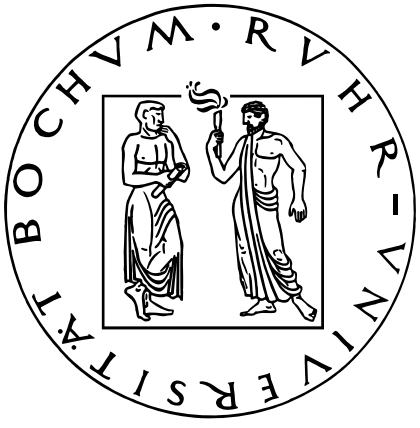
On the Security of Digital Video Broadcast Encryption

Markus Diett

October 26, 2007

Diploma Thesis

Ruhr-Universität Bochum



Chair for Communication Security (COSY)

Prof. Dr.-Ing. Christof Paar

Dipl.-Inf. Andy Rupp

Eidesstattliche Versicherung

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

I hereby certify that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by reference to other authors.

Bochum, Oktober 26, 2007

Markus Diett

Abstract

This thesis consists of two parts. One part is an efficient implementation of the Common Scrambling Algorithm (CSA) using bit slicing. The cipher and its parts are described and optimized to perform fast attacks on the CSA. The second part is a security analysis of the Common Scrambling Algorithm. Several existing attacks on the CSA are presented and evaluated. A brute force attack in software and hardware is considered and its significance regarding the Pay TV scenario is assessed. The resistance against Time-Memory Trade-Off (TMTO) attacks is tested by trying to perform an efficient TMTO attack using the fast bit sliced implementation of CSA.

The results show that the CSA is very resistant against the existing attacks and the TMTO attack in special in the Pay TV scenario. A brute force attack on special hardware requires approximately one day, which is much too long to perform a real time attack. The TMTO attack is not performable, because there seems to be no existing fixed plaintext, which can be used to build the needed tables. Next, we make the assumption that there exists some known plaintext. With this assumption it is possible to mount a TMTO attack. With a table size of several terabyte this attack is performable in about 120 seconds but it is not very comfortable, because the tables have to be either distributed by one table generator or precomputed by each attacker which takes very much time on common hardware.

Acknowledgement

I would like to thank a lot of people who helped me to write this diploma thesis. Thank you Prof. Dr.-Ing. Christof Paar for the chance to write this thesis at the COSY group. Thank you Andy Rupp for supervising me, and all your advises. Thank you Andrey Bogdanov for your stream cipher knowledge. Thank you Stefan Spitz for your Maple scripts and TMTO discussions. Thank you Sven Schäge for your hints on the errors in the CSA papers. Thank you Axel Poschmann for answering my questions concerning the boolean s-box representations. Thank you Martin Novotny for discussing the hardware brute force with me. Thank you to my whole family for supporting me in this time. And at last, thank you to all the others give me feedback and answer my questions in the last six month.

Contents

1. Introduction

The Common Scrambling Algorithm is used by almost all digital TV channels to encrypt their video and audio data. It was originally developed as a proprietary cipher and combines a block and a stream cipher. Since 2002 it is fully revealed and can be analyzed. Due to the fact that CSA is the basic encryption for almost all digital TV data, a successful attack on this cipher stands for a successful attack on nearly all digital TV providers. The existing attacks for viewing pay TV are attacks on the key exchange systems from the different providers. There exists several exchange systems which are frequently updated to block these attacks. The update can be easily performed, because the key exchange algorithm is stored on the key cards which are delivered by the providers and allows software updates. CSA is hard coded and cannot be remotely updated. An exchange of the whole DVB hardware is the only way to change the cipher which would be a large effort for all providers. A successful attack on CSA could revolutionize the pay TV scenery. The attack on CSA is commonly known as 'stream hack'.

This thesis takes a closer look on CSA and its security features. It consists of two parts: First, a fast software implementation of CSA based on bit slicing is developed and described in Chapter 3. It closely shows the different parts of the cipher and how they can be optimized. For efficient attacks on CSA this implementation is used. The second part consists of a security analysis on CSA. Chapter 4 presents some existing attacks. Ralf-Philipp Weinmann and Kai Wirt developed a attack on the stream cipher to determine its state. For this attack a large output stream from the stream cipher is needed, which is not given in the real pay TV environment. The second attack was a fault attack on the block cipher to determine the key. To perform this attack, one needs a box which decrypts ciphertext and allows to introduce errors in several registers. These attacks show a general cryptographic weakness of CSA, but this does not affect the security in a pay TV scenario. In Chapter 5 a brute force attack

on CSA is presented. It starts with the software implementation of this attack and measures its performance. After this, a short view on a brute force attack on special hardware is described. The performance of this attack on the Copacobana is tested. Next, in Chapter 6, the resistance of CSA against TMTO attack is analyzed. The short key length of only 48 bytes leads to the assumption that it could be possible to perform a real time TMTO attack on CSA. It is tested, if this attack is practicable and which conditions have to be accomplished. This thesis ends in the conclusion in Chapter 7.

2. Background

This chapter presents the needed background to understand the attack. It starts with the DVB standard and CSA, gives informations about TMTO, and describe the bit slicing technique to optimize algorithms.

2.1. DVB — Digital Video Broadcast

Digital Video Broadcasting is today's standard of transmitting digital video to the consumer. Three different distribution paths are used in combination with DVB: Satellite (DVB-S), cable (DVB-C), and terrestrial (DVB-T). The major advantage of using digital instead of analog video is that the digital signal can carry multiple channels in one stream and so in less amount of bandwidth compared to multiple analog channels. The following sections describe how to build such a digital stream called transport stream and its for this work relevant features. A full explanation of the DVB standard can be found in [ISO00].

2.1.1. Transport stream

The digital stream consists of several transport stream packets. Each of this packets has a fixed length of 188 Byte. The first four bytes are the transport stream header containing information about the content of the packet. Figure 1 shows the structure of the transport stream header.

- `sync_byte`: This is an 8 bit field containing the fixed value '0100 0111' (0x47). This value is always the same so a stream reader can find the start of a transport stream packet.
- `transport_error_indicator`: This is an one-bit flag. A '1' indicates an uncorrectable bit error in the transport stream packet.

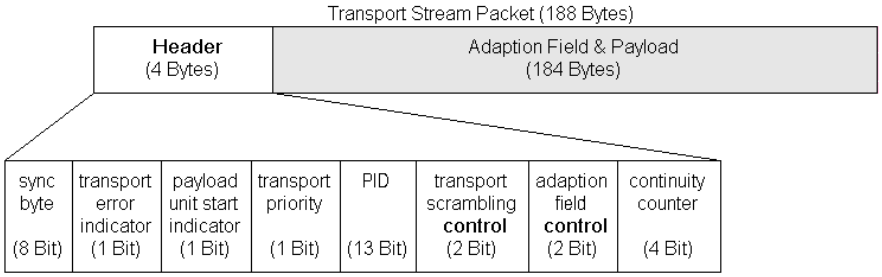


Figure 1.: Transport Stream Overview

- `payload_unit_start_indicator:` This one-bit flag marks the begin of a Primary Elementary Stream packet.
- `transport_priority:` When set to '1' the associated packet is of greater priority than other packets.
- `PID:` The program ID is the most important field to distinguish the several channels in one stream. Some IDs have a special meaning (see Table 1).

Value	**Description**
0x0000	Program Association Table
0x0001	Conditional Access Table
0x0002	Transport Stream Description Table
0x0003-0x000F	Reserved
0x0010-0x1FFE	Available for PES packets, program_map tables, etc
0x1FFF	Null packet

Table 1.: PID Values

- `transport_scrambling_control:` This field indicates the scrambling mode of the payload. The header and the optional adaption field should not be scrambled. If this packet is a null packet, this value should be '00'. The other values are user-defined.
- `adaption_field_control:` This field indicates if this transport stream packet contains an adaption field, payload, or both (see Table 2 for details).
- `continuity_counter:` This field is a counter which increments with each packet of the same ID. After its maximum value the counter starts again at zero.

Value	Description
00	Reserved for future use
01	Payload only
10	Adaption field only
11	Adaption field followed by payload

Table 2.: Adaption Field Control Values

2.1.2. Adaption field

If the adaption_field_control field is '10' or '11' the transport stream header is followed by an adaption field. The adaption field is commonly used for timing purposes and for stuffing bytes to make the data fit into the 184 transport stream data bytes. Figure 2 gives a short overview over the adaption field structure.

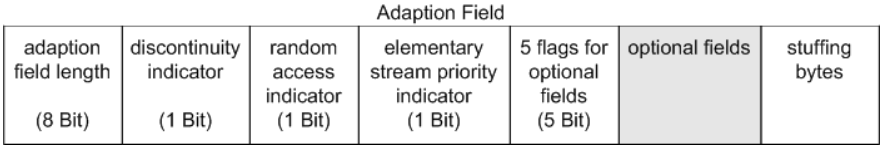


Figure 2.: Adaption Field Overview

- `adaption_field_length`: This field specifies the amount of bytes in the adaption field following the adaption_field_length. If there is only an adaption field with no payload in the transport stream packet the maximum value is 183 (188 bytes transport stream length - 4 bytes transport stream header - 1 byte adaption_field_length). If the adaption field is followed by payload, the maximum and minimum value is 182 and 1 byte, respectively. The value can also be '0'. This is used to put a single stuffing byte prior to the payload. The size of the payload is 184 - adaption_field_length bytes.

The further fields are described in [ISO00] but are not relevant for this work. The relevant fact about adaption fields is their use for stuffing.

2.1.3. Payload: Packetized Elementary Streams

The Packetized Elementary Stream (PES) contains the data to distribute, e.g., video. It consists of the PES header and the data. The PES does not have a fixed

length due to the fact that the data packets (e.g., video) have variable length. The PES is splitted to fit into the transport stream payload (see Figure 3). There are never two different PES packets in one TS packet.

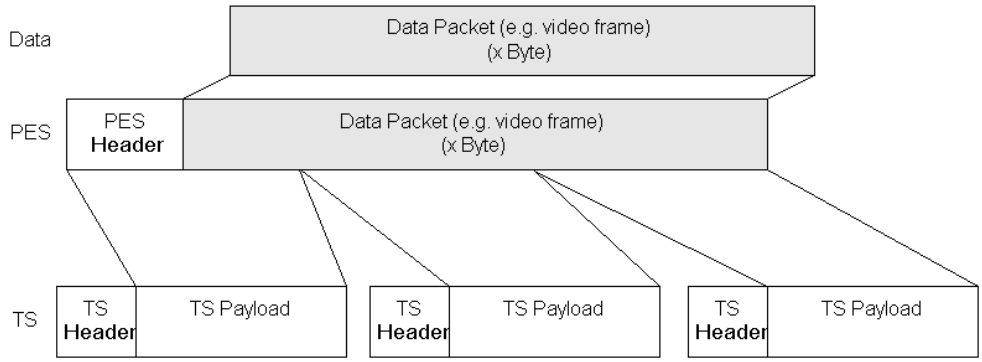


Figure 3.: PES Overview

In most cases, the PES packet does not fit exactly into multiple TS packets (e.g., a PES packet that is of 200 bytes, can be split into 184 bytes and 16 bytes. So the first TS packet contains the 184 bytes but the second packet only 16 bytes). Now, stuffing bytes are needed. There are two solutions for this problem. First, the stuffing bytes can be filled by the adaption field (see Subsection 2.1.2). Second, the packet is filled by zeros. Both options have their advantages and disadvantages. These are explained later on.

The PES packet gets a header with additional informations about the kind of data in its payload (see Figure 4).

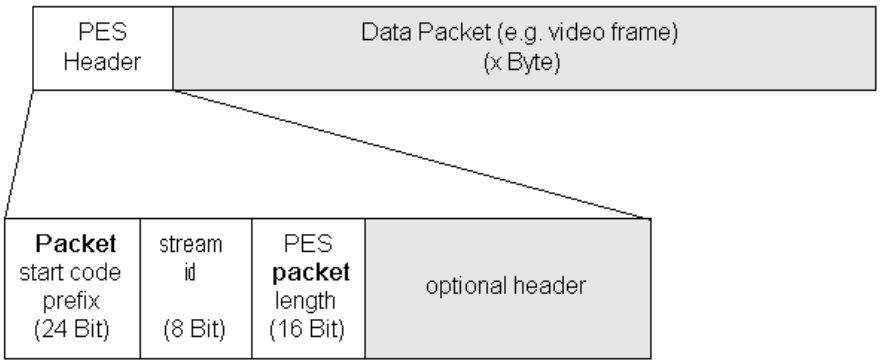


Figure 4.: PES Header Overview

The following fields are always in this header.

- `packet_start_code_prefix:` This is a 24 Bit value, which is always set to 0x000001. It marks the beginning of a PES packet.

- `stream_id:` This field contains information about the kind of data in the PES packet, e.g., video or audio. A full table of possible values can be found in [ISO00].

- `PES_packet_length:` It contains the length of the PES packet after this field in bytes. It can be zero, if the length is not specified. This is only allowed with video data.

After these fields, there are several optional headers related to the different stream_ids.

2.2. CSA — Common Scrambling Algorithm

Due to the fact, that DVB is commonly used by pay TV, there is a need to protect transmitted data against not paying viewers, because everybody can receive the broadcasted data. This is, where CSA is used. It encrypts data streams (e.g., audio or video) in the DVB system. The following sections describe the different parts of this cipher, and a few existing attacks against these parts.

2.2.1. History

CSA was specified by the European Telecommunications Standards Institute (ETSI) and used by the DVB consortium in May 1994. CSA was developed closed source and only the patent papers give some information about the design. But some important details like the S-boxes were kept secret. To avoid reverse engineering, the algorithm should only be implemented in hardware. With this lack of information, a free reimplementation could not be done. Some years later, in 2002, a CSA software implementation was released: FreeDec. This software was disassembled very fast and so the missing informations about CSA could be revealed. Now, it was possible to reimplement the whole cipher.

2.2.2. Overview

CSA cascades a stream cipher with a block cipher. As one can see from Figure 5, when decrypting a 184 bytes data block SB, it is first XORed with the stream cipher output stream. Each SB_i is one 64 bit block. The result (IB) is decrypted with the block cipher in Cipher Block Chaining (CBC) mode and results in the decrypted stream (DB). The initialisation vector (IV) is zero. Each block has to be 64 bit. Due to the use of adaption fields, after splitting the encrypted stream in 64 bit packets, there might be a residue smaller than 64 bit. In Figure 5 this residue is called SR (scrambled residue) and DR (descrambled residue). As one can see, this residue is only decrypted by the stream cipher.

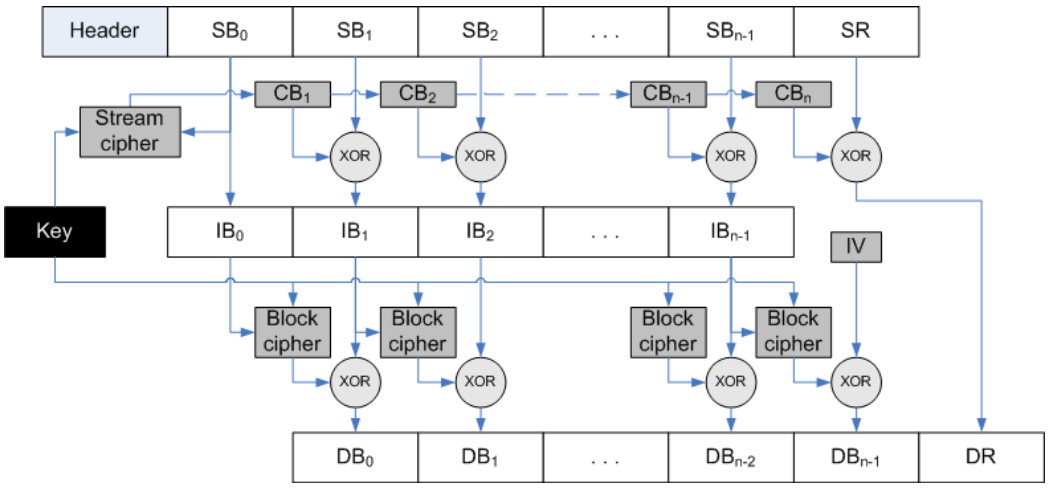


Figure 5.: CSA Decryption Overview

Both, stream cipher and block cipher, use the same key. The stream cipher is additionally seeded with the first scrambled block SB_0. Both ciphers need a 64 bit key, but, due to export restrictions, the effective key length is only 48 bit. The expansion from 48 bit key to 64 bit key is shown in Figure 6. The 48 bit key is splitted into 6 bytes. The first three bytes become the first three bytes of the cipher key. The forth cipher key byte is built by summing up the first three bytes of the given key module 2^8. The second half of the key is built the same way.

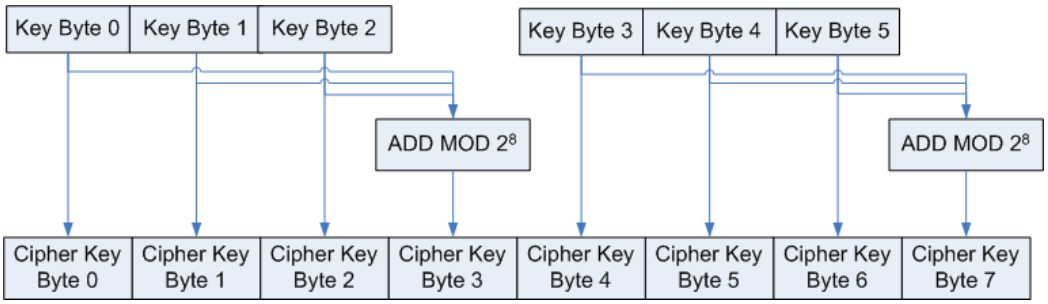


Figure 6.: CSA Key Expansion

2.2.3. Stream Cipher

The generation of the stream cipher output stream has two phases. At first there is a setup phase to initialize the internal state of the cipher. Figure 7 shows this phase. SB_0 is the first scrambled block, which is used to seed the cipher. Register A and B are two shift registers, each consists of 10 columns with 4 bits. X, Y, Z, E, and F are four bit registers. c, p, and q consist of one bit. Summing up all registers, the stream cipher has a state of 103 bits. D is not a real register which is a part of the ciphers state. It just simplify the understanding of the data path in this cipher and is directly built from other state registers.

At first all internal registers are set to zero. The key is loaded into the shift registers A and B. A and B each consist of ten four bit words $(a_0, \cdots, a_9, b_0, \cdots, b_9)$. The first 32 bits of the key are loaded into the first eight four bit words of register A. The second 32 bits of the key are loaded into register B the same way. So the last two words of register A and B are still zero. Now the cipher performs 32 clock cycles $(i = 0, \cdots, 31)$ in the initialisation mode. Additionally the first scrambled data block of the transport stream is used to seed the generator. The following formulas show how the next state S' is computed from the current state S.

At first, the next leftmost elements for the shift registers A and B are computed. I^A and I^B select the high nibble or the low nibble from the current byte from the first scrambled block in relation to the current clock cycle. The current byte means that it is started with byte zero from SB_0 and after four cycles, the next byte is the current byte. So, after 32 cycles all eight bytes are used from SB_0. Very important in generating a'_0 is the XOR with register D. This is the only feedback path from register B. If this were inexistent, a divide and conquer

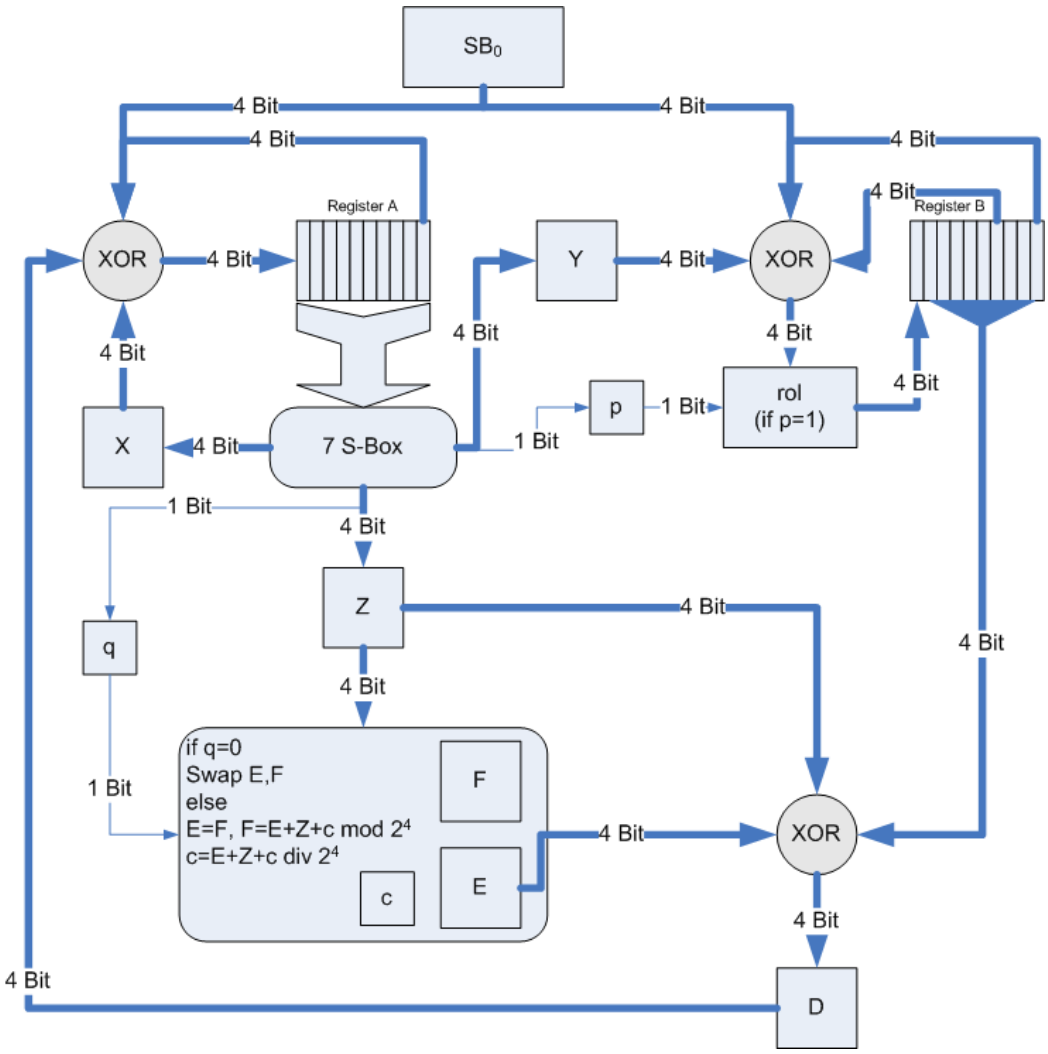


Figure 7.: CSA Stream Cipher Setup Phase

attack on the registers A and B were possible by just brute force the key bits in register A and calculating the resulting state in register B.

$$a'_0 := a_9 \oplus X \oplus D \oplus I^A \tag{1}$$

with

$$I^A := \begin{cases} SB_{0,x} \div 2^4, & \text{if cycle } i \bmod 2 = 0 \\ SB_{0,x} \bmod 2^4, & \text{if cycle } i \bmod 2 = 1 \end{cases} \tag{2}$$

$$b'_0 := b_6 \oplus b_9 \oplus Y \oplus I^B \tag{3}$$

with

$$I^B := \begin{cases} SB_{0,x} \div 2^4, & \text{if cycle } i \bmod 2 = 1 \\ SB_{0,x} \bmod 2^4, & \text{if cycle } i \bmod 2 = 0 \end{cases} \tag{4}$$

The following equations are equal in initialisation and generation mode.

$$A' := (a'_0, a_0, a_1, \cdots, a_8) \tag{5}$$

A' is just generated by right shifting all columns an set the leftmost column to A'_0.

$$B' := \begin{cases} (b'_0, b_0, b_1, \cdots, b_8), & p = 0 \\ (\mathrm{rol}(b'_0), b_0, b_1, \cdots, b_8), & p = 1 \end{cases} \tag{6}$$

B' is a little bit more complex. If p is set to zero, B' is generated like A'. If p is set to one, the generated element B'_0 is rotated left by one bit and then stored as the new leftmost column.

$$(E', F') := \begin{cases} (F, E), & q = 0 \\ (F, E + Z + c \bmod 2^4, & \text{else} \end{cases} \tag{7}$$

$$c' := \begin{cases} c, & q = 0 \\ 0, & q = 1 \wedge E + Z + c < 2^4 \\ 1, & q = 1 \wedge E + Z + c \geq 2^4 \end{cases} \tag{8}$$

E and F are calculated in relation to q. If q equals zero, they are just swapped. If q equals one, E becomes F and F is built by summing up E, Z and c where c

is the carry bit from a previous addition. An optional carry of this summing up is stored in c.

D' is generated by a simple XOR.

$$D' := E \oplus Z \oplus B^{output} \tag{9}$$

where B^{output} is bitwise defined as

$$B_3^{output} := b_{2,0} \oplus b_{5,1} \oplus b_{6,2} \oplus b_{8,3} \tag{10}$$

$$B_2^{output} := b_{5,0} \oplus b_{7,1} \oplus b_{2,3} \oplus b_{3,2} \tag{11}$$

$$B_1^{output} := b_{4,3} \oplus b_{7,2} \oplus b_{3,0} \oplus b_{4,1} \tag{12}$$

$$B_0^{output} := b_{8,2} \oplus b_{5,3} \oplus b_{2,1} \oplus b_{7,0} \tag{13}$$

X, Y, Z, p, and q are defined as output from seven S-boxes with five input and two output bits.

$$X := S_{4,0} \| S_{3,0} \| S_{2,1} \| S_{1,1} \tag{14}$$

$$Y := S_{6,0} \| S_{5,0} \| S_{4,1} \| S_{3,1} \tag{15}$$

$$Z := S_{2,0} \| S_{1,0} \| S_{6,1} \| S_{5,1} \tag{16}$$

$$p := S_{7,1} \tag{17}$$

$$q := S_{7,0} \tag{18}$$

Table 3 shows the used input bits for each box and Table 4 the binary output values.

S_1	$a_{3,0}$	$a_{0,2}$	$a_{5,1}$	$a_{6,3}$	$a_{8,0}$
S_2	$a_{1,1}$	$a_{2,2}$	$a_{5,3}$	$a_{6,0}$	$a_{8,1}$
S_3	$a_{0,3}$	$a_{1,0}$	$a_{4,1}$	$a_{4,3}$	$a_{5,2}$
S_4	$a_{2,3}$	$a_{0,1}$	$a_{1,3}$	$a_{3,2}$	$a_{7,0}$
S_5	$a_{4,2}$	$a_{3,3}$	$a_{5,0}$	$a_{7,1}$	$a_{8,2}$
S_6	$a_{2,1}$	$a_{3,1}$	$a_{4,0}$	$a_{6,2}$	$a_{8,3}$
S_7	$a_{1,2}$	$a_{2,0}$	$a_{6,1}$	$a_{7,2}$	$a_{7,3}$

Table 3.: S-box Input Bits

After 32 clock cycles, the stream cipher enters the generation mode. In this

In	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0x00	10	11	10	11	10	00	00
0x01	00	01	00	01	00	01	11
0x02	01	00	01	10	00	10	10
0x03	01	10	10	11	01	11	10
0x04	10	10	10	00	11	01	11
0x05	11	11	11	10	10	10	00
0x06	11	11	11	01	11	10	00
0x07	00	00	01	10	10	00	01
0x08	11	01	01	01	00	00	11
0x09	10	11	01	10	01	01	00
0x0A	10	10	00	00	11	11	01
0x0B	00	01	11	01	11	00	11
0x0C	01	00	11	11	01	10	01
0x0D	01	00	00	00	00	11	10
0x0E	00	01	10	00	10	01	10
0x0F	11	10	00	11	01	11	01
0x10	00	11	01	01	10	10	01
0x11	11	01	11	00	11	11	00
0x12	11	00	00	11	10	00	11
0x13	00	11	01	01	00	10	11
0x14	10	11	11	10	00	11	00
0x15	10	10	00	11	11	00	01
0x16	01	00	10	00	01	01	01
0x17	01	10	10	11	01	01	10
0x18	10	00	10	00	01	10	10
0x19	10	00	00	11	00	01	11
0x1A	00	01	01	10	11	01	01
0x1B	11	10	10	00	10	10	00
0x1C	01	10	00	01	11	00	10
0x1D	01	01	11	10	01	11	11
0x1E	11	11	11	10	00	11	00
0x1F	00	01	01	01	10	00	10

Table 4.: CSA Stream Cipher S-box

mode, every clock cycle produces two output bits. Figure 8 shows the differences between the initialisation and generation mode.

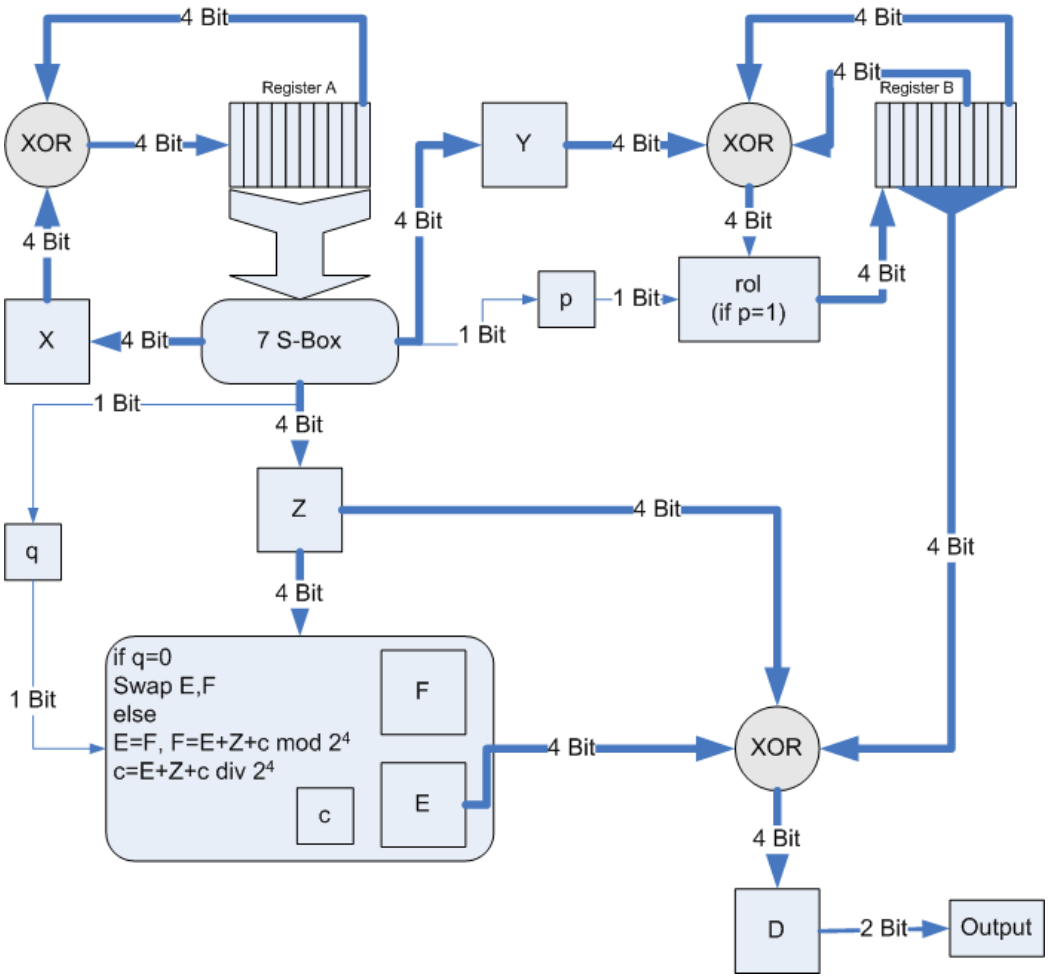


Figure 8.: CSA Stream Cipher Generation Mode

The feedback from D to register A is removed. SB_0 is not used anymore. The following equations are used for building the register input a_0 and b_0.

$$a_0' := a_9 \oplus X \tag{19}$$

$$b_0' := b_6 \oplus b_9 \oplus Y \tag{20}$$

The output is directly computed from D.

$$Output := D_2 \oplus D_3 \| D_0 \oplus D_1 \tag{21}$$

The other registers are computed the same way as in generation mode (Eq. 5 to 18).

2.2.4. Block cipher

The CSA block cipher operates with 64 input bits, output bits, and key bits. It consists of a simple round function which is applied 56 times. The key is expanded to 448 bits and every round uses 8 bits of this expanded key. At first, we take a closer look at the key expansion. The 64 key bits are expanded to 448 bits (56 rounds * 8 bits per round). ρ is a bit permutation as described in Table 5. Every bit i is moved to bit $\rho(i)$.

i	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
$\rho(i)$	0x11	0x23	0x08	0x06	0x29	0x30	0x1C	0x14
i	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
$\rho(i)$	0x1B	0x35	0x3D	0x31	0x12	0x20	0x3A	0x39
i	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
$\rho(i)$	0x17	0x13	0x24	0x26	0x01	0x34	0x1A	0x00
i	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
$\rho(i)$	0x21	0x03	0x0C	0x0D	0x38	0x27	0x19	0x28
i	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
$\rho(i)$	0x32	0x22	0x33	0x0B	0x15	0x2F	0x1D	0x39
i	0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
$\rho(i)$	0x2C	0x1E	0x07	0x18	0x16	0x2E	0x3C	0x10
i	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
$\rho(i)$	0x3B	0x04	0x37	0x2A	0x0A	0x05	0x09	0x2B
i	0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
$\rho(i)$	0x1F	0x3E	0x2D	0x0E	0x02	0x25	0x0F	0x36

Table 5.: Bit Permutation in Key Expansion

$$k^E_{384,\ldots,447} := k_{0,\ldots,63} \tag{22}$$

$$k^E_{64(i-1),\ldots,64(i-1)+63} := \rho(k^E_{64i,\ldots,64i+63}) \text{ with } i := 6 \text{ to } 1 \tag{23}$$

$$k^E_{64i,\ldots,64i+63} := k^E_{64i,\ldots,64i+63} \oplus 0x0i0i0i0i0i0i0i0i \text{ with } i := 0 \text{ to } 6 \quad (24)$$

The round function can be described as a byte wise transformation of the data. The 64 bits of data are splitted into eight 8 byte packets. The encryption round function is defined as the following equations.

$$x := sbox(k \oplus b_7) \quad (25)$$

$$y := perm(x) \quad (26)$$

$$(b'_0, b'_1, b'_2, b'_3, b'_4, b'_5, b'_6, b'_7) := (b_1, b_2 \oplus b_0, b_3 \oplus b_0, b_4 \oplus b_0, b_5, b_6 \oplus y, b_7, b_0 \oplus x) \quad (27)$$

$perm(x)$ is a simple bit permutation. It maps bit 0 to 1, bit 1 to 7, bit 2 to 5, bit 3 to 4, bit 4 to 2, bit 5 to 6, bit 6 to 0 and bit 7 to 3. The S-box is defined in Table 6. Each round uses eight bit of the expanded key. The first round uses bits 0-7, second round bits 8-15, etc. In decryption, the key is used backwards with the following round function:

$$x := sbox(k \oplus b_6) \quad (28)$$

$$y := perm(x) \quad (29)$$

$$(b'_0, b'_1, b'_2, b'_3, b'_4, b'_5, b'_6, b'_7) := (b_7 \oplus x, b_0, b_7 \oplus b_1 \oplus x, b_7 \oplus b_2 \oplus x, b_7 \oplus b_3 \oplus x, b_4, b_5 \oplus y, b_6) \quad (30)$$

After 56 rounds the blocks b_0 - b_7 are the plaintext in decryption and ciphertext in encryption.

2.3. TMTO — Time-Memory Trade-Off

TMTO is a generic attack on block ciphers to determine the used key. It can be varied to perform on a stream cipher's state, but in most cases it is used against block ciphers. Due to the fact that it is a chosen plaintext attack it can only attack block ciphers with no chaining mode. The basic idea is to divide the attack into a precomputation and an online phase. To reduce the complexity of the online phase, several precomputations are made and stored in tables. So, less computations have to be done in the online phase compared to a brute force attack on the same key length. In the following Hellman's original TMTO and a

0x00	0x3A	0xEA	0x68	0xFE	0x33	0xE9	0x88	0x1A
0x08	0x83	0xCF	0xE1	0x7F	0xBA	0xE2	0x38	0x12
0x10	0xE8	0x27	0x61	0x95	0x0C	0x36	0xE5	0x70
0x18	0xA2	0x06	0x82	0x7C	0x17	0xA3	0x26	0x49
0x20	0xBE	0x7A	0x6D	0x47	0xC1	0x51	0x8F	0xF3
0x28	0xCC	0x5B	0x67	0xBD	0xCD	0x18	0x08	0xC9
0x30	0xFF	0x69	0xEF	0x03	0x4E	0x48	0x4A	0x84
0x38	0x3F	0xB4	0x10	0x04	0xDC	0xF5	0x5C	0xC6
0x40	0x16	0xAB	0xAC	0x4C	0xF1	0x6A	0x2F	0x3C
0x48	0x3B	0xD4	0xD5	0x94	0xD0	0xC4	0x63	0x62
0x50	0x71	0xA1	0xF9	0x4F	0x2E	0xAA	0xC5	0x56
0x58	0xE3	0x39	0x93	0xCE	0x65	0x64	0xE4	0x58
0x60	0x6C	0x19	0x42	0x79	0xDD	0xEE	0x96	0xF6
0x68	0x8A	0xEC	0x1E	0x85	0x53	0x45	0xDE	0xBB
0x70	0x7E	0x0A	0x9A	0x13	0x2A	0x9D	0xC2	0x5E
0x78	0x5A	0x1F	0x32	0x35	0x9C	0xA8	0x73	0x30
0x80	0x29	0x3D	0xE7	0x92	0x87	0x1B	0x2B	0x4B
0x88	0xA5	0x57	0x97	0x40	0x15	0xE6	0xBC	0x0E
0x90	0xEB	0xC3	0x34	0x2D	0xB8	0x44	0x25	0xA4
0x98	0x1C	0xC7	0x23	0xED	0x90	0x6E	0x50	0x00
0xA0	0x99	0x9E	0x4D	0xD9	0xDA	0x8D	0x6F	0x5F
0xA8	0x3E	0xD7	0x21	0x74	0x86	0xDF	0x6B	0x05
0xB0	0x8E	0x5D	0x37	0x11	0xD2	0x28	0x75	0xD6
0xB8	0xA7	0x77	0x24	0xBF	0xF0	0xB0	0x02	0xB7
0xC0	0xF8	0xFC	0x81	0x09	0xB1	0x01	0x76	0x91
0xC8	0x7D	0x0F	0xC8	0xA0	0xF2	0xCB	0x78	0x60
0xD0	0xD1	0xF7	0xE0	0xB5	0x98	0x22	0xB3	0x20
0xD8	0x1D	0xA6	0xDB	0x7B	0x59	0x9F	0xAE	0x31
0xE0	0xFB	0xD3	0xB6	0xCA	0x43	0x72	0x07	0xF4
0xE8	0xD8	0x41	0x14	0x55	0x0D	0x54	0x8B	0xB9
0xF0	0xAD	0x46	0x0B	0xAF	0x80	0x52	0x2C	0xFA
0xF8	0x8C	0x89	0x66	0xFD	0xB2	0xA9	0x9B	0xC0

Table 6.: S-Box of CSA's Blockcipher

modification of this method bey Oechslin are described.

2.3.1. Hellman Time-Memory Trade-Off

In 1980 Martin E. Hellman firstly introduced a cryptanalytic time-memory trade-off. Taking a plaintext/ciphertext pair the key can be determined by the following two extremes: First, one can decrypt the ciphertext with all possible keys and compare the result with the given plaintext. This is commonly known as brute force attack. The second way is to encrypt the plaintext with all possible key and stores the resulting ciphertexts with the key in a table ordered by the ciphertext. Now, an attacker can take the ciphertext, look it up in the table and get the corresponding key. For this way, the plaintext has to be fixed. A cryptanalytic TMTO combines these to alternatives. Let us consider DES as example for breaking a cipher with a TMTO attack. DES takes 64 bit blocks of plaintext P and encrypts it to 64 bit blocks of ciphertext C using a 56 bit key K.

$$C = S_K(P) \tag{31}$$

Taking into account that P has to be fixed it is further called P_0. Next, the attacker has to build a function f with the key K as input and a data output with the same size as the key. To reduce the size of the 64 bit ciphertext to 56 bit key length a so called reduction function R is used. This function can be very simple, e.g., just dropping 8 bits of the ciphertext $S_K(P_0)$.

$$f(K) = R(S_K(P_0)) \tag{32}$$

After this, the precomputation phase begins. This phase is used to build the tables. A table consist of different chains. A full chain consists of a starting point and a chain length. A chain is built by taking the starting point and perform the function $f(K)$ on it. The result is used as input for this function again. The chain length defines how often this function has to be performed. The last result is taking as endpoint of the chain. The attacker has to choose m random starting points $SP_1, ..., SP_m$ from the key space. Next to m there is the chain length t the attacker has to choose. How to choose the parameters is discussed below. After choosing m and t, the attacker sets

$$X_{i,0} = SP_i \text{ with } 1 \leq i \leq m \tag{33}$$

and builds the chain by computing

$$X_{i,j} = f(X_{i,j-1} \text{ with } 1 \leq j \leq t. \tag{34}$$

The last element of this chain is the endpoint EP.

$$EP_i = X_{i,t} \text{ with } 1 \leq i \leq m \tag{35}$$

In this way we obtain a table of the following structure:

$$\begin{bmatrix} SP_1 = X_{1,0} & \xrightarrow{f} & X_{1,1} & \xrightarrow{f} & \cdots & \xrightarrow{f} & X_{1,t} = EP_1 \\ SP_2 = X_{2,0} & \xrightarrow{f} & X_{2,1} & \xrightarrow{f} & \cdots & \xrightarrow{f} & X_{2,t} = EP_2 \\ \vdots & & & & & & \\ SP_m = X_{m,0} & \xrightarrow{f} & X_{m,1} & \xrightarrow{f} & \cdots & \xrightarrow{f} & X_{m,t} = EP_m \end{bmatrix} \tag{36}$$

To save memory, just the starting point and the endpoint is stored in a table, ordered by the endpoint. The table needs $2 * keysize * m$ bytes of memory. At this point the precomputation phase is completed.

The online phase starts with receiving a ciphertext C_0 which is the fixed plaintext P_0 encrypted with a key K.

$$C_0 = S_K(P_0) \tag{37}$$

The attacker takes this ciphertext and computes:

$$Y_1 = R(C_0) = R(S_K(P_0)) = f(K) \tag{38}$$

Next, he checks if Y_1 is an endpoint in the table. If it is not, the element prior to the endpoint is not the used key. If Y_1 is equal to an endpoint EP_i, either the element $X_{i,t-1}$ is the used key or it is a false alarm because EP_i has more than one pre image. If $Y_1 = EP_i$ the attacker has to test if $X_{i,t-1}$ is the right key. Because all columns except the first and the last has been discarded in the precomputation phase, the attacker has to build the whole chain until element $X_{i,t-1}$ starting with SP_i. Then he can perform his test on $X_{i,t-1}$. If Y_1 is not an endpoint or not the correct key, the attacker computes

$$Y_2 = f(Y_1) \tag{39}$$

and checks if Y_2 is an endpoint. If not, the key is not the element at $X_{i,t-2}$. If it is an endpoint, he has to compute $X_{i,t-2}$ as described above and checks for the right key. He continues the computation the same way until he get the correct key or reach Y_t. If he does not get the correct value the key is chosen different from all values in the table. The success of this attack is not guaranteed. The success rate P is limited by the parameters m and t. At the first look P should be $\frac{mt}{N}$ with N as the number of all possible keys. But the table generated in the precomputation phase is not merge free. This means that some chains overlap and merge. Thus, having m starting points and a chain length of t does not mean to cover $m * t$ distinct keys. The problem is closely related to the birthday paradox. If $m * t$ becomes closer to N, the percentage of merges in the table becomes larger. Hellman shows in [Hel80] that at $mt^2 = N$ approximately 80 percent of the elements are distinct. So, it is not very effective to just raise m and t more and more. The probability of success is given by:

$$P \geq \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} (1 - \frac{it}{N})^{j+1} \tag{40}$$

A solution is to build l multiple tables with different reduction functions. A merge between chains belonging to different tables is not possible because of the different reduction functions the chain continues with different elements. An attacker has to perform the TMTO for each table. This results in a higher probability of success, which is defined as:

$$P \geq 1 - \left(1 - \frac{1}{N} \sum_{i=1}^{m} \sum_{j=0}^{t-1} (1 - \frac{it}{N})^{j+1}\right)^{l} \tag{41}$$

The attacker has to choose m, t, and l very carefully. He wants a high success rate which can be calculated with the equation above. He has to consider the memory needs which are related to m and t and the time for the online phase in which he has to compute $t * l$ times the function f. After each computation he has to do a table lookup which also costs time. The precomputation time is given by all parameters because the attacker has to compute l tables with m starting

points and each for each chain he has to compute t elements.

2.3.2. Oechslin's Time-Memory Trade-Off

Philippe Oechslin has developed a new table structure which eliminates the disadvantages of a Hellman table [Oec]. In a Hellman table a collision in a single table always results in a merge which drastically reduces the probability of success. Oechslin's idea was to use distinct reduction functions for different columns in one table. So, the chains can collide but do only merge if the collision appears in the same column. If they collide in different columns the chains continue with different reduction functions. The chance that a collision results in a merge is only $\frac{1}{t}$. The probability of success with one table is given by:

$$P = 1 - \prod_{i=1}^{t}(1 - \frac{m_i}{N}) \text{ with } m_1 = m \text{ and } m_{n+1} = N(1 - e^{-\frac{m_n}{N}}) \qquad (42)$$

The rainbow table can directly be compared to classical Hellman tables. The probability of an $mt * t$ rainbow table is approximately equal to t different $m * t$ classical tables. Both cover mt^2 different keys with t different reduction functions. A collision in a set of mt keys, either a single rainbow column or one classical table, results in a merge. Other collisions continue with different functions. The following tables show the relation between the classical tables (43) and the rainbow table (44).

$$
\begin{array}{l}
\text{Table 1} \\
\text{Table 2} \\
\vdots \\
\text{Table t-1} \\
\text{Table t}
\end{array}
\begin{bmatrix}
X^1_{1,1} & \xrightarrow{f_1} & X^1_{1,2} & \xrightarrow{f_1} & \cdots & \xrightarrow{f_1} & X^1_{1,t} \\
X^1_{2,1} & \xrightarrow{f_1} & X^1_{2,2} & \xrightarrow{f_1} & \cdots & \xrightarrow{f_1} & X^1_{2,t} \\
\vdots & & & & & & \\
X^1_{m,1} & \xrightarrow{f_1} & X^1_{m,2} & \xrightarrow{f_1} & \cdots & \xrightarrow{f_1} & X^1_{m,t}
\end{bmatrix}
\begin{bmatrix}
X^2_{1,1} & \xrightarrow{f_2} & X^2_{1,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_2} & X^2_{1,t} \\
X^2_{2,1} & \xrightarrow{f_2} & X^2_{2,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_2} & X^2_{2,t} \\
\vdots & & & & & & \\
X^2_{m,1} & \xrightarrow{f_2} & X^2_{m,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_2} & X^2_{m,t}
\end{bmatrix}
\vdots
\begin{bmatrix}
X^{t-1}_{1,1} & \xrightarrow{f_{t-1}} & X^{t-1}_{1,2} & \xrightarrow{f_{t-1}} & \cdots & \xrightarrow{f_{t-1}} & X^{t-1}_{1,t} \\
X^{t-1}_{2,1} & \xrightarrow{f_{t-1}} & X^{t-1}_{2,2} & \xrightarrow{f_{t-1}} & \cdots & \xrightarrow{f_{t-1}} & X^{t-1}_{2,t} \\
\vdots & & & & & & \\
X^{t-1}_{m,1} & \xrightarrow{f_{t-1}} & X^{t-1}_{m,2} & \xrightarrow{f_{t-1}} & \cdots & \xrightarrow{f_{t-1}} & X^{t-1}_{m,t}
\end{bmatrix}
\begin{bmatrix}
X^t_{1,1} & \xrightarrow{f_t} & X^t_{1,2} & \xrightarrow{f_t} & \cdots & \xrightarrow{f_t} & X^t_{1,t} \\
X^t_{2,1} & \xrightarrow{f_t} & X^t_{2,2} & \xrightarrow{f_t} & \cdots & \xrightarrow{f_t} & X^t_{2,t} \\
\vdots & & & & & & \\
X^t_{m,1} & \xrightarrow{f_t} & X^t_{m,2} & \xrightarrow{f_t} & \cdots & \xrightarrow{f_t} & X^t_{m,t}
\end{bmatrix}
\tag{43}
$$

$$
\begin{bmatrix}
X_{1,1} & \xrightarrow{f_1} & X_{1,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_{t-1}} & X_{1,t} \\
X_{2,1} & \xrightarrow{f_1} & X_{2,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_{t-1}} & X_{2,t} \\
\vdots & & & & & & \\
X_{t,1} & \xrightarrow{f_1} & X_{t,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_{t-1}} & X_{t,t} \\
\vdots & & & & & & \\
X_{2t,1} & \xrightarrow{f_1} & X_{2t,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_{t-1}} & X_{2t,t} \\
\vdots & & & & & & \\
X_{mt,1} & \xrightarrow{f_1} & X_{mt,2} & \xrightarrow{f_2} & \cdots & \xrightarrow{f_{t-1}} & X_{mt,t}
\end{bmatrix}
\tag{44}
$$

The online attack is a little bit different from the classical one. First, the reduction function R_{t-1} is applied on the cipher text. If the resulting key is an

endpoint in the table, the whole chain has to be reconstructed to get the prior element. If there is no endpoint, the attacker assumes that the key for the given ciphertext is at column $t-2$ in the table. He has to perform $R_{t-2}, f_{t-1}(R_{t-1}(C))$ and checks the endpoints. After that, he tries $R_{t-3}, f_{t-2}, f_{t-1}$, and so on. The total numbers of calculations is $\frac{t(t-1)}{2}$ and thus half as much as the t^2 calculations in the classical method with t tables of size $m * t$. As on can see, there are some advantages compared to Hellman's method:

- There are only t table lookups compared to t^2 lookups in the Hellman tables
- Merges of chains result in the same endpoint and are easily detectable
- Due to the unique functions f_i there are no loops

2.4. Optimizing Algorithms Using Bit Slicing

Bit slicing is a programming technique to reduce the runtime of algorithms. It is firstly presented by Eli Biham in[Bih97]. The idea is to take one data block bit by bit and store each bit into one register. For one data block only on bit of these registers is needed, thus, they can be filled with several data blocks to apply the algorithm on all data at the same time. The following text shows with examples how bit slicing works. Let us assume an example computer system which uses 8-bit words. The algorithm consist of a S-box, a key XOR, and a bit permutation, three basic operations in block ciphers.

2.4.1. Preparing Data

Due to the 8-bit registers, one can compute eight words of data at the same time. There are the data words $D_0, \cdots, D_7$. Each data word consists of four bits $D_{n,0}, \cdots, D_{n,3}$. One needs four registers $R_0, \cdots, R_3$ to put every bit of the data word in one registers. After putting the data into the registers, they are filled as shown in Table 7.

2.4.2. S-box

An S-box is more difficult to implement compared to a simple table lookup. Table 8 shows the example s-box. For bit slicing the s-box has to be described

R_0	$D_{0,0}$	$D_{1,0}$	$D_{2,0}$	$D_{3,0}$	$D_{4,0}$	$D_{5,0}$	$D_{6,0}$	$D_{7,0}$
R_1	$D_{0,1}$	$D_{1,1}$	$D_{2,1}$	$D_{3,1}$	$D_{4,1}$	$D_{5,1}$	$D_{6,1}$	$D_{7,1}$
R_2	$D_{0,2}$	$D_{1,2}$	$D_{2,2}$	$D_{3,2}$	$D_{4,2}$	$D_{5,2}$	$D_{6,2}$	$D_{7,2}$
R_3	$D_{0,3}$	$D_{1,3}$	$D_{2,3}$	$D_{3,3}$	$D_{4,3}$	$D_{5,3}$	$D_{6,3}$	$D_{7,3}$

Table 7.: Data Filled in Registers

as boolean equations. Each output bit has his own equation. In this example it is shown, how to build boolean terms in disjunctive normal form from a given S-box. In the following, the equation for output bit 3 (the leftmost bit) is build. First, one takes every input, where bit 3 results in an one. In this example these are the values 0001, 0010, 0100, 0101, 0111, 1010, 1101, 1111. For each of these input values, one can build an AND term which are connected with an OR. This leads to following equation for output bit 3:

$$\begin{aligned} Output_3 := &\overline{x_3}\overline{x_2}\overline{x_1}x_0 \vee \overline{x_3}\overline{x_2}x_1\overline{x_0} \vee \overline{x_3}x_2\overline{x_1}\overline{x_0} \vee \overline{x_3}x_2\overline{x_1}x_0 \\ &\vee\overline{x_3}x_2x_1x_0 \vee x_3\overline{x_2}x_1\overline{x_0} \vee x_3x_2\overline{x_1}x_0 \vee x_3x_2x_1x_0 \end{aligned} \tag{45}$$

In a similar way one has to create an equation for each output bit. These equations can be further optimized using the corresponding algorithms like QuineMcCluskey. In a bit slice implementation each register represents one bit of a data word. So, the registers $R_0, \cdots, R_3$ becomes the input $x_0, \cdots, x_3$. The outputs can be saved back to the registers. In this way, the s-box is applied to all 8 data words at the same time.

	000	001	010	011	100	101	110	111
0	0011	1011	1110	0001	1001	1111	0101	1010
1	0111	0000	1100	0010	0110	1101	0100	1000

Table 8.: Binary S-box, bits 2...0 select column, bit 3 select row

2.4.3. XORing the Key and Bit Permutation

First, the key has to be prepared like the data, this means, it has to write bit by bit in the registers. If one wants to use different keys for the data words the key has to be stored in the same column like the corresponding data. After this

preparation, the key registers have to simply XORed with the data registers. As an example, the following eight operations are computed:

$$X_i := D_i \oplus K_i \text{ with } 0 \leq i \leq 7 \tag{46}$$

After preparing the key, the key registers look like Table 9. After this preparation, one have just to solve the equation

$$X_i := R_i \oplus KR_i \text{ with } 0 \leq i \leq 3 \tag{47}$$

and store X_i back to R_i. So every column is XORed with the correct key.

KR_0	$K_{0,0}$	$K_{1,0}$	$K_{2,0}$	$K_{3,0}$	$K_{4,0}$	$K_{5,0}$	$K_{6,0}$	$K_{7,0}$
KR_1	$K_{0,1}$	$K_{1,1}$	$K_{2,1}$	$K_{3,1}$	$K_{4,1}$	$K_{5,1}$	$K_{6,1}$	$K_{7,1}$
KR_2	$K_{0,2}$	$K_{1,2}$	$K_{2,2}$	$K_{3,2}$	$K_{4,2}$	$K_{5,2}$	$K_{6,2}$	$K_{7,2}$
KR_3	$K_{0,3}$	$K_{1,3}$	$K_{2,3}$	$K_{3,3}$	$K_{4,3}$	$K_{5,3}$	$K_{6,3}$	$K_{7,3}$

Table 9.: Key Filled in Registers

Bit permutation is a very simple task in a bit slice implementation. Due to all registers represent one bit in a data word, a bit permutation simply permutates the registers. It is much faster to store a complete register to another than taking single bits and put in to a different bit position in a data word.

3. Efficient Software Implementation of CSA

To mount a TMTO attack on CSA, the encryption algorithm is needed and has to be optimized as fast as possible. Figure 9 shows the data path after inversing the decryption path.

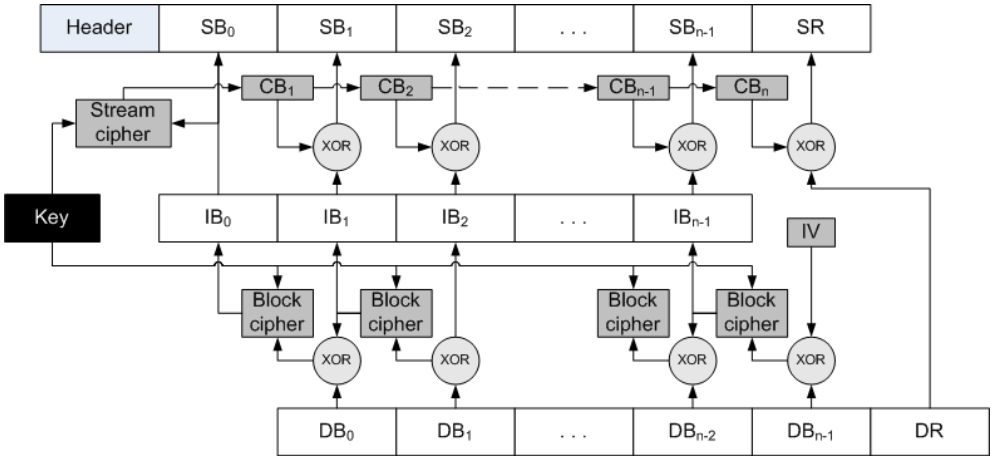


Figure 9.: CSA Encryption Overview

For the TMTO attack there is no need to use the whole algorithm. We have to encrypt a packet as fast as possible. One can ignore the stream cipher and take the CSA as a kind of keyed hash. The plaintext is encrypted in cipher block chaining mode from the end to the beginning. The result from the last encryption is the needed result. One can see the resulting data path in Figure 10. Only the needed parts of the cipher are shown. One encrypts the plaintext with the common key and the block cipher to a 64 bit block. This function is no longer reversible, but we can compare the resulting ciphertext with the given one. This is a mapping with a maximum of 184 bytes toward 8 bytes with 6 key bytes.

$$\{0,1\}^{48} \times \{0,1\}^{1472} \rightarrow \{0,1\}^{64} \tag{48}$$

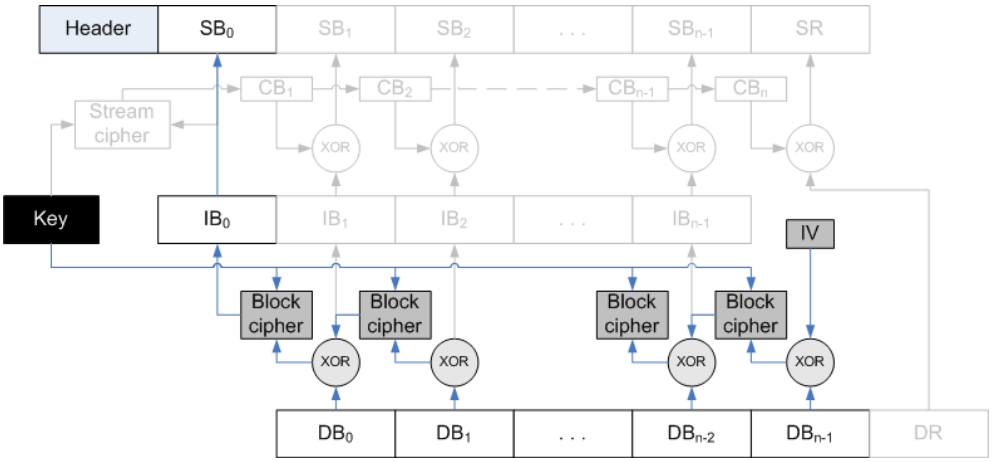


Figure 10.: CSA Encryption Overview for TMTO Attack

3.1. Block Cipher

For a fast TMTO attack, only the block cipher has to be optimized. The following sections describe the different parts of the cipher and our bit sliced implementation.

To compare the efficiency of the implementation, there has to be a reference. The C-code 'Linux DeCSA Tool' from [csa] is used for that. This implementation is only decryption but due to the fact that most of the basic operations (e.g., key schedule, s-box call) are the same in encryption this two implementations can be compared. The execution time values are the time for one execution of the specified function. Keep in mind that the reference implementation operates on one data packet whereas the bit sliced implementation operates on 64 packets in one execution. All these time measurements are done on a AMD Athlon 64 X2 3800+ with 3 GB Ram and Microsoft Windows Vista 64Bit as operating system.

3.1.1. Key Schedule

First, the key has to be expanded from 48 to 64 bit as described in Figure 6. After this the key is stored in the 64 bit slicing registers. Each register is 64 bit

wide, so one can store 64 keys at the same time (refer to Section 2.4). Now the key expansion function is called which expands the 64 bit key to 448 bits or in the bit slice implementation from 64 to 448 registers.

The expansion is done in two steps. First, the 448 expanded key stream registers are filled.

```
1  cw[64]: control word
2  ks=array(448)
3  ks[384..447) = cw[0..63]
4  for (i=6..1)
5    ks[(i-i)*64..(i-i)*64+63] = ks[perm(i*64..i*64+63)]
6  end
```

Listing 3.1: Key expansion step one

The stream is created from the end to the beginning. The last registers of ks are filled with the common key. Each 64 register block $(i-1)$ is created from block i. It is not a simple copy function because the permutation function is done at this point. E.g., referring to Table 5, one can see that bit 23 becomes bit 0 and so on.

```
1  for (i=0..6)
2    for (j=0..7)
3      if (i==1||i==3||i==5) ks[(i*64)+(j*8)+7]^=0xFFFFFFFFFFFFFFFF
4      if (i==2||i==3||i==6) ks[(i*64)+(j*8)+6]^=0xFFFFFFFFFFFFFFFF
5      if (i==4||i==5||i==6) ks[(i*64)+(j*8)+5]^=0xFFFFFFFFFFFFFFFF
6    end
7  end
```

Listing 3.2: XOR to finish the key stream generation

At last, the XOR function has to be applied. Variable i selects each block in the stream and identify the bits, which have to be XORed. The original XOR parameter was $0x0i0i0i0i0i0i0i0i$. This parameter has to be bit sliced to. For $i = 1$ bit 7 of each byte have to be XORed. This is performed by XOR $0xFFFFFFFFFFFFFFFF$ to register 7 of each eight register block, of course only for the registers 64 to 127 (block 1). Variable j selects each eight register block for better code reading.

Table 10 shows, that the bit sliced implementation is more than 40 percent faster in one function call. Because of the simultaneous expansion of 64 keys in one call, the bit sliced implementation is much faster than the reference. This is because of the basic operations in the key expansion: Bit permutation and XORs with fixed values. Both can be done very fast in the bit sliced implementation.

A bit permutation in the reference must be done by selecting the bit (with an AND mask) shifting it to the right position and store it. In bit slicing, simply the whole register is stored to another place (one read and write operation). The XORs in the key expansion are only values from one to six. In the bit sliced implementation a maximum of two registers have to be XORed with $0xFF\cdots$ (64 bits wide). In the key expansion the bit sliced implementation can show its strength.

	Variant: reference	**Variant: bit sliced**
average time	1.342	0.774

Table 10.: Time of 1 Million Key Expansions in Seconds

3.1.2. Bit Sliced S-box

The hardest part of the block cipher conversion was the S-box. The first idea was to represent the s-box as boolean equations as it is usually done for a bit sliced implementation. A disadvantage of the CSA block cipher S-box is the large size of eight bit input and output. This makes the resulting terms very huge. The terms were optimized with an internet tool at `http://www.stephan-brumme.com/programming/Joole/` and the different normal forms are compared. After a large number of time measurement experiments the algebraic normal form seems to be the fastest representation. Thus, this representation is chosen and a second optimizing operation is performed: Frequently occurrences of the same terms were substituted with a temporary variable t. At the end the equations given in A.1 are the result. Although the boolean operations like AND and XOR are drastically reduced in relation to the first conversations of the s-box, this representation is much slower than one simple table lookups. Therefore, the next approach was to take the bit sliced data, reconvert it into not-bit sliced data, do the table lookup, convert it into bit sliced data again and compare the execution time to the boolean representation. Perhaps the execution time of the conversion and the multiple lookups is faster than the boolean representation. For an effective comparison, an efficient way of the conversation and the lookups has to be found.

The S-box has to be performed on 64 blocks of data and consist of a table of 256 byte. Our idea was to do two simultaneous lookups at the same time. Instead

of 8 input and output bits, there were 16 input and output bits. The resulting S-box consists of 65536 integers. Due to the fact that this S-box does two lookups at the same time we called this double S-box. The generation of the double S-box has to be well planned. Here are some possibilities to minimize the complexity of the conversion from bit sliced to normal and reversed. The following equations describe the bitwise input and output of a normal S-box with two different data packets.

$$\begin{aligned}[o_0^1, o_1^1, o_2^1, o_3^1, o_4^1, o_5^1, o_6^1, o_7^1] &= sbox([i_0^1, i_1^1, i_2^1, i_3^1, i_4^1, i_5^1, i_6^1, i_7^1]) \\ [o_0^2, o_1^2, o_2^2, o_3^2, o_4^2, o_5^2, o_6^2, o_7^2] &= sbox([i_0^2, i_1^2, i_2^2, i_3^2, i_4^2, i_5^2, i_6^2, i_7^2])\end{aligned} \tag{49}$$

Taking two data blocks in the bit sliced implementation, they are arranged in the following way:

$$\begin{bmatrix} \cdots & i_0^2 & i_0^1 \\ \cdots & i_1^2 & i_1^1 \\ \cdots & i_2^2 & i_2^1 \\ \cdots & i_3^2 & i_3^1 \\ \cdots & i_4^2 & i_4^1 \\ \cdots & i_5^2 & i_5^1 \\ \cdots & i_6^2 & i_6^1 \\ \cdots & i_7^2 & i_7^1 \end{bmatrix} \tag{50}$$

It is possible to half the size of conversions, if there are not 64 resulting byte words, but 32 two-byte words. One can always take two bit of each register instead of one. After this approach, the input of the double s-box looks like

$$Input_{double\_sbox} := [i_0^2, i_0^1, i_1^2, i_1^1, i_2^2, i_2^1, i_3^2, i_3^1, i_4^2, i_4^1, i_5^2, i_5^1, i_6^2, i_6^1, i_7^2, i_7^1]. \tag{51}$$

The output of the double s-box has to be reconverted into bit slice. The same approaches which are made for the input can be made for the output. This leads to the following double s-box output structure:

$$[o_0^2, o_0^1, o_1^2, o_1^1, o_2^2, o_2^1, o_3^2, o_3^1, o_4^2, o_4^1, o_5^2, o_5^1, o_6^2, o_6^1, o_7^2, o_7^1] = double\_sbox(Input_{double\_sbox}) \tag{52}$$

After defining the input and output of the double s-box, the content has to created. Due to the fact that it consists of 65536 elements, merging two s-boxes by hand is not a reasonable task. We created a small script which builds the double s-box. It is written in PHP4.

The double s-box consists of 0x10000 elements. This loop creates all elements.

```
1   for (i=0..0x9999)
2     ia=i15|i13|i11|i9|i7|i5|i3|i1
3     ib=i14|i12|i10|i8|i6|i4|i2|i0
4
5     oa=sbox[ia]
6     ob=sbox[ib]
7
8     out=oa7|ob7|oa6|ob6|oa5|ob5|oa4|ob4|oa3|ob3|oa2|ob2|oa1|ob1|oa0|ob0
9     output out
10  end
```

Listing 3.3: Creating double sbox

Two inputs for the normal s-box are created, ia and ib. ix means bit x from i counting from right to left. ib consists of bit 0,2,4,6,8,10,12, and 14 of i and ia consists of the rest. Because i stays for the element in the double s-box, it is splitted as described above.

The generated variables ia and ib are used as input to the normal s-box. The output is stored in oa and ob.

The two output variables are merged as described above. The result is printed to screen with a comma separator, so the output of the script can be directly copy&pasted into the C code.

To use the double s-box, the input have to be generated from the data registers. Figure 11 describes the generation of the input values for the first S-box. The other 31 S-box inputs are generated the same way by selecting the rest of the 62 data bits. After this, one has 32 input values for the double S-box which is simply performed as table lookup.

The outputs of the double S-boxes have to be reconvert to the bit sliced data registers. Figure 12 shows, how to move one double S-box output to the data registers. Each output fills two bits of each data register. To fill the 64 bits of the whole data registers, the converting has to be performed 32 times.

But this kind of S-box call has a disadvantage: If one wants to resize the used word size from 64 bit to e.g., 128 bit, the whole function has to be recoded to the new word size. If doubling the word size, the double S-box calls are doubled too.

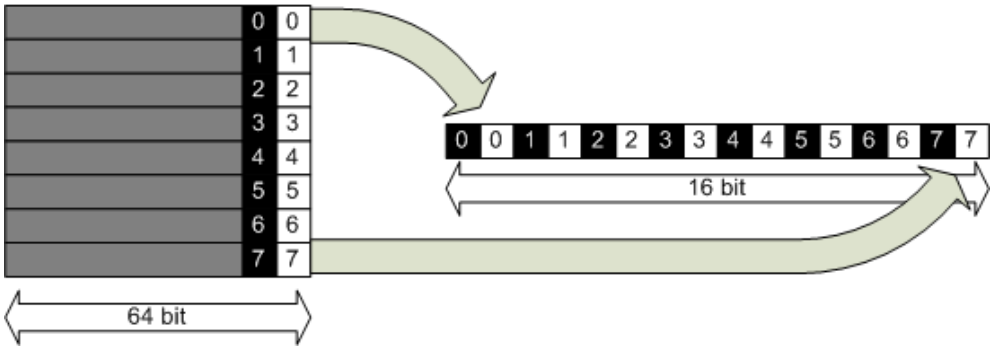


Figure 11.: Creating the Input for one Double S-box

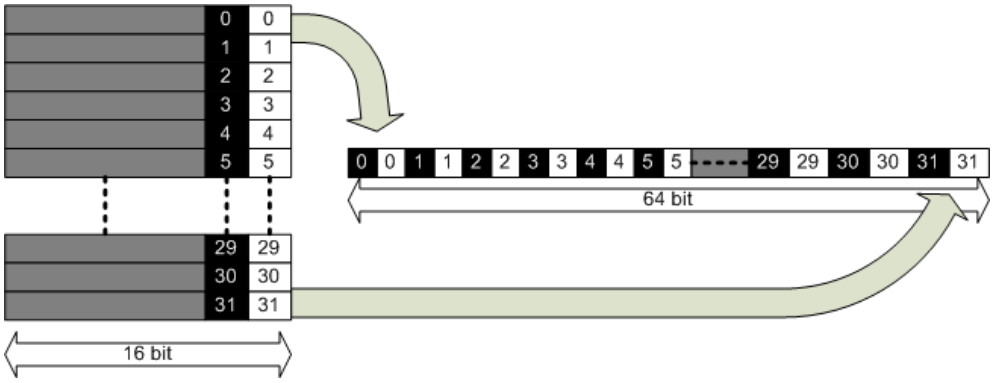


Figure 12.: Creating the Data Registers from the Double S-box Output

Additionally, more bits have to be moved which also takes more time. In the boolean representation, there is only need to change the data types. The next step is to determine the faster variant for a 64 bit word size. This can be easily done by just run the functions with different values and measure the resulting time. Table 11 shows the results of the time measurements. As one can see, the boolean representation is a about 11 percent faster than the double S-box variant. Therefore, the encryption function uses the boolean representation for S-box calls. If this function will be converted to 128 bit word size in future, the boolean representation becomes much better compared to the double S-box variant.

	Variant: double s-box	**Variant: boolean terms**
average time	4.529	4.017

Table 11.: Time of 10 Millions S-box Calls in Seconds

3.1.3. Round Function

The last step of the encryption algorithm is the round function.

```
1  block: data
2  ks: expanded key stream
3  func_sbox: bit sliced s-box function
4  for (i=0..55)
5    tmp[0..7]=block[56..63]^ks[i*8..i*8+7]
6    sbox=func_sbox(tmp)
7    tmp[0..7]=block[0..7]
8    block[0..7]=block[8..15]
9    block[8..15]=block[16..23]^tmp[0..7]
10   block[16..23]=block[24..31]^tmp[0..7]
11   block[24..31]=block[32..39]^tmp[0..7]
12   block[32..39]=block[40..47]
13   block[40..47]=block[48..55]
14   block[40..47]=block[48..55]^sbox[perm(0..7)]
15   block[48..55]=block[56..63]
16   block[56..63]=tmp[0..7]^sbox[0..7]
17 end
```

Listing 3.4: CSA encryption function

The parameters are block which contains the data for encryption and ks which is the expanded key stream. The round function is performed 56 times. Variable tmp is generated as input for the S-box. After the double S-box call, variable

sbox holds the result.Next, the transformations are performed. It just copies and combines the blocks as described in Equation 27 on Page 16. After this round function is finished, the array block holds the encrypted data.

3.1.4. Calling the Encryption

After creating the encryption function, the calling function has to be built.

```
1  cw: control word
2  block: data
3  plain: plaintext derived from transport stream packet
4  ks=key_stream(cw);
5  for (blockc=22..0)
6    block[0..63]=block[0..63]^plain[blockc*64..blockc*64+63];
7    encrypt (block,ks);
8  end
```

Listing 3.5: Calling the encryption function

Variable cw holds the bit sliced control word and is expanded to the key stream ks. This function acts on the assumption that there is a maximum payload length of 23 64-bit blocks. Figure 10 shows the data path. The last block is encrypted, the result is XORed with the previous plain block, this result is XORed again, and so on. The result of the last encryption is the needed result.

3.1.5. Decryption

To compare the TMTO attack with a brute force attack, the decryption is needed too. Like in most block ciphers, only the round function has to be modified and the key stream is used in the other direction.

```
1  block: data
2  ks: expanded key stream
3  func_sbox: bit sliced s-box function
4  for (i=55..0)
5    tmp[0..7]=block[48..55]^ks[i*8..i*8+7]
6    sbox=func_sbox(tmp)
7    tmp[0..7]=block[48..55]
8    block[48..55]=block[40..47]^sbox[perm(0..7)]
9    block[40..47]=block[32..39]
10   block[32..39]=block[24..31]^block[56..63]^sbox[0..7]
11   block[24..31]=block[16..23]^block[56..63]^sbox[0..7]
12   block[16..23]=block[8..15]^block[56..63]^sbox[0..7]
13   block[8..15]=block[0..7]
14   block[0..7]=block[56..63]^sbox[0..7]
15   block[56..63]=tmp[0..7]
```

```
16  end
```

Listing 3.6: CSA decryption function

Like in the encryption, just the state transition given in Equation 30 on Page 16 is applied. After the loop, block contains the decrypted data.

3.1.6. Performance of the Encryption

At the last step, the time of full processing a transport stream packet has to be compared to the reference implementation. Full processing means that at first there is one key expansion and then 23 times the algorithm is performed. Only one key expansion is needed because all blocks in the payload uses the same key. 23 is the maximum block count a transport stream payload packet can consist of. Table 12 shows the results. On the first view, the bit sliced implementation is much slower. But keeping in mind that it processes 64 payload packets at the same time it is about 22 percent faster in a per packet calculation. The largest time sink in the bit sliced implementation is the S-box. If there would exist a way to accelerate the call, the whole function could be accelerated significantly.

All these tests are done on a 64 bit word size. Increasing the word size makes this bit sliced implementation more efficient because the amount of packets which are processed in on call is doubled with no need of more operations.

	Variant: reference	**Variant: bit sliced**
average time	1.240	61.68

Table 12.: Time of 100,000 Full Transport Stream Packets Processions in Seconds

3.2. Stream cipher

For the brute force attack, the stream cipher is needed. It is useful to implement it bit sliced.

```
1  cipher: scrambled block for seeding the state
2  cw: control word
3  stream: output stream
4  outputlen: generated stream length in bytes
5
6  Setting the whole state to 0
```

```
7
8   A[0..31]=cw[0..31]
9   B[0..31]=cw[32..63]
10  init=true
11
12  for (i=0..outputlen+7)
13    if (i==8) init=false
14    if (init)
15      in1=cipher[i*8..i*8+3]
16      in2=cipher[i*8+4..i*8+7]
17    end if
18    for (j=0..3)
19      sbox[0..13]=boolean_sbox_functions[0..13]
20      extra_B[0..3]=output_from_B[0..3]
21      next_A1[0..3]=A[36..39]^X[0..3]
22      if (init) next_A1[0..3] ^= D[0..3] ^ ((j \% 2) ? in2[0..3] : in1[0..3])
23      next_B1[0..3]=B[24..27]^B[36..39]^Y[0..3]
24      if (init) next_B1[0..3] ^= D[0..3] ^ ((j \% 2) ? in1[0..3] : in2[0..3])
25      next_B1[0..3]=conditional_rotate(next_B1[0..3],p)
26      D[0..3]=E[0..3]^Z[0..3]^extra_B[0..3]
27      next_E[0..3]=F[0..3]
28      F[0..3]=conditional_add_or_swap(Z[0..3],E[0..3],q)
29      E[0..3]=next_E[0..3]
30
31      A[0..39]=shift_by_four(A[0..39]);
32      B[0..39]=shift_by_four(B[0..39]);
33
34      A[0..3]=next_A1[0..3]
35      B[0..3]=next_B1[0..3]
36
37      X,Y,Z,p,q <= output_from_sbox
38      if (!init)
39        stream[(i-8)*8+j*2]=D[0]^D[1]
40        stream[(i-8)*8+j*2+1]=D[2]^D[3]
41      end
42    end
43  end
```

Listing 3.7: CSA stream cipher

This implementation is a very simple task and fully described by the pseudo listing above. It simply processes the data as described in Subsection 2.2.3.

Just two points are interesting to examine: conditional_rotate and conditional_add_or_swap.

3.2.1. Conditional Bit Operations in Bit Slicing

In the not bit sliced implementation next_B1 is left shift by one if p equals one. In a bit sliced implementation is this a little bit difficult because there are several data packets at the same time and only some of them have to be rotated.

This means, some columns have to be rotated and the others remains untouched and all this in one bit sliced function. Rotating all packets is quite simple: tmp=next_B1[0]; next_B1[0..2]=next_B1[1..3]; next_B1[3]=tmp. Leaving them untouched is quite simpler. The solution to this is to express the condition inside the assignment without the condition. In the normal implementation 'if (p==1) next_B1=rol(next_B1)' is changed to 'next_B1=(rol(next_B1)&p)—(next_B1&$\tilde{p}$)'. In this case p has to be 0xff instead of 0x01. This assignment can be converted to bit slicing:

```
1 tmp[0]=next_B1[0]&p[0];
2 next_B1[0]=(next_B1[0]& ~p[0])|(next_B1[1]&p[0]);
3 next_B1[1]=(next_B1[1]& ~p[0])|(next_B1[2]&p[0]);
4 next_B1[2]=(next_B1[2]& ~p[0])|(next_B1[3]&p[0]);
5 next_B1[3]=(next_B1[3]& ~p[0])|(tmp[0]&p[0]);
```

Listing 3.8: CSA conditional rotate

The building of the conditional_add_or_swap function is very similar to conditional_rotate. If q equals zero, E and F are simply swapped. If q equals one, E becomes F and F becomes $Z + E + c$ and the new carry is stored in c. Swapping is quite simple, but the addition has to be done bit by bit. To add two 4-bit words the following operations have to be done:

$$\begin{aligned} F_3 &= Z_3 \oplus E_3 \oplus c \\ c &= (Z_3 \wedge E_3) \vee (Z_3 \wedge c) \vee (c \wedge E_3) \\ F_2 &= Z_2 \oplus E_2 \oplus c \\ c &= (Z_2 \wedge E_2) \vee (Z_2 \wedge c) \vee (c \wedge E_2) \\ F_1 &= Z_1 \oplus E_1 \oplus c \\ c &= (Z_1 \wedge E_1) \vee (Z_1 \wedge c) \vee (c \wedge E_1) \\ F_0 &= Z_0 \oplus E_0 \oplus c \\ c &= (Z_0 \wedge E_0) \vee (Z_0 \wedge c) \vee (c \wedge E_0) \end{aligned} \tag{53}$$

Like in conditional_rotate the needed operations are done and put together by an OR and the selector q.

3.2.2. Performance of the Stream Cipher

At last, the performance of the stream cipher has to be tested. Taking a closer look on the operations in the stream cipher, the first thought because of the experiences with the block cipher is that it must be much faster. There are small S-boxes and the other operations are relative fast in bit slicing. The time measurements approve this estimation. Table 13 shows the results for generating one million times an output of 8 bytes. It is interesting that the bit sliced implementation is faster even per function call. It seems that the basic operations are more efficient in bit slicing. Taking into account that the bit sliced variant generates 64 streams at the same time, it is about 68 times faster than the reference.

	Variant: reference	**Variant: bit sliced**
average time	5.463	5.148

Table 13.: Time of One Million 8 Byte Stream Generations

4. Existing Attacks

This chapter gives a short overview about existing attacks on the different parts of the CSA.

4.1. Stream Cipher Analysis

The following attack against the stream cipher was developed by Ralf-Philipp Weinmann and Kai Wirt [RPW04]. With a large given output stream from the cipher it is possible to determine the internal state.

They show that the stream cipher does not generate the maximum of possible states and runs into cycles. Furthermore, they show that there is a small cycle around register A which runs almost always in the same states. From this 'known state' it is possible to compute the rest of the ciphers state. Next, a detailed view on the attack is given.

The state of the CSA stream cipher consists of 103 bits, so its maximum period length is 2^{103}. To avoid a brute force attack, a minimum of 2^{80} states is recommended. In this attack, there are two cycles defined. First, there is the full cycle l_w, which is defined as the smallest number $l_w := j - i$, where the state t_i is equal to t_j. The second cycle is the small cycle l_s. It is defined to be the smallest number $l_s := j - i$, where the state t_i of X and A is equal to these values in state t_j. Using Floyd's cycle-finding algorithm [Flo67], they observed that after a short preperiod there are only a few different cycle length l_w. All these cycles have a length lower than 2^{30} which is much smaller than 2^{80}. The states in cycles with the same length are different, so many different cycles with the same length exists. Another observation was that if two cycles have the same length l_w, l_s is equal too. But in contrast to l_w, A and X go through the same sequence of states, so if l_w is equal, the values in A and X are equal too. Table 14 shows the results of 10^5 experiments to determine how often different cycle lengths occur.

n(l_s) is the number of times a cycle l_s occur after an average preperiod of a(l_s).

n(l_s)	l_s	a(l_s)
36106	22778	152854.6
24196	97494	83098.3
18054	121992	27726.2
15171	42604	65556.8
3244	25802	17643.8
1495	108	21051
131	2391	3138.5

Table 14.: Probability Distribution for Small Cycles [RPW04]

1.6 percent of observed cycle lengths are not in the table because they do not run in the cycle lengths given in the table. Now, one can construct an attack against the stream cipher. First, one has to create a table with all states of the small cycle. After this, every state is tested, if it is the searched one. If this test is positive, one has to reconstruct the remaining registers. But how can one test, if the state is the searched one. The simplest way is to try all possible values for A and X. As shown in Table 14 in most cases there are only a few different cycles. So it is sufficient to try the states in these cycles which are 313,169 (around 2^{19}) tests. This is much less than trying all 2^{44} possibilities for register A and X.

Now A and X can be considered to be known. The registers E, F, and c have to be guessed to. This are 2^9 guesses. The next step is to determine the state of register B for every guess. B is fully determined by A and so it is possible to generate a system of equations describing the two output bits in relation to the state of B. This system can be solved by Gaussian elimination. If there is no solution, E, F, and c might be wrong and have to be altered. If all systems for all guesses have been solved, there might be different solutions. The right state can be determined by simply compare the output of the computed state with the the original output which is assumed to be known.

For this attack, $2^{19} * 2 * 9 = 2^{28}$ systems of equations have to be computed. Each system approximately consists of 60 equations with 40 unknowns (each bit in register B). Their experiments showed, that this attack can be done in less than an hour on a 1.25 GHz PowerPC G4.

4.2. Comments on the Stream Cipher Analysis

This attack cannot be used to break CSA and watch scrambled video. When inspecting the attack, one can see that the state can be determined if the cipher runs in a cycle. These cycles occur after an average preperiod of several thousand bits. A transport stream packet consists of a maximum of 1472 bits payload. In most cases, in this 1472 bits there is no cycle. Without a cycle, the state cannot be determine. A second reason why this attack cannot be used, is the fact that the stream cipher output cannot be determine directly. Even if there is known plaintext, the scrambled data XORed with the stream cipher output are decrypted with the block cipher. So, after performing the stream cipher decryption, the data is still encrypted. The only scrambled packet which is only encrypted with the stream cipher is the residue. But even if this is known plaintext, only a maximum of seven bytes of the stream cipher output are known. This is not enough to perform this attack. This attack shows the general cryptographic weakness of the stream cipher but this weakness not really affect the security of scrambled video data. It just shows that this stream cipher should better not be used on its own to encrypt large amount of data.

4.3. Block Cipher Analysis

There are a few standard methods for the analysis of block ciphers. The first possibility is the classical linear cryptanalysis. But the maximum bias of the S-box is $\frac{17}{128}$, so with the high number of rounds it is very hard to find a linear path through the cipher. It seems that the block cypher offers no weakness against linear cryptanalysis. A second way is to polynomial interpolate the S-boxes. But the resulting polynomials are all of maximum degree, so this representation is not useful for algebraic cryptanalysis.

Kai Wirt [Wir] developed a fault attack on the block cipher which is closer described in the following. Fault attack is kind of attack, where the attack introduces errors in the decryption path and compares the faulted results with the correct ones. Sometimes it is possible to determine several informations about the internal state (e.g., the used key) of the block cipher with these errors. The attacker has to insert a random error in byte 6 of the last round of the decryption. This block becomes the decrypted plaintext byte 7 and so the attacker can build

the following function g(k). b_y^x means byte y of round x.

$$b_6^{55} = b_7^{56} = p_7 \tag{54}$$

$$b_6'^{55} = b_7'^{56} = p_7' \tag{55}$$

$$g(k^E_{0\ldots7}) := sbox(b_6^{55} \oplus k^E_{0\ldots7}) \oplus sbox(b_6'^{55} \oplus k^E_{0\ldots7}) = b_0^{56} \oplus b_0'^{56} = p_0 \oplus p_0' \tag{56}$$

Now the attacker starts a brute force attack against this round key with $g(k') = g(k^E_{0\ldots7})$ for every possible round key k'. It is tested if the key fulfills the equations above. Table 15) shows the number of possible solutions and the respective probability.

Possible number of keys	0	1	2	3	4	5	6	> 6
Probability	0.61	0.00	0.31	0.00	0.07	0.00	0.01	0.00

Table 15.: Probability for the Number of Round Keys for the Attack [Wir]

For every introduced error where the equation can be solved, there are approximately 2 possible keys. If the attacker repeats the attack two or three times with different errors, he can determine the real round key. The attacker now has the round key for the last round and can fully determine the values for each block. He can go one round backwards and perform the same attack to determine this round key. If he has the key for the rounds $i, i+1, \cdots, 56$ he can determine round $i-1$.

$$\begin{aligned} b_6^{i-1} &= b_7^i \\ b_6'^{i-1} &= b_7'^i \\ g(k^E_{8(55-i)\ldots8(55-i)+7}) &:= \\ & sbox(b_6^{55-i} \oplus k^E_{8(55-i)\ldots8(55-i)+7}) \oplus sbox(b_6'^{55-i} \oplus k^E_{8(55-i)\ldots8(55-i)+7}) \\ &= b_0^i \oplus b_0'^i \end{aligned} \tag{57}$$

An attacker has to break 8 rounds to get 8 bytes of the expanded key. For each

round he has to introduce approximately two bit errors, and thus 16 errors have to be introduced in order to break 8 rounds. He has to check all possible values k' for every error, so the whole attack has a complexity of 16 error introductions and 4096 evaluations of g. Another variation of this attack is that the attacker does not repeat the error introduction, but takes all possible keys into account which satisfying the equation $g(k') = g(k^E_{8(55-i)\ldots8(55-i)+7})$. The correct key can be discovered later. The advantage is that only 8 errors need to be introduced. The disadvantage is that the number of evaluations of the g-function increases to approximately $\sum_{i=0}^{7} 2^i * 256 = 65280$ resulting in 256 possible keys.

After this step the attacker has determined the round key $k^E_{0\ldots63}$. The computation of the whole expanded key and the common key is a simple task because of the simplicity of the key scheduling procedure. The Equations 23 and 24 have to be inversed:

$$k^E_{64(i+1),\ldots,64(i+1)+63} := \rho^{-1}(k^E_{64(i),\ldots,64(i)+63} \oplus 0x0i0i0i0i0i0i0i0i) \text{ with } i := 0 \text{ to } 5 \tag{58}$$

ρ^{-1} is the inverse key bit permutation ρ. It is defined in Table 16. The common key CK is the defined by:

$$CK := k^E_{384,\ldots,447} \oplus 0x0606060606060606 \tag{59}$$

4.4. Comments on the Block Cipher Analysis

This procedure given in Section 4.3 shows that an fault attack on the block cipher is possible and due to the simple key expansion a recovery of the common key can be easily accomplished. But on the other hand, this fault attack requires a system which makes it possible to introduce random errors and decrypt the ciphered data. If one does not have the common key already in the box this attack is not performable. Due to this fact, it is not possible to decrypt data just by reading the ciphered stream without having the key somewhere. The advantage is that if the common key is determined the whole CSA is broken, because the stream cipher uses the same key.

i	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
$\rho^{-1}(i)$	0x17	0x14	0x3C	0x19	0x31	0x35	0x03	0x2A
i	0x08	0x09	0x0A	0x0B	0x0C	0x0D	0x0E	0x0F
$\rho^{-1}(i)$	0x02	0x36	0x34	0x23	0x1A	0x1B	0x3B	0x3E
i	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
$\rho^{-1}(i)$	0x2F	0x00	0x0C	0x11	0x07	0x24	0x2C	0x10
i	0x18	0x19	0x1A	0x1B	0x1C	0x1D	0x1E	0x1F
$\rho^{-1}(i)$	0x2B	0x1E	0x16	0x08	0x06	0x26	0x29	0x38
i	0x20	0x21	0x22	0x23	0x24	0x25	0x26	0x27
$\rho^{-1}(i)$	0x0D	0x18	0x21	0x01	0x12	0x3D	0x13	0x1D
i	0x28	0x29	0x2A	0x2B	0x2C	0x2D	0x2E	0x2F
$\rho^{-1}(i)$	0x1F	0x04	0x33	0x37	0x28	0x3A	0x2D	0x25
i	0x30	0x31	0x32	0x33	0x34	0x35	0x36	0x37
$\rho^{-1}(i)$	0x05	0x0B	0x20	0x22	0x15	0x09	0x3F	0x32
i	0x38	0x39	0x3A	0x3B	0x3C	0x3D	0x3E	0x3F
$\rho^{-1}(i)$	0x1C	0x27	0x0E	0x30	0x2E	0x0A	0x39	0x0F

Table 16.: Inverse Key Bit Permutation

5. Brute Force Attack

The short key length of only 48 bit leads to the assumption that a brute force attack could be possible. This chapter shows some approaches on optimizing the brute force attack.

5.1. Setup of the Brute Force Attack

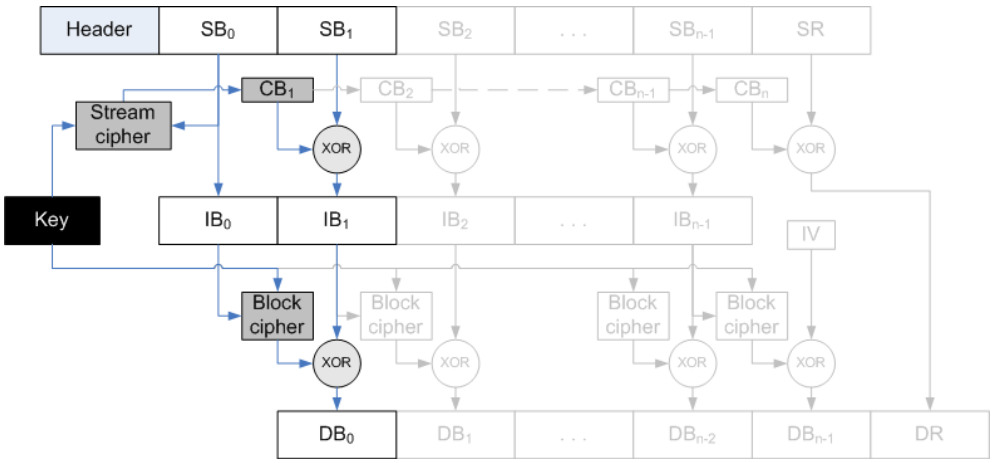


Figure 13.: CSA Decryption Overview for Brute Force Attack

In a brute force attack the attacker tries to decrypt the ciphertext with all possible keys and tests the result for validity. Due to the cipher design, one does not need to decrypt the whole transport stream packet. The data path in decryption is optimizable for this kind of attack. Figure 13 shows this data path for the brute force attack. Both ciphers are needed but only the second cipher block (SB_1) is decrypted with the stream cipher and the first scrambled block ($SB_0 = IB_0$) is decrypted with the block cipher. Thus, one have to generate eight stream cipher output bytes and perform one block cipher encryption. The plaintext (DB_0) is built by XORing the output of the block cipher, the second scrambled block

(SB_1), and the stream cipher output (CB_1). Thus, only the first two scrambled block and the first plaintext block is needed. In an optimum case the first plaintext block has to be known plaintext. But this condition can be weaken a little bit. Referring to Subsection 2.1.3, the first three bytes of an PES packet are always $0x000001$. So, there are three known bytes in every transport stream packet which transports encrypted data and has its payload_unit_start_indicator set to one. The brute force attack can now be done in two steps: First, all 2^{48} keys are tried on a given ciphertext which contains the start of a PES packet. The decrypted result is tested if it starts with $0x000001$. If this condition is fulfilled the key is stored as key candidate. Experiments shows that there are about one million key candidates. The second step is to test the key candidates. Therefor, another transport stream packet is used. It also has to be a PES start packet. The brute force attack is performed a second time but only with the key candidates. In most cases, only one key remains. It is theoretical possible that there is more than one key. If this happens, a third packet has to be decrypted. Until now, it is not shown how to compare the bit sliced result with the given plaintext. One can reconvert the bit sliced data to normal one and compare each data. But this is quite slow. Taking the structure of the fixed plaintext into account, one can create a very fast comparison method. If we have two variables a and b, we can test for equality by XORing these variables. The result has to be zero.

$$a \oplus b = 0, \text{ if } a = b \tag{60}$$

The test for equality with $0x000001$ can be done in the bit sliced implementation very fast by XORing register 23 with $0xFF..$ (64 bit wide). After this, one has to check if the result is zero. For this, all 24 registers have to be ORed. If the result has a zero in one column, this column was equal to the fixed plaintext. It is useful to have a fast check for testing the whole result and not only bit by bit. For this, just negate the result. If the negated result is zero there was no equal plaintext in all 64 columns of the bit sliced registers. If it is not zero, one has to check the result bit by bit and gets the columns with an one in the result. These columns are decrypted with the possible keys.

5.2. Results of the Brute Force Attack in Software

The setup shows that a brute force attack on CSA is possible. The next question is, how fast it can be done. The first step was to measure the execution time of 100,000 calculations. This was about 8.16 seconds. 100,000 calculations with 64 tested keys in one calculation results in approx. $784,000$ key tests per second. Because there are 2^{48} possible keys one can easy calculate the amount of time in which all keys are tested: $2^{48}/784,000 = 359,024,205$ seconds $\approx$ 4155 days. After this one have about 1 million key candidates which have to be tested again, but this can be done in just a few seconds. 4155 days is very long but it is only tested on a common consumer computer.

5.3. Results of the Brute Force Attack in Hardware

How fast can it be accelerated on special hardware? For this, Martin Novotny has done this brute force attack on the Copacobana (`http://www.copacobana.org/`). The brute force attack is tested on two frequencies: 96 MHz and 108 MHz. Each FPGA can test one key in four clock cycles. Due to the amount of 120 FPGAs 120 keys are tested in 4 clock cycles. This leads to the following results:

Frequency	96 MHz	108 MHz
Keys/second	$2.88 * 10^9$	$3.24 * 10^9$
Time for 2^{48} keys	$\approx$ 27.15h	$\approx$ 24.13h

Table 17.: Hardware Brute Force on Copacobana

This shows that the key can be recovered in about one day on special hardware.

The time needed to recover the key is much to long to break CSA in real time. Thus, one can try to reduce the complexity of an attack by a TMTO.

6. TMTO Attack

To launch a TMTO attack on CSA a fixed known plaintext is required. One has to find plaintext, which is repeated often in the transport stream.

6.1. Searching for Known Plaintext

The first task was to get many transport stream packets for analyzing. We used a Dreambox from Dream Multimedia to record several minutes from different channels. The Dreambox's advantage is to save these recording in the transport stream format. Furthermore, it has an ethernet port to copy the data to a PC. Next, we needed a tool for analyzing the streams. I chose Microsoft Visual C++ with QT4 for developing a graphical tool. A graphical tools has some advantages compared to an automatic analysis. It can arrange the results more clearly. One can browse through the several transport stream packets and view the detailed content of the packets. In an automatic analysis it is more difficult to view specified packets. There are already several tools on the web, but they are not very comfortable and it is not possible to browse through the stream and view the full content of one packet. The new tool has the following features: First, it takes a given stream and divides it into the different transport stream packets. These packets are sorted into different lists as shown in Figure 14. Every packet payload is hashed and this hash is stored in the list next to the item. After storing all packets, the tool orders the list by the hash column. If there are some packets with the same payload, these packets are next to each other. It also colors equal packets red for better overview. After processing the stream, one can browse the packets in the different program id lists.

For better analyzing the packets, this tool gives an detail view for each packet as can be seen from Figure 15. It shows all transport stream header fields and, in relation to the packet data, several additional headers. Refer to section 2.1.1

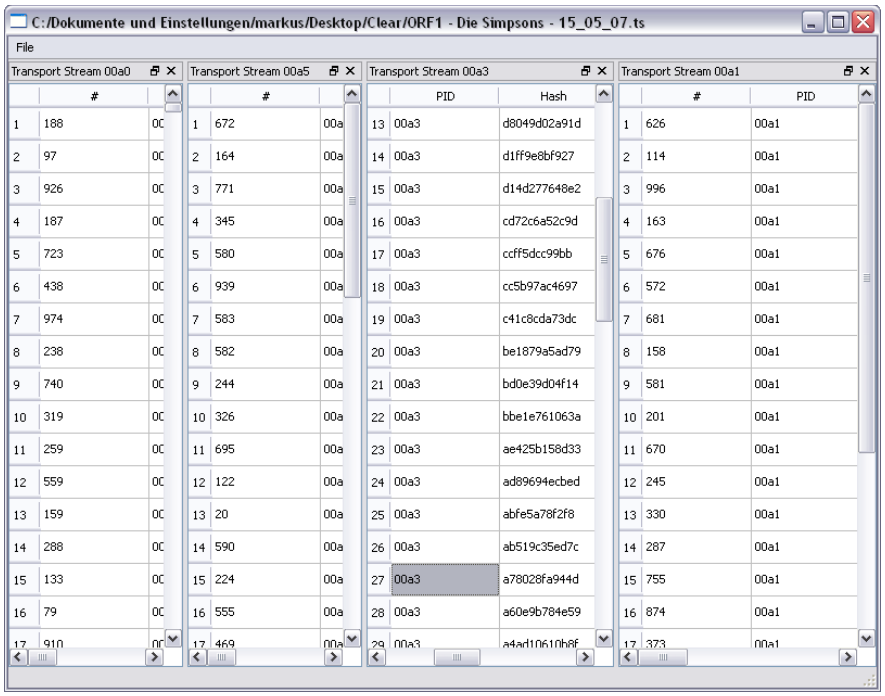


Figure 14.: Transport Stream Analyzer

for the meaning of the different header fields.

6.2. Results of the Search for Known Plaintext

The tool described in the previous section makes it easy to automatically scan a large amount of data. We scan several minutes transport streams of multiple channels to take the differences of channels into account. It was a simple but long task to scan all the streams. We found no transport stream payload which occurs more often even in one channel. Parts of the data payload, like the PES header, can be forecasted but not the whole packet. Next, we searched for known residues by automatically scanning the streams. But the tool does not find any residue which periodically occurs. After this experimental results we analyze, how the provider avoid known plaintexts and residues. It seems that there are different possibilities to create the transport stream packets. The different techniques used by the channels to avoid known residues is a reason why the chance of equal plaintext in different channels is very low. Some channels use the adaption field to stuff bytes into the transport stream. Thus, the end of the PES packet fits to

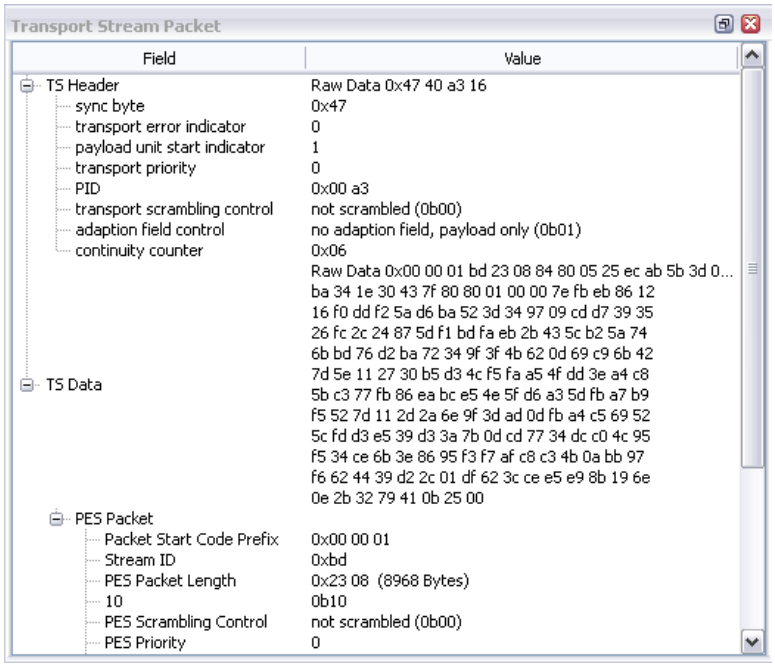


Figure 15.: Transport Stream Analyzer - Detailed Packet View

the end of the last transport stream packet and the residue contains the last bytes of the data. This data cannot be assumed as known. Another technique is to use no adaption fields and just puts zeros to fill the last transport stream packet. This method produces no residues but some known bytes in the payload. Due to the feedback mode of the blockcipher and the additional stream cipher, one needs the whole payload fixed and known. This shows that the same data can be transported in different ways which makes it hard to find plaintext. The zeros at the end of a PES packet gives some partial known plaintext like the header of the PES packet. But there is no complete transport stream packet with fixed plaintext.

The lack of known plaintext makes the TMTO attack impracticable. So the next step of this work is to test the theoretical resistance of CSA against TMTO with the assumption that there is existing fixed plaintext.

6.3. TMTO on CSA

On the first view, TMTO and bit sliced implementations are not very well attachable due to the parallel execution of multiple data in the implementation

and the serial data flow in a TMTO attack. One must find a way to merge these different approaches

The idea is to take the TMTO variant with rainbow tables and check several points at the same time in the online phase. To efficiently implement the attack one needs a reduction function which takes this into account. We take the following reduction function which reduces a 64 bits ciphertext to a 48 bits key:

$$f_i(c) := (c \mod 2^{48}) \oplus m_i, \tag{61}$$

where m_i is a unique 48 bit mask which is applied on the reduced 48 bit cipher. In the TMTO we make the approach that our ciphertext is within the last 64 points in a chain. In the bit sliced implementation we check 64 points at the same time. The first 48 bit ciphered data $D_{0,j}$ is filled into each bit of the registers.

$$\begin{bmatrix} D_{0,0} & D_{0,0} & \cdots & D_{0,0} & D_{0,0} \\ D_{0,1} & D_{0,1} & \cdots & D_{0,1} & D_{0,1} \\ \vdots & & & & \\ D_{0,46} & D_{0,46} & \cdots & D_{0,46} & D_{0,46} \\ D_{0,47} & D_{0,47} & \cdots & D_{0,47} & D_{0,47} \end{bmatrix} \tag{62}$$

Now, we make the assumption that the rightmost bit is the point $i-1$ in a chain of length i, the next bit is point $i-2$, and so on. So, at the rightmost bit, the reduction function f_{i-1} has to be applied, at the next bit f_{i-2}, and so on. We take another registers to store the bit masks.

$$\begin{bmatrix} m_{i-64,0} & m_{i-63,0} & \cdots & m_{i-2,0} & m_{i-1,0} \\ m_{i-64,1} & m_{i-63,1} & \cdots & m_{i-2,1} & m_{i-1,1} \\ \vdots & & & & \\ m_{i-64,46} & m_{i-63,46} & \cdots & m_{i-2,46} & m_{i-1,46} \\ m_{i-64,47} & m_{i-63,47} & \cdots & m_{i-2,47} & m_{i-1,47} \end{bmatrix} \tag{63}$$

By XORing the mask registers with the data registers the correct masks are applied to all data. After the encryption process, the reduction function is simply done by dropping 16 registers. The rightmost bit can be checked if it is in the table. Now, at the bit column where the mask t was applied, the mask t+1 have to be applied. This can be done by simple left shift the mask registers.

$$\begin{bmatrix} m_{i-63,0} & m_{i-62,0} & \cdots & m_{i-1,0} & 0 \\ m_{i-63,1} & m_{i-62,1} & \cdots & m_{i-1,1} & 0 \\ \vdots & & & & \\ m_{i-63,46} & m_{i-62,46} & \cdots & m_{i-10,46} & 0 \\ m_{i-63,47} & m_{i-62,47} & \cdots & m_{i-1,47} & 0 \end{bmatrix} \tag{64}$$

After the encryption and reduction, the next bit column can be checked if it is in the table. This can be done 64 times to check all points.

Now, the last 64 points in the row are checked. The next step is to check the previous 64 points. The corresponding masks have to be stored in the mask registers.

$$\begin{bmatrix} m_{i-128,0} & m_{i-127,0} & \cdots & m_{i-66,0} & m_{i-65,0} \\ m_{i-128,1} & m_{i-127,1} & \cdots & m_{i-66,1} & m_{i-65,1} \\ \vdots & & & & \\ m_{i-128,46} & m_{i-127,46} & \cdots & m_{i-66,46} & m_{i-65,46} \\ m_{i-128,47} & m_{i-127,47} & \cdots & m_{i-66,47} & m_{i-65,47} \end{bmatrix} \tag{65}$$

The only difference to the way described above is that after left shifting the masks the rightmost bits have to be filled with the next mask until the last point in the row is reached.

$$\begin{bmatrix} m_{i-127,0} & m_{i-126,0} & \cdots & m_{i-65,0} & m_{i-64,0} \\ m_{i-127,1} & m_{i-126,1} & \cdots & m_{i-65,1} & m_{i-64,1} \\ \vdots & & & & \\ m_{i-127,46} & m_{i-126,46} & \cdots & m_{i-65,46} & m_{i-64,46} \\ m_{i-127,47} & m_{i-126,47} & \cdots & m_{i-65,47} & m_{i-64,47} \end{bmatrix} \tag{66}$$

This has to be done with all points in the chain or until a point matches an entry in the table.

A small disadvantage of this way is that at the end of a chain, there are not all data registers in use. At the end of a chain the bit columns are ignored one by one. By each check of a 64 point block at the end there are $\sum_{i=1}^{63} i$ unnecessary computations. At the first block where only $\sum_{i=1}^{64} i$ are done this is a very high overhead. But due to the large chain length this overhead becomes very small.

For the creation of the tables only the masks have to be chosen. The only

requirement is that the masks are unique. A simple mask like m_i=i is possible.

6.4. Performance of the TMTO Attack

After having optimized the implementation of CSA and built the setup of the TMTO attack, we determine the TMTO parameters and evaluate the efficiency of such an attack. For this, one has to consider several general conditions:

- Success probability of at least 90 percent. There is no profit if half of the data stream remains encrypted.
- Execution time of the online phase at a maximum of 120 seconds. The standard key lifetime has a maximum of 120 seconds, so if one wants to attack the cipher in real time this is the maximum value. Some channels change the key more often, but this is not taken into account here.
- Tables of reasonable size: If the attack should be practicable on normal PCs the table size cannot be too large. Some gigabyte should be the maximum.

The next parameter to determine is the chain length. The maximum online time is given and the execution time of the algorithm is known from the last section. For a chain length t one needs $\frac{t*(t-1)}{2}$ calculations. With the number of algorithm executions per second x, the time of the online phase without the time of table lookups is given by:

$$onlinesec = \frac{t * (t-1)}{2}/x \tag{67}$$

With a chain length of $2^{12.28}$ the online phase requires 119.4 seconds. Now we make the assumption that the table lookup can be done without slowing the computation, e.g., on a second core in the computer system. The chain length of $2^{12.28}$ gives the number of table lookups which have to be done in these 120 seconds. Thus, the next step is to verify, if this amount of lookups can be done on a consumer computer. The lookup timing tests are done on a ATA133 hard disk with ext3 as file system. Because of the table size, these lookups cannot be done in the RAM. Our approach was to use the file system index structure to find a value faster. For this, one creates a special directory structure. At first, one takes the reduced ciphertext to store as hexadecimal value. This results in 12

hexadecimal digits. The first nine hexadecimal digits are taken as directory, with each digit as one directory. For example, the value $0x26876AG213E2$ becomes the path '/2/6/6/7/6/A/G/2/1'. In this directory, there is a file with a table of the last three hexadecimal digits and the starting points ordered by the three cipher digits. This results in many small files and a search for a cipher text can be done very fast. The first idea was to store the whole ciphertext as path and in each path is a small file with the starting point. But due to the block size in a file system, the file requires a minimum of 1024 bytes. If one only stores 6 bytes in one file there is a overhead of 17,000 percent and the tables become much to large. If there are only a very few starting points, one can take only seven or eight digits as path and the rest for the files. It is important to keep the block size in mind. Our timing experiments show that 100,000 lookups requires approximately 5.5 seconds. So, in 120 seconds about 2^21 lookups can be performed. This is much more than the required value. With the assumption that the lookup and the computation can be done separately the lookups are fast enough.

At a last step, the number of starting points has to be determined. The Maple script given in Appendix A.3 can be taken to try different numbers of starting points to get a success probability of at least 90 percent. Taking 2^{38} points, the result is a probability of 91.5 percent. But the file size of the computed tables are about three terabyte. The precomputation time is also very large, but this can be done on faster machines to shorten this time.

7. Conclusion

This thesis presents CSA and its strength against different kind of attacks in a pay TV scenario. The existing attacks offers some weaknesses of the different cipher parts. It shows that the stream cipher should not be used on its own to encrypt a large amount of data. But in the pay TV setup, it offers no weakness. A brute force attack on CSA requires at least one day on special hardware which is too long to take this as a real time attack. But this attack is performable because there exists some known plaintext. At last, the resistance against TMTO attacks is tested. In the pay TV scenario, this attack cannot be mounted due to the lack of fixed plaintext. Fixed plaintext is an imperative for this attack. Even if known plaintext will be found in the future, the TMTO on CSA is very hard. Our bit sliced implementation is not fast enough to perform a comfortable TMTO attack. It takes a table size of three terabyte to mount this attack and a very long precomputation time. If it is possible to speed up the implementation to $2^{33.7}$ executions per second, a successful attack can be done with a table size of 4.24 gigabytes. But the problem with the lack of fixed plaintext remains.

This analysis shows that there seems to be no weakness of CSA in the pay TV environment today. If the computers becomes fast enough to perform a real time TMTO attack on CSA (standalone or with distributed computing), the providers has to verify that there should be no fixed plaintext.

In this work CSA is implemented on a 64 bit consumer computer. It would be an interesting future task to convert it to an 128 bit system like the Playstation 3. This could be much faster due to the high CPU frequency and the large bus size. Perhaps, it could be possible store the whole needed tables on PS3's hard disk and perform a TMTO attack. If this is possible, only the lack of fixed plaintext would prevent the CSA from being broken.

A. Appendix

A.1. Optimized Boolean Representation of CSA Block Cipher S-box

```
1  // b[8] = inputbits of the s-box
2  // out[8] = outputbits of the s-box
3  // t = temporary variable for substituting
4
5
6  t[0]=b[0]&b[1];
7  t[1]=b[0]&b[2];
8  t[2]=b[0]&b[3];
9  t[3]=b[0]&b[4];
10 t[4]=b[0]&b[5];
11 t[5]=b[0]&b[6];
12 t[6]=b[1]&b[2];
13 t[7]=b[1]&b[3];
14 t[8]=b[1]&b[4];
15 t[9]=b[1]&b[5];
16 t[10]=b[1]&b[6];
17 t[11]=b[2]&b[3];
18 t[12]=b[2]&b[4];
19 t[13]=b[2]&b[5];
20 t[14]=b[2]&b[6];
21 t[15]=b[3]&b[4];
22 t[16]=b[3]&b[5];
23 t[17]=b[3]&b[6];
24 t[18]=b[3]&b[7];
25 t[19]=b[4]&b[5];
26 t[20]=b[4]&b[6];
27 t[21]=b[4]&b[7];
28 t[22]=b[5]&b[6];
29 t[23]=b[5]&b[7];
30 t[24]=b[6]&b[7];
31 t[25]=t[0]&t[11];
32 t[26]=t[0]&t[12];
33 t[27]=t[0]&t[13];
34 t[28]=t[0]&t[14];
35 t[29]=t[0]&t[15];
36 t[30]=t[0]&t[16];
```

```
37 t[31]=t[0]&t[17];
38 t[32]=t[0]&t[19];
39 t[33]=t[0]&t[20];
40 t[34]=t[0]&t[22];
41 t[35]=t[1]&t[15];
42 t[36]=t[1]&t[16];
43 t[37]=t[1]&t[17];
44 t[38]=t[1]&t[19];
45 t[39]=t[1]&t[20];
46 t[40]=t[1]&t[22];
47 t[41]=t[2]&t[19];
48 t[42]=t[2]&t[22];
49 t[43]=t[3]&t[22];
50 t[44]=t[6]&t[15];
51 t[45]=t[6]&t[16];
52 t[46]=t[6]&t[19];
53 t[47]=t[6]&t[20];
54 t[48]=t[6]&t[22];
55 t[49]=t[7]&t[19];
56 t[50]=t[7]&t[20];
57 t[51]=t[7]&t[22];
58 t[52]=t[11]&t[19];
59 t[53]=t[11]&t[20];
60 t[54]=t[12]&t[22];
61 t[55]=t[15]&t[22];
62
63
64 out[0]=(b[2])^(b[3])^(b[4])^(b[7])^(t[0])^(t[4])^(t[5])^(b[0]&b[7])^(t[6])^(t
       [7])^(t[8])^(t[9])^(t[10])^(t[11])^(t[12])^(t[14])^(t[15])^(t[16])^(t[17])^(
       t[21])^(t[22])^(t[0]&b[4])^(t[1]&b[4])^(t[1]&b[5])^(t[1]&b[6])^(t[2]&b[4])^(
       t[2]&b[6])^(t[3]&b[6])^(t[4]&b[6])^(t[4]&b[7])^(t[6]&b[3])^(t[6]&b[7])^(t
       [7]&b[6])^(t[8]&b[7])^(t[9]&b[6])^(t[12]&b[7])^(t[14]&b[7])^(t[15]&b[6])^(t
       [15]&b[7])^(t[16]&b[6])^(t[16]&b[7])^(t[20]&b[7])^(t[26])^(t[0]&b[2]&b[7])^(
       t[30])^(t[31])^(t[32])^(t[0]&t[24])^(t[35])^(t[36])^(t[1]&t[18])^(t[38])^(t
       [1]&t[21])^(t[1]&t[23])^(t[41])^(t[2]&t[20])^(t[2]&t[21])^(t[42])^(t[2]&t
       [23])^(t[43])^(t[3]&t[23])^(t[3]&t[24])^(t[4]&t[24])^(t[6]&t[17])^(t[47])^(t
       [6]&t[21])^(t[6]&t[24])^(t[49])^(t[7]&t[21])^(t[51])^(t[7]&t[23])^(t[8]&t
       [23])^(t[9]&t[24])^(t[53])^(t[11]&t[21])^(t[11]&t[24])^(t[54])^(t[15]&t[23])
       ^(t[15]&t[24])^(t[16]&t[24])^(t[19]&t[24])^(t[25]&b[4])^(t[26]&b[6])^(t[27]&
       b[6])^(t[28]&b[7])^(t[29]&b[5])^(t[30]&b[7])^(t[31]&b[7])^(t[32]&b[7])^(t
       [33]&b[7])^(t[34]&b[7])^(t[37]&b[7])^(t[38]&b[6])^(t[41]&b[7])^(t[2]&t[20]&b
       [7])^(t[44]&b[7])^(t[45]&b[7])^(t[46]&b[6])^(t[46]&b[7])^(t[48]&b[7])^(t
       [49]&b[6])^(t[49]&b[7])^(t[50]&b[7])^(t[8]&t[22]&b[7])^(t[52]&b[7])^(t[53]&b
       [7])^(t[54]&b[7])^(t[55]&b[7])^(t[25]&t[19])^(t[25]&t[21])^(t[25]&t[22])^(t
       [25]&t[23])^(t[25]&t[24])^(t[26]&t[23])^(t[27]&t[24])^(t[29]&t[23])^(t[32]&t
       [24])^(t[35]&t[22])^(t[35]&t[23])^(t[36]&t[24])^(t[41]&t[24])^(t[44]&t[24])
       ^(t[49]&t[24])^(t[25]&t[19]&b[6])^(t[25]&t[19]&b[7])^(t[25]&t[20]&b[7])^(t
       [25]&t[22]&b[7]);
65 out[1]=(b[3])^(b[6])^(b[7])^(t[0])^(b[0]&b[7])^(t[6])^(t[9])^(t[10])^(b[1]&b[7])
       ^(t[12])^(t[13])^(t[15])^(t[16])^(t[17])^(t[22])^(t[24])^(t[0]&b[2])^(t[0]&b
       [3])^(t[0]&b[6])^(t[0]&b[7])^(t[1]&b[3])^(t[1]&b[4])^(t[2]&b[6])^(t[3]&b[6])
       ^(t[3]&b[7])^(t[6]&b[3])^(t[6]&b[7])^(t[7]&b[4])^(t[7]&b[5])^(t[7]&b[6])^(t
```

```
[7]&b[7])^(t[8]&b[6])^(t[8]&b[7])^(t[11]&b[4])^(t[11]&b[7])^(t[12]&b[5])^(t
[12]&b[6])^(t[12]&b[7])^(t[13]&b[6])^(t[13]&b[7])^(t[15]&b[5])^(t[15]&b[7])
^(t[16]&b[7])^(t[17]&b[7])^(t[25])^(t[0]&b[2]&b[7])^(t[29])^(t[0]&t[18])^(t
[32])^(t[33])^(t[0]&t[21])^(t[34])^(t[0]&t[24])^(t[36])^(t[37])^(t[39])^(t
[1]&t[21])^(t[40])^(t[1]&t[24])^(t[2]&t[21])^(t[43])^(t[3]&t[24])^(t[45])^(t
[6]&t[17])^(t[48])^(t[7]&t[24])^(t[8]&t[23])^(t[8]&t[24])^(t[9]&t[24])^(t
[52])^(t[53])^(t[11]&t[24])^(t[54])^(t[55])^(t[15]&t[23])^(t[15]&t[24])^(t
[26]&b[5])^(t[26]&b[6])^(t[26]&b[7])^(t[27]&b[6])^(t[27]&b[7])^(t[29]&b[7])
^(t[30]&b[7])^(t[31]&b[7])^(t[32]&b[6])^(t[35]&b[5])^(t[35]&b[7])^(t[36]&b
[6])^(t[37]&b[7])^(t[38]&b[7])^(t[39]&b[7])^(t[40]&b[7])^(t[42]&b[7])^(t
[43]&b[7])^(t[44]&b[5])^(t[44]&b[6])^(t[44]&b[7])^(t[45]&b[6])^(t[45]&b[7])
^(t[6]&t[17]&b[7])^(t[46]&b[7])^(t[47]&b[7])^(t[48]&b[7])^(t[49]&b[6])^(t
[50]&b[7])^(t[8]&t[22]&b[7])^(t[52]&b[7])^(t[54]&b[7])^(t[25]&t[19])^(t[25]&
t[20])^(t[25]&t[23])^(t[27]&t[24])^(t[29]&t[23])^(t[29]&t[24])^(t[30]&t[24])
^(t[32]&t[24])^(t[35]&t[23])^(t[35]&t[24])^(t[38]&t[24])^(t[44]&t[22])^(t
[45]&t[24])^(t[52]&t[24])^(t[25]&t[19]&b[6])^(t[25]&t[19]&b[7])^(t[25]&t
[20]&b[7])^(t[26]&t[22]&b[7])^(t[29]&t[22]&b[7]);
out[2]=(0xffffffffffffffff)^(b[1])^(b[4])^(t[0])^(t[1])^(t[3])^(t[4])^(t[6])^(t
[7])^(t[9])^(t[10])^(b[1]&b[7])^(t[13])^(t[15])^(t[16])^(t[19])^(t[20])^(t
[22])^(t[0]&b[7])^(t[1]&b[4])^(t[2]&b[7])^(t[3]&b[5])^(t[3]&b[7])^(t[5]&b
[7])^(t[6]&b[3])^(t[6]&b[5])^(t[7]&b[6])^(t[7]&b[7])^(t[8]&b[5])^(t[8]&b[6])
^(t[9]&b[7])^(t[11]&b[5])^(t[13]&b[6])^(t[14]&b[7])^(t[15]&b[5])^(t[15]&b
[7])^(t[16]&b[7])^(t[17]&b[7])^(t[27])^(t[28])^(t[0]&b[2]&b[7])^(t[29])^(t
[30])^(t[31])^(t[0]&t[18])^(t[32])^(t[0]&t[24])^(t[35])^(t[37])^(t[1]&t[18])
^(t[38])^(t[39])^(t[40])^(t[41])^(t[2]&t[21])^(t[42])^(t[2]&t[23])^(t[2]&t
[24])^(t[43])^(t[44])^(t[45])^(t[6]&t[17])^(t[6]&t[18])^(t[46])^(t[47])^(t
[48])^(t[6]&t[23])^(t[6]&t[24])^(t[49])^(t[50])^(t[7]&t[21])^(t[8]&t[22])^(t
[8]&t[24])^(t[11]&t[21])^(t[11]&t[23])^(t[11]&t[24])^(t[54])^(t[12]&t[24])^(
t[55])^(t[15]&t[23])^(t[15]&t[24])^(t[19]&t[24])^(t[25]&b[6])^(t[25]&b[7])^(
t[26]&b[5])^(t[29]&b[5])^(t[30]&b[7])^(t[31]&b[7])^(t[32]&b[7])^(t[33]&b[7])
^(t[35]&b[5])^(t[35]&b[6])^(t[38]&b[7])^(t[40]&b[7])^(t[41]&b[6])^(t[44]&b
[5])^(t[45]&b[6])^(t[6]&t[17]&b[7])^(t[46]&b[6])^(t[46]&b[7])^(t[47]&b[7])^(
t[49]&b[7])^(t[50]&b[7])^(t[51]&b[7])^(t[52]&b[6])^(t[53]&b[7])^(t[11]&t
[22]&b[7])^(t[54]&b[7])^(t[55]&b[7])^(t[25]&t[19])^(t[25]&t[23])^(t[25]&t
[24])^(t[26]&t[23])^(t[27]&t[24])^(t[29]&t[22])^(t[29]&t[24])^(t[30]&t[24])
^(t[35]&t[23])^(t[36]&t[24])^(t[38]&t[24])^(t[41]&t[24])^(t[44]&t[22])^(t
[44]&t[24])^(t[52]&t[24])^(t[25]&t[19]&b[6])^(t[25]&t[20]&b[7])^(t[29]&t
[22]&b[7])^(t[44]&t[22]&b[7]);
out[3]=(0xffffffffffffffff)^(b[0])^(b[3])^(b[4])^(b[6])^(b[7])^(t[0])^(t[1])^(t
[2])^(t[3])^(t[5])^(t[6])^(t[7])^(t[8])^(t[11])^(t[13])^(b[2]&b[7])^(t[15])
^(t[17])^(t[18])^(t[19])^(t[20])^(t[21])^(t[0]&b[3])^(t[0]&b[4])^(t[0]&b[6])
^(t[0]&b[7])^(t[1]&b[5])^(t[1]&b[6])^(t[2]&b[5])^(t[3]&b[7])^(t[6]&b[6])^(t
[6]&b[7])^(t[7]&b[5])^(t[7]&b[7])^(t[8]&b[5])^(t[10]&b[7])^(t[11]&b[6])^(t
[13]&b[6])^(t[13]&b[7])^(t[15]&b[6])^(t[15]&b[7])^(t[16]&b[7])^(t[17]&b[7])
^(t[19]&b[7])^(t[20]&b[7])^(t[27])^(t[28])^(t[29])^(t[31])^(t[33])^(t[0]&t
[21])^(t[34])^(t[0]&t[23])^(t[0]&t[24])^(t[35])^(t[39])^(t[40])^(t[1]&t[24])
^(t[43])^(t[3]&t[23])^(t[3]&t[24])^(t[4]&t[24])^(t[44])^(t[45])^(t[48])^(t
[6]&t[23])^(t[6]&t[24])^(t[49])^(t[50])^(t[8]&t[22])^(t[9]&t[24])^(t[53])^(t
[11]&t[21])^(t[11]&t[23])^(t[54])^(t[12]&t[24])^(t[55])^(t[15]&t[23])^(t
[15]&t[24])^(t[16]&t[24])^(t[25]&b[6])^(t[26]&b[6])^(t[28]&b[7])^(t[29]&b
[5])^(t[29]&b[6])^(t[29]&b[7])^(t[30]&b[7])^(t[32]&b[7])^(t[33]&b[7])^(t
[34]&b[7])^(t[35]&b[5])^(t[35]&b[6])^(t[36]&b[6])^(t[36]&b[7])^(t[37]&b[7])
```

```
^(t[38]&b[6])^(t[39]&b[7])^(t[41]&b[6])^(t[2]&t[20]&b[7])^(t[43]&b[7])^(t
    [46]&b[6])^(t[47]&b[7])^(t[49]&b[6])^(t[51]&b[7])^(t[52]&b[6])^(t[52]&b[7])
    ^(t[53]&b[7])^(t[54]&b[7])^(t[55]&b[7])^(t[25]&t[20])^(t[25]&t[22])^(t[25]&t
    [24])^(t[26]&t[22])^(t[29]&t[22])^(t[29]&t[24])^(t[32]&t[24])^(t[35]&t[23])
    ^(t[38]&t[24])^(t[41]&t[24])^(t[49]&t[24])^(t[52]&t[24])^(t[25]&t[19]&b[6])
    ^(t[25]&t[20]&b[7])^(t[25]&t[22]&b[7])^(t[35]&t[22]&b[7]);
out[4]=(0xffffffffffffffff)^(b[1])^(b[4])^(b[5])^(t[0])^(t[5])^(t[6])^(t[9])^(t
    [10])^(b[1]&b[7])^(t[12])^(t[16])^(t[17])^(t[18])^(t[21])^(t[22])^(t[23])^(t
    [0]&b[2])^(t[0]&b[3])^(t[0]&b[4])^(t[0]&b[5])^(t[0]&b[6])^(t[0]&b[7])^(t[1]&
    b[5])^(t[1]&b[6])^(t[2]&b[4])^(t[2]&b[6])^(t[3]&b[5])^(t[3]&b[6])^(t[3]&b
    [7])^(t[4]&b[6])^(t[6]&b[7])^(t[7]&b[4])^(t[7]&b[6])^(t[8]&b[5])^(t[8]&b[7])
    ^(t[9]&b[6])^(t[9]&b[7])^(t[10]&b[7])^(t[11]&b[6])^(t[11]&b[7])^(t[12]&b[5])
    ^(t[12]&b[6])^(t[12]&b[7])^(t[13]&b[7])^(t[14]&b[7])^(t[15]&b[6])^(t[16]&b
    [6])^(t[16]&b[7])^(t[17]&b[7])^(t[19]&b[6])^(t[19]&b[7])^(t[22]&b[7])^(t
    [25])^(t[26])^(t[27])^(t[28])^(t[29])^(t[30])^(t[0]&t[18])^(t[0]&t[21])^(t
    [38])^(t[39])^(t[1]&t[24])^(t[42])^(t[2]&t[24])^(t[44])^(t[45])^(t[46])^(t
    [6]&t[21])^(t[6]&t[23])^(t[6]&t[24])^(t[7]&t[21])^(t[51])^(t[7]&t[23])^(t
    [9]&t[24])^(t[52])^(t[53])^(t[11]&t[21])^(t[11]&t[23])^(t[11]&t[24])^(t[54])
    ^(t[12]&t[23])^(t[13]&t[24])^(t[55])^(t[15]&t[23])^(t[16]&t[24])^(t[19]&t
    [24])^(t[25]&b[5])^(t[25]&b[6])^(t[25]&b[7])^(t[26]&b[6])^(t[27]&b[7])^(t
    [29]&b[5])^(t[30]&b[6])^(t[30]&b[7])^(t[31]&b[7])^(t[32]&b[6])^(t[34]&b[7])
    ^(t[35]&b[5])^(t[36]&b[6])^(t[36]&b[7])^(t[37]&b[7])^(t[39]&b[7])^(t[41]&b
    [6])^(t[42]&b[7])^(t[45]&b[6])^(t[45]&b[7])^(t[46]&b[6])^(t[49]&b[7])^(t
    [51]&b[7])^(t[52]&b[7])^(t[11]&t[22]&b[7])^(t[55]&b[7])^(t[25]&t[20])^(t
    [25]&t[21])^(t[26]&t[23])^(t[29]&t[22])^(t[29]&t[23])^(t[29]&t[24])^(t[30]&t
    [24])^(t[32]&t[24])^(t[35]&t[22])^(t[35]&t[23])^(t[44]&t[22])^(t[44]&t[23])
    ^(t[44]&t[24])^(t[46]&t[24])^(t[49]&t[24])^(t[52]&t[24])^(t[25]&t[19]&b[7])
    ^(t[25]&t[20]&b[7])^(t[26]&t[22]&b[7])^(t[29]&t[22]&b[7])^(t[35]&t[22]&b[7])
    ^(t[44]&t[22]&b[7]);
out[5]=(b[1])^(b[2])^(t[0])^(t[1])^(t[3])^(t[4])^(t[5])^(b[0]&b[7])^(t[6])^(t
    [7])^(t[8])^(t[9])^(b[1]&b[7])^(t[13])^(b[2]&b[7])^(t[16])^(t[18])^(t[21])^(
    t[24])^(t[0]&b[2])^(t[0]&b[3])^(t[0]&b[4])^(t[0]&b[6])^(t[0]&b[7])^(t[1]&b
    [3])^(t[1]&b[7])^(t[3]&b[5])^(t[3]&b[6])^(t[5]&b[7])^(t[6]&b[3])^(t[6]&b[6])
    ^(t[6]&b[7])^(t[7]&b[4])^(t[7]&b[5])^(t[8]&b[5])^(t[8]&b[6])^(t[8]&b[7])^(t
    [9]&b[6])^(t[9]&b[7])^(t[11]&b[7])^(t[12]&b[5])^(t[12]&b[7])^(t[13]&b[6])^(t
    [13]&b[7])^(t[15]&b[7])^(t[16]&b[7])^(t[17]&b[7])^(t[19]&b[7])^(t[20]&b[7])
    ^(t[22]&b[7])^(t[25])^(t[26])^(t[27])^(t[29])^(t[0]&t[18])^(t[32])^(t[33])^(
    t[0]&t[21])^(t[0]&t[24])^(t[35])^(t[37])^(t[1]&t[18])^(t[39])^(t[1]&t[21])^(
    t[40])^(t[1]&t[23])^(t[2]&t[20])^(t[2]&t[24])^(t[3]&t[23])^(t[44])^(t[45])^(
    t[47])^(t[6]&t[21])^(t[48])^(t[6]&t[23])^(t[49])^(t[50])^(t[7]&t[21])^(t
    [51])^(t[7]&t[23])^(t[7]&t[24])^(t[8]&t[24])^(t[52])^(t[53])^(t[12]&t[24])^(
    t[13]&t[24])^(t[15]&t[24])^(t[19]&t[24])^(t[25]&b[6])^(t[26]&b[5])^(t[26]&b
    [6])^(t[29]&b[5])^(t[32]&b[7])^(t[33]&b[7])^(t[34]&b[7])^(t[35]&b[5])^(t
    [36]&b[6])^(t[36]&b[7])^(t[37]&b[7])^(t[39]&b[7])^(t[40]&b[7])^(t[41]&b[6])
    ^(t[2]&t[20]&b[7])^(t[43]&b[7])^(t[44]&b[6])^(t[45]&b[6])^(t[45]&b[7])^(t
    [46]&b[6])^(t[47]&b[7])^(t[48]&b[7])^(t[50]&b[7])^(t[51]&b[7])^(t[52]&b[7])
    ^(t[54]&b[7])^(t[25]&t[19])^(t[25]&t[20])^(t[25]&t[21])^(t[25]&t[22])^(t
    [26]&t[22])^(t[26]&t[23])^(t[26]&t[24])^(t[29]&t[22])^(t[29]&t[23])^(t[30]&t
    [24])^(t[35]&t[23])^(t[41]&t[24])^(t[44]&t[22])^(t[44]&t[24])^(t[45]&t[24])
    ^(t[49]&t[24])^(t[52]&t[24])^(t[25]&t[19]&b[6])^(t[25]&t[20]&b[7])^(t[35]&t
    [22]&b[7]);
```

```
70 out[6]=(0xffffffffffffffff)^(b[0])^(b[3])^(b[6])^(t[4])^(t[6])^(t[9])^(t[11])^(t
       [12])^(t[13])^(t[15])^(t[17])^(t[18])^(t[23])^(t[24])^(t[0]&b[3])^(t[0]&b
       [6])^(t[1]&b[3])^(t[1]&b[5])^(t[1]&b[6])^(t[1]&b[7])^(t[2]&b[6])^(t[2]&b[7])
       ^(t[3]&b[5])^(t[3]&b[7])^(t[4]&b[6])^(t[4]&b[7])^(t[5]&b[7])^(t[6]&b[3])^(t
       [7]&b[7])^(t[8]&b[7])^(t[10]&b[7])^(t[11]&b[5])^(t[12]&b[5])^(t[12]&b[7])^(t
       [13]&b[7])^(t[15]&b[7])^(t[16]&b[7])^(t[19]&b[7])^(t[22]&b[7])^(t[28])^(t
       [0]&b[2]&b[7])^(t[29])^(t[0]&t[21])^(t[0]&t[24])^(t[38])^(t[39])^(t[1]&t
       [21])^(t[40])^(t[1]&t[23])^(t[1]&t[24])^(t[2]&t[21])^(t[2]&t[23])^(t[3]&t
       [23])^(t[4]&t[24])^(t[45])^(t[46])^(t[47])^(t[6]&t[21])^(t[6]&t[23])^(t[6]&t
       [24])^(t[7]&t[21])^(t[51])^(t[7]&t[24])^(t[8]&t[23])^(t[8]&t[24])^(t[9]&t
       [24])^(t[53])^(t[11]&t[23])^(t[54])^(t[12]&t[24])^(t[15]&t[23])^(t[16]&t
       [24])^(t[19]&t[24])^(t[25]&b[4])^(t[25]&b[6])^(t[27]&b[7])^(t[29]&b[5])^(t
       [29]&b[6])^(t[30]&b[6])^(t[35]&b[6])^(t[36]&b[6])^(t[36]&b[7])^(t[37]&b[7])
       ^(t[41]&b[7])^(t[42]&b[7])^(t[43]&b[7])^(t[44]&b[5])^(t[44]&b[7])^(t[45]&b
       [6])^(t[46]&b[6])^(t[46]&b[7])^(t[47]&b[7])^(t[49]&b[6])^(t[50]&b[7])^(t
       [51]&b[7])^(t[52]&b[7])^(t[53]&b[7])^(t[11]&t[22]&b[7])^(t[55]&b[7])^(t[25]&
       t[19])^(t[25]&t[20])^(t[25]&t[21])^(t[25]&t[22])^(t[25]&t[24])^(t[26]&t[22])
       ^(t[26]&t[24])^(t[29]&t[23])^(t[30]&t[24])^(t[35]&t[23])^(t[35]&t[24])^(t
       [36]&t[24])^(t[38]&t[24])^(t[49]&t[24])^(t[52]&t[24])^(t[25]&t[20]&b[7])^(t
       [26]&t[22]&b[7])^(t[44]&t[22]&b[7]);
71 out[7]=(b[0])^(b[4])^(b[5])^(t[0])^(t[3])^(t[4])^(t[7])^(b[1]&b[7])^(t[11])^(t
       [12])^(t[14])^(t[15])^(t[16])^(t[17])^(t[18])^(t[22])^(t[0]&b[2])^(t[0]&b
       [4])^(t[0]&b[5])^(t[0]&b[6])^(t[0]&b[7])^(t[1]&b[5])^(t[1]&b[6])^(t[1]&b[7])
       ^(t[2]&b[7])^(t[4]&b[6])^(t[5]&b[7])^(t[6]&b[6])^(t[7]&b[5])^(t[7]&b[6])^(t
       [9]&b[6])^(t[10]&b[7])^(t[11]&b[4])^(t[11]&b[5])^(t[11]&b[7])^(t[12]&b[7])^(
       t[15]&b[5])^(t[15]&b[6])^(t[15]&b[7])^(t[16]&b[6])^(t[16]&b[7])^(t[17]&b[7])
       ^(t[19]&b[6])^(t[26])^(t[28])^(t[0]&b[2]&b[7])^(t[34])^(t[36])^(t[1]&t[18])
       ^(t[38])^(t[39])^(t[1]&t[21])^(t[40])^(t[2]&t[20])^(t[42])^(t[2]&t[23])^(t
       [2]&t[24])^(t[3]&t[23])^(t[4]&t[24])^(t[6]&t[18])^(t[48])^(t[6]&t[24])^(t
       [50])^(t[7]&t[21])^(t[7]&t[23])^(t[8]&t[23])^(t[8]&t[24])^(t[9]&t[24])^(t
       [52])^(t[11]&t[21])^(t[11]&t[23])^(t[11]&t[24])^(t[12]&t[24])^(t[15]&t[23])
       ^(t[15]&t[24])^(t[25]&b[5])^(t[25]&b[6])^(t[26]&b[7])^(t[27]&b[6])^(t[27]&b
       [7])^(t[28]&b[7])^(t[29]&b[7])^(t[30]&b[7])^(t[31]&b[7])^(t[32]&b[7])^(t
       [33]&b[7])^(t[34]&b[7])^(t[35]&b[6])^(t[35]&b[7])^(t[36]&b[6])^(t[37]&b[7])
       ^(t[38]&b[7])^(t[40]&b[7])^(t[41]&b[7])^(t[2]&t[20]&b[7])^(t[42]&b[7])^(t
       [43]&b[7])^(t[44]&b[7])^(t[6]&t[17]&b[7])^(t[46]&b[6])^(t[46]&b[7])^(t[47]&b
       [7])^(t[49]&b[6])^(t[49]&b[7])^(t[50]&b[7])^(t[52]&b[7])^(t[53]&b[7])^(t
       [54]&b[7])^(t[55]&b[7])^(t[25]&t[20])^(t[25]&t[21])^(t[25]&t[23])^(t[25]&t
       [24])^(t[26]&t[22])^(t[29]&t[22])^(t[29]&t[23])^(t[29]&t[24])^(t[32]&t[24])
       ^(t[36]&t[24])^(t[38]&t[24])^(t[41]&t[24])^(t[44]&t[23])^(t[44]&t[24])^(t
       [45]&t[24])^(t[25]&t[20]&b[7])^(t[26]&t[22]&b[7])^(t[35]&t[22]&b[7])^(t[44]&
       t[22]&b[7]);
```

Listing A.1: Optimized Boolean Representation of CSA Block Cipher S-box

A.2. CSA Encryption Test Vectors

```
1 CSA key: de be 67 03 e6 ec 3b 0d
```

```
2 expanded key: d0 96 f7 0b 7d 3a 78 d7 e3 38 ec dd f8 cc 90 d5 dd ae ab ad 7d 93
      aa 28 74 09 bc f6 c5 70 4e c7 1d f8 da ee 0b 39 31 98 a6 65 77 7b b9 fe fe
      21 d8 b8 61 05 e0 ea 3d 0b
3 plaintext: 00 00 00 00 00 00 00 00
4
5 after round 0:
6 internal data: 00 00 00 00 00 0f 00 d1
7 used key: d0
8 s-box output this round: d1
9
10 after round 1:
11 internal data: 00 00 00 00 0f 74 d1 3c
12 used key: 96
13 s-box output this round: 3c
14
15 after round 2:
16 internal data: 00 00 00 0f 74 99 3c a0
17 used key: f7
18 s-box output this round: a0
19
20 after round 3:
21 internal data: 00 00 0f 74 99 59 a0 74
22 used key: 0b
23 s-box output this round: 74
24
25 after round 4:
26 internal data: 00 0f 74 99 59 1b 74 cf
27 used key: 7d
28 s-box output this round: cf
29
30 after round 5:
31 internal data: 0f 74 99 59 1b f1 cf 52
32 used key: 3a
33 s-box output this round: 52
34
35 after round 6:
36 internal data: 74 96 56 14 f1 2c 52 68
37 used key: 78
38 s-box output this round: 67
39
40 after round 7:
41 internal data: 96 22 60 85 2c bc 68 c3
42 used key: d7
43 s-box output this round: b7
44
45 after round 8:
46 internal data: 22 f6 13 ba bc 94 c3 28
47 used key: e3
48 s-box output this round: be
49
50 after round 9:
51 internal data: f6 31 98 9e 94 9a 28 ca
```

```
used key: 38
s-box output this round: e8

after round 10:
internal data: 31 6e 68 62 9a 92 ca 79
used key: ec
s-box output this round: 8f

after round 11:
internal data: 6e 59 53 ab 92 57 79 eb
used key: dd
s-box output this round: da

after round 12:
internal data: 59 3d c5 fc 57 57 eb fb
used key: f8
s-box output this round: 95

after round 13:
internal data: 3d 9c a5 0e 57 c3 fb dd
used key: cc
s-box output this round: 84

after round 14:
internal data: 9c 98 33 6a c3 d2 dd f9
used key: 90
s-box output this round: c4

after round 15:
internal data: 98 af f6 5f d2 e6 f9 51
used key: d5
s-box output this round: cd

after round 16:
internal data: af 6e c7 4a e6 df 51 8d
used key: dd
s-box output this round: 15

after round 17:
internal data: 6e 68 e5 49 df f2 8d e8
used key: ae
s-box output this round: 47

after round 18:
internal data: 68 8b 27 b1 f2 bc e8 22
used key: ab
s-box output this round: 4c

after round 19:
internal data: 8b 4f d9 9a bc 58 22 66
used key: ad
s-box output this round: 0e
```

```
104
105 after round 20:
106 internal data: 4f 52 11 37 58 57 66 f7
107 used key: 7d
108 s-box output this round: 7c
109
110 after round 21:
111 internal data: 52 5e 78 17 57 59 f7 92
112 used key: 93
113 s-box output this round: dd
114
115 after round 22:
116 internal data: 5e 2a 45 05 59 01 92 6d
117 used key: aa
118 s-box output this round: 3f
119
120 after round 23:
121 internal data: 2a 1b 5b 07 01 43 6d 34
122 used key: 28
123 s-box output this round: 6a
124
125 after round 24:
126 internal data: 1b 71 2d 2b 43 c9 34 3c
127 used key: 74
128 s-box output this round: 16
129
130 after round 25:
131 internal data: 71 36 30 58 c9 25 3c 53
132 used key: 09
133 s-box output this round: 48
134
135 after round 26:
136 internal data: 36 41 29 b8 25 62 53 c8
137 used key: bc
138 s-box output this round: b9
139
140 after round 27:
141 internal data: 41 1f 8e 13 62 66 c8 6a
142 used key: f6
143 s-box output this round: 5c
144
145 after round 28:
146 internal data: 1f cf 52 23 66 ea 6a 44
147 used key: c5
148 s-box output this round: 05
149
150 after round 29:
151 internal data: cf 4d 3c 79 ea db 44 51
152 used key: 70
153 s-box output this round: 4e
154
155 after round 30:
```

```
156 internal data: 4d f3 b6 25 db 57 51 86
157 used key: 4e
158 s-box output this round: 49
159
160 after round 31:
161 internal data: f3 fb 68 96 57 8b 86 e6
162 used key: c7
163 s-box output this round: ab
164
165 after round 32:
166 internal data: fb 9b 65 a4 8b f9 e6 0e
167 used key: 1d
168 s-box output this round: fd
169
170 after round 33:
171 internal data: 9b 9e 5f 70 f9 96 0e d7
172 used key: f8
173 s-box output this round: 2c
174
175 after round 34:
176 internal data: 9e c4 eb 62 96 c7 d7 79
177 used key: da
178 s-box output this round: e2
179
180 after round 35:
181 internal data: c4 75 fc 08 c7 bf 79 3a
182 used key: ee
183 s-box output this round: a4
184
185 after round 36:
186 internal data: 75 38 cc 03 bf 2a 3a ad
187 used key: 0b
188 s-box output this round: 69
189
190 after round 37:
191 internal data: 38 b9 76 ca 2a 66 ad cd
192 used key: 39
193 s-box output this round: b8
194
195 after round 38:
196 internal data: b9 4e f2 12 66 61 cd 8a
197 used key: 31
198 s-box output this round: b2
199
200 after round 39:
201 internal data: 4e 4b ab df 61 8e 8a d8
202 used key: 98
203 s-box output this round: 61
204
205 after round 40:
206 internal data: 4b e5 91 2f 8e 4d d8 3d
207 used key: a6
```

```
208 s-box output this round: 73
209
210 after round 41:
211 internal data: e5 da 64 c5 4d 13 3d a8
212 used key: 65
213 s-box output this round: e3
214
215 after round 42:
216 internal data: da 81 20 a8 13 7b a8 d4
217 used key: 77
218 s-box output this round: 31
219
220 after round 43:
221 internal data: 81 fa 72 c9 7b 8a d4 df
222 used key: 7b
223 s-box output this round: 05
224
225 after round 44:
226 internal data: fa f3 48 fa 8a 78 df 17
227 used key: b9
228 s-box output this round: 96
229
230 after round 45:
231 internal data: f3 b2 00 70 78 dc 17 bb
232 used key: fe
233 s-box output this round: 41
234
235 after round 46:
236 internal data: b2 f3 83 8b dc c6 bb 99
237 used key: fe
238 s-box output this round: 6a
239
240 after round 47:
241 internal data: f3 31 39 6e c6 51 99 15
242 used key: 21
243 s-box output this round: a7
244
245 after round 48:
246 internal data: 31 ca 9d 35 51 02 15 38
247 used key: d8
248 s-box output this round: cb
249
250 after round 49:
251 internal data: ca ac 04 60 02 47 38 18
252 used key: b8
253 s-box output this round: 29
254
255 after round 50:
256 internal data: ac ce aa c8 47 8e 18 d5
257 used key: 61
258 s-box output this round: 1f
259
```

```
after round 51:
internal data: ce 06 64 eb 8e 17 d5 7d
used key: 05
s-box output this round: d1

after round 52:
internal data: 06 aa 25 40 17 24 7d a0
used key: e0
s-box output this round: 6e

after round 53:
internal data: aa 23 46 11 24 52 a0 d3
used key: ea
s-box output this round: d5

after round 54:
internal data: 23 ec bb 8e 52 3a d3 21
used key: 3d
s-box output this round: 8b

after round 55:
internal data: ec 98 ad 71 3a 30 21 44
used key: 0b
s-box output this round: 67
```

Listing A.2: CSA Encryption Test Vectors

A.3. Maple Script for TMTO Rainbow Probability Computation by Stefan Spitz

$Chainlength := 12.28;$

$Startingpoints := 38;$

$Keylength := 48;$

$Operationspersecond := 2^{16.66};$

$t := 2^{Chainlength};$

$ms := 2^{Startingpoints};$

$ms2 := ms;$

$Speicherbit := 2 * Keylength;$

$temp := evalf(1 - ms/2^{Keylength});$

for i from 2 to t do

$ms := evalf(2^{Keylength} * (1 - exp(-ms/2^{Keylength})));$

$f := evalf(1 - ms/2^{Keylength});$

$temp := temp * f$

end do;

$Prob := 1 - temp;$

$Erfolgswahrscheinlichkeit = Prob;$

$Speicheraufwand(Gigabyte) = ms2 * Speicherbit/(1024 * (8 * 1024) * 1024);$

$PrecomputationTag = ms2 * t/(24 * (60 * 60) * Operationspersecond);$

$OnlineSekunde = t * ((t - 1)/2)/Operationspersecond;$

Results

$Erfolgswahrscheinlichkeit = 0.9149917891$

$Speicheraufwand(Gigabyte) = 3072$

$PrecomputationTag = 1.527969593 * 10^5$

$OnlineSekunde = 119.4042093$

List of Figures

List of Tables

Bibliography

[AB] Adi Shamir Alex Biryukov. Cryptanalytic time/memory/data tradeoffs for stream ciphers. Technical report, Computer Science department, The Weizmann Institute.

[Bih97] Eli Biham. A fast new DES implementation in software. *Lecture Notes in Computer Science*, 1267:260–??, 1997.

[Bir] Alex Biryukov. Some thoughts on time-memory-data tradeoffs. Technical report, Katholieke Universiteit Leuven, Dept. ESAT/SCD-COSIC.

[CDC] Bart Preneel Christophe De Canni‘ere, Joseph Lano. Comments on the rediscovery of time memory data tradeoffs. Technical report, Katholieke Universiteit Leuven, Dept. Elect. Eng.-ESAT/SCD-COSIC.

[csa] Csa - known facts and speculations. `http://csa.irde.to/` [Online; accessed 05-October-2007].

[EB] Adi Shamir Elad Barkan, Eli Biham. Rigorous bounds on cryptanalytic time/memory tradeoffs. Technical report, Computer Science Department Technion Israel Institute of Technology, Department of Computer Science and Applied Mathematics, The Weizmann Institute.

[Flo67] Robert W. Floyd. Nondeterministic algorithms. *J. ACM*, 14(4):636–644, 1967.

[Hel80] Martin E. Hellman. A cryptanalytic time - memory trade-off. Technical report, IEEE, July 1980.

[ISO00] ISO13818-1. Information technology generic coding of moving pictures and associated audio information: Systems. Technical report, ISO, December 2000.

[JH] Palash Sarkar Jin Hong. Rediscovery of time memory tradeoffs. Technical report, National Security Research Institute - 161 Gajeong-dong, Cryptology Research Group, Applied Statistics Unit, Indian Statistical Institute.

[Joh07] Thomas Johansson. Stream ciphers: Cryptanalytic techniques. Technical report, Department of Electrical and Information Technology, Lund University, Sweden, 2007.

[Koc04] Dr. Cetin Kaya Koc. The common scrambling algorithm implementation using c. Technical report, ECE 575 Project, 2004.

[NM05] Xu Zhao Nariman Molavi. A security study of digital tv distribution systems. Technical report, Department of Computer and Systems Sciences, Royal Institute of Technology, June 2005.

[Oec] Philippe Oechslin. Making a faster cryptanalytic time-memory tradeoff. Technical report, Laboratoire de Securite et de Cryptographie (LASEC).

[Paa06] Christof Paar. Efficient software implementation of block ciphers, September 2006.

[RPW04] Kai Wirt Ralf-Philipp Weinmann. Analysis of the dvb common scrambling algorithm. Technical report, Technical University of Darmstadt, October 2004.

[Wir] Kai Wirt. Fault attack on the dvb common scrambling algorithm. Technical report, Technical University of Darmstadt.