

# Institutionen för systemteknik

## Department of Electrical Engineering

Examensarbete

### Replacement of the DVB-CSA

**Analysis of new and alternative encryption algorithms and scrambling methods for digital-tv and implementation of a new scrambling algorithm (AES128) on an FPGA**

Examensarbete utfört i Elektroteknik  
vid Tekniska högskolan vid Linköpings universitet  
av

**Gustaf Bengtz**

LiTH-ISY-EX--14/4791--SE

Linköping 2014



**Linköpings universitet**  
**TEKNISKA HÖGSKOLAN**



# **Replacement of the DVB-CSA**

**Analysis of new and alternative encryption algorithms and scrambling methods for digital-tv and implementation of a new scrambling algorithm (AES128) on an FPGA**

Examensarbete utfört i Elektroteknik  
vid Tekniska högskolan vid Linköpings universitet  
av

**Gustaf Bengtz**


LiTH-isy-ex--14/4791--SE

Handledare: **Oscar Gustafsson**  
isy, Linköpings universitet  
**Patrik Lantto**  
WISI Norden

Examinator: **Kent Palmkvist**  
isy, Linköpings universitet

Linköping, 12 augusti 2014



	<b>Avdelning, Institution</b> Division, Department  ISY Embedded systems Department of Electrical Engineering SE-581 83 Linköping	<b>Datum</b> Date  2014-08-12				
<b>Språk</b> Language  <input type="checkbox"/> Svenska/Swedish <input checked="" type="checkbox"/> Engelska/English  <input type="checkbox"/> _____	<b>Rapporttyp</b> Report category  <input type="checkbox"/> Licentiatavhandling <input checked="" type="checkbox"/> Examensarbete <input type="checkbox"/> C-uppsats <input type="checkbox"/> D-uppsats <input type="checkbox"/> Övrig rapport <input type="checkbox"/> _____	<b>ISBN</b> _____ <b>ISRN</b> LiTH-ISY-EX--14/4791--SE <b>Serietitel och serienummer</b> <b>ISSN</b> Title of series, numbering      _____				
<b>URL för elektronisk version</b>						
<table border="0"> <tr> <td style="vertical-align: top;"><b>Titel</b> Title</td> <td>Ersättning av DVB-CSA Replacement of the DVB-CSA</td> </tr> <tr> <td style="vertical-align: top;"><b>Författare</b> Author</td> <td>Gustaf Bengtz</td> </tr> </table>			<b>Titel</b> Title	Ersättning av DVB-CSA Replacement of the DVB-CSA	<b>Författare</b> Author	Gustaf Bengtz
<b>Titel</b> Title	Ersättning av DVB-CSA Replacement of the DVB-CSA					
<b>Författare</b> Author	Gustaf Bengtz					
<b>Sammanfattning</b> Abstract  <p>This report addresses why the currently used scrambling standard CSA needs a replacement. Proposed replacements to CSA are analyzed to some extent, and an alternative replacement (AES128) is analyzed.</p> <p>One alternative being the CSA3, and the other being the CISSA algorithm. Both of the proposed algorithms use the AES algorithm as a base. The CSA3 combines AES128 with a secret cipher, the XRC, while CISSA uses the AES cipher in a feedback mode. The different utilizations makes CSA3 hardware friendly and CISSA software friendly.</p> <p>The implementation of the Advanced Encryption Standard (AES) is analyzed for a 128 bit key length based design, and a specific implementation is presented.</p>						
<b>Nyckelord</b> Keywords      DVB, scrambling, CISSA, cipher, CSA, CSA3, AES128, FPGA						



## **Abstract**

This report addresses why the currently used scrambling standard CSA needs a replacement. Proposed replacements to CSA are analyzed to some extent, and an alternative replacement (AES128) is analyzed.

One alternative being the CSA3, and the other being the CISSA algorithm. Both of the proposed algorithms use the AES algorithm as a base. The CSA3 combines AES128 with a secret cipher, the XRC, while CISSA uses the AES cipher in a feedback mode. The different utilizations makes CSA3 hardware friendly and CISSA software friendly.

The implementation of the Advanced Encryption Standard (AES) is analyzed for a 128 bit key length based design, and a specific implementation is presented.





---

# Notation

## EXPRESSIONS

Expression	Meaning
AES	Advanced Encryption Standard
CAM	Conditional Access Module
CAS	Conditional Access System
CBC mode	Cipher block chaining mode
CC	Content Control
Ciphertext	Encrypted plaintext
CISSA	Common IPTV Software-oriented Scrambling Algorithm
CPU	Central Processing Unit
CSA	Common Scrambling Algorithm
CTR mode	Counter mode
CW	Control Word, which is a key
DVB	Digital Video Broadcasting
ECM	Entitlement Control Message. CW encrypted by the CAS
EMM	Entitlement Management Messages
ES	Elementary stream
ETSI	European Telecommunications Standards Institute
FF	Flip-Flop
FPGA	Field-programmable gate array
HD	High-Definition
IPTV	Internet Protocol Television
IV	Initialization vector
LFSR	Linear Feedback Shift-Register
LSB	Least Significant Bit
LUT	Look-up Table
MSB	Most Significant Bit
Nibble	Half a byte (4 bits)
Nonce	A value that is only used once
P-Box	Permutation-Box
PES	Packetized Elementary Stream
Plaintext	Content, data
PS	Program Stream
SD	Standard-Definition
S-Box	Substitution-Box
STB	Set-top Box
TS	Transport Stream. Contains data
XRC	eXtended emulation Resistant Cipher

---

# Contents

<b>Notation</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Problem specification . . . . .	2
1.3 Constraints . . . . .	2
1.4 Methodology . . . . .	2
<b>2 Digital Video Broadcasting (DVB)</b>	<b>5</b>
2.1 Head-end . . . . .	5
2.2 Control word . . . . .	6
2.3 Conditional Access System . . . . .	7
2.3.1 Standards . . . . .	7
2.4 DVB-SimulCrypt . . . . .	8
2.5 Common Interface . . . . .	8
2.5.1 CI+ . . . . .	8
2.6 Conditional Access Module . . . . .	8
<b>3 Cryptography</b>	<b>11</b>
3.1 Why cryptography is needed . . . . .	11
3.2 Data packets . . . . .	12
3.2.1 TS packets . . . . .	12
3.2.2 PES packets . . . . .	14
3.3 Encryption and Decryption . . . . .	14
3.3.1 Symmetric-key encryption . . . . .	14
3.3.2 Public-key encryption . . . . .	15
3.3.3 Combination of encryption methods . . . . .	15
3.4 Ciphers . . . . .	16
3.4.1 Block cipher . . . . .	16
3.4.2 Stream cipher . . . . .	17
3.4.3 Decryption . . . . .	18
3.5 Confusion and Diffusion . . . . .	18
3.5.1 S-boxes and P-boxes . . . . .	18

3.6	Secrecy . . . . .	18
<b>4</b>	<b>Common Scrambling Algorithm</b>	<b>21</b>
4.1	The need for a new standard . . . . .	21
4.2	Layout of the CSA . . . . .	22
4.3	Security . . . . .	22
4.3.1	Breaking the CSA . . . . .	22
<b>5</b>	<b>CISSA or CSA3</b>	<b>25</b>
5.1	Replacements . . . . .	25
5.2	CISSA . . . . .	26
5.2.1	Software friendly . . . . .	26
5.3	CSA3 . . . . .	26
5.3.1	Hardware friendly . . . . .	26
5.4	Selection of the algorithm . . . . .	27
<b>6</b>	<b>Advanced Encryption Standard</b>	<b>29</b>
6.1	Introduction . . . . .	29
6.2	Method . . . . .	29
6.2.1	InitialRound . . . . .	30
6.2.2	SubBytes . . . . .	30
6.2.3	ShiftRows . . . . .	30
6.2.4	MixColumns . . . . .	31
6.2.5	AddRoundKey . . . . .	31
6.3	KeyExpansion . . . . .	31
6.3.1	Key-schedule core . . . . .	31
6.3.2	Rijndael's S-Box . . . . .	32
6.3.3	Rcon . . . . .	32
<b>7</b>	<b>Implementation</b>	<b>33</b>
7.1	Manager entity . . . . .	34
7.2	CBC entity . . . . .	34
7.3	Cipher entity . . . . .	35
7.4	Keyexpansion entity . . . . .	36
7.4.1	Keycore entity . . . . .	36
7.5	Round entity . . . . .	38
7.5.1	Addroundkey entity . . . . .	38
7.5.2	The mulblock entity . . . . .	39
<b>8</b>	<b>Result</b>	<b>41</b>
8.1	Problems . . . . .	41
8.2	Hardware . . . . .	43
8.2.1	Hardware usage . . . . .	43
8.3	Further development . . . . .	45
8.3.1	Rijndael's S-Box . . . . .	45
8.3.2	Critical Path . . . . .	45
8.4	Tests . . . . .	46

---

8.5	Comparison to other implementations . . . . .	47
8.6	Discussion . . . . .	47
8.7	Conclusions . . . . .	48
<b>A</b>	<b>Matrixes</b>	<b>51</b>
<b>B</b>	<b>Flowcharts</b>	<b>55</b>
<b>C</b>	<b>Test vectors</b>	<b>59</b>
<b>D</b>	<b>Examples</b>	<b>67</b>
	<b>List of Figures</b>	<b>81</b>
	<b>Bibliography</b>	<b>83</b>



# 1

---

## Introduction

WISI Norden AB, previously A2B Electronics, is a Swedish company founded in 1997. The company is a developer of head-end cable-TV distribution systems. WISI Norden develops and designs both hardware and software, with the purpose of providing Digital TV solutions.

The purpose of this thesis has been to find a replacement to the currently implemented scrambler, located in the head-end solutions. The previous scrambler needed to be replaced, since it was designed in 1994 and was supposed to last for ten years. The scrambler is used to render the digital television streams unreadable if the user does not subscribe to the encoded channels.

The task was to evaluate and analyze a few potential scrambling algorithms, and then choose one algorithm to implement in WISI Norden's devices.

### 1.1 Background

The formerly used *common scrambling algorithm* (CSA) has due to recent progresses in television broadcasting become obsolete. CSA was designed to make software descrambling hard, if possible, while making hardware descrambling fast.

There are two suggested replacements of CSA. The first one is named after the CSA, and is called CSA3. The reason as to why it is not called CSA2 is since there already exists an algorithm which is called CSA2. CSA2 is the same algorithm as CSA, just with a different key-length. The second algorithm is the software-friendly descrambling algorithm CISSA. Both of them are based on the public Advanced Encryption Standard - 128 (commonly known as the AES-128). There

are three versions of the AES, with varying numbers. The number depicts what key-length the AES uses.

WISI Norden wanted to evaluate the replacement algorithms, even though the CSA is still used in the DVB world. This was done to make sure that there was an alternative to the CSA, when other companies would start to switch scrambling methods. WISI Norden has also had some requests to implement other scrambling methods from clients.

## 1.2 Problem specification

The task was to analyze the possible replacements for the common scrambling algorithm, and decide which one was the most suitable replacement. After choosing an algorithm, that algorithm was to be implemented from scratch, making decisions to minimize the hardware usage while achieving the frequency used by the rest of the system. The decisions made were to be motivated either through simulations or reference literature.

There were two proposed replacements to be compared and analyzed to find what made one of them software-friendly and the other one hardware-friendly.

## 1.3 Constraints

The thesis has been limited to implement the scrambling algorithm chosen in consent between the author and the supervisor at WISI Norden. The algorithm that was chosen, after analysis of the two proposed algorithms, was the AES128 algorithm in CBC-mode (chapter 3.4.1) with a set IV, according to the CISSA standard. This corresponds to the CISSA algorithm [5]. The implementation is focused on minimizing the hardware usage, while achieving a throughput of at least 1 Gbits/s.

## 1.4 Methodology

The project was split into a set of tasks, to be performed in the order written below. Performing the tasks in this order was done to decrease the complexity of the separate tasks.

- Literature study
- Choosing an algorithm
- Design and test of entities
- Implementation
- Optimization



To gain some knowledge about cryptography, a literature study was first conducted. This provided some insight into what the strengths and weaknesses of the algorithms actually were. The AES cipher was chosen as an initial algorithm, since both of the proposed algorithms used the AES as a base. The other parts of the cipher, which ever was chosen, were to be added after the AES was finished. Using the gathered background information about how the algorithm worked made design and testing of the entities rather easy. The lower level entities were designed first, which allowed for easier testing of separate parts of the system. Knowing that the functionality, of low level entities, was already present allowed for easier merging of entities. This led to the system being implemented through bottom-up design.



# 2

---

## Digital Video Broadcasting (DVB)

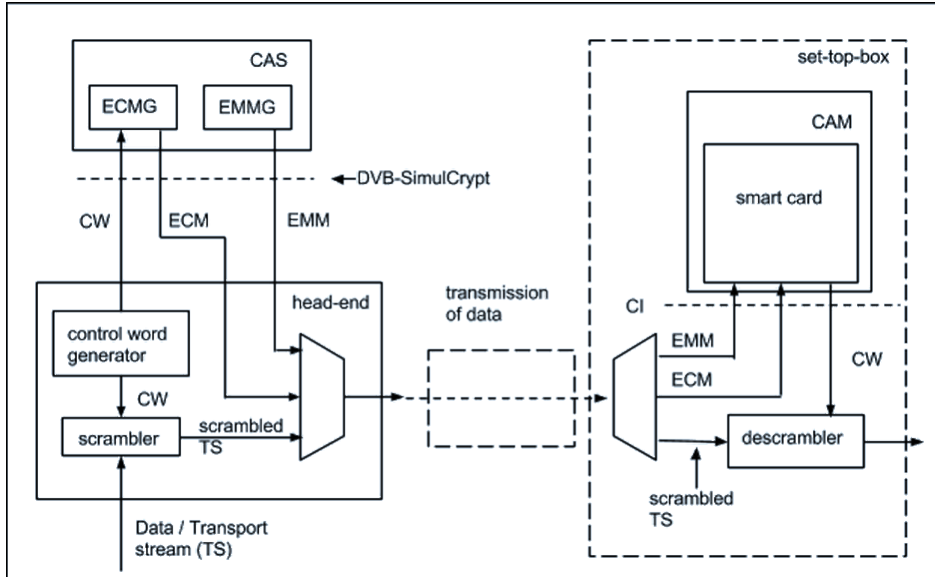
There are many parts that are needed to provide Digital Video Broadcasting (DVB) with a secure way of transmitting streams of data without facing the risk of content getting stolen. The following parts will be treated in this thesis:

- Head-end - explained in section 2.1
- CA system (CAS) - explained in section 2.3
- Common Interface - explained in section 2.5
- Scrambler - explained in chapter 3
- Descrambler - the inverse of a scrambler

The parts are connected according to figure 2.1. The CAM is explained in section 2.6, the ECM and EMM signals are mentioned briefly in section 2.3 and the DVB-SimulCrypt is described in section 2.4.

### 2.1 Head-end

The head-end is the system where the scrambler is located. Except for the scrambler, decoding and generation of program specific information takes place in the head-end. The head-end decodes, encrypts and encapsulates data which it has received from content providers, before transmitting it.



*Figure 2.1: The DVB setup*

## 2.2 Control word

TS packets (Transport Streams), which contain data received from distributors, are scrambled using a key which is called a control word. Control words are usually changed every 120th second, but might be changed more often. Some systems change the control word every 10th second. Finding out just one control word has very little effect on content theft, since it will only be usable for a few seconds before being changed. Because of the high frequency in which the control words are changed, one means of security is provided. The control words are generated randomly to make sure that consecutive control words can not be derived from each other.

The following section describes the setup of the DVB system, which can be viewed in figure 2.1. The control word is sent to a CA system (Conditional Access System) where the control word is encrypted as an ECM (Entitlement Control Message). The CA system also generates an EMM (Entitlement Management Message) which tells the smart card, which is located in the CAM (Conditional Access Module), contents the user is allowed access to. This could for instance be whether the user has paid to view premium football games or not. The ECM and EMM are then sent back to the head-end where they are attached to the scrambled TS packet using a multiplexer. This package is sent to a receiver, which is usually a TV. The ECM, EMM and TS packet are separated when they arrive. The ECM and EMM are sent through a CI (Common Interface) to the CAM, where the ECM (encrypted control word) is decrypted using a decryption algorithm located on the smart card. The resulting control word is then used to descramble the TS packet.

The TS packet is encrypted once more if the CI is a CI+, otherwise it is sent in the clear back to the receiver where the data is processed before it is dispatched to the user. The CI and CI+ as well as the extra encryption are all discussed in section 2.5.

## 2.3 Conditional Access System

To make sure that users fulfill a set of criteria, before being allowed to access content, *Conditional Access* (CA) is used. Conditional Access is provided, based on information about the user, in a system separate from the head-end. Content is first scrambled, and decoded in a head-end. The control word, used to scramble the data, is sent to the Conditional Access system (CAS) where it is encoded. The CA system consists of an EMM-generator (noted as EMMG in figure 2.1) and an ECM-generator (noted as ECMG in figure 2.1) among others. An ECM-generator encrypts the control word. The algorithms used in the generators differ between CA systems and is kept very secret, to make sure that the control word can not be stolen during transmission.

The ECM is generated using the control word, while the EMM is generated based on subscription- and payment information related to the user. The ECM-generators are deterministic, and differ from CAS to cas. The EMM can allow things, stretching from allowing a user to view a video for a few hours, to access a certain channel for an extended period of time. A TV will not display any channels without receiving an EMM allowing it to.

An example is that a user needs to pay for TV-services to be able to access content. The CAS generates an EMM which tells the smart card whether the user is allowed to access the requested material or not. The content provider also generates an ECM based on the control word, which the smart card decrypts and passes to the descrambler which decrypts the video stream. This is done if the EMM allows it.

### 2.3.1 Standards

Some of the CA systems currently in use are Viaccess, Conax, Irdeto, NDS, Strong and NagraVision. The CA systems are paired with *Conditional Access Modules* (CAM), which are located in the receiver. What CAS / CAM pair depends on the content provider. For instance, Conax is used by Com Hem, Viaccess is used by Boxer and Strong is used by Canal Digital.

CA system	Used by	Supports CI+
Viaccess	Boxer, SVT	Yes
Conax	Com Hem	Yes
Strong	Canal Digital	Yes

## 2.4 DVB-SimulCrypt

The control words used during scrambling can be sent to several different CA systems at once, resulting in several ECMs. This is called DVB-SimulCrypt, which is widespread in Europe. DVB-SimulCrypt works as an interface between the head-end and the CA system. DVB-SimulCrypt encourages the use of several CA systems at once [3]. This is done by sending the same control word to many CA systems at the same time, and then allowing them each to generate an ECM based on the control word. The multiplexer in the head-end then creates TS packets based on those, since the EMMs will determine whether the user is allowed access or not. A multiplexer is a basic logic circuit, which merges several signals into a single signal.

## 2.5 Common Interface

The Common Interface is the interface between the CAM and the host (Digital TV receiver-decoder). There are currently two versions of common interfaces in use, which are the CI and the CI+. The difference between them is that the output from the CI is unencrypted, while the output from the CI+ is encrypted [10]. This means that a clear TS packet is sent between the CI and the host, that can be copied. The data sent between the CI+ and host can not be copied due to it being encrypted, and therefore provides more security for content providers [6].

### 2.5.1 CI+

The CI+ realizes the possibility of yet another means of protecting content, which is called Content Control (noted as CC in Figure 2.2). Content control is a way of encrypting the content inside of the CAM, connected to the CI+ Module. The key used for the content control encryption is paired with the Digital TV Receiver, where the TS packet is decrypted before being made available to users. The general idea can be viewed in Figure 2.2.

CI+ encoding is often used to protect HD content, but not SD content. Since HD content is more high-profile, content distributors want to protect it more than the SD content. Protection of HD content requires scrambling using AES-128 in CBC-mode (explained in section 3.4.1). [10, 11]

## 2.6 Conditional Access Module

CA modules (CAMs) are responsible of decoding the scrambled TS packet received from the host. The CAM is inserted into a PCMCIA slot (Personal Computer Memory International Association) either into the TV or the set-top box. A set-top box is a box which is connected between the TV signal source and the TV. The set-top box is equipped with both a CI or CI+, and a CAM. The CAM consists of a slot for a smart card and a descrambler. The smart card decodes the

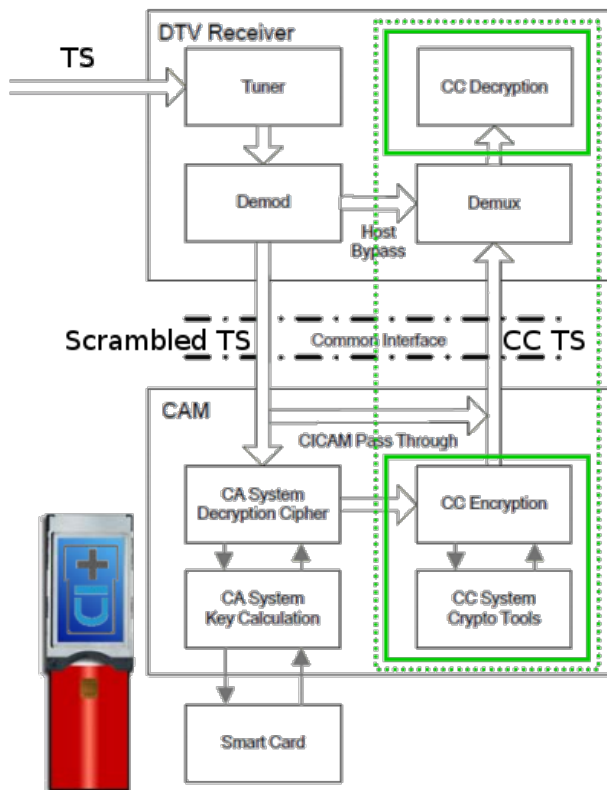


Figure 2.2: CI+ interface. [10, p. 10]

ECM and sends the control word back to the descrambler. The TS packet is then descrambled and the clear data is sent back to the host, from the CAM.





# 3

---

## Cryptography

Cryptography is the science of rendering content incomprehensible for undesired readers. Non-encrypted content is called plaintexts, and encrypted plaintexts are called ciphertexts in cryptography. However, securing content is not only about cryptography. The main reason why the encryption algorithms are attacked, is because an attack has a very low chance of being detected. There will be no traces of the attack, since the attacker's access will look just like an ordinary access. [13]

This can be compared to a real-life break-in. The break-in will be noticed if the thief breaks in using a crowbar. On the other hand, you might never notice that the security had been breached, if the the thief were to pick the lock instead. [13]

One of the more important cryptography rules, is to always assume that someone is out to get you. Because of this, Schneier and Fergusson [13, pp. 12–14] claims that it is always important to look for possible ways of breaking systems. Possible ways of bypassing the encryption and ways of breaking algorithms are more easily noted by looking at systems with this mindset, and doing so allows cryptographers to more easily notice faults, which can then be fixed and thereby provide a more robust security system.

### 3.1 Why cryptography is needed

Cryptography is the science of rendering plaintexts into ciphertexts to protect contents from unauthorized viewing. It is used in electronic communication for protection of e-mail messages and credit card information among other things. If data is sent without being encrypted, someone listening in to the transmission channel will be able to access the data.

For most people this is not a problem, but in some instances sending secure messages can be extremely important. One example is communication during war, where a single piece of intelligence might turn the tide of the entire war. Moreover, you do not want people to read your account information or credit card number when you do online shopping. Another reason is to make sure that users do not access premium content without paying for it, as the case is with DVB. All of these problems can be solved by cryptography and encryption.

## 3.2 Data packets

The data processed by the DVB systems is packaged into data packets before it is sent. All of them are created from ES packets (Elementary Stream) which are generally the output from an audio or video encoder. The ES-packets are packeted into PS- (Program Stream), TS- (Transport Stream) or PES packets (Packetized Elementary Stream) before being distributed. Among the three ways of packing data, only two are interesting from a DVB perspective. This is due to PS packets being used for storing data, while TS and PES are used for transmitting data. The interesting types, when working with DVB, are therefore the TS packets as well as the PES packets. PES packets are often packed into the payload of TS packets. TS packets are desirable since they are of a fixed length of 188 bytes, while the PES packets are desirable due to their strength. The payload is the part of the packets which is the actual data, which is everything except the header and adaptation field.

### 3.2.1 TS packets

TS packets are used by the DVB society due to their fixed length, and the fact that TS packets are meant to be used for streaming services, while PS packets are used for storing packets of data. TS packets have got a length of 188 bytes with a 4 byte long header. This means that the payload consists of a maximum of 184 bytes. The layout of a TS packet can be viewed in figure. 3.1[5]

TS header (clear)	Adaptation* Field (clear)	Encrypted TS Payload	Clear TS* Payload
-------------------------	------------------------------	----------------------	----------------------

*Figure 3.1: General layout of a data packet*

The TS packet consists of 4 different kinds of building blocks where only the header is guaranteed to be present. Those blocks are:

- Header
- Adaptation field
- Encrypted payload
- Clear payload

The byte-sizes of the building blocks of a TS packet are given in table 3.1. The `block_size` is the size of the blocks that are encrypted, and is 16 for all of the AES standards.

Part	Size in bytes
<code>header_size</code>	4
<code>adaptation_field_size</code>	the size of the adaptation field
<code>payload_size</code>	$188 - (\text{header\_size} + \text{adaptation\_field\_size})$
<code>encrypted_payload_size</code>	$\text{payload\_size} - [\text{payload\_size} \bmod \text{block\_size}]$
<code>clear_payload_size</code>	$[\text{payload\_size} \bmod \text{block\_size}]$ (or simply $\text{payload\_size} - \text{encrypted\_payload\_size}$ )

**Table 3.1:** Byte sizes of the parts in TS packets. The TS packet is 188 bytes.

The header is always 4 bytes, while the adaptation field can have any size between 0 and 183 bytes. This means that the clear payload can be of any size stretching from 0 bytes, to one byte smaller than the block size. The rest of the data consists of the payload.

### Header

The header consists of information regarding the packet, and has a `sync_byte` (with a hex-value of 0x47, or bit-value of 01000111) to announce the beginning of a packet. The value of the `sync_byte` corresponds to the ASCII-value of the letter G which stands for Go. The header also contains information as to whether there is an adaptation field and payload in the packet, what Packet ID (PID) the packet has, if it should be prioritized, whether the data is scrambled - and in that case if it was scrambled with an odd or even key, among others [4, pp. 25–26]. The header should never be encrypted and is always found at the beginning of a packet [5, pp. 10–11].

The header contains the following bits:

Bits	Name	Description
8	Sync byte	Fixed byte value 0x47
1	Transport Error Indicator	Uncorrectable bit errors exist
1	Payload Unit Start Indicator	TS packet contains PES packets or Program Specific Information (PSI data)
2	Transport Scrambling Control	00 No scrambling, 01 Reserved, 10 Even key, 11 Odd key
1	Transport Priority	1 gives this packet higher priority
13	PID	Packet identification number
1	Adaptation Field Control	Adaptation field exists
1	Contains Payload	Payload exists
4	Continuity Counter	Packet number. Used to make sure packets are not lost

### **Adaptation field**

The adaptation field is a padding field that is only inserted when the end of the data does not align with the end of the TS packet. This is done to make sure that the TS packet is filled with known data. Adaptation fields are never encrypted. [5, pp. 10–11]

### **Encrypted and clear payload**

Clear bytes of data tend to turn up, when working with block ciphers. This happens since block ciphers only encrypt data blocks of fixed sizes. The clear data is always located at the end of the received TS packet. When receiving a TS packet, the first thing to be done is to find the start of the payload. The start of the payload is found directly after the header, when there is no adaptation field. If an adaptation field is present, we can find the data after it. The length of the adaptation field is found in the beginning of it. When the start of the payload has been found, blocks of a given size are sent to the scrambler. The remainder of the data, when all of the blocks of the right size have been scrambled, is to be left clear. The number of unscrambled bytes might be of sizes up to one byte smaller than the block size. This means that the AES-128, which works on block sizes of 16 bytes, can have a maximum of 15 clear bytes. [5, pp. 10–11]

### **3.2.2 PES packets**

The PES packets have varying lengths of up to 64 kilo bytes, and are often packed into TS packets when distributed, due to the TS packets being strong. The payload data in the TS packets, when carrying PES packets, consist of the entire PES packets, which is the header as well as the data. PES packets do not use adaptation fields, since they are of adaptable lengths, as long as the length of the packet does not exceed 64 kilo bytes.

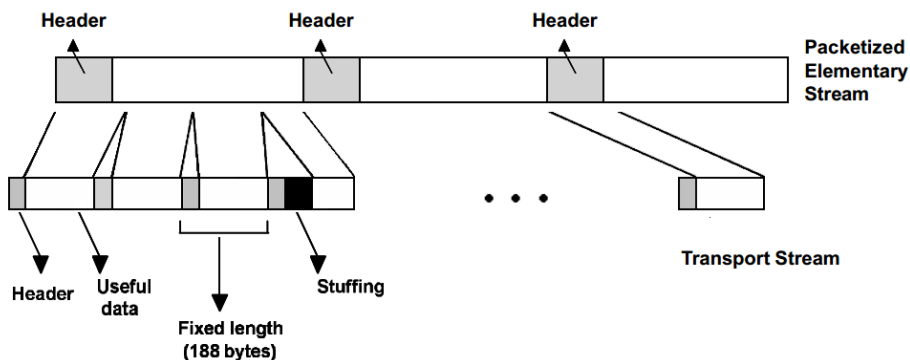
Since Digital Video Broadcasting seldom uses PES packets, an explanation of the elements of the PES header will not be done in this report. The derivation of PES packets from TS packets can be seen in Figure 3.2 [2, p. 9].

## **3.3 Encryption and Decryption**

There are three things that you need when you encrypt and decrypt messages. Those are the algorithm, plaintext and the key. Even though there are plenty of ways to encrypt messages, there are mainly two ways of sharing the encryption-key. The first method is the symmetric-key encryption, and the second method is the public-key encryption. [13]

### **3.3.1 Symmetric-key encryption**

The symmetric-key encryption uses the same key to encode and decode messages. Distribution of the key, when using the symmetric-key encryption is troublesome and the fact that both parties need access to the same secret key is a major



**Figure 3.2:** PES packet derived from TS packets. The packet in the top is a PES packet and the packets in the bottom are TS packets.

drawback of the symmetric key encryption, as compared to the public-key encryption method. Sending the key in an email is a bad idea, since the persons who want to read the sent messages are most likely already listening. They will therefore obtain the key as well as the means to decode the messages. Both the CSA and the AES encryption methods are symmetric-key encryptions, using the same key for encryption and decryption. [13]

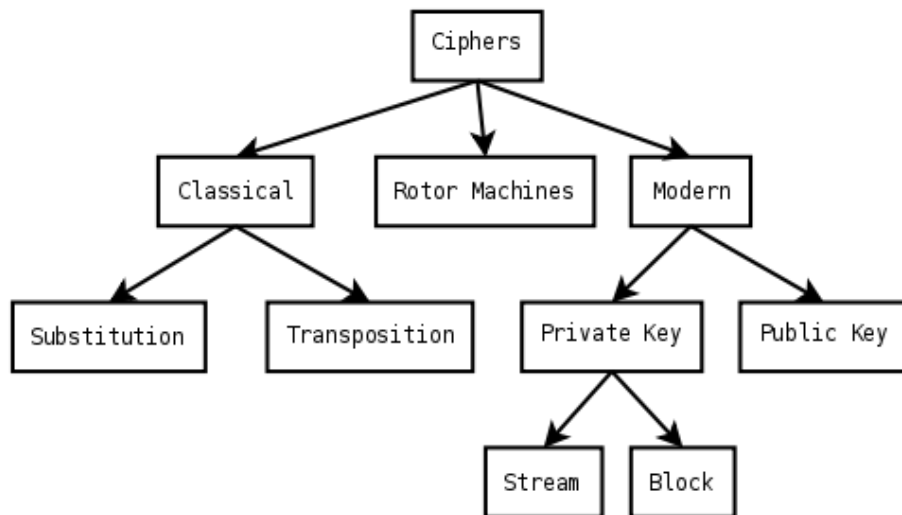
### 3.3.2 Public-key encryption

The public-key encryption uses a public key that anyone can look up, and a secret key that only one person knows [16, pp. 25–32]. For instance say that two persons, Bob and Alice, want to communicate. Bob produces a keypair  $P_{Bob}$  (Bob's public key) and  $S_{Bob}$  (Bob's secret key) and publishes  $P_{Bob}$  for anyone to see. When Alice wants to send Bob a message, she looks up Bob's public key  $P_{Bob}$ , which she uses to encode her message. When she sends Bob the message, Bob decodes the message using his secret key  $S_{Bob}$ . Since Alice now knows both the plaintext, and can find out what the corresponding ciphertext will be, she could potentially try to find Bob's secret key. [13]

### 3.3.3 Combination of encryption methods

If the public-key encryption seems secure and easy to manage, how come it is not the only encryption method used? The reason is that the public-key encryption is not as effective as the symmetric-key encryption. It is common to use a combination of those two when an easy, effective way to encrypt messages is desired.

To combine the two encryption methods, the symmetric-key algorithm encodes the plaintext into a ciphertext. Then the public-key encryption encrypts the key used by the symmetric-key encryption. The encoded key is then sent together with the ciphertext to the recipient, which decodes the symmetric key using the secret key. The plaintext is obtained by decrypting the ciphertext using the sym-



*Figure 3.3: Different kinds of ciphers. [23]*

metric key.

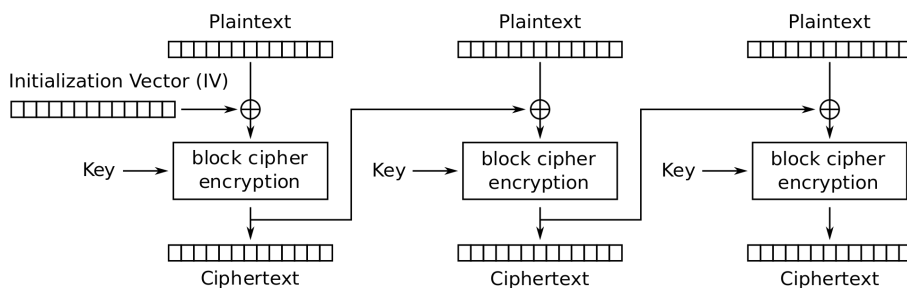
## 3.4 Ciphers

A cipher is the same as an encryption algorithm, which operates on either plaintexts or ciphertexts to perform encryption or decryption. Figure 3.3 shows how the different kinds of ciphers can be split into sub-groups. The first branch splits into Classical-, Rotor Machine- and Modern ciphers. Substitution and Transposition are still used in modern algorithms, even though they are considered classical ciphers. The Modern ciphers are the Private key and Public key (described in chapter 3.3.1 and 3.3.2). The CSA algorithm uses both the stream- and block ciphers, while the AES algorithm only uses a block cipher.

There are mainly two kinds of ciphers that are used when designing modern cryptosystems. Those ciphers are called block ciphers and stream ciphers. Many systems use a combination of block ciphers and stream ciphers to provide security.

### 3.4.1 Block cipher

A block cipher operates on fixed sized sets of data. These sets are called blocks, which is the reason why they are called block ciphers. Their being fixed sizes might cause a need for padding of the blocks, in case the plaintext contains a number of bytes that is not a multiplier of the blocksize. Block ciphers often use a combination of Substitution-boxes (S-boxes) and Permutation-boxes (P-boxes)



**Figure 3.4:** Cipher block chaining mode, [22]

in a so-called SP-network (S-box / P-box network) (Figure 3.5). There are many modes of block ciphers, but the two recommended by Schneier and Ferguson [13] are the CBC-mode and the CTR-mode, which are described in the following sections.

### CBC

CBC stands for *cipher block chaining* and is performed by encrypting the result of an XOR (basic logic component) between an Initialization Vector (IV) and the plaintext. The resulting ciphertext is then fed back to the XOR replacing the IV. This means that the data input into the cipher will be the result of an XOR between the previous result, and the next plaintext. This is then put into an XOR with the next plaintext, which is then encrypted in the cipher. For reference, see image 3.4. [18, pp. 109–111]

### CTR

CTR stands for *counter*, and refers to the way the IV is generated. The counter outputs a value, which is encoded with the key. The counter often uses a Linear Feedback Shift Register (LFSR) with some sort of logical function, most often XORs. The counter has got an internal state, which it manipulates in order to create the next state. The state is what is output from the counter and sent to an XOR together with the plaintext, producing the ciphertext. The counter is incremented and the procedure is iterated [18, p. 111].

### 3.4.2 Stream cipher

Stream ciphers work on streams of data. They usually consist of a keystream generator which performs an XOR with the data [16, pp. 67]. An effective implementation of the stream cipher is to use a linear feedback shift-register which uses the current internal state (key) to produce the next state by a simple XOR-addition between two or more of the bits in the state. This is mainly used because of how easy it is to construct in hardware [20].

### 3.4.3 Decryption

Decryption is often performed by reversing the encryption. You need to know the algorithm, preferably through a mathematical representation, to calculate how to obtain the plaintext from the ciphertext. An example of how this is done for the CBC-mode (described in 3.4.1) is described in D in appendix D. It is here assumed that the decryption algorithm is known, for simplicity.

## 3.5 Confusion and Diffusion

Two properties that are needed to ensure that a cipher provides security are confusion and diffusion [15]. *Confusion* refers to making the relationship between ciphertext and key as complex as possible. *Diffusion* refers to replacing and shuffling the data, to make it impossible to analyze data statistically. This is usually done by performing substitutions and permutations in a simple pattern multiple times. This can easily be done by using an SP-network [18, pp. 74–79]. The first as well as the last step of SP-Networks is usually an XOR between the subkey and the data. A subkey is a key generated from the provided key, and is used to provide systems with more complex encryptions. Performing an XOR between a subkey and data is called *whitening*, and is according to Stinson [18, p. 75] regarded as a very effective way to prevent encryption/decryption without the key. The goal of this is to make it hard to find the key, even though one has access to multiple plaintext/ciphertext pairs produced with the same key [15].

However, a cipher is not guaranteed to be secure just because it provides these two properties.

### 3.5.1 S-boxes and P-boxes

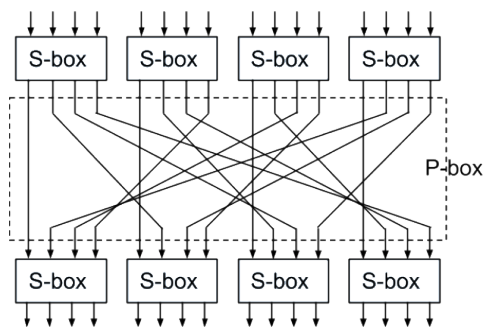
The S-box is one of the basic components that is used when creating ciphers. An S-box takes a number of input bits and creates an equal number of output bits. The way they are generated is non-linear. Implementing an S-box can effectively be done using lookup tables, since the function of an S-box is to substitute the input with a different output, which corresponds to the functionality of a lookup table. Each input has to correspond to a unique output, to make sure that the functionality of the S-Box can be uniquely reversed. If it can not be reversed, descrambling will be impossible. [18, pp. 74–75]

The second basic component used in cryptography is the P-box. A P-box shuffles and thereby rearranges the order of given bits. This can be viewed in the SP-network in figure ??, where the P-box is represented by the dotted rectangle in the middle.

## 3.6 Secrecy

Although encryption is important, as well as the strength of the encryption, neither using an algorithm designed for use in just a few applications nor using a





**Figure 3.5:** *SP-Network*

secret algorithm is ever a good idea. A simple mistake when designing an algorithm might turn an encryption that would otherwise have been strong, incredibly weak. If you use an open algorithm, faults will most likely be discovered and fixed by experienced cryptographers [13, pp. 23]. Keeping the key, which is used to encrypt the data, secret is what is important.



# 4

---

## Common Scrambling Algorithm

The Common Scrambling Algorithm (CSA) is currently the most commonly used encryption algorithm for encryption of video-streams in the DVB context. The CSA uses a combination of a block cipher, taking an input of 64-bit blocks, and a stream cipher. Both of the ciphers use the same key, which means that the entire system uses the same key. This means that the complete algorithm would break if the key would be recovered, as long as the person recovering the key knows what the decryption algorithm looks like. On the other hand, using the same key for the whole system allows for easily changing keys at regular intervals. [9, pp. 271–272]

CSA has been the official scrambling method for protecting DVB content since May 1994. CSA was to be easily implemented in hardware and hard to implement in software to, among other reasons, make reverse-engineering of the algorithm difficult [1].

There are two versions of the DVB-CSA, CSA1 and CSA2, where the key-length is the only difference. [1, p. 23]

### 4.1 The need for a new standard

The DVB-CSA standard offers short-term protection, while it assumes content is viewed in real time and not stored. Due to the development of how content has come to be consumed during recent years, the focus has shifted from transmitting contents to primarily being able to distribute content across homes. As a result of this, functionality needs to be changed from securing the delivery of content, to securing the content. [7]

Another thing to bear in mind is the fact that more CPU-based units, such as smart-phones, tablets and computers are used to access contents now more than ever. In order to allow for descrambling on CPU-based units, a software-friendly scrambling algorithm is required.

## 4.2 Layout of the CSA

The CSA consists of a block cipher and a stream cipher which are connected in sequence [9, p. 271]. The block cipher reads blocks of data, each consisting of 64-bits, which are run in CBC-mode (see section 3.4.1). The block cipher processes these blocks of data in 56 rounds. The output of the block cipher is sent to the stream cipher, where additional encoding is performed. The first block of data sent from the block cipher to the stream cipher is used as an IV for the stream cipher. It is therefore not encoded the stream cipher. [21]

## 4.3 Security

One of the problems associated with control word distribution is that control word sharing has become rather common [7]. Control word sharing is primarily done by connecting several set-top boxes into a network in order to share the decrypted control word, and get access to more content. It has probably become more popular since the control words are sent in the clear between the smart card and the STB, meaning that a user might grab the clear control word during transmission and redistribute it over the internet. This has become a financial problem for content distributors, since people have stopped paying for the content that they are watching.

One way of dealing with control word sharing is to decode the encrypted control word on the CI system. The control word is then encrypted once again on the CI before it is transmitted to the STB. The key used for the second encryption is setup between the CI system and the STB through a one time synchronization. This means that users are not able to grab the clear control word and redistribute it. [14, pp. 12–13]

Another security issue that you need to think of when designing the hardware, is to make sure that no contacts are ever accessible from the top layer of the circuit board. This is due to the fact that people would be able to connect hardware to the board and download the material that way, if they were. There also exists a need to be aware of people trying to break the algorithm through forced ways, as well as control word sharing and hardware methods of stealing content.

### 4.3.1 Breaking the CSA

There are a few standard ways to try to break ciphers. The most common ones are the brute force-, known plaintext-, chosen plaintext- and birthday attacks. You choose what method to use depending on what the design of the cipher is. The

most relevant ones, in the context of the CSA, will be explained in the following subsections. [13, pp. 31-34]

### Brute force

The number of unique keys that can be extracted depends solely on the length of the key. The number of combinations corresponds to the largest number, plus one. The formula for the largest possible number obtainable, when working on keys represented as binary numbers, where the key-length is represented by the letter  $n$ , can be viewed in equation 4.1. Note that the key-length is given in number of bits.

$$\text{Largest possible number} = 2^n - 1 \quad (4.1)$$

Since the CSA uses keys consisting of 64-bits (8 bytes), this gives us  $18.5 \times 10^{18}$  possible keys. However, byte 3 and 7 are often used as parity bytes in CA systems, which leads to only 48 bits being used in the key. This can be seen in figure 4.1. 48 bits on other hand would lead to  $2^{48}$  combinations, which corresponds to  $281 \times 10^{12}$  possible unique keys.

Testing a million keys per second is about what is possible through a modern x86 processor using software methods. This means it would take roughly 3258 days to force break the keys, which translates into roughly 8.8 years. The calculations are done in equation 4.2 through 4.4. [19]

Number of unique keys, for a 48-bit key:

$$2^{48} = 2,814,749,7 \times 10^{14} \text{ keys} \quad (4.2)$$

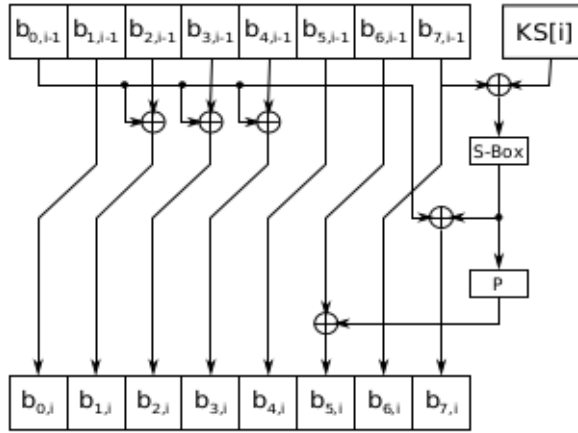
By dividing by the number of tested keys per second, the number of seconds to test all the keys will be found:

$$281 \times 10^{12} / 10^6 = 281,474,976,7 \times 10^6 \text{ seconds} \quad (4.3)$$

By substituting seconds with days  $\times (\text{seconds/day})$ , the number of keys per days is found instead  $10^6 / 86400 = 3257,8 \text{ days}$  (4.4)

Moreover, systems need to change the key at least every 120 seconds [17]. Changing the key every second minute would mean that 140 trillion keys would need to be scanned per minute, to cover the most of the keys in the two minutes available before the key is changed. However, most systems issues new keys between every 10th - 120th second, which means that for some systems 28.1 trillion keys need to be scanned per second [25].

It is possible to use dedicated hardware and FPGA implementations to speed up the force breaking methods through hardware acceleration. This could make it possible to scan through 2.8 trillion keys per second, just barely allowing us to be certain to find the key in two minutes. Even so, the key could be changed more



**Figure 4.1:** Number of bits in key used.

frequently than every second minute. As such, the brute force method of is not a reasonable method to obtain the key.

### Known plaintext attack

Known plaintext attacks are performed to figure out the key. What is interesting is that this kind of attack only is applicable for symmetric ciphers. That means that the known plaintext attack cannot be used to retrieve the secret key during public-key encryption. The key can then be used to decrypt following ciphertexts. To perform this kind of attack, a known plaintext-ciphertext pair is needed. You can try to find the key if you have the both of them. This is done by identifying ciphertexts known to correspond to zero-filled plaintexts, when trying to break the CSA [19]. Memories are then filled with precalculated keys, which are used to find which key the current plaintext-ciphertext pair corresponds to. This method is supposed to recover a key in roughly 7 seconds with a 97% certainty according to Tews et al. [19].

# 5

---

## CISSA or CSA3

There are currently two scrambling algorithms being assessed as replacements to the currently used DVB-CSA. A replacement is needed to assure content security for yet another ten years. A part of this thesis has been to compare the two proposed algorithms and decide which one is the most suitable replacement. This chapter is a basic introduction to the two algorithms, and the algorithm implemented in this thesis is presented.

### 5.1 Replacements

CISSA is meant to be a hardware-friendly as well as software-friendly algorithm designed to allow descrambling to be made on CPU-based units such as computers, smart phones and tablets [5, p. 9].

CSA3 is a hardware-friendly, software-unfriendly scrambling algorithm chosen by the ETSI to replace the currently used CSA [5, pp. 6–7]. Software-unfriendly means that descrambling is designed in such a way that it is highly impractical to perform in software.

Both of the algorithms are to be implemented in hardware for scrambling of data. The difference is that CSA3 is to make it hard to descramble the material using software. Since both of the algorithms are confidential, it is sadly impossible to find out what makes the CSA3 algorithm software-unfriendly, while the CISSA algorithm is software-friendly.

## 5.2 CISSA

CISSA stands for *Common IPTV Software-oriented Scrambling Algorithm* and is designed to be software-friendly. Opposite to the CSA3, CISSA is made to be easily descrambled in software, so that CPU-based systems such as computers and smart-phones can implement it. Although it is software-friendly, it is supposed to be able to be implemented efficiently on hardware as well as in software [5, p. 9].

CISSA is to use the AES-128 block cipher in CBC-mode with a 16 byte IV with the value 0x445642544d4350544145534349535341. Each TS packet is to be processed independently of other TS packets, but each block of data in the payload depends on the previous blocks of data in the same payload, except the first block of data, which depends on the IV. Both the header and adaptation field are to be left unscrambled. [5, p. 11]

### 5.2.1 Software friendly

An FPGA implementation of the CISSA algorithm is implementable, due to the fact that the scrambling of the content is supposed to be made in hardware, even though the descrambling is supposed to be made either in hardware or software.

While having a scrambling algorithm designed to enable viewing on CPU-based units opens up the market for more users, it might increase the risk for algorithm theft. Since reverse-engineering is possible for software implementations, one might find the algorithm for descrambling, as well as scrambling through inversion of the algorithm. Knowing the algorithm enables cryptanalysts to search for weaknesses in the algorithm, with the purpose of breaking it.

"A cryptosystem should be secure even if everything about the system, except the key, is public knowledge." according to Kerckhoffs's Principle. This means that the only result of having a descrambling method suited for hardware as well as software implementation should possibly only result in some free implementations showing up. But it being implemented in software should not lead to any problem.

## 5.3 CSA3

The CSA3 scrambling algorithm is based on a combination of an Advanced Encryption Standard (AES) block cipher using a 128-bit key, which is called the AES-128, and a confidential block cipher called the XRC [5, p. 8]. XRC stands for eXtended emulation Resistant Cipher and is a confidential cipher used in DVB [5, p. 8].

### 5.3.1 Hardware friendly

The CSA3 is designed to be hardware-friendly, meaning that descrambling through software methods is supposed to be next to impossible. Using a software-hostile



descrambling algorithm means that reverse-engineering and algorithm theft becomes hard, if even possible. Even though it would decrease the probability of content theft, it closes the door to expansion onto the CPU-based units market, which is becoming larger and larger.

## 5.4 Selection of the algorithm

CSA3 implements the AES-128 cipher for scrambling, combined with a confidential cipher, called the XRC cipher. CISSA does not on the other hand contain any confidential cipher. CISSA uses the AES-128 cipher in CBC-mode with a static IV [5].

CISSA sounds like a great idea, since it would allow CPU-based units to descramble data streams without using a dedicated HW-Chip. Regardless of which cipher is the best, or will prove to become the next standard, both of them use AES-128 as a building block. Therefore, starting out with an AES-128 cipher provided for a basis to continue developing the scrambler towards either CISSA or CSA3 on a later stage. The algorithm that was finally chosen was the CISSA algorithm due to three reasons. Firstly, software descrambling seems to be the future of content protection. Secondly, CISSA was a free and open algorithm. Finally, AES-128 in CBC mode (which is basically CISSA) is needed in order to use CI+ [10, p. 15].



# 6

---

## Advanced Encryption Standard

Both CSA3 and CISSA use the block cipher called AES, which will be explained in this chapter. The standard was determined by NIST in November 2001. AES is a symmetric block cipher which uses itself of key lengths of either 128, 192 or 256 bits. It is based on an SP-network which is fast in both hardware and software.

### 6.1 Introduction

The Rijndael cipher, which is used in AES, has key-sizes of at least 128 bits. The block length is 128 bits. It uses 8 to 8 bit S-boxes and a encryption is made with a minimum of 10 rounds of repetition [18, p. 79]. A round can corresponds to one iteration of a certain part of the algorithm, and uses a subkey to the provided key. The keys used in the rounds are called round keys. It is a symmetric-key algorithm with a fixed block size of 128 bits, where the key-size can vary between 128, 192 or 256 bits. The number of cycles needed to convert the plaintext into ciphertext depends on the size of the key. The 128-bit key requires 10 cycles of repetitions (rounds). The 192-bit key requires 12 rounds and the 256-bit key requires 14 rounds. [18, p. 103]

### 6.2 Method

The AES consists of a number of steps that are repeated for each block to be encoded. All of the steps are explained later in this chapter. The steps to be performed are, according to Stinson [18]:

*Set-up steps*

1. KeyExpansion - Produce round keys.
2. InitialRound - Combine each byte of the state with a byte of round key.

*Steps performed during the rounds*

1. SubBytes - Each byte is *substituted* using the Rijndael's S-box.
2. ShiftRows - The rows of the state matrix are *permuted*.
3. MixColumns - The columns of the state matrix are multiplied with a matrix.
4. AddRoundKey - The state matrix is once again combined with round-keys.

*In the final round everything except the MixColumns step are performed. Meaning that SubBytes, ShiftRows and AddRoundKey are performed.*

The ciphertext is then defined as the state-matrix [18, p. 103]. As mentioned in section 3.5 (Confusion and Diffusion), both confusion and diffusion are necessary to ensure a secure encryption. They can be seen in the SubBytes and ShiftRows steps above. These steps also performs whitening, which strengthens the cipher. Whitening is, as mentioned in 3.5, performed through an XOR between the roundkey and the data.

The KeyExpansion is explained in section 6.3.

Before anything can be done, the data needs to be put into a state-matrix, which can be seen in figure 6.1.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \quad (6.1)$$

**Figure 6.1:** State-Matrix

### 6.2.1 InitialRound

This is an initial AddRoundKey which is explained in section 6.2.5.

### 6.2.2 SubBytes

In the SubBytes step, each byte is sent to a Rijndael S-box (which is basically a lookup table, see figure A.1 in appendix A) where they are substituted in a non-linear fashion. This gives us a substituted state matrix.

### 6.2.3 ShiftRows

The next step is called the ShiftRows step, which left-shifts the rows  $n-1$  steps where  $n$  is the index of the row. This means that the first row is left as it is, the second row is shifted one step, the third row is shifted two steps, and the fourth row is shifted three steps. The data is shifted cyclically, meaning that data which

is shifted out of the left side of the state-matrix is shifted back in from the right side.

### 6.2.4 MixColumns

All of the multiplications performed in the MixColumns steps are take place in the Galois Field, which is why a lot of it might seem illogical at first.

In the MixColumns step, the four bytes of each row are combined through a matrix multiplication. The MixColumns function takes four bytes as input and multiplies them with a fixed matrix (figure A.3 in appendix A). While this might seem simple to do, it actually is not. The multiplication makes sure that each input byte affect all output bytes. [8]

The matrix is multiplied with the vector from the left, ( $4 \times 4 \times 4 \times 1 = 4 \times 4 \times 4 \times 1 = 4 \times 1$ ) where the vector is a column from the state-matrix. Multiplication with 1 means that the value is left untouched. Multiplication by 2 means left shift, then an XOR with 0x1B if the shifted value exceeds 0xFF. Multiplication with 3 is done in the same way as a multiplication with 2, except that the result after the shift and conditional XOR are then XOR:ed with the input value of the multiplication. All of the resulting values are then XOR:ed, leaving us with the result. All additions are replaced with XOR, since the calculations take place in the Galois Field ( $\text{GF}(2^8)$ ).

### 6.2.5 AddRoundKey

Each of the 16 bytes of the state are combined with a byte from the round key using a bitwise XOR. They are then combined to a state matrix (Figure A.2 in Appendix A) containing  $4 \times 4$  bytes.

## 6.3 KeyExpansion

To generate round keys from the provided key, the Rijndael's key schedule is used. The schedule consists of a couple of loops and a key- schedule core. The schedule core is the part that branches out if  $c \bmod 16$  is zero. The flowchart explaining the entire KeyExpansion can be viewed in Figure B.2 in Appendix B. To change the key schedule to fit a key size of 192 bits, you simply change the value that  $c$  is compared to in the first branch in the flowchart from 176 to 206.

This is done since AES requires a separate 128-bit (16-byte) round key for each round, plus one extra key for the initialization which means that the AES-128 requires 176 bytes, since AES-128 consists of 10 rounds.

### 6.3.1 Key-schedule core

The key-schedule core takes an input of 4 bytes (32 bits) which it then rotates 1 byte (8 bits) to the left. Let us say that our key is *AB CD EF 01*. This would give us the key *CD EF 01 AB* after the rotation. This operation is also called the RotWord- operation [18, p. 107]. The next step is to apply Rijndael's S-box to

each of these bytes, giving us 4 new bytes. The bytes AB CD EF 01 would give us 62 BD DF 7C, when substituted according to the Rijndael S-box (Figure A.1 in Appendix A).

The left-most byte is then XOR:ed with a value from the Rcon function depending on what round you are currently processing. You can read more about the Rcon function in section 6.3.3.

### 6.3.2 Rijndael's S-Box

Rijndael's S-box takes an input byte which it transforms according to a LUT (Figure A.1 in Appendix A). Where the most significant nibble is located on the Y axis, and the least significant nibble is located on the X axis of the table. Given the input *0x31*, the output *0xC7* would be received from the Rijndael's S-box.

### 6.3.3 Rcon

The value input into the Rcon function depends on what round you are currently at. Which means that you would choose Rcon(1) for the first round, Rcon(2) for the second round, and so on. The values in the Rcon array are calculated mathematically, but might as well be accessed from a vector, such as the one found in Figure A.4 in Appendix A.

The steps to be performed in the Rcon function are illustrated in a flowchart and can be viewed in figure B.1 in appendix B.

If the input value is 0, the output value is 0, otherwise the following steps are performed [24]. This can also be replaced by an S-box where you input your byte, and get another back, since the input byte is just used as a counter that decides how many times you perform steps 2 through 6

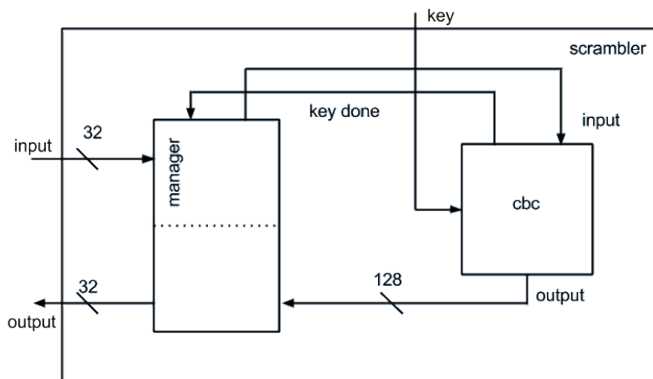
1. Set a variable *c* to 0x01.
2. If the input-value does not equal 1, set variable *b* to *c* & 0x80. Otherwise, go to 7.
3. Left shift *c* one step.
4. If *b* is equal to 0x80 proceed to 5, otherwise go to 6.
5. Store the result of a bitwise XOR between *c* and 0x1B in *c*.
6. Decrease the input value by one, then go back to 2.
7. The output is set to *c*.

# 7

## Implementation

This design is hierarchical. The top layer is an AES128 block in CBC-mode. It takes an input TS-packet, selects data from it which it scrambles and outputs the data in the form of a TS-packet once again.

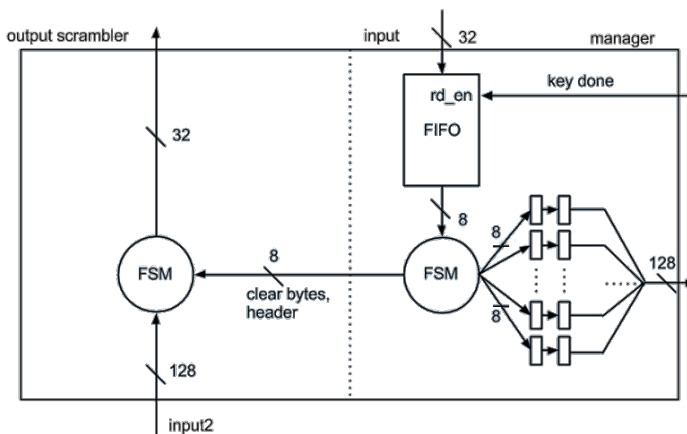
The scrambler (Figure 7.1) consists of two entities. An entity which is called the CBC-entity, which deals with the scrambling of the received data. The other entity is a data-manager. The manager deals with reading data from the interface towards the rest of the FPGA as well as sending data-bits to the CBC-entity at the correct time. It also tells the CBC-entity how to handle the data, since different tasks are to be done depending on if the data is the first data packet sent or not.



**Figure 7.1:** Scrambler-block.

## 7.1 Manager entity

The manager (Figure 7.2) consists of a FIFO (First in, First out), an FSM (Final State Machine) and a couple of registers. The FIFO is needed since the data sent to the scrambler from the FPGA is sent in bursts. The FIFO therefore writes the data bursts into a memory, from which it later reads, processes and sends the data to the CBC-entity. The data written to the FIFO is written in packets of 32 bits, but are read 8 bits at the time. The manager looks through the data packets to see if there is an adaptation field or not, since that changes the way that the data is handled. The payload is written to the first set of registers as the data is found, and then sent to the next set of registers. This is done to allow the manager to deal with two sets of data in parallel. However, only one packet is scrambled at any given time. When the packet is ready to be sent, a flag is set and the data is sent to the CBC-entity. The output of the registers is the input of the CBC-entity, which can be seen in figure 7.1.

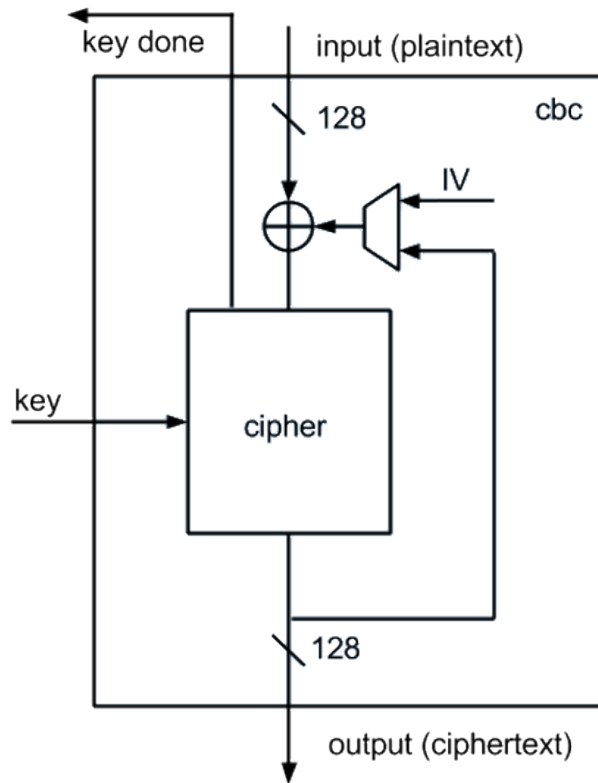


*Figure 7.2: Manager-block.*

## 7.2 CBC entity

The CBC-entity (Figure 7.3) consists of three small entities. An XOR, a multiplexer and a cipher-entity. The multiplexer is needed since the first plaintext should be sent to the XOR together with an IV. For the rest of the plaintexts contained within the same TS packet, the output ciphertext should be used instead of the IV. There is only going to be one AES128 cipher in the CBC-entity in order to save hardware. It will be run in sequence instead of in parallel, even though it might reduce the maximal speed of the circuit. CBC is explained in section 3.4.1.

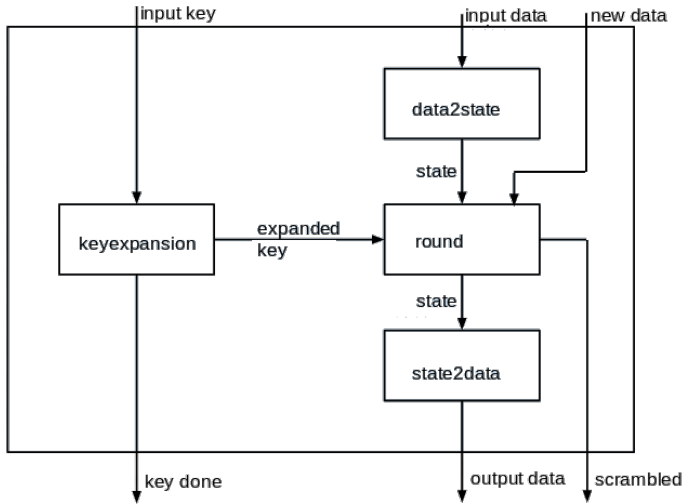




*Figure 7.3: CBC-block.*

## 7.3 Cipher entity

The AES128 cipher-entity (Figure 7.4) consists of 4 components. The data2state entity, which transforms the array into a matrix of data. A keyexpansion entity, which takes an input of a key, and generates an extended key as an output. An entity, which was named rounds, that deals with the encryption of the 16 byte data blocks. And finally a state2data entity, which transforms the data- matrix into an array once again. The cipher entity itself keeps track of timing mainly between the keyexpansion and the round entity. It uses an FSM to make sure that the round entity is provided with the correct roundkey at the right time, and data is output when it is scrambled. What can not be seen in figure 7.4 is that the keyexpansion entity also sends an enable signal, that tells the cipher entity that the expanded key is complete.



*Figure 7.4: Cipher-block.*

## 7.4 Keyexpansion entity

The keyexpansion-entity (Figure 7.5) is divided into three keyblock entities. The first keyblock entity decides what four bytes of the expanded key are to be expanded. The first time, the bytes are selected from the provided key, but after the first key has been expanded, the data bits are chosen from the newly expanded key. The second keyblock entity (Figure 7.6) contains the keycore, which is only performed on every fourth set of four bytes, and a demux entity. The third keyblock entity performs an XOR between either the first or second keyblock depending on if the keycore was supposed to be run and the key. It also increments the internal counter, which is used as an index when accessing and generating the 4 byte blocks of data.

The FSM seen in figure 7.5 keeps track of when the key generation is done, and produces a lock signal at that time. The lock signal is used by keyblock3 to produce the done signal, that is passed to other entities. The FSM also keeps track of when a new key is received, and forces a reset of keyblock2 and keyblock3, since they are not entirely combinatorial. The internal reset signal, `reset_i`, forces a reset of keyblock2 and keyblock3.

### 7.4.1 Keycore entity

The keycore entity consists of four entities. Rotword, Sbox, Rcon and a counter. The counter is used to get the correct output from the Rcon entity, and the index is only used in the keycore, and is thus best suited to be placed inside the keycore entity. Rotword rotates the bytes of the input one step to the left through a simple left shift. The Sbox replaces the input bytes according to the Rijndael

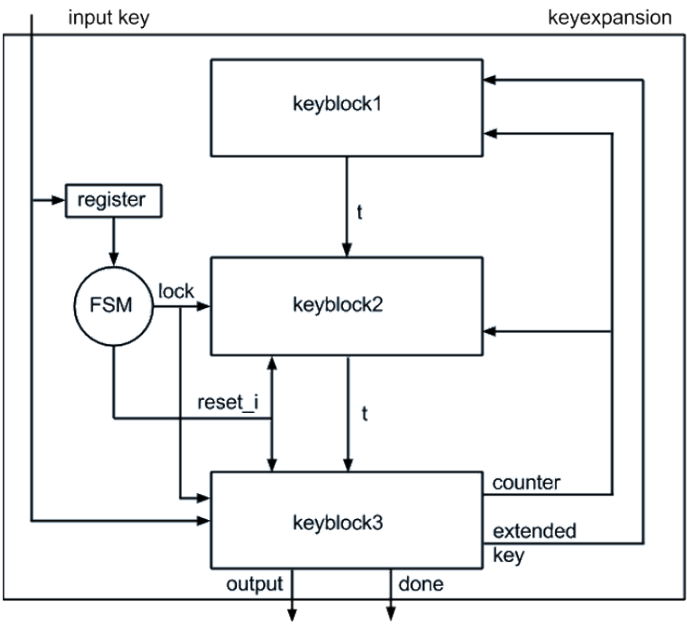


Figure 7.5: Keyexpansion-block.

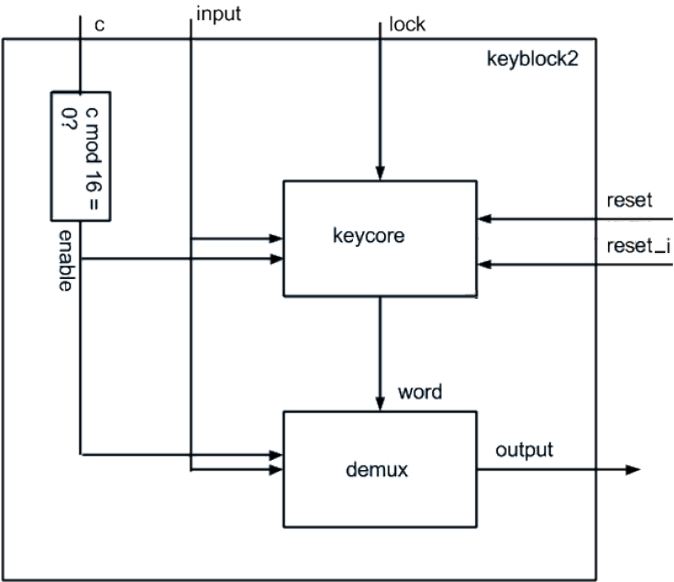


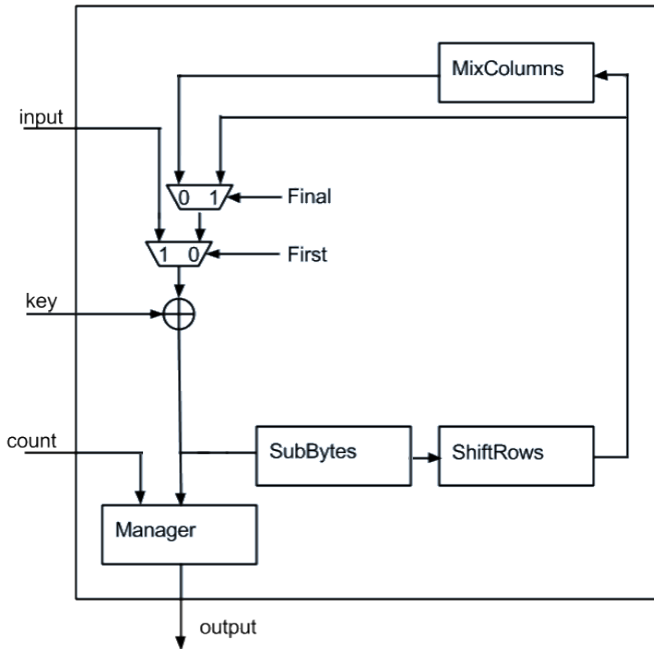
Figure 7.6: Keyblock2-block.

Sbox, through a LUT. The Rcon entity both collects the correct rcon value from a

precalculated vector, as well as inputs it into an xor together with the input.

## 7.5 Round entity

The round-entity (Figure 7.7) consists of four entities, which are called Subbytes, shiftrows, mixcolumns and addroundkey. Addroundkey is a special XOR, which changes input depending on what round is being processed. Subbytes is an Rijndael Sbox which takes an input 16-byte state, substitutes it and outputs another 16-byte state. Shiftrows shifts the rows of the second, third and fourth row of the state. Last, but not least, is the mixcolumns entity. It consists of 16 mulblock entities. The input state of mixcolumns is split into columns, and each column is sent to a mulblock entity, which multiplies the inputs with 1, 2 or 3, then performs a bitwise XOR on them and outputs the result of the XOR. The function of the mixcolumns block is a complex matrix multiplication.



*Figure 7.7: Round-block.*

### 7.5.1 Addroundkey entity

Addroundkey is an entity which takes different inputs depending on what round is currently being dealt with. On the first round, Addroundkey takes the input to the round entity. On the last round, it takes the output from the ShiftRows entity. The input to addroundkey is the output from MixColumns the rest of the time.

### 7.5.2 The mulblock entity

The mulblock entity is the multiplication used by the MixColumns entity. It consists of one mul3 entity and one mul2 entity, which performs a special kind of hardware multiplication of three, and two, on the input. It also takes two inputs which it does not process, those are the values multiplied by one. The four results are then XOR:ed with eachother, and returned to the mixcolumns entity. The result is then input into the correct index in the matrix.

Mul3 means multiplication with 3, and mul2 means multiplication with 2. A multiplication with 2 is a left-shift, followed by an XOR with the fix value 0x1B if the shifted value exceeds 0xFF. A multiplication with 3 is the same as a multiplication with 2, followed by an XOR with the input value. The choice of using an XOR instead of additions, and the XOR with 0x1B are explained in section 6.2.4.



# 8

---

## Result

The focus of this thesis has been to minimize the amount of hardware usage, while trying to meet the timing constraints provided from the rest of the circuit. Reaching a throughput of 1 Gbit/s was sufficient for the current design.

The implemented circuit was a scrambler, which can be found in the head-end of DVB systems. An analysis of two algorithms was done, and the AES128 algorithm, in CBC-mode, was chosen. It corresponds to the CISSA algorithm, which is designed to be software-friendly.

AES128 processes 16 bytes of data in 11 clock pulses with a clock frequency of 94MHz, which would correspond roughly to a throughput of 1.16 Gbits/s. The frequency of the design is further discussed in section 8.3.2. The scrambler needs to process the key first, before being able to scramble data. A keyexpansion takes roughly 45 clock pulses, and is only performed when a new key is sent, which is very seldom. The scrambler then deals with 16 bytes of data on 13 clock pulses, but outputs 1 byte of data per clock cycle. This is done so that one byte of data from the scrambled package is read into a register on every clock pulse. When four bytes are collected the 32-bit output is sent out. 32-bits are processed at a time, since the data-bus is a 32-bit bus.

### 8.1 Problems

The main problems encountered were:

- Not possible to get the license for CSA3.
- Small interest in CSA3 from customers.

- Next to no documentation of the CISSA algorithm.
- Hard finding reliable test vectors.
- Merging.
- Timing.

### **License and interest of the CSA3**

When the Thesis was first started, the idea was that the CSA3 algorithm was to be implemented. However, licensing problems, and the fact that AES-128 in CBC-mode seemed like a better idea, led to a rework of the project. Also, the interest in CSA3 from potential customers was small, while the interest in AES128 in CBC-mode was great.

### **Documentation of CISSA**

The only mention of the CISSA algorithm found in official documents has been found in ETSI TS [5]. What can be found from this documentation is that CISSA uses the AES128 cipher in CBC-mode encryption, with a static IV. While this has been sufficient to implement the algorithm, more documentation would have been good.

### **Test vectors**

Finding test vectors for the different blocks of the scrambler has been hard. It has been possible to find a number of them through the ETSI documentation though, which has been used to determine the functionality of the blocks.

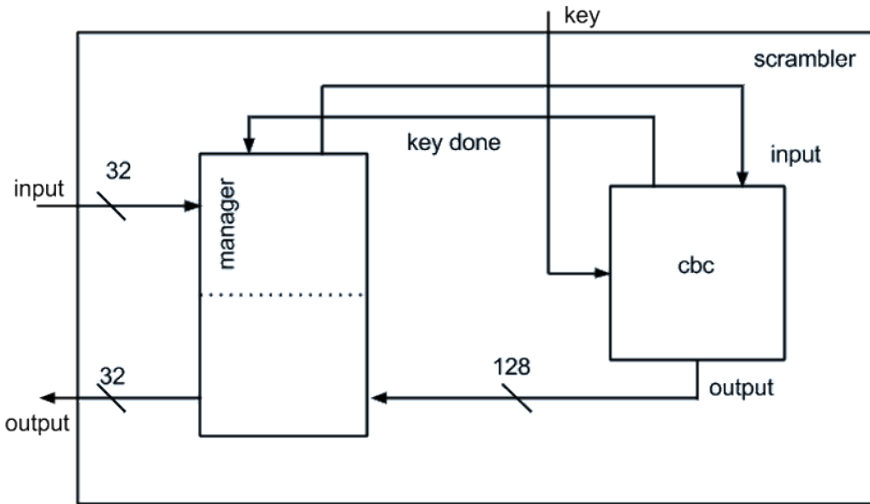
### **Merging and timing**

Merging has required more focus than expected, probably since this design was a bottoms-up design, instead of the more common top-down design. This project was done by implementing low level entities first, that were to be used in higher hierarchies. Doing this caused some problems when merging entities into higher level blocks, since some signals, needed to be produced. This was not a huge problem, and only occurred on a few instances, but were rather troublesome at those times.

The pro of this method has been that it produced results quickly. The con is that a large portion of the time has been spent on going back to entities that were already functional, and reworking them by adding signals, and finding the right timing conditions to make sure that they provided necessary information for entities higher up in the hierarchy.

Since the plan was to optimize this implementation to just meet the demands on speed, while trying to minimize the amount of hardware needed, timing was introduced into a few circuits that could have otherwise been completely combinatorial. This has, as expected, introduced quite a bunch of timing-issues. All of them appear to be gone now. It is however hard to know, without performing more exhaustive testing of the system.





*Figure 8.1: The top entity*

The solution described in this thesis includes the entire hardware usage, which includes the interface towards the FPGA, which is one of the reasons why it might appear large, when compared to other implementations.

## 8.2 Hardware

The top entity can be viewed in Figure 8.1, and figures of the rest of the entities are placed near the explanation of the entities in the following sections.

### 8.2.1 Hardware usage

The circuit that was first implemented has during the course of the project been optimized in a couple of ways it was synthesized. It has either been optimized to minimize hardware usage, or maximize the speed of the circuit. A total of six syntheses were run. A table displaying the differences of the results can be viewed in figure 8.2. Most focus has been put on minimizing the amount of registers. This has been done by adding control signals, that decrease the need to store values in registers. The most significant optimizations are discussed in this section.

#### Optimizations

The most significant optimization that was performed was an optimization of keyblock3. Keyblock3 is located in the keyexpansion entity which is explained in section 7.4 and can be seen in Figure 7.5. That entity used a lot of the provided resources. It was noticed that the circuit waited for the expanded key to become

completely filled before updating the output at one point. The expanded key was stored in a vector located in keyblock3 while it was being updated. However, the expanded key used by other entities was not updated until the entire expanded key located in keyblock 3 was completely filled. Rewriting the code decreased the amount of registers by 176 8-bit registers, which could be removed by adding an enable signal. The enable signal tells the other entities when the expanded key is ready for usage.

The third synthesis was performed on each block separately, to find out where further optimization would yield the biggest improvement. The hardware usage can be viewed in Table 8.1.

Entity	Slice LUTs out of 63288	Slice Registers out of 126576
scrambler	5167	2817
▷manager	858	699
▷cbc	4321	2127
▷cipher	4229	1994
▷keyexpansion	2914	1601
▷keyblock1	689	0
▷keyblock2	208	9
▷demux	32	0
▷keycore	183	9
▷ctr	14	9
▷rotw	0	0
▷sbox	128	0
▷rcon	40	0
▷keyblock3	1854	1365
▷data2state	0	0
▷round	1535	272
▷subbytes	512	0
▷shiftrows	0	0
▷mixcolumns	176	0
▷addkey	128	0
▷state2data	1	2

**Table 8.1:** Hardware usage of entities from synthesis number 3.

The final place and route yielded the following results:

Number of Slice Registers: 2,805 out of 126,576      2%  
 Number of Slice LUTs:      4,930 out of 63,288      7%

### Comparison of synthesis results

The most interesting numbers from the synthesizes are the number of LUTs used, number of ROMs (Read-only Memories) and number of Flip-Flops. However, only one ROM has been implemented in this design, which was present in all synthesizes, which is why it will not be included in the comparison.

	Second	Third	Fourth	Fifth	Final
LUTs	5121	5167	5167	5167	5167
Registers	4537	2945	2817	2817	2818

*Table 8.2: Synthesis results*

All of the synthesis results can be viewed in Appendix D.

There is a difference of one register between the fifth and sixth synthesis reports. That register is most likely a residue after an attempt to reduce the critical path, and can be disregarded.

## 8.3 Further development

There are, an amount of optimization that could be performed on the circuit. They consist of optimization of code, as well as some deeper research into how to rewrite VHDL code to turn the registers in this implementation into RAMs, ROMs or LUTs. By rewriting the code into those entities instead of the currently implementation, which consists of a network of multiplexers, would be a reduction in the critical path as well as a reduction in the number of used slices.

### 8.3.1 Rijndael's S-Box

The Rijndael S-box implemented in this design does not synthesize into a ROM, which it should be able to do. Other than a ROM, it should also be able to be synthesized into a couple of LUT6. It has not been possible to find out why the code was implemented into registers instead of more efficient solutions.

Both registers and ROMs are viable implementations for the S-box. However, for area minimization a ROM might be more favourable, while the registers, being clocked, help reduce the critical path of the signals.

### 8.3.2 Critical Path

To increase the maximum frequency of the circuit, the critical path needs to be decreased. This is done by adding FFs in the middle of the critical path. This will be hard to solve, due to the complexity of the keyexpansion, and would increase the amount of hardware as well as the complexity of the circuit if FFs were to be added.

The decision whether to reduce the critical path or not is a hard decision due to the amount of hardware that might need to be added to increase the frequency. However, since LUTs and FFs are located on same slice, it is not sure whether the design would use more slices or not. A slice is a section of the FPGA, which consists of some basic logic.

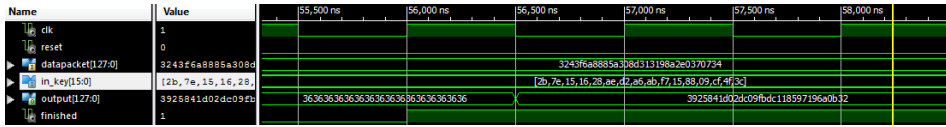


Figure 8.2: Test vector 1

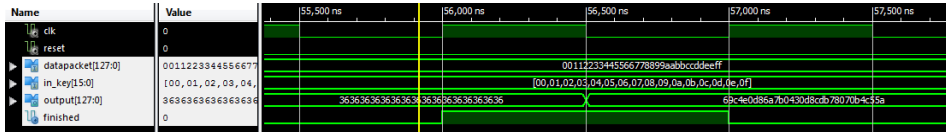


Figure 8.3: Test vector 2

## 8.4 Tests

All of the entities in the design have been simulated and evaluated separately before being merged and tested together, to make sure that they have the desired functionality both separately and when combined together. The simulations of the separate blocks are trivial, and therefore not included in the report.

Figure 8.2 through 8.4 are tests performed on the complete AES128 block, before CBC-mode. In the figures, `in_key` is the input key to be extended and used, and `datapacket` is one packet from a TS. Test vector 1 and 2 are taken from NIST [12], while test vector 3 is generated using webpage [? ].

### Test vector 1 (Figure 8.2)

Input key: 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Plaintext: 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Ciphertext: 39 25 84 1d 02 dc 09 fb dc 11 85 97 19 6a 0b 32

### Test vector 2 (Figure 8.3)

Input key: 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

Plaintext: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

Ciphertext: 69 c4 e0 d8 6a 7b 04 30 d8 cd b7 80 70 b4 c5 5a

### Test vector 3 (Figure 8.4)

Input key: 10 20 30 40 50 60 70 80 90 a0 b0 c0 d0 e0 f0 bb

Plaintext: 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

Ciphertext: bf 99 1f aa 8b 0f e6 48 36 46 a0 2d 33 9e de a5

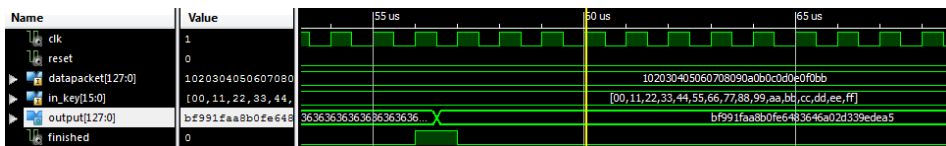


Figure 8.4: Test vector 3

## 8.5 Comparison to other implementations

Since there exist more implementations of the AES128 algorithm, a comparison between the implementations is interesting. To be noted is that this implementation has been focusing on minimizing the hardware usage, while achieving a throughput of at least 1 Gbit/s. Since it is possible to achieve an entirely combinatorial AES128 scrambler, as well as a pipelined version, the focus has been to try to find an implementation as similar as this one as possible to compare the results.

Note that this implementation has got a couple of entities, that are usually not present in scramblers. They are there in order to allow insertion of the implementation into the FPGA used by WISI.

The design implemented in this thesis scrambles a block of 16 bytes of data in 13 clock pulses. It uses 4930 LUTs, occupies 1727 slices and can be run at a maximum frequency of 94 MHz. The circuit is implemented on a Xilinx Spartan-6. This can be compared to the Fast AES XTS/CBC implementation (Helion) made by Xilinx. It uses 1041 slices and 4047 LUTs with a maximum frequency of 130 MHz. [26]

The implementation by Xilinx uses custom FPGA optimization techniques through hand crafted macros. A comparison between the two implementations can be found in table 8.3. When these two implementations are compared, the most important difference is the frequency at which the two circuits can be run. The amount of LUTs that are used does in fact not mean much, since a LUT is said to be used even if just a small part is used in the circuit. If only one LUT in a slice is used, the LUT is still considered used.

The percentage row in table 8.3 indicates how many more percent LUTs and slices that are used in this implementation compared to the Xilinx implementation. However, the frequency shows how much higher the frequency of the Xilinx implementation is.

	<i>Slices</i>	<i>LUTs</i>	<i>Frequency(MHz)</i>
<i>Current</i>	1727	4930	94
<i>Xilinx</i>	1041	4047	130
<i>Percentage</i>	65%	21.8%	27.7%

**Table 8.3:** Comparison between implementations.

## 8.6 Discussion

One of the first things noticed during this thesis was that industrial secrecy can put a quick halt to projects. A license had to be written and approved by ETSI before WISI Norden was allowed information about the specifications of one of the algorithms that were supposed to be analyzed. Due to restrictions, this could

not be done, meaning that this specific analysis came to a halt before it even started. This led to the comparison between a software- and hardware-friendly algorithm becoming impossible to do.

While WISI Norden and ETSI were discussing the license, specifics about the CSA3 and CISSa algorithms were investigated, since those were the ones to be analyzed. From the little information available about the CSA3, only the names of the two ciphers were possible to find. Since one of the two ciphers in the CSA3 algorithm corresponded to one of the ciphers in the CISSa algorithm, this cipher was examined as much as possible. This was the AES128 cipher.

Both the key generation and the functionality of the entities in AES-128 algorithm could be found in literature, since the AES encryption is a public algorithm. From what could be found out about the CISSa algorithm, through an official ETSI journal, it seemed to just use the AES-128 algorithm, but in a certain mode, with a specific Initialization Vector.

## 8.7 Conclusions

An analysis of two algorithms proposed to replace the current standard scrambling algorithm CSA has been done. This has been done through a study of literature and documents defining the algorithms. The system was built using a bottom-up design, after finding basic information about the components of the algorithm.

As explained in chapter ??, the lack of information available about the two algorithms made an analysis of what made one of them hardware friendly and the other software friendly impossible. However, CSI+ requires the scrambling to be done using the AES128 algorithm in CBC-mode, and therefore CISSa was chosen to be implemented.

As mentioned in chapter 8.2.1, optimization of the circuit has been mainly done by minimizing the usage of registers. This has been done by increasing the amount of signals. Most decisions were made through analysis of simulations and synthesis reports.

The biggest difficulties encountered during this thesis has been finding proper documentation. Moreover, as mentioned in 8.1, merging has taken more time and focus than was planned for.

# **Appendices**





# A

---

## Matrixes

This appendix contains a selection of vectors and matrixes used in the thesis. The contents found in this appendix take too much space, and are not relevant enough to be inserted into the chapters of the thesis.

All of the matrixes are defined as they are written here, and do not differ.

<i>Nibble</i>	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
10	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
20	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
30	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
40	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
50	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
60	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
70	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
80	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
90	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A0	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B0	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D0	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E0	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F0	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(A.1)

**Figure A.1:** Rijndael S-box.

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \quad (\text{A.2})$$

**Figure A.2:** State-Matrix.

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{1,i} \\ a_{2,i} \\ a_{3,i} \\ a_{4,i} \end{bmatrix}, i = \{1, 2, 3, 4\} \quad (\text{A.3})$$

**Figure A.3:** Rijndael MixColumns equation.

---

$Rcon[256] = \{$ 

8D	01	02	04	08	10	20	40	80	1B	36	6C	D8	AB	4D	9A
2F	5E	BC	63	C6	97	35	6A	D4	B3	7D	FA	EF	C5	91	39
72	E4	D3	BD	61	C2	9F	25	4A	94	33	66	CC	83	1D	3A
74	E8	CB	8D	01	02	04	08	10	20	40	80	1B	36	6C	D8
AB	4D	9A	2F	5E	BC	63	C6	97	35	6A	D4	B3	7D	FA	EF
C5	91	39	72	E4	D3	BD	61	C2	9F	25	4A	94	33	66	CC
83	1D	3A	74	E8	CB	8D	01	02	04	08	10	20	40	80	1B
36	6C	D8	AB	4D	9A	2F	5E	BC	63	C6	97	35	6A	D4	B3
7D	FA	EF	C5	91	39	72	E4	D3	BD	61	C2	9F	25	4A	94
33	66	CC	83	1D	3A	74	E8	CB	8D	01	02	04	08	10	20
40	80	1B	36	6C	D8	AB	4D	9A	2F	5E	BC	63	C6	97	35
6A	D4	B3	7D	FA	EF	C5	91	39	72	E4	D3	BD	61	C2	9F
25	4A	94	33	66	CC	83	1D	3A	74	E8	CB	8D	01	02	04
08	10	20	40	80	1B	36	6C	D8	AB	4D	9A	2F	5E	BC	63
C6	97	35	6A	D4	B3	7D	FA	EF	C5	91	39	72	E4	D3	BD
61	C2	9F	25	4A	94	33	66	CC	83	1D	3A	74	E8	CB	8D

 $\}$

**Figure A.4:** The Rcon function represented as a vector.

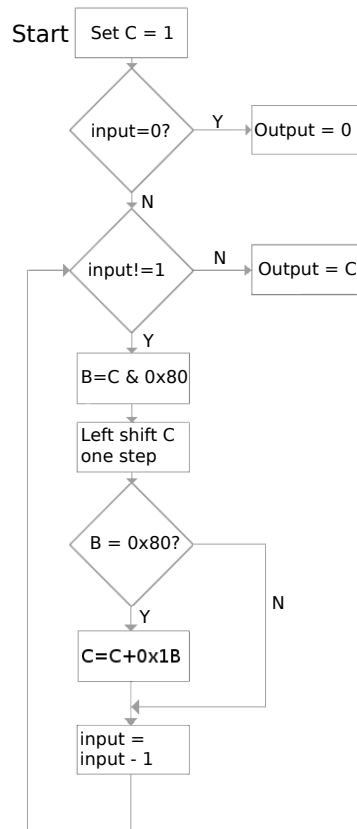


# B

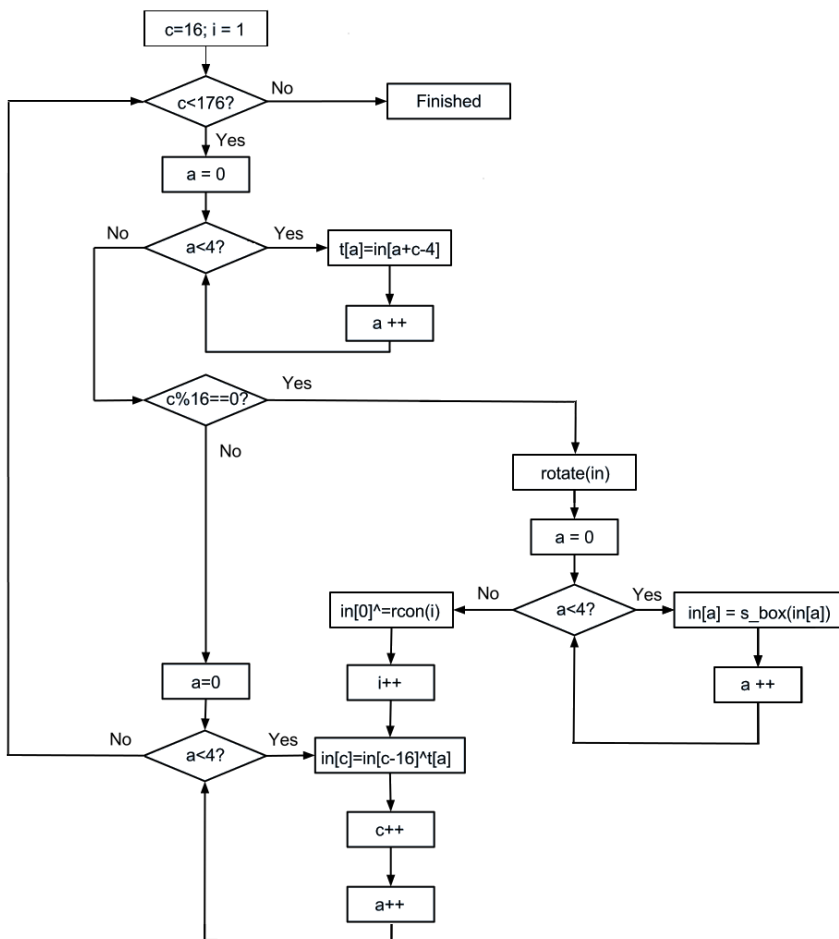
---

## Flowcharts

This appendix contains flowcharts that describe the extraction of the value from the rcon function, and how to perform a keyexpansion. They are placed in this appendix instead of the thesis since they can be used for reference, but are not necessary to understand how the results are obtained.



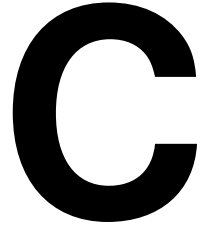
**Figure B.1:** Flowchart of the Rcon function



**Figure B.2:** Flowchart of the key schedule







---

## Test vectors

### Test cases

This section contains test cases, which can be followed one step at the time.

The following test case is taken from NIST [12, pp. 35–36]. The plaintext is input into a single AES128 cipher.

Plaintext: 00112233445566778899AABBCCDDEEFF  
Key: 000102030405060708090A0B0C0D0E0F

Cipher (Encrypt) :

round[0].input	00112233445566778899AABBCCDDEEFF
round[0].k_sch	000102030405060708090A0B0C0D0E0F
round[1].start	00102030405060708090A0B0C0D0E0F0
round[1].s_box	63CAB7040953D051CD60E0E7BA70E18C
round[1].s_row	6353E08C0960E104CD70B751BACAD0E7
round[1].m_col	5F72641557F5BC92F7BE3B291DB9F91A
round[1].k_sch	D6AA74FDD2AF72FADAA678F1D6AB76FE
round[2].start	89D810E8855ACE682D1843D8CB128FE4
round[2].s_box	A761CA9B97BE8B45D8AD1A611FC97369
round[2].s_row	A7BE1A6997AD739BD8C9CA451F618B61
round[2].m_col	FF87968431D86A51645151FA773AD009
round[2].k_sch	B692CF0B643DBDF1BE9BC5006830B3FE
round[3].start	4915598F55E5D7A0DACA94FA1F0A63F7
round[3].s_box	3B59CB73FCD90EE05774222DC067FB68
round[3].s_row	3BD92268FC74FB735767CBE0C0590E2D

round[3].m_col	4C9C1E66F771F0762C3F868E534DF256
round[3].k_sch	B6FF744ED2C2C9BF6C590CBF0469BF41
round[4].start	FA636A2825B339C940668A3157244D17
round[4].s_box	2DFB02343F6D12DD09337EC75B36E3F0
round[4].s_row	2D6D7EF03F33E334093602DD5BFB12C7
round[4].m_col	6385B79FFC538DF997BE478E7547D691
round[4].k_sch	47F7F7BC95353E03F96C32BCFD058DFD
round[5].start	247240236966B3FA6ED2753288425B6C
round[5].s_box	36400926F9336D2D9FB59D23C42C3950
round[5].s_row	36339D50F9B539269F2C092DC4406D23
round[5].m_col	F4BCD45432E554D075F1D6C51DD03B3C
round[5].k_sch	3CAAA3E8A99F9DEB50F3AF57ADF622AA
round[6].start	C81677BC9B7AC93B25027992B0261996
round[6].s_box	E847F56514DADDE23F77B64FE7F7D490
round[6].s_row	E8DAB6901477D4653FF7F5E2E747DD4F
round[6].m_col	9816EE7400F87F556B2C049C8E5AD036
round[6].k_sch	5E390F7DF7A69296A7553DC10AA31F6B
round[7].start	C62FE109F75EEDC3CC79395D84F9CF5D
round[7].s_box	B415F8016858552E4BB6124C5F998A4C
round[7].s_row	B458124C68B68A014B99F82E5F15554C
round[7].m_col	C57E1C159A9BD286F05F4BE098C63439
round[7].k_sch	14F9701AE35FE28C440ADF4D4EA9C026
round[8].start	D1876C0F79C4300AB45594ADD66FF41F
round[8].s_box	3E175076B61C04678DFC2295F6A8BFC0
round[8].s_row	3E1C22C0B6FCBF768DA85067F6170495
round[8].m_col	BAA03DE7A1F9B56ED5512CBA5F414D23
round[8].k_sch	47438735A41C65B9E016BAF4AEBF7AD2
round[9].start	FDE3BAD205E5D0D73547964EF1FE37F1
round[9].s_box	5411F4B56BD9700E96A0902FA1BB9AA1
round[9].s_row	54D990A16BA09AB596BBF40EA111702F
round[9].m_col	E9F74EEC023020F61BF2CCF2353C21C7
round[9].k_sch	549932D1F08557681093ED9CBE2C974E
round[10].start	BD6E7C3DF2B5779E0B61216E8B10B689
round[10].s_box	7A9F102789D5F50B2BEFFD9F3DCA4EA7
round[10].s_row	7AD5FDA789EF4E272BCA100B3D9FF59F
round[10].k_sch	13111D7FE3944A17F307A78B4D2B30C5
round[10].output	69C4E0D86A7B0430D8CDB78070B4C55A

Table C.1 displays a key expansion based on a test case taken from NIST [12, pp. 35–36].

Key = 2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

i(dec)	temp	After RotWord	After SubWord	Rcon(i)	After $\oplus$ with Rcon	w[i-16]	w[i] = temp $\oplus$ w[i - 16]
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafa17
5	a0fafa17					28aed2a6	88542cb1
6	88542cb1					abf71588	23a33939

7	23a33939					09cf4f3c	2a6c7605
8	2a6c7605	6c76052a	50386be5	02000000	52386be5	a0fafe17	f2c295f2
9	f2c295f2					88542cb1	7a96b943
10	7a96b943					23a33939	5935807a
11	5935807a					2a6c7605	7359f67f
12	7359f67f	59f67f73	cb42d28f	04000000	cf42d28f	f2c295f2	3d80477d
13	3d80477d					7a96b943	4716fe3e
14	4716fe3e					5935807a	1e237e44
15	1e237e44					7359f67f	6d7a883b
16	6d7a883b	7a883b6d	dac4e23c	08000000	d2c4e23c	3d80477d	ef44a541
17	ef44a541					4716fe3e	a8525b7f
18	a8525b7f					1e237e44	b671253b
19	b671253b					6d7a883b	db0bad00
20	db0bad00	0bad00db	2b9563b9	10000000	3b9563b9	ef44a541	d4d1c6f8
21	d4d1c6f8					a8525b7f	7c839d87
22	7c839d87					b671253b	caf2b8bc
23	caf2b8bc					db0bad00	11f915bc
24	11f915bc	f915bc11	99596582	20000000	b9596582	d4d1c6f8	6d88a37a
25	6d88a37a					7c839d87	110b3efd
26	110b3efd					caf2b8bc	dbf98641
27	dbf98641					11f915bc	ca0093fd
28	ca0093fd	0093fdca	63dc5474	40000000	23dc5474	6d88a37a	4e54f70e
29	4e54f70e					110b3efd	5f5fc9f3
30	5f5fc9f3					dbf98641	84a64fb2
31	84a64fb2					ca0093fd	4ea6dc4f
32	4ea6dc4f	a6dc4f4e	2486842f	80000000	a486842f	4e54f70e	ead27321
33	ead27321					5f5fc9f3	b58dbad2
34	b58dbad2					84a64fb2	312bf560
35	312bf560					4ea6dc4f	7f8d292f
36	7f8d292f	8d292f7f	5da515d2	1b000000	46a515d2	ead27321	ac7766f3
37	ac7766f3					b58dbad2	19fadc21
38	19fadc21					312bf560	28d12941
39	28d12941					7f8d292f	575c006e
40	575c006e	5c006e57	4a639f5b	36000000	7c639f5b	ac7766f3	d014f9a8
41	d014f9a8					19fadc21	c9ee2589
42	c9ee2589					28d12941	e13f0cc8
43	e13f0cc8					575c006e	b6630ca6

Table C.1: Keyexpansion

Table C.2 is a test case taken from NIST [12, pp. 35–36].  
16 bytes of data are run on a single AES128 cipher.

Plaintext:           32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 32 07 34  
Key:                2B 7E 15 16 28 AE D2 A6 AB F7 15 88 09 CF 4F 3C

Round	Start of	After	After	After	Round Key
-------	----------	-------	-------	-------	-----------

Number	Round	SubBytes	ShiftRows	MixColumns		Value
input	328831e0				$\oplus$	2b28ab09
	435a3137					7eae f7cf
	f6309807					15d2154f
	a88da234					16a6883c
1	19a09ae9	d4e0b81e	d4e0b81e	04e04828	$\oplus$	a088232a
	3df4c6f8	27bf b441	bf b44127	66cbf806		fa54a36c
	e3e28d48	11985d52	5d521198	8119d326		fe2c3976
	be2b2a08	ae f1e530	30ae f1e5	e59a7a4c		17b13905
2	a4686b02	49457f77	49457f77	581bdb1b	$\oplus$	f27a5973
	9c9f5b6a	dedb3902	db3902de	4d4be76b		c2963559
	7f35ea50	d2968753	8753d296	ca5aca b0		95b980f6
	f22b4349	89f11a3b	3b89f11a	f1acae8e5		f2437a7f
3	aa618268	ac ef1345	ac ef1345	752053bb	$\oplus$	3d471e6d
	8fdd d232	73c1 b523	c1b52373	ec0bc025		8016237a
	5fe34a46	cf11 d65a	d65acf11	0963cf d0		47fe7e88
	03ef d29a	7bdf b5b8	b87bdf b5	93337c dc		7d3e443b
4	48674dd6	5285e3f6	5285e3f6	0f606f5e	$\oplus$	ef a8b6db
	6c1de35f	50a411cf	a411cf50	d631c0b3		4452710b
	4e9db158	2f5ec86a	c86a2f5e	da381013		a55b25ad
	ee0d38e7	28d70794	9428d707	a9bfb6b01		4a7f3b00
5	e0c8d985	e1e83597	e1e83597	25bdb64c	$\oplus$	d47c ca11
	9263b1b8	4ffbc86c	fb c86c4f	d1113a4c		d18df2f9
	7f6335be	d2fb96ae	96aed2fb	a9d133c0		c69db815
	e8c05001	9bba537c	7c9bba53	ad688eb0		f887bc bc
6	f1c17c5d	a178104c	a178104c	4b2c3337	$\oplus$	6d11db ca
	0092c8b5	634f e8d5	4f e8d563	864a9dd2		880bf900
	6f4c8bd5	a8293d03	3d03a829	8d89f418		a33e8693
	55ef320c	fcdf23fe	fe fcd f23	6d80e8d8		7afdf41fd
7	263de8fd	f7279b54	f7279b54	14462734	$\oplus$	4e5f844e
	0e4164d2	ab8343b5	8343b5ab	1516462a		545fa6a6
	2eb7728b	31a9403d	403d31a9	b51556d8		f7c94f dc
	177da925	f0ff d33f	3ff0ff d3	bfecd743		0ef3b24f
8	5a19a37a	bed40ada	bed40ada	00b154fa	$\oplus$	ea b5317f
	4149e08c	833be164	3be16483	51c8761b		d28d2b8d
	42dc1904	2c86d4f2	d4f22c86	2f896d99		73ba f529
	b11f650c	c8c04dfe	fec8c04d	d1ffcd ea		21d2602f

9	<table><tr><td>ea</td><td>04</td><td>65</td><td>85</td></tr><tr><td>83</td><td>45</td><td>5d</td><td>96</td></tr><tr><td>5c</td><td>33</td><td>98</td><td>b0</td></tr><tr><td>f0</td><td>2d</td><td>ad</td><td>c5</td></tr></table>	ea	04	65	85	83	45	5d	96	5c	33	98	b0	f0	2d	ad	c5	<table><tr><td>87</td><td>f2</td><td>4d</td><td>97</td></tr><tr><td>ec</td><td>6e</td><td>4c</td><td>90</td></tr><tr><td>4a</td><td>c3</td><td>46</td><td>e7</td></tr><tr><td>8c</td><td>d8</td><td>95</td><td>a6</td></tr></table>	87	f2	4d	97	ec	6e	4c	90	4a	c3	46	e7	8c	d8	95	a6	<table><tr><td>87</td><td>f2</td><td>4d</td><td>97</td></tr><tr><td>6e</td><td>4c</td><td>90</td><td>ec</td></tr><tr><td>46</td><td>e7</td><td>4a</td><td>c3</td></tr><tr><td>a6</td><td>8c</td><td>d8</td><td>95</td></tr></table>	87	f2	4d	97	6e	4c	90	ec	46	e7	4a	c3	a6	8c	d8	95	<table><tr><td>47</td><td>40</td><td>a3</td><td>4c</td></tr><tr><td>37</td><td>d4</td><td>70</td><td>9f</td></tr><tr><td>94</td><td>e4</td><td>3a</td><td>42</td></tr><tr><td>ed</td><td>a5</td><td>a6</td><td>bc</td></tr></table>	47	40	a3	4c	37	d4	70	9f	94	e4	3a	42	ed	a5	a6	bc	⊕	<table><tr><td>ac</td><td>19</td><td>28</td><td>57</td></tr><tr><td>77</td><td>fa</td><td>d1</td><td>5c</td></tr><tr><td>66</td><td>dc</td><td>29</td><td>00</td></tr><tr><td>f3</td><td>21</td><td>41</td><td>6e</td></tr></table>	ac	19	28	57	77	fa	d1	5c	66	dc	29	00	f3	21	41	6e
	ea	04	65	85																																																																																		
	83	45	5d	96																																																																																		
	5c	33	98	b0																																																																																		
f0	2d	ad	c5																																																																																			
87	f2	4d	97																																																																																			
ec	6e	4c	90																																																																																			
4a	c3	46	e7																																																																																			
8c	d8	95	a6																																																																																			
87	f2	4d	97																																																																																			
6e	4c	90	ec																																																																																			
46	e7	4a	c3																																																																																			
a6	8c	d8	95																																																																																			
47	40	a3	4c																																																																																			
37	d4	70	9f																																																																																			
94	e4	3a	42																																																																																			
ed	a5	a6	bc																																																																																			
ac	19	28	57																																																																																			
77	fa	d1	5c																																																																																			
66	dc	29	00																																																																																			
f3	21	41	6e																																																																																			
10	<table><tr><td>eb</td><td>59</td><td>8b</td><td>1b</td></tr><tr><td>40</td><td>2e</td><td>a1</td><td>c3</td></tr><tr><td>f2</td><td>38</td><td>13</td><td>42</td></tr><tr><td>1e</td><td>84</td><td>e7</td><td>d2</td></tr></table>	eb	59	8b	1b	40	2e	a1	c3	f2	38	13	42	1e	84	e7	d2	<table><tr><td>e9</td><td>cb</td><td>3d</td><td>af</td></tr><tr><td>09</td><td>31</td><td>32</td><td>e2</td></tr><tr><td>89</td><td>07</td><td>7d</td><td>2c</td></tr><tr><td>72</td><td>5f</td><td>94</td><td>b5</td></tr></table>	e9	cb	3d	af	09	31	32	e2	89	07	7d	2c	72	5f	94	b5	<table><tr><td>e9</td><td>cb</td><td>3d</td><td>af</td></tr><tr><td>31</td><td>32</td><td>e2</td><td>09</td></tr><tr><td>7d</td><td>2c</td><td>89</td><td>07</td></tr><tr><td>b5</td><td>72</td><td>5f</td><td>94</td></tr></table>	e9	cb	3d	af	31	32	e2	09	7d	2c	89	07	b5	72	5f	94	<table><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr><tr><td></td><td></td><td></td><td></td></tr></table>																	⊕	<table><tr><td>d0</td><td>c9</td><td>e1</td><td>b6</td></tr><tr><td>14</td><td>ee</td><td>3f</td><td>63</td></tr><tr><td>f9</td><td>25</td><td>0c</td><td>0c</td></tr><tr><td>a8</td><td>89</td><td>c8</td><td>a6</td></tr></table>	d0	c9	e1	b6	14	ee	3f	63	f9	25	0c	0c	a8	89	c8	a6
	eb	59	8b	1b																																																																																		
	40	2e	a1	c3																																																																																		
	f2	38	13	42																																																																																		
1e	84	e7	d2																																																																																			
e9	cb	3d	af																																																																																			
09	31	32	e2																																																																																			
89	07	7d	2c																																																																																			
72	5f	94	b5																																																																																			
e9	cb	3d	af																																																																																			
31	32	e2	09																																																																																			
7d	2c	89	07																																																																																			
b5	72	5f	94																																																																																			
d0	c9	e1	b6																																																																																			
14	ee	3f	63																																																																																			
f9	25	0c	0c																																																																																			
a8	89	c8	a6																																																																																			
output	<table><tr><td>39</td><td>02</td><td>dc</td><td>19</td></tr><tr><td>25</td><td>dc</td><td>11</td><td>6a</td></tr><tr><td>84</td><td>09</td><td>85</td><td>0b</td></tr><tr><td>1d</td><td>fb</td><td>97</td><td>32</td></tr></table>	39	02	dc	19	25	dc	11	6a	84	09	85	0b	1d	fb	97	32																																																																					
39	02	dc	19																																																																																			
25	dc	11	6a																																																																																			
84	09	85	0b																																																																																			
1d	fb	97	32																																																																																			

**Table C.2:** AES128 Scrambling on 16 byte packet

Table C.3 is a test case for the CBC-mode scrambling performed on a TS-packet. The highlighted bytes are the bytes that are left unscrambled, since scrambling only works with packets consisting of 16 bytes. The framed bytes of data contain the header.

Clear Packet	<table><tr><td>47</td><td>60</td><td>80</td><td>11</td></tr></table> 54 68 69 73 20 69 73 20 74 68 65 20 70 61 79 6C 6F 61 64 20 75 73 65 64 20 66 6F 72 20 63 72 65 61 74 69 6E 67 20 74 68 65 20 74 65 73 74 20 76 65 63 74 6F 72 73 20 66 6F 72 20 74 68 65 20 44 56 42 20 49 50 54 56 20 73 63 72 61 6D 62 6C 65 72 2F 64 65 73 63 72 61 6D 62 6C 65 72 7E 20 54 68 69 73 20 69 73 20 74 68 65 20 70 61 79 6C 6F 61 64 20 75 73 65 64 20 66 6F 72 20 63 72 65 61 74 69 6E 67 20 74 68 65 20 74 65 73 74 20 76 65 63 74 6F 72 73 20 66 6F 72 20 74 68 65 20 44 56 42 20 49 50 54 56 20 73 63 72 61 6D 62 6C 65 72 2F 64 65 73 63 72 61 6D	47	60	80	11
47	60	80	11		
Scrambled Packet	<table><tr><td>47</td><td>60</td><td>80</td><td>11</td></tr></table> 15 CE 67 E0 CB 01 B5 3C E7 60 54 E5 7A 4A D1 20 A0 DF A4 EA AA E9 32 C6 78 3F 51 AE 19 FA EE 10 8B DB 78 F3 11 3E C2 B5 72 CC 20 85 00 A5 2C EC A1 14 12 6C 58 24 4D F5 63 E7 A9 B4 E0 41 CB C3 FB FF FB D8 3C 8F BF FB 10 E8 3E A3 82 04 BA D7 02 FB 01 A2 7B 62 2C 4F 85 AA B6 AA 75 55 97 20 D6 5A B8 44 CE A2 8C F2 E1 FE 5E 7A C1 9D 44 81 89 19 C2 32 49 F1 40 75 7B 5D 16 C0 AF 45 B2 5F 50 9B 9D A0 61 97 12 C5 9F 0B 30 B0 6F 1F BE 90 12 3F 21 29 83 93 6A 95 31 7F CB 62	47	60	80	11
47	60	80	11		

	F4 34 6A 1B 1E 16 48 40 30 3A FF 83 8A 01 9B F8
	10 A8 E0 B2 2F 64 65 73 63 72 6A 6D

Table C.3: TS packet scrambled by AES128 in CBC-mode.

Keyexpansion

This section only contains input keys, and the respective expanded keys.

Input key: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Output key:	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
	62	63	63	63	62	63	63	62	63	63	63	62	63	63	63
	9b	98	98	c9	f9	fb	fb	aa	9b	98	98	c9	f9	fb	fb
	90	97	34	50	69	6c	cf	fa	f2	f4	57	33	0b	0f	ac
	ee	06	da	7b	87	6a	15	81	75	9e	42	b2	7e	91	ee
	7f	2e	2b	88	f8	44	3e	09	8d	da	7c	bb	f3	4b	92
	ec	61	4b	85	14	25	75	8c	99	ff	09	37	6a	b4	9b
	21	75	17	87	35	50	62	0b	ac	af	6b	3c	c6	1b	f0
	0e	f9	03	33	3b	a9	61	38	97	06	0a	04	51	1d	fa
	b1	d4	d8	e2	8a	7d	b9	da	1d	7b	b3	de	4c	66	49
	b4	ef	5b	cb	3e	92	e2	11	23	e9	51	cf	6f	8f	18

Input key : ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff

Output key:	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff	ff
	e8	e9	e9	e9	17	16	16	16	e8	e9	e9	e9	17	16	16
	ad	ae	ae	19	ba	b8	b8	0f	52	51	51	e6	45	47	f0
	09	0e	22	77	b3	b6	9a	78	e1	e7	cb	9e	a4	a0	8c
	e1	6a	bd	3e	52	dc	27	46	b3	3b	ec	d8	17	9b	60
	e5	ba	f3	ce	b7	66	d4	88	04	5d	38	50	13	c6	58
	71	d0	7d	b3	c6	b6	a9	3b	c2	eb	91	6b	d1	2d	c9
	e9	0d	20	8d	2f	bb	89	b6	ed	50	18	dd	3c	7d	d1
	96	33	73	66	b9	88	fa	d0	54	d8	e2	0d	68	a5	33
	8b	f0	3f	23	32	78	c5	f3	66	a0	27	fe	0e	05	14
	d6	0a	35	88	e4	72	f0	7b	82	d2	d7	85	8c	d7	c3

Input key : 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f

Output key:	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e
	d6	aa	74	fd	d2	af	72	fa	da	a6	78	f1	d6	ab	76
	b6	92	cf	0b	64	3d	bd	f1	be	9b	c5	00	68	30	b3
	b6	ff	74	4e	d2	c2	c9	bf	6c	59	0c	bf	04	69	bf
	47	f7	f7	bc	95	35	3e	03	f9	6c	32	bc	fd	05	8d
	3c	aa	a3	e8	a9	9f	9d	eb	50	f3	af	57	ad	f6	22
	5e	39	0f	7d	f7	a6	92	96	a7	55	3d	c1	0a	a3	1f
	14	f9	70	1a	e3	5f	e2	8c	44	0a	df	4d	4e	a9	c0
	47	43	87	35	a4	1c	65	b9	e0	16	ba	f4	ae	bf	7a

54	99	32	d1	f0	85	57	68	10	93	ed	9c	be	2c	97	4e
13	11	1d	7f	e3	94	4a	17	f3	07	a7	8b	4d	2b	30	c5

Input key : 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

Output key:	00	11	22	33	44	55	66	77	88	99	aa	bb	cc	dd	ee	ff
	c0	39	34	78	84	6c	52	0f	0c	f5	f8	b4	c0	28	16	4b
	f6	7e	87	c2	72	12	d6	cd	7e	e7	2d	79	be	cf	3b	32
	78	9c	a4	6c	0a	8e	71	a1	74	69	5c	d8	ca	a6	67	ea
	54	19	23	18	5e	97	52	b9	2a	fe	0e	61	e0	58	69	8b
	2e	e0	1e	f9	70	77	4c	40	5a	89	42	21	ba	d1	2b	aa
	30	11	b2	0d	40	66	fe	4d	1a	ef	bc	6c	a0	3e	97	c6
	c2	99	06	ed	82	ff	f8	a0	98	10	44	cc	38	2e	d3	0a
	73	ff	61	ea	f1	00	99	4a	69	10	dd	86	51	3e	0e	8c
	da	54	05	3b	2b	52	9c	71	42	44	41	f7	13	7a	4f	7b
	36	d0	24	46	1d	84	b8	37	5f	c0	f9	c0	4c	ba	b6	bl





# D

---

## Examples

This appendix contains an algebraic example of how to perform an inverse scrambling in CBC-mode.

### CBC-mode calculations

The ciphertext is obtained through the following equation where

$C_0$  is the IV

XOR is noted with  $\oplus$ .

$C_i$  is the ciphertext

$P_i$  is the plaintext

$E_k$  is the encryption algorithm

$D_k$  is the decryption algorithm

$$C_i = E_k(P_i \oplus C_{i-1}) \quad (\text{D.1})$$

The inverse of the encryption algorithm  $E_k$  is the decryption algorithm  $D_k$ .

The inverse of an XOR-operation is an XOR-operation.

This gives us:

$$D_k(C_i) = P_i \oplus C_{i-1} \quad (\text{D.2})$$

which gives us

$$P_i = D_k(C_i) \oplus C_{i-1} \tag{D.3}$$

---

# Synthesis reports

In this appendix a chosen amount of the raw data received from the synthesis have been placed. Not all data is inserted, due to the vast amounts of data.

## Synthesis 2

Slice Logic Utilization:				
Number of Slice Registers:	4357	out of	126576	3%
Number of Slice LUTs:	5121	out of	63288	8%
Number used as Logic:	5113	out of	63288	8%
Number used as Memory:	8	out of	15616	0%
Number used as RAM:	8			
Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	7388			
Number with an unused Flip Flop:	3031	out of	7388	41%
Number with an unused LUT:	2267	out of	7388	30%
Number of fully used LUT-FF pairs:	2090	out of	7388	28%
Number of unique control sets:	103			
IO Utilization:				
Number of IOs:	194			
Number of bonded IOBs:	194	out of	296	65%
Specific Feature Utilization:				
Number of Block RAM/FIFO:	1	out of	268	0%
Number using Block RAM only:	1			
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

## Synthesis 3

Slice Logic Utilization:				
Number of Slice Registers:	2945	out of	126576	2%
Number of Slice LUTs:	5167	out of	63288	8%
Number used as Logic:	5159	out of	63288	8%
Number used as Memory:	8	out of	15616	0%
Number used as RAM:	8			
Slice Logic Distribution:				

Number of LUT Flip Flop pairs used:	6124			
Number with an unused Flip Flop:	3179	out of	6124	51%
Number with an unused LUT:	957	out of	6124	15%
Number of fully used LUT-FF pairs:	1988	out of	6124	32%
Number of unique control sets:	102			
IO Utilization:				
Number of IOs:	194			
Number of bonded IOBs:	194	out of	296	65%
Specific Feature Utilization:				
Number of Block RAM/FIFO:	1	out of	268	0%
Number using Block RAM only:	1			
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

## Synthesis 4

Slice Logic Utilization:				
Number of Slice Registers:	2817	out of	126576	2%
Number of Slice LUTs:	5167	out of	63288	8%
Number used as Logic:	5159	out of	63288	8%
Number used as Memory:	8	out of	15616	0%
Number used as RAM:	8			
Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	5996			
Number with an unused Flip Flop:	3179	out of	5996	53%
Number with an unused LUT:	829	out of	5996	13%
Number of fully used LUT-FF pairs:	1988	out of	5996	33%
Number of unique control sets:	101			
IO Utilization:				
Number of IOs:	194			
Number of bonded IOBs:	194	out of	296	65%
Specific Feature Utilization:				
Number of Block RAM/FIFO:	1	out of	268	0%
Number using Block RAM only:	1			
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

## Synthesis 5

### Addroundkey

Slice Logic Utilization:				
Number of Slice LUTs:	128	out of	63288	0%
Number used as Logic:	128	out of	63288	0%
Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	128			
Number with an unused Flip Flop:	128	out of	128	100%
Number with an unused LUT:	0	out of	128	0%
Number of fully used LUT-FF pairs:	0	out of	128	0%
Number of unique control sets:	0			

---

IO Utilization:				
Number of IOs:	384			
Number of bonded IOBs:	384	out of	296	129% (*)

### CBC

Slice Logic Utilization:				
Number of Slice Registers:	2127	out of	126576	1%
Number of Slice LUTs:	4321	out of	63288	6%
Number used as Logic:	4321	out of	63288	6%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	4740			
Number with an unused Flip Flop:	2613	out of	4740	55%
Number with an unused LUT:	419	out of	4740	8%
Number of fully used LUT-FF pairs:	1708	out of	4740	36%
Number of unique control sets:	51			

IO Utilization:				
Number of IOs:	390			
Number of bonded IOBs:	390	out of	296	131% (*)

Specific Feature Utilization:				
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

### Cipher

Slice Logic Utilization:				
Number of Slice Registers:	1994	out of	126576	1%
Number of Slice LUTs:	4229	out of	63288	6%
Number used as Logic:	4229	out of	63288	6%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	4571			
Number with an unused Flip Flop:	2577	out of	4571	56%
Number with an unused LUT:	342	out of	4571	7%
Number of fully used LUT-FF pairs:	1652	out of	4571	36%
Number of unique control sets:	58			

IO Utilization:				
Number of IOs:	390			
Number of bonded IOBs:	390	out of	296	131% (*)

Specific Feature Utilization:				
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

### Counter

Slice Logic Utilization:				
Number of Slice Registers:	9	out of	126576	0%
Number of Slice LUTs:	14	out of	63288	0%
Number used as Logic:	14	out of	63288	0%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	15			
Number with an unused Flip Flop:	6	out of	15	40%
Number with an unused LUT:	1	out of	15	6%
Number of fully used LUT-FF pairs:	8	out of	15	53%
Number of unique control sets:	2			

## IO Utilization:

Number of IOs:	13			
Number of bonded IOBs:	13	out of	296	4%

## Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16	6%
---------------------------	---	--------	----	----

## Data2state

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	0			
Number with an unused Flip Flop:	0	out of	0	
Number with an unused LUT:	0	out of	0	
Number of fully used LUT-FF pairs:	0	out of	0	
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	257			
Number of bonded IOBs:	256	out of	296	86%

## Specific Feature Utilization:

## Demux

## Slice Logic Utilization:

Number of Slice LUTs:	32	out of	63288	0%
Number used as Logic:	32	out of	63288	0%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	32			
Number with an unused Flip Flop:	32	out of	32	100%
Number with an unused LUT:	0	out of	32	0%
Number of fully used LUT-FF pairs:	0	out of	32	0%
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	97			
Number of bonded IOBs:	97	out of	296	32%

## Specific Feature Utilization:

## Keyblock1

## Slice Logic Utilization:

Number of Slice LUTs:	689	out of	63288	1%
Number used as Logic:	689	out of	63288	1%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	689			
Number with an unused Flip Flop:	689	out of	689	100%
Number with an unused LUT:	0	out of	689	0%
Number of fully used LUT-FF pairs:	0	out of	689	0%
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	1448			
Number of bonded IOBs:	1320	out of	296	445% (*)

## Specific Feature Utilization:

**Keyblock2**

Slice Logic Utilization:				
Number of Slice Registers:	9	out of	126576	0%
Number of Slice LUTs:	208	out of	63288	0%
Number used as Logic:	208	out of	63288	0%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	209			
Number with an unused Flip Flop:	200	out of	209	95%
Number with an unused LUT:	1	out of	209	0%
Number of fully used LUT-FF pairs:	8	out of	209	3%
Number of unique control sets:	2			

IO Utilization:				
Number of IOs:	76			
Number of bonded IOBs:	72	out of	296	24%

Specific Feature Utilization:				
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

**Keyblock3**

Slice Logic Utilization:				
Number of Slice Registers:	1365	out of	126576	1%
Number of Slice LUTs:	1854	out of	63288	2%
Number used as Logic:	1854	out of	63288	2%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	1907			
Number with an unused Flip Flop:	542	out of	1907	28%
Number with an unused LUT:	53	out of	1907	2%
Number of fully used LUT-FF pairs:	1312	out of	1907	68%
Number of unique control sets:	34			

IO Utilization:				
Number of IOs:	2989			
Number of bonded IOBs:	2989	out of	296	1009% (*)
IOB Flip Flops/Latches:	128			

Specific Feature Utilization:				
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

**Keycore**

Slice Logic Utilization:				
Number of Slice Registers:	9	out of	126576	0%
Number of Slice LUTs:	183	out of	63288	0%
Number used as Logic:	183	out of	63288	0%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	184			
Number with an unused Flip Flop:	175	out of	184	95%
Number with an unused LUT:	1	out of	184	0%
Number of fully used LUT-FF pairs:	8	out of	184	4%
Number of unique control sets:	2			

IO Utilization:				
Number of IOs:	69			
Number of bonded IOBs:	69	out of	296	23%

## Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16	6%
---------------------------	---	--------	----	----

## Keyexpansion

## Slice Logic Utilization:

Number of Slice Registers:	1601	out of	126576	1%
Number of Slice LUTs:	2914	out of	63288	4%
Number used as Logic:	2914	out of	63288	4%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	3183			
Number with an unused Flip Flop:	1582	out of	3183	49%
Number with an unused LUT:	269	out of	3183	8%
Number of fully used LUT-FF pairs:	1332	out of	3183	41%
Number of unique control sets:	45			

## IO Utilization:

Number of IOs:	1539			
Number of bonded IOBs:	1539	out of	296	519% (*)

## Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16	6%
---------------------------	---	--------	----	----

## Manager

## Slice Logic Utilization:

Number of Slice Registers:	699	out of	126576	0%
Number of Slice LUTs:	858	out of	63288	1%
Number used as Logic:	850	out of	63288	1%
Number used as Memory:	8	out of	15616	0%
Number used as RAM:	8			

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	1257			
Number with an unused Flip Flop:	558	out of	1257	44%
Number with an unused LUT:	399	out of	1257	31%
Number of fully used LUT-FF pairs:	300	out of	1257	23%
Number of unique control sets:	50			

## IO Utilization:

Number of IOs:	325			
Number of bonded IOBs:	325	out of	296	109% (*)

## Specific Feature Utilization:

Number of Block RAM/FIFO:	1	out of	268	0%
Number using Block RAM only:	1			
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

## Mixcolumns

## Slice Logic Utilization:

Number of Slice LUTs:	176	out of	63288	0%
Number used as Logic:	176	out of	63288	0%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	176			
Number with an unused Flip Flop:	176	out of	176	100%
Number with an unused LUT:	0	out of	176	0%



Number of fully used LUT-FF pairs:	0	out of	176	0%
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	256			
Number of bonded IOBs:	256	out of	296	86%

## Specific Feature Utilization:

**Rcon**

## Slice Logic Utilization:

Number of Slice LUTs:	40	out of	63288	0%
Number used as Logic:	40	out of	63288	0%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	40			
Number with an unused Flip Flop:	40	out of	40	100%
Number with an unused LUT:	0	out of	40	0%
Number of fully used LUT-FF pairs:	0	out of	40	0%
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	72			
Number of bonded IOBs:	72	out of	296	24%

## Specific Feature Utilization:

**Round**

## Slice Logic Utilization:

Number of Slice Registers:	272	out of	126576	0%
Number of Slice LUTs:	1535	out of	63288	2%
Number used as Logic:	1535	out of	63288	2%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	1550			
Number with an unused Flip Flop:	1278	out of	1550	82%
Number with an unused LUT:	15	out of	1550	0%
Number of fully used LUT-FF pairs:	257	out of	1550	16%
Number of unique control sets:	3			

## IO Utilization:

Number of IOs:	388			
Number of bonded IOBs:	388	out of	296	131% (*)

## Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16	6%
---------------------------	---	--------	----	----

**Rword**

## Slice Logic Utilization:

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	0			
Number with an unused Flip Flop:	0	out of	0	
Number with an unused LUT:	0	out of	0	
Number of fully used LUT-FF pairs:	0	out of	0	
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	64			
Number of bonded IOBs:	64	out of	296	21%

## Specific Feature Utilization:

## Scrambler

## Slice Logic Utilization:

Number of Slice Registers:	2817	out of	126576	2%
Number of Slice LUTs:	5167	out of	63288	8%
Number used as Logic:	5159	out of	63288	8%
Number used as Memory:	8	out of	15616	0%
Number used as RAM:	8			

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	5996			
Number with an unused Flip Flop:	3179	out of	5996	53%
Number with an unused LUT:	829	out of	5996	13%
Number of fully used LUT-FF pairs:	1988	out of	5996	33%
Number of unique control sets:	101			

## IO Utilization:

Number of IOs:	194			
Number of bonded IOBs:	194	out of	296	65%

## Specific Feature Utilization:

Number of Block RAM/FIFO:	1	out of	268	0%
Number using Block RAM only:	1			
Number of BUFG/BUFGCTRLs:	1	out of	16	6%

## Shiftrows

## Slice Logic Utilization:

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	0			
Number with an unused Flip Flop:	0	out of	0	
Number with an unused LUT:	0	out of	0	
Number of fully used LUT-FF pairs:	0	out of	0	
Number of unique control sets:	0			

## IO Utilization:

Number of IOs:	256			
Number of bonded IOBs:	256	out of	296	86%

## Specific Feature Utilization:

## State2data

## Slice Logic Utilization:

Number of Slice Registers:	2	out of	126576	0%
Number of Slice LUTs:	1	out of	63288	0%
Number used as Logic:	1	out of	63288	0%

## Slice Logic Distribution:

Number of LUT Flip Flop pairs used:	3			
Number with an unused Flip Flop:	1	out of	3	33%
Number with an unused LUT:	2	out of	3	66%
Number of fully used LUT-FF pairs:	0	out of	3	0%

Number of unique control sets:	2			
--------------------------------	---	--	--	--

IO Utilization:

Number of IOs:	259			
Number of bonded IOBs:	259	out of	296	87%

Specific Feature Utilization:

Number of BUFG/BUFGCTRLs:	1	out of	16	6%
---------------------------	---	--------	----	----

### Subbytes

Slice Logic Utilization:				
Number of Slice LUTs:	512	out of	63288	0%
Number used as Logic:	512	out of	63288	0%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	512			
Number with an unused Flip Flop:	512	out of	512	100%
Number with an unused LUT:	0	out of	512	0%
Number of fully used LUT-FF pairs:	0	out of	512	0%
Number of unique control sets:	0			

IO Utilization:				
Number of IOs:	256			
Number of bonded IOBs:	256	out of	296	86%

Specific Feature Utilization:

### Substitutebox

Slice Logic Utilization:				
Number of Slice LUTs:	128	out of	63288	0%
Number used as Logic:	128	out of	63288	0%

Slice Logic Distribution:				
Number of LUT Flip Flop pairs used:	128			
Number with an unused Flip Flop:	128	out of	128	100%
Number with an unused LUT:	0	out of	128	0%
Number of fully used LUT-FF pairs:	0	out of	128	0%
Number of unique control sets:	0			

IO Utilization:				
Number of IOs:	64			
Number of bonded IOBs:	64	out of	296	21%

Specific Feature Utilization:

## Synthesis 6

Slice Logic Utilization:				
Number of Slice Registers:	2,805	out of	126,576	2%
Number used as Flip Flops:	2,805			
Number used as Latches:	0			
Number used as Latch-thrus:	0			
Number used as AND/OR logics:	0			
Number of Slice LUTs:	4,930	out of	63,288	7%
Number used as logic:	4,872	out of	63,288	7%
Number using O6 output only:	4,476			

Number using O5 output only:	13		
Number using O5 and O6:	383		
Number used as ROM:	0		
Number used as Memory:	8 out of	15,616	1%
Number used as Dual Port RAM:	8		
Number using O6 output only:	4		
Number using O5 output only:	0		
Number using O5 and O6:	4		
Number used as Single Port RAM:	0		
Number used as Shift Register:	0		
Number used exclusively as route-thrus:	50		
Number with same-slice register load:	49		
Number with same-slice carry load:	1		
Number with other load:	0		

#### Slice Logic Distribution:

Number of occupied Slices:	1,727 out of	15,822	10%
Number of MUXCYs used:	196 out of	31,644	1%
Number of LUT Flip Flop pairs used:	5,554		
Number with an unused Flip Flop:	2,915 out of	5,554	52%
Number with an unused LUT:	624 out of	5,554	11%
Number of fully used LUT-FF pairs:	2,015 out of	5,554	36%
Number of slice register sites lost to control set restrictions:	0 out of	126,576	0%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element. The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

#### IO Utilization:

Number of bonded IOBs:	194 out of	296	65%
------------------------	------------	-----	-----

#### Specific Feature Utilization:

Number of RAMB16BWERs:	1 out of	268	1%
Number of RAMB8BWERs:	0 out of	536	0%
Number of BUFIO2/BUFIO2_2CLKs:	0 out of	32	0%
Number of BUFIO2FB/BUFIO2FB_2CLKs:	0 out of	32	0%
Number of BUFG/BUFGMUXs:	1 out of	16	6%
Number used as BUFGs:	1		
Number used as BUFGMUX:	0		
Number of DCM/DCM_CLKGENs:	0 out of	12	0%
Number of ILOGIC2/ISERDES2s:	0 out of	506	0%
Number of IODELAY2/IODRP2/IODRP2_MCBs:	0 out of	506	0%
Number of OLOGIC2/OSERDES2s:	0 out of	506	0%
Number of BSCANs:	0 out of	4	0%
Number of BUFHs:	0 out of	384	0%
Number of BUFPLLs:	0 out of	8	0%
Number of BUFPLL_MCBs:	0 out of	4	0%
Number of DSP48A1s:	0 out of	180	0%
Number of GTPA1_DUALs:	0 out of	2	0%
Number of ICAPs:	0 out of	1	0%
Number of MCBs:	0 out of	4	0%
Number of PCIE_A1s:	0 out of	1	0%
Number of PCILOGICSEs:	0 out of	2	0%
Number of PLL_ADVs:	0 out of	6	0%
Number of PMVs:	0 out of	1	0%

Number of STARTUPs:	0 out of	1	0%
Number of SUSPEND_SYNCs:	0 out of	1	0%



# List of Figures

2.1	The DVB setup . . . . .	6
2.2	CI+ interface. [10, p. 10] . . . . .	9
3.1	General layout of a data packet . . . . .	12
3.2	PES packet derived from TS packets. The packet in the top is a PES packet and the packets in the bottom are TS packets. . . . .	15
3.3	Different kinds of ciphers. [23] . . . . .	16
3.4	Cipher block chaining mode, [22] . . . . .	17
3.5	SP-Network . . . . .	19
4.1	Number of bits in key used. . . . .	24
6.1	State-Matrix . . . . .	30
7.1	Scrambler-block. . . . .	33
7.2	Manager-block. . . . .	34
7.3	CBC-block. . . . .	35
7.4	Cipher-block. . . . .	36
7.5	Keyexpansion-block. . . . .	37
7.6	Keyblock2-block. . . . .	37
7.7	Round-block. . . . .	38
8.1	The top entity . . . . .	43
8.2	Test vector 1 . . . . .	46
8.3	Test vector 2 . . . . .	46
8.4	Test vector 3 . . . . .	46
A.1	Rijndael S-box. . . . .	52
A.2	State-Matrix. . . . .	52
A.3	Rijndael MixColumns equation. . . . .	52
A.4	The Rcon function represented as a vector. . . . .	53
B.1	Flowchart of the Rcon function . . . . .	56
B.2	Flowchart of the key schedule . . . . .	57





---

## Bibliography

- [1] DVB Scene. Delivering the digital standard not so sure about the title. *DVB Scene*, September 2013. URL <http://www.dvb.org/resources/public/scene/DVB-SCENE42.pdf>. Accessed: 10 Feb 2014. Cited on page 21.
- [2] ETSI. Digital video broadcasting (dvb); support for use of scrambling and conditional access (ca) withing digital video broadcasting systems. *ETR 289*, October 1996. URL [http://www.etsi.org/deliver/etsi\\_etr/200\\_299/289/01\\_60/etr\\_289e01p.pdf](http://www.etsi.org/deliver/etsi_etr/200_299/289/01_60/etr_289e01p.pdf). Accessed: 21 Feb 2014. Cited on page 14.
- [3] ETSI TS. Digital video broadcasting (dvb); head-end implementation of dvb simulcrypt. *ETSI TS 103 197*, 10 2008. Accessed: 13 Feb 2014. Cited on page 8.
- [4] ETSI TS. Digital video broadcasting (dvb); specification for the use of video and audio coding in broadcasting applications based on the mpeg-2 transport stream. *ETSI TS 101 154*, 9 2009. URL [http://www.etsi.org/deliver/etsi\\_ts/101100\\_101199/101154/01.09.01\\_60/ts\\_101154v010901p.pdf](http://www.etsi.org/deliver/etsi_ts/101100_101199/101154/01.09.01_60/ts_101154v010901p.pdf). Accessed: 6 March 2014. Cited on page 13.
- [5] ETSI TS. Digital video broadcasting (dvb). *ETSI TS 103 127*, 05 2013. URL [http://www.etsi.org/deliver/etsi\\_ts/103100\\_103199/103127/01.01.01\\_60/ts\\_103127v010101p.pdf](http://www.etsi.org/deliver/etsi_ts/103100_103199/103127/01.01.01_60/ts_103127v010101p.pdf). Accessed: 10 Feb 2014. Cited on pages 2, 12, 13, 14, 25, 26, 27, and 42.
- [6] European Standard. Common interface specification for conditional access and other digital video broadcasting decoder applications. *EN 50221*, February 1997. URL <http://www.dvb.org/resources/public/standards/En50221.V1.pdf>. Accessed: 4 March 2014. Cited on page 8.
- [7] Farncombe Consulting Group. Towards a replacement for the dvb common scrambling algorithm. *Farncombe White Paper*, October 2009.

- URL <http://farncombe.eu/whitepapers/FTLCAWhitePaperTwo.pdf>. Accessed: 28 Jan 2014. Cited on pages 21 and 22.
- [8] Mr Internet. Mixcolumns step for aes. *Empty*, January 2014. URL [http://www.angelfire.com/biz7/atleast/mix\\_columns.pdf](http://www.angelfire.com/biz7/atleast/mix_columns.pdf). Accessed: 28 Jan 2014. Cited on page 31.
- [9] Wei Li. Security analysis of dvb common scrambling algorithm. In *Data, Privacy, and E-Commerce, 2007. ISDPE 2007. The First International Symposium on*, pages 271–273. IEEE, IEEE Xplore, 2007. URL <http://ieeexplore.ieee.org/lt.ltag.bibl.liu.se/stamp/stamp.jsp?tp=&arnumber=4402690>. Accessed: 12 Feb 2014. Cited on pages 21 and 22.
- [10] CI Plus LLP. Ci plus overview. *Common Interface Plus*, November 2011. URL [http://www.ci-plus.com/data/ci-plus\\_overview\\_v2011-11-11.pdf](http://www.ci-plus.com/data/ci-plus_overview_v2011-11-11.pdf). Accessed: 4 March 2014. Cited on pages 8, 9, 27, and 81.
- [11] CI Plus LLP. Ci plus specification. *CI Plus Specification v1.3.1*, September 2011. URL [http://www.ci-plus.com/data/ci-plus\\_specification\\_v1.3.1.pdf](http://www.ci-plus.com/data/ci-plus_specification_v1.3.1.pdf). Cited on page 8.
- [12] NIST. Specification for the advanced encryption standard (aes), November 2001. URL <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>. Accessed: 17 Feb 2014. Cited on pages 46, 59, 60, and 61.
- [13] Bruce Schneier and Niels Ferguson. *Practical Cryptography*. Wiley Publishing, Inc., first edition, 2003. Cited on pages 11, 14, 15, 17, 19, and 23.
- [14] G.J. Schrijen. Use case: Control word protection, May 2011. URL [http://www.hisinitiative.org/\\_lib/img/Intrinsic-ID\\_CWProtection\\_May\\_25.pdf](http://www.hisinitiative.org/_lib/img/Intrinsic-ID_CWProtection_May_25.pdf). Accessed: 18 Feb, 2014. Cited on page 22.
- [15] C. E. Shannon. Communication theory of secrecy systems\*. *Bell System Technical Journal*, 28(4), 1949. ISSN 1538-7305. doi: 10.1002/j.1538-7305.1949.tb00928.x. URL <http://dx.doi.org/10.1002/j.1538-7305.1949.tb00928.x>. Accessed: 28 Jan 2014. Cited on page 18.
- [16] Gustavus J. Simmons. *Contemporary Cryptology*. IEEE Press, 1992. Cited on pages 15 and 17.
- [17] Leonie Simpson, Matt Henricksen, and Wun-She Yap. Improved cryptanalysis of the common scrambling algorithm stream cipher. In *Information Security and Privacy*, pages 108–121. Springer, 2009. URL <http://eprints.qut.edu.au/27578/1/c27578.pdf>. Accessed: 12 Feb 2014. Cited on page 23.
- [18] Douglas R. Stinson. *Cryptography : Theory and practice*. Chapman & Hall / CRC, third edition, 2006. Cited on pages 17, 18, 29, 30, and 31.

- [19] Erik Tews, Julian Wälde, and Michael Weiner. Breaking dvb-csa. In *Research in Cryptology*, pages 45–61. Springer, 2012. URL [http://link.springer.com/lt.ltag.bibl.liu.se/chapter/10.1007%2F978-3-642-34159-5\\_4#page-14](http://link.springer.com/lt.ltag.bibl.liu.se/chapter/10.1007%2F978-3-642-34159-5_4#page-14). Accessed: 3 Feb 2014. Cited on pages 23 and 24.
- [20] Serge Vaudenay, Willi Meier, Simon Fischer, et al. Analysis of lightweight stream ciphers. *École Polytechnique Fédérale De Lausanne*, 2008. URL [http://biblion.epfl.ch/EPFL/theses/2008/4040/EPFL\\_TH4040.pdf](http://biblion.epfl.ch/EPFL/theses/2008/4040/EPFL_TH4040.pdf). Accessed: 3 Feb 2014. Cited on page 17.
- [21] Ralf-Philipp Weinmann and Kai Wirt. Analysis of the dvb common scrambling algorithm. In *Communications and Multimedia Security*, pages 195–207. Springer, October 2006. URL <http://sec.cs.kent.ac.uk/cms2004/Program/CMS2004final/p5a1.pdf>. Accessed: 31 Jan 2014. Cited on page 22.
- [22] Unknown Wikipedia. Cbc encryption, 2014. URL [http://upload.wikimedia.org/wikipedia/commons/thumb/8/80/CBC\\_encryption.svg/2000px-CBC\\_encryption.svg.png](http://upload.wikimedia.org/wikipedia/commons/thumb/8/80/CBC_encryption.svg/2000px-CBC_encryption.svg.png). Accessed: 7 Feb 2014. Cited on pages 17 and 81.
- [23] Unknown Wikipedia. Cipher-taxonomy, 2014. URL <http://upload.wikimedia.org/wikipedia/en/thumb/8/85/Cipher-taxonomy.svg/500px-Cipher-taxonomy.svg.png>. Accessed: 5 Feb 2014. Cited on pages 16 and 81.
- [24] Wikipedia Jr Wikipedia. Rijndael’s key schedule. *Empty*, January 2014. URL [http://en.wikipedia.org/wiki/Rijndael\\_key\\_schedule](http://en.wikipedia.org/wiki/Rijndael_key_schedule). Accessed: 28 jan 2014. Cited on page 32.
- [25] Kai Wirt. Fault attack on the dvb common scrambling algorithm, 2004. URL <https://eprint.iacr.org/2004/289.pdf>. Accessed: 13 Feb 2014. Cited on page 23.
- [26] Xilinx. Fast aes xts/cbc (helion), December 2007. URL [http://www.xilinx.com/products/intellectual-property/Fast\\_AES\\_XTS\\_CBC.htm](http://www.xilinx.com/products/intellectual-property/Fast_AES_XTS_CBC.htm). Accessed 3 June 2014. Cited on page 47.



## Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>

## Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a period of 25 years from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>