system. Then we describe our proposed DRM solution and we discuss the protection it provides against various attacks.

Streaming Media System

The following is a generic list of components and their functions in a (non-DRM) streaming media system; see Crowcroft, Handley and Wake-man (1999) for more information on streaming media.

1. A web server to stream the data to clients.
2. An authentication protocol to operate between the web server and clients.
3. A client side web browser, used to request the media.
4. A client application, such as Real Player Windows Media Player, to receive and render the media data.
5. A client side interface between the device driver (in kernel space) and the application (in user space).
6. A client device driver to utilize the media data.

Such an application can employ the Real-time Transport Protocol (RTP), which is a standard protocol to transport a media stream between two endpoints (RTP 2007). Secure RTP is a standard that makes use of cryptographic techniques to secure the RTP stream (Baughner 2004, Vovida 2004).

A generic streaming media system could function in the following manner.

1. The web server offers access to data.
2. A client requests a media file from the web server.

3. The web server and the user authenticate each other (mutual authentication).
4. After successful mutual authentication, the web server employs RTP to stream the data from the server to the client.
5. The client web browser opens the appropriate application (e.g., Real Player).
6. The media application removes the RTP headers and, if necessary, sequences the packets.
7. The media application uses system calls or library functions to write the data to the device that renders the data.
8. The device driver stores the data in its internal memory, writing the data to the appropriate port.

The above system has certain inherent security vulnerabilities. For example, the streamed data can be captured at any point between the two endpoints and the resulting data is subject to replay. From a DRM perspective, a more significant issue is the lack of any defense against client-side capture and subsequent redistribution of the data in an unprotected form. Below, we describe a DRM system that provides some measure of protection against the threat of data capture and redistribution.

DRM System

Our proposed security model augments the basic streaming media system described above. The two crucial DRM security components in our system are a proprietary device driver and the use of “scrambling” algorithms. Together, these features make reverse engineering any instance of the software more challenging, while also providing a reasonable degree of BOBE-resistance.

The server employs a scrambling algorithm and the client employs the corresponding de-scrambling algorithm. A scrambling algorithm should be unknown to a potential attacker. Ideally, we would like to force the attacker to “break” (by

reverse engineering or other means) the scrambling algorithm in order to recover any of the unprotected data. The server must have access to a large number of distinct scrambling algorithms and the client must have access to the appropriate de-scrambling algorithm.

Scrambling serves at least two purposes. First, the scrambling algorithm creates a layer of obfuscation, making reverse engineering of the client software more difficult. Second, scrambling provides for a significant degree of metamorphism at a critical location in the client software.

Perhaps the ideal scrambling algorithm is a secure cryptographic algorithm, since the algorithm could be applied to all of the data and we could be confident that there is no simple way to de-scramble the data without access to the key and knowledge of the algorithm. However, Kerckoffs' Principle rules this out. But we will not depend on the scrambling algorithm for cryptographic strength, since we also employ standard strong encryption. Therefore, a homemade cryptographic algorithm that provides minimal cryptographic strength will serve as a scrambling algorithm.

For our prototype implementation, we employed modified forms of the Tiny Encryption Algorithm (TEA); see Wheeler and Needham (1994). From this algorithm we derived a large class of scrambling algorithms. Without extensive analysis, none of these modifications could be claimed to provide substantial cryptographic strength, but each is suitable for use as a scrambling algorithm. The rationale behind scrambling is further discussed (within the context of DRM) in Stamp (2003a).

Given a set of scrambling algorithms, each client is equipped with a subset of the available algorithms and these algorithms are compiled into the client executable. The list of scrambling algorithms known to the client is encrypted with a key known only to the server, and this client-specific encrypted list is stored on the client. After authenticating the server, this encrypted list is passed from the client to the server.

When the server receives the list, the server decrypts it and randomly chooses from among the client's known scrambling algorithms. The client's identifier of the selected scrambling algorithm is then sent from the server to the client. Note that this process eliminates the need for a database containing the mappings between clients and scrambling algorithms. This approach is analogous to that used in Kerberos to manage "ticket granting tickets" (Stamp 2005).

By having different scrambling algorithms embedded within different clients, and by randomly selecting from a client's list of algorithms, each client is unique, and each communication between client and server depends not only on different keys, but also on different algorithms. An attacker who is able to break one particular piece of content, will likely still have a challenging task when trying to break another piece of content destined for the same client. And even if an attacker completely reverse-engineers one client, it is likely that a comparable amount of effort will be required to attack any other client.

On the server side, the data is scrambled then encrypted. On the client side, the data is decrypted, and the resulting scrambled data is passed to the media application. The media application passes the scrambled data to the proprietary device driver, which de-scrambles the data. In this way, the data is obfuscated until the last possible point in the process.

Given these security features, our DRM streaming media system functions as follows.

1. The secure web server offers streaming media services.
2. A client requests a media file from the secure web server.
3. The secure web server authenticates the user and the user authenticates the web server.
4. Upon successful mutual authentication, the client sends its encrypted list of supported scrambling algorithms to the server.

5. The server generates two random keys. The first key is for secure RTP packet encryption (using a strong encryption algorithm) and the second is the scrambling key.
6. The server randomly selects from among the scrambling algorithms supported by the client. The scrambling algorithm selection and the keys are encrypted (but not scrambled) and passed to the client. The client acknowledges receipt of this information.
7. The server scrambles the data using the appropriate key using cipher block chaining (CBC) mode (per packet), with a randomly selected initialization vector (IV). The IV is included within each packet (for cryptographic terminology, see (Schneier 1996, Stallings 2003)).
8. The secure RTP algorithm uses a secure encryption algorithm, such as the Advanced Encryption Standard (AES), and the appropriate key to encrypt the scrambled data in each packet. The packets are CBC encrypted with a randomly selected IV.
9. The scrambled and encrypted secure RTP packets are streamed from the server to the client.
10. The client web-browser opens the appropriate secure media application.
11. The media application requests the secure RTP decryption key. The user must authenticate in order for the client software to obtain access to the decryption key.
12. The media application strips off the secure RTP headers and sequences the packets, if necessary.
13. The media application initializes its secure device driver with the appropriate scrambling algorithm, and the algorithm is initialized with the scrambling key.
14. The media application is oblivious to the scrambling. It writes the scrambled data to the device that plays the file.

15. The secure device driver de-scrambles the data and writes the resulting plaintext to the appropriate device buffer and port.

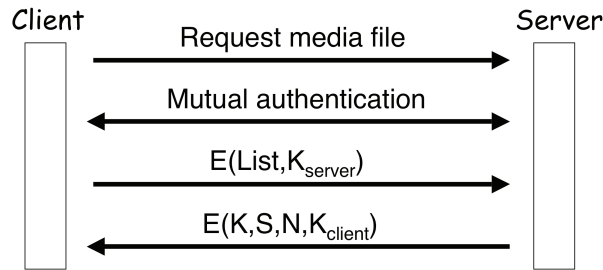
Figure 1 gives a simplified view of the interaction between the client and server, where

1. “List” is the list of scrambling algorithms that the client supports.
2. K_{server} is a key known only to the server.
3. $E(\text{List}, K_{\text{server}})$ is the list of scrambling algorithms that the client supports, encrypted with the key K_{server} .
4. K is the cryptographic key that the server will use to encrypt the data.
5. S is the scrambling key that the server will use to scramble the data.
6. N is the identifier of the client’s scrambling algorithm that the server has selected.
7. K_{client} is a key that is known to both the client and the server. This key could be shared in advance (and used for authentication) or it could be negotiated during the authentication process (in which case another method would be needed for authentication).
8. $E(K, S, N, K_{\text{client}})$ is the cryptographic key K , the scrambling key S , and the scrambling algorithm identifier N , all encrypted with the key K_{client} .

Comparison with Windows Media Player

Next, we give a brief comparison of the security features in our proposed streaming media system—as presented above—with the features available in Windows Media Player (Windows DRM 2004). The six points listed below are given on Microsoft’s website (Windows DRM 2004) as the primary security features of Windows Media Player. We describe how our system implements each of these features.

Figure 1. Simplified client server interaction



1. Persistent Protection: The protection in our system extends over the insecure network between client and server. Due to the scrambling algorithm and the secure device driver, the protection persists all the way to the client's media device.
2. Strong Encryption: Our system uses RTP with strong encryption. Our model also employs scrambling, but it does not rely on scrambling for cryptographic security.
3. Individualization: In our system, the scrambling algorithm is selected at random and the actual set of available scrambling algorithms is individual to each client. Therefore, the compromise of one client does not break the entire system. Moreover, a compromised client can easily be replaced with an upgraded device driver that employs a different set of scrambling algorithms. The secure device driver could easily be made unique in other ways as well. For example, the methods discussed in (Mishra and Stamp 2003) or methods similar to those employed by metamorphic virus writers (Balepin, 2003) could be implemented. Such protections would make the reverse engineering problem even more challenging for an attacker. We have not implemented this higher level of metamorphism, but it would certainly be feasible to do so.
4. Secure Media Path: Our system does not de-scramble the data until the last possible point in the process. The data passes through the entire system in scrambled form. When it is sent over the (insecure) network, the data is further protected by strong encryption.
5. Revocation and Renewability: In our approach, compromised players can be revoked and revoked clients will fail to authenticate. Moreover, if a particular scrambling algorithm has been compromised, the server can simply avoid using the compromised algorithm. Alternatively, all clients using the compromised algorithm can be upgraded with new device drivers that do not include the particular algorithm.
6. Secure End-to-End Streaming and Downloads: We employ secure RTP for end-to-end streaming and downloads. A strong encryption algorithm is used in secure RTP. It is worth noting that secure end-to-end transmission can also be accomplished using other well-known methods such as IPSec (Kaufman, Perlman, and Speciner 2002).

As can be seen from the discussion above, the secure device driver and the software metamorphism achieved via scrambling are the crucial security aspects of our proposed DRM system. We have developed a prototype system (in Linux) that implements the system described above. Our implementation is briefly discussed in Section 5.

ATTACKS

The obvious attacks on our proposed system include the analog hole and reverse engineering the software. Since the analog hole is beyond the scope of any DRM system, we do not consider it further here. To make the reverse engineering attacks more challenging, we could distribute the proprietary device drivers immediately before distributing the content. This would be particularly valuable for data that is somewhat “perishable”, such as a live broadcast. In this way, we could limit the time available to the attacker. This could prove to be a very effective approach in certain situations.

It would also be fairly straightforward for an attacker to modify the hardware on the client so that the plaintext bits are captured directly, without attacking the software. Although a serious threat, such modifications are also beyond the scope of a software-based DRM system.

IMPLEMENTATION

This section describes in some detail various implementation issues of our secure streaming media system.

Operating System Issues

The preferred operating systems for embedded multimedia devices are Real Time-Linux, Linux and VxWorks (Yodaiken 2000). Here, we discuss implementation issues of our proposed secure streaming media model related to these three operating systems.

VxWorks does not have any memory protection between application and system tasks. This makes the device driver memory buffer in the proposed model available to any module through a simple function calls. Moreover, decryption within the device driver adds overhead, making the process not as lightweight as we would like.

On a more positive note, if the decrypted media data is made available in the memory buffer only for periodic time slices, it makes it difficult for other hacker processes to contend for the decrypted data in the same time slice. The hacker process would need to synchronize with the availability of the decrypted data. Such synchronization would be difficult in a single processor system, and it would be many times more difficult in a multiprocessor system.

RT-Linux requires the user to divide the application into two distinct parts: the real-time part and the non-real-time part. The real-time part is serviced rapidly, allowing it to meet deadlines, while the non-real-time part has the full range of Linux resources available for use, but cannot have real-time requirements. The division of multimedia data into real-time and non-real-time part must be managed carefully. The constraints increase if the media data is streamed live or it is interactive. Decryption of media data on the device driver level further increases the timing constraints of live and interactive data.

Under the Linux operating system, the implementation issues are greatly simplified. This is primarily due to the fact that Linux provides memory protection between kernel processes and user space. Linux also has non-swappable memory, which provides some protection to key material.

Open Source Resources

The following specific open source resources were utilized in the implementation of the proposed security model

- OpenSSL (OpenSSL 2007)
- Tiny httpd (Tiny httpd 2007)
- Secure RTP (Secure RTP 2007)
- Linux (Linux 2007)
- Linux device drivers tutorials (Calbet 2006)

- Intel audio driver i810_audio (Audiogdriver 2007)
- Crystal audio driver cs46xx (Audiogdriver 2007)

Distinct Scrambling Algorithms

Our approach to BOBE-resistance requires that we have available a large selection of distinct “scrambling” algorithms. As previously noted, our scrambling algorithms consist of unique variations of the Tiny Encryption Algorithm (Wheeler and Needham 1994). Our proof-of-concept implementation supports sixteen scrambling algorithms on the server side. Of course, a far larger number of scrambling algorithms could be employed.

The scrambling and de-scrambling code is compiled as separate object code, which can be easily linked with different sender and receiver programs. In practice, it would be easy to generate multiple receivers supporting different scrambling algorithms. For demonstration purposes, we have created three receivers wherein one is totally secure, one receiver is partially broken and another one is totally broken. These receivers demonstrate the functionality of the server negotiation process in the case where some of the available scrambling algorithms have been hacked.

Hardware Requirements

To implement and test the security model proposed in this paper, the minimal configuration consists of the following:

- Two personal computers, both running Linux kernel 2.4.16 or higher, and both having sound cards and network adapters.
- Multimedia support must be enabled on the Linux operating system and sound drivers must be configured in modular mode.
- The specific sound card device driver must be available in open source.

The personal computers can be connected via a network or crossover cable. Of course, a more sophisticated network connection could be used.

It should be noted that the performance results vary depending on the available processing power as well as the network card performance at both the sender and receiver.

Implementation in Linux

This section briefly describes the implementation details in Linux using kernel 2.4.16

Server-Side Components

An HTTPS web server must be installed on the server. We employed the tiny httpd server (Tiny httpd 2000) and Open SSL (Open SSL 2000) was installed on the server side to support HTTPS. The server invokes cgi-scripts to start the secure RTP streams. The function of the cgi-script is to obtain the environment settings of the http username, the http client IP address, to locate the requested data and to invoke the sender program.

The sender program takes the following parameters

- Destination IP address
- Destination port
- Filename
- Username
- Sampling rate

The message sent by the server to the receiver includes the following parameters:

- Session key
- Server IP address
- Server port
- Scrambling key
- Sampling rate of audio file
- Total number of packets
- License Manager

As outlined above, the receiver sends an encrypted list of supported scrambling algorithms. The server randomly selects one of the supported algorithms, maps the client algorithm to its corresponding server algorithm, and sends the selected algorithm number to the receiver. The server then begins sending the streaming data (which is scrambled and encrypted) in packets of a predefined size.

The license manager maintains a list of multimedia data files and corresponding usernames and the number of times a user is permitted to invoke each file. On each invocation, the license manager decrements by one the allowed number for that particular user for the accessed file. However, if the user has no access restriction to the particular file, then the license manager will not decrement the access count. In practice this logic could be easily implemented using a secure database system.

The license manager also maintains a list of broken (i.e., hacked) scrambling algorithms. If the license manager detects that all of the supported algorithms at the receiver end are broken, it will ask the server to terminate its connection with the receiver without giving any explanation to the receiver.

Receiver-Side Components

The receiver side listens to a predefined secure RTP port. The server program sends a session key encrypted using the receiver's shared symmetric key. When the receiver receives the session key, it sends its encrypted list of supported scrambling algorithms to the sender. The sender program chooses one of the scrambling algorithms and sends the multimedia data packets. Note that the receiver application works in close communication with the receiver device driver. The receiver initializes the secure device driver using the selected scrambling algorithm and the scrambling key sent by the server. When the re-

ceiver is compiled, a private key for the receiver is built into the receiver executable file.

The receiver has an encrypted list of its supported scrambling algorithms, which the sender decrypts to determine the scrambling algorithms supported by the device driver at the receiver end. If all the supported scrambling algorithms of the device driver are found in the broken list maintained by the server, the device driver at the receiver end is considered broken. When the server receives a request from a broken receiver, it immediately terminates the connection without explanation.

Secure Device Driver

The receiver application talks directly to the secure device driver and the secure device driver in turn talks directly to the device. The Linux `artsd` daemon, which is used to monitor access to sound, is killed to allow direct access to the sound device. The secure device driver includes the de-scrambling algorithms and it also includes a modified write function. This modified write function enables the driver to de-scramble data before writing to the Direct Memory Access (DMA) buffer. The device reads the de-scrambled data from the DMA buffer directly.

All Linux device drivers follow a uniform structure invoking read, write and setting the parameters (also known as `ioctl` calls). This makes the implementation of a secure device driver on different hardware platforms relatively easy under Linux. The secure driver implements an `ioctl` call to initialize the selected de-scrambling algorithm with the de-scrambling key.

If any user tries to implement his or her own insecure device driver, the device driver will fail to understand the security parameters initialized by the application. The receiving application will immediately terminate, since the device driver does not understand the security parameter.

Streaming Data

The implementation on the receiver end uses two threads in round-robin mode. The receiving thread listens on a secure port for packets from the sender. The device driver thread begins writing to the sound device driver after receiving an initial buffer of data.

Our implementation of streaming uses a simple methodology. Sophisticated streaming with more control of startup latency and throughput is consistent with the proposed security techniques.

HTTPS and RTP Clients

The HTTPS protocol is used in our security model for username and password authentication, as well as for client-server mutual authentication. The client's request for a particular file is transmitted to the server using HTTPS. After user authentication, the server starts a secure RTP session, which is used for handshaking and the streaming data transmission.

In practice, multimedia transmission is usually done using the Real Time Transport Protocol (RTP) with UDP at the transport layer. Our proposed secure model uses secure RTP for transmission between the endpoints.

A comparison between the three protocols secure RTP, RTP and HTTPS, with respect to startup latency and throughput, is of interest. This comparison helps us to estimate the performance penalty of our proposed secure streaming media model. Simple HTTPS and RTP clients were implemented for the sole purpose of obtaining timing information. Details of this comparison appear below in Section 5.6.2.

Creating Receiver Components

We have developed a "create client" program, which is used to automate the process of generating a shared symmetric key for the receiver

and the encrypted list of supported scrambling algorithms. The openssl library function `crypt`, which implements DES encryption (Grabbe 2003) and the MD5 hash (Rivest 2003), among other crypto algorithms, is used to encrypt the supported algorithms string in the receiver.

Deployment

This section discusses test cases, performance data, and we provide a brief analysis of the proposed system.

Performance Issues

We performed a variety of some timing tests. Our primary goal was to determine the relative cost imposed by the security measures we have adopted. The tests were performed on a 1Ghz Pentium 4 personal computer. The tests consisted of looping over many different calls of the encryption algorithms and using the "clock" C library function to make timing measurements. This method is portable, if not the most refined.

Figure 2 shows two graphs, the transmission throughput between two endpoints and the throughput using our secure device driver. Obviously that the proposed secure streaming media model is slower than the generic (no security) model but it is within reasonable limits. Primarily, the added cost results from the scrambling and de-scrambling of packets.

The time required for scrambling and de-scrambling of data packets is constant. The transmission throughput saturates as we increase the number of octets in the packet. The cryptographic mechanisms used by the server and client will be the bottleneck for performance. Crypto hardware accelerators could be used on both ends to improve performance but based on our test results, this appears to be unnecessary.

Figure 2. Performance graph

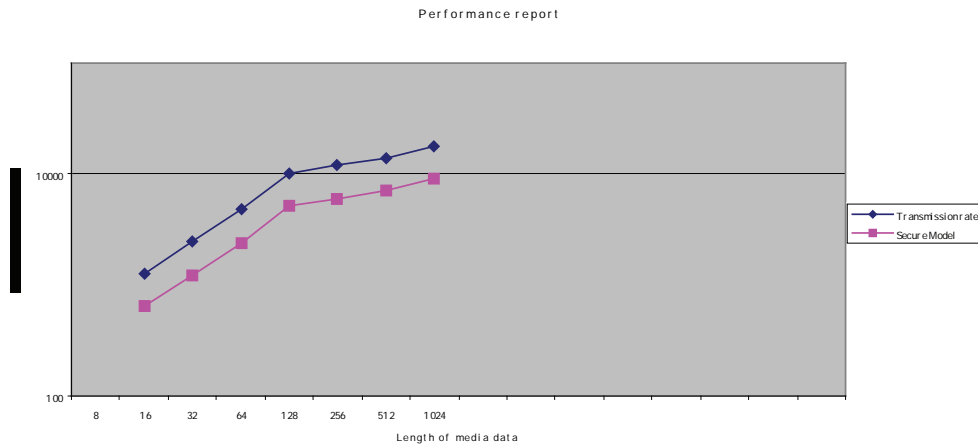


Table 1. Comparison of secure RTP, RTP and HTTPS

(Milliseconds)	Secure RTP	RTP	HTTPS
Startup time	2423.07	1616.61	2046.29
End receiving	15313.86	23305.93	5353.86

Comparison of Secure RTP, RTP and HTTPS

HTTPS and RTP clients were also implemented for the purpose of comparison to our proposed secure RTP model. Table 1 contains the startup latency and throughput times for streaming 4003604 bytes under each of the three protocols.

From the results in Table 1, it can be seen that the difference in startup latency for the three protocols is negligible. HTTPS transmission has a slight advantage over the other two protocols for transmission of longer files. HTTPS uses TCP as the transport layer protocol whereas in our implementations, RTP and secure RTP use UDP as the transport layer protocol (secure RTP allows for the usage of either UDP or TCP as the transport layer protocol). In any case, the performance penalty for secure RTP appears to be negligible.

Testing of Secure Driver

Our implementation has been successfully tested using two audio device drivers, namely, the Intel 810 audio driver and the Crystal Sound Fusion audio driver. For both of these device drivers the implementation was straightforward. The most significant driver implementation issues included compiling the de-scrambling algorithms file and modifying the write functions. The device driver maintains the initialized de-scrambling key and chosen algorithm in the current state structure.

Startup Latency and Throughput

The implementation uses a very simple streaming mode with a predefined initial buffer size. As soon as the initial buffer is filled, the receiver begins writing to the audio device driver. Depending on

the available processing power and network cards, the size of the initial buffer can be increased or decreased. Currently, our implementation assumes that the network card is available to be used at its maximum throughput. Our initial buffer size was determined by trial and error using a variety of different parameters.

In practice, several streaming servers and receiver plug-ins are available which have variable settings for startup latency. Consequently, further fine-tuning can be easily determined for a specific platform. From the test cases we have conducted, it can be concluded that the proposed security model achieves security without a severe performance penalty.

CONCLUSION

In this chapter we discussed various issues concerning digital rights management in general. We then considered a software-based DRM system designed to protect streaming media. Our proposed system provides a reasonable level of robustness against attack and a high degree of BOBE-resistance.

It would be interesting to develop an analogous DRM system based on tamper-resistant hardware by making use of, say, TCG/NGSCB (Trusted Computing Group 2007, Next Generation Secure Computing Base 2007). Such a system would offer an inherently higher degree of security due to the difficulty of attacking the hardware as compared to attacking software. However, with streaming media it may be the case that a well designed, software based DRM system is sufficiently robust to be of significant practical value.

REFERENCES

- Ackerman, E. (2003). *Student skirts CD's piracy guard*, http://www.newmediamusings.com/blog/2003/10/student_skirts_.html
- Anderson, R. (2001). *Security Engineering: A Guide to Build Dependable Distributed Systems*, John Wiley & Sons, Inc. <http://www.cl.cam.ac.uk/~rja14/book.html>
- Audiodriver (2007). <http://community.qnx.com/sf/wiki/do/viewPage/projects.bsp/wiki/Audiodriver>
- Balepin, I. (2003). *Superworms and crpytovirology: A deadly combination*, http://www.csif.cs.ucdavis.edu/~balepin/new_pubs/worms-cryptovirology.pdf
- Biddle, P., P. England, M., Peinado, & Willman, B. (2002). The darknet and the future of content distribution. *2002 ACM Workshop on Digital Rights Management*, November 18, 2002, <http://crypto.stanford.edu/DRM2002/prog.html>
- Calbet, X. (2006). *Writing device drivers in Linux: a brief tutorial*. http://www.freesoftwaremagazine.com/articles/drivers_linux
- Chang, H., & Atallah, M. J. (2001). *Protecting software code by guards*. Workshop on Security and Privacy in Digital Rights Management
- Cerven, P. (2002). *Crackproof Your Software*, No Starch Press
- Collberg, C., Thomborson, C., & Low, D. (1997). *A taxonomy of obfuscating transformations*. Technical Report 148, Department of Computer Science, University of Auckland, July 1997
- Crowcroft, J., Handley, M., & Wakeman, I. (1999). *Internetworking Multimedia*. Morgan Kaufmann, 1999
- CDFreaks. (2002). *Easy solution to bypass latest CD-audio protection*, 2002, <http://www.cdfreaks.com/news/4068>
- Doctorow, C. (2002). EFF Consensus at Lawyerpoint, Hollywood want to plug the 'analog hole', May 23, 2002, <http://bpdg.blogs.eff.org/archives/000113.html>

- Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc.
- Guignard, B. (2002). *How secure is PDF?* <http://www.cs.cmu.edu/~dst/Adobe/Gallery/PDFsecurity.pdf>
- Holankar, D., & Stamp, M. (2004). Secure streaming media and digital rights management. *Proceedings of the 2004 Hawaii International Conference on Computer Science*. Honolulu, Hawaii, January 2004
- Jacob, M., Boneh, D., & Felten, E. (2003). Attacking an obfuscated cipher by injecting faults. *Lecture Notes in Computer Science*, 2696, 16-31. Springer
- Kaufman, C., Perlman, R., & Speciner, M. (2002). *Network Security: Private Communications in a Public World*. Prentice Hall
- Linux (2007). <http://www.drfruitcake.com/linux/stest.html>
- Mishra, P., & Stamp, M. (2003). Software uniqueness: how and why. *Proceedings of ICCSA 2003*, P.P. Dey, M.N. Amin and T.M. Gattton, editors, July 2003
- Next Generation Secure Computing Base. (2007). <http://www.microsoft.com/resources/ngscb/default.msp>
- OpenSSL (2007). <http://www.openssl.org>
- Real-time Transport Protocol (RTP), RFC 1889. (2007). <http://www.faqs.org/rfcs/rfc1889.html>
- Schneier, B. (1996). *Applied Cryptography*, second edition. John Wiley & Sons, 1996
- Baughner, M., et al. (2004). The Secure Real Time Transport Protocol, *RFC 3711* <http://www.ietf.org/rfc/rfc3711.txt>
- Vovida.org. (2004). *Secure RTP*, <http://www.vovida.org/protocols/downloads/srtp/>
- “Beale Screamer”. (2001). *Microsoft’s digital rights management scheme—technical details*, <http://cryptome.org/ms-drm.htm>
- Shamir, A., & van Someren, N. (1999). Playing “hide and seek” with stored keys. *Lecture Notes in Computer Science*, 1648, 118-124. Springer
- Secure RTP. (2007). <http://srtp.sourceforge.net>
- Stallings, W. (2003). *Cryptography and Network Security: Principles and Practices*. Prentice Hall
- Stamp, M. (2003a). Digital rights management: the technology behind the hype. *Journal of Electronic Commerce Research*, 4(3), 102-112, <http://www.csulb.edu/web/journals/jecr/issues/20033/paper3.pdf>
- Stamp, M. (2003b). Digital rights management: For better or for worse? *ExtremeTech*, May 1, 2003, <http://www.extremetech.com/article2/0,3973,1051610,00.asp>
- Stamp, M. (2005). *Information Security: Principles and Practice*. John Wiley & Sons, Inc.
- Stamp, M. (2002, September). Risks of digital rights management, Inside Risks 147, *Communications of the ACM*, 45(9), 120
- Stamp, M. (2004). Risks of monoculture, Inside Risks 165, *Communications of the ACM*, 47(3), 120.
- Tiny httpd (2007). http://www.acme.com/software/thttpd/thttpd_man.html
- Trusted Computing Group. (2007), <https://www.trustedcomputinggroup.org/home>
- Wheeler, D., & Needham, R. (1994). *TEA, a tiny encryption algorithm*, <http://www.ftp.cl.cam.ac.uk/ftp/papers/djw-rmn/djw-rmn-tea.html>
- Windows Media Digital Rights Management Offering. (2004). <http://www.microsoft.com/windows/windowsmedia/wm7/drm/offering.aspx>

Yodaiken, V. ELEC (2000). <http://www.linuxdevices.com/articles/AT3792919168.html>

KEY TERMS

Break Once, Break Everywhere Resistance (BOBE): A highly desirable property of a DRM system. A system is BOBE-resistant if a successful attack on a piece of digital content does not break the entire system.

BOBE Resistance: (See break once, break everywhere resistance.)

Device Driver: Low-level software that the operating system uses to control a hardware device.

Digital Rights Management (DRM): Consists of the methods used to control access to copyrighted digital content.

Linux: On open source operating system based on Unix.

Open Source Software: Software for which the source code is freely available.

Streaming Media: Consists of digital media transmitted over a network in such a way that it can be consumed while the transmission is in progress.

Windows Media Player: A digital media player from Microsoft.