The CSS Decryption Algorithm

David S. Touretzky
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213-3891

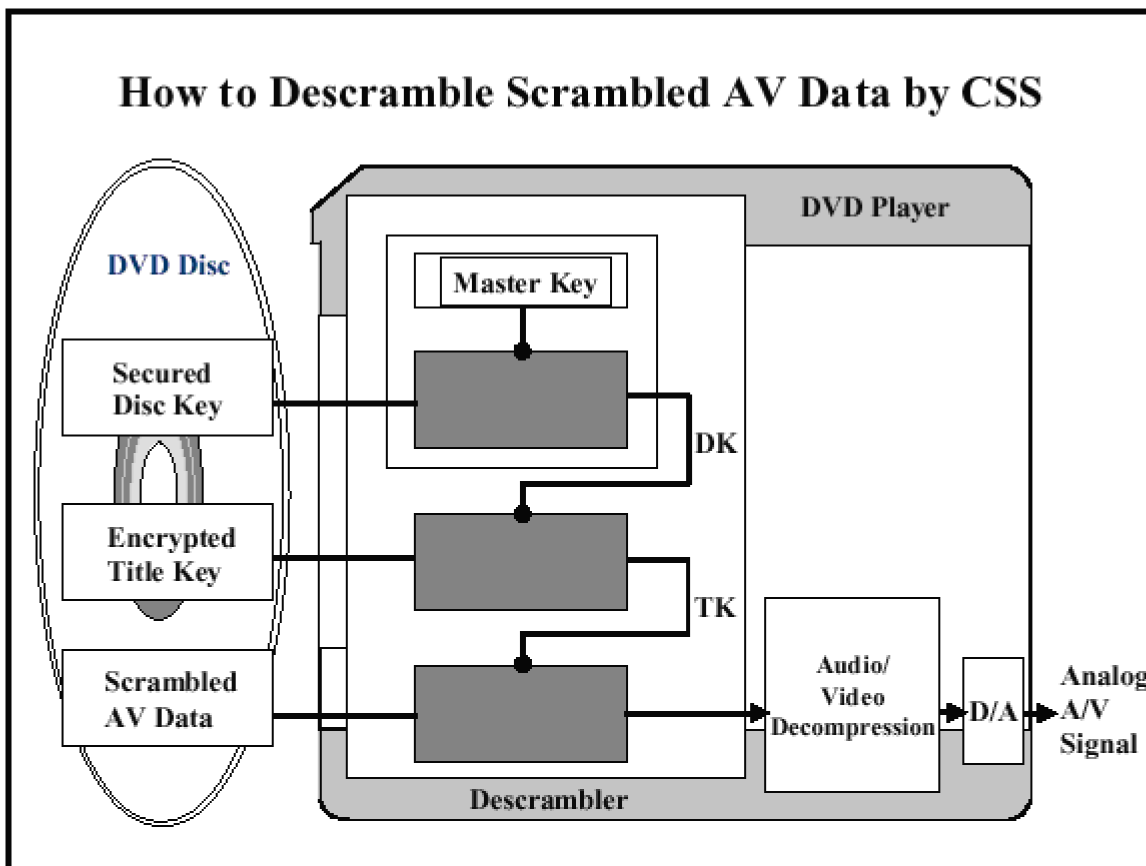
email: dst@cs.cmu.edu

Version 2.2, dated July 10, 2000

In this paper I describe, in plain English, the algorithm for decrypting DVDs that have been encrypted using CSS, the Content Scrambling System. The C source code for this algorithm was posted anonymously to the LiViD mailing list. I have relied upon this posting, and the css-auth source code written by Derek Fawcus, in creating my description. Further insight into the operation of the algorithm can be obtained by reading Frank Stevenson's essay, "Cryptanalysis of Contents Scrambling System", available on the web.

In this paper, arrays are indexed from zero. Integers are assumed to be four bytes in length. Bytes are assumed to be automatically coerced to integers as necessary. Hexadecimal constants are represented using C notation, 0x##, or 0x###, where ## or ### are hexadecimal numerals from 0 to F.

The tables referenced in the procedures below are given at the end of the paper.



(Illustration borrowed from a web site in Norway.)

Decrypting a DVD movie is done in several steps, as shown in the figure above. First one must have a **master key**, which is unique to the DVD player manufacturer. It is also known as a **player key**. The player reads an encrypted **disk**

key from the DVD, and uses its player key to decrypt the disk key. Then the player reads the encrypted **title key** for the file to be played. (The DVD will likely contain multiple files, typically 4 to 8, each with its own title key.) It uses the decrypted disk key (DK) to decrypt the title key. Finally, the decrypted title key, TK, is used to descramble the actual content.

I will begin by describing a procedure named **CSStitlekey1** that uses a player key to decrypt the disk key.

The procedure returns no value. (It is of type "void".)

The procedure takes two arguments.

The first argument is named KEY, and is a pointer to a vector of six unsigned bytes. These bytes initially contain an encrypted disk key. They will eventually hold the decrypted disk key computed by the procedure.

The second argument is named im, and is a pointer to a vector of six unsigned bytes. These bytes are the decryption key (the player key) that the procedure will use to decrypt the bytes in the variable named KEY.

The procedure makes use of several temporary (local) variables.

Temporary variables t1 through t6 are unsigned integers.

Temporary variable k is a vector of five unsigned bytes.

Temporary variable i is an integer, used as a loop index.

The body of procedure CSStitlekey1 is as follows:

1. Take byte 0 of im, OR it with the hexadecimal constant 0x100, and store the result in t1.
2. Take byte 1 of im and store it in t2.
3. Take bytes 2-5 of im and store them in t3.
4. Take the low order three bits of t3, which can be computed by the AND of t3 with the constant 7, and store the result in t4.
5. Multiply t3 by 2, add 8, subtract t4, and store the result back in t3.
6. Store 0 in t5.
7. Begin a loop by initializing i to 0. This variable will range from 0 to 4, and will be used to index the variable k, which holds a five byte intermediate result in the decryption of the six byte key.
8. Continue looping while i is less than 5, incrementing i by 1 on each subsequent pass through the loop. When i is equal to 5, exit the loop by jumping to step 20.
9. Use t2 as an index into the table CSStab2, and retrieve a byte, which we'll call b1. Use t1 as an index into table CSStab3, and retrieve another byte, which we'll call b2. Compute b1 XOR b2 and store the result in t4.
10. Shift t1 right by 1 bit, and store the result in t2.
11. Take the low-order bit of t1 (which can be obtained by taking the AND of t1 and the constant 1), shift it left by 8 bits, and XOR it with t4. Store the result back in t1.
12. Use t4 as an index into the table CSStab4, and retrieve a byte. Store the result in t4.

13. Shift the contents of t3 right by 3 bits, XOR it with t3, shift the result right by 1 bit, XOR it with t3, shift the result right by 8 bits, XOR it with t3, shift the result right by 5 bits, and extract the low order byte by ANDing it with the hexadecimal constant 0xff. Store the result in t6.

14. Shift the contents of t3 left by 8 bits, OR it with t6, and store the result in t3.

15. Use t6 as an index into the table CSStab4, and retrieve a byte. Store the result in t6.

16. Add together t6, t5, and t4, and store the result back into t5.

17. Extract the low order byte of t5 (which can be done by ANDing t5 with the hexadecimal contant 0xff), and store the result in the i-th byte of the vector k.

18. Shift t5 right by 8 bits and store the result back into t5.

19. Return to step 8 to continue looping.

20. This is where we end up when the first loop is complete.

21. Begin another loop by initializing the variable i to 9. This variable will range from 9 down to 0. The values of (i+1) and i will be used to index into the 11 byte table CSStab0, whose elements are of course numbered from 0 to 10. This table describes a permutation of the 6 byte key; its elements are integers from 0 to 5.

22. Continue looping while i is greater than or equal to 0, decrementing i by 1 on each subsequent pass through the loop. When i is less than 0, exit the loop by jumping to step 25.

23. Use i+1 as an index into the table CSStab0, and call the retrieved value p1. Use i as an index into the table CSStab0, and call the retrieved value p0. Use p1 as an index into the vector k, and call the retrieved value b1. Use p1 as an index into the vector KEY, and use the retrieved value as an index into the vector CSStab1; call the retrieved value b2. Use p0 as an index into the vector KEY, and call the retrieved value b3. Compute b1 XOR b2 XOR b3, and store the result in KEY, in the byte indexed by p1.

24. Return to step 22 to continue looping.

25. This is where we end up when the second loop is complete.

26. Return from the procedure.

Now I will describe a procedure named **CSStitlekey2**. This procedure uses the decrypted disk key to decrypt a title key.

The argunents to this procedure, KEY and im, are the title key and the decrypted disk key, respectively.

Procedure CSStitlekey2 is identical to CSStitlekey1, except that in step 15, it uses the table CSStab5 instead of CSStab4. Note that CSStab5 is the bitwise complement of CSStab4.

Now I will describe a procedure named **CSSdescrypttitlekey**. This procedure uses a built-in player key to decrypt a disk key and a title key.

The procedure returns no value. (It is of type "void".)

The procedure takes two arguments.

The first argument is named TKEY, and is a pointer to a vector of six unsigned bytes. These bytes initially contain an encrypted title key. They will eventually hold the decrypted title key computed by the procedure.

The second argument is named DKEY, and is a pointer to a vector of six unsigned bytes. These bytes contain the encrypted disk key.

The procedure makes use of several temporary (local) variables.

Temporary variable i is an integer, used as a loop index.

Temporary variable im1 is a vector of six unsigned bytes.

Temporary variable im2 is a vector of six unsigned bytes holding the player key. It is initialized to the hexadecimal constants 0x51, 0x67, 0x67, 0xc5, 0xe0, and 0x00.

The body of procedure CSSdecrypttitlekey is as follows:

1. Copy the six bytes of the vector DKEY to the vector im1. This can be done with a for loop using i as the index variable.

2. Call CSStitlekey1 with arguments im1 and im2. The side effect of this call will be to leave a decrypted disk key in im1.

3. Call CSStitlekey2 with arguments TKEY and im1. The side effect of this call will be to leave a decrypted title key in tkey.

Now I will describe a procedure named **CSSdescramble**. This procedure decrypts one sector of a DVD, which is 2048 bytes long. (The length is 0x800 in hexadecimal.)

The procedure returns no value. (It is of type "void".)

The procedure takes two arguments.

The first argument is named SEC, and is a pointer to a vector of 2048 unsigned bytes. These bytes initially contain the encrypted disk sector. They will eventually hold the decrypted sector computed by the procedure.

The second argument is named KEY, and is a pointer to a vector of six unsigned bytes. These bytes contain the decrypted title key that will be used to decrypt the disk sector.

The procedure makes use of several temporary (local) variables.

Temporary variagles t1 through t6 are unsigned integers.

Temporary variable END is a pointer to the end of the 2048 byte vector to be decrypted. It is initialized to SEC plus 0x800.

The body of procedure CSSdescramble is as follows:

- 1. Retrieve byte 0 of KEY, XOR it with byte 84 (0x54 in hexadecimal) of SEC, treat the result as an integer, OR it with the hexadecimal constant 0x100, and store the result in t1.

- 2. Retrieve byte 1 of KEY, XOR it with byte 85 (0x55 in hexadecimal) of SEC, and store the result in t2.

- 3. Take bytes 2 through 5 of KEY and XOR them with bytes 86 through 89 (0x56 through 0x59) of SEC; store the result in T3.
- Steps 4 through 6 are the same as CSStitlekey1, but add a step 5A:
- 5A. Advance SEC by 128 bytes (hexadecimal 0x80).
- 7. Begin a while loop.
- 8. Continue iterating while SEC does not equal END.
- Steps 9 through 20 are the same as CSStitlekey1, except change CSStab4 to CStab5 in step 12, and chage step 17 to read as follows:
- 17. Use the byte pointed to by SEC as an index into the table CSStab1. Take the retrieved byte and XOR it with the low order byte of t5, which can be extracted by ANDing t5 with the hexadecimal constant 0xff. Store the result back in the byte pointed to by SEC. Then advance the pointer SEC by one byte.
- 21. Return from the procedure.

Table **CSStab0** is eleven bytes in length. Its elements are: 5, 0, 1, 2, 3, 4, 0 , 1, 2, 3, 4.

Table **CSStab1** is 256 bytes in length. It implements a simple substitution cipher. Its elements, expressed as hexadecimal constants, are:

```
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 **CSStab2** is 256 bytes in length. Its elements, expressed as hexadecimal constants, are:

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

Table **CSStab3** is 512 bytes in length. It consists of 64 repetitions of the following six-byte sequence: 0x00, 0x24, 0x49, 0x6d, 0x92, 0xb6, 0xdb, 0xff.

Table **CSStab4** is 256 bytes in length. It is a lookup table for efficiently reversing the order of bits in a byte. If we regard it as a 16x16 matrix stored in row major order, then it can be described as the Cartesian product of two 16-element sequences.

Define seqI as [0x00, 0x08, 0x04, 0x0c, 0x02, 0x0a, 0x06, 0x0e, 0x01, 0x09, 0x05, 0x0d, 0x03, 0x0b, 0x07, 0x0f].

Define seqJ as [0x00, 0x80, 0x40, 0xc0, 0x20, 0xa0, 0x60, 0xe0, 0x10, 0x90, 0x50, 0xd0, 0x30, 0xb0, 0x70, 0xf0].

With i and j each varying from 0 to 15, with j varying faster than i, the table entries can be described as table[i,j] = seqI [i] OR seqJ[j].

We can write out the table explicitly as:

```
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
```

Table **CSStab5** is 256 bytes in length. It is the bit-wise complement of table CSStab4.

We can write out the table explicitly as:

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

This concludes the description of the CSS descrambling algorithm.

Dave Touretzky
Last modified: Mon Jul 10 22:28:44 EDT 2000