Digital Rights Management and Code Obfuscation

by

Amit Sethi

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2003

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Digital Rights Management (DRM) involves retaining control over digital information, even after it has been made public. Preventing illegal file sharing on the Internet, which is a topic that has recently received a large amount of media attention, is just one instance where DRM is needed.

In this thesis, we attempt to create formal definitions for DRM. Currently, there is a lack of such formal definitions, which is one reason why DRM schemes have achieved little success. We will also examine two DRM schemes that can be cracked easily: Microsoft DRM 2.0, and the Content Scrambling System. We then discuss the reasons why DRM schemes have been unsuccessful so far, and why a good DRM scheme must incorporate secure hardware, secure software, and an efficient legal system. We also briefly discuss several issues related to DRM, such as privacy.

Code Obfuscation involves hiding a program's implementation details from an adversary. One application of code obfuscation involves hiding cryptographic keys in encryption and decryption programs for a cryptosystem. Code obfuscation is directly applicable to DRM schemes, where the adversary has access to a program that contains secret information. For example, a music player may contain a secret key that it uses to decrypt content. The secret key must be hidden from the adversary, since otherwise, he/she could use the key to write his/her own decryption program, and distribute it to circumvent the DRM scheme.

We discuss the proof from [2] that shows that code obfuscation is impossible in general. This, however, does not mean that code obfuscation cannot be achieved in specific cases. We will examine an obfuscated version of the Data Encryption Standard, and discuss the circumstances under which it is insecure. We also examine a toy example of a block cipher called *Simple Block Cipher (SBC)*, and apply obfuscation techniques to SBC to hide the secret key, and then attempt to obtain the secret key.

Contents

Part I

Digital Rights Management

Chapter 1

Introduction

Informally, Digital Rights Management (DRM) involves retaining control over digital information after it has been made public. It can restrict several aspects of file access, such as the number of times a person is allowed to access the file, the amount of information that the person is allowed to obtain from the file, and the operations that a person is allowed to perform on the file (such as altering, copying, and printing). DRM is not restricted to files, however, and restrictions can also be applied to other types of data, such as streaming content. Its applications include protection of intellectual property, and access control for sensitive data.

Currently, a provably secure DRM scheme does not exist, and most DRM schemes in existence have been cracked. A major obstacle in the development of a secure DRM scheme, is a lack of formal definitions and requirements. Clearly, we cannot design a secure DRM scheme unless we have formal definitions for what DRM is, and what it means for a DRM scheme to be secure. We attempt to address the problem of a lack of definitions in this thesis. This problem is being examined in detail by those working on *Open Digital Rights Language (ODRL)*, a language for formally describing digital rights [26]. ODRL will provide a formal and precise way of describing digital rights, whereas we will take a broader look at the problem. Another problem is that it can be proven that a *code obfuscator* cannot exist [2]. Code obfuscation is the topic of the second part of this report, and seems to be a requirement for any software-only DRM solution.

Protection of intellectual property has always been a concern, and there has never been

a perfect solution. In the digital world, the problem has become much more widespread, since one can copy data effortlessly and quickly. With the popularity of the Internet and the ease of file sharing, information can be distributed around the world extremely quickly. This, coupled with the difficulty of tracking distributors of the illegal content, has resulted in illegal file sharing becoming commonplace. That is why DRM has become such a topic of interest in recent years.

Currently, access control for sensitive data is usually accomplished with tools such as firewalls and encryption schemes that prevent unauthorized people from gaining access to the data. However, these tools do not protect against other threats such as an authorized person providing the data to an unauthorized person. This is a real threat for corporations with employees who may have a conflict of interest, or for the military, who cannot afford to have any of its secrets revealed to the enemy. DRM can be a great asset in this case.

DRM is a relatively new concept, but it is gaining popularity quickly because of its importance. Record companies would like to release music in digital formats because of the cost and convenience benefits that it offers, but this makes it easy for users to make exact copies of the music, and distribute them for free all across the world. Similarly, software companies would like to ensure that only those users who have paid for their software can use it. The military would like to ensure that if any of their files containing sensitive or confidential information are accidentally or intentionally released to an unauthorized person, then he/she cannot obtain the information contained in the files. All of these are examples of situations where DRM is needed.

We can see analogues of DRM in the physical world. For example, if we wanted somebody to only be able to read a particular book once (and not be able to do anything else useful with the book, such as copy it), we could lock him/her in a room containing just the book, and then watch him/her using a video camera to ensure that he/she does not read it more than once, or tear out pages from the book and carry them out to copy later, or alter the contents of the book. We could then allow the person to leave the room. This would be difficult to perform, but would be secure for most purposes. However, there are many reasons why it is much more difficult (or impossible) to do this when the book is in a digital format, and is being read on a general-purpose computer. One reason is that when the book is being read on a general-purpose computer, it is in memory at some point. The

reader could simply perform a memory dump, and obtain the contents of the book. Even if the memory contents are encrypted, the information must be decrypted at some point before it can be displayed. In this case, the user can get a copy of the book by determining when and where the information is being decrypted, and then recording it at that point. It is not difficult to see that the main problem is that the user has complete control over the hardware and the execution environment. There are ongoing (separate) efforts by Microsoft [24] and the Trusted Computing Platform Alliance [28] to develop a secure operating system and tamper-resistant hardware that would enable software manufacturers to restrict the users' control over the hardware and execution environment.

The music recording industry has been aggressively pursuing a secure CD copy protection scheme for several years. There have been many attempts at creating a secure DRM scheme for music CDs, but none of these attempts have been successful so far. A major problem in designing DRM schemes for audio CDs is that backward compatibility needs to be retained. Consumers cannot be expected to purchase a new CD player every time a new DRM scheme is implemented on a CD. The backward compatibility requirement severely limits what DRM scheme designers can do. There has been consumer backlash against CD copy protection schemes that cause CDs to not be playable on computers, and on some CD players [30]. There are extreme cases where copy protection schemes inflict damage by causing any computer attempting to read a protected CD to crash [21]. The reasoning behind such schemes is that if the user cannot read the CD on a computer, then he/she cannot copy it using the computer. Schemes such as these, that can potentially cause damage, even if the CD is being used legitimately, are clearly not acceptable.

In other cases, there are no backwards compatibility issues, and somewhat secure DRM schemes have been designed and implemented. For example, in some video game consoles, a copied game CD is not playable, unless the user solders a microchip somewhere inside the console [17]. This keeps most people from using copied CDs, since they do not want to risk damaging the console and losing the manufacturer's warranty by attempting to solder in a microchip. In addition, distributors of hardware designed to circumvent DRM schemes can be tracked down and prosecuted fairly easily. They can be found by tracking payments for the illegal hardware, whether they are made online, or using the postal service. Strict laws and severe punishments can effectively deter people from distributing such hardware.

The DRM schemes that we will examine and analyze in this thesis are based on cryptographic techniques. The main idea of such schemes is to encrypt the information to be protected, and then provide the decryption key only to users that are authorized to view the information. The key is somehow hidden from the users such that they cannot distribute it to others.

Although the type of DRM schemes described above are the only ones that we will examine, there are also other types that are used in practice. There are some that scramble data such that only special hardware provided by the content provider can descramble it. Such schemes are widely used for broadcasting television signals. In the case of TV signal descramblers, pirated hardware can usually be remotely detected, and in some cases, can even be destroyed [27]. Similarly, scrambling data or the table of contents, is commonly used to protect audio CDs [18].

Currently, most DRM schemes focus on providing information to those who have paid for it, and completely hiding it from others. They also usually prevent the users who have paid for it from making any copies of the information. However, some schemes let unpaying users view a small part of the information. For example, one particular DRM scheme lets unpaying users listen to the first 30 seconds of songs [4]. However, as mentioned earlier, DRM schemes can have many different types of restrictions. These could include restrictions on the amount of information that a user can access, what the user can do with the information, what applications can access the information, how many times a user can access the information, etc.

We will develop a DRM model in the next chapter. In the context of video and audio, we need to be especially careful about designing a DRM model. Obviously, we cannot prevent a person from using a camcorder to record a movie, or a tape recorder to record a song. Our DRM model must be restricted to how the protected file is handled until it reaches any output device.

The first part of this thesis is organized as follows. Chapter 2 contains DRM definitions. Chapter 3 analyzes Microsoft DRM 2.0, and presents a crack for it. Chapter 4 analyzes the Content Scrambling System, and discusses how it can be cracked. Chapter 5 discusses various possible DRM solutions, and how they need to be combined in order to implement good DRM schemes. Finally, Chapter 6 examines issues related to DRM, such as privacy

and fair-use laws.

Chapter 2

Definitions

In this chapter, we will define DRM, describe attack models for DRM, discuss adversarial goals for attacks on DRM schemes, and discuss what it means for a DRM scheme to be secure. Currently, there do not exist any formal definitions for DRM, and the ones given in this chapter are not complete by any means. They are created mainly for the purpose of this thesis, but cover most, if not all informal DRM definitions and current DRM schemes.

2.1 Digital Rights Management

A DRM scheme consists of the following:

$\mathcal{B}$ is a finite set of possible *plaintexts*
$\mathcal{U}$ is the (finite) set of clients/users
$\mathcal{C}$ is a (finite) set of *contracts*
$\mathcal{E}$ and $\mathcal{D}$ are algorithms

Contracts
For each client $u \in \mathcal{U}$ and plaintext $b \in \mathcal{B}$, there is a contract $c_{u,b} \in \mathcal{C}$, which is a triple $(S_{u,b}, T_{u,b}, A_{u,b})$ where

$S_{u,b}$ is the (possibly empty) subset of bits of b that u is allowed to access
$A_{u,b}$ is a list of operations that u is allowed to perform on $S_{u,b}$
$T_{u,b}$ is a list of the number of times that u is allowed to perform each of the operations in $A_{u,b}$ on $S_{u,b}$

Algorithms

- $\mathcal{E}$ is an algorithm that takes as input $b \in \mathcal{B}, P \subset \mathcal{U}$, and $Q \subset \mathcal{C}$ (where Q contains $c_{u,b}$ for all $u \in P$), and outputs a sequence of bits, $\mathcal{Z}$, called the encoded-text. That is, $\mathcal{Z} = \mathcal{E}(b, P, Q)$. We will refer to $\mathcal{E}$ as the encoding algorithm.

- $\mathcal{D}$ is an algorithm that takes as input a sequence of bits $\mathcal{Z}, u \in P \subset \mathcal{U}$ and $c_{u,b} \in Q \subset \mathcal{C}$ (where b is the plaintext corresponding to the encoded-text $\mathcal{Z}$), and enables u to perform one of the operations in $A_{u,b}$ on $S_{u,b}$ if u hasn't already performed the operation the maximum number of times allowed by $T_{u,b}$. If a user $u \notin P$ attempts to run $\mathcal{D}$ on $\mathcal{Z}$, he/she cannot get access to any information in $\mathcal{Z}$ unconditionally. We will refer to $\mathcal{D}$ as the decoding algorithm.

Informally, the encoding algorithm creates a sequence of bits for a set of users, P, such that the information that the bits represent can only be accessed by users in P, using the decoding algorithm, and such that the restrictions in the set of contracts, Q, are enforced. We use subsets P and Q of $\mathcal{U}$ and $\mathcal{C}$, because the length of the encoded-text could depend on the number of users and contracts. Since each encoded-text probably will not need to be accessed by the entire set of users, we could potentially obtain a much more efficient scheme by using appropriate subsets of $\mathcal{U}$ and $\mathcal{C}$ for each encoded-text, instead of using $\mathcal{U}$ and $\mathcal{C}$ themselves.

We have purposely omitted any references to *keys* that are usually present in definitions of cryptographic systems. This is because not all DRM schemes are based on cryptography, and there are several schemes that do not use keys.

Of course, for our definitions to make sense, we must have security conditions that make it impossible (or infeasible) for a user to determine any bits of $b \in \mathcal{B}$, since otherwise, all of the above is meaningless. The user could simply determine b, and use it any way he/she wants. It should also be impossible to add, delete or modify contracts. We will examine security conditions later. For now, we will consider a toy example of a DRM

scheme (without the algorithms).

Example

$\mathcal{B} =$ {01011}
$\mathcal{U} =$ {Alice, Bob}
$\mathcal{C} =$ $\{c_{\text{Alice,01011}} = (\text{01xxx}, (\text{view, print}), (\text{unlimited, 2})),$
$c_{\text{Bob,01011}} = (\text{01011}, (\text{view, copy, print}), (\text{unlimited, 1, 2}))\}$

In the above example, Alice is only allowed to perform two operations. She is allowed to view the information "01" an unlimited number of times, and is allowed to print it twice. The actual bits "01" are of course hidden from Alice, and she can only view them in specific applications, and can print the information that they represent. Bob is allowed to copy the information once, in addition to being allowed to perform all operations that Alice can perform.

2.2 Attack Models

As in traditional cryptography, there are several attack models for DRM. We will discuss these below:

- *Encoded-text only attack*: This attack model is analogous to the ciphertext-only attack in traditional cryptography. In this model, the adversary only has access to some strings of encoded-text.

- *Known plaintext attack*: As in traditional cryptography, the adversary has access to some strings of plaintext, and corresponding encoded-texts in this attack model.

- *Chosen plaintext attack*: As in traditional cryptography, the adversary can choose some plaintext strings, and construct corresponding encoded-text strings in this attack model.

- *Chosen encoded-text attack*: This attack model is analogous to the chosen ciphertext attack in traditional cryptography. In this attack model, the adversary can choose some encoded-text strings, and construct the corresponding plaintext strings.

- Side-channel attack: In this attack model, the adversary does not attack the DRM scheme, but instead, attacks its actual implementation. This is the most important attack model for DRM, since the adversary typically has complete control over the execution environment that extracts information from the encoded-text. He/she can perform fault analysis, timing analysis, etc. to crack a DRM scheme's implementation.

2.3 Adversarial Goals

Using one of the attack models discussed in the previous section, an adversary, u, might try to achieve one or more of several goals. These goals are given below:

- Obtain some information represented by bits in b that are not in $S_{u,b}$.
- Do operations on $S_{u,b}$ that are not allowed by $A_{u,b}$.
- Use the information in $S_{u,b}$ more than the number of times allowed by $T_{u,b}$.
- Determine k bits of b with probability significantly greater than $1/2^k$.
- Gain ability to do one or more of the above for any given encoded-text.

2.4 Security

In this section, we will discuss conditions on DRM schemes that will ensure their security. Informally, we will call a DRM scheme secure if it binds $\mathcal{B}$,$\mathcal{U}$ and $\mathcal{C}$ such that the rules in $\mathcal{C}$ are enforced. More formally, we have the following:

- A DRM scheme is *perfectly secure* if:
  Given any $P' \subset \mathcal{U}$, and $\mathcal{Z} = \mathcal{E}(b, P, Q)$ where $b \in \mathcal{B}$, $P \subset \mathcal{U}$ and $Q \subset \mathcal{C}$, the set of users P' cannot determine any k bits in b, with probability $> 1/2^k$ (which can be obtained by randomly guessing the bits), and no $u \in P'$ can violate the restrictions in its contract.

- A DRM scheme is *N-secure* if:

  Given a security parameter N, any $P' \subset \mathcal{U}$ where $|P'| \leq N$, $\mathcal{Z} = \mathcal{E}(b, P, Q)$ where $b \in \mathcal{B}$, $P \subset \mathcal{U}$ and $Q \subset \mathcal{C}$, the set of users P' cannot determine any k bits in b with probability $> 1/2^k + \mathcal{O}(1/2^N)$, and no $u \in P'$ can violate any restrictions in its contract, except with probability $< \mathcal{O}(1/2^N)$.

In general, perfect security is probably impossible to achieve, and we should strive to obtain N-secure DRM schemes. If we have a N-secure DRM scheme, we can increase security as required, by adjusting the security parameter.

The above definitions are similar to many other cryptographic definitions, and they seem to cover most, if not all existing DRM schemes. Of course, the definitions are relatively informal and are not meant to be complete. Complete definitions will probably require several years of research.

Now that we have DRM related definitions, we can examine some DRM schemes, discuss their implementations, and analyze their strengths and weaknesses. In the next two chapters, we will examine Microsoft DRM 2.0, as applied to WMA files, and the Content Scrambling System used to encode information on Digital Versatile Discs (DVDs), in detail.

Chapter 3

Case Study 1: Microsoft DRM 2.0

3.1 Introduction

Microsoft DRM 2.0 was cracked soon after its release by a programmer using the pseudonym "Beale Screamer." We will examine the scheme as applied to audio (WMA) files in detail, and see that even though the scheme is very clever, and was designed carefully, there is a major flaw in the implementation that enables an adversary to remove any restrictions imposed by the DRM scheme. This case study illustrates that DRM schemes are much more difficult to design than traditional cryptographic protocols that are designed for secure communication over an insecure channel. Most of the information in this chapter is obtained from [14].

Microsoft DRM 2.0 incorporates many cryptographic techniques to protect information. The main idea is to encrypt the information to be protected with a "content key," and send the content key only to clients who have paid for the information. Before the key is transmitted to a client, it is encrypted using another key that is specific to the client. The scheme relies on keeping all (client-specific) private keys secret from their owners. Note that this is very different from a traditional encryption scheme where the owner of a private key always has access to it. Of course, the difference here is that no clients can be trusted.

Microsoft DRM 2.0 uses several cryptographic algorithms, which are listed below, along

with reasons for their use:

- An *Elliptic Curve Cryptosystem* (ECC) is used as a public-key cryptosystem. The main reason for using ECC instead of a more standardized cryptosystem such as RSA, is performance. RSA requires 1024-bit keys for sufficient security, whereas ECC only requires 160-bit keys for the same level of security. That is one reason why ECC operations are much faster than RSA operations. This is very important for some portable devices, where the processors are relatively slow, and the memory sizes are quite small.

- *DES* is a block cipher used to hide keys in the encoded-text. We will discuss the process later in this chapter

- The stream cipher, *RC4*, is used to encrypt the plaintexts, as the first step in protecting them. *RC*4 is used instead of a block cipher for performance reasons. Stream ciphers tend to be more efficient than block ciphers.

- The hash function, *SHA-1*, is used to generate some keys. The process used to generate the keys is described later in this chapter.

- A proprietary cipher, which is referred to as *MultiSwap* by Beale Screamer, is used to generate Message Authentication Codes.

Detailed information about how all of the above algorithms are used, is given later in this chapter. An implementation detail, which is important mainly if one wants to implement a crack, or analyze actual client-server communication, is that all communication is performed using a modified *Base64* encoding. First, the standard *Base64* encoding is applied. *Base64* is used to simplify communications, so that printable characters can be transferred, rather than raw binary data. After the encoding is applied, in some places, Microsoft's algorithm substitutes '*' for '/', and '!' for '+', and at other places, it substitutes '@' for '/' and '%' for '!'.

3.2 MultiSwap

Microsoft's proprietary block cipher, MultiSwap, uses two main operations: 32-bit multiplications, and swaps of the two halves of 32-bit words. MultiSwap is a very weak cipher, and it can be broken using a chosen-plaintext attack with running time 2^{14}, or a known-plaintext attack with running time 2^{25} [3]. Thus, it is not safe to use for hiding information. The weakness of the cipher could possibly be used to crack Microsoft DRM 2.0, but the attack discovered by Beale Screamer is more practical, and so, there has not been much interest in finding a crack that exploits the weaknesses of MultiSwap.

A MultiSwap key consists of twelve 32-bit words, $k[0], k[1], ..., k[11]$. In Microsoft's implementation, the least significant bits of $k[0], k[1], ..., k[11]$ are all set to 1, even though only 10 of them actually need to be set to 1 for the cipher to work properly. A 64-bit initialization vector, IV, is also used in the cipher. Let the state be denoted by $s[0], s[1]$. The state is initialized to IV.

We first define a function f such that
$f(a) = swap(swap(swap(swap(swap(a * k[0]) * k[1]) * k[2]) * k[3]) * k[4]) + k[5]$ where a is a 32-bit word, *swap* swaps the two 16-bit halves of a 32-bit word, '$*$' is multiplication modulo 2^{32}, and '$+$' is addition modulo 2^{32}.

Now, to encrypt a 64-bit plaintext p, we do the following:

1. x = first 32 bits of p.
2. $s[0]$ = first 32 bits of IV.
3. $s[1]$ = second 32 bits of IV.
4. $s[1] = s[1] + f(x + s[0])$ where $+$ is addition modulo 2^{32}.
5. $s[0] = f(x + s[0])$.
6. x = second 32 bits of p.
7. $s[1] = s[1] + f(x + s[0])$ where instead of keys $k[0], \ldots, k[5]$, we use the keys $k[6], \ldots, k[11]$ in f.

8. $s[0] = f(x + s[0])$ where instead of keys $k[0], \ldots, k[5]$, we use the keys $k[6], \ldots, k[11]$ in f.

9. Output $s[0], s[1]$.

The first half of the algorithm is illustrated in figure 3.1.

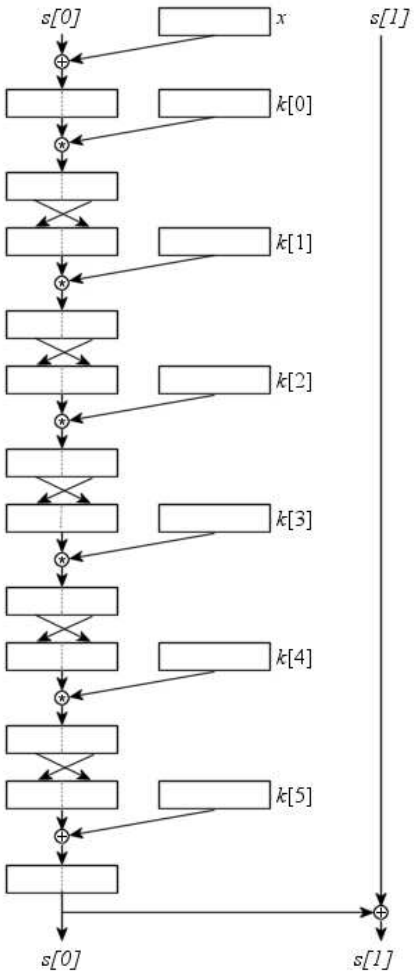

Figure 3.1: First half of MultiSwap encryption

The result can be decrypted because all keys are odd, and hence, they have multiplicative inverses modulo 2^{32}. We can easily see that $f^{-1}(c) = swap(swap(swap(swap(swap(c-$

$k[5]) * k^{-1}[4]) * k^{-1}[3]) * k^{-1}[2]) * k^{-1}[1]) * k^{-1}[0]$ where $k^{-1}[x]$ for $x = 1, 2, ..., 5$ is the inverse of $k[x]$ modulo 2^{32}. Hence, we can decrypt y using the following steps:

1. $s[0] =$ first 32 bits of y.
2. $s[1] =$ second 32 bits of y.
3. $s[1] = s[1] - s[0]$.
4. $s[0] = f^{-1}(s[0])$ where instead of keys $k[0], \ldots, k[5]$, we use the keys $k[6], \ldots, k[11]$ in f^{-1}.
5. $x[1] = s[0] - s[1] + b$ where $b =$ second 32 bits of IV.
6. $s[0] = s[1] - b$.
7. $s[0] = f^{-1}(s[0])$.
8. $x[0] = s[0] - a$ where $a =$ first 32 bits of IV.
9. Output $x[0], x[1]$.

MultiSwap is never used for actual encryption, but instead, is used to create a MAC. Assuming that the length of the message that we want to generate the MAC for, is a multiple of 64 bits, we set $IV = 0$, and then encrypt the entire message, where each ciphertext block is used as the IV for encrypting the next block. The output for the last block of the message is the MAC for the message. We will discuss how the MAC is used, later in this chapter.

3.3 Elliptic Curve Cryptography

For Elliptic Curve Cryptography (ECC), Microsoft uses the ElGamal encryption scheme, and a curve over $\mathbb{Z}_p$ where p is the 160-bit prime given below. The curve is defined by $y^2 = x^3 + ax + b$, where a and b are coefficients given below. All numbers below are given in hexadecimal.

p	89abcdef012345672718281831415926141424f7
a	37a5abccd277bce87632ff3d4780c009ebe41497
b	0dd8dabf725e2f3228e85f1ad78fdedf9328239e
x	8723947fd6a3a1e53510c07dba38daf0109fa120
y	445744911075522d8c3c5856d4ed7acda379936f
Order of curve	89abcdef012345672716b26eec14904428c2a675

(x', y') is a point on the elliptic curve, and is a generator for all points on the elliptic curve. The above constants are fixed, and are used by all parties in the Microsoft DRM 2.0 scheme.

The server's public and private ECC keys are included in the server software by default. The server's public ECC key is also included in client software. The client generates its own public/private key pair (and stores it in an obfuscated form) when the client software is initialized on a computer. The process, during which a server gets a client's public key is discussed later, in section 3.7.

3.4 Keys

Each file protected by Microsoft DRM 2.0 is encrypted with a content key. The scheme can be cracked by determining the content key, and we will discuss that process later in this chapter. The content key is not used directly in an encryption scheme, but is processed for several uses. First, we use SHA-1 to produce a 20-byte hash of the content key. The first 12 bytes of the hash are used as an RC4 key (K_R), and the second 8 bytes are used as a DES key (K_D). K_R is used to encrypt a 64-byte string of all zeroes. The least significant bit of each of the first 12 words of the output is then set, and the resulting 12 words (48 bytes) are used as a MultiSwap key ($k[0], k[1], ..., k[11]$). The next 8 bytes are used as a DES encryption in-whitening mask (IWM), and the last 8 bytes are used as a DES encryption out-whitening mask (OWM).

3.5 Encoding

The plaintext cannot be encrypted as a single stream, because we want the information to be randomly accessible. We cannot use the same RC4 key to encrypt each packet either, since that compromises security. To ensure that the above problems do not occur, Microsoft uses the following scheme to encrypt packets. Note that the size of each packet depends on the streaming server's configuration, and on the content size.

1. Given packet P, P' is P with size rounded down to a multiple of 8 bytes.
2. $M' = MultiSwapMAC_{k[0],k[1],...,k[11]}(P')$.
3. Swap the two 32-bit halves of M' to obtain M.
4. $R = RC4Encrypt_M(P)$. $R' = R$ with size rounded down to a multiple of 8 bytes.
5. $D = DESEncrypt_{K_D}(M \oplus IWM) \oplus OWM$.
6. Overwrite the 8-byte block of R corresponding to the last 8-byte block of R', with D, to obtain the packet, E, or encoded-text.

3.6 Decoding

Decoding involves basically the same steps as encoding, except that they are performed in the reverse order. However, there is a complication due to the fact that a part of the encrypted packet was overwritten with other information during the encoding process. The following steps are taken to decode a packet E, of encoded-text.

1. Given encoded packet E, let E' be E with size rounded down to a multiple of 8 bytes.
2. Let D be the last 8-byte block of E'.
3. $M = DESDecrypt_{K_D}(D \oplus OWM) \oplus IWM$.
4. $Q = RC4Decrypt_M(E)$. Q is the correct plaintext, except for the last full 8-byte block.

5. Swap the two halves of M to obtain M'.

6. Run $MultiSwapMAC_{k[0],k[1],\ldots,k[11]}$ on Q up to the last full 8-byte block. This is the next-to-last state $(s[0], s[1])$ seen by MultiSwapMAC when it computed the MAC during encryption. We know that M' is the final output of MultiSwapMAC. So, we can run the MultiSwap decryption algorithm on M' with $IV = (s[0], s[1])$ to obtain the original 8-byte block. Replace the last full 8-byte block of Q with this data to obtain the original plaintext packet, P.

3.7 Obtaining a License

The process of obtaining a license is described in this section. A license is simply a contract that allows the user owning it to obtain all information in the protected file, an unlimited number of times using Microsoft's software (Windows Media Player for WMA files). The absence of a license for a protected file on a client's computer implies that the client is not allowed to view any part of the file. A protected WMA file contains a DRMV2 object in its header. This object contains an XML object 6 bytes into the data part of the object. The XML object contains a `KID` element that identifies the content key for the file. If a license with this `KID` exists locally, it is used to decode the file. Otherwise, a license request is sent to the license server. This request contains a random challenge, and is 168 bytes in length. It is encoded using Microsoft's modified Base64 encoding. The first 80 bytes are two elliptic curve points, which contain an ECC-encrypted random RC4 session key, K_{RR}. It is encrypted using the server's public key, which is stored on each client's computer by default (since the public/private ECC key pair for all servers is the same). The remaining 88 bytes contain the "Client ID" encrypted using K_{RR} and RC4. After some client-server interaction, the server sends the license, encrypted with K_{RR}, back to the client. Finally, the client decrypts the license, and stores it locally.

3.8 Crack

In this section, we will examine a side-channel attack where the adversary attempts to obtain the complete plaintext corresponding to a given encoded-text. The adversary must

have a contract that allows him/her to access the information in the encoded-text. We will see that Microsoft DRM 2.0 is completely insecure in this attack context.

3.8.1 Determining Public/Private Keys

On clients' computers, secret keys are stored in linked lists that contain 32 bits per node. The linked list is interspersed with the code in the library. Since the files are shuffled in a random way for each client, the keys are very difficult to extract from the files themselves. Instead, we can load the library, which maintains an object state containing the keys, and then read the keys directly from the object. This is possible because the offset of the keys within the object is known. This is the basis of the crack that we discuss. Using a client's public/private keys, we can extract the content key from the client's license.

3.8.2 Determining the Content Key

To obtain the content key, a client first needs to determine its public/private ECC key pairs, and obtain a copy of the license. Once the content key is determined, the client can use the decryption algorithm to remove all protection from the WMA file.

Licenses are stored in a file named drmv2.lic. Each license entry in the file is an XML object containing the element `ENABLINGBITS`, which in turn contains subelements `ALGORITHM`, `PUBKEY`, `VALUE`, and `SIGNATURE`. `PUBKEY` must match one of the client's public keys. `VALUE` is the content key encrypted using `PUBKEY`, and can be decrypted using the corresponding private key.

Note that even though all clients who have a license for a particular file have the same content key, the licenses are not transferrable, because each license has a different value for `ENABLINGBITS`.

The file drmv2.lic can be accessed through Microsoft's *IStorage* and *IStream* interfaces. The top-level file has a lower level *IStorage* object for each `KID`, which can contain a set of licenses for each `KID`. To guarantee valid *IStorage* names, the `KID` is first processed to change all '/' characters to '@', and all '!' characters to '%'. The names of the *IStream* objects containing the licenses are the same as the `LID` (License ID) elements stored in the licenses. It is currently not known how `LID`'s can be generated from the content, and

so, we cannot directly open the required `LID` stream. Instead, we can enumerate through all available streams for the `KID`, testing each one to see if it contains a `PUBKEY` element that we know. If it does contain a `PUBKEY` element that we know, we guess that it is the license for the content. Then, we run `PUBKEY` and `VALUE` through a Base64 decoder, and finally, decrypt the content key, which is encrypted using the private key corresponding to `PUBKEY`.

3.9 Discussion

The complexity and features of the Microsoft DRM 2.0 make it evident that a lot of work was invested in the creation of the scheme. Microsoft managed to create a DRM scheme in which each packet of data could be encrypted with a different key, and the key could be hidden inside the packet itself, without increasing its size. This has the advantage that protected files can have the same format as unprotected files. In spite of the amount of thought put into the design, a fairly simple attack manages to crack the scheme.

The attack outlined in this chapter illustrates one of the major problems in creating secure DRM schemes. In traditional cryptographic protocols, there is at least one trusted party that attempts to communicate with a possibly untrusted party. In designing DRM protocols, a complication is that no clients can be trusted. Any client could act as an adversary, and attempt to crack the scheme.

There are several security problems in Microsoft DRM 2.0 that need to be addressed. Some or most may have been fixed in later versions of Microsoft DRM. Due to the fact that details of DRM scheme implementations are usually not provided by manufacturers, verifying the security of newer versions of Microsoft DRM is difficult. Hiding implementation details, i.e. security by obscurity, is generally considered a bad practice in cryptographic contexts. But, the lack of proper DRM definitions and security proofs for DRM schemes makes security by obscurity necessary. Once implementation details are made public, most DRM schemes available today can be cracked. Hence, it is difficult to determine how secure newer versions of Microsoft DRM are.

As for Microsoft DRM 2.0, the scheme's major flaw is a failure to properly obfuscate keys. As we will see in the second part of this thesis, obfuscation is actually impossible

in general. Even if Microsoft manages to make the scheme secure, it will not comprise a complete DRM solution on its own. Information pathways to monitors, speakers, and other output devices need to be secured as well. Whether Microsoft's Palladium, or another trusted computing platform manages to do this still remains to be seen.

Chapter 4

Case Study 2: Content Scrambling System

4.1 Introduction

Content Scrambling System (CSS) is used to protect Digital Versatile Discs (DVDs) against unauthorized copying. CSS takes a different approach to copy protection, compared to Microsoft DRM 2.0. Like Microsoft DRM 2.0, it also encrypts the content to protect it, but the main keys are fixed, and are hidden in hardware. Legal agreements are used to ensure that hardware manufacturers who are given access to the keys do not distribute them to others. However, the CSS scheme is very weak, and even if the keys can be hidden securely, the scheme can be cracked easily. Most of the information in this chapter is obtained from [15] and [11].

Each DVD contains an area containing encrypted content, and a hidden area. The contents of the hidden area cannot be read, except by an authenticated device. The hidden area contains a table of encrypted *disk keys*, a *disk key hash*, a *title key*, and a *region code*. The region code is not important for our discussion, and we will not discuss it.

Each DVD player contains some *player keys*, a region code, and an *authentication key* that is used for authentication with the host (computer, processor connected to the DVD drive, etc.). The host also stores the authentication key in its DVD playing software.

4.2 Overview of Decoding Process

Decoding the contents of a DVD requires several steps. These are listed below:

1. The host and DVD player use a challenge-response protocol for authentication, and for establishing a bus key (session key).
2. The DVD player finds the disk key using the player keys it possesses.
3. The DVD player uses the disk key to decrypt the title key.
4. The title key and disk key are sent to the host by the player. The bus key is used to encrypt the title and disk keys in transit.
5. The DVD player sends a sector to the host.
6. The host uses the sector key (described in the next section) to decrypt the sector.

4.3 Keys

An *authentication key* is used for the authentication between the DVD player and the host. During authentication, a *bus key* (session key) is negotiated. It is used to encrypt keys before sending them over the unprotected bus.

Each DVD player contains a small set of *player keys*, which is a subset of the entire set of player keys. There are a total of 409 player keys, $PK_1, PK_2, ..., PK_{409}$. This way, if a manufacturer releases its subset of player keys to the public, the manufacturer can be tracked down (as not all manufacturers have the same subset of player keys).

Each DVD has a *disk key*, DK. The disk key is stored in a data block that contains the following:

Hash of DK (DK encrypted with DK)	$H = E_A(DK, DK)$
DK encrypted with PK_1	$DK_1 = E_A(DK, PK_1)$
DK encrypted with PK_2	$DK_2 = E_A(DK, PK_2)$
$\vdots$	
DK encrypted with PK_{409}	$DK_{409} = E_A(DK, PK_{409})$

Here, E_A is the disk key encryption algorithm, which has the decryption inverse D_A. Similarly, we will also use E_B, the content encryption algorithm, which has the decryption inverse D_B. We will describe these algorithms later.

Suppose that the player contains PK_{52}. It uses it to try to decrypt the disk key, i.e. $DK = D_A(DK_{52}, PK_{52})$. Then, the player verifies the disk key using the following check: $DK = D_A(H, DK)$. If the check fails, the player tries another player key. There are several reasons why the check might fail for a particular player key. An error might have occurred on the disc during the manufacturing process, so that one or more of the encrypted disk keys may not have been recorded properly. A scratch on the disk might cause one of the encrypted disk keys to be improperly read. There might also be an error in the DVD player while it attempts to read the disk. There are many other reasons why the check might fail.

To decrypt the contents of the DVD, another key is required. This key is the *title key*, TK, and is found using the disk key. $ETK = E_B(TK, DK)$ is what is stored on the DVD, and the player decrypts it as following: $TK = D_B(ETK, DK)$.

Each sector of the data files is optionally encrypted using a *sector key* that is the exclusive-OR of TK and bytes 80-84 of the unencrypted first 128 bytes of the 2048 byte sector. Encryption is done using the CSS stream cipher, which is described later in this chapter.

4.4 Mutual Authentication

The DVD player performs mutual authentication with the host before sending any data to the host over the bus. During this process, it negotiates a bus key (session key) to prevent an adversary from reading plaintext from the bus. The authentication protocol proceeds as follows:

1. Host requests an *Authentication Grant ID (AGID)* from the DVD Player.
2. DVD Player sends the AGID back to the Host.
3. Host generates a *challenge* consisting of an arbitrary stream of bytes, and sends it to the DVD Player.

4. The DVD Player encrypts the challenge using the authentication key, and sends it to the host.

5. The host decrypts the encrypted challenge sent by the DVD Player, and ensures that it is the same as the challenge sent earlier.

6. The DVD Player is now authenticated.

7. The DVD Player generates a challenge consisting of an arbitrary stream of bytes, and sends it to the host.

8. The host encrypts the challenge using the authentication key, and sends it to the DVD Player.

9. The DVD Player decrypts the encrypted challenge sent by the host, and ensures that it is the same as the challenge sent earlier.

10. The host is now authenticated.

11. The host and the DVD Player add the two decrypted challenges, and encrypt them using the authentication key. The result is the bus key.

This authentication protocol is quite weak, since it relies on a secret key, which is probably stored in the firmware inside DVD players. The encryption and decryption of keys is similar to encryption and decryption of data (described below), except that an S-Box permutation is applied before the encryption, and after the decryption. A different S-Box is used for each type of key.

4.5 Encoding

Encoding and Decoding algorithms in CSS use Linear Feedback Shift Registers (LFSRs). LFSRs are commonly used by cryptographic protocols to generate pseudo-random bits.

4.5.1 Linear Feedback Shift Registers

An LFSR is a shift register that, when clocked, advances the input signal through the register from one bit to the next most significant bit, and shifts a *feedback* bit into the least significant bit. If the initial value of an LFSR is 0, it will produce an output of all 0's. This is referred to as *null cycling*. LFSRs are usually combined using addition, multiplexors or logic gates to generate less predictable bit streams.

Some terms regarding LFSRs that we will need, are given in italics below. An LFSR is *seeded* with an initial value. With each clock tick, *tapped* bits are evaluated by a *feedback function*. The output of the feedback function is shifted into the register at the input. The output of the register is the bit that is shifted out. A generic LFSR is given in figure 4.1.

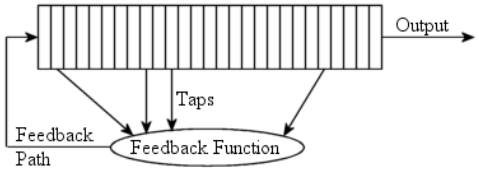

Figure 4.1: Generic LFSR

4.5.2 Encoding Algorithm

The CSS encoding algorithm uses two LFSRs. The first one is a 17-bit LFSR, and the second one is a 25-bit LFSR. We will refer to these LFSRs as LFSR-17 and LFSR-25 respectively. LFSR-17 is seeded with the first two bytes of the 5-byte key. The least significant bit is then set to 1, in order to prevent null cycling. Similarly, LFSR-25 is seeded with the last three bytes of the 5-byte key. The three least significant bits are then shifted up one position, and the fourth least significant bit is set to 1, in order to prevent null cycling. LFSR-17 has 2 taps (bits 1 and 15), and its feedback function is exclusive-OR. LFSR-25 has 4 taps (bits 1, 4, 5 and 15), and its feedback function is exclusive-OR. In both LFSRs, the value of the feedback function is used as the output, and the typical output bit is discarded.

The output from the two LFSRs is combined using 8-bit addition. After each LFSR outputs 8 bits, the outputs may be bitwise inverted before being added to form an output byte. The carry-out from this addition is used as the carry-in for the addition for the next output byte.

CSS has four different output modes. Depending on the mode, the output of one or both of the LFSRs may be inverted bitwise before addition. The modes and the inverter settings are given below:

Mode	LFSR-17	LFSR-25
Authentication	Invert	
Session Key		
Title Key		Invert
Data	Invert	Invert

To encode a sector (plaintext), each LFSR is first seeded with the sector key as described earlier. After the LFSRs are seeded, the output forms a pseudo-random bit stream. The exclusive-OR of the bit stream with the plaintext, is the encoded-text. The plaintext bytes may have an S-Box permutation applied before the exclusive-OR operation. The algorithm described above, is E_B.

4.6 Decoding

Decoding involves the same steps as encoding, except that the optional S-Box transformation must be reversed. First, the same pseudo-random bit stream generated during encryption, is created. Then, it is XORed with the encoded-text. Finally, the S-Box permutation is reversed, to obtain the plaintext. The resulting algorithm is D_B.

4.6.1 Key Decryption

Before the information on a DVD can be decoded, some keys must be decrypted. In addition to the usual encoding, CSS performs a two-step mangling operation for keys, including the title key and the bus key. This process is shown in figure 4.2, where each

column represents one byte of the key. The resulting algorithm is E_A. If we reverse the key mangling step along with applying D_B, we obtain the algorithm D_A.

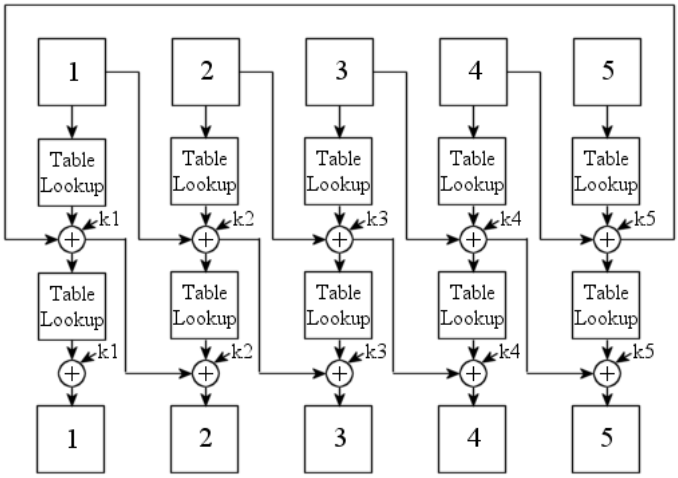

Figure 4.2: CSS Key Mangling

This process can be easily reversed if we have a known plaintext-ciphertext pair. We try all possibilities for $k5$, and work backwards to find $k4, k3, k2$, and $k1$, where $k1, k2, k3, k4$, and $k5$ are the 5 bytes of the key.

4.7 Crack

CSS has several weaknesses that lead to attacks of varying complexities. An obvious weakness in the CSS cipher is that we can simply try all possible 2^{40} disk keys to determine the disk key for a DVD. This is possible because we can guess a disk key K, and then determine whether $E_A(K, K)$ equals the hash of the disk key stored on the DVD. If so, then there is a very high probability of K being the disk key. This is an encoded-text only attack, where the adversary can obtain the complete plaintext corresponding to the given encoded-text.

There is a more efficient attack that attempts to reverse the disk key hash, $E_A(DK, DK)$, and has time complexity 2^{25}. Even faster attacks have been studied, but they are not practical. The 2^{25} attack can recover a disk key from the hash alone, in a matter of seconds. The attack is complex, and does not readily illustrate any obvious weaknesses in the cipher. Hence, it is not discussed here. The attack is presented in [15], and is also an encoded-text only attack, where the adversary can obtain the complete plaintext.

4.8 Discussion

CSS is clearly a very weak DRM scheme. This is not necessarily due to poor design, but due to US government export regulations. When CSS was designed, the US government had severe restrictions on exports of products containing cryptographic schemes. The Data Encryption Scheme (DES) was similarly weakened after it was designed. Its key size was restricted by the US government before it was made an official standard [12]. Even though regulations effectively restricted the sizes of the CSS keys, the scheme is still not as secure as it could have been. This is because a 2^{25} attack can be used to find a 40-bit disk key, which is much faster than an exhaustive search.

The Motion Picture Association of America (MPAA) attempts to gain security by obscurity. It does this by using the legal system to ensure that player keys are not made public. The MPAA is trying to stop the distribution of software called DeCSS that decodes the contents of DVDs, and let users save the plaintexts to disk. The decoded versions can be freely copied and/or distributed. The practice of using the legal system instead of good design to obtain security, is clearly not effective. Even though player keys are not widely available, DeCSS can be downloaded from numerous web sites, and be used to circumvent the CSS DRM scheme.

Even though CSS has been cracked, and export restrictions in USA have been eased somewhat, it will probably continue to be used for years to come. Like audio CD manufacturers, DVD manufacturers must retain backward compatibility. DVDs have already become an internationally accepted standard, and switching to a new DRM scheme for DVDs would require consumers to purchase new hardware. This illustrates a disadvantage of hardware-based DRM schemes, i.e. limited potential for modifications. Advantages and

disadvantages of hardware-based DRM schemes are discussed in the next chapter.

Chapter 5

Addressing DRM Requirements

Now that we have examined two different DRM schemes in detail, we will discuss various approaches to obtaining an effective DRM solution. Technological approaches include hardware and software solutions. Sometimes, the legal system is also used in conjunction with hardware and software solutions to enforce contracts. We will examine each of these approaches, and then discuss why they must be combined to obtain an effective DRM solution.

5.1 Hardware

Hardware-based DRM schemes usually consist of some (possibly tamper-resistant) hardware that only allows a specified set of operations to be performed. An example of this is the hardware component of Microsoft's Palladium operating system, which contains securely stored cryptographic keys, and only allows a few operations such as encryption and decryption, to be performed. The keys are never revealed to anybody, including the owner of the hardware [23]. Another example of a hardware-based DRM scheme is a currently available product that allows protected software to run only if a hardware "key" is present [22]. The "key" is simply a device attached to a serial, parallel, SCSI, or USB port on a computer. This scheme relies on the assumption that the hardware cannot be duplicated. As we saw in the previous chapter, CSS is also largely a hardware-based scheme. There are many other DRM schemes that are hardware-based, or have a hardware component.

DRM schemes based on hardware have several advantages. The hardware can be manufactured in such a way that it cannot be duplicated easily. If information can also be stored "securely" in hardware, we can be reasonably certain that the hardware component of the DRM scheme cannot be cracked. DRM schemes can use such hardware to store keys and other secret information, to ensure that nobody can gain access to them. Another advantage of hardware-based DRM schemes is that they are generally faster than software-based DRM schemes. This is because hardware can generally be optimized more, because it is designed specially for the required tasks.

Hardware-based DRM schemes have several drawbacks, however. The main disadvantage is that they are relatively expensive compared to software-based schemes. They also cannot be modified or updated easily. For example, if a particular key length is no longer considered secure, it would be very difficult, if not impossible, to modify existing hardware designed to store smaller keys to accommodate the new key sizes. Similarly, if a weakness is found in the hardware component of a DRM scheme, it cannot simply be "patched." Each new hardware release carries a significant cost with it.

5.2 Software

Software-based DRM schemes tend to be more flexible than hardware-based DRM schemes, but are usually less secure on current computing platforms. An example of a software-based DRM scheme is Microsoft DRM 2.0, which we examined in detail in Chapter 3. Microsoft's Palladium operating system has a large software component as well, which creates "secure execution environments," where no applications (including the operating system) can obtain any information about protected applications, except for trivial details such as their size [23]. Keys and other secret information can be hidden in software if a secure execution environment is provided.

Software-based DRM solutions have many advantages. They are flexible and easily modifiable. If a weakness is found in the software component of a DRM scheme, a patch to remedy the problem can usually be created and distributed easily. Parameters such as key sizes can be changed easily, in accordance with security standards. Software-based DRM solutions are relatively cheap to create, modify and distribute.

Software-based solutions do have several drawbacks. They tend to be slower and less secure than hardware-based solutions. This is because general-purpose computers currently allow running applications to monitor and interfere with other applications. Techniques such as fault injection have successfully been used to crack some DRM schemes. We will see an example of a fault-injection technique in Chapter 9. Due to the current state of operating systems, anybody can write device drivers that perform operations such as redirect the (plaintext) output meant for a set of speakers, to a file on the hard disk. Another possible weakness in software-based DRM schemes is that one could distribute all relevant software along with the protected information, such that an unauthorized user could access the protected information by impersonating the authorized user. In Microsoft DRM 2.0, this would involve distributing a license for the protected file, and the adversary's public/private keys along with the protected file.

5.3 Legal System

Legal approaches to DRM are very ineffective on their own. However, there are cases where manufacturers have tried to use the legal system to enforce contracts. One example is the Content Scrambling System, which is technologically a very weak DRM scheme. Legal agreements are used to retain control over player keys [19]. CSS descramblers are currently facing legal scrutiny to determine whether their distribution should be allowed to protect freedom of speech, or if it should be disallowed to protect copyrights [13]. Also, the Recording Industry Association of America (RIAA) has been actively searching for, and prosecuting individuals distributing music files illegally [25].

Severe punishments, and an efficient legal system are good deterrents for preventing illegal content distribution. They may also help deter people from attempting to crack DRM schemes. For example, the Digital Millennium Copyright Act (DMCA) introduced in the USA in 1998, created civil and criminal penalties for the creation or distribution of DRM circumvention tools [20]. This has caused some people who discovered cracks for DRM schemes to be prosecuted. Clearly, punishments for offenders can be effective in preventing contract violations, as long as they are accompanied by technologically sound DRM schemes.

The legal system is not an adequate DRM solution by itself, since laws have never been completely effective at preventing any type of crime. Another difficulty is that many users may not even know that they are illegally distributing content using their computers. Many file-sharing programs automatically allow distribution of downloaded files. A user may illegally download some files, but their distribution may be unintentional. Proving a user's intent to distribute files may be difficult in a lot of cases. Another difficulty with using the legal system is that offenders can be located in countries that do not have laws such as the DMCA. A good DRM solution needs to consider the problem globally, and not restrict its focus to a small number of countries.

5.4 Solution

We have briefly examined the advantages and disadvantages of hardware-based, software-based, and legal system-based DRM solutions. It is clear from examining the disadvantages of each that none of the three is an effective DRM solution by itself. An effective DRM solution must use all three components, as we will see below.

Due to the result on the impossibility of code obfuscation presented in the second part of this thesis, it may be impossible to hide secret information such as keys, in software. We can hide such information securely in tamper-resistant hardware, however. Also, we need hardware components to create an effective DRM scheme, since we need to secure the information pathways to output devices. For example, we need to protect audio signals until they are actually output from the speakers, and we need to protect video signals until they are actually displayed on the monitor.

Clearly, a software-only solution cannot accomplish all goals of content protection. We do need a software component in an effective DRM scheme, however. This is because hardware is expensive, and usually cannot be upgraded easily. Chances are that no DRM scheme's implementation would be perfect, and that updates would be necessary to ensure continued security. Updating hardware is much more difficult than updating software, and users cannot be expected to purchase new devices whenever a security flaw is discovered in the old devices. A software component is also usually required to provide an interface for users on computing platforms.

Even if a DRM scheme is fairly secure technologically, it can probably never be perfect, and so, we need a good legal system to ensure that individuals do not attempt to circumvent the DRM scheme and distribute the content. Punishments should be severe enough to discourage people from illegally distributing content. Currently, the Digital Millennium Copyright Act in the USA is fairly effective at achieving its goals, but it has many unwanted side-effects, which we will examine in the next chapter.

Chapter 6

Other DRM-Related Issues

There are several issues related to DRM that we have not yet discussed. We will examine some of the more important ones in this chapter.

6.1 Open Source Software

Open-source software is software to which no one holds exclusive rights, and for which the source code is available to the public. For example, Linux is an open-source operating system. Open-source software has many advantages over other software [8]. Firstly, unlimited improvements of the software are possible, and these improvements can be made by anybody in the world. Secondly, modifying code to run on new hardware is easy. Conversely, if a piece of software is not open-source, and the manufacturer shuts down, or chooses not to support new hardware, there is no easy way of making the software run on the new hardware. Thirdly, the public availability of source code allows programmers from around the world to find and repair bugs. There are many more advantages of open-source software, but the reasons given above illustrate why open-source software is valuable.

DRM schemes and laws endanger the development of open-source software, since open-source software developers usually rely on reverse-engineering to write programs that can interact with hardware [20]. Reverse-engineering of DRM schemes is illegal under the DMCA, which means that open-source versions of existing closed-source DRM scheme implementations cannot be created. An example of the adverse affects of DRM schemes

and laws was the lack of legal DVD-playing software for Linux for several years. Even now, most DVD players for Linux are illegal, and the only one currently considered legal is not an open-source product [7]. The resulting lack of competition on the Linux platform is not desirable from the point of view of consumers.

6.2 Fair Use Rights

"Fair use" rights limit an author's exclusive rights under copyright law to encourage citizens to fully and openly exchange and build upon information to increase the public's knowledge [9]. Fair use refers to an individual's right to use copyrighted materials in a reasonable manner without the consent of the copyright owner. The reason for fair use rights is the belief that not all copying of copyrighted content should be banned. For example, using copyrighted material for the purposes of criticism, comment, parody, news reporting, teaching, scholarship, research, or personal use, are generally considered legal.

One aspect of current DRM schemes that is undesirable, is a lack of recognition of fair use rights. In the past, it was easy for people to exercise their fair use rights, but today, some DRM schemes prevent users from copying information for any purpose. For example, some audio CDs cannot be copied (until somebody discovers a crack for the copy protection scheme), and therefore, a user cannot exercise his/her fair use rights. It seems unlikely that fair use rights can ever be completely recognized by DRM schemes, since it is difficult for a computer to determine the intentions of a person who is copying protected information. Fair use rights probably need to be redefined in accordance with the limitations of DRM schemes, and business models need to be changed to conform to the new definitions. For example, if consumers are not allowed to make copies of a CD for personal use, they should pay a discounted price for it. The business model in this case would change from consumers purchasing one CD and making copies of it for personal use, to them purchasing several original copies of the CD at a much smaller price for each one. Otherwise, consumers would receive less value, which could provide more fuel for anti-DRM campaigns.

6.3 Competition

A competitive market has many advantages for consumers. Firstly, competition generally leads to lower prices, since many companies competing for a market share leads to the companies offering their products at cheaper prices to attract consumers. Secondly, companies have more incentives to improve their products, and offer better value to consumers. Thirdly, competition prevents companies from controlling consumers by forcing them to purchase their products, upgrade regularly, etc.

Secure DRM schemes would have a very serious adverse side-effect. By definition, a DRM scheme restricts the operations that can be performed on protected information, and the applications that can be used to performed those operations. Unfortunately, software manufacturers could use such power to prevent files saved by their software from being opened in software created by their competitors [1]. If the DRM scheme used is secure, reverse-engineering of the file format would be impossible or impractical. This could be used by manufacturers of popular software to dissuade users from converting to their competitors' software, since most users would want to be able to use file formats that are the most popular. This would create monopolies, and switching to different software would become virtually impossible.

6.4 Privacy

Simply put, privacy is the right of people to be left alone. Many individuals may not want their activities to be monitored, and they have a right to disallow any type of monitoring, as long as they are not engaged in illegal activities. Privacy is an important fundamental right, and unreasonable threats to privacy are generally met with overall dissatisfaction from the public. Privacy is possibly the most popular DRM-related issue.

Revisiting the example in the introduction (Chapter 1), where we monitor a user using a video camera, we can see that it can be considered an invasion of privacy. The problem is very real in the digital world. A lot of DRM schemes need to monitor users in order to enforce contracts. For example, consider a user who has paid to watch a movie twice online. The server providing the movie must somehow bind personal information about

the user with the number of times the user has watched the movie. This may not seem like a significant problem, but consider that if the movie contains some inappropriate content, and information about a particular user watching the movie is accidentally or intentionally released to an unauthorized party, the consequences could be serious. In many instances, there is a tradeoff between security and privacy [6]. This problem is analogous to new anti-terrorism laws in some countries. The government's will to monitor individuals conflicts with the citizens' right to privacy. Similarly, the interests of manufacturers of protected information conflict with the consumers' right to privacy.

6.5 Development Cost

An issue related to DRM that is not usually considered is the cost of developing DRM schemes. Once again, we consider audio CDs. Record companies spend millions of dollars to develop copy protection schemes. So far, the trend has been that the schemes get cracked soon after their release, rendering them useless. Therefore, the record companies cannot recover the costs of developing DRM schemes from higher CD sales, and the only way to recover them is by increasing the price of the CDs. Unfortunately, this is exactly what has been done in the past, and the consumers have to pay for the development costs of useless DRM schemes. This is an important issue, since DRM scheme development costs must be paid by somebody, and it is usually not clear who should be responsible for the cost.

6.6 Laws

Laws that are designed to protect DRM schemes and copyrights can have serious side-effects. For example, the Digital Millennium Copyright Act (DMCA) in the USA was designed to stop people from cracking DRM schemes. However, it has been used to stifle legitimate activities. Lawsuits against magazines and threats against researchers are some examples of how it is being used to stifle free speech and scientific research [29]. Other side-effects of the DMCA include some of the problems discussed earlier in this chapter, such as threats against privacy, fair use rights, and open-source software.

Problems such as the above also cost the government a lot of money, since a lot of

unintended lawsuits can result from laws designed for protecting copyrights. Laws such as the DMCA can have many unintended consequences, because it is difficult to find a balance between protecting copyrights, and protecting consumers' rights.

Part II

Code Obfuscation

Chapter 7

Introduction

Informally, an *obfuscator* $\mathcal{O}$ is a compiler that takes a program P as input, and produces a new program $\mathcal{O}(P)$ that has the same functionality as P, yet is "unintelligible" in some sense. The interpretation of the "unintelligibility" condition that we adopt is that we can efficiently compute with oracle access to P, anything that we can efficiently compute using $\mathcal{O}(P)$ [2]. If obfuscators did exist, they would have many applications, including the one that we are most interested in: software protection.

We saw in the first part of this thesis that an adversary has far greater power when attacking DRM schemes, than when attacking traditional cryptographic protocols. This is because the adversary has full control over the execution environment. This more powerful attack model is sometimes called a "white-box" attack model, as opposed to the traditional "black-box" attack model. Since the white-box attack model allows the adversary to view the inner workings of an application, we need to conceal the inner workings such that the adversary cannot gain access to secret information, such as keys. This is exactly what obfuscation attempts to accomplish. It tries to transform a "white-box" application into a "black-box" application, where having the executable code does not reveal anything useful about the inner workings of the application.

Obfuscation has DRM-related applications other than concealing secret information such as keys, as well. Obfuscators can also be used to *watermark* software. A software manufacturer can slightly modify their product's behaviour for each client. The programs can then be obfuscated so that the watermark is difficult to remove [2]. This could help

track illegal distributors of software, and would be very helpful in DRM schemes where users may be able to distribute entire applications along with pirated content, to circumvent the DRM schemes. An efficient legal system, along with the ability to track down offenders, would act as an excellent deterrent against illegal content distribution.

Code obfuscation also has many other cryptographic applications, but they are not relevant to DRM, and we will not discuss them in this thesis. See [2] for these applications.

In Chapter 8, we will present a theoretical proof that an efficient obfuscator cannot exist. It can be shown that even in weaker models than the one that we will examine, an obfuscator cannot exist. The weaker models include ones where the obfuscators are not necessarily computable in polynomial time, where the obfuscators only approximately preserve the functionality of P, and where the model of computation in P is restricted. Most of these results can be proven for Turing Machines, where the input can be of arbitrary length, and for circuits, where the input is of fixed length. We will only examine the simplest result, since most of the results use the same intuitive idea, but with significant technical differences.

Even if a general obfuscator does not exist, it may still be possible to obfuscate specific programs in some meaningful way. In Chapter 9, we will see an example of how this can be done, by examining a white-box DES implementation. We will also examine a crack for the obfuscated DES implementation, and discuss the situations where it may be a practical attack.

Chapter 8

Impossibility of Code Obfuscation

We will show in this chapter that a Turing Machine obfuscator does not exist. Impossibility proofs for circuit obfuscators, approximate obfuscators, etc., are not given here. First, we will present some definitions and notations that we will need, and then we will present the impossibility proof. Most information in this chapter is obtained from [2]. Other impossibility proofs are also contained in [2].

8.1 Definitions and Notations

8.1.1 Notation

We will use the following notations throughout this chapter:

- TM will denote *Turing machine*.
- $PPTTM$ will denote *probabilistic polynomial-time Turing machine*.
- $A^M(x)$ will denote the output of algorithm A on input x, with oracle access to M.
- $A(x;r)$ will denote the output of A on input x and random tape r, if A is a probabilistic TM.
- $A(x)$ will denote the distribution of $A(x;r)$ created by choosing r uniformly.

- $\mu : \mathbb{R}^+ \to \mathbb{R}^+$ will denote a *negligible* function, i.e. a function that grows slower than the inverse of any polynomial. So, for any positive polynomial $p(\cdot)$, there exists $N \in \mathbb{R}^+$, such that $\mu(n) < 1/p(n)$ for any $n > N$.

An algorithm A's oracle queries are of the form $M(x, 1^t)$, where

$$M\left(x, 1^t\right) = \begin{cases} y & M(x) \text{ halts with output } y \text{ after at most } t \text{ steps} \\ \perp & \text{otherwise} \end{cases}$$

If M is a TM, then the function $\langle M \rangle : \{0,1\}^* \times 1^* \to \{0,1\}^*$ is given by:

$$\langle M \rangle \left(x, 1^t\right) = \begin{cases} y & M(x) \text{ halts with output } y \text{ after at most } t \text{ steps} \\ \perp & \text{otherwise} \end{cases}$$

8.1.2 Adversarial Goals

We informally defined obfuscators in Chapter 7 by stating that anything that an adversary can compute from an obfuscation $\mathcal{O}(P)$ of a program P, he/she can also compute given just oracle access to P. We now need to define what it means for an adversary to successfully compute something. There are several models that can be applied (see [2]), but we will adopt the weakest requirement. This weakest requirement is to consider the adversary as trying to evaluate a {0,1}-valued function of the original program P, and require that it is possible, given just oracle access to P, to succeed with roughly the same probability as the adversary does when given $\mathcal{O}(P)$. Clearly, if we cannot satisfy the weakest requirement, we cannot satisfy any stronger requirements either.

8.1.3 TM Obfuscator

A probabilistic algorithm $\mathcal{O}$ is a TM obfuscator if the following three conditions hold [2]:

- Functionality: For every TM M, the string $\mathcal{O}(M)$ describes a TM that computes the same function as M.

- Polynomial Slowdown: The description length and running time of $\mathcal{O}(M)$ are at most polynomially larger than that of M. That is, there is a polynomial p, such that for

every TM M, $|\mathcal{O}(M)| \leq p(|M|)$, and if M halts within t steps on some input x, then $\mathcal{O}(M)$ halts within $p(t)$ steps on x.

- "Virtual Black Box" Property: For any $PPTTM$ A, there is a $PPTTM$ S and a negligible function α, such that for all TMs M,
$\left|Pr\left[A\left(\mathcal{O}\left(M\right)\right)=1\right]-Pr\left[S^{\langle M\rangle}\left(1^{|M|}\right)=1\right]\right| \leq \alpha\left(|M|\right)$.

We say that $\mathcal{O}$ is efficient if it runs in polynomial time.

8.2 Impossibility Result

We defined obfuscators such that they possess the "virtual black box" property when a single program is obfuscated, but the definition does not say anything about what happens when the adversary can inspect more than one obfuscated program. We will extend the previous definition to the case of obfuscating two programs, and then prove that a 2-TM obfuscator cannot exist.

8.2.1 2-TM Obfuscator

First, we need to define a 2-TM obfuscator. A 2-TM obfuscator is defined in the same way as a TM obfuscator, except that the "virtual black box" property is strengthened as follows [2]:

- "Virtual Black Box" Property: For any $PPTTM$ A, there is a $PPTTM$ S and a negligible function α such that for all TMs M, N,
$\left|Pr\left[A\left(\mathcal{O}\left(M\right),\mathcal{O}\left(N\right)\right)=1\right]-Pr\left[S^{\langle M\rangle,\langle N\rangle}\left(1^{|M|+|N|}\right)=1\right]\right| \leq \alpha\left(\min\{|M|,|N|\}\right)$.

Theorem 1 *A 2-TM Obfuscator does not exist.*

Proof: Suppose that there exists a 2-TM obfuscator $\mathcal{O}$. The main idea of this proof (and other impossibility proofs in [2]) is that we can run a program on another program, but we cannot run an oracle on another oracle.

We need a function that cannot be exactly learned using a few oracle queries, since otherwise, we can obtain a complete description of the function in a small amount of time

with only oracle access to it. In this case, there is virtually no difference between having access to the function's description, and having oracle access to the function.

We define the following function, that cannot be exactly learned using a few oracle queries:

$$C_{\alpha,\beta}(x) = \begin{cases} \beta & \text{if } x = \alpha \\ 0^k & \text{otherwise} \end{cases}$$

We assume that $C_{\alpha,\beta}(x)$ runs in $10 \cdot |x|$ steps, where the constant 10 is arbitrary.

Now, we define $D_{\alpha,\beta}$ as follows:

$$D_{\alpha,\beta}(C) = \begin{cases} 1, & \text{if } C(\alpha) = \beta \\ 0, & \text{otherwise.} \end{cases}$$

As defined above, $D_{\alpha,\beta}$ is uncomputable, since it does not limit the running time of $C(\alpha)$. To fix this problem, we choose a polynomial $poly(\cdot)$, and use a modified version of $D_{\alpha,\beta}$ that only considers the execution of $C(\alpha)$ for $poly(k)$ steps, and outputs 0 if $C(\alpha)$ does not halt in that many steps.

Finally, we define Z_k as follows:

$$Z_k(x) = 0^k \text{ for all } x \in \{0,1\}^k$$

We consider an adversary A, that when given two TMs as input, simply runs the second TM on the first one. That is, $A(C, D) = D(C)$. By definition, for all $\alpha, \beta \in \{0,1\}^k$,

$$Pr\left[A\left(\mathcal{O}\left(C_{\alpha,\beta}\right), \mathcal{O}\left(D_{\alpha,\beta}\right)\right) = 1\right] = 1$$

$$Pr\left[A\left(\mathcal{O}\left(Z_k\right), \mathcal{O}\left(D_{\alpha,\beta}\right)\right) = 1\right] = 0$$

So,

$$\left|Pr\left[A\left(\mathcal{O}\left(C_{\alpha,\beta}\right), \mathcal{O}\left(D_{\alpha,\beta}\right)\right) = 1\right] - Pr\left[A\left(\mathcal{O}\left(Z_k\right), \mathcal{O}\left(D_{\alpha,\beta}\right)\right) = 1\right]\right| = 1$$

Note that A is uncomputable, like $D_{\alpha,\beta}$ was. We fix this in the same way that we fixed $D_{\alpha,\beta}$, and restrict the running time of $D(C)$ to $poly(k)$ steps, for a fixed polynomial $poly(\cdot)$. A outputs 0 if D does not halt in that many steps.

Now, any algorithm S that runs in polynomial-time, and has oracle access to $C_{\alpha,\beta}$ and $D_{\alpha,\beta}$ has exponentially small probability (for a random α, β) of querying either oracle where its value is nonzero. So, for all $PPTTM$ S,

$$\left|Pr\left[S^{C_{\alpha,\beta},D_{\alpha,\beta}}\left(1^k\right)=1\right]-Pr\left[S^{Z_k,D_{\alpha,\beta}}\left(1^k\right)=1\right]\right|\leq 2^{-\Omega(k)}$$

This contradicts the fact that $\mathcal{O}$ is a 2-TM obfuscator, since

$$\left|Pr\left[A\left(\mathcal{O}\left(C_{\alpha,\beta}\right),\mathcal{O}\left(D_{\alpha,\beta}\right)\right)=1\right]-Pr\left[S^{C_{\alpha,\beta},D_{\alpha,\beta}}\left(1^k\right)=1\right]\right|\not\leq\alpha\left(\min\{\left|C_{\alpha,\beta}\right|,\left|D_{\alpha,\beta}\right|\}\right)$$

and/or

$$\left|Pr\left[A\left(\mathcal{O}\left(Z_k\right),\mathcal{O}\left(D_{\alpha,\beta}\right)\right)=1\right]-Pr\left[S^{Z_k,D_{\alpha,\beta}}\left(1^k\right)=1\right]\right|\not\leq\alpha\left(\min\{\left|Z_k\right|,\left|D_{\alpha,\beta}\right|\}\right)$$

for any negligible function α.

Note that we restricted the running times of A and $D_{\alpha,\beta}$. We need to ensure that the above equations and inequalities still hold. Executing $A\left(\mathcal{O}\left(C_{\alpha,\beta}\right),\mathcal{O}\left(D_{\alpha,\beta}\right)\right)$ leads to executing
$\mathcal{O}\left(D_{\alpha,\beta}\right)\left(\mathcal{O}\left(C_{\alpha,\beta}\right)\right)$. By definition, this has the same functionality as executing
$D_{\alpha,\beta}\left(\mathcal{O}\left(C_{\alpha,\beta}\right)\right)$, which involves executing $\mathcal{O}\left(C_{\alpha,\beta}\right)$. $\mathcal{O}\left(C_{\alpha,\beta}\right)$ has the same functionality as $C_{\alpha,\beta}$. By the polynomial slowdown property of obfuscators, $\mathcal{O}\left(C_{\alpha,\beta}\right)$ runs in $poly\left(10\cdot k\right)=poly\left(k\right)$ steps, which means that $D_{\alpha,\beta}\left(\mathcal{O}\left(C_{\alpha,\beta}\right)\right)$ need only run for $poly(k)$ steps. Again, applying the polynomial slowdown property, $\mathcal{O}\left(D_{\alpha,\beta}\right)\left(\mathcal{O}\left(C_{\alpha,\beta}\right)\right)$ takes $poly(k)$ steps, which finally implies that A need only run for $poly\left(k\right)$ steps. The same reasoning holds for $A\left(\mathcal{O}\left(Z_k\right),\mathcal{O}\left(D_{\alpha,\beta}\right)\right)$. Note that all polynomials involved are fixed, once we fix the polynomial $p\left(\cdot\right)$ of the polynomial slowdown property in the definition.

Therefore, a 2-TM obfuscator does not exist.

8.2.2 1-TM Obfuscator

Now, we will extend the 2-TM obfuscator impossibility result to the original case. We do this by combining the two programs constructed above into one. For functions or TMs, $f_0, f_1 : X \to Y$, we define $f_0 \# f_1 : \{0,1\} \times X \to Y$ by $\left(f_0 \# f_1\right)(b,x) = f_b\left(x\right)$. Conversely, if we are given a TM $C : \{0,1\} \times X \to Y$, we can decompose C by setting $C_b(x) = C(b,x)$.

Theorem 2 *A TM obfuscator does not exist.*

Proof: Suppose that there exists a TM obfuscator $\mathcal{O}$. Define the following functions:

$$F_{\alpha,\beta} = C_{\alpha,\beta} \# D_{\alpha,\beta}$$

$$G_{\alpha,\beta} = Z_k \# D_{\alpha,\beta}$$

Consider an adversary A that when given a TM $f_0 \# f_1$, decomposes it into f_0 and f_1. If the adversary is given $F_{\alpha,\beta}$, it decomposes it into $C_{\alpha,\beta}$, and $D_{\alpha,\beta}$. If it is given $G_{\alpha,\beta}$, it decomposes it into Z_k and $D_{\alpha,\beta}$. After the decomposition step, the adversary continues exactly as in the proof of Theorem 1. As in the proof of Theorem 1, we have

$$\left| Pr\left[A\left(\mathcal{O}\left(F_{\alpha,\beta}\right)\right) = 1\right] - Pr\left[A\left(\mathcal{O}\left(G_{\alpha,\beta}\right)\right) = 1\right]\right| = 1$$

$$\left| Pr\left[S^{F_{\alpha,\beta}}\left(1^k\right) = 1\right] - Pr\left[S^{G_{\alpha,\beta}}\left(1^k\right) = 1\right]\right| \leq 2^{-\Omega(k)}$$

Hence, as in the proof of Theorem 1,

$$\left|\Pr\left[A\left(\mathcal{O}\left(F_{\alpha,\beta}\right)\right) = 1\right] - Pr\left[S^{F_{\alpha,\beta}}\left(1^k\right) = 1\right]\right| \not\leq \alpha\left(\left|F_{\alpha,\beta}\right|\right)$$

and/or,

$$\left|\Pr\left[A\left(\mathcal{O}\left(G_{\alpha,\beta}\right)\right) = 1\right] - Pr\left[S^{G_{\alpha,\beta}}\left(1^k\right) = 1\right]\right| \not\leq \alpha\left(\left|G_{\alpha,\beta}\right|\right)$$

for any $PPTTM$ S, and any negligible function $\alpha(\cdot)$.

This contradicts the definition of a TM obfuscator. Hence, a TM obfuscator does not exist.

8.3 Discussion

The above impossibility result may make obfuscation seem hopeless, but in reality, it does not have much practical significance. We will see in the next chapter that we can achieve obfuscation in a meaningful way. Even though an "obfuscator" does not exist, we can still obfuscate specific programs by examining the operations that they perform, and finding ways of hiding each one of the operations individually. However, we will also see that the obfuscation scheme described in the next chapter is useful only in certain models.

Specifically, if the adversary gains access to both the encryption and decryption programs for the obfuscated version of DES, then he/she may be able to find the key quickly. This attack can be prevented, and we will discuss how this can be done, in Chapter 9.

Chapter 9

Case Study: Obfuscated DES

In this chapter, we will examine a white-box implementation of DES, where the secret key is hidden inside the implementation such that it cannot be retrieved easily. Clearly, any black-box attack against DES will still apply in this case, but we will examine whether we can obtain the same level of security that we can obtain in the black-box model of computation.

9.1 DES

In this section, we will present the traditional DES algorithm, without any obfuscation techniques applied to it. It will help us understand the obfuscation process better. Most information in this section is obtained from [16].

DES is a special type of iterated cipher known as a *Feistel cipher*. In a Feistel cipher, each state u^r is divided into two halves of equal length, L^r and R^r. The round function g has the form $g\left(L^{r-1}, R^{r-1}, K^r\right) = \left(L^r, R^r\right)$ where

$$L^r = R^{r-1}$$

$$R^r = L^{r-1} \oplus f\left(R^{r-1}, K^r\right)$$

DES is a 16-round Feistel cipher having block length 64. It encrypts a plaintext x of length 64 using a 56-bit key, K, obtaining a ciphertext of length 64. Prior to the 16 rounds of encryption, there is a fixed initial permutation λ that is applied to the plaintext. We denote $\lambda(x) = L^0R^0$. After the 16 rounds of encryption, the inverse permutation λ^{-1} is applied. The application of λ and λ^{-1} have no cryptographic significance, and are often ignored during DES security discussions.

Each L^r and R^r has length 32 bits. The function

$$f : \{0,1\}^{32} \times \{0,1\}^{48} \to \{0,1\}^{32}$$

takes as input a 32-bit string (R^{r-1}) and a 48-bit round key (K^r). The key schedule, $s(K) = (K^1, K^2, \ldots, K^{16})$, consists of 48-bit round keys that are derived from the 56-bit key, K. Each K^r is a permuted selection of bits from K.

The f function consists of an S-Box substitution (described later), after a (fixed) permutation, denoted P. Suppose we denote the first argument of f by A, and the second argument by J. Then, in order to compute $f(A, J)$, the following steps are executed.

1. A is "expanded" to a bitstring of length 48 using a fixed expansion function, E. $E(A)$ consists of the 32 bits from A, permuted in a certain way, with 16 of the bits appearing twice.

2. Compute $E(A) \oplus J$ and write the result as the concatenation of eight 6-bit strings, $B = B_1B_2B_3B_4B_5B_6B_7B_8$.

3. The next step uses eight S-Boxes, denoted $S_1, \ldots, S_8$. Each S-Box

   $$S_i : \{0,1\}^6 \to \{0,1\}^4$$

   maps six bits to four bits, and is traditionally depicted as a 4×16 array whose entries are from the integers $0, \ldots, 15$. Given a bitstring of length six, say $B_j = b_1b_2b_3b_4b_5b_6$, we compute $S_j(B_j)$ as follows. The two bits b_1b_6 determine the binary representation of a row r of S_j (where $0 \le r \le 3$), and the four bits $b_2b_3b_4b_5$ determine the binary representation of a column c of S_j (where $0 \le c \le 15$). Then, $S_j(B_j)$ is defined to be the entry $S_j(r, c)$, written in binary as a bitstring of length four. In this fashion, we compute $C_j = S_j(B_j)$, $1 \le j \le 8$.

4. The bitstring $C = C_1C_2C_3C_4C_5C_6C_7C_8$ of length 32 is permuted using the permutation P. The resulting bitstring $P(C)$ is defined to be $f(A, J)$.

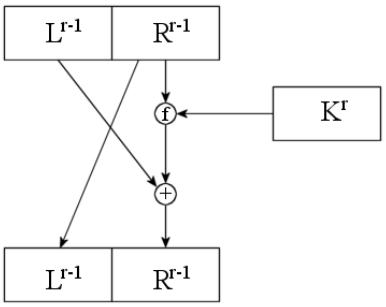

Figure 9.1: One round of DES encryption

One round of DES encryption is shown in Figure 9.1, and the DES f function is shown in Figure 9.2.

Clearly, the DES S-Boxes are not permutations, because the number of possible inputs is 64, and the number of possible outputs is 16. However, it can be verified that each row of each of the eight S-Boxes is a permutation of the integers $0, \ldots, 15$.

9.2 Obfuscated DES

The main goal of the obfuscated DES implementation that we will discuss, is to make key extraction difficult. There are currently no security proofs for it, but there are no known practical attacks against the implementation, either. Obfuscated DES is a lot bulkier and slower than the DES discussed in the previous section. This seems to be a disadvantage of all white-box solutions, since they must protect against many more threats, compared to black-box solutions. We will not discuss efficiency of the obfuscated DES implementation in this thesis. Most information in this section is obtained from [5].

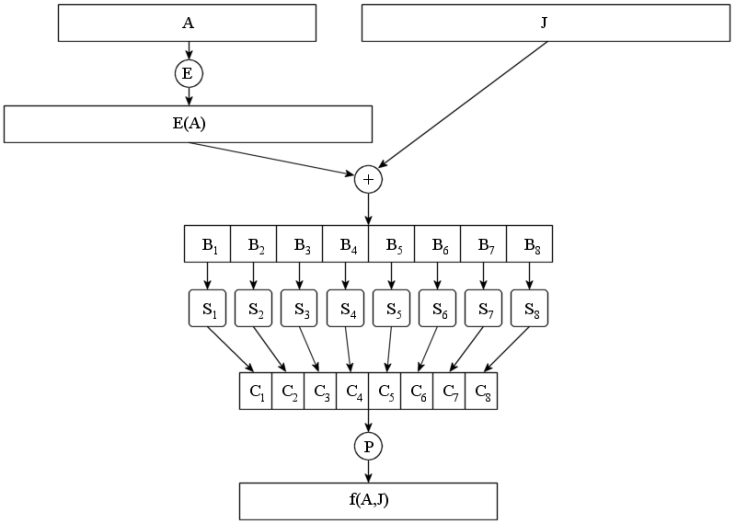

Figure 9.2: DES f function

9.2.1 Definitions and Notations

The main concept that we need, is the *encoding* of a transformation, since it is used extensively in our DES implementation. S-Box lookups, the entire DES, and several other operations are all considered to be transformations. Some definitions that we will need are given below:

Encoding: Let $X : \{0,1\}^m \rightarrow \{0,1\}^n$ be a function. Let $F : \{0,1\}^m \rightarrow \{0,1\}^m$ and $G : \{0,1\}^n \rightarrow \{0,1\}^n$ be bijections. We call $X' = G \circ X \circ F^{-1}$ an *encoded* version of X. F is the *input encoding*, and G is the *output encoding*. For any function X, the encoded version of X is denoted by X'.

Concatenated Encoding: The concatenation of encoding functions $F_1 \| F_2 \| \ldots \| F_k$ (where each F_i is an encoding function) is the bijection F such that for any n-bit vector $b = (b_1, b_2, \ldots, b_n)$, $F(b) = F_1(b_1, \ldots, b_{n_1}) \| F_2(b_{n_1+1}, \ldots, b_{n_1+n_2}) \| \ldots \| F_k\left(b_{n_1+\ldots+n_{k-1}+1}, \ldots, b_n\right)$. Clearly, $F^{-1} = F_1^{-1} \| F_2^{-1} \| \ldots \| F_k^{-1}$.

Networked Encoding: The *networked encoding* for computing $Y \circ X$ is an encoding of the form $Y' \circ X' = (H \circ Y \circ G^{-1}) \circ (G \circ X \circ F^{-1}) = H \circ (Y \circ X) \circ F^{-1}$

${}^{n}_{m}F$ denotes F, emphasizing its input and output sizes. Here, $F : \{0,1\}^m \rightarrow \{0,1\}^n$. ${}^{k}_{k}I$ is the identity function on k bits. An *entropy transference* function is a function ${}^{n}_{m}E$ where if $m \leq n$, then no bits of information are lost, and if $m > n$, then at most $n - m$ bits of information are lost.

$\langle v_1, v_2, v_3, \ldots, v_k \rangle$ is a k-dimensional vector with components v_i. $x \| y$ is the concatenation of vectors x and y. v_i is the i^{th} component of the vector v. $v_{i\ldots j}$ is the subvector of v containing components i through j of v. ${}_{k}v$ denotes explicitly that v has k elements. An *entropy vector*, ${}_{k}e$ is any vector with k elements.

`AT` (an *affine transformation*) denotes a linear function F, which can be defined for all ${}_{m}e$ by ${}^{n}_{m}F\,({}_{m}e) =^{n}_{m} M_m e +_n d$, or $F(e) = Me + d$, where M is a constant matrix and d is a constant *displacement vector*. We will only consider `AT`s over `GF`(2). Note that if A and B are `AT`s, then so are $A \| B$ and $A \circ B$ (where defined).

9.2.2 Producing Encoded Implementations

Our goal is to obtain a DES implementation consisting entirely of S-Boxes, all of which implement non-linear transformations. We need to discuss several techniques before we can obtain such an implementation.

We will apply encodings to all DES operations, including expansions, S-Box lookups, permutations, and XOR operations. For XOR operations and S-Box lookups, simply encoding the operation along with its input and output provides sufficient security [5]. For permutations and expansions, we express them as matrix transformations, and apply anti-sparseness treatments, and then apply non-linear encodings to them, in order to effectively hide information.

We will use the following techniques to achieve our goal:

Partial Evaluation: When we know part of the input to a function at implementation time, we can pre-evaluate all expressions involving only constants and the known part of the input. In DES, we replace standard S-Boxes with key-specific S-Boxes. See section 9.4.2 for a concrete example.

Mixing Bijection: A mixing bijection is a bijective `AT` that attempts to maximize the dependency of each output bit on all input bits. Permutations and Expansions have very

sparse matrices in DES. In order to hide information better, we represent an operation F by $J \circ K$, where K is a mixing bijection, and $J = F \circ K^{-1}$. See section 9.4.2 for a concrete example.

I/O-Blocked Encoding: We cannot represent arbitrary ${}^{n}_{m}F$ (where m is large) using arbitrary bijective encodings as $F' = G \circ F \circ H^{-1}$ using an S-Box representation, since that would require $n2^m$ bits of memory. So, we need to devise a different approach to encoding functions with large input widths. We divide the input into j blocks of a bits each, and the output into k blocks of b bits each, such that $m = ja$, and $n = kb$. Let ${}^{m}_{m}J$ and ${}^{n}_{n}K$ be two mixing bijections. Choose arbitrary encoding bijections for each block of input and output: ${}^{a}_{a}H_1, \ldots, {}^{a}_{a}H_j$, and ${}^{b}_{b}G_1, \ldots, {}^{b}_{b}G_k$. Define $H_F = ({}^{a}_{a}H_1 \| \ldots \| {}^{a}_{a}H_j) \circ J$, and $G_F = ({}^{b}_{b}G_1 \| \ldots {}^{b}_{b}G_k) \circ K$. Then, we encode F as $F' = G_F \circ F \circ H_F^{-1}$. We can represent `AT`s with large input sizes using networks of S-Boxes, as we will see later. Then, if F receives its input from an `AT` X, and sends its output to an `AT` Y, we simply use $J \circ X$ and $Y \circ K^{-1}$ in the implementation instead. In this way, we can ignore J and K during the encoding process. During encoding, we only need to deal with $H_1, \ldots, H_j$ and $G_1, \ldots, G_k$, which can easily be handled using S-Boxes.

Combined Function Encoding: Two functions J and K that are evaluated together can be encoded as $G \circ (J \| K) \circ F^{-1}$. This makes it harder for an attacker to separate and determine the components J and K.

By-Pass Encoding: We usually want extra entropy in the input and output of functions, so that it is difficult for an adversary to identify the transform. For example, we could encode ${}^{n}_{m}F$ as ${}^{n+b}_{m+a}F' = G \circ \left(F \| {}^{b}_{a}E\right) \circ H^{-1}$, where $a \geq b$. ${}^{b}_{a}E$ is called the by-pass component of F'.

Split-Path Encoding: A function ${}^{n}_{m}F$ can be encoded as ${}^{n+k}_{m}Q\,({}_{m}e) = F\,({}_{m}e) \parallel {}^{k}_{m}G\,({}_{m}e)$ for some function G. If F is lossy, Q may lose less information, and be more locally secure.

9.2.3 Notes about S-Boxes and Local Security

When a function F is bijective, the S-Box for F' is locally secure, since it is impossible to extract any useful information from the S-Box. This is because any bijective F could have F' as its encoding. For example, consider the scenario below:

Let F be the following 2-bit bijection:

$$\begin{aligned} 00 &\to 10 \\ 01 &\to 00 \\ 10 &\to 01 \\ 11 &\to 11 \end{aligned}$$

Let G be the following 2-bit bijection:

$$\begin{aligned} 00 &\to 01 \\ 01 &\to 00 \\ 10 &\to 11 \\ 11 &\to 10 \end{aligned}$$

Let H be the following 2-bit bijection:

$$\begin{aligned} 00 &\to 10 \\ 01 &\to 11 \\ 10 &\to 01 \\ 11 &\to 00 \end{aligned}$$

Now, $F' = G \circ F \circ H^{-1}$ is the following function:

$$\begin{aligned} 00 &\to 10 \\ 01 &\to 00 \\ 10 &\to 11 \\ 11 &\to 01 \end{aligned}$$

But, if we select any other bijective F, there exist G and H that give the same result. For example, if F is defined as:

$$\begin{aligned} 00 &\to 01 \\ 01 &\to 11 \\ 10 &\to 10 \\ 11 &\to 00 \end{aligned}$$

Then, if we define G as:

$$\begin{aligned} 00 &\rightarrow 11 \\ 01 &\rightarrow 10 \\ 10 &\rightarrow 01 \\ 11 &\rightarrow 00 \end{aligned}$$

And, if we define H as:

$$\begin{aligned} 00 &\rightarrow 00 \\ 01 &\rightarrow 01 \\ 10 &\rightarrow 11 \\ 11 &\rightarrow 10 \end{aligned}$$

Then, $F' = G \circ F \circ H^{-1}$ is:

$$\begin{aligned} 00 &\rightarrow 10 \\ 01 &\rightarrow 00 \\ 10 &\rightarrow 11 \\ 11 &\rightarrow 01 \end{aligned}$$

which is the same as the previous F'. Similarly, given any other F, we can find G and H that give the same F'. This means that local attacks against such S-Boxes are impossible.

When F is not bijective, however, attacks such as the statistical bucketing attack are possible.

9.2.4 Wide-Input Encoded Affine Transformations

Even though we cannot represent arbitrary wide-input functions using S-Boxes due to large memory requirements, we can construct networks of S-Boxes to implement `AT`s with wide inputs. For an `AT`, we partition its matrix and vectors into blocks, and use well-known formulas for matrix operations using those blocks. We use smaller S-Boxes to encode functions defined by the blocks, and combine the result into a network. For more details on this, see [5].

9.2.5 White-Box DES Implementation Example

We will now construct an embedded, fixed-key DES implementation. The original construction will have some weaknesses that we will address later. In Section 9.4, we will see a concrete toy example that demonstrates some of the obfuscation techniques.

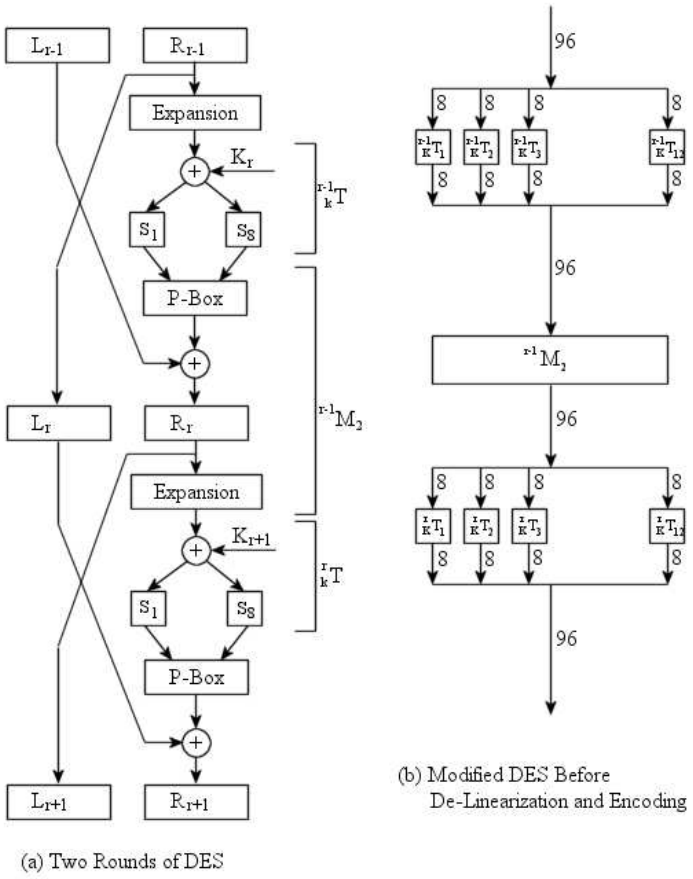

Figure 9.3: Original and Modified DES

Replacing DES S-Boxes: As we saw in section 9.1, each DES S-Box is an instance of ${}^4_6E$. We start with the DES structure in Figure 9.3 (a), and work towards obtaining the implementation in Figure 9.3 (b). Each round is replaced by 12 T-Boxes. Between rounds, the left and right sides are combined into a single 96-bit representation. We use

a single transform ${}^{r}M_2$ to compute the P-Box (DES Permutation), XOR, side flip, and E-Box (DES Expansion). We also use M_1 for the initial expansion of input, and M_3 for the final shrinking of the output. See section 9.4.2 for a concrete example.

Eliminating Explicit Key by Partial Evaluation: We merge the round key into the S-Boxes to obtain key-dependent S-Boxes. We produce S-Boxes denoted by ${}^{r}_{K}S_i$, where K is the DES key, r is the round number, and i is the corresponding S-Box number. This permits us to remove the XOR of inputs with the round key. For a concrete example, see section 9.4.2.

Preparing modified S-Boxes for Local Security: As discussed earlier, we would like to have bijective S-Boxes to obtain local security. We use split-path encoding to achieve this, and let ${}^{r}_{K}T_i\,({}_8e) = {}^{r}_{K}S_i\,({}_8e_{1\ldots6}) \,\|\, R\,({}_8e)$ where $R\,({}_8e) = \langle {}_8e_1,\ {}_8e_6,\ {}_8e_7,\ {}_8e_8 \rangle$. The first six bits of the input of ${}^{r}_{K}T_i$ will be the 6-bit input to the DES S-Box i in round r. We then add two extra input bits. The first 4 bits of the output of ${}^{r}_{K}T_i$ is the output of S_i in round r. The last 4 bits of the output contain the first and last input bit to the corresponding S-Box, followed by the two extra input bits. Note that each T-Box is a bijection, since each "row" of the DES S-Boxes is a permutation [16], and the row number and the extra input bits are simply copied to the output.

Providing 64 bits of By-Pass Capacity: Each ${}^{r}M_2$ requires both the left and right sides of the previous round, which means that it needs 64 bits of by-pass (from the previous round). Each ${}^{r}_{K}T_i$ carries eight bits to the next ${}^{r}M_2$: 4 bits of S-Box output, 2 bits from the right side of the previous round, and 2 bits that we choose to be from the left side of the previous round. Therefore, eight T-Boxes will carry 16 bits of the left side, and 16 bits of the right side to the next round. We need 32 more bits of by-pass capacity. So, we add four more T-Boxes for each round, denoted by ${}^{r}_{K}T_9, \ldots,\ {}^{r}_{K}T_{12}$. Each is a bijective `AT`. This increases the by-pass capacity to 64 bits, as required.

Connecting and Encoding the T-Boxes to Implement DES: The DES implementation immediately prior to de-linearization and encoding is shown in Figure 9.3 (b). The figure will look the same after de-linearization and encoding, but each `AT` will be replaced by a network of S-Boxes, and then, all S-Boxes will have a non-linear encoding applied to them.

Transfer Functions: Due to the transforms involved, the matrices M_1, M_3, and M_2's

are very sparse. We use the method proposed in section 9.2.2 to handle this situation. That is, we use a mixing bijection to remove the sparseness from the matrices. See section 9.4.2 for a concrete example. We will now look at what M_1, M_3, and the M_2's contain.

M_1 contains:

- initial DES permutation: λ
- DES expansion
- delivery of bits to first six inputs of each of ${}^1_K T_1, \ldots, {}^1_K T_8$
- delivery of 16 left side data path bits to be passed through the by-pass provided by inputs 7 and 8 of ${}^1_K T_1, \ldots, {}^1_K T_8$.
- delivery of 32 bits to by-pass provided at randomly chosen positions in ${}^1_K T_9, \ldots, {}^1_K T_{12}$

Each M_2 contains:

- P-Box transform
- XOR of left-side data with the P-Box output
- extraction of right-side of previous round
- DES expansion
- by-pass, as in M_1

M_3 contains:

- P-Box transform
- XOR of left-side data with the P-Box output
- swapping the left and right sides of the data
- inverse of the initial DES permutation: λ^{-1}

Recommended Variants

We described the *naked* variant of white-box DES above. The *recommended* variant applies input and output encodings to the entire DES operation. That is, we replace M_1 by $M_1 \circ M_0$, and M_3 by $M_4 \circ M_3$, where M_0 and M_4 are mixing bijections. This prevents any attacks similar to the one presented in the next section. $M_1 \circ M_0$ and $M_4 \circ M_3$ are implemented as single `AT`s using networks of de-linearized S-Boxes.

There is debate regarding whether this recommended variant is still an implementation of DES. Valid arguments exist for both sides, but we will just note that the recommended variant is very useful for DRM applications. The DRM application could encode its interface to DES in the way described above. This is something that would naturally be done for DRM applications anyway, since plaintext should not be in memory at any point during execution. Otherwise, a simple memory dump would allow the adversary to obtain the plaintext, and bypass the DRM scheme.

9.3 Crack

In this section, we will present an attack on the obfuscated DES described in the previous section. The attack only works on the naked variant of the scheme, and requires that both the encryption and decryption programs be available. This attack is not very practical from the point of view of cracking a DRM application using the obfuscated DES, since an adversary is not likely to possess a copy of the encryption program. Also, if an application implements the scheme properly, the attack does not work. Despite the limitations of the attack, it is interesting to examine from the point of view of code obfuscation, since it shows that obtaining security is very difficult in white-box attack models, and that the nature of attacks can be quite unpredictable.

9.3.1 Notation

We will examine a crack for the naked variant of obfuscated DES in the next section. In this section, we will discuss the notation that we will use in the next section.

P is the DES expansion permutation, and $P_i(x) = P(x)_{6i \ldots (6i+6)}$. π_n^K is the DES

transformation in round n using key K, such that the encryption of plaintext M is $E(M) = \left[\lambda^{-1} \circ \pi_{16}^K \circ \pi_{15}^K \circ \ldots \circ \pi_1^K \circ \lambda\right](M)$. The information (L_r, R_r) immediately after round r of the obfuscated cipher is encoded using the non-linear transformation σ_r. In our explanations, we will remove λ from computations, since it does not play any role in the attack, and can be easily inverted.

9.3.2 Algorithm

Note that this algorithm is a modified version of the algorithm presented in [10]. The version presented in [10] seems to have several errors, which have been corrected here.

1. **Initialization**
   `Set` $L_{16} \leftarrow 0, R_{16} \leftarrow 0$
   `Compute` $\sigma_{15}(L_{15}, R_{15}) = E_{15}^K\left(D^K(L_{16}, R_{16})\right)$
   `Result` $L_{15} = f_{16}^K(0), R_{15} = 0$
   `Derive` $\Omega = \sigma_{15}(L_{15}, R_{15}) = \sigma_{15}\left(f_{16}^K(0), 0\right)$

2. **Reconstruct** $\Delta(x)$
   For $j \leftarrow 0$ to 23
   `Set` $m(j) \leftarrow 0$
   For $i \leftarrow 0$ to 31
   `Set` $L_{16} \leftarrow 2^i, R_{16} \leftarrow 0$
   `Compute` $\sigma_{15}(L_{15}, R_{15}) = E_{15}^K\left(D^K(L_{16}, R_{16})\right)$
   `Set` $\Delta(L_{16}) \leftarrow \sigma_{15}(L_{15}, R_{15}) \oplus \Omega$
   For $j \leftarrow 0$ to 23
   If $\left(\Delta(L_{16})_{4j\ldots(4j+4)} \neq 0\right)$
   `Set` $b[j][m(j)] \leftarrow i$
   `Set` $m(j) \leftarrow m(j) + 1$

For $j \leftarrow 0$ to 23

For $l \leftarrow 0$ to $2^{m(j)} - 1$

Set $e \leftarrow 0$

For $k \leftarrow 0$ to $m(j)$

If $(((l >> k)\,\&1) = 1)$

`Set` $e \leftarrow e + 2^{b[j][k]}$

`Set` $L_{16} \leftarrow e, R_{16} \leftarrow 0$

`Compute` $\sigma_{15}(L_{15}, R_{15}) = E_{15}^{K}\left(D^{K}(L_{16}, R_{16})\right)$

`Set` $\Delta(L_{16}) \leftarrow \sigma_{15}(L_{15}, R_{15}) \oplus \Omega$

3. **Reset** L_{15} **to** $f_{16}^{K}(0)$

For $i \leftarrow 0$ to 31

`Set` $L_{16} \leftarrow 0, R_{16} \leftarrow 2^{i}$

`Compute` $\sigma_{15}(L_{15}, R_{15}) = E_{15}^{K}\left(D^{K}(L_{16}, R_{16})\right)$

`Result` $L_{15} = f_{16}^{K}(2^{i}), R_{15} = 2^{i}$

`Derive` $w \leftarrow \sigma_{15}(L_{15}, R_{15}) \oplus \Omega = \sigma_{15}\left(f_{16}^{K}(2^{i}), 2^{i}\right) \oplus \sigma_{15}(0, 0)$

For x in Δ^{-1}

For $j \leftarrow 0$ to 23

If $\left(\Delta(x)_{4j\ldots(4j+4)} = w_{4j\ldots(4j+4)}\right)$

$w'_{4j\ldots(4j+4)} \leftarrow \Omega_{4j\ldots(4j+4)}$

Else

$w'_{4j\ldots(4j+4)} \leftarrow (w \oplus \Omega)_{4j\ldots(4j+4)}$

`Compute` $(L'_{16}, R'_{16}) = (M'_{3} \circ {}_{K}^{16}T')(w)$

`Result` $L'_{16} \approx f_{16}^{K}(2^{i}) \oplus f_{16}^{K}(0), R'_{16} \approx 2^{i}$

4. **Differential Cryptanalysis to Extract Key for Round Function**

   `Notation:` $l^s = P^{-1}\left(L'_{16}\right)_{4(s-1)\ldots(4(s-1)+4)}, r^s = E\left(R'_{16}\right)_{6(s-1)\ldots(6(s-1)+6)}$

   For $s \leftarrow 1$ to 8

   $d^s \leftarrow 0$

   For $s = 1$ to 8

   For $i \leftarrow 0$ to 31

   `Compute` $c^s[i] : S_s\left(c^s[i]\right) \oplus S_s\left(r^s[i] \oplus c^s[i]\right) = l^s[i]$

   `Compute` $d^s\left[c^s[i]\right] = d^s\left[c^s[i]\right] + 1$

   `Set` $c'^s \leftarrow c^s\left[\max_{i=1}^{m} d^s[i]\right]$

5. **Reconstruct the Original Key**

   $K^{16} \leftarrow c'^1|c'^2|c'^3|c'^4|c'^5|c'^6|c'^7|c'^8$

   `Compute` $s^{-1}\left(K^{16}\right)$ to retrieve 48 bits of the original key

   Perform brute-force search on the remaining 8 bits of the key

In Step 1 of the algorithm, we set L_{16} and R_{16} to 0. Note that we can only do this in the naked variant of obfuscated DES. We then compute $\Omega = \sigma_{n-1}\left(f_{16}^K(0), 0\right)$.

In Step 2 of the algorithm, we map bits in L_{16} to the 24 4-bit bijections used in σ_{16}. We do this by setting R_{16} to 0, and setting L_{16} to 2^i for $i = 1, 2, \ldots, 32$. The attacker builds a table of $\Delta(c) = \sigma_{15}(c, 0) \oplus \sigma_{15}(0, 0)$ for $c = 1, 2, 4, 8, \ldots, 2^{32}$. Now, using the table, the attacker can reconstruct the left-hand side of the input to round 16 in the scenario where the right-hand side of the output is 0. Furthermore, different bits of the left-hand side L_{15} can correspond to the same T-Box, and in this case, the encoding depends on two bits from L_{15}. Determining the original value $L_{15} \oplus f_{16}^K(0)$ given the intermediate representation is just a table lookup.

In Step 3 of the algorithm, we inject faults into the input to round 16, and observe the output. The attacker does not know how the right hand side gets encoded in σ_{15}. In order to get around this problem, the attacker feeds a value x into R_{15} that is different from 0, and then resets L_{n-1} to 0. Finally, L_{16} contains $f_{16}^K(x) \oplus f_{16}^K(0)$.

In step 4 of the algorithm, the attacker extracts the key for round 16 using differential cryptanalysis. To see how step 4 works, note that $R_{16} = R_{15}$, $L_{15} = f_n^K(0)$, and $L_{16} = f_n^K(R_{15}) \oplus f_n^K(0)$. Also, note that $f_n^K(0) = P\left(\left(S_1\| \ldots \|S_8\right)(K)\right)$, and $f_n^K(R_{15}) = P\left(\left(S_1\| \ldots \|S_8\right)\left(E\left(R_{15}\right) \oplus K\right)\right)$. So, $L_{16} = P\left(\left(S_1\| \ldots \|S_8\right)(K)\right) \oplus P\left(\left(S_1\| \ldots \|S_8\right)\left(E\left(R_{15}\right) \oplus K\right)\right)$.

That is, $L_{16} = P\left((S_1\| \ldots \|S_8)\left(K\right) \oplus (S_1\| \ldots \|S_8)\left(E\left(R_{15}\right) \oplus K\right)\right)$, and hence, $P^{-1}\left(L_{16}\right) = (S_1\| \ldots \|S_8)\left(K\right) \oplus (S_1\| \ldots \|S_8)\left(E\left(R_{15}\right) \oplus K\right)$.

In step 5 of the algorithm, the attacker performs a 2^8 brute search for the remaining bits of the DES key.

9.4 Example: Obfuscating Simple Block Cipher and Applying Crack

In this section, we will examine a toy example of a block cipher, obfuscate it using the techniques presented earlier, and then demonstrate the crack in the previous section on it. We will call the cipher Simple Block Cipher (SBC). Obviously, SBC is very simple and insecure, and should never actually be used for encrypting sensitive information.

9.4.1 Description of SBC

SBC has two rounds that are similar to DES rounds. SBC plaintexts and ciphertexts are 8 bits long. The key length is 8 bits, and the round keys are 6 bits long. The encryption algorithm is depicted in Figure 9.4. The components of the algorithm are described below.

The *Expansion* operation is a function $E : \{0,1\}^4 \rightarrow \{0,1\}^6$ such that $E(b_0b_1b_2b_3) = b_0b_1b_0b_2b_3b_2$. If $K = k_0k_1k_2k_3k_4k_5k_6k_7$, then $K_1 = k_0k_1k_2k_4k_5k_6$ and $K_2 = k_1k_2k_3k_5k_6k_7$ are the round keys.

The SBC S-Box S_1 is the same as the DES S-Box S_1:

Input	Output	Input	Output	Input	Output	Input	Output
000000	1110	000001	0000	100000	0100	100001	1111
000010	0100	000011	1111	100010	0001	100011	1100
000100	1101	000101	0111	100100	1110	100101	1000
000110	0001	000111	0100	100110	1000	100111	0010
001000	0010	001001	1110	101000	1101	101001	0100
001010	1111	001011	0010	101010	0110	101011	1001
001100	1011	001101	1101	101100	0010	101101	0001
001110	1000	001111	0001	101110	1011	101111	0111
010000	0011	010001	1010	110000	1111	110001	0101
010010	1010	010011	0110	110010	1100	110011	1011
010100	0110	010101	1100	110100	1001	110101	0011
010110	1100	010111	1011	110110	0111	110111	1110
011000	0101	011001	1001	111000	0011	111001	1010
011010	1001	011011	0101	111010	1010	111011	0000
011100	0000	011101	0011	111100	0101	111101	0110
011110	0111	011111	1000	111110	0000	111111	1101

Finally, the P-Box permutation is a function $P : \{0,1\}^4 \to \{0,1\}^4$ defined by $P(b_0b_1b_2b_3) = b_2b_0b_3b_1$.

9.4.2 Obfuscating SBC

In this section, we will obfuscate SBC using the techniques presented earlier in this chapter.

Replacing SBC S-Box

The SBC S-Box is an instance of ${}^4_6E$. In each round of the obfuscated DES, the S-Box will be replaced by a T-Box. Between rounds, the left and right sides will be combined into a single 12-bit representation, and we will use a single transform to compute the P-Box, XOR, side flip, and E-Box. We will first choose an 8-bit key for our example. We arbitrarily choose $K =$10010110.

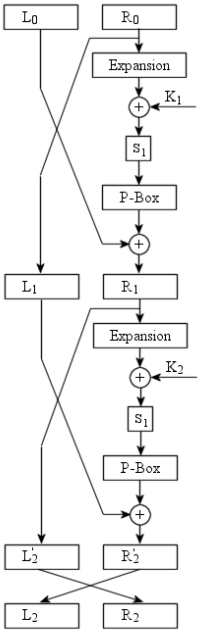

Figure 9.4: Simple Block Cipher Encryption

Eliminating Explicit Key by Partial Evaluation We will first merge our chosen key, and the S-Boxes into new S-Boxes that are dependent on the key and round. Hence, we produce two S-Boxes, identified as ${}^{r}_{K}S_1$, where ${}^{r}_{K}S_1[i] = S_1[i] \oplus K_r$:

For the first round, we obtain ${}^1_K S_1$:

Input	Output	Input	Output	Input	Output	Input	Output
000000	1100	000001	0001	100000	1111	100001	0100
000010	1111	000011	0100	100010	0000	100011	1110
000100	0010	000101	1000	100100	0100	100101	0001
000110	1000	000111	1110	100110	0111	100111	1101
001000	1001	001001	0110	101000	0010	101001	1111
001010	0100	001011	1101	101010	1110	101011	0010
001100	0111	001101	1011	101100	0001	101101	1000
001110	0001	001111	0010	101110	1101	101111	1011
010000	1011	010001	1100	110000	0110	110001	1010
010010	0101	010011	1111	110010	1010	110011	0011
010100	1110	010101	0111	110100	1011	110101	1100
010110	0011	010111	1001	110110	1100	110111	0110
011000	0000	011001	1010	111000	0101	111001	1001
011010	1010	011011	0011	111010	1001	111011	0101
011100	1101	011101	0000	111100	1000	111101	0111
011110	0110	011111	0101	111110	0011	111111	0000

For the second round, we obtain ${}^{2}_{K}S_1$:

Input	Output	Input	Output	Input	Output	Input	Output
000000	1000	000001	0001	100000	1011	100001	0111
000010	1011	000011	1101	100010	0010	100011	0001
000100	1111	000101	0010	100100	0110	100101	1001
000110	0010	000111	1110	100110	1101	100111	0100
001000	0001	001001	0100	101000	1000	101001	0010
001010	1101	001011	0111	101010	1110	101011	1000
001100	0100	001101	1111	101100	0001	101101	1100
001110	1110	001111	0000	101110	0100	101111	1111
010000	0111	010001	1000	110000	0000	110001	1101
010010	0000	010011	0011	110010	0101	110011	0110
010100	1001	010101	0101	110100	1010	110101	0000
010110	0101	010111	1001	110110	0011	110111	1010
011000	1100	011001	1011	111000	0111	111001	1110
011010	0110	011011	1100	111010	1001	111011	0011
011100	1010	011101	0110	111100	1100	111101	1011
011110	0011	011111	1010	111110	1111	111111	0101

Now, we modify the key-specific S-Boxes to obtain local security. We do this by appending the first and last input bits to the end of the output. We call the modified S-Boxes, T-Boxes. The T-Boxes are:

For the first round, we obtain ${}^{1}_{K}T_{1}$:

Input	Output	Input	Output	Input	Output	Input	Output
000000	110000	000001	000101	100000	111110	100001	010011
000010	111100	000011	010001	100010	000010	100011	111011
000100	001000	000101	100001	100100	010010	100101	000111
000110	100000	000111	111001	100110	011110	100111	110111
001000	100100	001001	011001	101000	001010	101001	111111
001010	010000	001011	110101	101010	111010	101011	001011
001100	011100	001101	101101	101100	000110	101101	100011
001110	000100	001111	001001	101110	110110	101111	101111
010000	101100	010001	110001	110000	011010	110001	101011
010010	010100	010011	111101	110010	101010	110011	001111
010100	111000	010101	011101	110100	101110	110101	110011
010110	001100	010111	100101	110110	110010	110111	011011
011000	000000	011001	101001	111000	010110	111001	100111
011010	101000	011011	001101	111010	100110	111011	010111
011100	110100	011101	000001	111100	100010	111101	011111
011110	011000	011111	010101	111110	001110	111111	000011

For the second round, we obtain ${}^{2}_{K}T_1$:

Input	Output	Input	Output	Input	Output	Input	Output
000000	100000	000001	000101	100000	101110	100001	011111
000010	101100	000011	110101	100010	001010	100011	000111
000100	111100	000101	001001	100100	011010	100101	100111
000110	001000	000111	111001	100110	110110	100111	010011
001000	000100	001001	010001	101000	100010	101001	001011
001010	110100	001011	011101	101010	111010	101011	100011
001100	010000	001101	111101	101100	000110	101101	110011
001110	111000	001111	000001	101110	010010	101111	111111
010000	011100	010001	100001	110000	000010	110001	110111
010010	000000	010011	001101	110010	010110	110011	011011
010100	100100	010101	010101	110100	101010	110101	000011
010110	010100	010111	100101	110110	001110	110111	101011
011000	110000	011001	101101	111000	011110	111001	111011
011010	011000	011011	110001	111010	100110	111011	001111
011100	101000	011101	011001	111100	110010	111101	101111
011110	001100	011111	101001	111110	111110	111111	010111

Now, we need to add another T-Box in every round, so that we can provide the required total 8 bits of by-pass capacity. Hence, we obtain the following T-Box for ${}^{1}_{K}T_2$ and ${}^{2}_{K}T_2$:

Input	Output	Input	Output	Input	Output	Input	Output
000000	000000	000001	000001	100000	100000	100001	100001
000010	000010	000011	000011	100010	100010	100011	100011
000100	000100	000101	000101	100100	100100	100101	100101
000110	000110	000111	000111	100110	100110	100111	100111
001000	001000	001001	001001	101000	101000	101001	101001
001010	001010	001011	001011	101010	101010	101011	101011
001100	001100	001101	001101	101100	101100	101101	101101
001110	001110	001111	001111	101110	101110	101111	101111
010000	010000	010001	010001	110000	110000	110001	110001
010010	010010	010011	010011	110010	110010	110011	110011
010100	010100	010101	010101	110100	110100	110101	110101
010110	010110	010111	010111	110110	110110	110111	110111
011000	011000	011001	011001	111000	111000	111001	111001
011010	011010	011011	011011	111010	111010	111011	111011
011100	011100	011101	011101	111100	111100	111101	111101
011110	011110	011111	011111	111110	111110	111111	111111

So, if $L_i = l_0l_1l_2l_3$, and $R_i = r_0r_1r_2r_3$, then r_0 and r_2 are already carried to the next round in ${}^{i}_{K}T_1$. For the remaining bits, ${}^{i}_{K}T_2(l_0l_1l_2l_3r_1r_3) = l_0l_1l_2l_3r_1r_3$. We will switch the order of the T-Boxes in both rounds. In general, the T-Boxes can be arranged in any order in any of the rounds.

We now need to find the matrices for M_1, M_2 and M_3. M_1 performs the E expansion, and then reorganizes the bits to deliver them to the T-Boxes. Note that

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

The following matrix, F, reorganizes the bits to deliver them to the T-Boxes.

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

So, the matrix corresponding to the operation M_1 is:

$$M_1 = FE = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Now, the matrix corresponding to the permutation P (on the 12-bit intermediate representation) is:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

The XOR operation with the left side is:

$$X = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, we set the left side equal to the right side from the previous round using the following matrix:

$$S = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Now, we need to apply the E expansion.

$$E = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Finally, we distribute bits to the correct T-Boxes using F:

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Therefore, the complete transformation M_2 is:

$$M_2 = FESXP = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Finally, we have to determine M_3, which consists of first applying P, then X, and then, S. Then, we shrink the output to its 8-bit form, and swap the two halves. The matrix for shrinking the output is:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

The matrix for the final swap is:

$$W = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

So, the matrix M_3 is:

$$M_3 = WHSXP = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Hence, the complete transformation is: M_1, followed by the two T-Boxes for the first round, followed by M_2, followed by the two T-Boxes for the second round, followed by M_3. Now, we need to apply a non-linear encoding to each of the transformations to make key extraction difficult. We will use a very insecure approach for doing this, but it will be sufficient for our purposes. We will use the following encoding g for both the input and

the output:

$$\begin{aligned} 000 &\rightarrow 011 \\ 001 &\rightarrow 001 \\ 010 &\rightarrow 100 \\ 011 &\rightarrow 000 \\ 100 &\rightarrow 111 \\ 101 &\rightarrow 110 \\ 110 &\rightarrow 010 \\ 111 &\rightarrow 101 \end{aligned}$$

We will apply the encoding to the two 3-bit halves of each T-Box. We will apply anti-sparseness treatments to M_2 and M_3 only. We will use the following matrices to encode M_2 and M_3:

$$Z = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$Z^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Hence, we get the following encoded matrices:

$$M_2' = M_2 Z^{-1} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

$$M_3' = M_3 Z^{-1} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \end{bmatrix}$$

We then apply g to the three-bit halves of the T-Box outputs, g^{-1} to the T-Box inputs, and apply g and g^{-1} wherever necessary to match the input/output encodings of the T-Boxes. If we let ${}^1T = {}^1_K T_2 \| \, {}^1_K T_1$, and ${}^2T = {}^2_K T_2 \| \, {}^2_K T_1$, and $g_c = g \| g \| g \| g$, then the complete transformation is:

$$\left(M_3' \circ g_c^{-1}\right)\left(g_c^{-1} \circ Z \circ g_c^{-1}\right)\left(g_c \circ \ {}^2T \circ g_c^{-1}\right)\left(g_c \circ M_2' \circ g_c^{-1}\right)\left(g_c^{-1} \circ Z \circ g_c^{-1}\right)\left(g_c \circ \ {}^1T \circ g_c^{-1}\right)\left(g_c \circ M_1\right)$$

Here, the transformations inside a set of parentheses are implemented as one set of S-Boxes.

9.4.3 Cracking Obfuscated SBC

In this section, we will apply the crack described earlier in this chapter, on SBC.

1. **Initialization**
   $\Omega = 100011011011$

2. **Reconstruct** $\Delta(x)$
   $\Delta(0001) = 100111011011 \oplus 100011011011 = 000100000000$
   $b[1][0] = 0$
   $\Delta(0010) = 000011011011 \oplus 100011011011 = 100000000000$
   $b[0][0] = 1$
   $\Delta(0100) = 011011011011 \oplus 100011011011 = 111000000000$
   $b[0][1] = 2$
   $\Delta(1000) = 010011011011 \oplus 100011011011 = 110000000000$
   $b[0][2] = 3$

$\Delta(0110) = 001011011011 \oplus 100011011011 = 101000000000$
$\Delta(1010) = 101011011011 \oplus 100011011011 = 001000000000$
$\Delta(1100) = 111011011011 \oplus 100011011011 = 011000000000$
$\Delta(1110) = 110011011011 \oplus 100011011011 = 010000000000$

3. **Reset L_{n-1} to $f_n^k(0)$**
   $R_n = 0001$
   $w \leftarrow 101001011100 \oplus 100011011011 = 001010000111$
   $w' \leftarrow 100001011100$
   $L'_n = 1010, R'_n = 0001$
   $R_n = 0010$
   $w \leftarrow 111011011110 \oplus 100011011011 = 011000000101$
   $w \leftarrow 100011011110$
   $L'_n = 1100, R'_n = 0010$
   $R_n = 0100$
   $w \leftarrow 110010100011 \oplus 100011011011 = 010001111000$
   $w \leftarrow 100010100011$
   $L'_n = 1110, R'_n = 0100$
   $R_n = 1000$
   $w \leftarrow 100011110011 \oplus 100011011011 = 000000101000$
   $w \leftarrow 100011110011$
   $L'_n = 0000, R'_n = 1000$

4. **Differential Cryptanalysis to extract key for round 2**
   $c^1[0] = 001110$
   $c^1[1] = 001110$
   $c^1[2] = 001100$
   $c^1[3] = 001110$
   $c'^1 = 001110$

5. **Reconstruct the original key**
   The original key is $x001y110$.
   Brute force search to find remaining two bits results in $x = 1$ and $y = 0$.

Hence, the key is 10010110.

9.5 Discussion

In this chapter, we described an obfuscated version of DES in detail, and presented a crack for it. The *recommended variant* of the obfuscated DES seems very useful for DRM applications. If we consider the *naked variant*, we can see that it is not desirable for DRM applications. This is because the naked variant causes plaintext to be in memory after decryption, which would cause a major weakness in a DRM scheme. In a DRM scheme, we always want the content to be somehow encoded in memory, so that the adversary cannot simply obtain the plaintext by performing a memory dump. Also, in a DRM scheme, the adversary is very unlikely to possess the encryption program. We can usually safely assume that the adversary does not have such power. So, we can probably use the recommended variant safely in a DRM application.

This chapter shows that despite the fact that an obfuscator does not exist, we can still obtain obfuscation in meaningful ways in practice. Just because there is no generic way of obfuscating all programs, does not mean that we cannot obfuscate specific programs in different ways. Such obfuscation may not be perfect, and some attacks against it may be possible, but the lack of practical attacks against the above obfuscation scheme makes the topic of code obfuscation seem promising.

Part III

Conclusion

This thesis provided a broad overview of DRM, and examined code obfuscation in some detail. In the first part of the thesis, we started by providing an informal introduction to DRM in Chapter 1. We then attempted to define DRM formally. The definitions given in Chapter 2 are not complete by any means, but due to the current lack of formal definitions, we need a start. In Chapter 3, we examined Microsoft DRM 2.0 in detail, and discussed its implementation, and a crack for it. We saw how one simple weakness in a DRM scheme can lead to a very efficient crack. In Chapter 4, we examined the Content Scrambling System in detail, and saw how it is technologically a very weak DRM scheme. We also saw how the legal system can be incorporated into a DRM scheme. In Chapter 5, we discussed how none of software, hardware, or the legal system can provide an adequate DRM solution, and how each must be present as a component of a good DRM scheme. In Chapter 6, we examined several issues related to DRM, such as privacy and side-effects of the legal system.

In the second part of the thesis, we examined code obfuscation in some detail. In Chapter 7, we provided an informal introduction to code obfuscation. In Chapter 8, we presented a theoretical proof that a Turing Machine obfuscator cannot exist. We noted that this does not imply that we cannot obfuscate specific programs in some meaningful way. In Chapter 9, we examined an obfuscated version of DES in detail, and presented a crack for it. We noted that the crack is not applicable if the obfuscation is applied properly, and is probably impossible to execute on DRM schemes (since the crack requires the encryption program).

We attempted to provide an overview of major DRM-related issues, but our overview is certainly not complete. It would be virtually impossible to cover all DRM-related issues in one paper, since it is such a broad topic. We did examine a lot of important DRM-related material, however. An example of a topic that we did not cover, is *watermarking*. Watermarks employ software and the legal system, and can be useful in preventing contract violations by helping to identify illegal distributors of content. We also did not examine implementation details, such as building secure hardware in this thesis.

We did not discuss one possible solution to the problem of illegal music sharing on the Internet (a very popular DRM topic), that is currently being experimented with. This solution attempts to discourage illegal copying of music by changing the business model

for selling music. It provides high quality unprotected music files for a relatively low price. Consumers are then faced with the choice of downloading music illegally, which might require sorting through files of poor quality, or downloading it legally for a small fee, and ensuring a quick download of a high quality file. Whether most consumers would be willing to pay a small fee for such a service remains to be seen. If this type of business model is successful, DRM may lose some of its importance, since it will not be needed to protect some types of intellectual property.

DRM is a fairly new topic, and there has not been a lot of research done on it. Research in many fields needs to be done before we can design a good DRM solution. We need to examine formal definitions, business models, legal issues, privacy issues, and much more before we can successfully design effective DRM schemes. This thesis attempts to provide an overview of these topics.

Code obfuscation is a topic of interest in DRM, since it would be very desirable to have the ability to securely hide keys and other secret information in software. This is because software tends to be more flexible than hardware, and if security flaws are ever found in a DRM scheme, the software component can be updated easily. We know that an obfuscator cannot exist, but we need to determine what levels of obfuscation can be achieved in practical situations. For example, the obfuscated DES scheme presented in chapter 9 seems to be sufficient for use in a DRM scheme, because there are no known efficient cracks for it that would be applicable for cracking DRM schemes. In this case, a meaningful level of obfuscation can be achieved because the program to be obfuscated can be represented as a series of encoded lookup tables.

A lot of further research in the fields of DRM and code obfuscation is required, before we can understand these topics well.

Bibliography

[1] Anderson, R. (2003, August) *'Trusted Computing' Frequently Asked Questions*. Retrieved November 4, 2003, from
`http://www.cl.cam.ac.uk/ rja14/tcpa-faq.html`

[2] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S. Yang, K., *On the (Im)possibility of Obfuscating Programs*, pp. 1-18 in: Advances in Cryptology - Crypto 2001 (LNCS 2139), Springer-Verlag, 2001.

[3] Borisov, N., Chew, M., Johnson, R., Wagner, D. (n.d.) *Cryptanalysis of Multiswap*. Retrieved November 4, 2003, from
`http://www.cs.berkeley.edu/~rtjohnso/multiswap/`

[4] Borland, J. (2001, January 21) *Big Blue Touts New Napster-proof Music Locks*. Retrieved November 4, 2003, from
`http://news.com.com/2100-1023-251270.html?legacy=cnet`

[5] Chow, S., Eisen, P., Johnson, H., van Oorschot, P., *A White-Box DES Implementation for DRM Applications*, Proceedings of DRM 2002 2nd ACM Workshop on Digital Rights Management, November 18, 2002.

[6] Cohen, J. (2003) *DRM and Privacy*. Retrieved November 4, 2003, from
`https://www.law.berkeley.edu/institutes/bclt/drm/papers/`
`cohen-drmandprivacy-btlj2003.html`

[7] Costello, S. (2000, November 15) *Legal Linux DVD Player on the Horizon*. Retrieved November 4, 2003, from
http://www.cnn.com/2000/TECH/computing/11/15/linux.dvd.idg/index.html

[8] Gonzalez-Barahona, J. (2000, April 24) *Advantages of Open Source Software*. Retrieved November 4, 2003, from
http://eu.conecta.it/paper/Advantages_open_source_soft.html

[9] Gross, R. (n.d.) *Understanding Your Rights: The Public's Right of Fair Use*. Retrieved November 4, 2003, from
http://www.eff.org/cafe/gross1.html

[10] Jacob, M., Boneh, D., Felten, E., *Attacking an Obfuscated Cipher by Injecting Faults*, pp. 16-31 in: Digital Rights Management: ACM CCS-9 Workshop (LNCS 2696), Springer-Verlag, 2003.

[11] Kesden, G. (2000, December 6) *Content Scrambling System (CSS): Introduction*. Retrieved November 4, 2003, from
http://www-2.cs.cmu.edu/~dst/DeCSS/Kesden/

[12] Leong, P. (2002, February 16) *Data Encryption Standard (DES)*. Retrieved December 2, 2003, from
http://www.cse.cuhk.edu.hk/~phwl/ceg5010/des.pdf

[13] Morris, S. (2000, November) *The Tension Between Free Speech and Copyright*. Retrieved November 4, 2003, from
http://www.gigalaw.com/articles/2000-all/morris-2000-11-all.html

[14] Screamer, B. (2001, October 18) *Microsoft's Digital Rights Management Scheme - Technical Details*. Retrieved November 4, 2003, from
http://www.nanocrew.net/software/FreeMe/

[15] Stevenson, F. (1999, November 13) *Cryptanalysis of Contents Scrambling System*. Retrieved November 4, 2003, from
http://www-2.cs.cmu.edu/~dst/DeCSS/FrankStevenson/analysis.ps

[16] Stinson, D. (2002). *Cryptography: Theory and Practice*. Boca Raton: Chapman & Hall/CRC.

[17] *A1Chips: Mod Chip FAQ* (n.d.). Retrieved November 4, 2003, from http://www.a1chips.com/a1-modchipfaq.htm

[18] *CD/DVD Protections* (2003). Retrieved November 4, 2003, from http://www.cdmediaworld.com/hardware/cdrom/cd_protections.shtml

[19] *CSS Demystified* (n.d.). Retrieved November 4, 2003, from http://www-cse.stanford.edu/class/cs201/projects-99-00/dmca-2k/css.html

[20] *Digital Rights Management and Privacy* (2003, August 11). Retrieved November 4, 2003, from http://www.epic.org/privacy/drm/default.html

[21] *Dion's CD Can Crash PCs* (2002, April 5). Retrieved November 4, 2003, from http://news.bbc.co.uk/2/hi/entertainment/1912466.stm

[22] *EVERKEY Copy Protection System (Hardware Based)* (n.d.). Retrieved November 4, 2003, from http://www.az-tech.com/everkey.htm

[23] *Microsoft Next-Generation Secure Computing Base - Technical FAQ* (2003, July). Retrieved November 4, 2003, from http://www.microsoft.com/technet/treeview/default.asp?url=/technet/security/news/NGSCB.asp

[24] *Microsoft Palladium: A Business Overview* (2003, January 25). Retrieved November 4, 2003, from http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp

[25] *More Lawsuits Filed Against Downloaders* (October 31, 2003). Retrieved November 4, 2003, from

http://www.cnn.com/2003/TECH/internet/10/31/downloading.suits.ap/index.html

[26] *Open Digital Rights Language: A Rights Expression Language for Digital Asset Management and E-Commerce* (2003, October 20). Retrieved November 4, 2003, from http://odrl.net/docs/ODRL-brochure.pdf

[27] *Satellite TV Station Wages War on Pay-Per-View Pirates* (2001, January 27). Retrieved December 2, 2003, from http://www.ananova.com/news/story/sm_187692.html?menu=sport.gridiron

[28] *TCPA: Trusted Computing Platform Alliance* (2003). Retrieved November 4, 2003, from http://www.trustedcomputing.org/home

[29] *Unintended Consequences: Five Years under the DMCA* (2003, September 24). Retrieved November 4, 2003, from http://www.eff.org/IP/DMCA/20031003_unintended_cons.php

[30] *Will CD Copy Protection Backfire?* (2002, June 24). Retrieved November 4, 2003, from http://www.windeler.net/~brahm/archives/000020.html