dCryptology

DVD Copy Protection

By

Kasper Kristensen, 20072316
Asger Eriksen, 20073117
Mads Paulsen, 20072890

Content Scrambling System

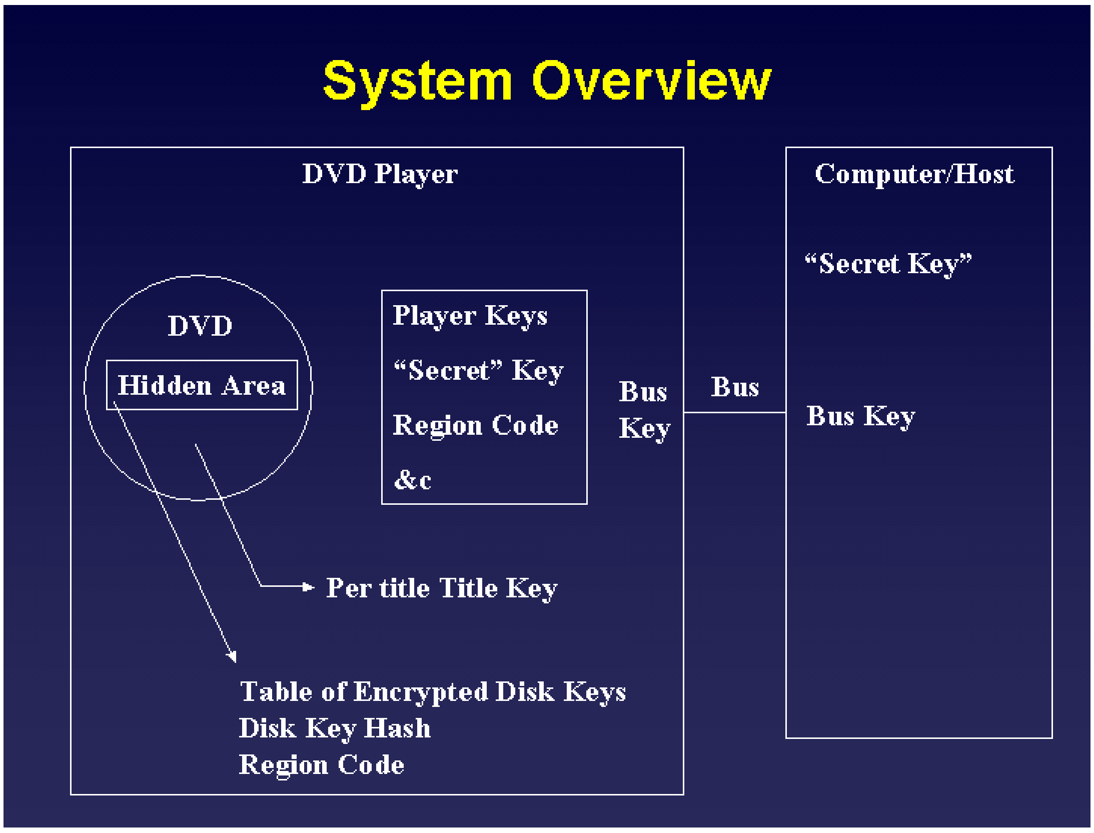
Content Scrambling System or CSS is being used to protect DVDs againts piracy and to add the region-based viewing restrictions, so that you only can watch DVDs bought in your own region.

To explain CSS, we will look at the DVD player as three components namely the DVD, the DVD player and the host.

The DVD contains 2 things, the video itself and "a hidden area", which is used to store some keys, that will only be given to an authenticated device.

The DVD player stores a couple of different things used to authenticate itself. It has a playerkey, which is used to decrypt the key on the DVD, a region code, that defines what area this player is from and a "secret" key used for authentication.

The host contains the secret, that they use for the private key authentication.



The keys

The playerkey, which is the key, that is given to the manufacturer of the DVD player by the *DVD Copy Control Association***.** This is stored within the player, and is used to decrypt the disk key. There's a total of 409 playerkeys.

The Disk-Key. This key is used to encrypt the title key, and is itself decrypted by using the playerkey. A table of the disk key encrypted under all the 409 playerkeys is stored on a hidden sector of the DVD. In that sector, the disk key encrypted under the disk key is also stored. The reason to do this, is so the player can verify, that it got the right disk key.

The title key, which is XORed with the sector key, and is then used to encrypt the data in a sector.

The sector key, which is a 128 byte plain-text header for each of the 2048 byte sectors. The bytes 80-84 in the header, is an additional key, that is used to encode the data in the sector.

The Authentication key, which is used as the “secret” key for the mutual authentication process.

The session key, that is negotiated when we authenticate, and is used to encrypt disk keys and more, while sending them over the bus, which is unprotected. We need this to prevent eavesdropping.

How it works

<u>The first step:</u> is, that they, host and player, have to authenticate to each other, to make sure, that they actually are who they say they are. This is done by a challenge-response system, and in the process, they come up with a session key.

<u>Second step:</u> Now the DVD player have to decode the information on the DVD. The DVD player now tries the different playerkeys, until it can decode the disk key, which is diskwide.

<u>Third step:</u> this is where the title and bus key are sent from the player to the host. Here we use the session key found in the first step, to prevent a man-in-the-middle attack.

<u>Fourth step:</u> simply sending a sector from the dvd to the host.

<u>Fifth step:</u> is the host decoding the title key by using the disk key.

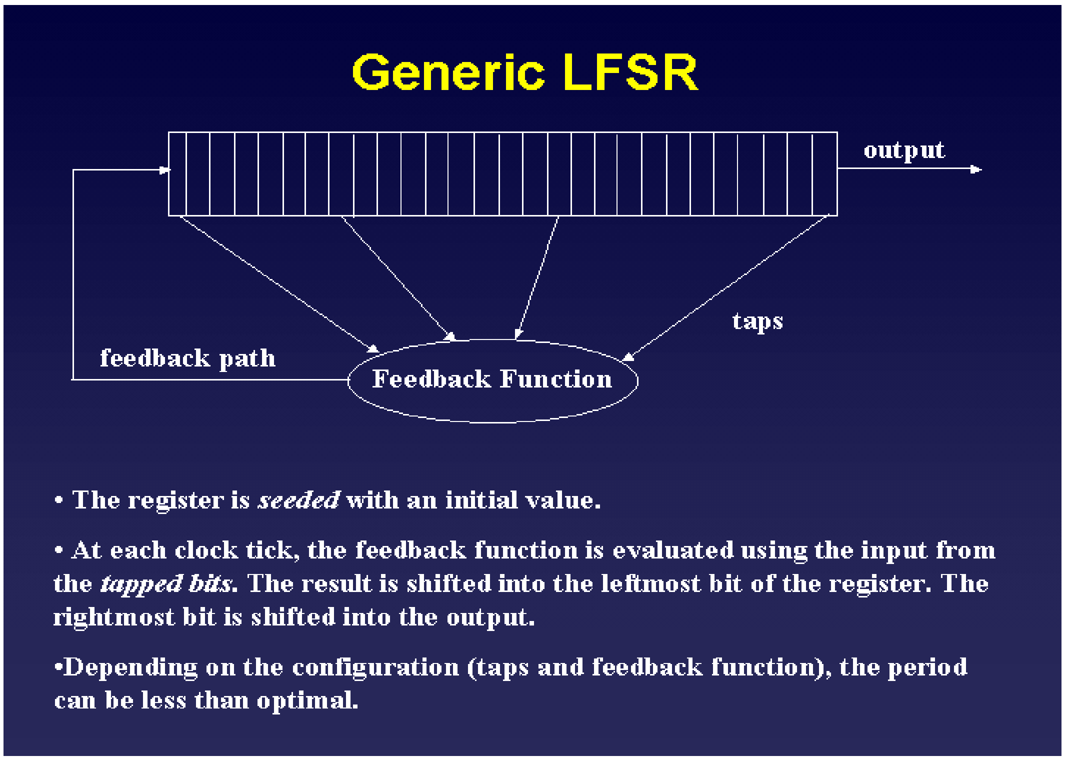
<u>Sixth step:</u> here the host finally decodes the sector by using the title key and the sector key, that can be found in the sectors header.

The decryption

The data is decrypted using the Linear Feedback Shift Registers (LFSRs). From this we get a stream of pseudo-random bits, that we XOR with the data that we want to encrypt. Now the host can recreate the bit-stream and XOR again, that thus get the plaintext.

LFSR

The generic LFSR takes a seed as input and with each clock tick some predefined bits are given to a "feedback function". Now the output of the feedback function is given being shifted into the register again. The output from the normal LFSR is the bits that is shifted out.



Generic LFSR

- The register is ***seeded*** with an initial value.
- At each clock tick, the feedback function is evaluated using the input from the ***tapped bits***. The result is shifted into the leftmost bit of the register. The rightmost bit is shifted into the output.
- Depending on the configuration (taps and feedback function), the period can be less than optimal.

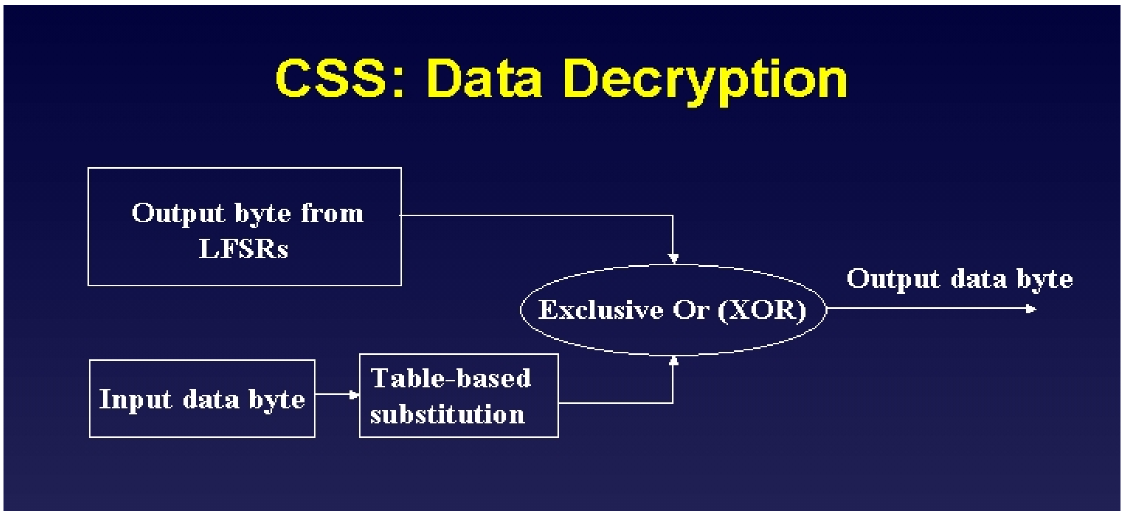
They way CSS use LFSR is a little different from the normal way, because they actually throw the bit that is normally considered output away. They actually use 2 LFSRs, one with 17 bits and one with 25. They set bit4 (counting from 0) to 1 in the initial state of both registers to prevent null-cycling, in case all bits of the seeds are 0. Instead of using the bit shifted out as output, it uses the output of the feedback function as output. So far we have seen three articles that state three different sets of bits that are used for the feedback function, meaning for the rest of the the report, we will not refer to the actual bits that are used for taps, and any pictures taken from our sources may have different opinions on which bits are used.
This is not a serious problem as it should be possible to actually test which bits are used, using a legal player... We have not done this, though.

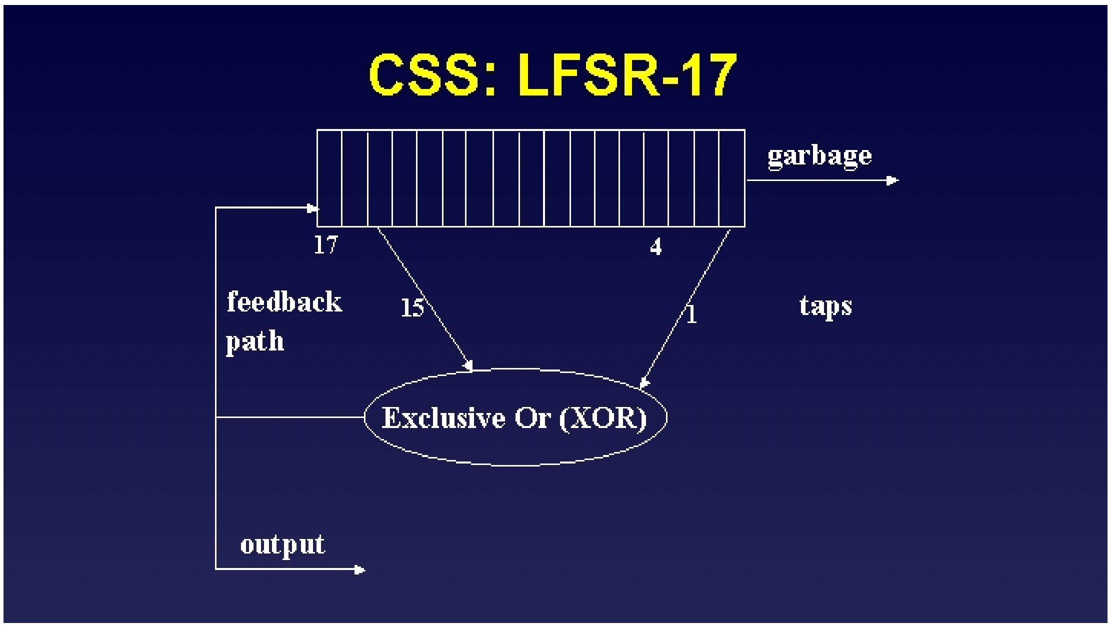
How is the output used:

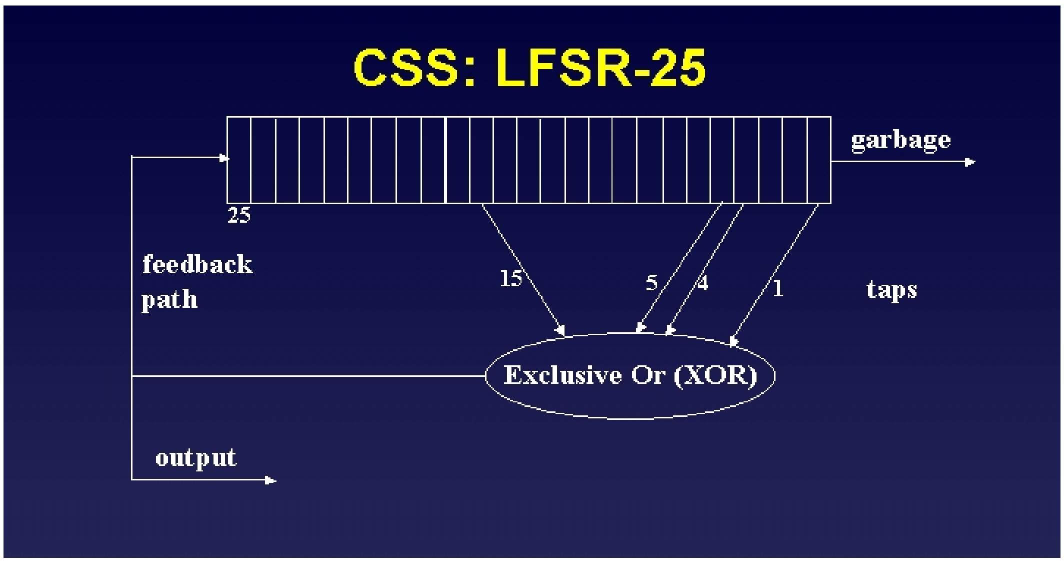
We combine the output from the 2 LFSRs using 8-bit addition, meaning when we have made 8 ticks, 1 byte, from each register, then we add the bytes together with an 8-bit adder, including the carry from the last round. This means that if we output 2 bytes from each, then the second byte will be LFSR17Out(2) + LFSR25Out(2) + carry(1).

The fun, or sad, thing about this, is that its not really a good way of using the LFSRs, its actually pretty weak. They could have used more LFSRs and done something more complicated fx using different clock rates, but they didnt.

The way we actually use the LFSRs is by seeding them with the key given to the en/decrypt, first two bytes are given to the LFSR17 and the three last are given to the LFSR25.
Now that we have seeded both the LSFRs, we can create the bit stream that we talked about above, and XOR it with the plaintext to generate the ciphertext. They also used a so called S-box, before they do the XOR. What the S-box does, is basically obscure the relationship between the key and the ciphertext. The reason is really unknown, since we know the S-Box, we will not reverse the S-Box, but refer to it as a one-way function F, since it is not bijective.



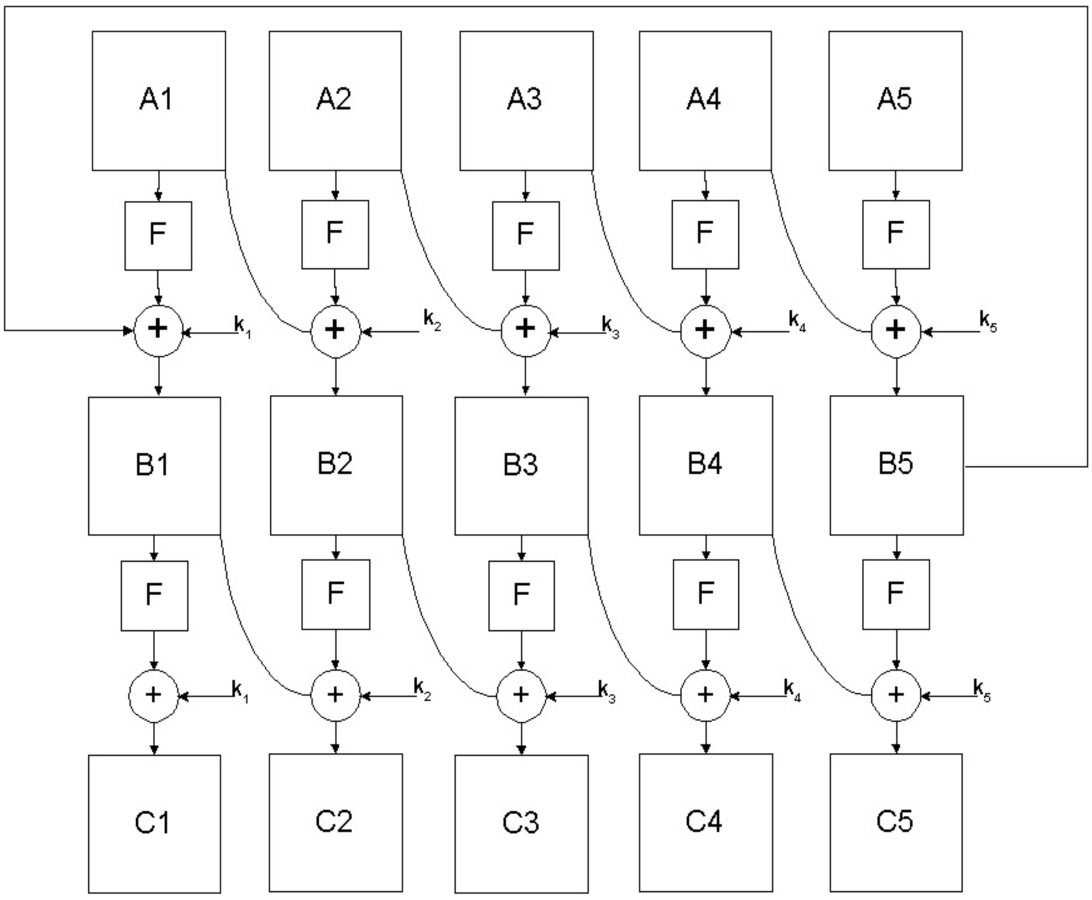




CSS: LFSR-25

Key decryption

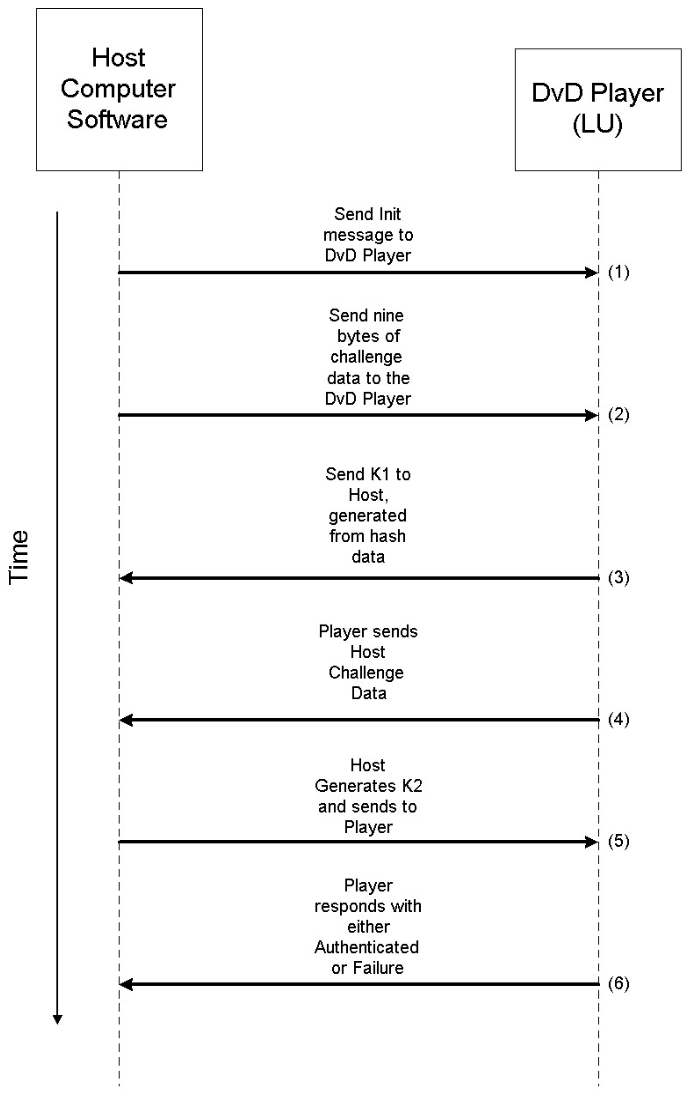
Besides decryption described earlier, CSS does a little more, to protect the keys. It uses a two-step mangling operation, so that the keys arent just in plaintext. This is done as shown in the picture. Decryption(data, key), data and key are 5 bytes long each. A1...A5 is the data bytes, k1...k5 is the first 5 output bytes when the LFSR streamcipher is given the key as seed and C1...C5 is the output bytes of the decryption. F is a one-way function (not-bijective) and is implemented with lookup-tables. B1...B5 are intermediate stages that will be used when discussing the attacks on CSS.



Mutual Authentication:

This is used between the player and the host, so that they “know” that they are talking to each other. During the authentication, they also make the session key, that they need for data transfer later, so that people cant just read the plaintext when they are communicating. The session key created is called the session key or bus key. Authentication starts with the host requesting an Authentication Grant ID(AGID) from the drive. Next, the host created a nonce and sends it to the drive. Now the drive encrypts the nonce, and sends it back to the host, so the host can decrypt it, and verify that the drive actually has the correct key and knows the right algorithm. The same thing is done the other way around, for the drive to make sure, that its actually talking to the host. A problem here is that this is a secret key system, and that the key has to be within all DVD players.

But now, both the host and the drive have a nonce from each other, and then they combine them, encrypt them, and uses the result as the session key, which is now used to encrypt the data they send between each other.


Host Computer Software
DvD Player (LU)
Time
Send Init message to DvD Player
(1)
Send nine bytes of challenge data to the DvD Player
(2)
Send K1 to Host, generated from hash data
(3)
Player sends Host Challenge Data
(4)
Host Generates K2 and sends to Player
(5)
Player responds with either Authenticated or Failure
(6)

CSS ATTACKS:

CSS has a number of flaws.

Firstly, the key bit-length is only 40 bits, which isn't much, so it's susceptible to brute-force attack. Secondly, one can find another playerkey in ~2^{16} time, by using an already known playerkey.

Mangling Process

If we look at the Mangling function then we see it uses output from the LFSR streamcipher (5 bytes to be exact).
Let us assume we know C1...C5 and A1...A5 (input and output).
The problem is that if we guess 1 byte correct, then we can calculate the other bytes.

If we guess k5 (2^8 possibilities, not much), then we can just reverse the whole operation.

B5 = XOR(F(A5), A4, k5)
B4 = XOR(F(B5), C5, k5)
k4 = XOR(F(A4), A3, B4)
B3 = XOR(F(B4), C4, k4)
k3 = XOR(F(A3), A2, B3)
B2 = XOR(F(B3), C3, k3)
k2 = XOR(F(A2), A1, B2)
B1 = XOR(F(B2), C2, k2)
k1 = XOR(F(A1), B5, B1)

We can verify by calculating the output, if we do not get what we already had, try another possibility of k5.
ie. C1 = XOR(F(B1), k1)

LFSR attacks

There are 2 attacks on the LFS Registers, both of which require that we know the output of the streamcipher.

We have come across a good deal if inconsistency, as to how many bytes are needed for verification. Either 2 or 3 bytes, where 2 bytes are questionable with regards to accuracy of your verification, 3 bytes means we have to assume that Stevenson is correct in his statement (which he does not prove).

6 bytes known:
If we know 6 bytes of output from the streamcipher, then we can calculate the key used for the cipher in 2^{16} time.

This is done simply by guessing the initial state of LFSR17 and then clocking out 3 bytes.
These 3 bytes will then uniquely determine 3 bytes as output from LFSR25 in order to match the 3 first bytes of the cipher output that you already know.

Since the new bit after a clock is linearly dependant on the state before the clock, we get a system of linear equations with 24 equations and 24 unknowns (bit 4 is 1 in the initial state, hence known).

Solving these gives us the initial state of LFSR25, and thereby we have the entire key used for the cipher.

We can verify by clocking out an additional 3 bytes from each register and checking them against the last 3 bytes of the known cipher output.

Alternatively we could clock 4 bytes out of LFSR17 instead of 3, and we'd then have a complete state of LFSR25, which means we can backtrack to the initial state rather easily.
This only leaves us with 2 bytes to verify our choice with, and it might be debatable wether that is enough.

5 bytes known:
Okay, we rarely have 6 bytes of known output, but we CAN actually get our hands on 5 bytes.
This is rather similar to the case with 6 bytes, except for a few oddities.

Stevenson states (with no proof) that by clocking out 2 bytes from LFSR17 and calculating the corresponding bytes from LFSR25 - from which we get a system of linear equations - we can calculate everything about the initial state of LFSR25 except for the most significant bit, which means we have 2 possible states, which is correct can be tested by using it to decrypt more data.

However if we only use 2 bytes to verify with, then we can clock out 3 bytes and calculate the initial state of LFSR25 completely.

We have not tried to prove the statement that Stevenson makes, nor have we looked much into wether or not 2 bytes is enough for verifying...
Intuitively we have $1 / 2^{16}$ chance to happen to get a "false" match from the 2 bytes, compared to the $1 / 2^{24}$ chance we have with 3 bytes... That's a factor 256.

Playerkey attack

Assuming you have the disk key, then you can actually calculate all other playerkeys that have been used to encrypt the disk key, since there is an encrypted version of the disk-key for every playerkey, on the DVD.
We know the disk-key (DK) and the encryption of the DK under a playerkey PK_i (call it DK-PK_i).
We already know DK and DK-PK_i, which is the output and the input of the decryption algorithm respectively. This means we can use the attack on the mangling process to get 5 bytes of output from the LFSRs, and then use the attack on the LFSRs to figure out the key.

This is a $2^8 + 2^{16}$ attack.

Hash attack

The REALLY big flaw is the fact that they verify the disk-key by decrypting the hash of the disk-key with the disk-key, which should produce the disk-key.

This is very bad because we can reverse the process, combined with the other attacks.
The point of the attack is by knowing ONLY the hash-value of the disk-key, we can figure out the disk-key itself:

Figure out k1, ... , k5, which are used in the mangling process.
These are 5 bytes of output from the streamcipher, and we know we can break that then, which will give us the disk-key.

To begin with we need to write a table with possibilities of C2 and B1 from the mangling process, indexing it.
At each entry will be the possible values of k2.

You calculate k2 = XOR(F(A2), A1, B2), for all values of B2.

And for each k2 we do:
C2 = XOR(F(B2), B1, k_2) for all B1.

Here F is the lookup table for mangling.

For the table this means, if we choose B1 and C2 as indexes, what possibilities of k2 can then make this happen?
Since C2 is dependant on k2 and B1 and also on the xor between k2 and F(B2), then you can get different values of k2, for a fixed C2 and B(1), where k2 xor F(B2) of course is fixed as well... (If it wasn't you wouldn't get C2 :P)
This is because, even though k2 and B2 differ, then k2 xor F(B2) might "hit" the same value multiple times.

It's also noted that there can be 0-8 possibilities for k2 at each entry, if 0 then that C2 was never calculated, which means we can ignore that combination of C2 and B1 later on.
This takes 2^{16} time since there are 2^8 possibilities for B1 and 2^8 possibilities for k2, and you go through all k2 for each B1 possibility.

In any case this is very little time used compared to the 2^{24} time we will use next. (factor 2^{8} smaller to be precise!)

We will make a table that for all possibilities of 3 output bytes from LFSR25, namely the first, second and fifth one, shows the initial state of LFSR25. This is because we know the output of the streamcipher of these bytes at that time (noted later).
This table has 2^{24} entries since 3 bytes = 24 bits, and each entry corresponds to an initial state of LFSR25, which means if we know these three bytes we also know the initial state of LFSR25.
We can do this because there is a one-to-one correspondence between byte 1,2 and 5 and the initial state of the register. (not proven by our sources)
This takes 2^{24} time obviously.

Now we go through all possibilities for LFSR17 initial state.
For each guess we output the first five bytes of output from it, and guess on B1.
Since we guessed the initial state of LFSR17 we also have C1 and C2 since the key input is the output. Herein lies the fault.
We use this knowledge to calculate k1 = XOR(F(B1), C1)
and use k1 to calculate B5 = XOR(F(A1), B1, k1)
and finally use B5 to calculate k5 = XOR(F(A5), A4, B5).

For the next part we go through all k2 indexed by B1 and C2.
Now we know k1, k2 and k5 which are the output bytes of the streamcipher, as mentioned when we made the table with 2^{24} entries.

Lookup into the table and get the initial state of LFSR25.
Now we potentially have C1...C5.

In order to verify we can check if our calculated k2 is valid.
Just reverse the mangling process and calculate k3, k4, B2, B3 and B4.
Then use B2 to calculate k2* = XOR(F(A2), A1, B2)
If k2* == k2, celebrate, if not go back and continue looping until you find something.

This attack is complexity 2^{25} (due to the 2^{24} table plus more work) and is said to be extremely fast (~18 seconds on a Pentium III 450MHz CPU).

DeCSS:

DeCSS is a program that decrypts the CSS manually, by using a playerkey.
If one does not have a playerkey, it's not hard to find one, as described in the cryptanalysis of CSS.

It was developed by reverse-engineering the algorithm on a Xing player, as the playerkey was apparently easily available on it, so they could do chosen-plaintext attacks to figure it out.
The algorithm / cipher of CSS was posted shortly after to the mailing list of LiViD, a linux video player project, that was going to use DeCSS to make a linux DVD-player.
2 days later Frank Stevenson posted his cryptanalysis on CSS, which showed that it could be broken in far less time than the bitlength of the key.

DVD-CCA, the company that was licensing CSS, then sued more than 500 people who had posted DeCSS code on the net, with Jon Johansen as the creator of it.
His case ended in aquittal, as he had not actually devised the decryption program, but only the graphical user interface (GUI) for it.

The real creators of the DeCSS algorithm are unknown members of a group called Masters of Reverse engineering (MoRe), and Johansen said that both MoRe and a russian group called Drink or Die (DoD), was working on breaking CSS at the same time, however MoRe was faster with the GUI.

Johansen also stated that they had based DeCSS on a cryptanalysis by Derek Fawcus.

MPAA joined DVD-CCA in their lawsuits and sued 3 people for publishing the DeCSS code on the net and in a magazine, in New York, while DVD-CCA was dragging people from around the world to California to stand trial for posting DeCSS code on the net.

While the New York case (based on the Digital Millenium Rights Act (DMCA)) was successful, and imposed a restraining order on the 3 defendants, that barred them from ever posting the code again, it was not successful in barring them from doing anything else, and as such the defendants urged everyone to spread the DeCSS everywhere, by any means. The defendants themselves also linked to sites that spread DeCSS code, as this was not in the restraining order.

Meanwhile in California, the DVD-CCA had much more trouble winning, as they based their lawsuits on CSS being a trade secret, but it was ruled that CSS was not a trade-secret at the time the defendants were posting it, as the information was readily available on the net.

Furthermore, it was ruled that it was unconstitutional to drag people from outside California, to stand trial in California.

All of this was an attempt from the MPAA and DVD-CCA to prevent people from knowing about how CSS works, but this proved to backlash quite a bit, as the communities involved reacted by spreading the information and code everywhere, by any means, even by Haiku poems or by putting the information into the DNS.

As a sidenote, it was largely presented as an argument against DeCSS that it created the possibility of lossless copying of DVDs, where you could copy a DVD without losing quality like you do when copying a VHS.

This was quite false, as lossless copying had been used for as long as a year before the DeCSS case, by simply copying the DVD incl. encryption. Since they would be sold to people with legal players, then decryption was not really a problem. The decryption was only necessary for people who had no DVD players for their OS.

Alternatively, you could feed into the drivers of you computer and leech the datastream that is sent through them after a DVD player has decrypted the data, this is quite more sofisticated though, and requires a special system.

For more details about the lawsuits, see the resource list.

CPPM/CPRM and the 4C Entity

The CPPM(or CPRM) is a form of Digital Rights Management(DRM) for securing recorded entertainment content. The CPPM specification was developed by 4C Entity, which is the “four company entity” with people from Intel, IBM, Toshiba and Matsushita. It was designed for flexibility and renewability, balanced with ease of implementation. It utilizes cryptography as the way to keep media secure. The target was portable and removable physical media such as DVD media and flash memory, but is not limited to these types of media.

The official specification mentions no restrictions on the kind of data that this system can be used on. It is hard though, to find information about whether CPPM/CPRM is actually in use on normal DVD videos. The average user who wants to take a copy of a DVD might still be stopped by the CSS as it requires additional tools, to circumvent it, CSS is still in use today on DVD-video. CPPM is in use in DVD-Audio discs though.

The reason for CPPM being in use in DVD-Audio media, is that it the initial motivation for making this system, was the lack of copy protection on audio media. Mainly the big recording labels were interested in some kind of protection. Old CDs weren’t protected, but with the introduction of DVD-Audio media, there was a chance to do something about it. The music industry looked to DVD-Videos where CSS was already in use. This is when the 4C Entity was assembled, with the goal of finding a successor to CSS, to be used on DVD-Audio. They came up with a new system, with the very innovative name CSS2. Unfortunately for 4C CSS was rendered useless as a copy prevention mechanism, as the secret keys were revealed and the algorithm was reverse engineered. Fortunately this happened before CSS2 ever made it out of the design phase, so they had the chance to discard it and come up with a new system. The new system turned out to be CPPM/CPRM. The reason why CSS2 was dropped, was because no one wanted to have anything to do, with something related to CSS anymore.

The market for DVD-Audio never really got as big as CDs or DVD-Video. In fact the place where 4C encryption gets used the most is in SD Cards, and DVD-R, -RW and –RAM formats, using the CPRM system.

When saying CPPM/CPRM and not just CPPM, it’s because they are actually two different schemes that get applied in different cases, but very similar in the way they function. CPRM is Content Protection for Recordable Media, that is all the media people buy when they want to make copies at home (DVD-R, DVD-RW and so on). The idea is that when you initially burn some copy-right protected data onto the disc, it is encrypted using a unique serial number on the disc, as part of the encryption key. This still allows you to copy the data to another disc, but the new disc won’t be able to decrypt correctly, because it will of course have a different serial number. CPPM (Content Protection for Pre-recorded media) on the other hand, is used on discs sold by official manufacturers and not something used on normal writable media.

CPPM does not use the discs serial number, but instead something that 4C invented for this particular system. It was something IBM and Intel had been working on together in the past, and it resulted in a system called Media Key Block.

Media Key Blocks

The motivation for making Media Key Blocks was the need for some kind of authentication between the DVD media and the players and recorders. DVD discs are very passive in nature, and there is no chance that you can interact with it - you can only read data from it.
The Media Key Block solves that problem, by implicitly authenticating the device playing the disc. The key block is an array that is stored on each DVD, containing encrypted values. To be able to play the content of the DVD, a compliant device has to have a valid set of keys. This key set is used to process the Media Key Block and obtain a key that can be used to decrypt the content of the DVD.

The key block stored on a DVD is sold to licensed DVD media manufacturers and likewise with the key sets in players/recorders. Because these are kept secret, they can be used to do the authentication. If a device is able to play the protected DVD, then the authentication has already happened implicitly because a non-authentic device wouldn't have been able to play the content. The Media Key Block also has the property of being able to block stolen keys. If it is discovered that someone have got their hands on licensed keys, then future DVD media will have their Media Key Block modified, so that it no longer works with the leaked keys. Media Key Blocks are only unique for each released title, so two original copies of some Disney movie, will have the same key block, given that they are produced roughly at the same time.

I mentioned that a player/recorder device would be able to play the content of a DVD if it can obtain the right key from processing the Media Key Block. Obtaining the key though, only takes you half the way. Next step is to use to derived key to decrypt the content on the DVD. For this is used something called the Cryptomeria Cipher.

Cryptomeria

Cryptomeria (also called C2) is a proprietary block cipher made also by the 4C Entity. It is basically a 10 round Feistel cipher using a symmetric key, meaning that the keys for encryption and decryption are trivially related or identical. The advantage you get using a Feistel cipher is that encryption and decryption is very much the same. For encryption, there is a key schedule that produces a key for each of the 10 rounds that you iterate, and to decrypt all you have to do is do the encryption again, but using the keys in reverse order.

What distinguishes different Feistel ciphers is the choice of round function. This is a special function that can be very complex and it doesn't have to be invertible, as many other functions have to be and other crypto-ciphers.

The general scheme for a Feistel cipher is to cut the plaintext in two halves – a right and left half. We call these L_i and R_i for the i'th round. At first the plaintext is whatever text we want to encrypt. In the second round, the plaintext is the output from the first round and the output from the second goes as input to the third round and so on. For some i we let R_i become L_{i+1}, so the right side goes unmodified into the next round and become the left side of input for round i+1.
The right side also goes into the round function, which also take a roundkey as input. The output from the round function is then XOR'ed with the left half and the result of that become R_{i+1}.

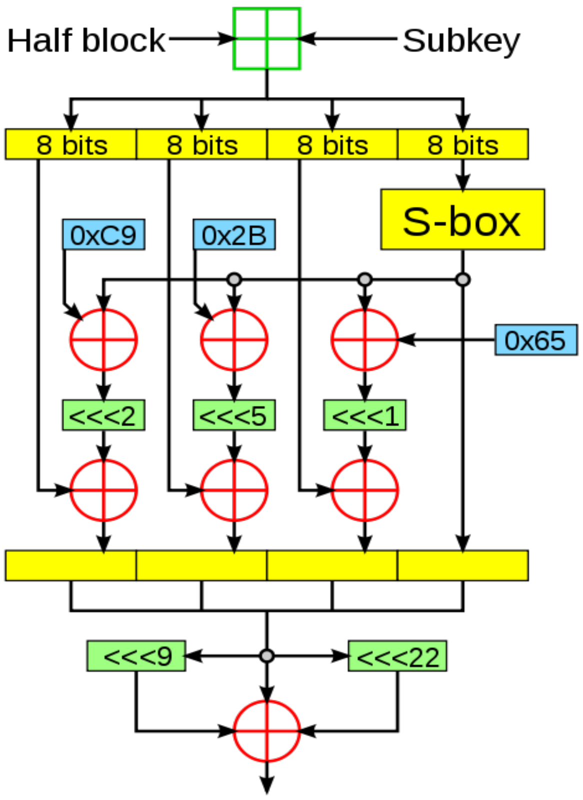
This describes one round of a Feistel cipher, and this is also how the Cryptomeria cipher works. What we also want to look at is what make Cryptomeria unique, namely its round function. This is a bit more complicated, and the official description and specification of this function is made public as a code example written in C++. This makes interpretation a little more difficult due to a very low level description, but of course helps those who want to implement it.

Here follows a basic description of what happens in the round function of Cryptomeria. First half of the plaintext and a key are added modulo 32 into a 32 bit string. The 32 bit is cut in four pieces with 8 bit in each part. We call these four pieces for Z_i for i = 1,...,4. Then we take Z_1 and run it through an S-box. S-box is short for substitution box, and all it does is basically substitute bits in the output. In the case of Cryptomeria the S-box outputs a bit string of length equal to the input string. What exactly is inside this S-box is the secret in CPPM/CPRM. What you pay for if you want to buy a license from 4C is some secret constants that make up this S-box. The rest of the algorithm is open for peer-review, which by the way was a quite new approach back when the algorithm was released.

The next thing that happens is that we do three XOR operations. What we XOR is a copy of the output from the S-box in each of them and a predefined byte constant that is different for each of the XOR operations. Then the three values we that get are rotated 1, 5 and 2 places respectively, to the left. After that we XOR each of the results with Z_2, Z_3 and Z_4 respectively. Then we construct a new 32 bit string consisting of the three results from the XOR operations we just made, and the S-boxed 8 bit string. This new string is made in three copies. One copy is rotated 9 places left, one copy is rotated 22 times left and the last copy is unchanged. These three 32 bits strings get XOR'ed and the result of that is the output of the round function.

The key that is part of the input of the round function is a new key for each round.
The 10 keys that are needed for this, is constructed from a “master key” K. With this you make a intermediate key K_i' which is used to make a round key rk_i. In the making of these round keys the S box is also used – the same S-box as is used in the round function.

The figure below illustrates how the round-function works:



Attacks on Cryptomeria

There are four different attacks that we have looked at, on Cryptomeria.
The first one is an attack where you try to recover the S-box with a chosen key attack. Apparently there is a flaw in the key schedule, so that some master keys generate only three different inputs to the S-box. This happens in the part of the key scheduling where you take out 8 bits of the intermediate key, and use these as input to the S-box. The fact that only three different round keys are created, allows you to somehow guess what the ciphertext will be after the 7th round in the encryption. When you do the actual encryption, you can compare your guess with the result and if your guess what right, you have learned something about what is inside the S-box.

The second attack is a boomerang attack. Boomerang attacks are based on differential cryptanalysis. The basic idea is to observe the effect differences between plaintexts have on the ciphertexts, and discover where the cipher does something non-random during the encryption.
We use a boomerang attack to observe when the S-box gets the same input in different rounds. The attack is used to recover bits of the first round key used.

The third attack is a chosen ciphertext attack, used to recover both key and S-box. This is also based on a boomerang attack, which is used to recover the least significant 22 bits of the first and last round key. The remaining bits of the two keys and 1 S-box entry can be recovered in 2^{52}. After that we can get the second round key aswell, and then determine the master key uniquely.

A brute-force attack was also attempted in 2004 to recover S-box values. This was made as a distributed computing project. The motivation behind it was that a japanese tv station was sending HDTV with an encrypted signal using CPRM. After three months the entire 56bit key space had been scanned, but for unknown reasons no valid keys were found.

Further information about the Cryptomeria cipher and attacks on it, see the cryptanalysis in the resource list.

WinDVD

At some point in time there was a DVD player called WinDVD, it had a peculiar flaw that was exploited by some programmers.

They made a patch for it that because of the flaw, made it possible to change the destination of the output, meaning you could send the output of WinDVD to a file.

This practically disabled every and all security measures on all medias that were playable by WinDVD, and easily so, since anyone could just apply the patch and start ripping the media of its protection.

It's uncertain wether or not this has been successfully patched, and if you can still use the older version to rip new DVDs and the like. We have been unable to find any concluding statements at least.

Conclusion

People question wether the cipher used for CSS could even be regarded as a real cipher, considering how easily it's broken. This, and all the lawsuits trying to keep CSS a secret even after it's been figured out, tell us that the way CSS protected DVDs were by being a secret technology.

Everything that happened around CSS testaments to the futility of trying to protect information by having a secret way of doing so, because once the secret is out then there's no protection.
And the secret always gets out sooner or later.

On top of that we have the poor design of CSS, coupled with the export restriction on cryptographic algorithms not having keylength more than 40 bits, which obviously makes a huge impact on the complexity of attacks made against it.
By that time's standards 40 bits was bruteforceable with a lot of computing power, like f.x. the US government has, but as time goes on, even 40 bits is bruteforceable now by smaller scale computer networks in a reasonable time.

Then we have the supposed successor of CSS: CPPM
CPPM is using a cipher called Cryptomeria which works on 56 bit keys...
56 bit is simply too short, only a 16 bit upgrade from CSS which was bruteforceable even back then.
Everyone knows we have to amp up the bitlength as time goes by, because computers get more and more powerful, so brute-forcing becomes more and more of an actual option at those bitlengths.

In the end we are left with a so-called protection technology which can be broken in seconds and a successor which is rarely used for anything except DVD-audio and a few other medias.

We conclude that getting confidentiality by keeping something a secret or by having a confusing process is simply futile. The secret will be figured out and the process will be mapped and understood at some point, and once those are done, we can find and attack whatever vulnerabilities the algorithm may have.

Resource List

DVD:
noget til disk sectors osv: http://www.osta.org/technology/dvdqa/dvdqa6.htm
(Definition of Host: and a host device, which usually refers to a decoder board or software application.)

CSS and cryptanalysis of CSS:
http://www.cs.cmu.edu/~dst/DeCSS/Kesden/index.html
http://www.lemuria.org/DeCSS/crypto.gq.nu/
(Alternative: http://cyber.law.harvard.edu/openlaw/DVD/resources/crypto.gq.nu.html)
http://www.cs.cmu.edu/~dst/DeCSS/Cherry/index.html
http://dvd-copy.blogspot.com/2005/08/dvd-content-scrambling-system.html

DVD-CCA / MPAA lawsuit:
http://w2.eff.org/IP/Video/DVDCCA_case/
http://www.legal.wao.com/decss.html
http://cyber.law.harvard.edu/openlaw/DVD/resources.html (and sublinks)
http://w2.eff.org/IP/Video/DVDCCA_case/20040122_eff_pr.php
http://www.lemuria.org/DeCSS/gilc.html
http://w2.eff.org/effector/HTML/effect12.04.html

MoRe / Jon Johansen and the creation of DeCSS:
http://www.lemuria.org/DeCSS/dvdtruth.txt
(Alternative: http://www.defacto2.net/groups/dod/more+dod_decss.txt)
http://www.lemuria.org/DeCSS/decss.html
http://web.archive.org/web/20001202051300/http://livid.on.openprojects.net/pipermail/livid-dev/1999-October/000548.html

General DeCSS:
http://www.lemuria.org/DeCSS/
http://www.cs.cmu.edu/~dst/DeCSS/
http://www.cs.cmu.edu/~dst/DeCSS/Gallery/index.html

Source code for DeCSS:
http://torch.cs.dal.ca/~hannon/decss.c

42 methods for spreading DeCSS:
http://decss.zoy.org/
http://www.cs.cmu.edu/~dst/DeCSS/Gallery/decss-haiku.txt

CPPM:
http://events.iaik.tugraz.at/weworc09/9aa510c7c7aab1/abstracts/04.pdf (cryptanalysis)
http://www.intel.com/standards/case/case_cp.htm
http://edipermadi.files.wordpress.com/2008/08/cryptomeria-c2-spec.pdf
http://www.4centity.com/docs/How%20CPRM%20Works.pdf

WinDVD:
http://www.highfidelityreview.com/news/news.asp?newsnumber=14550899