Breaking DVB-CSA

Erik Tews[1], Julian Wälde[1], and Michael Weiner[2]

[1] Technische Universität Darmstadt, Fachbereich Informatik,
Hochschulstraße 10, 64289 Darmstadt
<{e_tews,jwaelde}@cdc.informatik.tu-darmstadt.de>
[2] Technische Universität München
<michaelweiner@mytum.de>

Abstract. Digital Video Broadcasting (DVB) is a set of standards for digital television. DVB supports the encryption of a transmission using the Common Scrambling Algorithm (DVB-CSA). This is commonly used for PayTV or for other conditional access scenarios. While DVB-CSA support 64 bit keys, many stations use only 48 bits of entropy for the key and 16 bits are used as a checksum. In this paper, we outline a time-memory-tradeoff attack against DVB-CSA, using 48 bit keys. The attack can be used to decrypt major parts a DVB-CSA encrypted transmission online with a few seconds delay at very moderate costs. We first propose a method to identify plaintexts in an encrypted transmission and then use a precomputed rainbow table to recover the corresponding keys. The attack can be executed on a standard PC, and the precomputations can be accelerated using GPUs. We also propose countermeasures that prevent the attack and can be deployed without having to alter the receiver hardware.

1 Introduction

Digital Video Broadcasting (DVB) is a set of standards for digital television in Europe. It has been standardized by the European Telecommunication Standardization Institute (ETSI) in 1994. DVB defines multiple standards in the field of digital television; well-known standards include terrestrial (DVB-T) [5], satellite (DVB-S) [3], and cable television (DVB-C) [4]. However, there are many more standards, e.g. for data broadcasting.

DVB supports encryption of payload using the proprietary Common Scrambling Algorithm (DVB-CSA). The main use cases are PayTV and conditional access for TV stations that only have regionally limited broadcasting rights. Other use cases are possible as well, e.g. encryption of IP-over-satellite data traffic [7].

DVB-CSA was not intended for public disclosure and manufacturers implementing it need to sign a non-disclosure agreement to get access to the specifications [1]. Only a few details about the structure of the encryption scheme were known from the standard [2], a publication [8], and a patent application [6]. The breakthrough leading to the full disclosure of DVB-CSA took place when FreeDec, a software implementation of DVB-CSA, appeared on the Internet in

2002. This implementation was reverse-engineered to extract the missing details of the cipher, such as the S-Box used.

The first academic publication analyzing DVB-CSA appeared one year later [13]. Other publications we found about DVB-CSA consider physical attacks, i.e. fault attacks [14] and side-channel attacks [9] or only analyze the stream cipher part of DVB-CSA [12], while DVB-CSA also contains a block cipher (see Section 2). However, those attacks do not work in a real-life scenario.

Common PayTV setups consist of four core components: a Smart Card, a Conditional Access Module (CAM), a set-top box and the television. The Smart Card is personalized to the PayTV subscriber and provides the DVB-CSA keys, which are changed frequently. It is able to compute the DVB-CSA keys based on a secret stored on the card and control messages from the TV station. The Conditional Access Module is the interface between the Smart Card and the set-top box. The CAM is either a PCMCIA card connected to the set-top box over the Common Interface (CI) or it is integrated into the set-top box. The set-top box decodes the MPEG stream and forwards it to the television, and the television finally displays the video.

All public practical attacks on encrypted DVB streams we know consider attacking the DVB-CSA key derivation scheme – this includes physical attacks against SmartCards as well as Card Sharing, i.e. distributing the DVB-CSA keys generated by a SmartCard to multiple users.

1.1 Our contribution

To the best of our knowledge, we provide the first practical attack on DVB-CSA itself. It can be used to determine DVB-CSA session keys within a few seconds, regardless of the key derivation scheme. We use pre-computed rainbow tables [10] for our attack and reduce the key space by exploiting the fact that most 64-bit DVB-CSA keys use 16 of the key bits as a checksum.

This paper is organized as follows: We first introduce the reader to the DVB-CSA encryption algorithm in Section 2. In Section 3, we outline that DVB-CSA keys (64 bit) usually contain only 48 bits of entropy and 16 bit of the key are used as a checksum. In Section 4, we examine the MPEG2 Video broadcasted by many TV stations, and show that it usually contains constant plaintexts. In Section 5, we use this fact combined with the reduced key space to show that a time-memory tradeoff can be used against DVB-CSA, using *rainbow tables*. In Section 6, we show how tools that generate such tables can be efficiently implemented on CPUs and GPUs, and benchmark their performance. In Section 7, we suggest appropriate parameters for the generation of these tables. In Section 8, we present experimental results with a small table. In Section 9, we outline the overall attack scenario. In Section 10 we suggest countermeasures that prevent the attack and can be implemented with very low costs. We finally conclude in Section 11.

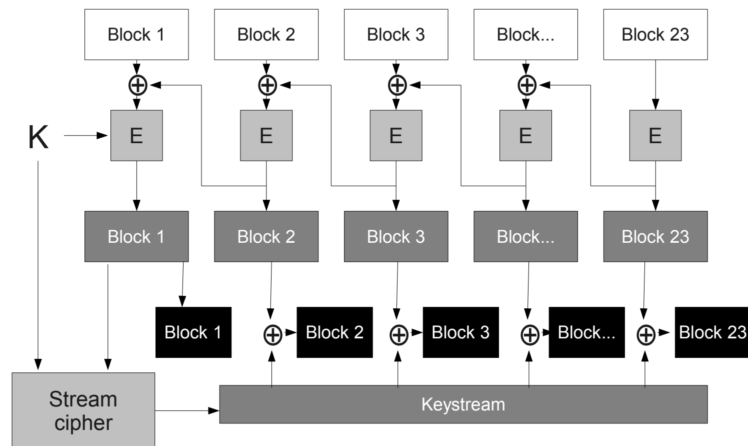
2 DVB-CSA in a nutshell

DVB-CSA is the symmetric cipher used to protect content of MPEG2 Transport Streams in DVB. To transmit multiple audio, video or general data streams on a single transponder, MPEG Transport Stream (MPEG TS) encapsulates all data streams in *cells* of 188 bytes. These cells consist of a 4 byte header and 184 bytes of payload. Optionally, an extended header can be embedded, reducing the payload size by the size of the extended header.

A flag in the MPEG TS header indicates whether the packet is unencrypted, or encrypted with the *even* or *odd* key. Usually, only one key is used, while the other key is updated by the CAM/SmartCard. For encrypted cells, only the payload is encrypted. The header or optional extended header is never encrypted.

DVB-CSA works in 2 passes. The first pass splits the plaintext of the payload into blocks of 64 bit length and a remainder that is smaller than 64 bit; all blocks except this remainder are then encrypted with a custom block cipher in CBC mode, using reverse block order and all zero initialization vector. In the second pass, a stream cipher using the first block (last block in the order used with the block cipher) as initialization vector encrypts all data again, except the first block. Note that DVB-CSA does not randomize the ciphertexts: Equal plaintexts are always mapped to the same ciphertexts.

Fig. 1. DVB-CSA structure



2.1 The DVB-CSA block cipher

We will concentrate on the custom block cipher used by DVB-CSA as our attack focuses on the first block of the ciphertext which does not depend on the stream cipher (see Figure 1). We define the variables used for the block cipher as follows:

Definitions

KEY	the 64 bit KEY
K_i	the ith byte of the 64 bit key
KS[i]	the ith round key (1 byte)
$R[i]$	the ith byte of the 64 bit plaintext
S	a 8 bit permutation
P	a 8 bit permutation that only swaps bits

Key Schedule The block cipher encrypts a single block in 56 rounds without any kind of additional input or output transformation. The key schedule of the block cipher expands the 64 bit key into 56 8 bit round keys using the bit permutation from Table 1.

$X_6 = \mathrm{KEY}$
for $i = 6 \to 0$ **do**
  $X_{i-1} \leftarrow \mathrm{permute_block}(X_i)$
end for
for $i = 0 \to 6$ **do**
  $KS[8 * i \ldots 8 * (i + 1) - 1] \leftarrow X_i \oplus (0x0101010101010101 * i)$
end for

Round function One round of the DVB-CSA block cipher consists of six 8 bit XORs, one S-Box lookup S (see the sourcecode in Appendix A) and a 8 bit bit-permutation P.

P permutes bit indices as in this table:

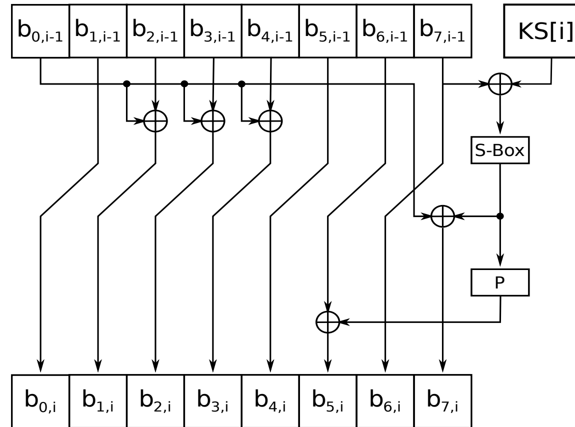
in	0	1	2	3	4	5	6	7
out	1	7	5	4	2	6	0	3

Table 1. permute_block(X)

swaps the bits in X as in this table (decimal notation):

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
00	19	27	55	46	01	15	36	22	56	61
10	39	21	54	58	50	28	07	29	51	06
20	33	35	20	16	47	30	32	63	10	11
30	04	38	62	26	40	18	12	52	37	53
40	23	59	41	17	31	00	25	43	44	14
50	02	13	45	48	03	60	49	08	34	05
60	09	42	57	24						

Fig. 2. DVB-CSA block cipher round function



3 Usage of DVB-CSA

In spite of the fact that DVB-CSA works with 64 bit keys, we observed that only 48 bit of entropy are used for many TV stations. The fourth and the eigth byte of the key in this case are the sum of the previous three bytes modulus 256:

k_0	k_1	k_2	$k_0 + k_1 + k_2 \bmod 256$	k_4	k_5	k_6	$k_4 + k_5 + k_6 \bmod 256$

This reduces the effort needed for an exhaustive search from 2^{64} to 2^{48} trial decryptions. This fact was not mentioned in previous academic publications [13, 14, 12] but is actually documented on the english Wikipedia (as of 2006)[3]. Because DVB-CSA is a non public standard that has been reverse engineered, we do not know whether these checksums are part of the specification or originate from cryptography export restriction.

Since 2^{48} trial decryptions are clearly possible for small corporations and even individuals, DVB-CSA poses more likely a hindrance than a perfect protection of the payload. All TV stations (broadcasted on Astra 19.2) we monitored change the DVB-CSA key every 7 to 10 seconds using a smart card based key distribution system instead of using one (then manually entered) key over a longer

[3] http://en.wikipedia.org/w/index.php?title=Common_Scrambling_Algorithm&diff=41583343&oldid=22087243

period of time. Some TV stations use a constant DVB-CSA key for a longer period, that is manually set at the receiver. This mode is called Basic Interoperable Scrambling System (BISS).

4 Recovering plaintexts

In order to attack DVB-CSA, we began searching for a constant known plaintext in the MPEG-2 video data (H262). Because every bit of the ciphertext depends on every bit of the plaintext, we were looking for a repeating plaintext spawning a full MPEG-2 Transport Stream (TS) cell. Surprisingly, we found out that the video stream of many TV stations contains a lot of cells with a payload completely filled with zero-bytes. We had not expected this as MPEG-2 video uses various compression techniques to reduce the bandwidth, and a video stream which contains a lot of cells filled with zero bytes can be easily compressed.

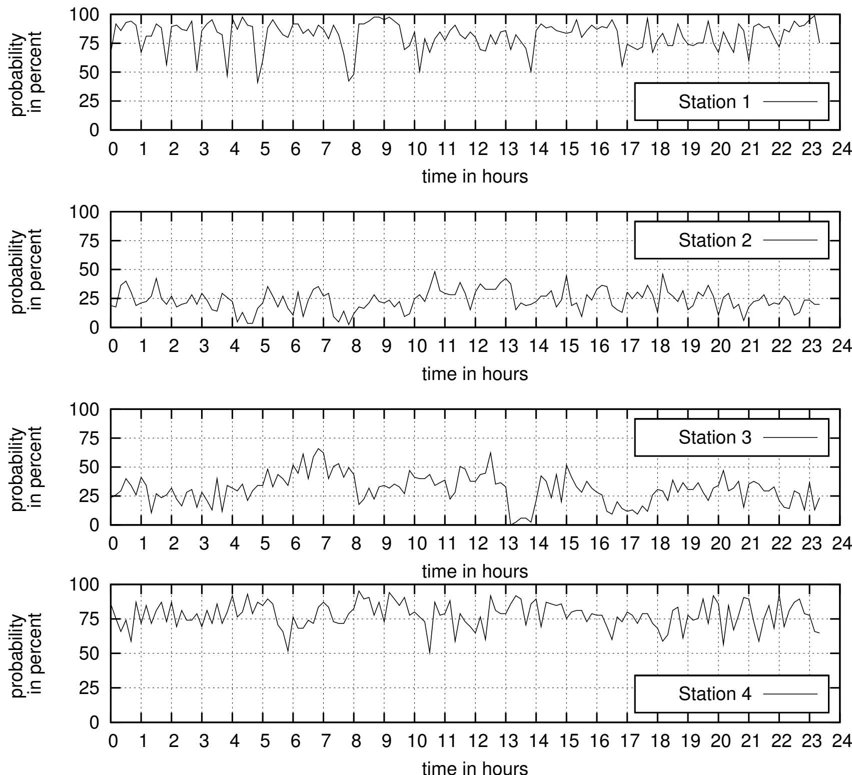
In MPEG-2 video (H262, ISO 13818-2), the video compression codec used for many DVB variants supports so-called *stuffing bytes* – they are used to ensure a minimum bit rate. The ISO 13818-2 standard allows only zero bytes to be inserted between elements of the bitstream [11]. Since DVB-CSA is a completely deterministic encryption scheme: Encryptions of the same plaintext with the same key always result in the same ciphertext. Therefore, if at least two zero-filled frames are broadcasted during the lifetime of one key, these frames result in colliding ciphertexts. Cells completely filled with zero-bytes were the only constant plaintexts that are broadcasted very frequently. We can detect the corresponding encrypted cells by looking for repeating cells (collisions) in the encrypted video stream. We decided to assume that the most frequently colliding cell during a key lifetime corresponds to an encrypted cell filled with zeros.

Unfortunately, the occurrence of zero-filled cells heavily depends on the video content. For example, an interview with a person sitting in a chair without any movement in the background produces many zero-filled stuffing frames. In contrast, when old analog film that contains a lot of scratches and other artifacts is digitized, nearly no cells are filled with zero. We also checked that these cells usually occur when the data rate of the video stream slightly drops.

To make a quantitative statement about the occurrence of zero-filled cells, we measured the relative occurrence of collisions of zero-filled cells on unencrypted stations within intervals of 7 seconds. The detection of a zero-filled cell is considered successful if at least one collision is detected within an interval of seven seconds, and if the most frequently occurring collision actually corresponds to an all-zero cell. The evaluation is done with a granularity of 10 minutes on four popular and unencrypted TV stations airing on the *Astra 19.2* satellite for 24 hours.

This approach of analyzing unencrypted stations is equivalent to measuring how often ciphertext collisions can be found on an encrypted stream for which the key changes every 7 seconds, and evaluating how often these collisions actually decrypt to all-zero cells.

Fig. 3. Plaintext recovery success probability



The results of this experiment can be found in Figure 3. We also found a station on *Astra 19.2* sending so many zero-filled cells that the recovery rate was 100% in our analysis interval.

This rate of intervals in which a zero cell appears most often is an *upper bound* to the success rate of any kind of attack, that is based on this plaintext recovery heuristic.

5 A time memory tradeoff

Since DVB-CSA keys are changed frequently in most use cases, an attacker is interested in recovering DVB-CSA keys very fast. To break DVB-CSA, we use a time memory trade-off as described by Oechslin [10] to recover keys within seconds from a single known plain-text/cipher-text pair. Oechslin invented a general method known as *Rainbow Tables* to invert one way functions faster than by exhaustive search, but precomputations need to be made.

As a one-way function upon which a rainbow table can be built, we propose a mapping f that takes an 48 bit key (without the two checksum bytes) as input and returns the first 6 ciphertext bytes of an all zero cell encrypted with this key. Note that in order to compute f, only 23 calls of the block cipher are required (as described in Section 2, one cell consists of 23 blocks of 8 bytes and all of them need to be processed). As reduction function R_i one could simply XOR the input of f with i.

If we can find the first 6 bytes of a ciphertext c that corresponds to a zero-filled plaintext cell, then $f^{-1}(c)$ is a key (without checksum bytes) that encrypts a zero-filled cell to that specific ciphertext. With some luck it is also the current decryption key (multiple keys could exist encrypting a zero-filled cell to these first 6 bytes of the specific ciphertext).

5.1 Construction

To generate such a rainbow table, we need to generate many chains of length t of the form:

$$k_0 \xrightarrow{R_0 \circ f} k_1 \xrightarrow{R_1 \circ f} k_2 \cdots k_{t-3} \xrightarrow{R_{t-3} \circ f} k_{t-2} \xrightarrow{R_{t-2} \circ f} k_{t-1}$$

where $f : k \rightarrow csablock_k^{23}(0)$. R_i reduces the 64 bit output of *csablock* to a new 48 bit input for f and also is different for each i:

$$R_i(x) := x \oplus (i || table_{id})$$

where $table_{id}$ is an 16 bit unsigned and i is an 32 bit unsigned integer in little-endian[4] representation and || denotes concatenation.

For these chains we only store the head k_0 and the tail k_{t-1} for each chain. The table is sorted by the tail of the chains to make fast lookups possible.

5.2 Coverage and Costs

The time for the generation of the table as well as the storage and the lookup time and success probability for the attack is controlled by four parameters:

t	length of the rainbow chains
m	number of chains in a table
N	total number of distinct keys (2^{48} for DVB-CSA)
T	number of tables

The coverage of a single table is given by [10]:

$$R(t, m, N) := \prod_{i=1}^{t} \left(1 - \frac{m_i}{N}\right)$$

[4] 1000_{10} in little-endian is $e8$ 03

with $m_1 = m$ and

$$m_{i+1} = N(1 - e^{-\frac{m_i}{N}})$$

For T equivalent tables with different reduction functions we get:

$$C(t, m, N, T) := 1 - (1 - R(t, m, N))^T$$

as coverage of all tables combined.

The computational costs for a look-up in these tables is given by:

$$\sum_{i=1}^{t} T * i = \frac{T * t * (t + 1)}{2}$$

and the number of accesses to external memory (i.e. seeks on a HDD) is $T * t$.

To compute reasonable values for these parameters, we need an (efficient) implementation of f first, so that we know how many calls to f can be made during the attack.

6 Implementation

For our attack, we need to evaluate many instances of the one-way function. It is therefore desirable to either evaluate the one-way function in a very fast manner or to compute multiple instances of the function at once.

For the evaluation of our one-way function we need to do 1288 invocations (56 rounds × 23) of the DVB-CSA block cipher round function. As mentioned before, the round function consists of 8 bit XORs and two 8 bit mappings S and P. On most modern computer architectures, computations are carried out on larger bit vectors (mostly 32 or 64 bit). Also, sometimes there are vector instruction sets that allow to use even wider vectors, e.g. 128 or 256 bit.

We use a bitsliced implementation of the DVB-CSA block cipher encryption (except for the S-Box lookup). We do not use a lookup table for the permutation used in the key schedule but an bitsliced implementation of it. Many parts of our implementation are inspired by *libdvbcsa*[5], an open-source implementation of the DVB-CSA cipher. However, the current version of *libdvbcsa* only supports parallel encryption with the same key, but not with many different keys, as required for our precomputations. Also, our key schedule has a lower memory footprint, making it more suitable for GPUs.

6.1 SSE

SSE2[6] allows to do operations on 128 bit registers and thereby allows us to do e.g. 16 8 bit xor operations at the same time. Of course SSE2 has no implementation

[5] http://www.videolan.org/developers/libdvbcsa.html

[6] Streaming SIMD Extensions 2

of either S nor P so they have to be implemented in some other way. Since P merely permutes the bit indices of a byte there is a very simple way to compute many instances of it at once on larger bit vectors. The developers of libdvbcsa[5] already implemented this in a very efficient way. Now for S it is a lot harder to find an algorithmic description of the function that is faster then an lookup table in memory. We decided to expand the 8 bit permutation S to an 16 bit permutation

$$S' : x||y \to S(x)||S(y)$$

and store precomputed values for it in a lookup table. This way we only need to do 8 instead of 16 lookups for 16 parallel computations of the block round function. The 16 bit lookup table does not necessarily fit into the L1 cache of a standard processor but will most likely fit into the L2 cache.

6.2 OpenCL

Modern Graphic Accelerators (GPU) are made up of many (hundreds) independent computing units that can be used to run the same code on different data. These so called compute units provide only limited amounts of fast memory/cache suited for lookup tables. We chose to implement the DVB-CSA block cipher here with a version of the permutation code that was originally written by the developers of *libdvbcsa* and a normal 8 bit lookup table for S. Our GPU implementation operates on 32 bit words and therefor computes 4 instances of the function per compute unit at the same time. The source code of our OpenCL kernel can be found in Appendix A.

7 Parameters

These high-speed implementations could be used to generate rainbow tables for various attack scenarios. Assuming that a hard-disk is able to perform 100 random accesses per second, and an adversary can encrypt about 4,000,000 cells on a GTX460 and about 500,000 cells on a single core CPU, we generated 3 parameter sets.

An adversary might be interested in recovering a DVB-CSA transmission in real-time. He needs to recover a single DVB-CSA key in less than 7 seconds. Using a GPU, the precomputations require 6 hard-disks and 6 TB of storage. Such a table can be precomputed on a single graphics card in less than 9 years. Using multiple graphics cards or faster graphics cards reduces the required time. If the key changes only every 10 seconds, the same tables can be stored on just 4 hard drives, without having to recompute them.

Alternatively, an adversary might not be interested in decoding a transmission in real time, or he would like to recover a static key from a station that only changes the key manually. If a key should be recovered within 30 minutes, this can be done with 120GB of precomputations on a graphics card (less than 8 years of precomputations on a single graphics card) or 525GB of precomputations on a CPU (less than 5 years of precomputations on a graphics card).

Some sets of possible rainbow table parameters are based on the desired speed at which keys would be recovered. We aimed at about 90% coverage. One interesting application of this attack would be PayTV with rapidly changing keys. For the purpose of breaking long term keys, a slower recovery rate would suffice.

Table 2. Suggested parameters

Scenario	# Tables	# Chains per table	Chain-length	Coverage	Storage
GPU 7sec per key	2	2^{38}	2000	93.457%	6TB
GPU 30min per key	3	2^{32}	68410	91.953%	120GB
SSE 30min per key	18	$2^{31.542}$	10000	85.722%	525GB

8 Experimental Results

We computed a small rainbow table with chains of 2000 elements and $2^{32.9008}$ chains. The table is round about 100 GB in size and has approx 5.4 % coverage. We created 23419 random keys and searched for their corresponding one-way function outputs in this table we found 2464 preimages of which 1057 were the actual preimage we had been looking for. This corresponds with our expected success proabability.

9 Attack options

There are several variants of this attack. First of all the rainbow table generation can be optimized:

9.1 Rainbow table optimizations

For our largest suggested parameter set (*GPU 7sec per key*), there are 2^{39} chains in total, stored in two tables with 2^{38} chains per table. Accordingly, chains with 2^{38} different inputs need to be computed. We can choose to use a counter or a similar method to generate the head of these chains. Therefor, only 5 instead of 6 bytes are required to encode the head of the chains. This allows us to save 1 byte per chain and reduce the size of the rainbow table without any side effects.

Also, for a table filled with 2^{38} chains that is sorted by the tail of the chains two consecutive chains will differ in 10 bits on average. Using a variable length encoding for the tails here will allow us to store the tails of the chains in only 2 instead of 6 bytes, for most chains. This additionally reduces the size of the table.

9.2 Harddisk seek performance

Our parameter sets have been chosen in a way that they allow the recovery of a key in 7 seconds or less, if it can be recovered using the table. Even if an adversary wants to decode a video stream in near real time, this requirement can be relaxed.

Assuming that a single table with chain length 2000 and 2^{40} chains is used, the total coverage is 96%. If 10 keys should be recovered with this table, at most 20000 seeks need to be performed. However, the probability that more than 7765 seeks need to be performed is below 0.1%. As a result, one can use a much lower number of harddisks, if the computed table has a high coverage. For tables with a small coverage, unsuccessful searches are more common so that the average number of seeks is closer to the maximum number of seeks for a lookup.

10 Countermeasures

For our attack, we exploit two properties: The key space of DVB-CSA is very small (only 2^{48} keys) and there are full MPEG TS cells that repeat often. Several countermeasures against this attack are possible:

Only 2^{48} possible keys is definitely a too small key space for an encryption system that is used to protect sensitive data. If the two checksum bytes of the key would be chosen freely, 2^{64} keys would be possible. That would slow down our attack by a factor of 2^{16} and render it impossible with today's hardware for an attacker with a small budget. However, we do not know if that would cause interoperability problems with receivers and other equipment that check the checksum bytes of the key, or with key distribution systems, that can only generate keys with a correct checksum. Therefore, we cannot recommend this solution until compatibility with existing hardware has been ensured.

Even 2^{64} possible keys are not sufficient to protect highly sensitive data for a long time. If really high security is required, DVB-CSA should be redesigned and extended to at least 2^{128} possible keys. As far as we know, DVB-CSAv3 has been designed with a key space of 128 bits, but the design of DVB-CSAv3 is not open, so that we cannot evaluate the cryptographic strength of this algorithm. To use DVB-CSAv3, all Conditional Access Modules (CAM) need to be updated. If DVB-CSA is implemented in hardware, what we assume, they even need to be replaced. We think that this solution cannot be used on the short term, but is a great long term solution.

As a short-term countermeasure, we suggest a solution that can be deployed by changing only the equipment used at the sender, and receivers do not need to be updated. Our attack is based on the fact that MPEG TS cells filled completely with zero-bytes repeat frequently. The MPEG TS header specifies separately for every cell whether it is encrypted or not, and which key is used. We think that an DVB-CSA encryption device should check all cells to be encrypted for all-zero cells. Such cells should be sent unencryptedly. As a result, the attacker will not get a single zero-filled encrypted cell and will not be able to launch the attack.

Depending on the video codec used, one should also check if there are other common cell plaintexts, and send them in plaintext too.

If all these countermeasures are not possible, there is still another way to prevent the attack, if only a single or a small number of tables have been generated and are publicly available. A sender can generate a random key, and check if that key can be recovered using these tables. If so, it is not used and the procedure is repeated. As a result, all keys used by a sender cannot be recovered using the public tables, but probably with tables that are not available to the public.

11 Conclusion

This paper shows that DVB-CSA can be broken in real time using standard PC hardware, if precomputed tables are available. These precomputations can be performed on a standard PC in years. This makes DVB-CSA useless for any application where real confidentiality is required. DVB-CSA might still be used to protect digital content, where an adversary is not interested in attacks on the system that recover less than 99% of the payload, and *can not be used to produce pirated Smart Cards*. The attack can be prevented with small changes to the DVB-CSA encryption equipment without having to alter the receivers.

We would like to thank everybody contributing to this paper. This especially includes Academica Senica in Taipei, Taiwan, that provided hardware to compute parts of the rainbow table used in this paper.

References

1. DVB Common Scrambling Algorithm - Distribution Agreements. Technical report, ETSI, 06 1996.
2. ETSI Technical Report 289 - Digital Video Broadcasting (DVB); Support for use of scrambling and Conditional Access (CA) within digital broadcasting systems. Technical report, ETSI, 10 1996.
3. ETSI EN 300 421 - Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for 11/12 GHz satellite services. Technical report, ETSI, 08 1997.
4. ETSI EN 300 429 - Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for cable systems. Technical report, ETSI, 04 1998.
5. ETSI EN 300 744 - Digital Video Broadcasting (DVB); Framing structure, channel coding and modulation for digital terrestrial television. Technical report, ETSI, 01 2009.
6. G. J. Kühn and others. System and apparatus for blockwise encryption/decryption of data. Technical report, 08 1998.
7. G. J. Kühn and others. ETSI EN 301 192 - Digital Video Broadcasting (DVB); DVB specification for data broadcasting. Technical report, 04 2008.
8. Won-Ho Kim, Kyung-Jae Chen, and Hyun-Suk Cho. Design and implementation of MPEG-2/DVB scrambler unit and VLSI chip. *Consumer Electronics, IEEE Transactions on*, 43(3):980 –985, aug 1997.
9. W. Li. Security Analysis of DVB Common Scrambling Algorithm. In *Data, Privacy, and E-Commerce, 2007. ISDPE 2007.*, pages 271–273. IEEE, 2007.
10. P. Oechslin. Making a faster cryptanalytic time-memory trade-off. *Advances in Cryptology-CRYPTO 2003*, pages 617–630, 2003.
11. I. Rec. H. 262— iso/iec 13818-2. *Information technology—Generic coding of moving pictures and associated audio information—Video*, 2000.
12. L. Simpson, M. Henricksen, and W.S. Yap. Improved cryptanalysis of the common scrambling algorithm stream cipher. In *Information Security and Privacy*, pages 108–121. Springer, 2009.
13. Ralf-Philipp Weinmann and Kai Wirt. Analysis of the DVB Common Scrambling Algorithm, 2003.
14. K. Wirt. Fault attack on the dvb common scrambling algorithm. *Computational Science and Its Applications–ICCSA 2005*, pages 577–584, 2005.

A OpenCL Kernel

```
 1  #define u8  unsigned char
 2  #define u32 unsigned int
 3  #define uint64_t ulong
 4  #define TID tid
 5  #define CHAINLEN chainlen
 6
 7  __constant uchar sbox[256] =
 8    {
 9      0x3a, 0xea, 0x68, 0xfe, 0x33, 0xe9, 0x88, 0x1a, 0x83, 0xcf, 0xe1, 0x7f, 0xba, 0xe2, 0x38, 0x12,
10      0xe8, 0x27, 0x61, 0x95, 0x0c, 0x36, 0xe5, 0x70, 0xa2, 0x06, 0x82, 0x7c, 0x17, 0xa3, 0x26, 0x49,
11      0xbe, 0x7a, 0x6d, 0x47, 0xc1, 0x51, 0x8f, 0xf3, 0xcc, 0x5b, 0x67, 0xbd, 0xcd, 0x18, 0x08, 0xc9,
12      0xff, 0x69, 0xef, 0x03, 0x4e, 0x48, 0x4a, 0x84, 0x3f, 0xb4, 0x10, 0x04, 0xdc, 0xf5, 0x5c, 0xc6,
13      0x16, 0xab, 0xac, 0x4c, 0xf1, 0x6a, 0x2f, 0x3c, 0x3b, 0xd4, 0xd5, 0x94, 0xd0, 0xc4, 0x63, 0x62,
14      0x71, 0xa1, 0xf9, 0x4f, 0x2e, 0xaa, 0xc5, 0x56, 0xe3, 0x39, 0x93, 0xce, 0x65, 0x64, 0xe4, 0x58,
15      0x6c, 0x19, 0x42, 0x79, 0xdd, 0xee, 0x96, 0xf6, 0x8a, 0xec, 0x1e, 0x85, 0x53, 0x45, 0xde, 0xbb,
16      0x7e, 0x0a, 0x9a, 0x13, 0x2a, 0x9d, 0xc2, 0x5e, 0x5a, 0x1f, 0x32, 0x35, 0x9c, 0xa8, 0x73, 0x30,
17      0x29, 0x3d, 0xe7, 0x92, 0x87, 0x1b, 0x2b, 0x4b, 0xa5, 0x57, 0x97, 0x40, 0x15, 0xe6, 0xbc, 0x0e,
18      0xeb, 0xc3, 0x34, 0x2d, 0xb8, 0x44, 0x25, 0xa4, 0x1c, 0xc7, 0x23, 0xed, 0x90, 0x6e, 0x50, 0x00,
19      0x99, 0x9e, 0x4d, 0xd9, 0xda, 0x8d, 0x6f, 0x5f, 0x3e, 0xd7, 0x21, 0x74, 0x86, 0xdf, 0x6b, 0x05,
20      0x8e, 0x5d, 0x37, 0x11, 0xd2, 0x28, 0x75, 0xd6, 0xa7, 0x77, 0x24, 0xbf, 0xf0, 0xb0, 0x02, 0xb7,
21      0xf8, 0xfc, 0x81, 0x09, 0xb1, 0x01, 0x76, 0x91, 0x7d, 0x0f, 0xc8, 0xa0, 0xf2, 0xcb, 0x78, 0x60,
22      0xd1, 0xf7, 0xe0, 0xb5, 0x98, 0x22, 0xb3, 0x20, 0x1d, 0xa6, 0xdb, 0x7b, 0x59, 0x9f, 0xae, 0x31,
23      0xfb, 0xd3, 0xb6, 0xca, 0x43, 0x72, 0x07, 0xf4, 0xd8, 0x41, 0x14, 0x55, 0x0d, 0x54, 0x8b, 0xb9,
24      0xad, 0x46, 0x0b, 0xaf, 0x80, 0x52, 0x2c, 0xfa, 0x8c, 0x89, 0x66, 0xfd, 0xb2, 0xa9, 0x9b, 0xc0,
25    };
26
27  inline uint64_t rol64(uint64_t word, unsigned int shift)
28  {
29    return (word << shift) | (word >> (64 - shift));
30  }
31
32  uint64_t permute_block(uint64_t k) {
33          uint64_t n = 0;
34          n  = rol64(k & 0x0080000002000000UL, 5);
35          n ^= rol64(k & 0x0000000404000000UL, 6);
36          n ^= rol64(k & 0x0000000080000000UL, 7);
37          n ^= rol64(k & 0x0800000000000820UL, 10);
38          n ^= rol64(k & 0x0000000000020000UL, 12);
39          n ^= rol64(k & 0x1040000000108000UL, 13);
40          n ^= rol64(k & 0x0000008000200000UL, 14);
41          n ^= rol64(k & 0x0200002000000080UL, 15);
42          n ^= rol64(k & 0x0004000000000000UL, 16);
43          n ^= rol64(k & 0x0000020000000000UL, 18);
44          n ^= rol64(k & 0x0000200000000001UL, 19);
45          n ^= rol64(k & 0x0000000001000000UL, 23);
46          n ^= rol64(k & 0x8000000000000000UL, 25);
47          n ^= rol64(k & 0x0008000000000002UL, 26);
48          n ^= rol64(k & 0x0002000000000400UL, 29);
49          n ^= rol64(k & 0x0000000100000040UL, 30);
50          n ^= rol64(k & 0x0000000000040000UL, 33);
51          n ^= rol64(k & 0x0000000008004000UL, 36);
52          n ^= rol64(k & 0x0000080040000000UL, 38);
53          n ^= rol64(k & 0x0400001000000000UL, 40);
54          n ^= rol64(k & 0x0000000000001000UL, 42);
55          n ^= rol64(k & 0x0000400000000008UL, 43);
56          n ^= rol64(k & 0x2000000000002000UL, 45);
57          n ^= rol64(k & 0x0000000030000000UL, 46);
58          n ^= rol64(k & 0x0000010800000000UL, 47);
59          n ^= rol64(k & 0x0000000000000100UL, 48);
60          n ^= rol64(k & 0x0000100000080000UL, 51);
61          n ^= rol64(k & 0x0000000000000200UL, 52);
62          n ^= rol64(k & 0x0000000000000004UL, 53);
63          n ^= rol64(k & 0x0000000000010000UL, 55);
64          n ^= rol64(k & 0x0110000200800000UL, 57);
65          n ^= rol64(k & 0x4020000000000000UL, 59);
66          n ^= rol64(k & 0x0001800000000000UL, 60);
67          n ^= rol64(k & 0x0000000000000010UL, 61);
68          n ^= rol64(k & 0x0000000000400000UL, 62);
69          n ^= rol64(k & 0x0000044000000000UL, 63);
70          return n;
71  }
72
73  #define I(i) (i*0x0101010101010101UL)
74
```

```
75  //4*6byte in ... 4 * 56 byte out
76  void keyschedule(const __private uchar *in, __private uint *out) {
77          ulong ks[7];
78          uint i,j;
79          for (i = 0; i < 4; i++) {
80                  ks[6] =                 in[3+(i*6)]+in[4+(i*6)]+in[5+(i*6)]; // checksum
81                  ks[6] = (ks[6] << 8) ^ in[5+(i*6)];
82                  ks[6] = (ks[6] << 8) ^ in[4+(i*6)];
83                  ks[6] = (ks[6] << 8) ^ in[3+(i*6)];
84                  ks[6] = (ks[6] << 8) ^ ((in[0+(i*6)]+in[1+(i*6)]+in[2+(i*6)])&0xff); // checksum
85                  ks[6] = (ks[6] << 8) ^ in[2+(i*6)];
86                  ks[6] = (ks[6] << 8) ^ in[1+(i*6)];
87                  ks[6] = (ks[6] << 8) ^ in[0+(i*6)];
88
89                  for (j = 6; j > 0; j--) {
90                          ks[j-1] = permute_block(ks[j]);
91                  }
92                  for (j = 0; j < 7; j++) {
93                          ks[j] ^= I(j);
94                  }
95                  for(j = 0; j < 56; j++) {
96                          out[j] = (out[j]<<8) ^ ((ks[j/8]>>(8*(j%8))) & 0xff);// << (24 - i*8);
97                  }
98          }
99  }
100
101 // csa roundfunction for any implementation of SBOX and P
102 #define RX(w0,w1,w2,w3,w4,w5,w6,w7,t0,t1,k){    \
103         t1 = SBOX(w7 ^ k);                      \
104         t0 = w1;                                \
105         w1 = w0 ^ w2;                           \
106         w2 = w0 ^ w3;                           \
107         w3 = w0 ^ w4;                           \
108         w4 = w5;                                \
109         w5 = w6 ^ P(t1);                        \
110         w6 = w7;                                \
111         w7 = w0 ^ t1;                           \
112         w0 = t0;}                               \
113
114 //lets keep it simple for the first shot
115
116
117 #define BS_BATCH_SIZE 32
118 #define BS_BATCH_BYTES 4
119
120 #define BS_VAL(n)       ((uint)(n))
121 #define BS_VAL32(n)     BS_VAL(0x##n)
122 #define BS_VAL16(n)     BS_VAL32(n##n)
123 #define BS_VAL8(n)      BS_VAL16(n##n)
124
125 #define BS_AND(a, b)    ((a) & (b))
126 #define BS_OR(a, b)     ((a) | (b))
127 #define BS_XOR(a, b)    ((a) ^ (b))
128 #define BS_XOREQ(a, b)  ((a) ^= (b))
129 #define BS_NOT(a)       (~(a))
130
131 #define BS_SHL(a, n)    ((a) << (n))
132 #define BS_SHR(a, n)    ((a) >> (n))
133 #define BS_SHL8(a, n)   ((a) << (8 * (n)))
134 #define BS_SHR8(a, n)   ((a) >> (8 * (n)))
135 #define BS_EXTRACT8(a, n) ((a) >> (8 * (n)))
136
137 #define BS_EMPTY()
138
139 inline uint bsperm(uint sbox_out) {
140         return BS_OR(
141                         BS_OR(
142                              BS_OR (BS_SHL (BS_AND (sbox_out, BS_VAL8(29)), 1),
143                                     BS_SHL (BS_AND (sbox_out, BS_VAL8(02)), 6)),
144                              BS_OR (BS_SHL (BS_AND (sbox_out, BS_VAL8(04)), 3),
145                                     BS_SHR (BS_AND (sbox_out, BS_VAL8(10)), 2))),
146                         BS_OR(      BS_SHR (BS_AND (sbox_out, BS_VAL8(40)), 6),
147                                     BS_SHR (BS_AND (sbox_out, BS_VAL8(80)), 4)));
148 }
149
150
151 #define P(x) bsperm(x)
152 #define SBOX(x) lookup(x,sbox)
```

```
153
154  uint lookup(uint x, __constant uchar *lut) {
155          uint r = 0;
156          r = lut[x&0xff];x>>=8;
157          r ^= lut[x&0xff]<<(8);x>>=8;
158          r ^= lut[x&0xff]<<(16);x>>=8;
159          r ^= lut[x&0xff]<<(24);x>>=8;
160          return r;
161  }
162
163  void encrypt(__private const uint *key, __private uint *i) {
164          uint t1,t2,k;
165          uint w0,w1,w2,w3,w4,w5,w6,w7;
166
167          w0 = i[0];
168          w1 = i[1];
169          w2 = i[2];
170          w3 = i[3];
171          w4 = i[4];
172          w5 = i[5];
173          w6 = i[6];
174          w7 = i[7];
175
176          int r;
177          for (r = 0; r < 56; r++){
178                  k = key[r];
179                  RX(w0,w1,w2,w3,w4,w5,w6,w7,t1,t2,k)
180          }
181
182
183          i[0] = w0;
184          i[1] = w1;
185          i[2] = w2;
186          i[3] = w3;
187          i[4] = w4;
188          i[5] = w5;
189          i[6] = w6;
190          i[7] = w7;
191  }
192
193
194  #define expand(x) {x^=(x<<8);x^=(x<<16);}
195  void r(uint tid, uint idx, __private uint *in) {
196          int i;
197          for (i = 0; i < 4; i++) {
198                  uint x = idx&0xff;
199                  expand(x)
200                  in[i] ^= x;
201                  idx>>=8;
202          }
203          for (i = 0; i < 2; i++) {
204                  uint x = tid&0xff;
205                  expand(x)
206                  in[i+4] ^= x;
207                  tid>>=8;
208          }
209  }
210
211
212  __kernel void csa (__global const u8* input,
213                  __global u8* output,
214                  const int tid,
215                  const int chainlen,
216                  const int num)
217  {
218          const int idx = get_global_id(0);
219          if (idx > num) return;
220          int i;
221          __private u8 k[24];
222          for (i = 0; i < 24; i++) {
223                  k[i] = input[(24*idx)+i];
224          }
225          __private uint key[56];
226          __private uint zero[8];
227          int j,l;
228          for (j = 0; j < CHAINLEN; j++){
229                  keyschedule(k,key);
230                  for(i = 0; i < 8; i++)
```

```
231                     zero[i] = 0;
232             for (i = 0; i < 23; i++){
233                     encrypt(key,zero);
234             }
235             r(TID,j,zero);
236             for (i = 0; i < 4; i++) {
237                     for (l = 0; l < 6; l++)
238                             k[i*6+l] = (zero[l]>>(24-8*i));
239             }
240     }
241     for (i = 0; i < 24; i++)
242             output[24*idx+i] = k[i];
243 }
```