FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Brute-force Angriff auf DVB-CSA1 mithilfe eines preisgünstigen FPGA-Clusters

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Ioannis Daktylidis
Matrikelnummer 1128193

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. DI Dr. Wolfgang KASTNER
Mitwirkung: DI Dr. Markus Kammerstetter

Wien, 15. April 2017

Ioannis Daktylidis

Wolfgang KASTNER

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Brute-force Attacks on DVB-CSA1 using a low-cost FPGA-Cluster

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Ioannis Daktylidis
Registration Number 1128193

to the Faculty of Informatics
at the TU Wien

Advisor: Ao.Univ.-Prof. DI Dr. Wolfgang KASTNER
Assistance: DI Dr. Markus Kammerstetter

Vienna, 15th April, 2017

Ioannis Daktylidis   Wolfgang KASTNER

Erklärung zur Verfassung der Arbeit

Ioannis Daktylidis
e1128193@student.tuwien.ac.at

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 15. April 2017

Ioannis Daktylidis

Abstract

The DVB Common Scrambling Algorithm (CSA) is widely used to encrypt PayTV MPEG transport streams around the world. A known weakness is that only 48-bits of the 64-bit key are actually used, bringing the entire key space to a level where a brute-force attack could succeed in an acceptable time period. In this thesis, a brute-force attack on hardware accessible by casual attackers is implemented. The thesis describes a high-performance brute-force implementation targeted for Xilinx Spartan 6 FPGAs operated in a 36-FPGA cluster. The results show that on the used hardware the key can be brute forced in approximately one and a half hours. Estimated hardware resources needed to brute-force a transport stream in a time faster than the key can be updated, exceed the resources available by casual attackers.

Contents

CHAPTER 1

Introduction

The Common Scrambling Algorithm (CSA) was adopted by the DVB Consortium in 1994 and has since then been used by digital TV providers all around the world to encrypt MPEG transport streams. The technical details of the scrambling algorithm are only made available to bona-fide users, upon signature of a Non-Disclosure Agreement administered by ETSI [12]. The NDA disallowed - and still does - the implementation of the CSA on software. [11]

Because of this, only parts of the structure of the CSA were known from a patent application [13], the ETSI Technical Specification [12], and a publication [14]. In fall of 2002, a windows program called FreeDec came out, which implemented the CSA in software. Though released as a binary only, it was quickly disassembled to reveal the missing details of the CSA, such as the S-Boxes used. Since then the attack potential on CSA has been mitigated by the use of Conditional Access Systems like Irdeto, Betacrypt, Nagravision and CryptoWorks that are responsible for the exchange of the common words used by the scrambling algorithm. New common words are usually issued every 10-120 seconds. That was done, because these CA-Systems are easier updated than the underlying scrambling algorithm that is implemented in hardware.

Nevertheless, DVB has recognized the security implications of using a now over 20 years old algorithm and the old algorithm was superseded by CSA version 3 that was adopted in 2007. The new version uses a 128 bit key and is based on a variation of the "Advanced Encryption Standard" (AES128) defined in NIST FIPS 197 and the "eXtended emulation Resistant Cipher" (XRC) - a DVB-confidential cipher [12]. The old version was dubbed CSA-1 and the new CSA-3. According to ETSI, "CSA-1 was designed in 1994 to provide adequate security for a period of at least ten years. In 2007, DVB adopted CSA-3 in order to continue providing an adequate level of security taking into account advances in technology" [12]. Despite that, CSA-3 is not yet in significant use, and CSA-1 continues to be the dominant algorithm used for protecting DVB broadcasts.

Because the 3rd and 7th Byte of the common word are usually used as checksum Bytes, the effective key length is only 48 bits, which for today's hardware capabilities is a fairly small key space. If it is possible to brute-force the entire key space in less time than the CA-Systems need to issue a new common word, severe implications to all broadcasters still using CSA-1 would arise.

This thesis describes a brute-force implementation targeted for a cluster equipped with 36 Xilinx Spartan 6 FPGAs (XC6SLX150-3CSG484). The CSA was first implemented bottom up in hardware via a Hardware Description Language (VHDL), beginning with the S-Box of the block cipher. Afterwards the VHDL code was used to synthesize for the actual FPGAs, adding additional target specific constraints like floorplanning. In the end, the software responsible for the controlling of the FPGAs was implemented. An additional implementation for Xilinx Artix 7 FPGA boards (XC7A200TFBG484-2) is also presented.

Chapter 2 gives an insight to the internal structure of a DVB transport stream and the Common Scrambling Algorithm. Chapter 3 gives an overview of the attacks possible on the Common Scrambling Algorithm and already existing implementations. Chapter 4 describes the actual implementation of the brute-force procedure used in this thesis. Chapter 5 states the evaluation methods used. Chapter 6 presents the results of the used implementations and Chapter 7 holds the conclusion reached by this thesis. Note that the terms key and common word are used interchangeably in this thesis.

CHAPTER 2

Background

This Chapter gives an insight in the transport mechanism used by DVB, which will present the attack vector used in this thesis. This Chapter also details the internal structure of the Common Scrambling Algorithm version 1, needed for implementing it in hardware.

2.1 DVB

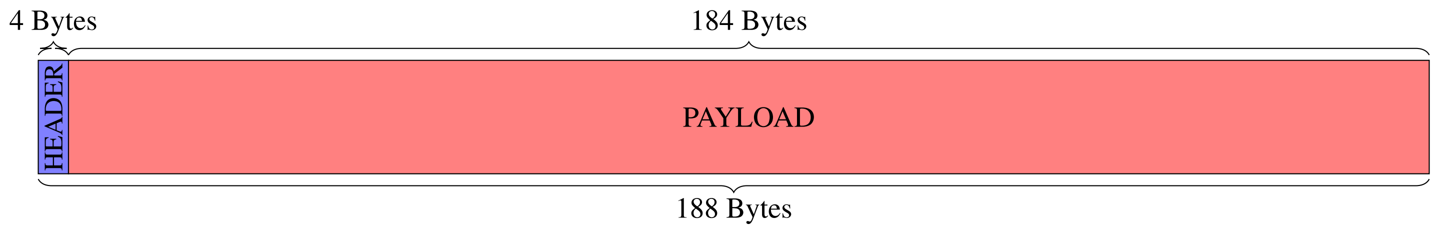
DVB stands for Digital Video Broadcasting and is a suite of internationally accepted standards for digital television, which are maintained by the DVB Project - an international industry consortium with more than 200 members [5].

DVB systems distribute data using Satellite (DVB-S, DVB-S2, DVB-SH), Cable (DVB-C, DVB-C2), Terrestrial Television (DVB-T, DVB-T2), and Microwave Signals (DVB-MT, DVB-MC, DVB-MS).
All data is transmitted in MPEG transport streams with some additional constraints, which differ mainly in the modulation schemes and error correcting codes used.
In the following, a more in detailed description of the MPEG transport stream is provided.

2.1.1 Transport Stream

The Transport Stream (TS) consists of TS Packets with a fixed length of 188 Bytes. Each packet has a 4 Byte Header and a 184 Byte payload.

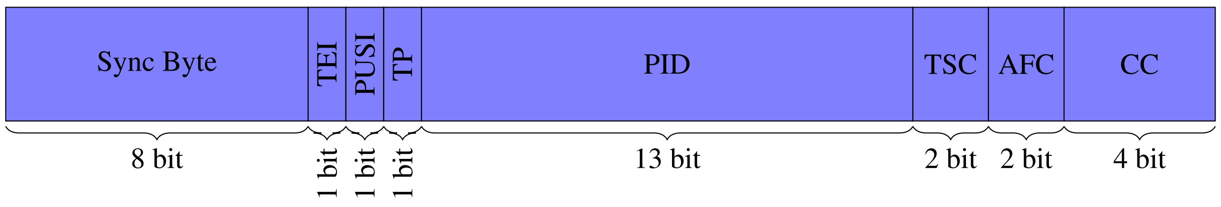
Figure 2.1: Transport Stream Packet



2.1.1.1 TS Header

The TS Header consists of the following:

Figure 2.2: Transport Stream Header



- **Sync Byte** A Byte with a fixed value of 0x47, that is used to indicate the start of the TS packet and Header.
- **Transport Error Indicator** A bit flag that is set, when the TS packet is corrupt (Has uncorrectable errors).
- **Payload Unit Start Indicator** A bit flag set when a PES (Packet Elementary Stream), PSI (Program Specific Information) or DVB-MIP packet begins in the payload.
- **Transport Priority** A bit flag set when the current TS packet has higher priority than the other packets of the same PID.
- **PID** A 13 bit Packet IDentifier, describing the payload data. This field is used to distinguish the different channels inside one stream.
- **Transport Scrambling Control** A 2 bit field indicating the scrambling mode used on the payload. The Header and the optional Adaptation Field should not be scrambled.
  - **00** Not scrambled. (Used for Null Packets)
  - **01** Reserved for future use.
  - **10** Scrambled with even key.
  - **11** Scrambled with odd key.

  The transport stream uses the concept of even and odd keys to allow a seamless key update procedure. During the time that one key is used for the decryption of the transport stream, a new value can be assigned to the other key.
  Note that when TS-level scrambling is used, PES-level scrambling cannot be used at the same time (mutual exclusive).
- **Adaptation Field Control** A 2 bit field indicating if this TS packet contains an Adaptation Field and/or a Payload.
  - **00** Reserved for future use.
  - **01** Payload only
  - **10** Adaptation Field only
  - **11** Adaptation Field followed by Payload
- **Continuity Counter** A 4 bit sequence counter that is incremented per PID inside a stream, when the TS packet contains a payload.

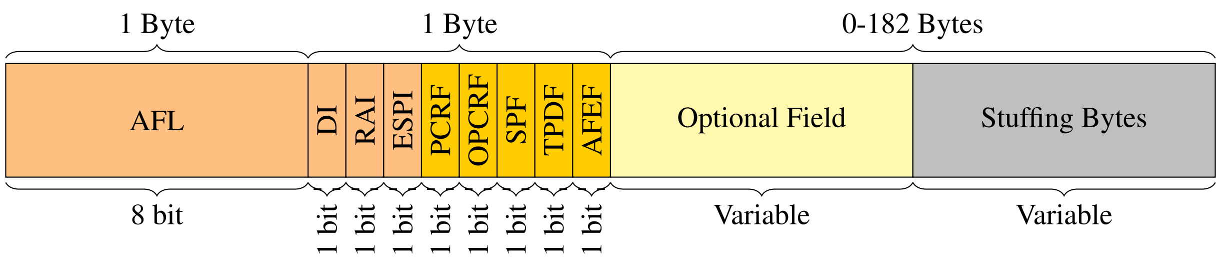
Table 2.1: PID Values

Value	Description
0x0000	Program Association Table (PAT)
0x0001	Conditional Access Table (CAT)
0x0002	Transport Stream Description Table (TSDT)
0x0003	IPMP Control Information Table
0x0004-0x000F	Reserved for future use
0x0010-0x001F	Used for DVB metadata
0x0020-0x1FFA	May be assigned as needed to Program Map Tables, Elementary Streams and other data tables
0x1FFB	Used by DigiCipher 2/ATSC MGT metadata
0x1FFC-0x1FFE	May be assigned as needed to Program Map Tables, Elementary Streams and other data tables
0x1FFF	Null Packet

2.1.1.2 Adaptation Field

If the TS packet contains an Adaptation Field, the TS Header is directly followed by the variable length Adaptation Field. The Adaptation Field is commonly used for stuffing Bytes, since the TS Packet has a fixed length of 188 Bytes, but the included MPEG Stream has variable length packets.

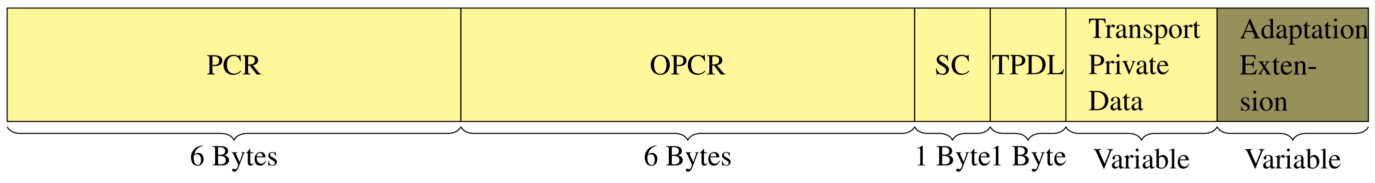
Beyond the first 8 bits of the Adaptation Field - which indicate the Adaptation Field Length - the data is unimportant for the purpose of this thesis, but are nevertheless described for the sake of completeness.



- **Adaptation Field Length** Indicates the number of Bytes in the Adaptation Field immediately following this Byte. Can also be 0. (Used for stuffing a single Byte before the payload).
- **Discontinuity Indicator** A bit flag set when the current TS packet is in a discontinuity state with respect to either the Continuity Counter or the Program Reference Clock.
- **Random Access Indicator** A bit flag set when the stream may be decoded without errors from this point.
- **Elementary Stream Priority Indicator** A bit flag set when this stream should be considered "high priority"
- **PCR Flag** (OPTIONAL FIELD) A bit flag set when the PCR Field is present in the Optional Field.
- **OPCR Flag** (OPTIONAL FIELD) A bit flag set when the OPCR Field is present in the Optional Field.
- **Splice Countdown** (OPTIONAL FIELD) A bit flag set when the Splice Countdown Field is present in the Optional Field.
- **Transport Private Data Flag** (OPTIONAL FIELD) A bit flag set when the Private Data Field is present in the Optional Field.

- **Adaptation Field Extension** (OPTIONAL FIELD) A bit flag set when the Extension Field is present.
- **Optional Field** See figure 2.3.
- **Stuffing Bytes** Bytes with a fixed value of 0xFF.

Figure 2.3: Optional Field



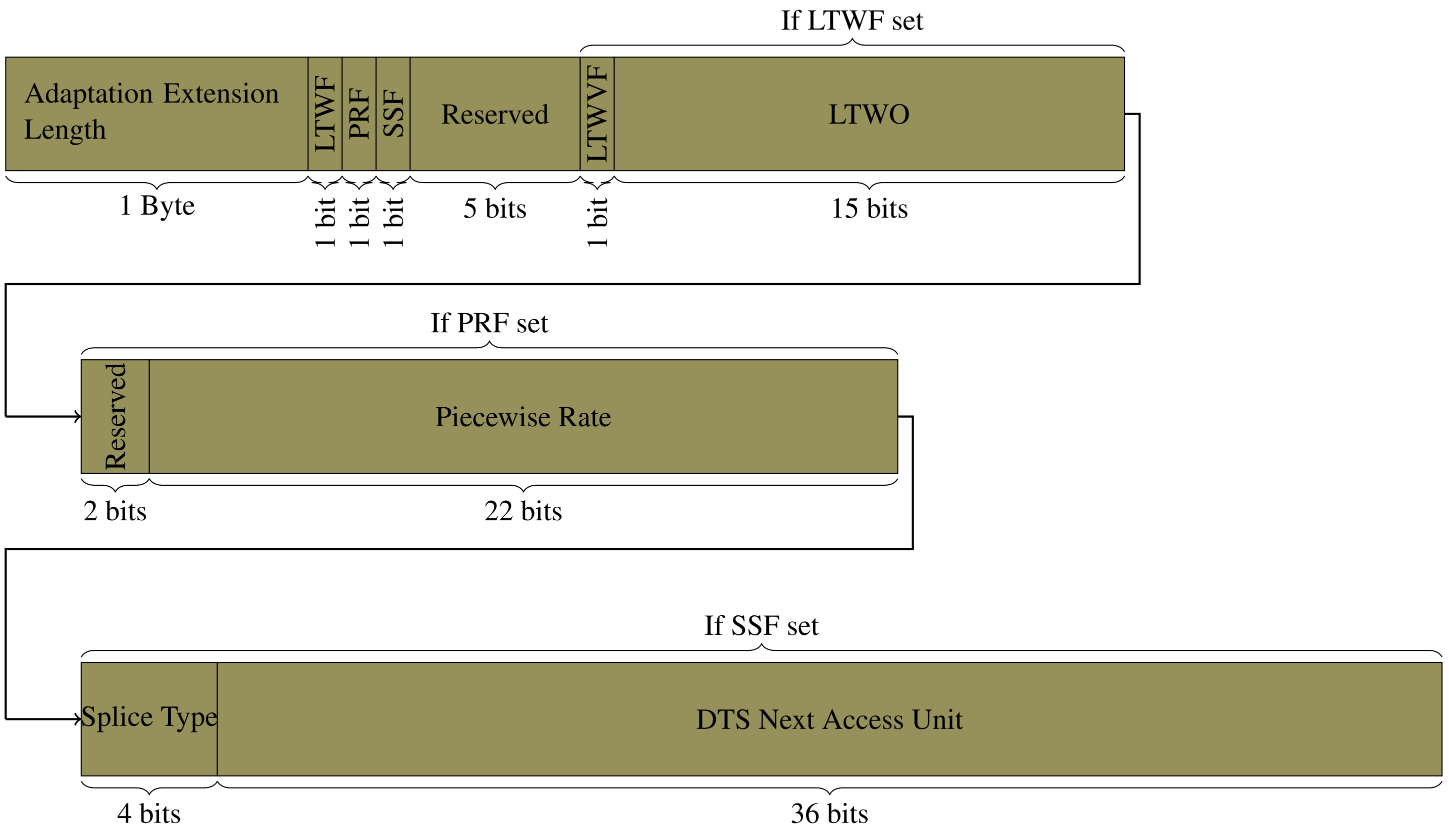
Optional Field

- **Program Clock Reference** A 6 Byte field containing the Program Reference Clock, stored as 33 bits base, 6 bits reserved and 9 bits extension.
  The value is calculated as $base * 300 + Extension$
- **Original Program Clock Reference** A 6 Byte field containing the Original Program Clock Reference, stored in a similar way as the PCR Field. This field helps when one TS is copied into another.
- **Splice Countdown** A Byte counter field indicating how many TS packets from this one a splicing point occurs, stored as a signed two's complement(May be negative).
- **Transport Private Data Length** A Byte field indicating the size of the Transport Private Data Field (The following Field)
- **Transport Private Data** Private Data
- **Adaptation Extension** See figure 2.4.

Adaptation Extension Field

- **Adaptation Extension Length** A 1 Byte field indicating the size of the Adaptation Extension Field.
- **Legal Time Window Flag** A bit flag set when the Legal Time Window Valid Flag and Legal Time Window Offset Fields are present.
- **Piecewise Rate Flag** A bit flag set when the Piecewise Rate Fields are present.
- **Seamless Splice Flag** A bit flag set when the Splice Type and DTS Next Access Unit Fields are present.
- **Legal Time Window Valid Flag**
- **Legal Time Window Offset** A 15 bit field giving extra information for broadcasters to determine the state of buffers when packets may be missing.
- **Piecewise Rate** A 22 bit field indicating the rate of the stream, measured in 188-Byte packets, to define the end-time of the Legal Time Window.
- **Splice Type** A 4 bit field indicating the parameters of the H.262 splice.
- **DTS Next Access Unit** A 36 bit field indicating the PES DTS splice point. Stored as:
  3 data bits, 1 marker bit (0x1), 15 data bits, 1 marker bit (0x1), 15 data bits, 1 marker bit (0x1).
  (33 Data bits in total)

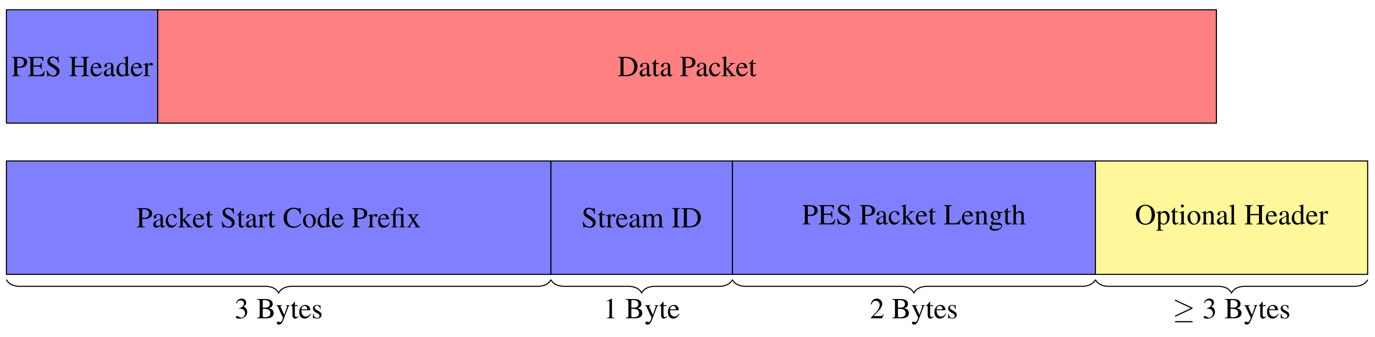
Figure 2.4: Adaptation Extension Field



2.1.2 Packetized Elementary Stream

The Packetized Elementary Stream (PES) is the target payload of this thesis and is specified in the MPEG-2 Part 1 (ISO/IEC 13818-1) and ITU-T H.222.0 that defines carrying of elementary streams in TS packets. Each PES packet has a variable length Header and is variable in its overall length as well, because the Video data packets it is carrying have also variable lengths. A PES packet is split to fit in the 184 Bytes of a TS packet payload. The rest of the TS packet payload area is filled with stuffing Bytes (0xFF). Each PES Header is put at the beginning of a new TS packet payload area.

Figure 2.5: PES Packet and Header



The PES Header consists of the following:

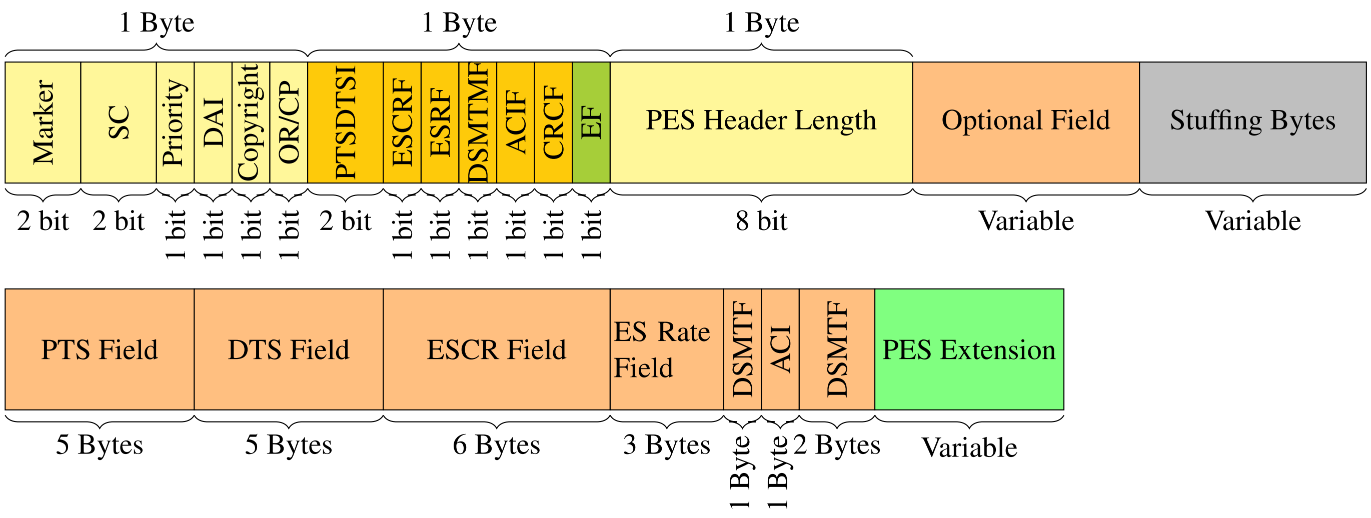
- **Packet Start Code Prefix** A fixed value of 0x000001 indicating the start of a PES Header.

- **Stream ID** A 1 Byte field indicating the kind of data in the PES packet.
- **PES Packet Length** A 2 Byte field specifying the number of remaining Bytes in this PES packet. If set to zero, the PES packet can be of any length (used when payload is a video Elementary Stream).
- **Optional Header** If present, the Optional PES Header has a minimum size of 3 Bytes.

Since the PES Header is contained inside the encrypted TS payload, when TS-level scrambling is used, the constant Packet Start Code Prefix can be considered a 3 Byte known plaintext of the ciphertext. This fact is used in this thesis to perform a brute-force attack on DVB-CSA, as described in Section 3.3.

2.1.2.1 Optional PES Header

Figure 2.6: Optional PES Header



- **Marker** A 2 bit Field containing a fixed value of 0b10 (0x2 in hex)
- **Scrambling Control** A 2 bit Field indicating the scrambling mode used on the PES. The PES Header should not be scrambled.
  - **00** Not scrambled.
  - **01** Reserved for future use.
  - **10** Scrambled with even key.
  - **11** Scrambled with odd key.

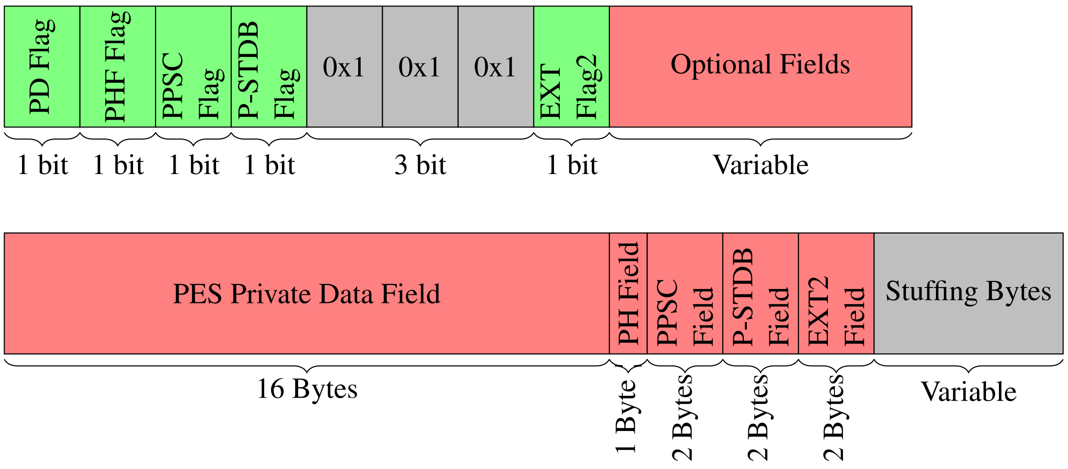
  As the case for TS-level scrambling, the concept of even and odd keys is used to allow a seamless key update procedure. During the time the one key is used for the decryption of the PES, a new value can be assigned to the other key.
  Note that when PES-level scrambling is used, TS-level scrambling cannot be used at the same time (mutual exclusive).
- **Priority** A 1 bit flag allowing to set 2 priority levels (0 and 1).

- **Date Alignment Indicator** A 1 bit field indicating that the PES Header is immediately followed by the video start code or audio sync word.
- **Copyright** A 1 bit flag indicating if the packet contains copyright material (1 = Copyright).
- **Original or Copy** A 1 bit flag indicating if the content of the packet is original or a copy (1 = original, 0 = copy).
- **PTS DTS Indicator** A 2 bit field indicating if the Header is followed by a Presentation Time Stamp/Decode Time Stamp in the Optional Fields.
  - **00** no PTS or DTS
  - **01** forbidden
  - **10** only PTS (5 Bytes)
  - **11** PTS and DTS (10 Bytes)
- **ESCR Flag** A 1 bit flag indicating if the Optional Fields contain the Elementary Stream Clock Reference. The ESCR is used if the stream and system levels are not synchronized (i.e. ESCR differs from SCR in the PACK Header). (6 Bytes)
- **ES Rate Flag** A 1 bit flag indicating if the Optional Fields contain the ES Rate. The ES Rate is the rate at which data is delivered for this stream, in units of 50 Bytes/second. (3 Bytes)
- **DSM Trick Mode Flag** A 1 bit flag indicating if the Optional Fields contain the DSM Trick Mode Field. The trick mode is used to describe the playback mode. First 3 bits of the optional field indicate the trick mode used.
  - **000** Fast Forward
  - **001** Slow Motion
  - **010** Freeze Frame
  - **011** Fast Forward
  - **100** Slow Reverse
  - **else** Reserved

  (1 Byte)
- **Additional Copy Info Flag** A 1 bit flag indicating if the Optional Field contains additional copy info data. (1 Byte)
- **CRC Flag** A 1 bit flag indicating if the Optional Field contains the CRC checksum of the previous PES packet. Polynomial used: $x^{16} + x^{12} + x^5 + 1$. (2 Bytes)
- **Extension Flag** A 1 bit flag indicating if the Optional Field contains a PES Extension Field. (min 1 Byte)
- **PES Header Length** A 1 Byte field indicating the length of the remainder of the PES Header.
- **Optional Fields**
  - **PES Private Data Flag** A 1 bit flag indicating if the Optional Field contains a user defined data field. (16 Bytes)
  - **Pack Header Field Flag** A 1 bit flag indicating if the Optional Field contains a pack field. (1 Byte)

  - **Program Packet Sequence Counter Flag** A 1 bit flag indicating if the Optional Field contains a program sequence counter field. (2 Bytes)
  - **P-STD Buffer Flag** A 1 bit flag indicating if the Optional Field contains a P-STD buffer field. (2 Bytes)
  - **PES Extension Flag 2** A 1 bit flag indicating if the Optional Field contains a PES Extension field. This extension field contains the length of Bytes immediately following the PES Extension field. (2 Bytes)
- **Stuffing Bytes**

Figure 2.7: Optional Field of Optional PES Header



2.2 CSA

The CSA version 1 uses two cryptographic primitives in its core: a 64-bit block cipher and a stream cipher initiated with a 64-bit nonce. Both ciphers use a 64-bit common word as the encryption/decryption key.

During encryption, the plaintext is split into 64-bit blocks (PB) and a plain residue block (PR). The 64-bit outputs of the block cipher form the intermediate blocks (IB). Each of the plaintext blocks is XORed with the intermediate block of the previous block cipher - the first plaintext block is XORed with a special initialization vector (IV) - and then fed into the respective block cipher. The stream cipher uses the first intermediate block and the common word to initialize itself. The stream cipher output is then XORed with the intermediate blocks to build the final ciphertext blocks (CB). Note that the plain residue (PR) is only XORed with the stream cipher output. The encryption path can be seen in figure 2.8.

The decryption path is just the inverted encryption path. Internally the stream cipher is the same as in encryption, but the Block cipher uses for decryption a different round function. The decryption path can be seen in figure 2.9.

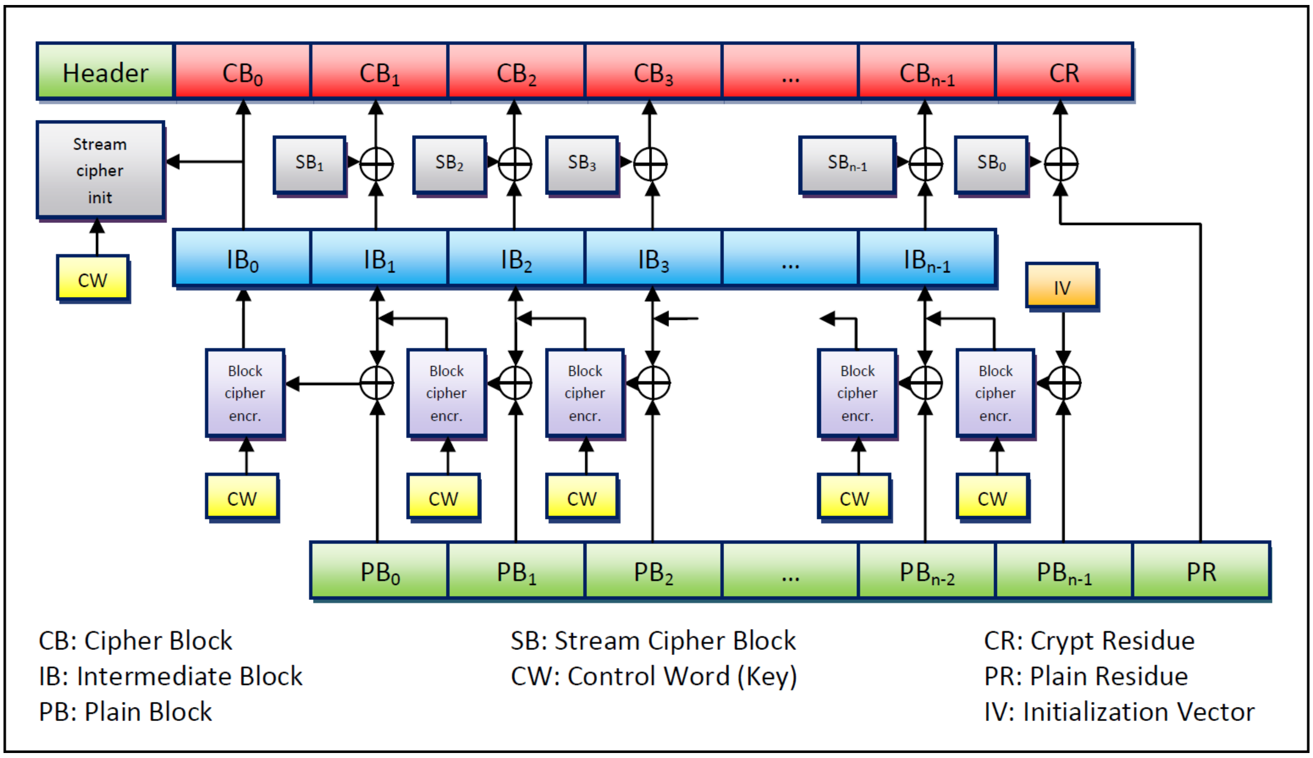


Figure 2.8: DVB-CSA1 encryption path - Colibri[4]

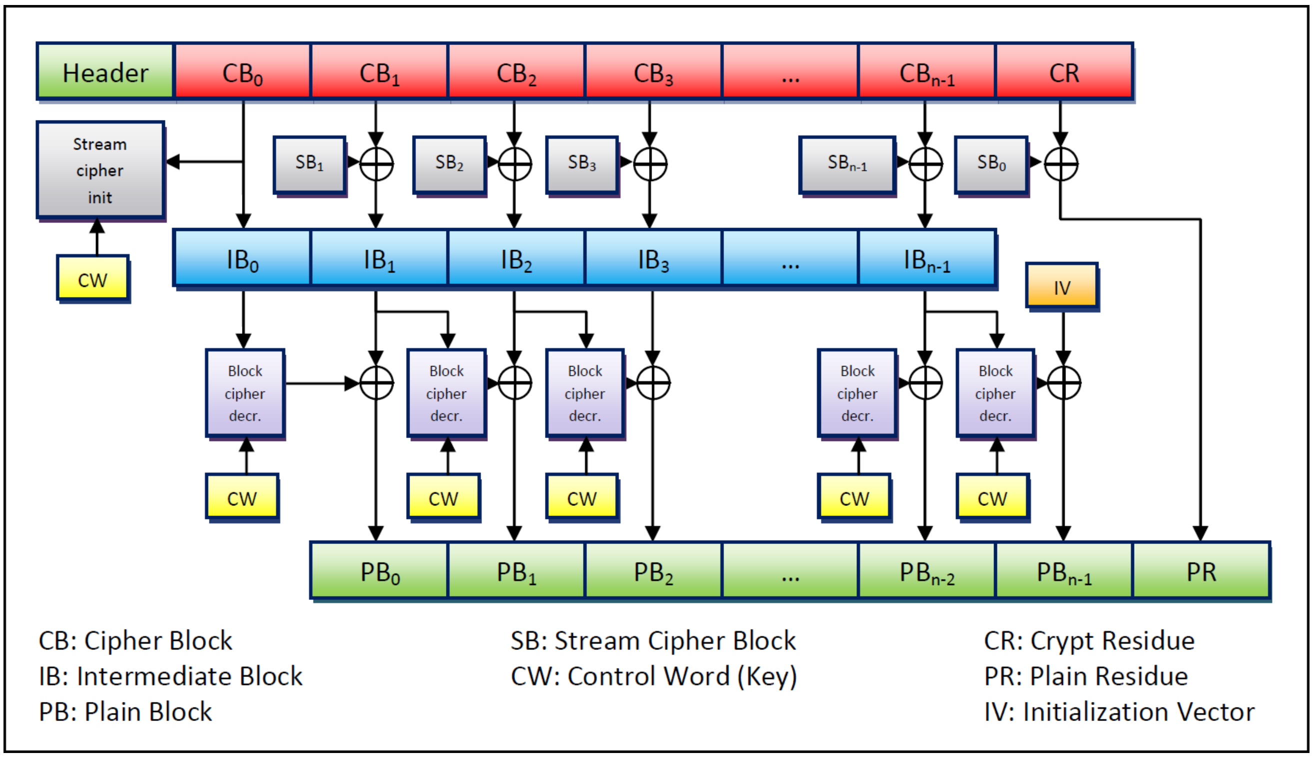


Figure 2.9: DVB-CSA1 decryption path - Colibri[4]

2.2.1 Stream Cipher

The stream cipher has two operation modes. The first mode is the initialization mode, in which all the states of the stream cipher are set up with the help of the common word (CW). The second mode is the generation mode, in which the stream cipher produces 2 bits of cipher every clock cycle.

The core of the stream cipher consists of two Feedback Shift Registers (FSR): A and B. Each FSR has 10 4-bit wide columns, making them 40-bit in size. The stream cipher also consists of 5 4-bit registers (X, Y, Z, E, F) and 3 1-bit registers (c, p, q). Therefore the stream cipher has a state of 103 bits with a theoretical maximum period length of 2^{103}, which is found to actually be much less [1]. There is also another 4-bit variable (D) which is not a register, but is directly calculated by the other registers and is therefore not needed for the calculation of the next stream cipher state with the only exception being during the initialization mode.

2.2.1.1 Initialization Mode

During the initialization mode the state of the stream cipher is setup with the help of the common word (CW) and the first 64-bit cipher block (CB0). A diagram of the stream ciphers initialization mode can be seen in figure 2.10.

For the stream cipher, all Bytes of the CW are nibble swapped. At first the nibble swapped CW is loaded into the two FSR as follows:
Assuming FSR $A = a_{0,j}, ..., a_{9,j}$ and $B = b_{0,j}, ..., b_{9,j}$ with $j \in \{0, ..., 3\}$

$$a_{i,j} = \begin{cases} cw_{(4i)+j} & i \leq 7 \\ 0 & else \end{cases}$$

$$b_{i,j} = \begin{cases} cw_{32+(4i)+j} & i \leq 7 \\ 0 & else \end{cases}$$

So, the first 32-bits of the CW are loaded into the first 8 columns of A, the last 32-bits of the CW are loaded into the first 8 columns of B and the last 2 columns of each FSR are still zero. The rest of the Registers are all initialized with zero (Even D).
From now on, the stream cipher performs 32 clock cycles in initialization mode, using each Byte of the CB0 for 4 clock cycles, starting from the Most Significant Byte (MSB). At first, the high nibble of the currently used Byte is stored in IA and the low nibble in IB, which are used for the calculation of the next leftmost elements of the A and B FSRs. Each clock cycle, the IA and IB are swapped and every 4th clock cycle the next Byte of CB0 is used.

$$(IA, IB) = \begin{cases} (CB0[i] \; div \; 2^4, CB0[i] \; mod \; 2^4) & clk \; mod \; 2 = 0 \\ (CB0[i] \; mod \; 2^4, CB0[i] \; div \; 2^4) & clk \; mod \; 2 = 1 \end{cases}$$

$$\text{with} \quad i = 7 - (clk \; div \; 4), \quad clk \in \{0, ..., 31\}$$

$$\text{and} \quad CB0[7], ..., CB0[0] \quad \text{being the Bytes of the first 64-bit cipher block.}$$

The following equations are executed each cycle to calculate the next state of the stream cipher. In the following the 4-bit columns of FSR A and B will be referenced with $a_0, ..., a_9$ and $b_0, ..., b_9$, respectively.

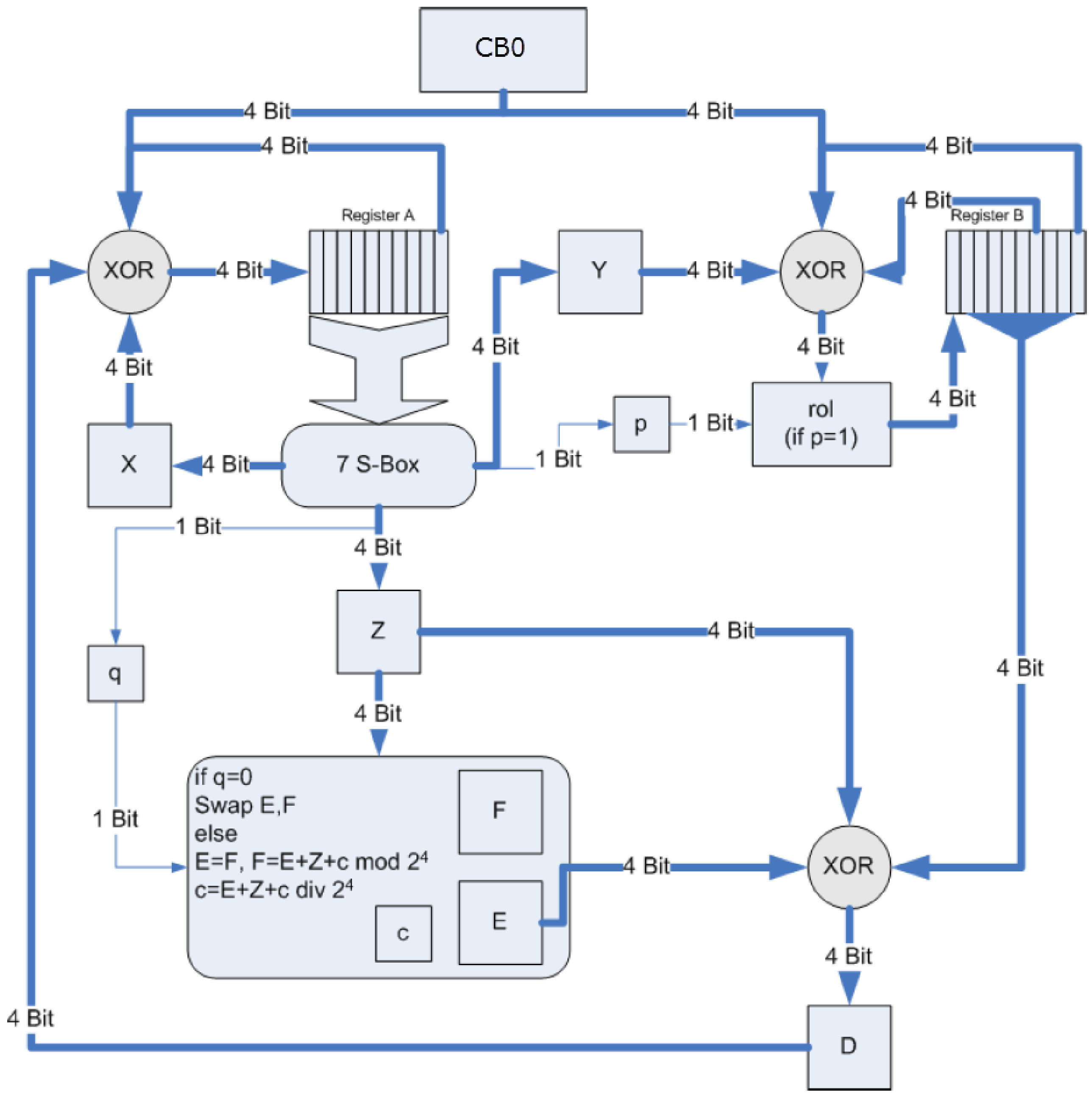


Figure 2.10: Stream Cipher Initialization Mode - Markus Diett[3]

D Register/Variable

$$D' = E \oplus Z \oplus B^{out}$$

$$B_3^{out} = b_{2,0} \oplus b_{5,1} \oplus b_{6,2} \oplus b_{8,3}$$
$$B_2^{out} = b_{5,0} \oplus b_{7,1} \oplus b_{2,3} \oplus b_{3,2}$$
$$B_1^{out} = b_{4,3} \oplus b_{7,2} \oplus b_{3,0} \oplus b_{4,1}$$
$$B_0^{out} = b_{8,2} \oplus b_{5,3} \oplus b_{2,1} \oplus b_{7,0}$$

Feedback Shift Register

$$A' = (a_0', a_0, ..., a8)$$

$$B' = (b_0', b_0, ..., b8)$$

$$a_0' = a_9 \oplus X \oplus D \oplus IA$$

$$b_0' = \begin{cases} b_6 \oplus b_9 \oplus Y \oplus IB & p = 0 \\ rol(b_6 \oplus b_9 \oplus Y \oplus IB) & p = 1 \end{cases}$$

Note that in the calculation of the next FSR A element, D is not the value calculated by the current register states, but that of the previous clock. In others words it is necessary to implement D as a register during the initialization mode, in order to have the D value of the previous cycle. Also note that the very first D value is zero, and not a calculation of the other registers.

Combiner (E,F,c)

$$(E', F') = \begin{cases} (F, E) & q = 0 \\ (F, (E + Z + c)\ mod\ 2^4) & q = 1 \end{cases}$$

$$c' = \begin{cases} c & q = 0 \\ 0 & q = 1\ and\ E + Z + c < 2^4 \\ 0 & q = 1\ and\ E + Z + c \geq 2^4 \end{cases}$$

The E, F and c registers are calculated in relation to q. If q is 1, the previous registers are all summed together. The result of this sum is used for the new F value, E gets the old F value and the carry bit of the calculation (5th bit) is stored in register c.

S-Box (X,Y,Z,p,q) The rest of the registers are put together with the help of 7 5x2 S-Boxes, with each taking 5 bits of FSR A as input and giving 2 bits output. The relation is shown in Table 2.2.

Table 2.2: Input of S-Boxes and new register values

$S-Box$	$bit4$	$bit3$	$bit2$	$bit1$	$bit0$
$S1$	$a_{3,0}$	$a_{0,2}$	$a_{5,1}$	$a_{6,3}$	$a_{8,0}$
$S2$	$a_{1,1}$	$a_{2,2}$	$a_{5,3}$	$a_{6,0}$	$a_{8,1}$
$S3$	$a_{0,3}$	$a_{1,0}$	$a_{4,1}$	$a_{4,3}$	$a_{5,2}$
$S4$	$a_{2,3}$	$a_{0,1}$	$a_{1,3}$	$a_{3,2}$	$a_{7,0}$
$S5$	$a_{4,2}$	$a_{3,3}$	$a_{5,0}$	$a_{7,1}$	$a_{8,2}$
$S6$	$a_{2,1}$	$a_{3,1}$	$a_{4,0}$	$a_{6,2}$	$a_{8,3}$
$S7$	$a_{1,2}$	$a_{2,0}$	$a_{6,1}$	$a_{7,2}$	$a_{7,3}$

$Register$	$bit3$	$bit2$	$bit1$	$bit0$
X	$S4_0$	$S3_0$	$S2_1$	$S1_1$
Y	$S6_0$	$S5_0$	$S4_1$	$S3_1$
Z	$S2_0$	$S1_0$	$S6_1$	$S5_1$
p				$S7_1$
q				$S7_0$

2.2.1.2 Generation Mode

During the generation mode, the stream cipher produces 2 bits of pseudo random stream cipher output every clock cycle. A diagram of the stream ciphers generation mode can be seen in figure 2.11.

All the calculations of the next stream cipher state are the same as in the initialization mode, with the exception of the next FSR element calculation and the calculation of the output itself:

Feedback Shift Register

$$A' = (a'_0, a_0, ..., a8)$$

$$B' = (b'_0, b_0, ..., b8)$$

$$a'_0 = a_9 \oplus X$$

$$b'_0 = \begin{cases} b_6 \oplus b_9 \oplus Y & p = 0 \\ rol(b_6 \oplus b_9 \oplus Y) & p = 1 \end{cases}$$

Stream Cipher Output

$$Out_1 = D'_2 \oplus D'_3$$

$$Out_0 = D'_0 \oplus D'_1$$

As visible in the above equation, during the generation mode only the new value of D (D') is used and therefore the D has not to be implemented as a register, since its value can be calculated with the current values of the other registers.

Also note that the 2-bits of stream cipher output are used in a "downto" order, meaning that the first 2 bits of the first generation mode clock cycle are the two highest bits of the 64-bit stream cipher output (63rd and 62nd bit).

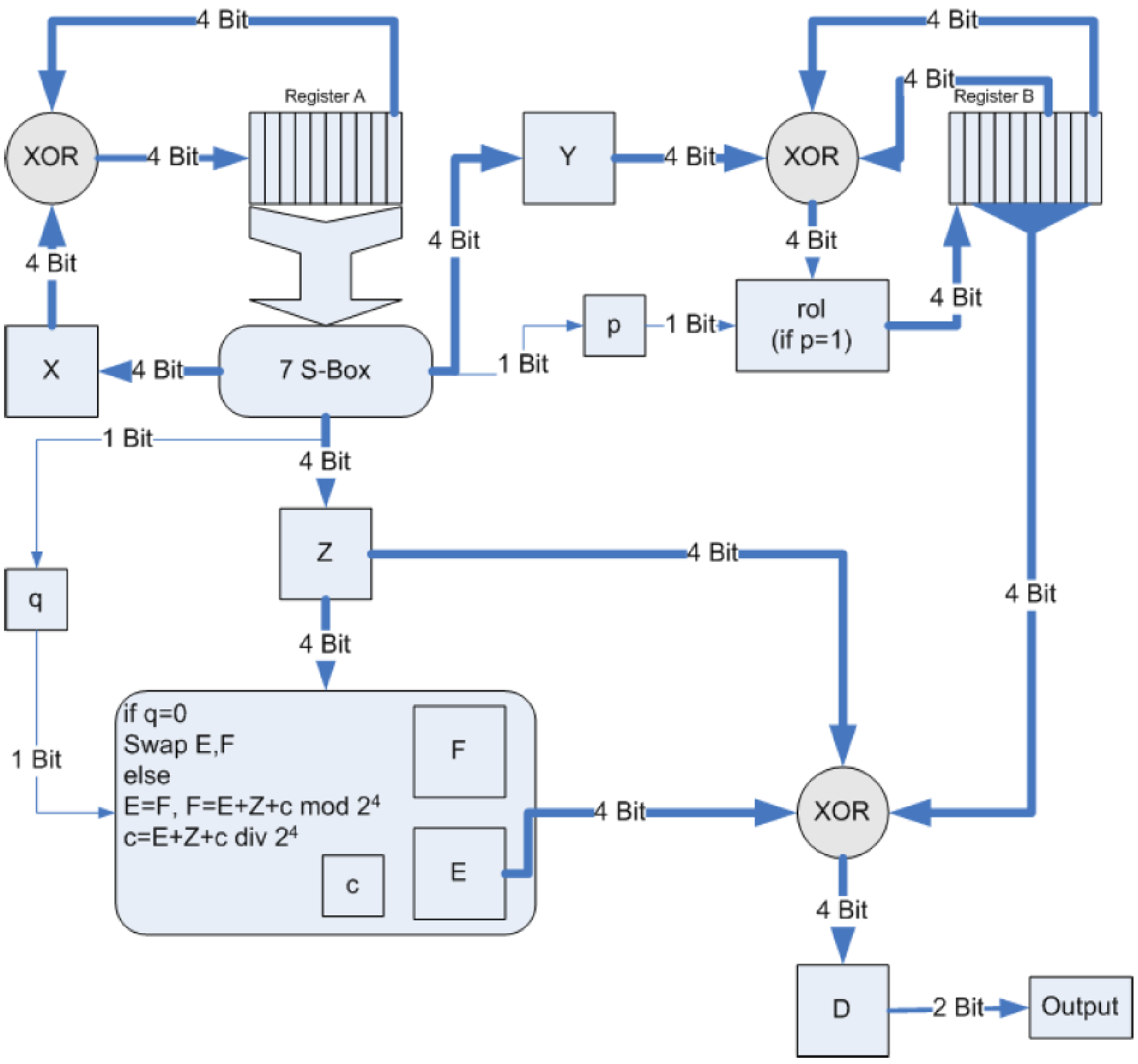


Figure 2.11: Stream Cipher Generation Mode - Markus Diett[3]

2.2.2 Block Cipher

The block cipher used by the CSA operates with 64 bits of input, output and key. The core of the block cipher consists of a round function that is applied 56 times and operates Byte-wise on the 64-bit Input data, using 1 Byte of the expanded 448-bit key.

2.2.2.1 Block Cipher Key Expansion

Since the round function of the block cipher uses 1 Byte of the expanded key each time, and the round function is applied 56 times, we need an expanded key with a length of 448 bits.
The expansion of the 64-bit CW into the expanded 448-bit key takes place in two steps:

Step 1 In the first step of the expansion, a 64-bit permutation is recursively used on the 64-bit CW in order to get 448-bit of expanded bits.
The permutation used is described in Table 2.3.

Table 2.3: Bit Permutation of Key Expansion in Block Cipher

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

Assuming $K^{E'}_{447,...,0}$ are the expanded bits of the first step and p(i) is the 64-bit permutation as shown in Table 2.3, the bits have following relation:

$$K^{E'}_{447,...,384} = CW_{63,...,0}$$

$$K^{E'}_{64(i-1)+63,...,64(i-1)} = p(K^{E'}_{64i+63,...,64i}) \quad \text{with } i \in \{6, ..., 1\}$$

Step 2 The second step takes the 7 64-bit expanded key blocks of step 1 and XORs them with 7 predefined constants:

$$K^{E}_{64i+63,...,64i} = K^{E'}_{64i+63,...,64i} \oplus 0\text{x}0i0i0i0i0i0i0i0i \quad \text{with } i \in \{0, ..., 6\}$$

2.2.2.2 S-Box

The main feature of the block cipher, is its 8x8 S-Box with 256 entries, that is used in the round function of the block cipher. The S-Box is described in Table 2.4.

Table 2.4: Block Cipher S-Box.

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

2.2.2.3 Round Function

The round function is applied 56 times on the 64-bit input, using 1 Byte of the expanded key for each iteration. The round function uses the 8x8 S-Box described in Table 2.4 and a 8-bit permutation described in Table 2.5. There are two kinds of round functions used, depending on if the block cipher performs a decryption or an encryption.

Table 2.5: Block Cipher Round Permutation

i	0	1	2	3	4	5	6	7
p(i)	1	7	5	4	2	6	0	3

In the following $d_7, ..., d_0$ and $d'_7, ..., d'_0$ describe the 8 input and output Bytes of the round function respectively, k describes the currently used expanded key Byte, p(i) describes the block cipher round permutation and SBox(i) the S-Box Byte substitution.

Encryption

$$k = K^E_{8(r-1)+7,...,8(r-1)} \quad where\ r\ is\ the\ current\ round\ iteration\ , r \in \{1, ..., 56\}$$

$$x = SBox(k \oplus d_0)$$

$$y = p(x)$$

$$\begin{aligned} d'_7 &= d_6 \\ d'_6 &= d_5 \oplus d_7 \\ d'_5 &= d_4 \oplus d_7 \\ d'_4 &= d_3 \oplus d_7 \\ d'_3 &= d_2 \\ d'_2 &= d_1 \oplus y \\ d'_1 &= d_0 \\ d'_0 &= d_7 \oplus x \end{aligned}$$

The first iteration of the round function takes the 64-bit plaintext as input, the second iteration of the round function takes the output of the first round iteration and so forth. The output of the 56th round iteration is the block cipher output.

Decryption

$$k = K^E_{447-8(r-1),...,440-8(r-1)} \quad where\,r\,is\,the\,current\,round\,iteration\;, r \in \{1, ..., 56\}$$

$$x = SBox(k \oplus d_1)$$

$$y = p(x)$$

$$\begin{aligned} d'_7 &= d_0 \oplus x \\ d'_6 &= d_7 \\ d'_5 &= d_0 \oplus d_6 \oplus x \\ d'_4 &= d_0 \oplus d_5 \oplus x \\ d'_3 &= d_0 \oplus d_4 \oplus x \\ d'_2 &= d_3 \\ d'_1 &= d_2 \oplus y \\ d'_0 &= d_1 \end{aligned}$$

The first iteration of the round function takes the 64-bit ciphertext as input, the second iteration of the round function takes the output of the first round iteration and so forth. The output of the 56th round iteration is the block cipher output. Note that the expanded key is used in inverted order for the decryption.

2.2.3 Software DVB-CSA Implementations

The two most widely known software implementations of the DVB-CSA are libdvbcsa [9] and FFdecsa [7]. FFdecsa was considered the fastest software implementation of the Common Scrambling Algorithm, but development stopped and libdvbcsa - that is actively developed by the VideoLAN Organization [10] - soon surpassed it in portability, reliability and overall performance. Both implementations are released under the General Public License (GPL).

CHAPTER 3

CSA Attacks

There exist several attacks on different parts of the DVB-Common Scrambling Algorithm. It could either be an attack on the fundamental components of the CSA itself (stream cipher and block cipher) using a cryptographic weakness, or an straight forward brute force attack using a weakness on the overall layout of the CSA. This Chapter informs what attacks against CSA are known until today.

3.1 Cryptanalytic Attacks

In their paper "Analysis of the DVB Common Scrambling Algorithm", Ralf-Philipp Weinmann and Kai Wirt [1] analyze the stream cipher and block cipher of the Common Scrambling Algorithm, report their weaknesses and give some attack vectors. These attacks are theoretical crypt-analytical attacks on the CSA and cannot be directly applied to find the common word of a scrambled DVB transport stream.

3.2 Time Memory Trade-Off Attack

The TMTO attack is a known plaintext attack and uses the fact that the first encrypted Scrambled block bypasses the stream cipher entirely and is only dependent on the block cipher, as seen in figure 3.1. The encryption path used by this attack goes through 23 cascaded block ciphers that map the first 184 Bytes of plaintext into the first 8 Bytes of the ciphertext.

The TMTO attack procedure is to find a known plaintext - in our case a known DVB transport stream payload - and perform the encryption operation for the whole 2^{48} key space, storing the calculated ciphertext block with its corresponding key into a look up table (also known as a Rainbow table). This necessary precalculation takes time and has large memory needs in order to store the lookup table, but once completed a simple lookup of the ciphertext with the ciphertexts stored in the table would give us the key, if the original plaintext of the ciphertext we try to decrypt is the same as the one used in the precalculation.

The most challenging aspect of this attack is to find a known plaintext in the DVB transport stream. Because of the compressed nature of the MPEG Codec, there are no repeating Bytes in the MPEG stream itself. The only potential source for repeating Bytes would be due to stuffing procedures needed to fit a PES packet into the fixed payload of the transport stream. But as stated in Diett's paper "On the Security of Digital Video Broadcast Encryption" [3], that is also not reliable, because the PES data that is being stuffed into the payload of the transport stream is unknown, and there exist different methods to perform the stuffing. Nevertheless, in their

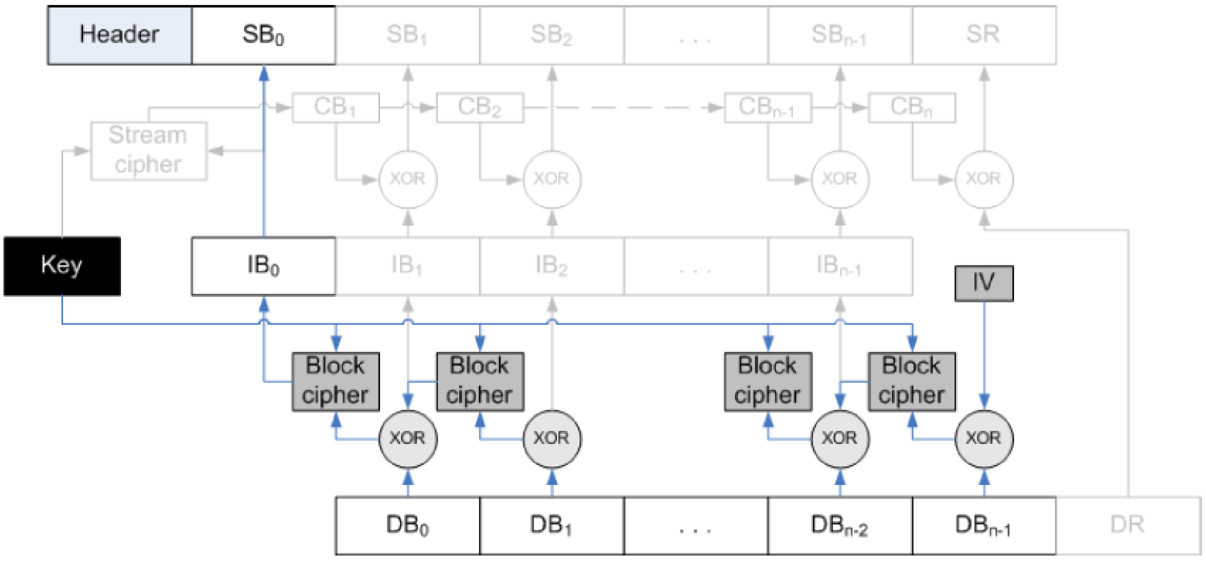


Figure 3.1: Known Ciphertext Attack Overview - Markus Diett[3]

paper "Breaking DVB-CSA" Tews et al. [2] state that they succeeded in finding zero filled TS packets on the Astra 19.2 satellite, and statistically documented their occurrences. It is also stated that the occurrence of zero filled TS packets is heavily dependent on the actual video content.

It also has to be noted that some providers use the unconventional encryption method in which they encapsulate the whole transport stream content including management PIDs (like PAT,CAT,TSDT) into a single PID and then scramble this PID. Because transport stream packets are encapsulated into other transport stream packets, which means that 188 Byte packets are split into 184 Byte payloads, stuffing and alignment is necessary. As stated in the research from Colibri [4], this can lead to repeating payloads not only due to the stuffing method used, but also because management PIDs like PAT (Program Association Table) can contain static information.

3.2.1 Existing Implementations

In the thesis of Markus Diett [3], an attempt was made to implement and perform a TMTO attack on DVB CSA. After failing to find a known plaintext on recordings of various transport streams, the TMTO attack was performed in a controlled test scenario to measure the theoretical resistance of CSA against TMTO attacks.

In the paper "Breaking DVB-CSA" of Erik Tews et al., a successful TMTO attack is done on DVB-CSA. In contrast to the work of Markus Diett [3] they succeeded in finding the known plaintexts necessary for the attack. Since the entropy of the key is only 2^{48}, the entropy of the 64-bit block cipher output also has to have an entropy of 2^{48}. Due to this fact a reduction function mapping the 64-bit cipher block output to a 48-bit value can be used. This optimization was used to further decrease the space requirements for the rainbow table. The paper also states additional optimizations to the implemented TMTO attack.

Colibri [4] also performed a TMTO attack on a transport stream of a Spanish provider. He succeeded in finding a known plaintext in the transport stream, due to the fact that the provider used the unconventional encapsulation scrambling method explained above. He could consistently detect the relevant packet in a live transport stream, because the 4 Bytes residue that bypass the block cipher are only XORed with the stream cipher output and can be used as bit comparison. Several rainbow tables for various plaintexts can be downloaded from his site [4].

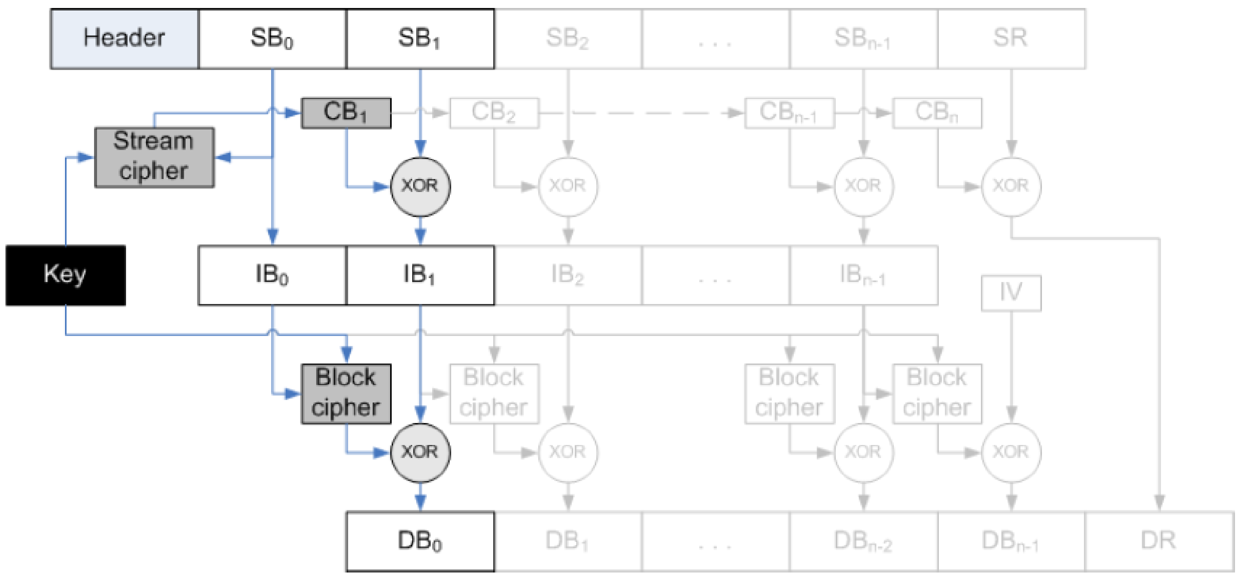


Figure 3.2: Known Plaintext Attack Overview - Markus Diett[3]

3.3 Brute-force Attack

The brute-force attack is also a known plaintext attack that uses the fact that the first 8 Bytes of the decrypted plaintext are only dependent on the first 16 Bytes of ciphertext, as seen in figure 3.2. In the decryption path of the first plaintext block, only one block cipher instance and the first 64 bits of the stream cipher output are needed. That drastically reduces the complexity and resources needed to perform a brute-force attack.

As in the TMTO attack, it is necessary to find a known plaintext, but in the case of a brute-force attack this condition can be weakened a bit. If we did find a known 128 bit plaintext, a successful comparison with the generated plaintext would yield the wanted common word. If on the other hand we compare less than 64 bits, we would get many false positives (key candidates) on the generated plaintexts. But if we iteratively compare these key candidates with plaintexts generated from new ciphertexts (that are encrypted with the same key), we would eventually be left with only one key, which would be the wanted common word.

Because of the constant Header prefix of the PES packet (0x000001) contained in every transport stream that has the PUSI bit set, we effectively have a 3 Byte known plaintext that can be consistently found in a live DVB transport stream. Also combined with the fact that more than one PES packets are encrypted with the same common word before the common word is updated, we also have enough ciphertext samples to find the wanted common word amongst the key candidates.

3.3.1 Existing Implementations

Markus Diett [3] implemented a brute-force approach in software as well as in hardware. The software implementation could test approximately 784000 keys per second, which could iterate the entire 2^{48} key space in approximately 4155 days. Note that these results are of course hardware specific, but unfortunately Diett did not elaborate on the testing system specifications. The hardware implementation on the Copacobana [6] could operate in two different frequencies (96 MHz and 108 MHz) with each of the 120 FPGAs testing a key in 4 clock cycles. This brings the rate of the hardware implementation to $2.88 * 10^9$ with 96 MHz and $3.24 * 10^9$ with 108

MHz. The 2^{48} key space could be searched in approximately 27.15 hours with 96 MHz and 24.13 hours with 108 MHz.

There also exists another software based brute-force implementation called AYCWABTU [8] that is also based on the bit splice method as in the implementation of Diett [3]. It was implemented as a proof of concept that a brute-force attack on the DVB CSA is indeed possible. On a computer equipped with an Intel i5 3337U (1.80-2.70 GHz) running on Debian 8, a rate of approximately 4000000 keys per second was achieved. The entire 2^{48} key space could be iterated in approximately 814 days.

CHAPTER 4

Design & Implementation

The Common Scrambling Algorithm is hardware oriented, since it consists only of XOR operations, Shift operations, and substitutions, which can all be implemented efficiently in hardware. For this reason, not much optimization or reimplementation is needed when implementing CSA on FPGAs.

As a first step, the block and stream cipher were implemented in hardware. This gave a first approximation on the resource utilisation on a FPGA. In the second step, the attack vector was decided upon, going by the existing attack vectors known today and presented in Chapter 3. The Brute Force approach was selected, because of it only requiring one instance of the block and stream cipher, therefore being very resource friendly. It does also not require extended communication between FPGAs, which is perfect for our targeted FPGA cluster.

In order to make a modular design, it was decided to implement a brute-force core, with as minimum resources as possible. A brute-force core is capable of decrypting a given ciphertext with all keys from a given key space and outputting the resulting plaintexts. The brute-force core has to also operate in the highest operating frequency achievable, so that the assigned key space is iterated as fast as possible. In order to achieve that, it was decided on a fully pipelined design, with as little logic between pipeline stages as possible. A pipelined design means more resources than an iterative/looped approach, but the achieved throughput of a pipelined approach far outweighs the cost of the additional resources needed.

Once the working brute-force core is implemented with the required constraints, the FPGA is fitted with as many of these cores as possible, with additional overlying logic for managing and communicating with the cores.

The full implementation of this thesis is targeted for ZTEX boards containing a Cypress FX2 Microcontroller for USB communication. That means that the implementation contains logic and configurations not directly associated with the brute-force procedure itself, but with the communication and management of the whole FPGA cluster. Nevertheless, the actual brute-force cores are written modularly and can therefore be fitted on any FPGA with minimal effort, assuming the FPGA has enough available resources.

The implementation is divided into the hardware description (VHDL) of the FPGA design, describing the actual brute-force cores, the firmware uploaded on the Cypress FX2 Microcontroller, responsible for the communication between host and FPGA, and a Java program running on the host computer, responsible for managing the entire FPGA cluster.

Two implementations were performed. One for the ZTEX USB-FPGA Module 1.15y, containing 4 Xilinx Spartan 6 (XC6SLX150-3CSG484) FPGAs with a Cypress EZ-USB FX2 Microcontroller for USB

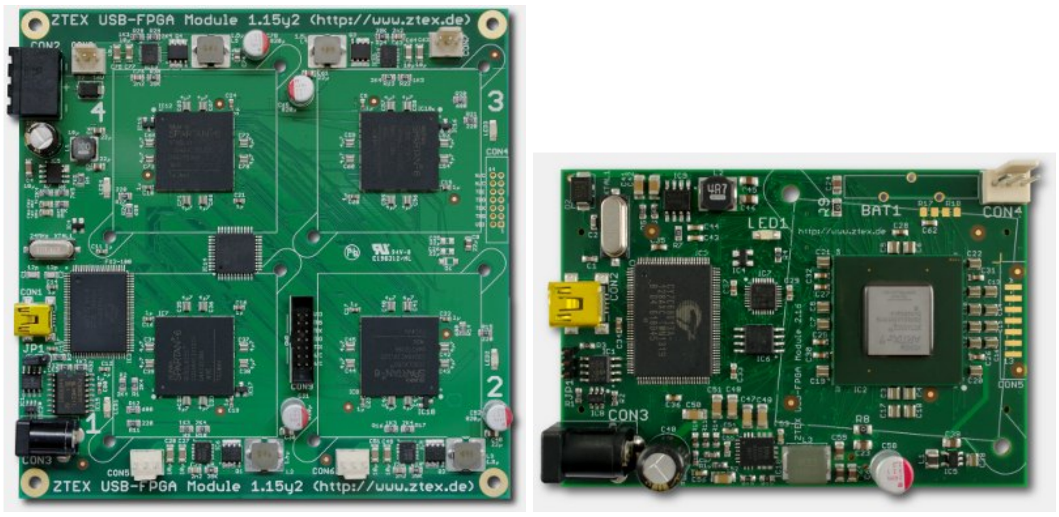

ZTEX USB-FPGA Module 1.15y2 (http://www.ztex.de)
ZTEX USB-FPGA Module 1.15y2 (http://www.ztex.de)

Figure 4.1: ZTEX 1.15y (left), ZTEX 2.16 (right)

2.0 communication (figure 4.1), and one for the ZTEX USB-FPGA Module 2.16, containing a single Artix 7 (XC7A200TFBG484-2) with a Cypress EZ-USB FX2 Microcontroller for USB 2.0 communication (figure 4.1).

A high level overview of the FPGA cluster (with the ZTEX 1.15y Boards) can be seen in figure 4.2.

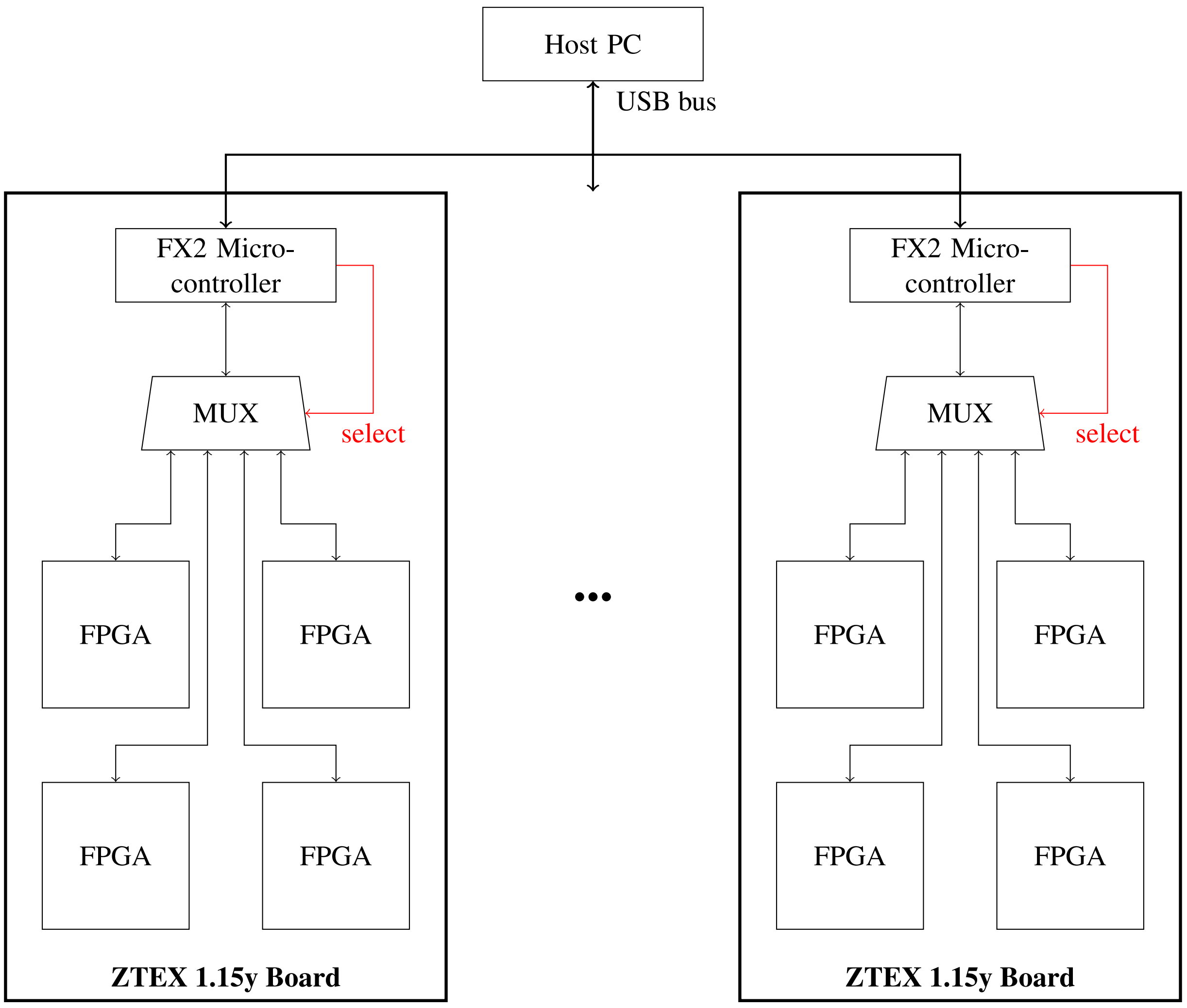


Figure 4.2: ZTEX 1.15y Board Cluster

4.1 HOST Software

The host software is written in Java, since the ZTEX SDK for configuring and communicating with the ZTEX boards has only full support for Java.

The software is programmed to read from a transport stream file of any size and brute-force all the keys with which the transport stream has been encrypted. The found keys are written out on a file in the order in which they are used in the transport stream. If a key is not found, the software automatically reduces the operating frequency of the brute-force cores and retries the brute-force procedure. If the key is not found even when in the lowest frequency setting, an error is logged and the software terminates.

The communication between software and FPGA has no direct impact on the performance and speed of the brute-force procedure, as the brute-force cores do not require any communication during their operation. The required data is given to the brute-force cores before they start their operation. Nevertheless, the communication has an indirect impact on the whole brute-force procedure, as the key is considered found the moment the software knows that the key has been found, and not the moment a brute-force core finds it. For this reason, the software polls every ZTEX board at an interval of 500 ms.

The main classes of the software are:

- **Main**
  The Main class is responsible for the parsing of the arguments, scanning of the USB communication bus for ZTEX devices, and starting and monitoring of the BoardControlThreads. It also registers a shutdown hook to allow a clean exit of the program even when an interrupt is sent.

- **BoardControlThread**
  Each BoardControlThread is responsible for the communication with a single board and all of its contained FPGAs. It is reading and writing directly to the USB endpoints. The read status bits of each FPGA are monitored and handled accordingly. The class provides a restart variable, allowing the threads to be restarted without the necessity to recreate the threads.

- **TranportStreamParser**
  The TranportStreamParser class is responsible for the parsing of the given transport stream file. The used keys inside the stream are differentiated with the help of the Scrambling Control Field (see Section 2.1.1.1) of the transport stream Header.

4.2 FX2 Microcontroller Firmware

As seen in figure 4.2, the FX2 Microcontroller is responsible for the communication with multiple FPGAs. This is done by having all FPGAs connected to the same bus, and only allowing the FPGA selected by the Microcontroller to drive the bus signals. This effectively means that the FPGAs cannot communicate with each other, and that only one FPGA per board can communicate with the software at any time. The latter is not a problem, because a communication between software and FPGA is only needed during the initialization of the brute-force procedure and later during the periodic polling of the brute-force results.

Even though the Cypress FX2 Microcontroller provides a vast amount of options, it was decided to remove the Microcontroller CPU from the communication path between software and FPGA. To accomplish that, the FX2 Microcontroller is put into "Slave FIFO" mode. In this mode, the USB endpoint FIFOs of the FX2 Microcontroller are directly interfaced by an external master (the FPGAs in our case).

Three USB endpoints are used for the communication between software and FPGA. endpoint 2 is used for sending the initialization data required by the brute-force cores. endpoint 4 is used for sending the frequency parameters required for the dynamic frequency reconfiguration. endpoint 6 is used by the FPGA for sending the status bits and the key to the software.

Vendor Commands and Requests were configured to reset, suspend and resume the FPGA, as well as reset the frequency generators (DCM/MMCM). An extra Vendor Request was added for providing Debug information of the endpoints and FIFOs.

The Cypress FX2 Microcontroller is also used to configure the FPGAs, by uploading the compiled Bitstreamfiles onto them.

4.3 FPGA Design

The FPGA design is the heart of the implementation. The Cypress firmware and host program are only there for configuring, synchronizing and managing the FPGAs, and providing a user-friendly front end.

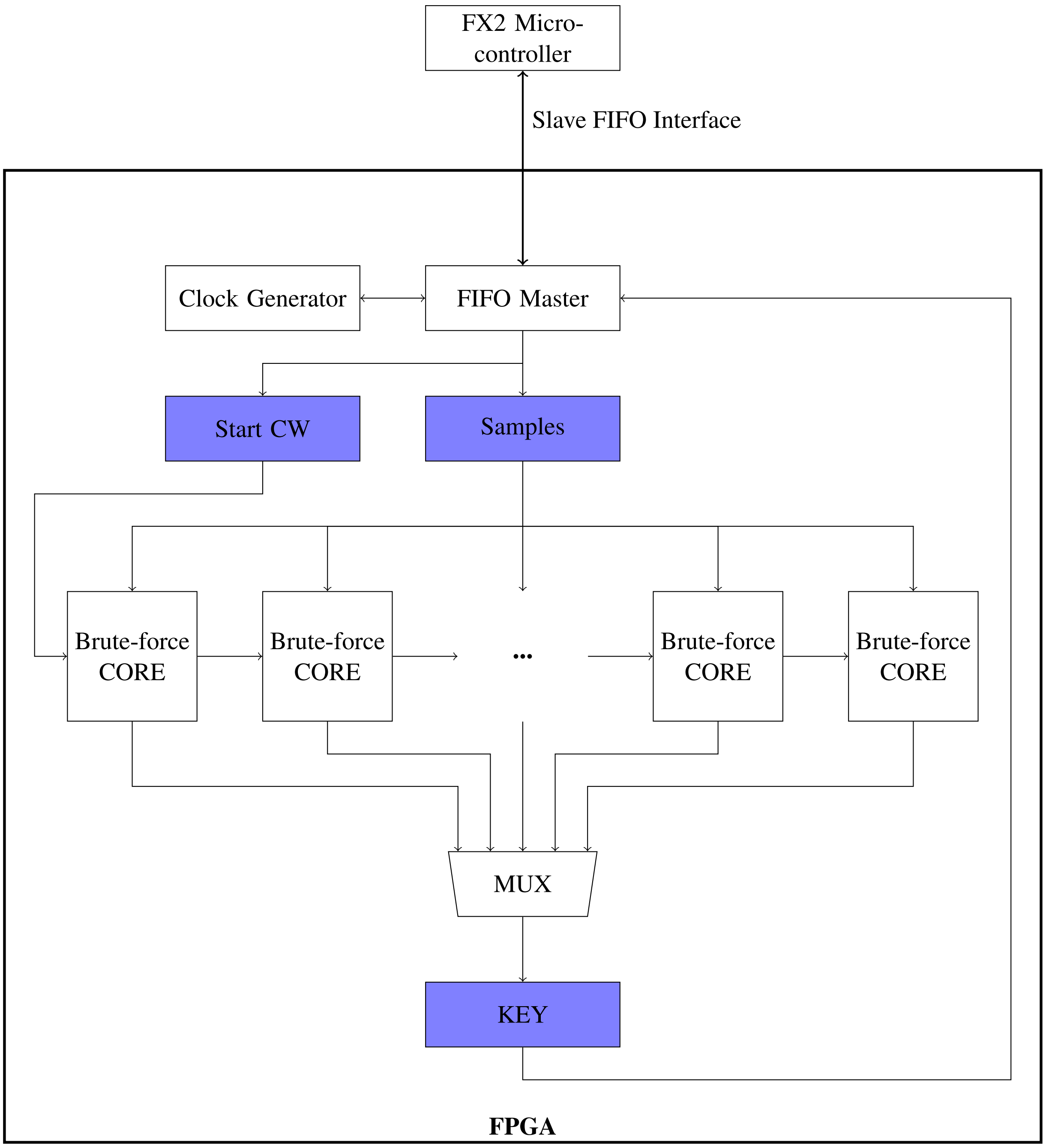


Figure 4.3: Overview of the FPGA

The design was done bottom-up, starting with the implementation of the block and stream ciphers of the CSA. The whole design is divided into several Entities from which the primary ones are:

- block_cipher
- stream_cipher
- decode_core_unit
- core_control_unit

- fpga_top
- top

As stated in the intro of this section, the logic levels between pipeline stages were kept small, in order to achieve a high operating frequency, and thus a higher key brute-force rate. For the stream cipher all operations performed in a single clock cycle are considered a pipeline stage. For the block cipher all operations performed in a single round are also considered a pipeline stage. All other operations were divided into steps, with each step containing at most one operation on each signal. Each step is also considered a pipeline stage. Since the design was done bottom-up, every time a super-entity was completed, it was compiled and analyzed if there is need to further divide a pipeline stage.

A high level overview of the FPGA structure can be seen in figure 4.3.

4.3.1 block_cipher

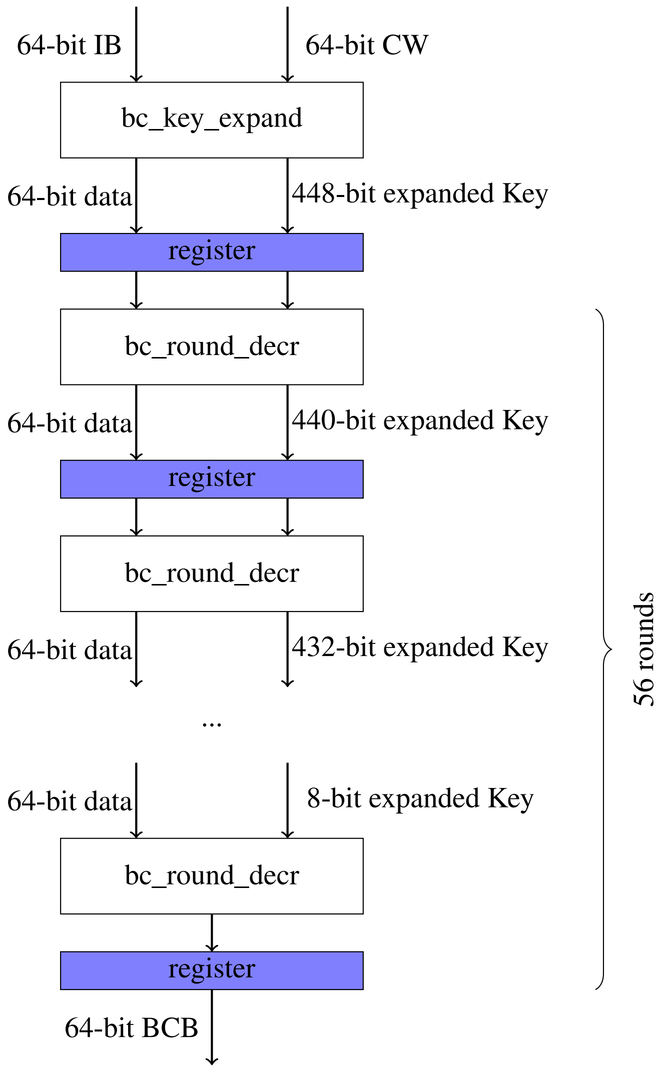


Figure 4.4: Internal Structure of block_cipher

The block cipher round function, consisting only of XORs and S-Box look-up operations, was implemented as a single pipeline stage, instantiated 56 consecutive times. The S-Box was implemented as a 256x8-bit ROM, but because every round function uses a S-Box concurrently, it needs to be instantiated 56 times. The key expansion function, expanding the common word from 64 to 448 bits, is also implemented in a single pipeline stage. Thus the block cipher consists of a total 57 pipeline stages.

Since every round "consumes" 8 bit of the 448 bit expanded key, there is no need to pass the whole 448-bit expanded key through all the pipeline stages. Instead only the still relevant bits of the expanded key are passed down to the next pipeline stage. This was done in VHDL via generics. Every round stage is aware of its position and connects only the remaining bits of the key to the next stage, effectively making a triangle structure. The synthesizer recognizes the unconnected signals and trims them out. This optimization cuts the needed register resources almost by half.

This entity also contains the longest logic path of the whole implementation. The path lies in the round function, which does an XOR operation, followed by S-Box lookup, followed by two additional XOR Operations (see Section 2.2.2.3 - Decryption).

An overview of the block_cipher structure can be seen in figure 4.4.

4.3.2 stream_cipher

The stream cipher needs 32 clock cycles in initialization mode and 32 clock cycles in generation mode in order to generate the required 64 bit cipher output. Every clock cycle of the stream cipher was implemented as a single pipeline stage. Every pipeline stage calculates and passes the new internal state of the stream cipher to the next pipeline stage. Thus the stream cipher consists of a total 64 pipeline stages.

A similar triangular connection scheme to the one used in the block cipher was used on the Scrambled block - used during the initialization mode (first 32 pipeline stages) - and the output key signal - generated during the generation mode (last 32 pipeline stages) - in order to save on register resources.

An overview of the stream_cipher structure can be seen in figure 4.5.

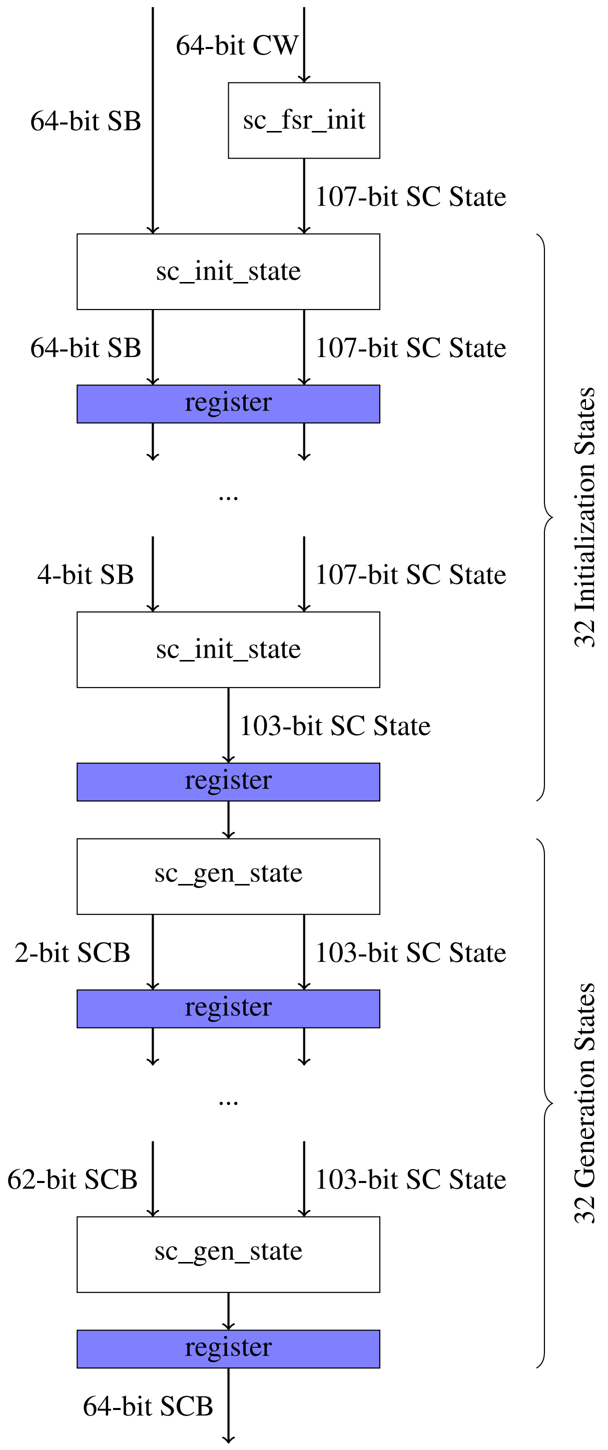


Figure 4.5: Internal Structure of stream_cipher

4.3.3 decode_core_unit

The decode_core_unit is the wrapper instantiating the block and stream cipher, and performing the final XOR operations in order to produce the first 64-bit plaintext block. It is also responsible for the initial expansion of the common word from 48 to 64 bits. The decode_core_unit, completely implementing the decryption of the first plaintext block, needs a total of 67 pipeline stages.

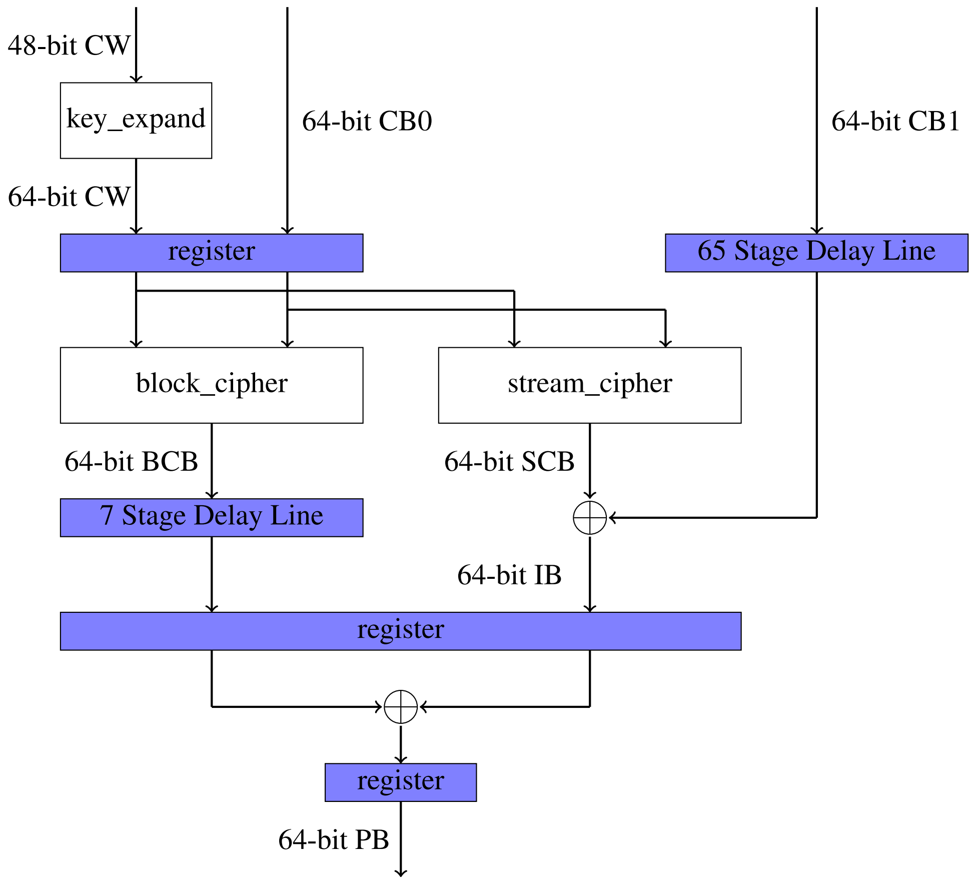


Figure 4.6: Internal Structure of decode_core_unit

An overview of the decode_core_unit structure can be seen in figure 4.6.

4.3.4 core_control_unit

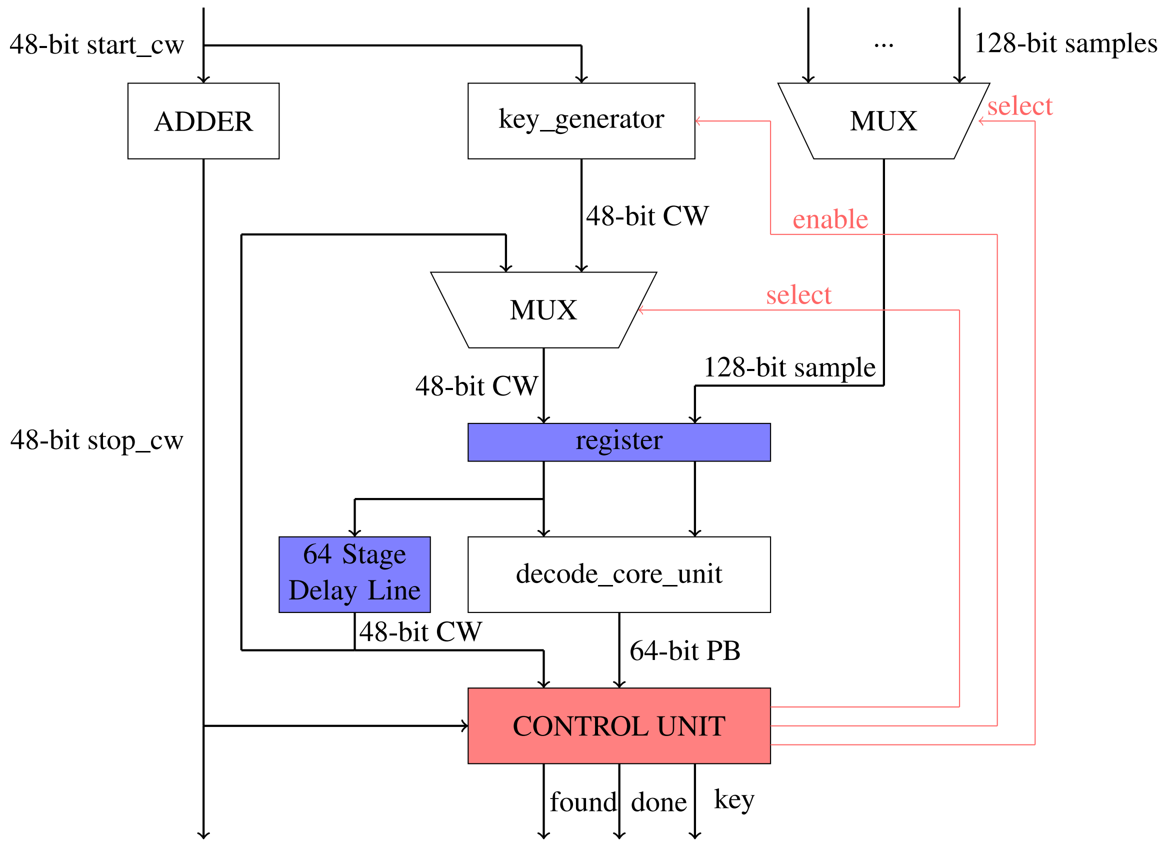


Figure 4.7: Internal Structure of core_control_unit

The core_control_unit describes a single brute-force core. The core_control_unit instantiates the decode_core_-unit and provides a key generator implemented as a counter for feeding the decode_core_unit keys for the decryption. The 64-bit output of the decode_core_unit is compared with the 3 Byte known plaintext (0x000001), and on match the same key is re-fed into the decode_core_unit with a new set of ciphertext samples. This procedure is necessary due to the high number of false positives as stated in Section 3.3. A pipelined delay line tracks which pipeline stage has which key and which samples. When a key successfully decodes all provided ciphertext samples, the found and done flag are set, the key is written to the output port, and the key generator is stalled, until a reset occurs. If on the other hand the end of the key iteration space is reached without finding the key, the done flag is set and the key generator is stalled, until a reset occurs.

Note that because a key needs to go through the whole decode_core_unit pipeline sample times before the found flag is asserted, it could happen that the end of the iterated space is reached before that, meaning that the done flag can be set before the found flag is set. In a worst case scenario (last key of the key generator is the valid key) the found flag is set $68 * (N - 1)$ clock cycles after the done flag is asserted (where N is the number of samples). This special case has to be handled by the overlying logic/super-entity.

The goal is to fit as many of these brute-force cores (core_control_unit) onto the FPGA as possible. In order to keep communication between cores and top entity to a minimum, further increasing the potential operation frequency and numbers of fit cores, it was decided that every core already knows how many keys it needs to iterate. The core merely needs to be informed from which key it needs to start iterating (start_cw).

An overview of the core_control_unit structure can be seen in figure 4.7.

4.3.5 fpga_top

The fpga_top entity is a wrapper instantiating all the brute-force cores (core_control_unit) and combining their outputs. Each core has a start key input - the key from which it starts iterating - and a stop key output - the key until which it will iterate. The stop key is calculated by the core internally via a simple addition. The fpga_top entity connects the stop key output from one core to the start key input of the next core. The first core gets the FPGA-wide start key directly from the top entity.

An overview of the core connections can be seen in figure 4.3.

Consequently, because it takes time until every core has its start and stop keys, it has to be made sure that the initial start key is stable long enough before starting the brute-force cores. In order to avoid the need to count internally the clock cycles necessary before enabling the key generators of the core_control_units, the decision was made to handle this externally by resetting the cores after having a stable start key signal for a sufficient time period. For this reason, the start and stop key signals of the cores are not affected by the reset signal. This also allows the timing constraints of the connections between top entity and brute-force cores, as well as between the brute-force cores themselves to be relaxed, allowing a higher operation frequency and performance.

4.3.6 top

The top entity is the communication bridge between the software and the brute-force cores. It is responsible for the instantiation of all subcomponents and the direct communication with the USB endpoint FIFOs, provided by the Cypress FX2 Microcontroller via the Slave FIFO Interface.

The top entity also instantiates the clock_gen component, containing the DCM (Digital Clock Manager) or MMCM (Mixed-Mode Clock Manager) responsible for the generation of the operating frequency for the brute-force cores. The clock_gen component also contains a state machine, which is used to dynamically set the frequency parameters of the DCM during operation.

A state machine with 17 states is used to handle the Slave FIFO Interface of the Cypress FX2 Microcontroller and is operated with a reference clock provided by the Cypress FX2 Microcontroller (30 MHz). This also provides a fail-safe measure in case the DCM fails to generate the operating frequency, since the communication with the software will still be possible. A counter is used to detect transmission errors, if the data transmitted is of incorrect length.

An overview of top can be seen in figure 4.3.

Even though the data to and from the software has to cross the two clock domains, it is not necessary to synchronize the data. The done, found and key signals of the brute-force cores, which are read out by the software, have a initial reset value of zero and once set, keep their values until a reset occurs. From a metastable point of view, if a bit is read out during a transition, the read level can be either zero or one. Since the only relevant transition in our case is from zero to one - when the brute-force cores are finished - the correct operation on either read out value is guaranteed. The status bits that are coming from the DCM, are synchronized in a two stage FlipFlop Synchronizer. On the other hand, the data coming from the software is also not synchronized, because the software is first sending the data and then resetting the brute-force cores. Because the registers holding the data are not affected by the reset, the brute-force cores have stable and constant signals during their operation.

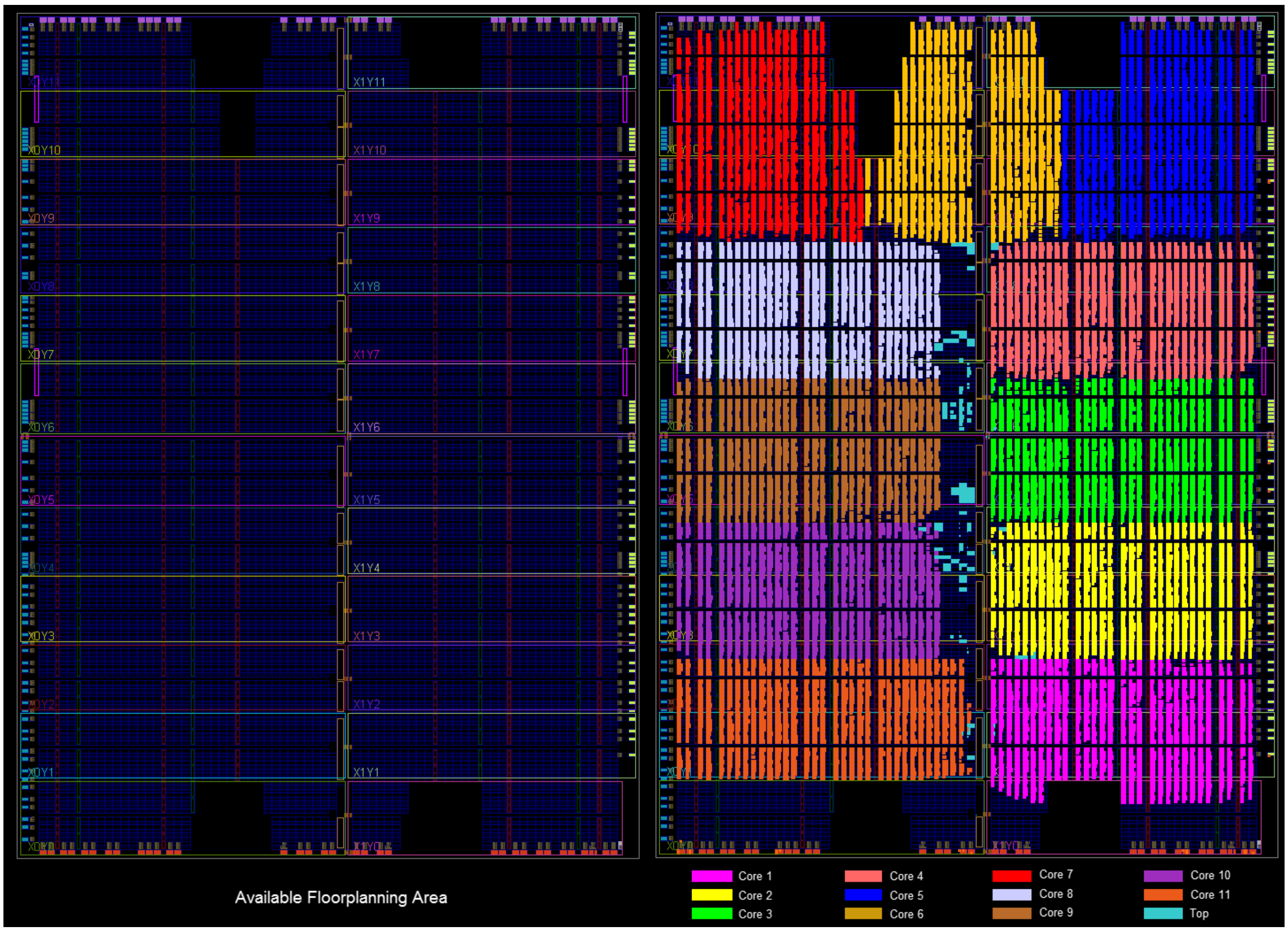


Figure 4.8: Spartan 6 (XC6SLX150) Floorplanning

4.3.7 Final Design

In this section the final designs for both FPGA types are presented.

4.3.7.1 Spartan 6

In order to fit as many brute-force cores (core_control_unit) into the FPGA as possible and still have a high operating frequency, floorplanning was necessary.

During floorplanning, positions of certain primitives (components that are native to the FPGA) are locked or restricted to specific areas of the FPGA. This gives the PAR (Place And Route) utility a hint as to the underlying structure of the routed design, and allows it to achieve shorter interconnect routes.

As shown in figure 4.8, the available floorplanning area was divided in 11 blocks, into which the primitives of each core were restricted. It was made sure to avoid unroutable areas inside each core, to avoid possible high routing delays.

The final design has 11 brute-force cores and can operate with a frequency of 150 MHz. The resource utilization on the Spartan 6 LX150 is shown in Table 4.1.

Table 4.1: Resource Utilization of Spartan 6 LX150

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	88346	184304	47%
Number used as Flip Flops	88346		
Number of occupied Slices	20521	23038	89%
Number of Slice LUTs	72174	92152	78%
Number used as logic	51954		56%
Number used as Memory	11762		13%
Number used as Shift Register	11762		
Number used exclusively as route-thrus	8458		9%
Number of MUXCYs used	1468	46076	3%
Number of bonded IOBs	26	338	7%
Number of BUFG/BUFGMUXs	2	16	12%
Number used as BUFGs	1		6%
Number used as BUFGMUX	1		6%
Number of DCM_CLKGENs	1	12	8%
Number of ILOGIC2s	2	586	1%
Average Fanout of Non-Clock Nets	3.04		

4.3.7.2 Artix 7

While the Spartan 6 FPGAs need the outdated Xilinx ISE Utility to compile, synthesize and generate the Bitstreamfiles, Artix 7 FPGAs require the newer Xilinx Vivado Utility. Vivado has far better runtimes than ISE and can also handle large complex design better and more efficient. Tests on our design showed that Floorplanning had an actual negative performance impact, and therefore the placement decision is left out for the tool.

As stated in the intro of this section, the FPGA implementation is quickly ported to any FPGA without much hassle. The only things needed to be changed were the constraints - as the newer Vivado uses a different syntax for the constraints - and an instantiation of a different frequency generator - as the Artix 7 has no DCMs.

As the Artix 7 FPGA gives the feature to read the operating temperature from inside the FPGA, the implementation was slightly changed to be more robust. Now the frequency generator is generating two frequencies, one low and one high, and the operating frequency of the brute-force cores is selected according to the current temperature. This should prevent temperature related errors. As a consequence, dynamic frequency reconfiguration is not possible on the Artix 7 FPGA.

The final design has 17 brute-force cores and can operate with a frequency of 190 MHz. An overview of the placed design can be seen in figure 4.9. Each color belongs to a different brute-force core. The resource utilization on the Artix 7 A200T is shown in Table 4.2.

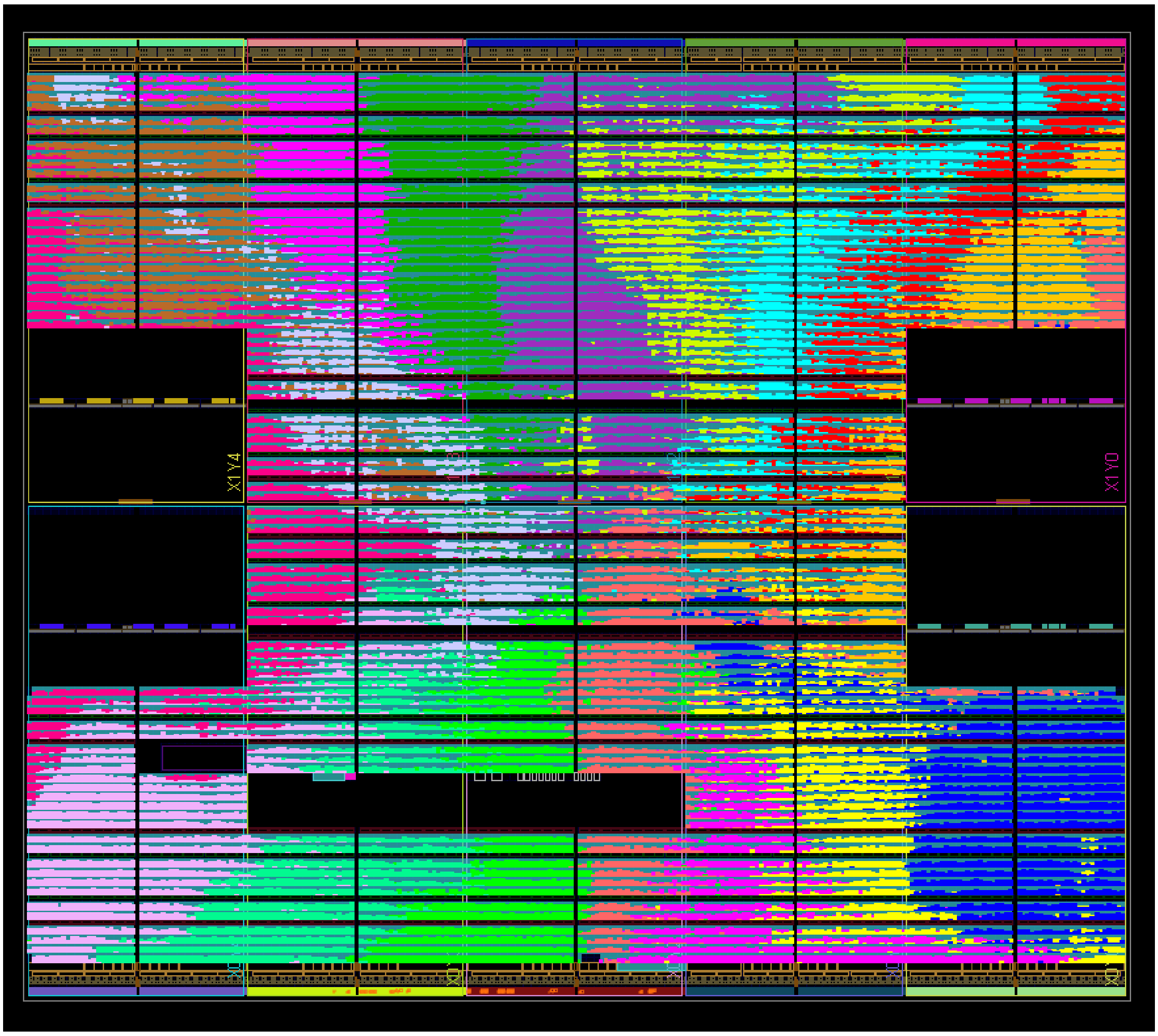


Figure 4.9: Artix 7 (XC7A200T) Placement Overview

Table 4.2: Resource Utilization of Artix 7 A200T

Slice Logic Utilization	Used	Available	Utilization
Number of Slice Registers	191133	267600	71.42%
Number used as Flip Flops	191133		
Number of occupied Slices	33427	33450	99.93%
Number of Slice LUTs	124617	133800	93.14%
Number used as logic	111995	133800	83.70%
Number used as Memory	12622	46200	27.32%
Number used as Shift Register	12622		
Number of bonded IOBs	22	285	7.72%
Number of BUFG/BUFGCTRLs	3	32	9.38%
Number used as BUFGs	1		
Number used as BUFGCTRLs	2		
Number of MMCME2_ADVs	1	10	10%
Number of XADCs	1	1	100%

CHAPTER 5

Results

This chapter presents the evaluation methods used, the final hardware implementations and the results of the brute force attacks.

5.1 Evaluation

We evaluated our implementations using custom generated test transport streams and checking if the keys (common words) were correctly found.

Since the design consists of fully pipelined brute-force cores, the brute-force key rate, and therefore the performance, is directly proportional to the operating frequency of the cores and the total number of brute-force cores. More specifically, $Freq * 10^6 * Cores$ gives the number of keys iterated per second, where $Freq$ is the operating frequency of the brute-force cores in MHz and $Cores$ is the total number of brute-force cores. This number is static, since none of the parameters are changed during the brute-force procedure.

As a result, the worst case execution time of a single key can be easily calculated by the following formula $\frac{2^{48}}{Freq*10^6*Cores}$, which gives the worst case time in seconds.

Except for the performance comparison that can be made with the existing brute-force implementations stated in Section 3.3, an additional comparison is made with the CUDABISS 2.5 Utility, which provides a brute-force implementation for GPUs. The average key rate is given by the utility itself. The utility is tested on a NVidia GeForce GTX 570 and a NVidia GeForce GTX 980 Ti.

Power statistics are also provided, firstly by giving the estimated power analysis of the tools themselves and secondly by measuring the actual power consumption of the boards when idle and under Load. Where as the estimated power consumption of the tools are for the FPGAs only, the measured power consumption contains also the power going to the peripheral, like the FX2 Microcontroller and the power supply itself.

5.2 Results

The final generated Bitstreamfile for the Spartan 6 FPGAs contains 11 brute-force cores that can operate at a Frequency of 150 MHz and the Bitstreamfile for the Artix 7 FPGAs contains 17 brute-force cores that can be operate at a frequency of 190 MHz.

Table 5.1: Worst case execution time of a single Spartan 6 FPGA for different operating frequencies.

Operating Frequency	Brute-force Key Rate (k/s)	Time to iterate 2^{48} key space
150 MHz	$1.65 * 10^9$	47h 23m
140 MHz	$1.54 * 10^9$	50h 46m
130 MHz	$1.43 * 10^9$	54h 41m
120 MHz	$1.32 * 10^9$	59h 14m
108 MHz	$1.188 * 10^9$	65h 49m
96 MHz	$1.056 * 10^9$	74h 2m
72 MHz	$792 * 10^6$	98h 43m
48 MHz	$528 * 10^6$	148h 5m

The frequencies stated above are in both cases actually lower than the possible frequencies reported by the timing analyzers. Running the actual boards on the frequencies reported by the timing analyzers did not work, because the found flag would never be asserted and the whole key space would be iterated without finding the key. Reason for this behavior is probably the undistributed power consumption of the board, which produces high local temperatures on high operating frequencies that bring timing violations to the foreground.
On the Spartan 6 this was solved by changing the synthesis and implementation Settings of the tool to be more power oriented. This reduced the maximum achievable operating frequency from 153 MHz to 150 MHz and increased the resource utilization by a small percentage.
On the Artix 7 a change of the synthesis and implementation settings was not possible due to the high resource utilization (99% Slice Utilization). So experiments were conducted to find the highest stable operating frequency and the frequency was then manually restricted from 200 MHz to 190 MHz.

The fact that the success of the brute-force procedure is indirectly temperature dependent, is also the reason why, as stated in Section 4.1, the host software has an auto reduce frequency feature on the occasion that the key isn't found.

Table 5.1 and Table 5.3 report the worst case time needed to complete a brute-force attack on a Spartan 6 and Artix 7 FPGA, respectively. Where as a single Spartan 6 needs around 47 and a half hours at worst, the higher number of bruteforce cores and frequency of the Artix 7 allow it to do the same in around 24 hours. The 36 Spartan 6 FPGA cluster will complete in around 1 and a half hours, as seen in Table 5.2.
Comparing these worst case execution times with the worst time execution times of a NVidia GTX 980 Ti GPU running CUDABISS - as shown in Table 5.4 - shows a 3.5 speedup for a single Spartan 6, and a 127 speedup for the FPGA cluster.

Table 5.5 shows the estimated wattage of the FPGA implementation and the actual measured idle and load wattage of the boards. As explained in the previous section, the measured wattage contains also the power consumption of the peripheral.

Table 5.2: Worst case execution time of a FPGA cluster containing 36 Spartan 6 FPGAs for different operating frequencies.

Operating Frequency	Brute-force Key Rate (k/s)	Time to iterate 2^{48} key space
150 MHz	$59.4 * 10^9$	1h 19m
140 MHz	$55.44 * 10^9$	1h 24m
130 MHz	$51.48 * 10^9$	1h 31m
120 MHz	$47.52 * 10^9$	1h 38m
108 MHz	$42.768 * 10^9$	1h 50m
96 MHz	$38.016 * 10^9$	2h 3m
72 MHz	$28.512 * 10^9$	2h 45m
48 MHz	$19.008 * 10^9$	4h 7m

Table 5.3: Worst case execution time of a single Artix 7 FPGA for different operating frequencies.

Operating Frequency	Brute-force Key Rate (k/s)	Time to iterate 2^{48} key space
190 MHz	$3.23 * 10^9$	24h 12m
100 MHz	$1.7 * 10^9$	46h 0m

Table 5.4: Worst case execution time of CUDABISS 2.5 for different GPUs.

GPU	Brute-force Key Rate (k/s)	Time to iterate 2^{48} key space
NVidia GTX 570	$166 * 10^6$	471h 1m
NVidia GTX 980 Ti	$465 * 10^6$	168h 9m

Table 5.5: Power statistics for the different FPGAs.

FPGA	Tool Wattage (Estimated)	Idle Wattage (Measured)	Load Wattage (Measured)
Spartan 6	6.834 W	N/A	N/A
Artix 7	10.233 W	2 W	14 W

CHAPTER 6

Conclusion

Even though the DVB Consortium has superseded the CSA with CSA-3, which uses a 128-bit key for encryption, the CSA Algorithm is still widely used. A successful real time attack on a transport stream encrypted with DVB-CSA would have to be countered with drastic measures by the DVB Consortium and their clients.

Going by the Conditional Access Systems used today, the common word of the DVB-CSA is updated every 10-120 seconds. The time goal to brute-force a transport stream in real time would therefore be under 10 seconds. As stated in Chapter 5, with our 36 FPGA cluster we can brute-force a transport stream package within 1 hour and 19 minutes.

In order to find a key in less time, proportionally more FPGAs are needed. Table 6.1 tries to put that into perspective, and shows the estimated number of FPGAs needed to achieve certain time goals. Specifically 17060 Spartan 6 FPGAs or 8715 Artix 7 FPGA would be necessary in order to find the key within 10 seconds.
It also has to be noted that on those numbers of FPGAs, the communication overhead between host and FPGA is not negligible anymore, as it is in our current 36 FPGA Cluster.

The estimated costs seen in Table 6.1 are the costs of buying the respective number of FPGAs from DigiKey[16]. Those are only the costs of the unconnected FPGAs themselves and do not contain the costs of communication Microcontrollers, power supplies, connection and PCB boards, which would increase the cost considerably. The costs of a single FPGA can be found in Table 6.2.

As is evident, even 23 years after its adoption, the DVB-Common Scrambling Algorithm can be considered safe against casual attackers. Nevertheless, the speed of the brute-force attack has been increased considerably by our hardware-based design.

Table 6.1: FPGA number estimation for specific timing goal.

Time Goal	FPGA	Operating Frequency	Number of Brute-force Cores	Number of FPGAs	Estimated Cost
1 minute	Spartan 6	150 MHz	31275	2844	469’544 USD
1 minute	Artix 7	190 MHz	24691	1453	281’446 USD
10 seconds	Spartan 6	150 MHz	187650	17060	2’816’606 USD
10 seconds	Artix 7	190 MHz	148145	8715	1’688’096 USD

Table 6.2: Current costs of FPGAs on the DigiKey Electronics website [16].

FPGA	Cost
Spartan 6 (XC6SLX150-2FGG484C)	165.1 USD
Artix 7 (XC7A200T-1FBG484C)	193.7 USD

References

[1] *Analysis of the DVB Common Scrambling Algorithm*,
Weinmann RP., Wirt K. (2005). In: Chadwick D., Preneel B. (eds) Communications and Multimedia Security. IFIP - The International Federation for Information Processing, vol 175. Springer, Boston, MA

[2] *Breaking DVB-CSA*,
Tews E., Wälde J., Weiner M. (2012). In: Armknecht F., Lucks S. (eds) Research in Cryptology. WEWoRC 2011. Lecture Notes in Computer Science, vol 7242. Springer, Berlin, Heidelberg

[3] *On the Security of Digital Video Broadcast Encryption*,
Markus Diett (2007), Diploma Thesis, Ruhr-University Bochum, Chair for Communication Security (COSY) Prof. Dr.-Ing. Christof Paar, Dipl.-Inf. Andy Rupp Ruhr-University Bochum

[4] *DVB TS Vollverschluesselung geknackt*,
Colibri (Pseudonym), dec 2011, colibri.dvb@googlemail.com,
homepage http://colibri-dvb.info

[5] Homepage of the DVB Consortium,
https://www.dvb.org/

[6] Homepage of Copacobana cluster system,
http://www.copacobana.org/

[7] Homepage and repository of FFdcsa software,
https://github.com/gfto/tsdecrypt/tree/master/FFdecsa

[8] Homepage and repository of AYCWABTU software,
https://github.com/aycwabtu/aycwabtu

[9] Homepage of libdvbcsa library,
http://www.videolan.org/developers/libdvbcsa.html

[10] Homepage of the VideoLAN Organization,
http://www.videolan.org/

[11] *DVB Common Scrambling Algorithm Distribution Agreements*,
DVB Document A011 rev.1, June 1996,
http://www.broadcasting.ru/pdf-standard-specifications/conditional-access/dvb-csa/a011r1.pdf

[12] *Digital Video Broadcasting(DVB); Content Scrambling Algorithms for DVB-IPTV Services using MPEG2 Transport Streams*,
ETSI TS 103 127 v1.1.1 (2013-05), Technical Specification,
http://www.etsi.org/deliver/etsi_ts/103100_103199/103127/01.01.01_60/ts_103127v010101p.pdf

[13] *Descrambling DVB data according to ETSI common scrambling specification*,
Simon Bewick, UK Patent Application GB2322995A, 1998

[14] *Design and implementation of MPEG-2/DVB scrambler unit and VLSI chip*,
Won-Ho Kim, Kyung-Jae Chen, and Hyun-Suk Cho, Consumer Electronics, IEEE Transactions on, 43(3):980-985, aug 1997.

[15] Webpage of dvb.sourceforge.net analyzing the Packetized Elementary Stream Headers,
http://dvd.sourceforge.net/dvdinfo/pes-hdr.html

[16] Homapage of Digi-Key Electronics,
https://www.digikey.com/