CRYPTOGRAPHY
2006

THE RISE AND FALL OF DVD ENCRYPTION

by
Kim Rauff Schurmann, 20033033 (rauff@daimi.au.dk)
Claus Andersen, 20030583 (ca@daimi.au.dk)
Jacob Styrup Bang 20030585 (styrup@daimi.au.dk)
Jakob Løvstad Funder 20033047 (funder@daimi.au.dk)

Department of Computer Science - Daimi
University of Aarhus
December 15, 2006

Abstract

The motivation for this project is to understand the failures made in the past in order to avoid repeating them. There were two main reasons for the fall of DVD encryption. One is a very insecure cryptosystem and the other is poor key management. Our focus will be on the former and we will spend a great deal of time describing and analyzing CSS, and the attacks on it. To fully understand and demonstrate those attacks, we will implement two of the attacks.. These are not the only two, but should be enough to understand the weaknesses in CSS

This paper is for a large part based on information retrieved by reverse engineering CSS. This is because the DVD encryption was supposed to be a closed source encryption scheme.

As a closure we will take a short look on the succeeding encryption schemes for HD-DVD and Blu-ray and what have been learned from the failures of CSS.

Contents

Chapter 1

About DVD

1.1 Securing the DVD

The DVD was developed as a successor to the analog VHS (Video Home System) by Toshiba in the early 1990s and after a struggle with other companies the final specifications were released in September 1998.

There are different types of DVDs, 8 cm or 12 cm with either a single or dual layer. The vast majority of DVD movies sold, are on 12 cm dual layer (9.5 GB). Writable DVDs, however, are usually 12 cm single layer (4.7 GB). The data on a video DVD is divided into sectors of 2048 bytes.

Right from the start DVD was implemented with two types of security. The Motion Picture Association of America(MPAA) would like to control where the disc could be played, so that movies would not leak into parts of the world ahead of first showing in cinemas. This was done using region codes, so that a player and a disc must have a region code that matches.

This paper will not go into the region code and the technology behind it.

The other type of security is concerned with making sure that a user is not able to make unauthorized digital copies of the DVDs. This is done by requiring the player and the disc drive to authenticate each other and by encrypting the data on the DVD. We will primarily be concerned with the latter.

1.2 The hidden sector

There is a hidden sector on the DVD that contains the key to decrypt the data on the DVD. This key is known as the disc key. That key itself is encrypted with what is known as a player key. All DVD players (software or hardware) must have a player key to enable it to play a DVD. There are a number of different player keys, so in order to make a DVD playable on different players with different keys, the disc key is encrypted with all the possible player keys. All sources we have found claim that there are 409 different player keys, but we have found that the number is in fact much lower than this. We will return to this later. The hidden sector contains the following 5 byte blocks:

- Disk key encrypted with the it self (referred to as the hash of the disc key)
- Disk key encrypted with player key 1
- Disk key encrypted with player key 2
- ...

It is possible to buy a dual layer writable DVD, but in order to prevent making a direct copy of a movie, those have the hidden area filled with all 0s. This makes it impossible to make a copy of the hidden area and hence decrypt the data on the DVD.

Chapter 2

Mutual authentication

As a part of the security measures used to protect the data on the DVD there is a mutual authentication between DVD drive and the DVD player. This is done to agree on a key[1] that can be used to encrypt data sent on the bus, and to ensure that it is only legal programs that reads the output from the DVD drive.

Ideally this will result in that the data is protected from a man in the middle attack on the bus commutation line. In order to do this, they have a private key as a shared secret prior to the authentication.

The encryption in this part is done using the Content Scrambling System(CSS), used in mode 3[2].

The host, in this case the computer that hosts the DVD drive, request an Authentication Grant ID (AGID). This is done by first invalidating the AGID that was used for the last session and then requesting a new AGID from the drive. The AGID is used as an ID for the session. The AGID is either a value of 0x00, 0x40, 0x80 or 0xC0 which is added on the subsequent communication between the host and the drive. After agreeing on a AGID, the host generates pseudo random bits and sends it as a nonce to the drive, which responds with an encryption of the nonce using the shared secret. There are actually 32[1] different encryptions (often called variants). The difference is a simple permutation of the nonce prior to the encryption. The host then decrypts the respons with the same shared secret, going through all 32 permutations and authenticates the drive, if one of the decryptions matches the nonce sent. The hosts notes which variant was used and the first 5 bytes of the encrypted nonce is called KEY1.

Since this is a mutual authentication, the drive performs the same operations. The drive generates a nonce sends it to the host, the host encrypts it with the variant noted earlier and sends it back. The drive decrypts it and accepts and authenticates the host, if it is equal

[1]Session key
[2]Table 3.1

to the nonce it sent. The first 5 bytes of the encrypted nonce is called KEY2. Now the host and drive have authenticated each other and combined KEY1 and KEY2 by XORing them and then encrypts them using the shared secret. The result is used as a session key. The host is now able to request the hidden sector (more on this later) on the DVD and decrypt it using the session key.

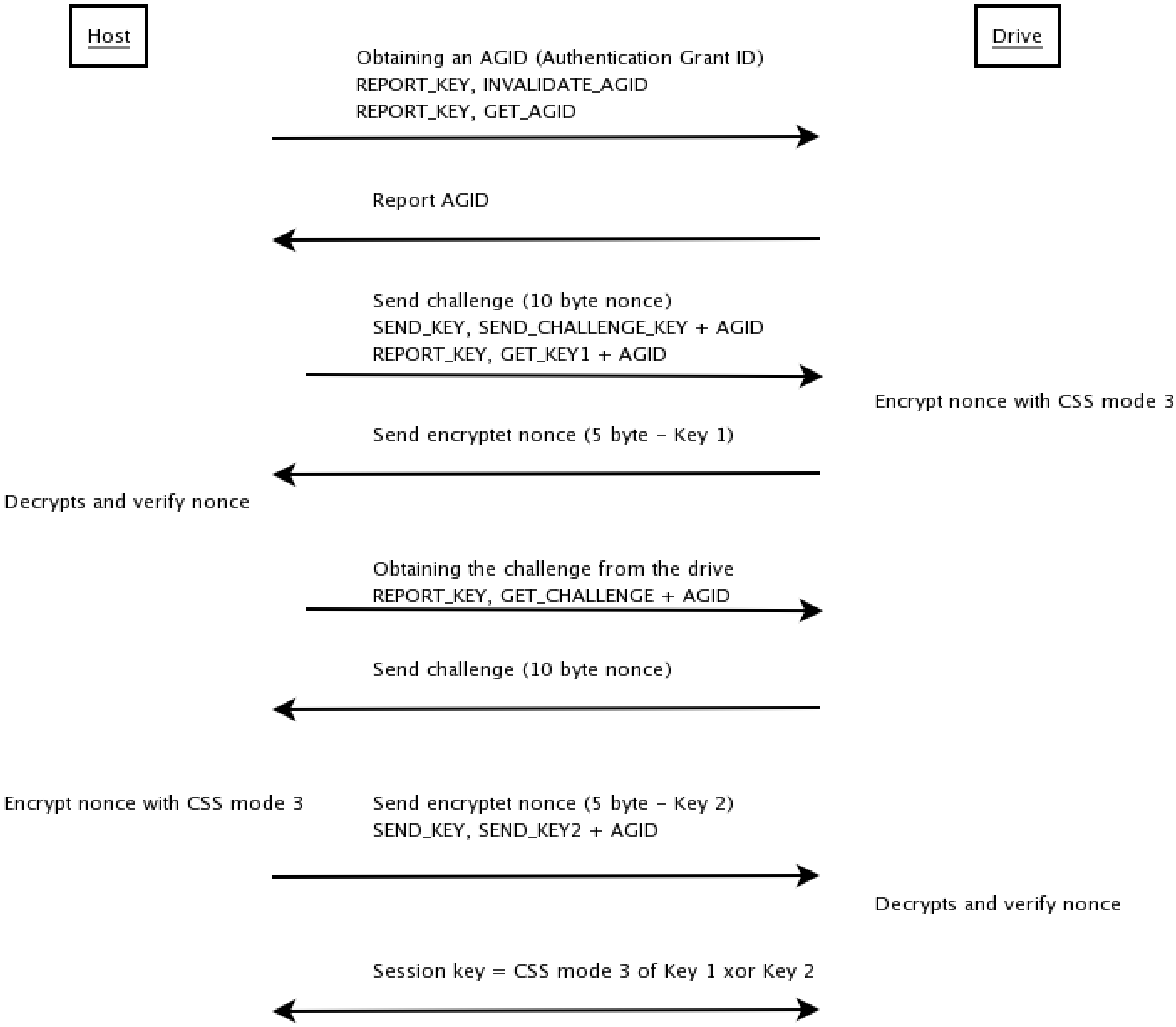

Figure 2.1: The host and drive authentication.

Chapter 3

Description of CSS

CSS is a simple synchronous stream product cipher[8, p. 21] using two Linear Feedback Shift Registers(LFSR)[8, p. 23] as the keystream generator. The set of plaintexts and ciphertexts are any 40 bit string and as we will see so are the keystream alphabet.

We will describe the decryption part of CSS. Encryption is simply doing the reverse function[1] .

3.1 Keys

The DVD encryption/decryption scheme is based on symmetric-key algorithms and utilize a hierarchical key structure. Clearly it should be impossible to gain knowledge of any of the keys, and even if an attack should gain such knowledge, he should not be able to move upwards in the hierarchy.

This section will give a brief overview of the six keys used, all keys are of length 40 bits (5 bytes).[6]

[1] In the case of the LFSRs, this is actually just doing the same with the same key

Figure 3.1: DVD Keys

- Authentication Key: This permanent key is used in the authentication process between the host and the drive, as the shared secret. This key must be in the firmware of all DVD drives[2].
- Session Key: This temporary key is created when the disc is inserted and used to keep the communication on the bus between the host and the drive secret. This prevents someone reading the information in the hidden area from the bus.
- Player Key: This permanent key is licensed to the manufacturer of the player and is stored inside the player. This key is used to decrypt the disc key by methods described later.
- Disc Key: Each DVD has its own unique disc key, which is used to decrypt the title key(s) the same way as the player key was used to decrypt the disc key.
- Title Key: Each title on a DVD[3] has its own unique title key, which is used to decrypt the sector keys by XORing with some specific bytes on each sector (basically a one time pad).
- Sector Key: This key is the last key in the chain and is used to decrypt the actual data on the DVD by methods described below. It is different for each sector.

To verify that the player is allowed to gain access to the disc, it does the following.
Using its player key k_{player} it attempts to decrypt dk_1, which is the disc key K_{disc} encrypted with the first player key.

$$K_{disc} = D_A(dk_1, K_{player})$$

And to verify that K_{disc} is correct its checks the following:

$$K_{disc} = D_A(hash, K_{disc})$$

The check works because $hash = E_A(K_{disc}, K_{disc})$.
If this check fails, the player it will continue with the next encryption of the disc key (dk_i) until it finds the correct disc key.

[2] And is therefore not so private anymore:55, D6, C4, C5, 28[1]

[3] As an example, "Behind the scenes" and the actual movie are two different titles

3.2 Linear Feedback Shift Registers

In general LFSRs are shift registers which are meant to be capable of produce a continuous and apparently random output giving a fixed sized random start state. It is run by shifting all the bits one position to the right, hence pushing a bit out, which is the random output, and put itself into a different state using one or more input bits. The new state is a linear function of the previous state. Because there are a only a finite number of possible states, it will contain cycles - although good LFSRs will have very long cycles and appear to be random. In the case of DVD encryption, we generate random bits using two LFSRs, one with 17 state bits and one with 25 state bits. This seems like a 42 bit encryption scheme, but is really only 40 bit, because two bits are fixed to being 1's, to ensure that the LFSR not go into a cycle of all zeros. [4]

Notice that throughout this section we will refer to the least significant bit (LSB), eg. the rightmost bit, as the first bit in sequence.

The two registers used in DVD encryption are referred to as LFSR-17 and LFSR-25.

3.2.1 LFSR-17

This register is initialized with the first 2 bytes (16 bits) of the 5 bytes (40 bits) in the key, plus an extra bit set to 1. This is bit number 9 (letter i) in the register, making a total of 17 bits. As mentioned before this is done to ensure that it does not return a cycle containing only zeros. Giving the 16 bit key is written out as `jklmnopq abcdefgh`, the start state of the LFSR-17 is `qponmlkj i hgfedcba`. For each round, the LSB (1st bit) is XOR'ed with the 15th bit. The register is shifted one position to the right, and the result bit of the XOR operation is placed in the newly empty tab in the left end at position 17, and is also used as our random output bit.

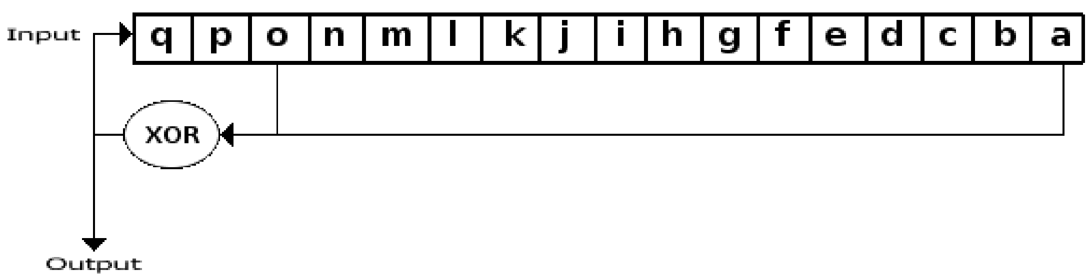

Figure 3.2: LFSR-17 register

3.2.2 LFSR-25

This is basically the same as the LFSR-17. It is initialized with the last 3 bytes (24 bits) of the 5 bytes (40 bits) in key, plus a extra bit a position 22 (letter V) is set to 1. This makes a total of 25 bits. Giving the 24 bit key written out as `QRSTUWXY IJKLMNOP ABCDEFGH`, the start state of the LFSR-25 is `Y XWVUTSRQ PONMLKJI HGFEDCBA`. For each round the bits from position 1, 4, 5 and 13 are XOR'ed. The register is shifted one position to the right, and the result bit of the XOR operation is placed in the newly empty tab in the left end at position 25, and also used as our random output bit.

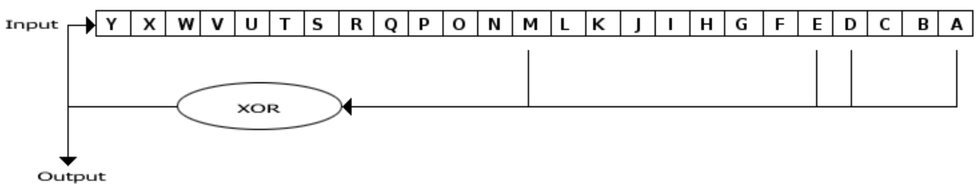

Figure 3.3: LFSR-25 register

3.2.3 LFSR optimization

To speed up the process, it is possible to do 8 rounds at one time. This will produce one byte which is the 8 MSBs. The state after the first 8 rounds of the LFSR-17 is

```
  hgfedcba q ponmlkji
⊕ edcbaqpo 0 00000000
⊕ baqpo000 0 00000000
⊕ po000000 0 00000000
```

and the state of the LFSR-25 it is

```
  HGFEDCBA YXWVUTSRQ PONMLKJI
⊕ KJIHGFED 000000000 00000000
⊕ LKJIHGFE 000000000 00000000
⊕ TSRQPONM 000000000 00000000
```

This can be generalized to every 8bit-round step by again viewing the LFSR states as either `qponmlkjihgfedcba` or `YXWVUTSRQPONMLKJIHGFEDCBA`, before doing the above operations.

3.2.4 LFSR addition (Keystream generation)

The two LFSR's are run eight times each, as shown above. The 8 MSBs of the new states is exactly the bits that are used for output from both of the LFSRs. These bit strings are added together to produce an output byte and a carry bit in case there is an overflow. The carry bit is saved and added to the next output byte. These 8 bits are what we mean when we refer to the total output of the LFSRs. The output of the individual LFSRs will not be used for anything other than to produce this keystream.

To use CSS to both key and data decryption, CSS has 4 modes of operation. This is done by inverting the output of one or both of the LFSRs before the addition. Table 3.1 shows the modes of CSS and if the result of the LFSRs must be inverted.

Mode	LFSR-17	LFSR-25
Authentication	Y	N
Session Key	N	N
Title Key	N	Y
Data	Y	Y

Table 3.1: Invert Output of the LFSRs

3.3 Decryption

There are two different ways of decrypting, depending on whether it is a key or an sector of data that is being decrypted. We assume the reason was that the DVD player has to be able to decrypt large amount of data in very little time, and the it time it takes to go through the many steps involved in decrypting the keys were considered too much to be done in real time when watching a movie. Therefore simpler encryption rules were used the actual data.

3.3.1 Decrypting the sectors (actual data)

First the sector key is decrypted by XORing the title key on the 80th, 81st, 82nd, 83rd and 84th byte of the sector, producing a new 40 bit key. The LFSRs are then initialized with this key vector as described above. The LFSRs are then run 8 times to produce a one byte key for each byte of sector data[4], and is then XORed with the sector byte, producing the plaintext byte. Hence the LFSRs (the keystream generator) combined with the XORing

[4]Starting at the 129th byte of the sector, as the first 128 bytes are plaintext containing information about the sector

(encryption rules) are used as a stream cipher, as shown in figure 3.4 (the first part as mangling is not used when decrypting sectors).

3.3.2 Decrypting the keys

Decryption of the keys uses a more complicated encryption rule. Depending on whether it is the disc key or the title key that is to be decrypted, the LFSRs are seeded with the player or disc key, respectively. The LFSRs are used to produce a total of 5 bytes of output as in figure 3.4. Only one key of 40 bits are produced from the LFSRs and are divided up into one byte keys, which are called mangling keys (mKey[0 ... 4]). The 5 bytes of the encrypted keys are called the hash (hash[0 ... 4]). These bytes are then put through a so-called mangling step pictured in figure 3.5 and produce the state2[0 ... 4] output bytes which are then the decrypted key. The stage1[0 ... 4] bytes are some intermediate results between the first and second step of the decryption and T is a non-linear 8 bit substitution table shown starting on line 6 in tables.h [5].

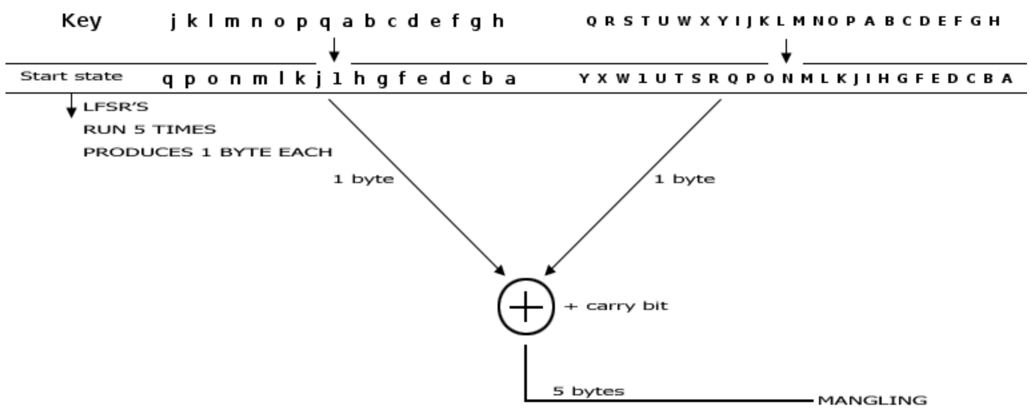

Figure 3.4: Encryption scheme

[5]Appendix section 9.4.1 on page 39

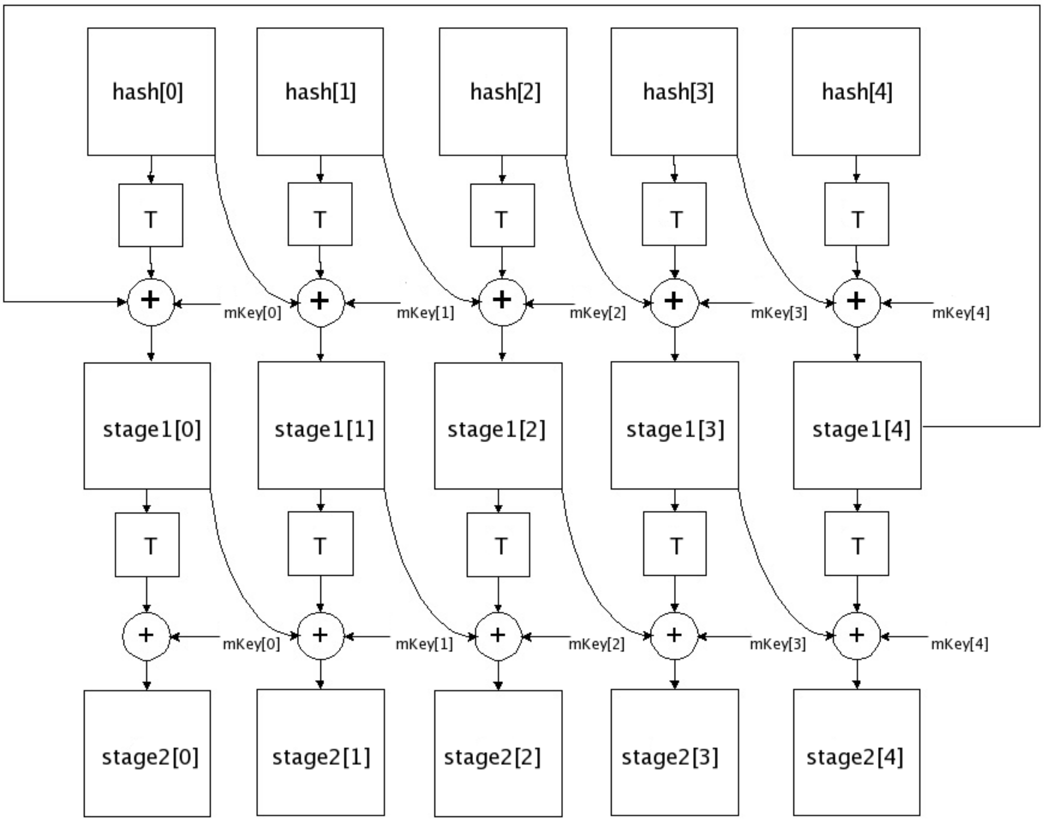

Figure 3.5: Key mangling

Chapter 4

Attacks made on CSS

4.1 A word on time complexity

Time complexity in this chapter is specified as the number of times we have to decrypt a 5 byte key. However, we will not actually be doing entire CSS decryptions in the inner for loops. Instead we will do things that relate to the actual decryption such as reconstructing the start state of the LFSR-25. As such the time complexities should be considered rough estimates.

4.2 Attacking the hashed disc key

4.2.1 Description of Attack

As mentioned earlier, the first 5 bytes of the hidden sector is a hash of the disc key. It's simply the disc key encrypted with the disc key. Using nothing but those 5 bytes it is possible to obtain the disc key. Because of the encryption being non-linear there will be collisions and hence unusually more than one possible disc key (and sometimes none). This is a ciphertext Only Attack[2] without any assumptions about the distribution of the plaintext[1]. It is the special condition that plaintext is actually the key used to encrypt, that enables us to find it in much less time(2^{25}), than what the 40 bit key would suggest. The attack was originally developed by Frank A. Stevenson[5] shortly after the DeCSS source code was released in late October 1999. We will describe his attack and show how we can greatly reduce the (rather high) space complexity of the algorithm. We highly recommend you keep the diagram of the mangling step used for encrypting the keys handy for review (Fig. : 3.5 on the previous page). You might even want to have a pen to cross out the bytes

[1]Except, of course, for the fact that we know it is 5 bytes long.

as the encryption scheme breaks down.

4.2.2 The attack

As input we have hash[0 ... 4] (the disc key encrypted) and we wish to get stage2[0 ... 4] (the disc key in plaintext). First we guess the two first bytes of the disc key. This gives us the bytes stage2[0] and stage2[1]. Then we guess the single byte stage1[0]. This is a total of (2^{24}) possibilities and apart from guessing a single bit later in the process this is all that is needed to find all the bytes of the disc key.

The first 8 bit output of the LFSR's, the mangling mKey[0], can be found as

$$mKey[0] = T[stage1[0]] \ \oplus \ stage2[0]$$

Using the diagram it's also easy to see that,

$$stage1[4] = stage1[0] \oplus T[hash[0]] \ \oplus \ mKey[0]$$

and

$$mKey[4] = stage1[4] \oplus hash[3] \ \oplus \ T[hash[4]]$$

Next step is to aquire mKey[1] from stage1[0] and stage2[1]. Look at the diagram we can see that the following equation is true:

$$stage2[1] = stage1[0] \ \oplus \ mKey[1] \oplus T[hash[0] \ \oplus \ T[hash[1]] \ \oplus \ mKey[1]]$$

Unfortunately T is a non-linear function and it's not clear at all how this could be inverted, to make mKey[1] a function of stage1[0] and stage2[1]. Luckily, however, there's a simple brute force method that only adds 2^{16} time and 2^{16} space. We add an initializing phase before running the algorithm where we create a table we call K1T. The table takes stage1[0] and stage2[1] as index and return mKey[1]. It is made by running through all possible stage1[0] and mKey[1] values, calculating stage2[0] and adding mKey[1] to the index of stage1[0] and stage2[1]. Since T is non-linear we might have multiple results which is why we make room for MAXCOLLISIONS keys at each index. Setting the number to 8 is enough, but this is not something we will prove. Using this table we are now able to get mKey[1]. Looking the diagram we see that

$$stage1[1] = hash[0] \oplus T[hash[1]] \ \oplus \ mKey[1]$$

What is more important is that we have the three bytes mKey[0], mKey[1] and mKey[4]. These are the first, second and last output bytes of LFSR's. We can get all the output bytes we want from the LFSR-17, because we started the algorithm guessing the first 2 bytes of the disc key, and the disc key is encrypted with itself, which means the two bytes are also the first two bytes of the key and the LFSR-17 start state is, as mentioned earlier, derived directly from these 2 bytes. The 1st, 2nd and 5th output byte of the LFSR-25 can be found simply by subtracting the same output bytes of the LFSR-17 from the total output bytes of the LFSR's. There's a small issue concerning the carry, that is solved by adding 1 to the 2nd output byte, if there were overflow on the 1st output byte. Because we don't know anything about the 4th output byte, and hence nothing about a possible carry for the 5. output byte, we are trying both possibilities, giving the total complexity of the algorithm, 2^{25}.

This gives us output byte 1, 2 and 5 from the LFSR-25. As we will show later there is a one to one mapping of these bytes to the start state, and given the start state it is trivial to get there the 3 byte key used for the LFSR-25. Retrieving these 3 bytes is the final piece of the attack, since the entire disc key is the 2 bytes of the LFSR-17, which we are guessing, and these 3 bytes of the LFSR-25. There is a back tracing algorithm to get the start state (except for the highest bit) from the 1st, 2nd and 3rd output bytes of the LFSR-25, but there is no clear way to do this for byte 1,2 and 5. We'll describe two possibilities. The first one is based Frank A. Stevensons attack and the second is our own method which require much less memory.

The big LFSR-25 table

One way of doing it is to repeat the trick for the mKey[1] byte. We simply try all possible keys of the LFSR-25 and compute the corresponding (1st, 2nd and 5th) output bytes. From these bytes it is possible to build a table which have the 3 output bytes as index and the key for the LFSR-25 as value. Finding the key giving the output bytes is a simple look up in constant time. This table is called LFSR-25T in the code. We can add this to the initializing phase, calculating the output bytes 2^{24} times and using $2^{24} * 24 = 48MB$ of space. In the code we use an 32 bit integer type[2] for the key which uses a total of 64MB of memory. One thing to note is that this table is the same for all disc keys, which mean it can be stored for later use.

Getting rid of the big table

The reason for creating the K1T was that we had a non-linear function, but the LFSR-25 is obviously linear (hence the name). We will use this fact to get rid of the big table.

Each bit of the output is directly related to each of the bits of the start state. Using the same convention as on figure **??** all the bits are named using the letters A through Y (you might want to review that section). If we run the LFSR-25 five times and name the output of the first 16 and last 8 bits[3] as b1 to b24, it is possible the relations can be written as:

[2] Only assumed to be true on 32bit platforms

[3] Getting 1st, 2nd and 5th output byte of the LFSR-25

Out bits	Start bits
1st byte	
b1	HKLT
b2	GJKS
b3	FIJR
b4	EHIQ
b5	DGHP
b6	CFGO
b7	BEFN
b8	ADEM
2nd byte	
b9	CFGOPST
b10	BEFNORS
b11	ADEMNQR
b12	MPQY
b13	LOPX
b14	KNOW
b15	JMNV
b16	ILMU
5th byte	
b17	ADEMNOQRUW
b18	MNPQTVY
b19	LMOPSUX
b20	KLNORTW
b21	JKMNQSV
b22	IJLMPRU
b23	HIKLOQT
b24	GHJKNPS

Table 4.1: The 1st, 2nd and 5th output bytes of the LFSR-25 as a function of its keys

Now we have the 3 output bytes expressed as a function of the start state. Unfortunately we want the inverted function, having the start state (or key) expressed as a function of the 3 output bytes, so we need to solve these 24 linear equations. You might have noticed there are 25 unknowns, but we know that V is always 1, which reduce the number to 24. This is easily solved and can be done in much less time than 2^{24} and stored in $O(24^2)$ space. We will not discuss the methods here, but simply state the result, having the bits of the key [4] as a function of the bits of the output.

[4] The 3 byte key for the LFSR-25 is A through Y excluding the V

Key bit	Output bits
Q	1 ⊕ b1 ⊕ b7 ⊕ b9 ⊕ b11 ⊕ b12 ⊕ b17 ⊕ b19 ⊕ b20 ⊕ b22 ⊕ b24
R	1 ⊕ b7 ⊕ b10 ⊕ b11 ⊕ b12 ⊕ b14 ⊕ b22 ⊕ b24
S	1 ⊕ b1 ⊕ b2 ⊕ b3 ⊕ b6 ⊕ b8 ⊕ b9 ⊕ b13 ⊕ b14 ⊕ b15 ⊕ b16 ⊕ b18 ⊕ b19 ⊕ b23
T	b1 ⊕ b3 ⊕ b5 ⊕ b7 ⊕ b9 ⊕ b11 ⊕ b12 ⊕ b14 ⊕ b16 ⊕ b17 ⊕ b21 ⊕ b22
U	b1 ⊕ b3 ⊕ b4 ⊕ b7 ⊕ b8 ⊕ b12 ⊕ b13 ⊕ b16 ⊕ b20 ⊕ b21 ⊕ b22 ⊕ b23 ⊕ b24
W	1 ⊕ b1 ⊕ b2 ⊕ b3 ⊕ b6 ⊕ b7 ⊕ b9 ⊕ b10 ⊕ b11 ⊕ b12 ⊕ b13 ⊕ b14 ⊕ b15 ⊕ b20 ⊕ b21 ⊕ b22 ⊕ b23
X	b1 ⊕ b3 ⊕ b5 ⊕ b6 ⊕ b10 ⊕ b12 ⊕ b14 ⊕ b15 ⊕ b16 ⊕ b18 ⊕ b19 ⊕ b20 ⊕ b22
Y	1 ⊕ b1 ⊕ b2 ⊕ b4 ⊕ b5 ⊕ b10 ⊕ b11 ⊕ b15 ⊕ b17 ⊕ b18 ⊕ b21
I	b3 ⊕ b7 ⊕ b8 ⊕ b9 ⊕ b10 ⊕ b12 ⊕ b13 ⊕ b14 ⊕ b16 ⊕ b17 ⊕ b19 ⊕ b21 ⊕ b22 ⊕ b23 ⊕ b24
J	1 ⊕ b3 ⊕ b8 ⊕ b14 ⊕ b16 ⊕ b18 ⊕ b19 ⊕ b20 ⊕ b21 ⊕ b24
K	1 ⊕ b1 ⊕ b3 ⊕ b7 ⊕ b8 ⊕ b10 ⊕ b11 ⊕ b13 ⊕ b14 ⊕ b16 ⊕ b17 ⊕ b20 ⊕ b21 ⊕ b23
L	1 ⊕ b2 ⊕ b8 ⊕ b9 ⊕ b10 ⊕ b13 ⊕ b14 ⊕ b15 ⊕ b18 ⊕ b20 ⊕ b21 ⊕ b23 ⊕ b24
M	1 ⊕ b2 ⊕ b3 ⊕ b7 ⊕ b9 ⊕ b10 ⊕ b11 ⊕ b12 ⊕ b13 ⊕ b15 ⊕ b16 ⊕ b17 ⊕ b18 ⊕ b20 ⊕ b21 ⊕ b22 ⊕ b23
N	b2 ⊕ b7 ⊕ b11 ⊕ b13 ⊕ b14 ⊕ b15 ⊕ b17 ⊕ b18 ⊕ b19 ⊕ b20 ⊕ b23 ⊕ b24
O	1 ⊕ b1 ⊕ b2 ⊕ b3 ⊕ b7 ⊕ b9 ⊕ b10 ⊕ b11 ⊕ b15 ⊕ b16 ⊕ b17 ⊕ b18 ⊕ b24
P	1 ⊕ b1 ⊕ b2 ⊕ b7 ⊕ b9 ⊕ b10 ⊕ b12 ⊕ b15 ⊕ b17 ⊕ b18 ⊕ b19 ⊕ b21 ⊕ b22 ⊕ b24
A	1 ⊕ b1 ⊕ b9 ⊕ b10 ⊕ b13 ⊕ b14 ⊕ b19
B	b2 ⊕ b3 ⊕ b11 ⊕ b12 ⊕ b15 ⊕ b16 ⊕ b21
C	b1 ⊕ b2 ⊕ b10 ⊕ b11 ⊕ b14 ⊕ b15 ⊕ b20
D	1 ⊕ b1 ⊕ b2 ⊕ b3 ⊕ b8 ⊕ b10 ⊕ b11 ⊕ b14 ⊕ b15 ⊕ b16 ⊕ b18 ⊕ b19 ⊕ b21
E	b1 ⊕ b3 ⊕ b7 ⊕ b8 ⊕ b10 ⊕ b11 ⊕ b14 ⊕ b16 ⊕ b17 ⊕ b20 ⊕ b21
F	b1 ⊕ b3 ⊕ b9 ⊕ b11 ⊕ b12 ⊕ b13 ⊕ b16 ⊕ b19 ⊕ b20 ⊕ b22 ⊕ b23 ⊕ b24
G	1 ⊕ b1 ⊕ b2 ⊕ b8 ⊕ b9 ⊕ b11 ⊕ b12 ⊕ b13 ⊕ b14 ⊕ b15 ⊕ b18 ⊕ b21 ⊕ b22 ⊕ b23
H	b1 ⊕ b2 ⊕ b3 ⊕ b7 ⊕ b9 ⊕ b10 ⊕ b12 ⊕ b14 ⊕ b15 ⊕ b16 ⊕ b17 ⊕ b18 ⊕ b20 ⊕ b22

Table 4.2: The key for the LFSR-25 as a function of the 3 output bytes 1, 2 and 5. The 1s are a result of the 22nd bit always being set to 1

In practice it is rather slow to do all these XORs for every iteration, so instead we build 3 tables, one for each output byte. Each byte has an independent contribution to the key, so we can run through all possibilities for the 1st, calculate its contribution to the key, then repeat this for the 2nd and 5th byte resulting in 3 tables of 2^8 integers[5] and doing only 2^8 iterations for each byte. We call these tables LFSR-25t0, LFSR-25t1 and LFSR-25t4 for the 1st, 2nd, and 5th byte respectively. By comparing the function buildLFSR-25Tables in table.c [6] to table 4.2 it should be clear how these tables are constructed. The key can now be calculated simply by looking up each byte in the corresponding table and XORing the result. This is done on line 148 of the dehash.c[7].

Making 3 tables of 2^8 integers compared to 1 table of 2^{24} is a huge improvement on the space usage.

[5]As with the big LFSR-25 table, only 24 bits are actually needed, but it is convenient to use integers
[6]Appendix section 9.4.2 on page 40
[7]Appendix section 9.2.3 on page 34

4.2.3 Conclusion on the hashed disc key attack

The fact that it is possible to find the disc key in 2^{25} and hence decrypt the entire DVD, shows that the CSS cryptosystem is completely broken. Tests show that our attack on the hashed disc key take less than 2 seconds[8] on a 3 GHz Pentium4 to find possible disc keys (usually between 0 and 3 on random input). Finding the correct key could be done by actually decrypting a DVD and watch the output.

4.3 Attacking the player key

4.3.1 Description of attack

This attack takes as input the 5 byte disc key and the disc key encrypted with a player key, which is also 5 bytes. It can therefore be considered a Known plaintext attack[9] with still no assumptions on the plaintext distribution. As the hashed disc key attack, this attack was also originally developed by Frank A. Stevenson[5] and released about the same time. Apparently he has made an error and switched the ciphertext and plaintext. We will switch it back and make other small modifications that take advantage of the tables LFSR-25t0, LFSR-25t1 and LFSR-25t4 described earlier in section 4.2. Again we recommend keeping the mangling diagram and a pen handy.

4.3.2 The attack

As input we have hash[0 ... 4] (the disc key encrypted) and stage2[0 ... 4] (the disc key in plaintext). We wish to find the player key (or start state of the LFSR's) that was used to encrypt the disc key.

First we guess the value of mangling key mKey[4] and the encryption falls apart like this:

$$stage1[4] = T[hash[4]] \oplus hash[3] \oplus mKey[4]$$
$$stage1[3] = T[stage1[4]] \oplus stage2[4] \oplus mKey[4]$$
$$mKey[3] = stage1[3] \oplus hash[2] \oplus T[hash[3]]$$
$$stage1[2] = T[stage1[3]] \oplus stage2[3] \oplus mKey[3]$$
$$mKey[2] = stage1[2] \oplus hash[1] \oplus T[hash[2]]$$
$$stage1[1] = T[stage1[2]] \oplus stage2[2] \oplus mKey[2]$$

[8]Using the big LFSR-25 it takes about 6 seconds

[9]Note that the plaintext was found with a ciphertext Only Attack

$$mKey[1] = stage1[1] \oplus hash[0] \oplus T[hash[1]]$$
$$stage1[0] = T[stage1[1]] \oplus stage2[1] \oplus mKey[1]$$
$$mKey[0] = stage1[0] \oplus stage1[4] \oplus T[hash[0]]$$

This gives us all the bytes you see on the mangling diagram mangling keys. Using this we will find the start states of the LFSR's. But first, to reduce the number of possible values for mKey[4], we can check if our guess for the mangling key was a good guess by testing if this is true.

$$stage2[0] == (mKey[0] \oplus T1[stage1[0]])$$

There will be some collisions, but practice have shown that there will rarely be more than 3 possible values for mKey[4].

Having all the mangling keys means we have 5 bytes of output of the LFSR's. So for each possible mangling key we will, much like in the previous attack, run through all possible values for the key for the LFSR-17. Giving the key for LFSR-17, we can easily get the start state and produce 5 output bytes. Given 5 bytes of total output of the LFSR's and the output of the LFSR-17, we can find 5 bytes of output of the LFSR-25 simply by subtracting the output of the LFSR-17 from the total output. Repeating the procedure for the attack on the hashed disc key we take the 1st, 2nd and 5th output byte of the LFSR-25 and find its start state. We now produce the 3rd and 4th output byte from the LFSR-25, add it with the LFSR-17 and compare it to the total output of the LFSR's. If they match we save the key as a possible player key.

4.3.3 Conclusion on the player key attack

A player key can be verified by repeating this for several different DVDs and eliminating those that do not repeat. The algorithm runs through all possible keys for the LFSR-17 for each possible mangling key. As mentioned there is rarely more than 3 mangling keys (and usually only 1), so it is fair to claim that the algorithm runs in 2^{16}.

4.4 CSS attacks in practice

4.4.1 Researching the hidden sector

Using a tool called tstdvd 7.1 on page 26 it is possible to authenticate a DVD and download the hidden sector from the DVD and store it to disc. The sector is 2048 bytes making room for a possible 409 different keys of 5 bytes, leaving 3 bytes unused[10]. It has been widely reported that the disc key is encrypted with 409 different player keys on each

[10] $5 \cdot 409 = 2045$

DVD. We have found, however, that this is not true. The sector actually only use mere 32 different player keys, which are the repeated throughout the sector in what seems as a random order[11]. As an example the first player key at position 2 (that is byte 6 through 10 as each position is 5 bytes apart and the first position is the hashed disc key) is repeated at the positions 50, 58, 84, 121, 137, 145, 209, 230, 260, 285, 337, 407. Each encrypted key is repeated 12 or 13 times with position 114, 408 and 409 not containing a key. The reason for this layout is unknown, one possibility is that, if a part of the sector is unreadable, it might be possible to recover the disc key elsewhere in the sector.

4.4.2 Finding player keys

We used five different DVDs in our attempt to find player keys. We used the hashed disc key attack to find each possible disc key for each DVD and then used the player key attack to find possible player keys for each disc key for each DVD. Adding all these player keys together we found that 53 player keys were found on all five DVDs and eight were found on two of them, finding a total of 61 keys listed below in Table 4.3.

It might seem strange that we can find 61 different player keys when only 32 different player keys has been used for encryption. It is possible that two false player keys match just by random collision, but with 2^{40} possible player keys and less than a 1000 being compared it is so unlikely[12] that we do not even consider it. There is, fortunately, another explanation for this is, and it is that there is not a one to one mapping between the start state of the individual LFSR's and the total 5 byte output of the LFSR's. In other words there are sometimes more than one player key that produce the same 5 one byte mangling keys. Only the mangling keys are of importance, when the disc key is decrypted, which mean that any of the player keys that result in the same mangling keys can be used.

Not all keys are on all DVDs, which reduce the number of possible player keys even further, since all DVDs has to be playable on all players. In particular, two DVDs, The Fly (1986) and Pirates of the Caribbean: The Curse of the Black Pearl (2003), had eight player keys that did not appear on any of the other DVDs. We were not able to find any connection between these two movies[13] so the reason for this remains unknown.

[11] Disc key from Pirates of the Caribbean: The Curse of the Black Pearl can be found here: [9]

[12] According to the birthday paradox this probability is $1 - e^{-(1000(1000-1))/(2*2^{40})} = 4.542925 * 10^{-7}$ and we have not found any reason not to think that false player keys are uniformly distributed

[13] Pirates of the Caribbean: The Curse of the Black Pearl is from Walt Disney Pictures, and The Fly is from Brooks films and 20th Century Fox

2	00 58 08 25 D3	5	01 AF E3 12 80	5	12 11 CA 04 3B
5	14 0C 9E D0 09	5	14 71 35 BA E2	5	1A A4 33 21 A6
5	26 EC C4 A7 4E	5	2C B2 C1 09 EE	5	2F 25 9E 96 DD
5	30 52 FE 1D 7D	5	33 2F 49 6C E0	5	35 5B C1 31 0F
5	36 67 B2 E3 85	5	39 3D F1 F1 BD	5	3B 31 34 0D 91
5	45 ED 28 EB D3	5	48 B7 6C CE 69	5	4B 65 0D C1 EE
5	4C BB F5 5B 23	5	51 67 67 C5 E0	5	52 CC 4F BA 12
5	53 94 E1 75 BF	5	54 35 3B AF 4B	5	57 2C 8B 31 AE
5	5F 5F 24 59 EA	5	63 DB 4C 5B 4A	5	69 D2 E3 92 AE
5	6E 4E 9B 31 22	2	6F 8E EA 50 75	5	71 F6 3E 92 CC
2	73 ED 89 7D C6	5	7B 1E 5E 2B 57	5	85 F3 85 A0 E0
5	90 32 62 54 1D	2	90 56 8D 62 C8	2	97 5A 73 EB 6D
5	99 D9 61 44 B8	5	A3 14 69 0E 4C	2	A5 74 B4 8C 86
5	AB 1E E7 7B 72	5	AB 36 E3 EB 76	5	B1 B8 F9 38 03
5	B7 3F D4 AA 14	5	B7 FE 8B 83 24	5	B8 5D D8 53 BD
5	BF 92 C3 B0 E2	2	C6 74 7C 55 B3	5	C9 DD DD DB B1
5	CE FD CA 02 CD	5	CF 1A B2 F8 0A	5	D2 49 27 50 53
5	DB AF 25 67 9D	5	E6 14 D8 28 6E	5	EC A0 CF B3 FF
5	EE C2 7B 19 AD	5	EF 49 73 01 F6	2	F0 1F 04 D6 47
5	F8 BE EE E9 7B	5	FB 9B FC 60 7A	5	FC 95 A9 87 35
5	FE 21 3C 0B C9	-	-	-	-

Table 4.3: Player keys found from 5 different DVDs. The number indicating the number of DVDs on which each player key was found

Chapter 5

CSS issues

As we have just seen, CSS is a weak encryption. This is due to a number of different issues and we will in this chapter explore the most important. Some of them are technical and one of them are based of the more philosophical aspects of creating a secure system.

5.1 Key length

The most clear and obvious problem with CSS is the key length itself. The U.S. Government would not allow a stronger encryption than the already broken DES 56bit encryption, so the engineers settled for a 40 bit key. A key size of 40 bits is not enough to prevent adversaries from brute forcing the key. With only 2^{40} possibilities all keys can be tried in less than 24 hours on a modern computer[1].

5.2 Key management

Another of the major problems with the CSS encryption is the key management. The weakest point of the encryption is often the top key in the hierarchy. The problem is that at some point you cannot do anymore encryption and have to rely on physical, non-cryptographic mean[3, p. 5]. As for the case of DVD, the top key is inside every DVD player, either software or hardware. This is of course a problem that we cannot seem to get rid of because the key obviously needs to be available to every player. Here is a real need for being careful how this key is stored in software or hardware as one compromise will compromise the encryption scheme permanently. They did not put in a way for them to replace the player keys, which mean they were meant to be kept secret forever. Considering the number of people who have access to player keys, it is just unimaginable

[1] We consider a computer with 3 GHz Pentium4 CPU and 1GB of RAM a modern computer

that they can stay secret for that long a period. If you are not careful it can be fairly easy to reverse engineer the software to obtain the player key, as you have the player on your own hard drive and not just as a black box to use for deciphering. This is basically what went wrong when Jon Johansen and two unnamed hackers released DeCSS in October 1999. The Xing DVD player had its object code disassembled in order to obtain a player key.

5.3 Key hierarchy

As with any key hierarchy, it should be possible to go down, but not up. We have shown that it is possible to get a player key from a disc key in about 2^{16} iterations, which take about 20 ms on a modern computer. Giving that it is of paramount importance to protect the player keys we find this rather disturbing.

5.4 Security through obscurity

There has never been released an official description of the cryptosystem behind CSS. Its creators must have based part of its security the fact that those algorithms were kept unknown to the public, and thus ignoring Kerckhoffs' principle[8, 26] that we must assume that an opponent knows everything about our cryptosystem, except the key. This is underlined by the large number lawsuits that were filed by the MPAA in the period after this information was released in 1999.

5.5 Weak cryptosystem

As showed in the attacks the cryptosystem the actual time it takes to break to cryptosystem is nowhere near what the 40 bit key would suggest. The 2^{25} iterations could be done in a couple of seconds on a modern computer. As mentioned earlier, CSS, uses a product cipher of only 2, which is simply not enough to obtain security. In comparison, the Feistel cipher[13] used for the 56 bit DES algorithm, it has 16 rounds of

- permutation
- substitution
- linear mixing using XOR

The idea is that, even though each round is not enough to secure the system, adding more rounds will make the scheme more secure. In CSS there is only two rounds and as we have shown, there are attacks that use this fact[2]. Adding more rounds would as Shannon described it, add a large amount of confusion and diffusion[7].

[2] As an example, the table for the mKey[1] would not have been so easily build, if there had been more rounds, making mKey[1] into a function of all the mangling keys.

Chapter 6

Advanced Access Content System

The future brings larger storage medias for better quality and while making new standards for medias along comes new standards for encryption. The need for a stronger encryption is obvious. It is clear that the new major standard have learned a lesson from the faults in CSS by looking at what went wrong.

That is why the new Blu-ray[11] and HD-DVD[14] standards are encrypted under one or more title keys using Advanced Encryption Standard (AES) [12] in the Advanced Access Content System(AACS) [10]. The title keys are derived from several elements like the media key, volume ID of the disc and a hash of the title usage rules.

In trying to prevent the same attacks made on CSS to apply to AACS the manufacturers have made some new approaches.

One is that AACS provisions each individual DVD player with a unique set of decryption keys. This allows licensors to revoke individual players, or more specifically, the decryption keys associated with the player. If a given players keys are compromised by an attacker, the AACS licensing authority can simply revoke those keys in future content, making the keys and the player useless for decrypting new movies.

And also Blu-ray discs have a digital watermark technology that all players must check is correct. This is called the ROM-Mark and all Blu-ray device manufactures must have a license to insert the ROM-Mark into a media during replication. The digital rights management believe this will prevent copying Blu-ray discs as easily as with DVD medias.

But the question is: Have they prevented people from copying the discs as it was the real purpose of encrypting the media. It seems not to be the case.

Already attacks are made on Blu-ray, although the standard is yet to be acknowledged properly. One attack is to just take screenshots of the window playing the movie and then add sound later. This is of course a pretty straight forward way that you will always be able to unless the information is first decrypted in the monitor. This is actually the purposes of HDCP[1], but while the information is encrypted between the disc and the computer and between the computer and the monitor it has to be in plaintext in the computer to enable it to decode the MPEG-2 video stream, which has been shown to be a weak link. The AACS cryptosystem has not been broken, however this clearly not enough to secure the content.

Another simple attack is to have a PlayStation 3 running a Linux version and then simply use the Linux command dd(Disc Dump) in to dump the content of the disc to the hard drive[2].

Now the only problem is to copy it to a blu-ray disc with the right digital watermark in order to redistribute the disc, but you are although able to play the content from your hard drive.

It remains doubt full that one will break the encryption itself as it is based on AES. However it was not the cryptosystem CSS that was compromised at first, but only a player key - which led to a breake of CSS.

[1]High-Bandwidth Digital Content Protection[15]

[2]http://www.ps3news.com/forums/site-news/breaking-news-worlds-first-ps3-blu-ray-movie-dumped-40441.html

Chapter 7

Resources

7.1 Tstdvd

Tstdvd is an open source Linux tool to authenticate a computer with the DVD drive and then reading the hidden sector containing the disc keys. Tstdvd have also a function to descramble the DVD content and get more information about the DVD.

This tool was used to fake the authentication process and actually obtain the hidden sector so we could break the code, get the disc key and thereby the player keys.

7.2 Libdvdcss

libdvdcss is a open source library to access DVDs developed by the team behind the multiplatform media player VLC. Libdvdcss uses a set of predefined player keys to access the DVD, but if that fails Libdvdcss initiate a 2^{16} attack on the title key.

For more information:
http://developers.videolan.org/libdvdcss/

Used to compare found keys with others results and inspiration and understanding of the decryption.

Chapter 8

Conclusion

It should be clear by now that basing your security on the fact that nobody knows how your encryption scheme works[1] is not the best way to provide security.

CSS has so many issues and stands as one fine example of how not to define a cryptosystem. The makers of AACS have clearly learned a lesson from the flaws in CSS and have made a new standard based on cryptographic schemes that we today and for the near future rely on to be secure.

The attack on the hashed disc key described by Frank Stevenson was already fast, but had a pretty huge[2] space complexity, and using about six seconds on a 40 bit key is pretty fast. We have shown not how to implement an attack requiring much less space, and also gave an optimized with regards to the speed, so the code actually runs a factor three faster than the version by Frank Stevenson. As we have not focused on optimizing the code, it surely could be optimized. The optimization is of course not of practical importance, since it does not really matter if you permanently break the CSS encryption in two or six seconds or use 64 or 1 MB of RAM. It was only made of cryptographic interest to see if it was possible to improve the algorithm.

From a more general point of view encrypting public data such as a movie or book, it will always be an extremely difficult task. Thinking that you can accomplish this without making the algorithm available for extensive scrutiny by researches around the world can not be recommended.

All the data must be available to the user, because nobody wants to buy a DVD that is impossible to watch. So somewhere between the disc and your eyes, there must be something that turns the encrypted data into plaintext. Postponing it as much as possible would for the most part increase security.

[1]and suing anyone who finds out, which we hope does not include us
[2]especially for a computer in 1999

Bibliography

[1] Stéphane Borel and Håkan Hjort. Functions for dvd authentication and descrambling. http://www.daimi.au.dk/~rauff/crypto/css.c, 2005.

[2] Ivan Damgård. Definitions and results for crypto systems, 2004.

[3] Ivan Damgård. Key management, 2004.

[4] Frank A. Stevenson (frank@funcom.com). Cryptanalysis of contents scrambling system. http://www.cs.cmu.edu/~dst/DeCSS/FrankStevenson/analysis.html, 1999.

[5] Frank A. Stevenson (frank@funcom.com). Frank stevenson's css cracks. http://www.cs.cmu.edu/~dst/DeCSS/FrankStevenson/index.html, 1999.

[6] Gregory Kesden. Lecture 33 notes. http://www.cs.cmu.edu/~dst/DeCSS/Kesden/index.html, 2000.

[7] Claude Shannon. Communication theory of secrecy systems. http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf, 1949.

[8] Douglas Stinson. *Cryptography: Theory and Practice,Third Edition*. CRC/C&H, 2006.

[9] tstdvd /dev/hdc. The hidden disc key sector from pirates of the caribbean: The curse of the black pearl. http://www.daimi.au.dk/~rauff/crypto/disc-key_pir, 2006.

[10] wikipedia.org. Advanced access content system. http://en.wikipedia.org/wiki/Advanced_Access_Content_System, 2006.

[11] wikipedia.org. Advanced encryption standard. http://en.wikipedia.org/wiki/Advanced_Encryption_Standard, 2006.

[12] wikipedia.org. Blu-ray disc. http://en.wikipedia.org/wiki/Blue-ray, 2006.

[13] wikipedia.org. Feistel ciphers. http://en.wikipedia.org/wiki/Feistel_scheme, 2006.

[14] wikipedia.org. Hd dvd. http://en.wikipedia.org/wiki/HD_DVD, 2006.

[15] wikipedia.org. High-bandwidth digital content protection. http://en.wikipedia.org/wiki/High-Bandwidth_Digital_Content_Protection, 2006.

Chapter 9

Appendix

This appendix contains all the source code for the attacks we have implemented. It is implemented in the C language, and is compiled with gcc in a Linux environment.
In total there are three programs available:

- The ciphertext only attack on the hashed disc key. The main function is located in `dehash_main.c`. The actual algorithm is implemented in `dehash.c`.
- The known plaintext attack on the disc key and encrypted disc key. The main function is located in `playerkeyattack_main.c`. The actual algorithm is implemented in `playerkeyattack.c`.
- The combined attack, which takes a DVD hidden sector as input, and outputs all possible player keys. The main function is located in `fullattack.c`.

Information about how to compile a program is descripted in the top of the file containing the main function.

The source is available as a tarball at
http://www.daimi.au.dk/~rauff/crypto/source.tar
This tarball also includes three executable compile commands: `cdehash`, `cpk` and `cfullattack`. When the programs are compiled, they can be executed by

- `./dehash`
- `./playerkeyattack`
- `./fullattack`

A hidden sector to fullattack is available at
http://www.daimi.au.dk/~rauff/crypto/disc-key_pir

9.1 playerkeyattack

9.1.1 playerkeyattack_main.c

```
/**
  Program for running the playerkey attack only.
  Finds possible player keys from a disc key and a encrypted disc key.

  Compile command:
  gcc -Wall -o playerkeyattack playerkeyattack_main.c playerkeyattack.c lfsr.c util.c tables.c
**/


#include <stdio.h>
#include "playerkeyattack.h"
#include "util.h"


/**
  The main program. Takes a disc key and a encrypted disc key as parameters.
**/
int main(int nArgs, char *ppcArgs[] ) {
  const unsigned int MAX = 10;  // maximum player keys
  unsigned char dKey[5];        // disc key
  unsigned char edKey[5];       // encrypted disc key
  unsigned char pKeys[5*MAX];   // output player keys
  int nKeys;  // Number of player keys found
  int i;

  if( nArgs!=11 )
    printExit("Usage:  playerkeyattack DD DD DD DD DD  EE EE EE EE EE   (discKey encryptedDiscKey)");

  for( i=0; i<5; i++ ) {
    dKey[i] = getArg(ppcArgs[i+1]);
    edKey[i] = getArg(ppcArgs[i+6]);
  }

  print40bits("Disc key: ",dKey);
  print40bits("Enrypted disc key: ",edKey);

  nKeys = playerkeyattack( edKey, dKey, pKeys, MAX );   // Running the attack

  // Prints the output:
  printf("%d possible player keys:\n",nKeys);
  for(i=0;i<nKeys;i++)
    print40bits("",pKeys+i*5);

  return 0;
}
```

9.1.2 playerkeyattack.h

```
int playerkeyattack( const unsigned char* edKey, const unsigned char* dKey, unsigned char* pKeys, const unsigned int
    maxkeys );
```

9.1.3 playerkeyattack.c

```
/**
  This file contains the algorithm for the playerkey attack.
  For running this attack only, see playerkeyattack_main.c for a main() function.
**/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdint.h>

#include "tables.h"
#include "util.h"
#include "lfsr.h"



int getManglingKeys( const unsigned char* stage2, const unsigned char* edKey, unsigned char mKey4, unsigned char* mKey
    );
int checkPossibleKey( const unsigned char *mKey, const unsigned char *pKey );
void reconstructLFSR25key( const unsigned char *out17, const unsigned char *mKey, unsigned char *lfsr25key);
void getPlayerKeys( const unsigned char *mKey );

```

```c

// tables for out25{0,1,4} --> LFSR-25 key
static unsigned int lfsr25t0[256];
static unsigned int lfsr25t1[256];
static unsigned int lfsr25t4[256];

// Output bookkeeping
static unsigned char* outputs;
static unsigned int output_num, output_size;

// Boolean value indicating if the small tables are build.
static char bTablesBuild = 0;  // false





/**
  The playerkey attack algorithm.
  edKey:     Encrypted disc key.
  dKey:      Disc key.
  pKeys:     Output player keys.
  maxkeys:   Maximum player keys.
  Returns the number of player keys found.
 **/
int playerkeyattack( const unsigned char* edKey, const unsigned char* dKey, unsigned char* pKeys, const unsigned int maxkeys ) {
  unsigned char mKey[5]; // Possible mangling key
  unsigned int mKey4;

  // Set up output environment
  outputs = pKeys;
  output_size = maxkeys;
  output_num = 0;

  // Only build tables if they not already are build
  if( !bTablesBuild ) {
    buildLFSR25Tables( lfsr25t0, lfsr25t1, lfsr25t4 );
    bTablesBuild = 1;  // true
  }

  for(mKey4=0;mKey4<256;mKey4++) { // Guess mKey[4]
    if( getManglingKeys(dKey,edKey,mKey4,mKey) ) { // Find possible mangling keys
      //print40bits("Possible mangling key:", mKey);
      getPlayerKeys(mKey);   // Find possible player keys for a single mangling key.
    }
  }
  return output_num;
}



/**
  Constructs the full mangling key from the dics key (stage2), encrypted dics key, and the 5th mangling key byte.
  stage2:  Pointer to 5 bytes of unencrypted disc key (plain text).
  edKey:   Pointer to 5 bytes of encrypted disc key (cipher text).
  mKey4:   Value of mKey[4] (the 5th mangling key byte).
  mKey:    Pointer to 5 bytes of output mangling key.
  Returns boolean (1/0), true if mKey4 produces a possible mangling key, false otherwise.
 **/
int getManglingKeys( const unsigned char* stage2, const unsigned char* edKey, const unsigned char mKey4, unsigned char *mKey) {
  unsigned char stage1[5];

  mKey[4] = mKey4;
  stage1[4] = T[edKey[4]] ^ edKey[3] ^ mKey[4];   // Now we have the entire last column
  stage1[3] = T[stage1[4]] ^ stage2[4] ^ mKey[4]; // And we can use this to find stage1[3]
  mKey[3] = stage1[3] ^ edKey[2] ^ T[edKey[3]];   // Now we have the two last columns

  stage1[2] = T[stage1[3]] ^ stage2[3] ^ mKey[3]; // We can find stage1[2]
  mKey[2] = stage1[2] ^ edKey[1] ^ T[edKey[2]];   // Now we have the three last columns

  stage1[1] = T[stage1[2]] ^ stage2[2] ^ mKey[2]; // We can find stage1[1]
  mKey[1] = stage1[1] ^ edKey[0] ^ T[edKey[1]];   // Now we have the four last columns

  stage1[0] = T[stage1[1]] ^ stage2[1] ^ mKey[1]; // We can find stage1[0]
  mKey[0] = stage1[0] ^ stage1[4] ^ T[edKey[0]];  // Now we have it all

  if( stage2[0] == (mKey[0]^T[stage1[0]]) ) {   // Check if the mangling keys give the correct result for the first byte of the disc key.
    return 1; // true
  }

  return 0; // false
}





/**
  Finds possible player keys from a possible mangling key.
  mKey:  Pointer to 5 byte mangling key.
**/
void getPlayerKeys( const unsigned char *mKey ) {
  unsigned char key[5]; // tmp key
```

```c
  unsigned char lfsr17out[5];
  int i;
  for (i = 0; i < 256 * 256; i++) { // Guess start key of LFSR17
    key[0] = (i >> 8) & 0xFF;
    key[1] = i & 0xFF;
    lfsr17_produce5bytes(i,lfsr17out); // Produce 5 bytes
    reconstructLFSR25key(lfsr17out, mKey, &key[2]); //Reconstruct start key of LFSR25
    if (checkPossibleKey(mKey, key))
    {
      //print40bits("  Possible player key:", key);
      if( output_num<output_size ) {
        memcpy(outputs+output_num*5,key,5);
      } else {
        printf("Too␣many␣player␣keys!\n");
      }
      output_num++;
    }
  }
}


/**
  Takes a mangling key and a player key, and tests if the combination is possible.
  mKey:  Pointer to 5 byte mangling key.
  pKey:  Pointer to 5 byte player key.
  Returns boolean (1/0), true if possible, false otherwise.
**/
int checkPossibleKey( const unsigned char *mKey, const unsigned char *pKey ) {
  unsigned char lfsr17out[5];  // output from LFSR-17
  unsigned char lfsr25out[5];  // output from LFSR-25
  unsigned char totalout[5];   // 8 LSBs of the sum of the two preceding outputs
  int i;
  int cc; // carry (1 or 0)

  lfsr17_produce5bytes( (pKey[0]<<8) | (pKey[1]), lfsr17out );
  lfsr25_produce5bytes( (pKey[2]<<16) | (pKey[3]<<8) | (pKey[4]) ,lfsr25out );

  cc=0;
  for (i = 0; i < 5; i++) {
    totalout[i] = (lfsr17out[i]+lfsr25out[i]+cc) & 0xFF;
    cc = ((lfsr17out[i]+lfsr25out[i]+cc) & 0x100) >> 8;
    if( mKey[i]!=totalout[i] )
      return 0; // false
  }
  return 1; // true
}




/**
  Constructs the LFSR-25 key from the output of LFSR-17 and a mangling key.
  out17:      Pointer to 5 output bytes from LFSR-17.
  mKey:       Pointer to 5 bytes mangling key.
  lfsr25key:  Pointer to 3 LFSR-25 key bytes (output).
**/
void reconstructLFSR25key( const unsigned char *out17, const unsigned char *mKey, unsigned char *lfsr25key ) {
  //Reconstruct LFSR25 output(byte 1,2 and 5) from total output bytes of LFSRs)
  unsigned int test;
  unsigned int k;
  unsigned char out25[5];

  test = (0x100+mKey[0])-out17[0];
  out25[0] = test&0xFF;  // 8 LSBs
  if(test&0x100) test = 0x100+mKey[1]-out17[1];    // no carry
  else           test = 0x100-1+mKey[1]-out17[1];  // carry
  out25[1] = test&0xFF; // 8 LSBs

  test = (0x100+mKey[3])-out17[3];

  if(test&0x100) test = 0x100+mKey[4]-out17[4];    // no carry
  else           test = 0x100-1+mKey[4]-out17[4];  // carry
  out25[4] = test&0xFF;

  k = lfsr25t0[out25[0]] ^ lfsr25t1[out25[1]] ^ lfsr25t4[out25[4]]; // Use output to get start key of LFSR25
  lfsr25key[0] = (k>>16)&0xFF;
  lfsr25key[1] = (k>>8)&0xFF;
  lfsr25key[2] = k&0xFF;
}
```

9.2 dehash

9.2.1 dehash_main.c

```c
/**
  Program for running the dehash attack only.
```

```
  Finds possible disc keys from the hash.

  Compile command:
  gcc -Wall -o dehash dehash_main.c dehash.c lfsr.c util.c tables.c
**/


#include <stdio.h>
#include "util.h"
#include "dehash.h"


/**
  The main program. Takes the hash (disc key encrypted with it self) as parameter.
**/
int main( int nArgs, char *ppcArgs[] ) {
  const unsigned int MAXKEYS = 10;  // Maximum number of disc keys
  unsigned char h[5];               // The hash value
  unsigned char keys[5*MAXKEYS];    // Output disc keys
  int nKeys; // Number of disc keys found
  int i;

  if( nArgs != 6 )
    printExit("\nUsage:   dehash  xx xx xx xx xx   (hash)\n");

  for( i=0; i<5; i++ )
    h[i] = getArg( ppcArgs[i+1] );

  nKeys = dehash(h,keys,MAXKEYS);  // Running the attack

  // Prints the output
  printf("%d possible disc keys:\n",nKeys);
  for(i=0;i<nKeys;i++)
    print40bits("",keys+5*i);

  return 0;
}
```

9.2.2 dehash.h

```
int dehash( const unsigned char* hash, unsigned char* keys, const int maxkeys );
```

9.2.3 dehash.c

```
/**
  This file contains the algorithm for the dehashing attack.
  For running this attack only, see dehash_main.c for a main() function
**/


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <stdint.h>

#include "tables.h"
#include "util.h"
#include "lfsr.h"


#define LFSR25_TABLE_FILE "lfsr25table"
#define MAXCOLLISIONS 8
#define K1T(s10,s21,i) (k1t[ (s10*256+s21)*(MAXCOLLISIONS+1) + i ])
#define LFSR25T(out0,out1,out4) ( lfsr25t[ ((out0)<<16) | ((out1)<<8) | (out4) ] )
// out0,out1,out4 must be bytes! eg. unsigned char


void buildrest();
void decryptHash( unsigned char* output, const unsigned char* dKey );
void buildK1table();
void buildLFSR25table();
void buildLFSR25Tables();



// 5 byte stages/keys
static unsigned char hash[5];     // The input hash
static unsigned char stage1[5];   // The middle stage of the mangling cipher
static unsigned char stage2[5];   // The bottom stage of the mangling cipher
static unsigned char mKey[5];     // The mangling key. Eg. final outputs from the LFSR's and inputs to the mangling cipher.
static unsigned char out17[5];    // Output bytes from LFSR-17
static unsigned char out25[5];    // Output bytes from LFSR-25


// tables:
```

```
static unsigned int* lfsr25t;        // Huge table for out25{0,1,4} --> key25 mapping
static unsigned char* k1t;           // Possible (stage1[0],stage2[1]) --> mKey[1] mappings
// Small tables for out25{0,1,4} --> key25 mapping
static unsigned int lfsr25t0[256];
static unsigned int lfsr25t1[256];
static unsigned int lfsr25t4[256];



// Output bookkeeping
static unsigned char* outputs;
static unsigned int output_num, output_size;



/**
  h: Pointer to input hash (5 bytes)
  keys: Pointer to output disc keys (5*maxkeys bytes assumed)
  maxkeys: see keys.
  Returns the number of disc keys found.
**/
int dehash( const unsigned char* h, unsigned char* dKeys, const int maxkeys )
{
  unsigned int lfsr17;  // LFSR-17 key
  unsigned int s10;     // stage1[0]
  int i;

  // Copy the hash
  memcpy(hash,h,5);

  // Set up output environment
  outputs = dKeys;
  output_size = maxkeys;
  output_num = 0;


  print40bits("Hash:␣",hash);

  // Building tables
  buildK1table(); // k1t
  buildLFSR25Tables( lfsr25t0, lfsr25t1, lfsr25t4 ); // 3x 256byte tables
  //buildLFSR25table(); // 64MB table

  // Iteration through all possible LFSR-17 keys
  printf("Searching␣for␣keys:\n");
  for( lfsr17=0; lfsr17<0x10000; lfsr17++ ) {
    // Progress indication
    if((lfsr17&0x0FFF)==0x0FFF) {
      printf("\r%u%%.␣␣",((lfsr17+1)*100)/0x10000);
      fflush(stdout);
    }
    // Gets the output bytes from LFSR-17 with the key
    lfsr17_produce5bytes(lfsr17,out17);
    // The two first bytes of the demangling is equal to the initial key for lfsr17, because the the plaintext and the key is the same.
    stage2[0] = lfsr17 >> 8;
    stage2[1] = lfsr17 & 0xFF;
    // Iteration through all possible stage1[0] values
    for( s10=0; s10<0x100; s10++ ) {
      stage1[0] = s10;
      mKey[0]    = T[stage1[0]] ^ stage2[0];           // Calculates mKey[0]
      stage1[4] = stage1[0] ^ T[hash[0]] ^ mKey[0];   // Calculates stage1[4]
      mKey[4]    = stage1[4] ^ hash[3] ^ T[hash[4]];  // Calculates mKey[4]

      // Number of possible values for mKey[1], given stage1[0] and stage2[1]
      unsigned char nKeys = K1T(stage1[0],stage2[1],0);
      // Iteration through all the possible values for mKey[1]
      for ( i=1; i<=nKeys ; i++ ) {
        mKey[1] = K1T(stage1[0],stage2[1],i);      // Look up the next possibility for mKey[1]
        stage1[1] = hash[0]^T[hash[1]]^mKey[1];   // Calculates stage1[1]

        // Reconstruct the 1st, 2nd and 5th LFSR25 output byte from the output bytes of LFSR-17 and the summed output from LFSR-17 and LFSR-25
        unsigned int test;
        test = (0x100+mKey[0])-out17[0];
        out25[0] = test&0xFF;  // 8 LSBs
        if(test&0x100) test = 0x100+mKey[1]-out17[1];     // no carry from mKey[0]
        else           test = 0x100-1+mKey[1]-out17[1];  // carry from mKey[0]
        out25[1] = test&0xFF;  // 8 LSBs

        // It's unknown if there was a carry from mKey[3]

        // Tries out25[4] with no carry from mKey[3]
        out25[4] = (0x100+mKey[4])-out17[4];
        buildrest();

        // Tries out25[4] with a carry from mKey[3]
        out25[4] = (0x100-1+mKey[4])-out17[4];
        buildrest();
      }
    }
  }

  printf("\n");
  free(k1t);
```

```
  return output_num;
}




void buildrest() {
  unsigned int key25;          // The LFSR-25 key
  unsigned char output[5];     // Output from the decryption algorithm

  // Calculates the LFSR-25 key from 1st, 2nd and 5th output byte from LFSR-25
  key25 = lfsr25t0[out25[0]] ^ lfsr25t1[out25[1]] ^ lfsr25t4[out25[4]];  // 3x (256 * 4) byte tables
  // Looks up LFSR-25 key from 1st, 2nd and 5th output byte from LFSR-25 (our first implementation)
  //key25 = LFSR25T(out25[0],out25[1],out25[4]);  // 64MB table

  // Converts the 24bit key to 3 bytes
  stage2[2] = (key25>>16)&0xFF;
  stage2[3] = (key25>>8)&0xFF;
  stage2[4] = key25&0xFF;

  // Calculating the remaining values of the mangling cipher
  stage1[3] = stage2[4]^mKey[4]^T[stage1[4]];
  mKey[3] = stage1[3]^hash[2]^T[hash[3]];
  stage1[2] = stage2[3]^mKey[3]^T[stage1[3]];
  mKey[2] = stage1[2]^hash[1]^T[hash[2]];

  // If stage2[2] is correct according to its "generators", stage2 is a possible disc key
  if( (stage1[1]^T[stage1[2]]^mKey[2]) == stage2[2] ) {
    // Running the decryption algorithm with stage2 as the disc key
    decryptHash(output, stage2);
    // If the decrypted key equals stage2, this can be a disc key
    if( memcmp(output, stage2, 5)==0 ) {
      //print40bits("\rPossible key found: ", stage2);
      if( output_num<output_size ) {
        memcpy(outputs+(5*output_num), stage2, 5);
      } else {
        printf("Output␣size␣is␣too␣small!\n");
      }
      output_num++;
    }
  }
}




/**
  The mangling decryption algorithm. (Also uses the global hash as input)
  dKey: Pointer to 40bit disc key (5bytes)
  output: Pointer to the hash decrypted with this key.
**/
void decryptHash( unsigned char* output, const unsigned char* dKey ) {
  unsigned char
     lfsr17[5], // Output from LFSR-17
     lfsr25[5], // Output from LFSR-25
     mKey[5],   // Sum of the two above outputs (mangling key)
     s1[5]      // Stage1 - middle mangling stage
  ;
  unsigned char cc;  // carry
  unsigned int test; // carry test
  unsigned int i;


  // Get output from the LFSR's given the disc key
  lfsr17_produce5bytes( (dKey[0]<<8)|(dKey[1]), lfsr17 );
  lfsr25_produce5bytes( (dKey[2]<<16)|(dKey[3]<<8)|(dKey[4]), lfsr25 );

  // Calculate the mangling key (sum of the two LFSR outputs)
  cc=0;
  for(i=0;i<5;i++) {
    test = lfsr17[i]+lfsr25[i]+cc;
    if(test&0x100) cc=1;
    else cc=0;
    mKey[i] = test&0xFF;
  }

  // Calculates the middle stages
  for(i=1;i<5;i++)
    s1[i] = hash[i-1]^T[hash[i]]^mKey[i];
  s1[0] = s1[4]^T[hash[0]]^mKey[0];

  // Calculates the bottom stages (outout)
  output[0] = T[s1[0]]^mKey[0];
  for(i=1;i<5;i++)
    output[i] = s1[i-1]^T[s1[i]]^mKey[i];
}


/**
  Builds the huge 64MB lookup table (lfsr25t)
  The table can be seen as a fixed 3d with
  (out25[0],out25[1],out25[4]) = key25
**/
void buildLFSR25table() {
```

```
  unsigned int size;        // Memory size
  unsigned int key25;       // LFSR-25 key
  unsigned char output[5]; // Output bytes
  unsigned int i,j;
  FILE *fp;

  // Allocation of 2^24 times 32 bit = 64MB
  size =  0x10 * 0x100000 * sizeof(unsigned int); // 0x1000000 times 32 bit
  printf("Allocating %u bytes of memory: ",size);
  lfsr25t = (unsigned int*)malloc(size);
  if(lfsr25t==NULL)
    printExit("Error.");
  printf("OK.\n");

  // Tries to open table file
  fp = fopen(LFSR25_TABLE_FILE,"rb");
  if( fp==NULL ) {
    // File not fooound.
    printf("Building 64MB table for out{0,1,4} -> lfsr25 key: ");
    fflush(stdout);
    memset(lfsr25t,0,size);
    key25=0;
    // Iteration through (0x10*0x100000=0x1000000=2^24) possible keys for LFSR-25
    for(i=0;i<0x10;i++) {
      for(j=0;j<0x100000;j++,key25++) {
        // Runs the LFSR-25 on this key and get the 5 output bytes
        lfsr25_produce5bytes(key25,output);
        // Create the mapping from 1st, 2nd and 5th output byte to the key
        LFSR25T(output[0],output[1],output[4]) = key25;
      }
      printf(".");
      fflush(stdout);
    }
    printf(" Done.\n");
    printf("Writing 64MB table to disc:  ");
    fflush(stdout);
    // Write to table file
    fp = fopen(LFSR25_TABLE_FILE,"wb");
    if( fp==NULL ) {
      printf(" [[[Can't write to %s]]]\n",LFSR25_TABLE_FILE);
    } else {
      fwrite(lfsr25t,size,1,fp);
      fclose(fp);
      printf("OK.\n");
    }
  }
  else {
    // File found - loading in.
    printf("LFSR25-table file found, loading:  ");
    fflush(stdout);
    fread(lfsr25t,size,1,fp);
    fclose(fp);
    printf("Done.\n");
  }
}



/**
  Builds the table mapping from (stage1[0],stage2[1]) to possible values for mKey[1]
  The table can be seen as a fixed size 3d table, with
  (stage1[0],stage2[1],0) = Number of possible values for mKey[1]
  (stage1[0],stage2[1],1...MAXCOLLISIONS) = Space for the possible mKey[1]'s
  eg. only (stage1[0],stage2[1],0...(stage1[0],stage2[1],0)) is used.
**/
void buildK1table() {
  unsigned int size;  // Memory size
  unsigned int s10;   // stage1[0]
  unsigned int s21;   // stage2[1]
  unsigned int mKey1; // mKey[1]

  // We hope that no more than MAXCOLLISIONS stage1[0] and stage2[1] result in the same key
  // Look in paper for a reasonable estimate.
  size = 256*256*(MAXCOLLISIONS+1);
  printf("Allocating %u bytes of memory:  ",size);
  k1t = (unsigned char*)malloc(size);
  if( k1t==NULL )
    printExit("Error.");
  printf("OK.\n");

  // (stage1[0],stage2[1]) --> mKey[1] possible keys
  printf("Building k1-table:  ");
  memset(k1t,0,size);
  // Iteration through all possible values of mKey[1]
  for( mKey1=0; mKey1<0x100; mKey1++ ) {
    // Iteration through all possible values of stage1[0]
    for( s10=0; s10<0x100; s10++ ) {
      s21 = s10 ^ mKey1 ^ T[hash[0]^T[hash[1]]^mKey1]; // For each mKey1 and stage1[0] we find stage2[1]
      int nKeys = K1T(s10,s21,0);               // nKeys is the number of stored mKey[1]'s for stage1[0] and stage2[1]
      nKeys++;                                  // We have one more key
      if (nKeys > MAXCOLLISIONS)                // Test if we have overrun
        printExit("Too many collisions, aborting...");
      K1T(s10,s21,0) = nKeys;                   // Update the number of keys
      K1T(s10,s21,nKeys) = mKey1;               // Add mKey1 to the list of keys for the specific stage1[0] and stage2[1]
```

```
    }
  }
  printf("Done.\n");
}
```

9.3 fullattack

9.3.1 fullattack.c

```
/**
  This file contains the program that finds all the possible player keys on a single DVD disc,
  given the first (hidden) sector of the disc as input. (From a file)

  Compile command:
  gcc -o fullattack fullattack.c dehash.c playerkeyattack.c lfsr.c tables.c util.c
**/


#include <stdio.h>
#include <string.h>

#include "util.h"
#include "playerkeyattack.h"
#include "dehash.h"


#define MAX_DISC_KEYS 10
#define MAX_PLAYER_KEYS 10
#define MAX_SET_KEYS 200

#define N(data,index) (data+5*index)


static unsigned char playerKeySet[5*MAX_SET_KEYS];
static unsigned int playerKeySetSize;



/**
  Very slow implementation of adding a element to a set.
  It tests the new element (player key) for equality with any existing player keys in the set.
**/
void addToSet( unsigned char* e ) {
  int i;
  if(playerKeySetSize>=MAX_SET_KEYS) {
    printExit("Too many keys in set.\n");
  }
  for(i=0;i<playerKeySetSize;i++) {
    if( memcmp(e,N(playerKeySet,i),5)==0 )
      return;
  }
  memcpy(N(playerKeySet,playerKeySetSize),e,5);
  playerKeySetSize++;
}



/**
  The main program.
  Takes a hidden sector filename as parameter.
**/
int main( int argc, char *argv[] ) {
  FILE *fp;
  unsigned char
    buf[2048],                        // The hidden sector data
    discKeys[5*MAX_DISC_KEYS],        // Space to store disc keys from the dehash algorithm
    playerKeys[5*MAX_PLAYER_KEYS]     // Space to store player keys form the playerkey attack algorithm
  ;
  unsigned int
    nDiscKeys,    // Number of dics keys found
    nPlayerKeys,  // Number of player keys found (for one pair of disc key and encrypted disc key)
    i,j,k
  ;

  if(argc!=2)
    printExit("Usage:  fullattack sectorfile");

  // Read 2048 bytes from file
  fp = fopen(argv[1],"rb");
  if( fp==NULL )
    printExit("Cannot open file.");
  fread(buf,2048,1,fp);
  fclose(fp);

  // Find possible disc keys from the hash located at the begining of the sector
  nDiscKeys = dehash(buf+0,discKeys,MAX_DISC_KEYS);
```

```c
  if(nDiscKeys>MAX_DISC_KEYS)
    printExit("Too many disc keys.\n");

  // Iteration through all the disc keys
  for(i=0;i<nDiscKeys;i++) {
    playerKeySetSize=0;
    print40bits("Trying player keys for disc key  ",N(discKeys,i));
    // Iteration through all the encrypted disc keys
    for(j=1;j<409;j++) {
      // Progress indication
      printf("\r%03d  ",j);
      fflush(stdout);
      // Run the player key attack on the encrypted dics key and the disc key
      nPlayerKeys = playerkeyattack( N(buf,j), N(discKeys,i), playerKeys, MAX_PLAYER_KEYS );
      // Adds all the found player keys to our set
      for(k=0;k<nPlayerKeys;k++)
        addToSet(N(playerKeys,k));
    }
    // Print out all the distinct player keys
    print40bits("\nPlayer keys for ",N(discKeys,i));
    for(j=0;j<playerKeySetSize;j++)
      print40bits("",N(playerKeySet,j));
    printf("(Total %d)\n",playerKeySetSize);
  }

  return 0;
}
```

9.4 tables

9.4.1 tables.h

```c
#include<stdio.h>

/**
  Substitution table for the mangeling cipher.
**/
static const unsigned char T[256]=
{
  0x33,0x73,0x3b,0x26,0x63,0x23,0x6b,0x76,0x3e,0x7e,0x36,0x2b,0x6e,0x2e,0x66,0x7b,
  0xd3,0x93,0xdb,0x06,0x43,0x03,0x4b,0x96,0xde,0x9e,0xd6,0x0b,0x4e,0x0e,0x46,0x9b,
  0x57,0x17,0x5f,0x82,0xc7,0x87,0xcf,0x12,0x5a,0x1a,0x52,0x8f,0xca,0x8a,0xc2,0x1f,
  0xd9,0x99,0xd1,0x00,0x49,0x09,0x41,0x90,0xd8,0x98,0xd0,0x01,0x48,0x08,0x40,0x91,
  0x3d,0x7d,0x35,0x24,0x6d,0x2d,0x65,0x74,0x3c,0x7c,0x34,0x25,0x6c,0x2c,0x64,0x75,
  0xdd,0x9d,0xd5,0x04,0x4d,0x0d,0x45,0x94,0xdc,0x9c,0xd4,0x05,0x4c,0x0c,0x44,0x95,
  0x59,0x19,0x51,0x80,0xc9,0x89,0xc1,0x10,0x58,0x18,0x50,0x81,0xc8,0x88,0xc0,0x11,
  0xd7,0x97,0xdf,0x02,0x47,0x07,0x4f,0x92,0xda,0x9a,0xd2,0x0f,0x4a,0x0a,0x42,0x9f,
  0x53,0x13,0x5b,0x86,0xc3,0x83,0xcb,0x16,0x5e,0x1e,0x56,0x8b,0xce,0x8e,0xc6,0x1b,
  0xb3,0xf3,0xbb,0xa6,0xe3,0xa3,0xeb,0xf6,0xbe,0xfe,0xb6,0xab,0xee,0xae,0xe6,0xfb,
  0x37,0x77,0x3f,0x22,0x67,0x27,0x6f,0x72,0x3a,0x7a,0x32,0x2f,0x6a,0x2a,0x62,0x7f,
  0xb9,0xf9,0xb1,0xa0,0xe9,0xa9,0xe1,0xf0,0xb8,0xf8,0xb0,0xa1,0xe8,0xa8,0xe0,0xf1,
  0x5d,0x1d,0x55,0x84,0xcd,0x8d,0xc5,0x14,0x5c,0x1c,0x54,0x85,0xcc,0x8c,0xc4,0x15,
  0xbd,0xfd,0xb5,0xa4,0xed,0xad,0xe5,0xf4,0xbc,0xfc,0xb4,0xa5,0xec,0xac,0xe4,0xf5,
  0x39,0x79,0x31,0x20,0x69,0x29,0x61,0x70,0x38,0x78,0x30,0x21,0x68,0x28,0x60,0x71,
  0xb7,0xf7,0xbf,0xa2,0xe7,0xa7,0xef,0xf2,0xba,0xfa,0xb2,0xaf,0xea,0xaa,0xe2,0xff
};


/**
  Table for reversing the order of bits in a byte.
**/
static const unsigned char reverse[256]=
{
  0x00,0x80,0x40,0xc0,0x20,0xa0,0x60,0xe0,0x10,0x90,0x50,0xd0,0x30,0xb0,0x70,0xf0,
  0x08,0x88,0x48,0xc8,0x28,0xa8,0x68,0xe8,0x18,0x98,0x58,0xd8,0x38,0xb8,0x78,0xf8,
  0x04,0x84,0x44,0xc4,0x24,0xa4,0x64,0xe4,0x14,0x94,0x54,0xd4,0x34,0xb4,0x74,0xf4,
  0x0c,0x8c,0x4c,0xcc,0x2c,0xac,0x6c,0xec,0x1c,0x9c,0x5c,0xdc,0x3c,0xbc,0x7c,0xfc,
  0x02,0x82,0x42,0xc2,0x22,0xa2,0x62,0xe2,0x12,0x92,0x52,0xd2,0x32,0xb2,0x72,0xf2,
  0x0a,0x8a,0x4a,0xca,0x2a,0xaa,0x6a,0xea,0x1a,0x9a,0x5a,0xda,0x3a,0xba,0x7a,0xfa,
  0x06,0x86,0x46,0xc6,0x26,0xa6,0x66,0xe6,0x16,0x96,0x56,0xd6,0x36,0xb6,0x76,0xf6,
  0x0e,0x8e,0x4e,0xce,0x2e,0xae,0x6e,0xee,0x1e,0x9e,0x5e,0xde,0x3e,0xbe,0x7e,0xfe,
  0x01,0x81,0x41,0xc1,0x21,0xa1,0x61,0xe1,0x11,0x91,0x51,0xd1,0x31,0xb1,0x71,0xf1,
  0x09,0x89,0x49,0xc9,0x29,0xa9,0x69,0xe9,0x19,0x99,0x59,0xd9,0x39,0xb9,0x79,0xf9,
  0x05,0x85,0x45,0xc5,0x25,0xa5,0x65,0xe5,0x15,0x95,0x55,0xd5,0x35,0xb5,0x75,0xf5,
  0x0d,0x8d,0x4d,0xcd,0x2d,0xad,0x6d,0xed,0x1d,0x9d,0x5d,0xdd,0x3d,0xbd,0x7d,0xfd,
  0x03,0x83,0x43,0xc3,0x23,0xa3,0x63,0xe3,0x13,0x93,0x53,0xd3,0x33,0xb3,0x73,0xf3,
  0x0b,0x8b,0x4b,0xcb,0x2b,0xab,0x6b,0xeb,0x1b,0x9b,0x5b,0xdb,0x3b,0xbb,0x7b,0xfb,
  0x07,0x87,0x47,0xc7,0x27,0xa7,0x67,0xe7,0x17,0x97,0x57,0xd7,0x37,0xb7,0x77,0xf7,
  0x0f,0x8f,0x4f,0xcf,0x2f,0xaf,0x6f,0xef,0x1f,0x9f,0x5f,0xdf,0x3f,0xbf,0x7f,0xff
};


void buildLFSR25Tables( unsigned int* lfsr25t0, unsigned int* lfsr25t1, unsigned int* lfsr25t4 );
```

9.4.2 tables.c

```c
#include <stdio.h>


/**
  Builds the 3 tables for looking up the start state of LFSR-25 from the 1st, 2nd and 5th output byte of LFSR-25.
  See the report for further information.
**/
void buildLFSR25Tables( unsigned int* lfsr25t0, unsigned int* lfsr25t1, unsigned int* lfsr25t4 ) {
  unsigned int i,j;
  unsigned int b[25];  // bit value on position i (position 1-25)

  // 1st output byte:
  for (i = 0; i < 256; i++) {
    for(j=0;j<8;j++) {
      b[j+1] = (i>>j)&1;
    }
    lfsr25t0[i]  = 1 ^ b[1] ^ b[7];
    lfsr25t0[i] |= (1 ^ b[7]) << 1;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[3] ^ b[6] ^ b[8]) << 2;
    lfsr25t0[i] |= (b[1] ^ b[3] ^ b[5] ^ b[7]) << 3;
    lfsr25t0[i] |= (b[1] ^ b[3] ^ b[4] ^ b[7] ^ b[8]) << 4;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[3] ^ b[6] ^ b[7]) << 5;
    lfsr25t0[i] |= (b[1] ^ b[3] ^ b[5] ^ b[6]) << 6;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[4] ^ b[5]) << 7;
    lfsr25t0[i] |= (b[3] ^ b[7] ^ b[8]) << 8;
    lfsr25t0[i] |= (1 ^ b[3] ^ b[8]) << 9;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[3] ^ b[7] ^ b[8]) << 10;
    lfsr25t0[i] |= (1 ^ b[2] ^ b[8]) << 11;
    lfsr25t0[i] |= (1 ^ b[2] ^ b[3] ^ b[7]) << 12;
    lfsr25t0[i] |= (b[2] ^ b[7]) << 13;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[3] ^ b[7]) << 14;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[7]) << 15;
    lfsr25t0[i] |= (1 ^ b[1]) << 16;
    lfsr25t0[i] |= (b[2] ^ b[3]) << 17;
    lfsr25t0[i] |= (b[1] ^ b[2]) << 18;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[3] ^ b[8]) << 19;
    lfsr25t0[i] |= (b[1] ^ b[3] ^ b[7] ^ b[8]) << 20;
    lfsr25t0[i] |= (b[1] ^ b[3]) << 21;
    lfsr25t0[i] |= (1 ^ b[1] ^ b[2] ^ b[8]) << 22;
    lfsr25t0[i] |= (b[1] ^ b[2] ^ b[3] ^ b[7]) << 23;
  }
  // 2nd output byte:
  for (i = 0; i < 256; i++) {
    for(j=0;j<8;j++) {
      b[j+9] = (i>>j)&1;
    }
    lfsr25t1[i]  =  b[9] ^ b[11] ^ b[12];
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[12] ^ b[14]) << 1;
    lfsr25t1[i] |= (b[9] ^ b[13] ^ b[14] ^ b[15] ^ b[16]) << 2;
    lfsr25t1[i] |= (b[9] ^ b[11] ^ b[12] ^ b[14] ^ b[16]) << 3;
    lfsr25t1[i] |= (b[12] ^ b[13] ^ b[16]) << 4;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[11] ^ b[12] ^ b[13] ^ b[14] ^ b[15]) << 5;
    lfsr25t1[i] |= (b[10] ^ b[12] ^ b[14] ^ b[15] ^ b[16]) << 6;
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[15]) << 7;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[12] ^ b[13] ^ b[14] ^ b[16]) << 8;
    lfsr25t1[i] |= (b[14] ^ b[16]) << 9;
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[13] ^ b[14] ^ b[16]) << 10;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[13] ^ b[14] ^ b[15]) << 11;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[11] ^ b[12] ^ b[13] ^ b[15] ^ b[16]) << 12;
    lfsr25t1[i] |= (b[11] ^ b[13] ^ b[14] ^ b[15]) << 13;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[11] ^ b[15] ^ b[16]) << 14;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[12] ^ b[15]) << 15;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[13] ^ b[14]) << 16;
    lfsr25t1[i] |= (b[11] ^ b[12] ^ b[15] ^ b[16]) << 17;
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[14] ^ b[15]) << 18;
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[14] ^ b[15] ^ b[16]) << 19;
    lfsr25t1[i] |= (b[10] ^ b[11] ^ b[14] ^ b[16]) << 20;
    lfsr25t1[i] |= (b[9] ^ b[11] ^ b[12] ^ b[13] ^ b[16]) << 21;
    lfsr25t1[i] |= (b[9] ^ b[11] ^ b[12] ^ b[13] ^ b[14] ^ b[15]) << 22;
    lfsr25t1[i] |= (b[9] ^ b[10] ^ b[12] ^ b[14] ^ b[15] ^ b[16]) << 23;
  }
  // 5th output byte:
  for (i = 0; i < 256; i++) {
    for(j=0;j<8;j++) {
      b[j+17] = (i>>j)&1;
    }
    lfsr25t4[i]  =  b[17] ^ b[19] ^ b[20] ^ b[22] ^ b[24];
    lfsr25t4[i] |= (b[22] ^ b[24]) << 1;
    lfsr25t4[i] |= (b[18] ^ b[19] ^ b[23]) << 2;
    lfsr25t4[i] |= (b[17] ^ b[21] ^ b[22]) << 3;
    lfsr25t4[i] |= (b[20] ^ b[21] ^ b[22] ^ b[23] ^ b[24]) << 4;
    lfsr25t4[i] |= (b[20] ^ b[21] ^ b[22] ^ b[23]) << 5;
    lfsr25t4[i] |= (b[18] ^ b[19] ^ b[20] ^ b[22]) << 6;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[21]) << 7;
    lfsr25t4[i] |= (b[17] ^ b[19] ^ b[21] ^ b[22] ^ b[23] ^ b[24]) << 8;
    lfsr25t4[i] |= (b[18] ^ b[19] ^ b[20] ^ b[21] ^ b[24]) << 9;
    lfsr25t4[i] |= (b[17] ^ b[20] ^ b[21] ^ b[23]) << 10;
    lfsr25t4[i] |= (b[18] ^ b[20] ^ b[21] ^ b[23] ^ b[24]) << 11;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[20] ^ b[21] ^ b[22] ^ b[23]) << 12;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[19] ^ b[20] ^ b[23] ^ b[24]) << 13;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[24]) << 14;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[19] ^ b[21] ^ b[22] ^ b[24]) << 15;
```

```
    lfsr25t4[i] |= (b[19]) << 16;
    lfsr25t4[i] |= (b[21]) << 17;
    lfsr25t4[i] |= (b[20]) << 18;
    lfsr25t4[i] |= (b[18] ^ b[19] ^ b[21]) << 19;
    lfsr25t4[i] |= (b[17] ^ b[20] ^ b[21]) << 20;
    lfsr25t4[i] |= (b[19] ^ b[20] ^ b[22] ^ b[23] ^ b[24]) << 21;
    lfsr25t4[i] |= (b[18] ^ b[21] ^ b[22] ^ b[23]) << 22;
    lfsr25t4[i] |= (b[17] ^ b[18] ^ b[20] ^ b[22]) << 23;
  }

}
```

9.5 lfsr

9.5.1 lfsr.h

```
void lfsr17_produce5bytes( const int key, unsigned char* output );
void lfsr25_produce5bytes( const int key, unsigned char* output );
```

9.5.2 lfsr.c

```
#include "tables.h"


/**
  key: the 16bit key
  output: pointer to 5 output bytes
**/
void lfsr17_produce5bytes( const int key, unsigned char *output ) { // http://www.tinyted.net/eddie/css_basic.html
  unsigned int lfsr17; // state
  unsigned int bits;   // temp value
  int i;

  // The initial state (bit 9 high)
  lfsr17 = (reverse[ (key>>8)&0xFF ]<<9)
         | 0x100
         | reverse[key&0xFF];
  // Produce 5 bytes
  for (i=0;i<5;i++) {
    lfsr17 = (lfsr17<<9) | (lfsr17>>8); // rotate 8 positions to the right
    bits = lfsr17 & 0x03FC0;  // bits to be xor'ed
    lfsr17 ^= (bits<<3)^(bits<<6)^(bits<<9);
    lfsr17 &= 0x1FFFF;       // We only need the first 17 bits
    output[i] = lfsr17>>9;  // Output byte = the 8 MSBs in the LFSR
  }
}



/**
  key: the 24bit key
  output: pointer to 5 output bytes
**/
void lfsr25_produce5bytes( const int key, unsigned char* output ) {
  unsigned int lfsr25;   // state
  unsigned int highbyte; // The 8 MSBs
  unsigned int i;

  // The initial state
  highbyte = reverse[ (key>>16)&0xFF ];
  lfsr25 = ((highbyte&0xE0)<<17) | 0x200000 | ((highbyte&0x1F)<<16)
         | (reverse[ (key>>8)&0xFF ]<<8)
         | (reverse[ (key)&0xFF ])
  ;
  // Produce 5 bytes
  for(i=0;i<5;i++) {
    highbyte = (lfsr25 ^ (lfsr25>>3) ^ (lfsr25>>4) ^ (lfsr25>>12)) & 0xFF;  // The new 8 MSBs
    lfsr25 = (highbyte<<17) | (lfsr25>>8);  // The new state (shift 8 bits right and putting in the high byte
    output[i] = highbyte;  // Output byte = the 8 MSBs in the LFSR
  }
}
```

9.6 util

9.6.1 util.h

```c
int getArg( char* arg );
int hexdigitToInt( unsigned char d );
void print40bits( const char* text, const unsigned char* bytes );
void printExit( const char* msg );
```

9.6.2 util.c

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>
#include "util.h"


/**
  Converts one hexdigit to an integer (0-15).
**/
int hexdigitToInt( unsigned char d ) {
  if( d>='0' && d<='9' )
    return d-'0';
  if( d>='A' && d<='F' )
    return d-'A'+10;
  printExit("Wrong hexdigit.");
  return -1; // Dead code, but pleases the compiler
}


/**
  Converts a string of one or two hexdigits to an integer (0-255).
**/
int getArg( char* arg ) {
  unsigned char high,low;
  unsigned int len = strlen(arg);

  if( len!=1 && len!=2 )
    printExit("Use one or two digits per argument.");
  if(arg[1]==0) {
    high='0';
    low=toupper(arg[0]);
  } else {
    high=toupper(arg[0]);
    low=toupper(arg[1]);
  }
  return 0x10*hexdigitToInt(high)+hexdigitToInt(low);
}


/**
  Prints 40 bits (5 bytes) in two-digits-per-byte hexadecimal format followed by a line shift.
  text:   Preceding text.
  bytes:  Pointer to to 5 bytes.
**/
void print40bits( const char* text, const unsigned char* bytes ) {
  int i;
  printf(text);
  for(i=0;i<5;i++)
    printf(" %02X",bytes[i]);
  printf("\n");
}



/**
  Prints out a messing and exit the program.
**/
void printExit( const char* msg ) {
  printf(msg);
  printf("\n");
  exit(-1);
}
```