Examensarbete

Replacement of the DVB-CSA

Analyzis 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--YY/NNNN--SE

Linköping 2014

Replacement of the DVB-CSA

Analyzis 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--YY/NNNN--SE

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

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

Linköping, 06 mars 2014

LINKÖPINGS UNIVERSITET TEKNISKA HÖGSKOLAN	**Avdelning, Institution** Division, Department ISY Embedded systems Department of Electrical Engineering SE-581 83 Linköping	**Datum** Date 2014-03-006

Språk Language ☐ Svenska/Swedish ☒ Engelska/English ☐ ________	**Rapporttyp** Report category ☐ Licentiatavhandling ☒ Examensarbete ☐ C-uppsats ☐ D-uppsats ☐ Övrig rapport ☐ ________	**ISBN** — **ISRN** LiTH-ISY-EX--YY/NNNN--SE **Serietitel och serienummer** Title of series, numbering **ISSN** —

URL för elektronisk version

Titel Title

Ersättning av DVB-CSA

Replacement of the DVB-CSA

Författare Author

Gustaf Bengtz

Sammanfattning
Abstract

This report adresses 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 them are based on the public AES algorithm. Both of the proposed algorithms use the AES algorithm as a base. The CSA3 combines it 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 the general implementation is displayed.

Nyckelord
Keywords

problem, solving, DVB, scrambling, CISSA, cipher, CSA, CSA3

Abstract

This report adresses 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 them are based on the public AES algorithm. Both of the proposed algorithms use the AES algorithm as a base. The CSA3 combines it 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 the general implementation is displayed.

Notation

Abbreviations

Abbrevation	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 Unig
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
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
S-Box	Substitution-Box
STB	Set-top Box
TS	Transport Stream. Contains data
XRC	eXtended emulation Resistant Cipher

Contents

1 Introduction

WISI Norden AB, previously A2B Electronics, is a Swedish company founded in 1997. The company is a developer of headend 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 which was the most suitable to replace the currently implemented algorithm 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. There already exists an algorithm called CSA2 , which is basically the same 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 a suitable frequency. The decisions made were to be motivated either through simulations or reference litterature.

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 scrambing algorithm chosen in consent between the author and the supervisor at WISI Norden. The implementation should be optimized towards hardware usage, while achieving a suitable frequency, preferably the one used in the rest of the Field-Programmable Gate Array (FPGA).

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 seperate tasks.

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

To gain some knowledge about cryptography, a litterature study was first conducted. This provided some insight into what the strenghts and weaknesses the algorithms actually were. This provided some deeper understanding, and a cipher that was used in both analyzed algorithms was chosen. 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 seperate 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 built bottoms-up.

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

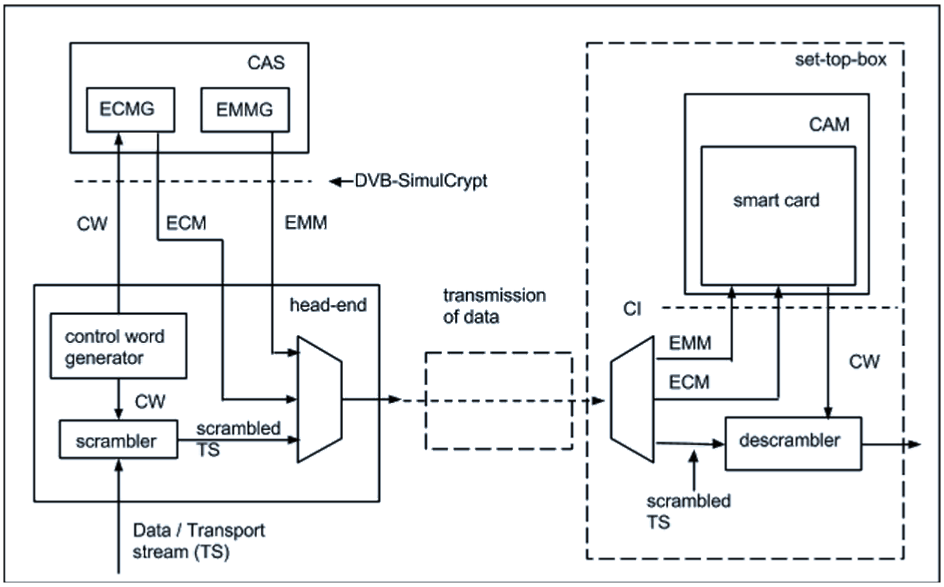

Figure 2.1: *The DVB setup*

2.2 Control word

*Transport Stream*s (TS), 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 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 control word is sent to a *Conditional Access System* (CA system) where the control word is encrypted as an *Entitlement Control Message* (ECM). The CA system also generates an *Entitlement Management Message* (EMM) which tells the smart card what 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 TS packet are sent through a *Common Interface* (CI) to a *Conditional Access Module* (CAM), where the ECM (previous control word) is decrypted using a decryption algorithm located on a 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-Plus, 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-Plus as well as the extra encryption are all discussed in section 2.5.

2.3 Conditional Access System

To make sure that users fulfills 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 seperate 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 (EMMG) and an ECM-generator (ECMG) 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 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 CA system 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, to decrypt 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 Module*s (CAM), which are located in the receiver. What CAS / CAM pair depends on the content provider. For instance, NDS is used by Viasat, 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 (for reference, see section 2.3), 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 to generate an ECM and EMM 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 severals 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-Plus

The CI-Plus realizes the possibility of yet another means of protecting content, which is called *Content Control*. Content control is a way of encrypting the content inside of the CAM, connected to the CI-Plus 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.

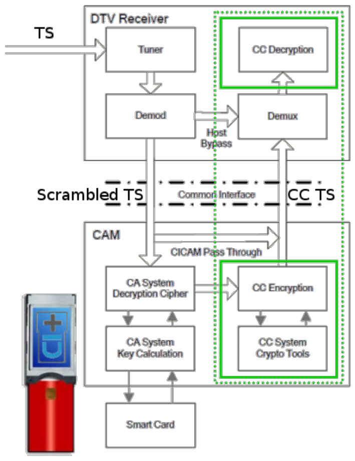

Figure 2.2: *CI-Plus interface. [10, p. 10]*

CI-Plus 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. [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 you connect between the TV signal source and the TV. The set-top box is equipped with both a CI or CI-Plus, and a CAM. The CAM consists of a slot for a smart card and a descrambler. The smart card decodes the 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. However, securing content is not only about cryptography. The main reason why cryptography is 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 noteworthy cryptography rule is that you always are to assume that someone is out to get you. Because of this, Schneier and Fergusson [13, pp. 12–14] claims that it is always needed to look for possible ways to break systems, to ensure that the security can not be breached, and thereby provides security.

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 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. Both of these problems can be solved by cryptography and encryption.

3.2 Scrambling and Encrypting

Scrambling can be seen as the distortion of a plain-text, using a control word. However, encryption can be seen as the entire process of protecting content. This includes everything from generating the control word to scrambling data. Scrambling can therefore be seen as a part of encryption.

Yet another reason to scramble, is to reduce the number of adjacent data bits with the same value, like strings of zeroes or ones. It could also serve to balance the number of zeroes and ones in strings. This is done as to try to obtain DC balance. DC balance is desired since it avoids voltage imbalance during communication between connected systems.

3.3 Data packets

The data processed by the DVB systems is sent in data packets. All of them are created from Elementary Stream packets (ES) which are generally the output from an audio or video encoder. The ES-packets are then packeted into Program Stream- (PS), Transport Stream- (TS) or Packatized Elementary Stream (PES) packets and then distributed. Among the three ways of packing data, only two are interresting 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 interresting 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. The payload is the part of the packets which is the actual data, which is everything except the header and adaptation field.

3.3.1 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].

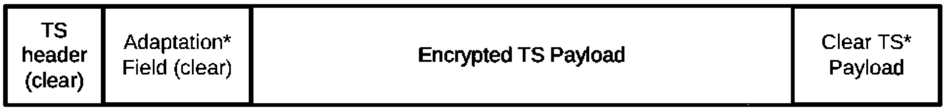

Figure 3.1: *General layout of a data packet*

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:

- header_size = 4
- adaptation_field_size = the size of the adaptation field
- payload_size = 188 - (header_size + adaptation_field_size)
- encrypted_payload_size = payload_size - [payload_size mod block_size]
- clear_payload_size = [payload_size mod block_size] (or simply payload_size - encrypted_payload_size)

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, 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	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 scramling, 01 Reserved, 10 Even key, 11 Odd key
1	Transport Priority	1 gives this packet higher priority
13	PID	Type and number of data stored in packet payload
1	Adaptation Field Control	1 means that an adaptation field exists
1	Contains Payload	1 means that payload exists

4	Continuity Counter	Sequence number of payload packets

Adaptation field

The adaptation field is a sort of padding that is 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. The only time that the adaptation field can exist is when the last string of data is processed, if the end of data does not align with the end of the TS packet. Adaptation fields are never encrypted. [5, pp. 10–11]

Encrypted and clear payload

When working with block ciphers, clear bytes of data tend to turn up. This is since block ciphers only encrypt data blocks of fixed sizes. The clear data is always the data 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. While there is no adaptation field, this is the data right after the header. If there is an adaptation field, some searching for the data is needed to be done. After the start of the payload is found, blocks of a given size are sent to the scrambler. The remainder of 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 at the most. The encrypted payload is always located in front of the clear payload. [5, pp. 10–11]

3.3.2 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 strength of TS packets. 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 itself of PES packets, an analyzation of the 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].

3.4 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.

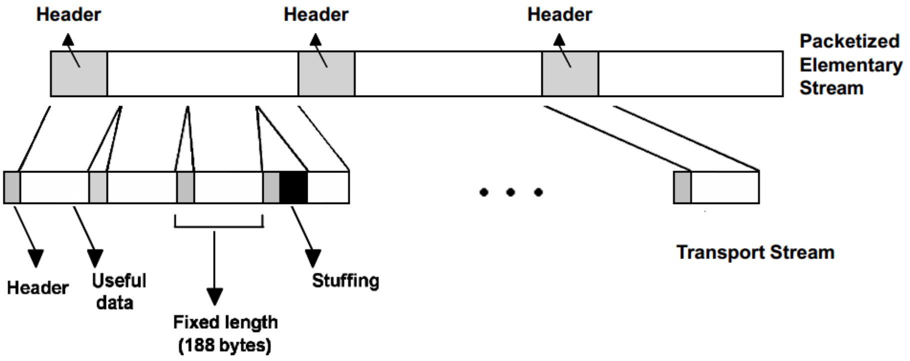

Figure 3.2: *PES packet derived from TS packets*

3.4.1 Symmetric-key encryption

The symmetric-key encryption uses the same key to encode and decode messages. Distrubution 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 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 our 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.

3.4.2 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 the 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 then uses to encode her message. When she sends Bob the message, Bob decodes the message using his secret key S_{Bob} [13]. 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, as described in section 4.3.1.

3.4.3 Combination of encryption

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

To combine the two encryption methods, the symmetric-key algorithm encodes

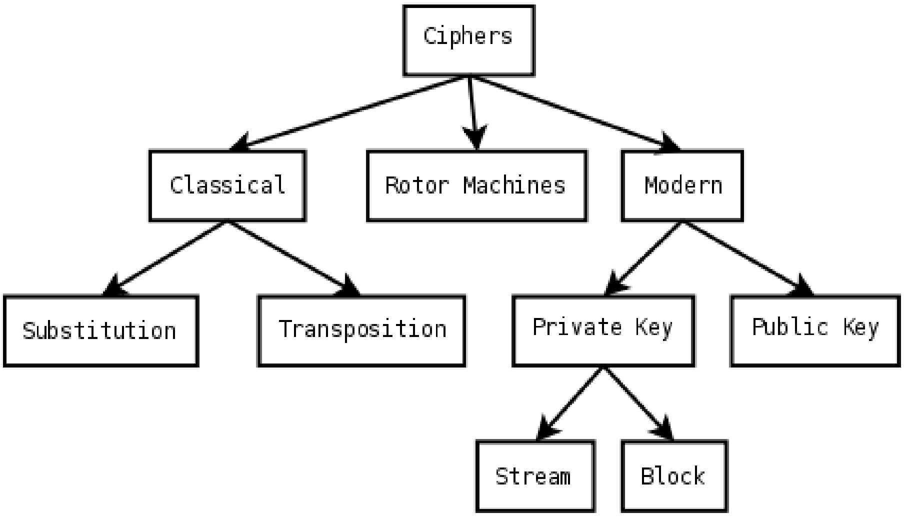

Figure 3.3: *Different kinds of ciphers. [23]*

the plaintext into a ciphertext. Then the public-key encryption encrypts the key used by the symmeteric-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 once again received through decrypting the ciphertext with that key.

3.5 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 describes 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. The Modern ciphers are the Private key and Public key (descibed in chapter 3.4.1 and 3.4.2). 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.

3.5.1 Block cipher

A block cipher operates on fixed sized sets of data. These sets are called blocks, hence the reason that they are called block ciphers. Them being fixed sizes might cause a need for padding the blocks, in case the plaintext contains a number of bytes that is not a denominator of the blocksize. Block ciphers often use a combination of S-boxes (Substitution boxes) and P-boxes (Permutation boxes) in a so-called SP-network (S-box / P-box network) (Figure 3.4).

There are many modes of block ciphers, but the two recommended by Schneier and Fergusson [13] are the CBC-mode and the CTR-mode.

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, this time 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 upcoming data. This is then put into an XOR with the next plaintext, which is then encrypted in the cipher. For reference, see image 19 in appendix 7.8. [18, pp. 109–111]

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 encoded output is sent to an XOR together with the plaintext, producing the ciphertext. The counter is incremented and the procedure is iterated [18, p. 111].

3.5.2 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].

3.5.3 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. A description of how this is done for the CBC-mode (described in 3.5.1) is described in 7.8 in appendix 7.8. It is here assumed that the decryption algorithm is known, for simplicity.

3.6 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 (S-box / P-box network) [18, pp. 74–79]. The very first, as well as the last step, of SP-Networks is usually an XOR between the subkey and the data. This is called *whitening*, and is according to Stinson [18, p. 75] regarded as a very effective way to prevent encryption/decryption without a known 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.

3.6.1 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. 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 3.4, where the P-box is represented by the dotted rectangle in the middle.

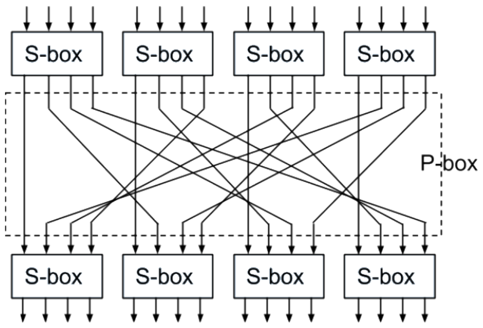

Figure 3.4: *SP-Network*

3.7 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 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 a 64-bit block, 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. Using the same key does on the other hand allow us to easily change the key 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 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 might be needed.

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 3.5.1). 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. Therefore it is not encoded at this point, in 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]. This is possibly 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, and then encrypt it once again on the CI, before sending it to the STB. The latter key is setup between the CI system and the STB through a one time sychronization. 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, to prevent content theft, 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 CW 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 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 in bit size.

$$\text{Largest possible number} = 2^n - 1 \tag{4.1}$$

Since the CSA uses keys consisting of 64-bits (8 bytes), this gives us 18.5 Quintillion (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 Trillion (10^{12}) possible unique keys.

Testing a million keys per second is about what is possible through a modern x86 processor using software methods roughly 3258 days to force brake 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:

$$2^{48} = 2,814749767 * 10^{14} \text{ keys} \tag{4.2}$$

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

$$281 * 10^{12}/10^{6} = 281,4749767 * 10^{6} \text{ seconds} \tag{4.3}$$

By substituting seconds, with day * (seconds/day), the number of keys per days is found insteal:

$$281,4749767 * 10^{6}/86400 = 3257,8 \text{ days} \tag{4.4}$$

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 this up, through hardware accelerations and other methods. 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 just be changed more frequenctly than every second minute. As such, the brute force method of is not a reasonable method to obtain the key.

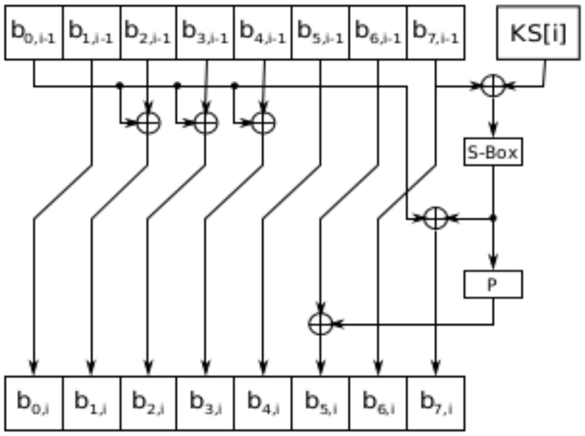

Figure 4.1: *Number of bits in key used*

Known plaintext attack

The known plaintext attacks are performed to figure out the key. What is interresting is that this kind of attack only is applicable for symmetric ciphers. That means that the known plaintext attack can not be used to retrieve the secret key during public-key encryption (section 3.4.2). 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 and CSA3

There are currently two scrambling algorithms being assessed as replacements to the currently used DVB-CSA. This is done to assure content security for yet another ten years.

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 so that it is highly impractical to perform in software, but easily done in hardware.

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.

5.1 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 also implement it. Although it is software-friendly, it is supposed to 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]

5.1.1 Software friendly

An FPGA implementation of the CISSA algorithm seems likely to be implementable, due to the fact that the scrambling of the content is supposed to be made in hardware, regardless as to whether 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 cryptoanalysists 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 therefore not lead to any problem.

5.2 CSA3

The CSA3 scrambling algorithm is based on a combination of an AES (*Advanced Encryption Standard*) block cipher using a 128-bit key, which is simply 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].

5.2.1 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.

5.3 Conclusion

CSA3 implements the AES-128 cipher for scrambling, combined with a confidential cipher, called the XRC cipher. CISSA does not, on the other hand, seem to be using any confidential cipher. It does however use 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 chiper would provide for a basis to continue developing the scrambler towards either CISSA or CSA3 on a later stage. Due to the CIplus interface being implemented, software descrambling and theft will very likely not be as big of a problem using a software friendly scrambling method. AES-128 in CBC-mode seems to be the best to implement, since it is mandatory for HD hosts using the CI+ interface [10, p. 15].

6 Advanced Encryption Standard

The AES is based on an SP-network and is fast both in hardware as well as software. Rijndael, which is used in AES, has key-sizes of at least 128 bits, block lengths of 128 bits, 8 to 8 bit S-boxes and a minimum of 10 rounds of repetition [18, p. 79]. 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].

6.1 Method

The AES consists of a number of steps that are repeated for each block to be encoded. 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 in rounds

1. SubBytes - Each byte is *substituted* using the Rijndael's S-box.
2. ShiftRows - The rows of the state matrix are *permutated*.
3. MixColums - The columns of the matrix are multiplicated 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 done

1. SubBytes
2. ShiftRows
3. AddRoundKey

The ciphertext is then defined as the state-matrix [18, p. 103]. As mentioned in section 3.6 (Confusion and Diffusion), both confusion and diffusion are nescessary. 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 3.6, performed through an XOR between the roundkey and the data.

The KeySchedule is explained in section 6.2.

6.1.1 InitialRound

This is simply an initial AddRoundKey.

6.1.2 SubBytes

In the SubBytes step, each byte is sent to a Rijndael S-box (which is basically a lookup table, see Matrix 13) where they are substituted in a non-linear fashion. This gives us a substituted state matrix.

6.1.3 ShiftRows

The next step is called the ShiftRows step, which left-shift 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.

6.1.4 MixColumns

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 15 in appendix 7.8). While this might seem simple, it really is not. The multiplication makes sure that each input byte affect all output bytes.[8]

The matrix is multiplicated with the vector from the left, (4x4*4x1 = 4x~~4*4~~x1 = 4x1) 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 GF(2^8) (Galois field).

6.1.5 AddRoundKey

Each of the 16 bytes of the state are then combined with a byte from the round key using a bitwise xor. They are then combined to a state matrix (figure 14 in appendix 7.8) containing 4x4 bytes.

6.2 KeyExpansion

To generate round keys from the cipher key, the Rijndael's key schedule is used. 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.

The schedule consists of a couple of loops and a key-schedule core. The schedule core is the part that branches out if c modulo 16 is zero. The entire KeyExpansion can be viewed in Figure 18 in appendix 7.8. To change the key schedule to fit a key size of 192 bits, you simply change the value c is compared to in the first branch in the flowchart from 176 to 206.

6.2.1 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 13).

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 6.2.3.

6.2.2 Rijndael's S-Box

Rijndael's S-box takes an input byte which it transforms according to a LUT (Figure 13 in appendix 7.8). Where the most significant nibble is placed on the Y axis, and the least significant nibble is set on the X axis. Given the input *0x31*, the output *0xC7* would be received from the Rijndael's S-box.

6.2.3 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 16 in appendix 7.8.

I illustrated the steps to be performed in the Rcon function, using a flowchart, which can be viewed in figure 17 in appendix 7.8.

If the input value is 0 or 1, the same value is returned, 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
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 scrambler 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 7.3.2. 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.

7.1 Problems

The main problems that were encoutered were:

- Not possible to get the license for CSA3
- Small interrest for CSA3
- Next to no documentation of the CISSA algorithm
- Hard finding reliable test vectors
- Merging

- Timing

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.

Most of these problems are self-explanatory. The one that will be discussed here is the merging problem. This problem occured due to the fact that 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 of working 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 nescessary 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 seem to be gone now, but without more exhaustive testing, it is quite hard to know.

This is the entire hardware usage, including the interface towards the FPGA, which is one of the reasons why it might appear large, when compared to other implementations.

7.2 Hardware

The top entity can be viewed in Figure 7.1, and the rest of the entities are placed near the explanation of the entities in the following sections.

7.2.1 Hardware usage

The circuit that was first implemented has during the course of the project been optimized, after synthesis have been run, to either minimize hardware usage, or the speed of the circuit. A total of six synthesises were run. A table displaying the differences of the results can be viewed in figure 7.2. The most significant optimizations are discussed in this section.

Optimizations

The most significant optimization that was performed was an optimization of keyblock 3, which used a lot of the provided resources. It was noticed that the

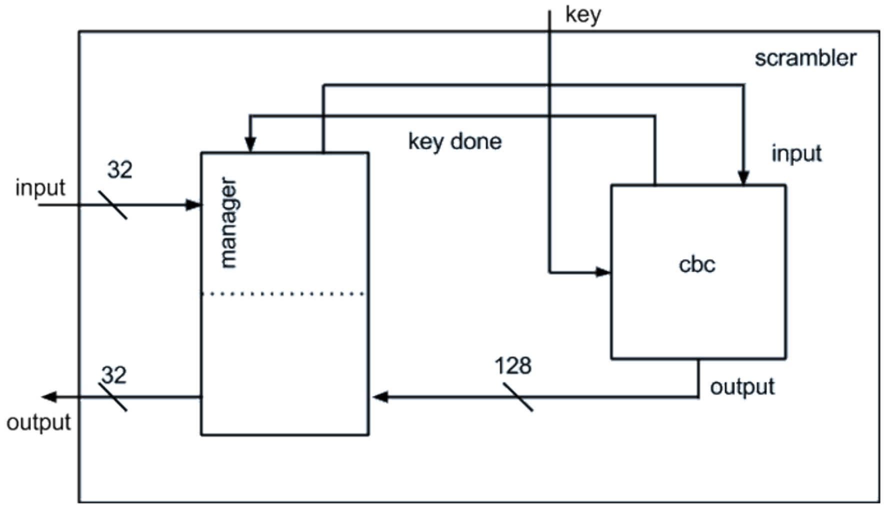

Figure 7.1: *The top entity*

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 seperately, to find out where further optimization would yield the biggest improvement. The hardware usage can be viewed in Table 7.1.

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
▹*keyblock*1	689	0
▹*keyblock*2	208	9
▹*demux*	32	0
▹*keycore*	183	9
▹*ctr*	14	9
▹*rotw*	0	0
▹*sbox*	128	0
▹*rcon*	40	0

▹*keyblock*3	1854	1365
▹*data2state*	0	0
▹*round*	1535	272
▹*subbytes*	512	0
▹*shiftrows*	0	0
▹*mixcolumns*	176	0
▹*addkey*	128	0
▹*state2data*	1	2

Table 7.1: *Hardware usage of entities*

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 interresting numbers from the synthesizes are the number of LUTs used, number of ROMs and number of FFs. 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

(7.1)

Figure 7.2: *Synthesis results*

All of the synthesis results can be viewed in Appendix 7.8

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.

7.3 Further development

There are, as usual, 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.

7.3.1 Rijndael's S-Box

The Rijndael Sbox 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.

7.3.2 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 wether the design would use more slices or not.

7.4 Implementation

This design is very 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 then outputs the data in the form of a TS-packet once again.

The scrambler (Figure 7.3) 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 the right data-bits to the CBC-entity. It also tells the CBC-entity how to handle the data, since different things are to be done depending on if the data is the first data packet sent, or not.

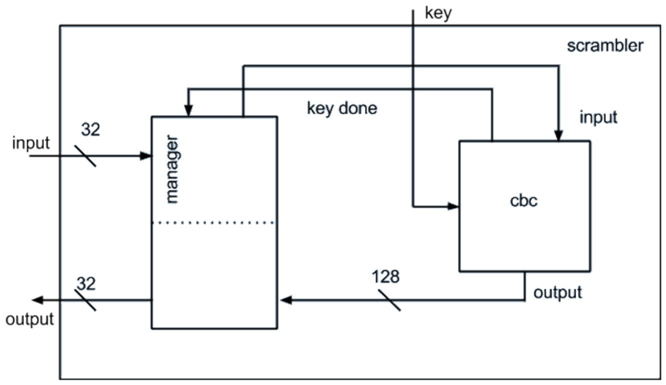

Figure 7.3: *Scrambler-block*

7.4.1 Manager entity

The manager (Figure 7.4) consists of a FIFO, an FSM 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 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 simply done to allow the manager to deal with two sets of data in parallell. When the packet is ready to be sent, a flag is set and the data is sent to the CBC-entity.

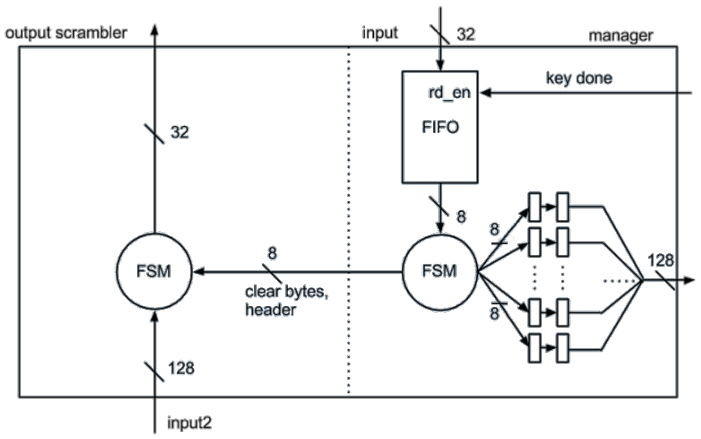

Figure 7.4: *Manager-block*

7.4.2 CBC entity

The CBC-entity (Figure 7.5) 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 parallell, even though it might reduce the maximal speed of the circuit.

7.4.3 Cipher entity

The aes-128 cipher-entity (Figure 7.6) 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

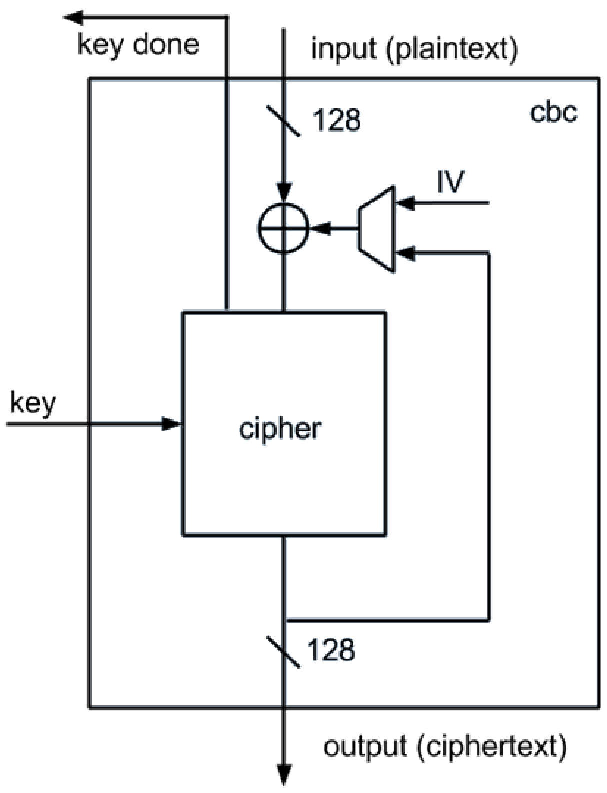

Figure 7.5: *CBC-block*

into an array once again. The cipher entity itself keeps track of timing mainly between the keyexpansion and the round entity. It uses itself of 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 7.6 is that the keyexpansion entity also sends an enable signal, that tells the cipher entity that the expanded key is complete.

7.4.4 Keyexpansion entity

The keyexpansion-entity (Figure 7.7) is divided into 3 keyblock entities. The first keyblock entity decides what 4 bytes of the expanded key are to be expanded. The second keyblock entity (Figure 7.8) contains the keycore, which is only performed on every 4th set of 4 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. 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 7.7 keeps track of when the key generation is done, and

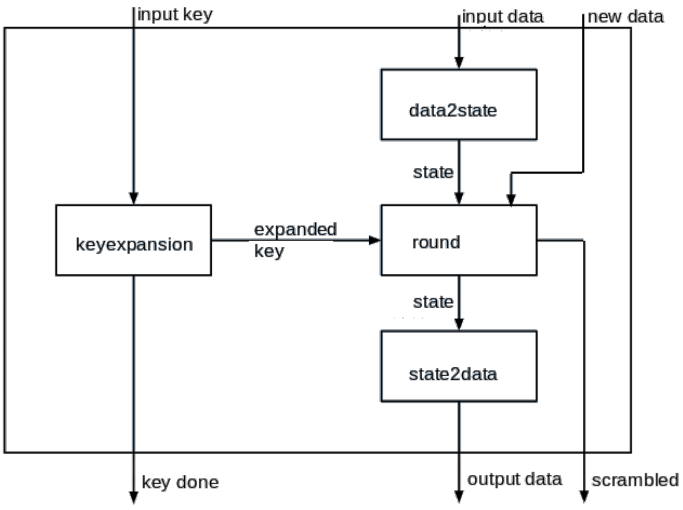

Figure 7.6: *Cipher-block*

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 reset_i signal is the force reset signal.

Keycore entity

The keycore entity consists of four entities. Rotword, Sbox, Rcon and a counter. The counter is used to get the right data-byte 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. Sbox replaces the input bytes according to the Rijndael 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.

7.4.5 Round entity

The round-entity (Figure 7.9) consists of four entities. Subbytes, shiftrows, mixcolumns and addroundkey. Addroundkey is a somewhat special XOR. Subbytes is an Rijndael Sbox, which takes an input 16-byte state, substitutes it, and outputs another 16-byte state. Shiftrows transposes 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, outputting the result of the XOR. The function of the mixcolumns block is a rather complex matrix multiplication.

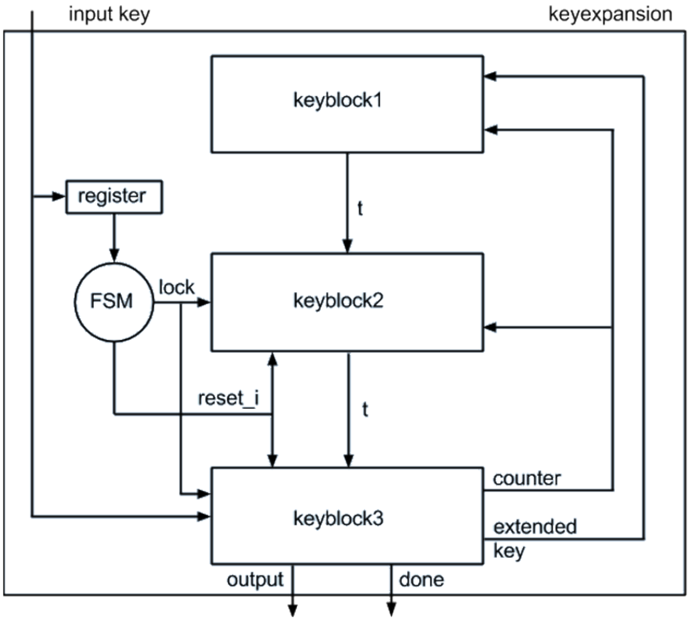

Figure 7.7: *Keyexpansion-block*

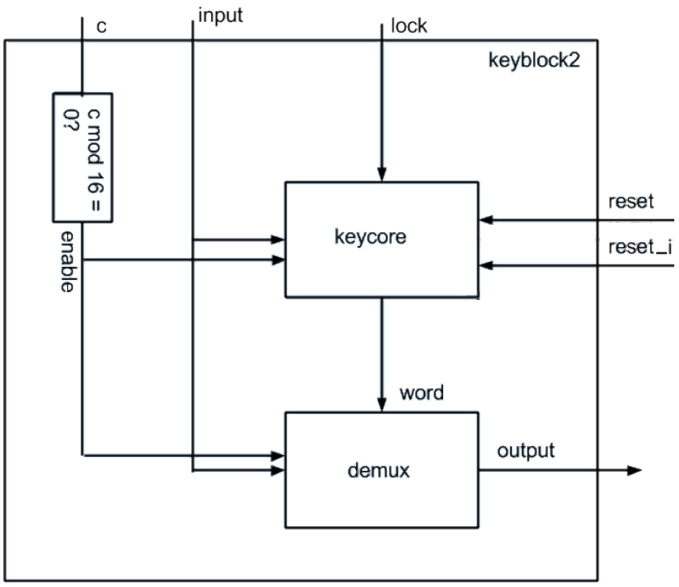

Figure 7.8: *Keyblock2-block*

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

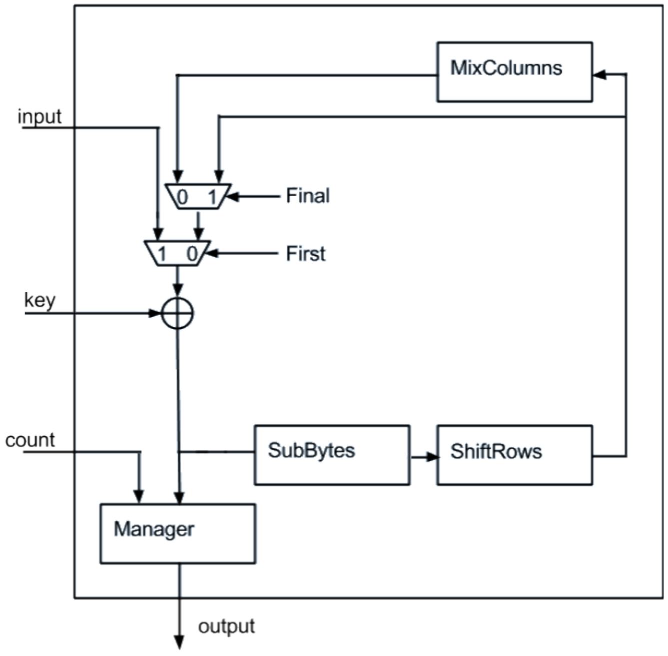

Figure 7.9: *Round-block*

the round entity. On the last round, it takes the output from the subbytes entity. The input to addroundkey is the output from mixcolumns the rest of the time.

The mulblock entity

The mulblock entity consists of one mul3 entity and one mul2 entity, which performs a special kind of hardware multiplication of 3, and 2, on the input. It also takes two inputs which it leaves alone. 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.

7.5 Tests

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

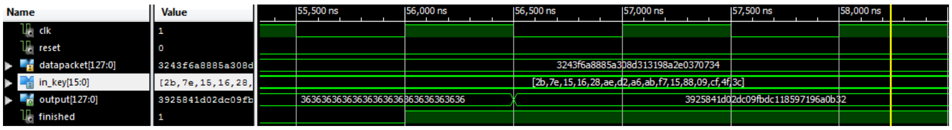
Name
Value
clk 1
reset 0
datapacket[127:0] 3243f6a8885a308d
in_key[15:0] [2b,7e,15,16,28,
output[127:0] 3925841d02dc09fb
finished 1
55,500 ns
56,000 ns
56,500 ns
57,000 ns
57,500 ns
58,000 ns
3243f6a8885a308d313198a2e0370734
[2b,7e,15,16,28,ae,d2,a6,ab,f7,15,88,09,cf,4f,3c]
36363636363636363636363636363636
3925841d02dc09fbdc118597196a0b32

Figure 7.10: *Test vector 1*

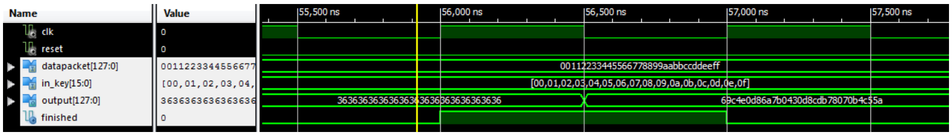
Name
Value
clk 0
reset 0
datapacket[127:0] 00112233445566778
in_key[15:0] [00,01,02,03,04,
output[127:0] 3636363636363636
finished 0
55,500 ns
56,000 ns
56,500 ns
57,000 ns
57,500 ns
00112233445566778899aabbccddeeff
[00,01,02,03,04,05,06,07,08,09,0a,0b,0c,0d,0e,0f]
36363636363636363636363636363636
69c4e0d86a7b0430d8cdb78070b4c55a

Figure 7.11: *Test vector 2*

Figure 7.10 through 7.12 are tests performed on the complete aes-128 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 [12], while test vector 3 is generated using a webpage.

Test vector 1 (Figure 7.10)
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 7.11)
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 7.12)
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

7.6 Comparison to other implementations

Since there exists more implementations of the AES-128 algorithm, an analyzis between the implementations is interresting. To be noted is that this report and implementation have been made in effort to minimize the hardware usage, while

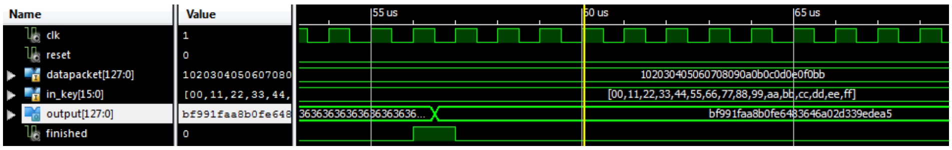
Name
Value
clk 1
reset 0
datapacket[127:0] 1020304050607080
in_key[15:0] [00,11,22,33,44,
output[127:0] bf991faa8b0fe648
finished 0
55 us
60 us
65 us
102030405060708090a0b0c0d0e0f0bb
[00,11,22,33,44,55,66,77,88,99,aa,bb,cc,dd,ee,ff]
363636363636363636...
bf991faa8b0fe6483646a02d339edea5

Figure 7.12: *Test vector 3*

achieving a suitable frequency. Since it is possible to achieve an entirely combinatorial AES128 scrambler, as well as pipelined version, the focus has been to try to find an implementation as similar as this one as possible, to compare results.

Note that this implementation has got a couple of entities, that are usually not present in these kinds of designs, 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 themselves. 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 7.2. 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 tell a lot, since a LUT is said to be used even if just a small part is used in the circuit. Also, LUTs can spand from anything between The same goes for slices. If only one LUT in a slice is used, it still counts the slice as used.

The percentage row in table 7.2 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.

	$Slices$	$LUTs$	$Frequency(MHz)$
$This$	1727	4930	94
$Xilinx$	1041	4047	130
$Percentage$	65%	21.8%	27.7%

Table 7.2: *Comparison between implementations*

7.7 Discussion

The main reason for choosing a bottoms-up methodology, was since the functionality of the smaller blocks were very basic, but the timing was rather complex. In addition to the fact that there were no concrete guidlines, performing the work on the smaller entities, and then implementing the higher ones minimized the probability of performing tasks that were to be discarded in later stages of the implementation.

7.8 Conclusions

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 analyzis 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 AES-128 cipher.

Both the key generation, and the functionality of the entities in AES-128 algorithm could be found in litterature, 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.

Matrixes

Nibble	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

(2)

Figure 13: *Rijndael S-box*

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \tag{3}$$

Figure 14: *State-Matrix*

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a_{1,i} \\ a_{2,i} \\ a_{3,i} \\ a_{4,i} \end{bmatrix}, i = \{1, 2, 3, 4\} \tag{4}$$

Figure 15: *Rijndael MixColumns equation*

$$Rcon[256] = \{\begin{array}{llllllllllllllll}
8D & 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B & 36 & 6C & D8 & AB & 4D & 9A \\
2F & 5E & BC & 63 & C6 & 97 & 35 & 6A & D4 & B3 & 7D & FA & EF & C5 & 91 & 39 \\
72 & E4 & D3 & BD & 61 & C2 & 9F & 25 & 4A & 94 & 33 & 66 & CC & 83 & 1D & 3A \\
74 & E8 & CB & 8D & 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B & 36 & 6C & D8 \\
AB & 4D & 9A & 2F & 5E & BC & 63 & C6 & 97 & 35 & 6A & D4 & B3 & 7D & FA & EF \\
C5 & 91 & 39 & 72 & E4 & D3 & BD & 61 & C2 & 9F & 25 & 4A & 94 & 33 & 66 & CC \\
83 & 1D & 3A & 74 & E8 & CB & 8D & 01 & 02 & 04 & 08 & 10 & 20 & 40 & 80 & 1B \\
36 & 6C & D8 & AB & 4D & 9A & 2F & 5E & BC & 63 & C6 & 97 & 35 & 6A & D4 & B3 \\
7D & FA & EF & C5 & 91 & 39 & 72 & E4 & D3 & BD & 61 & C2 & 9F & 25 & 4A & 94 \\
33 & 66 & CC & 83 & 1D & 3A & 74 & E8 & CB & 8D & 01 & 02 & 04 & 08 & 10 & 20 \\
40 & 80 & 1B & 36 & 6C & D8 & AB & 4D & 9A & 2F & 5E & BC & 63 & C6 & 97 & 35 \\
6A & D4 & B3 & 7D & FA & EF & C5 & 91 & 39 & 72 & E4 & D3 & BD & 61 & C2 & 9F \\
25 & 4A & 94 & 33 & 66 & CC & 83 & 1D & 3A & 74 & E8 & CB & 8D & 01 & 02 & 04 \\
08 & 10 & 20 & 40 & 80 & 1B & 36 & 6C & D8 & AB & 4D & 9A & 2F & 5E & BC & 63 \\
C6 & 97 & 35 & 6A & D4 & B3 & 7D & FA & EF & C5 & 91 & 39 & 72 & E4 & D3 & BD \\
61 & C2 & 9F & 25 & 4A & 94 & 33 & 66 & CC & 83 & 1D & 3A & 74 & E8 & CB & 8D\}
\end{array}$$

Figure 16: *The Rcon function represented as a vector*

Illustrations

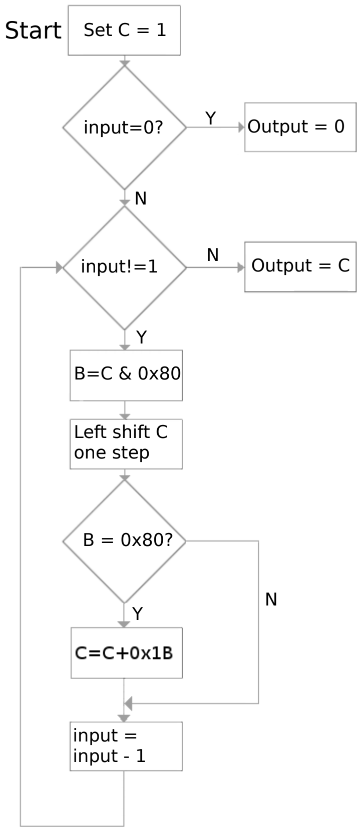

Figure 17: *Flowchart of the Rcon function*

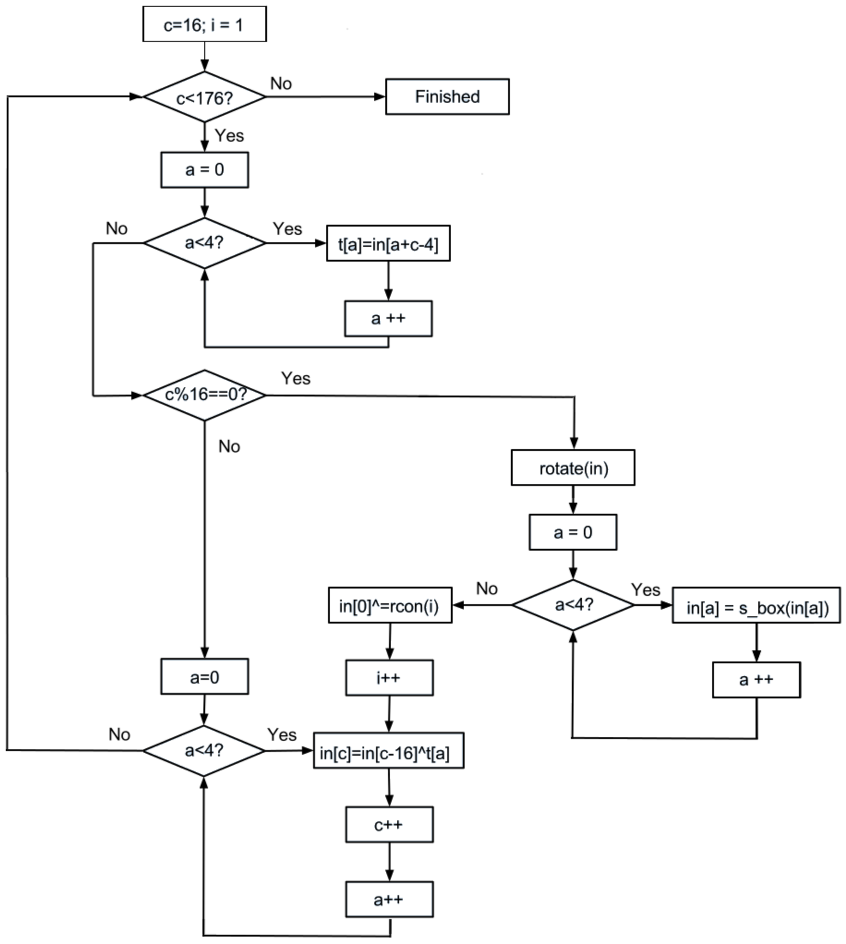

Figure 18: *Flowchart of the key schedule*

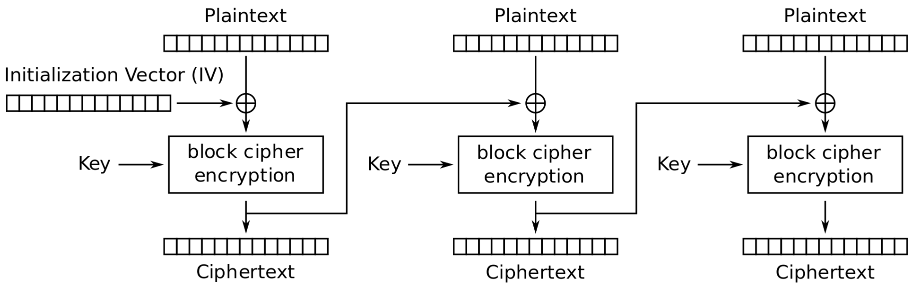

Figure 19: *Cipher block chaining mode, [22]*

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 aes-128 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 3 displays a keyexpansion 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 ⊕ with Rcon	w[i-16]	w[i] = temp $\oplus w[i-16]$
4	09cf4f3c	cf4f3c09	8a84eb01	01000000	8b84eb01	2b7e1516	a0fafe17
5	a0fafe17					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 3: *Keyexpansion*

Table 4 is a test case taken from NIST [12, pp. 35–36].
16 bytes of data are run on a single aes-128 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 Number	Start of Round	After SubBytes	After ShiftRows	After MixColumns		Round Key Value
	32 88 31 e0					2b 28 ab 09
input	43 5a 31 37				⊕	7e ae f7 cf
	f6 30 98 07					15 d2 15 4f
	a8 8d a2 34					16 a6 88 3c

$$1 \qquad \begin{array}{|c|c|c|c|}\hline 19&a0&9a&e9\\\hline 3d&f4&c6&f8\\\hline e3&e2&8d&48\\\hline be&2b&2a&08\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline d4&e0&b8&1e\\\hline 27&bf&b4&41\\\hline 11&98&5d&52\\\hline ae&f1&e5&30\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline d4&e0&b8&1e\\\hline bf&b4&41&27\\\hline 5d&52&11&98\\\hline 30&ae&f1&e5\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 04&e0&48&28\\\hline 66&cb&f8&06\\\hline 81&19&d3&26\\\hline e5&9a&7a&4c\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline a0&88&23&2a\\\hline fa&54&a3&6c\\\hline fe&2c&39&76\\\hline 17&b1&39&05\\\hline\end{array}$$

$$2 \qquad \begin{array}{|c|c|c|c|}\hline a4&68&6b&02\\\hline 9c&9f&5b&6a\\\hline 7f&35&ea&50\\\hline f2&2b&43&49\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 49&45&7f&77\\\hline de&db&39&02\\\hline d2&96&87&53\\\hline 89&f1&1a&3b\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 49&45&7f&77\\\hline db&39&02&de\\\hline 87&53&d2&96\\\hline 3b&89&f1&1a\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 58&1b&db&1b\\\hline 4d&4b&e7&6b\\\hline ca&5a&ca&b0\\\hline f1&ac&a8&e5\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline f2&7a&59&73\\\hline c2&96&35&59\\\hline 95&b9&80&f6\\\hline f2&43&7a&7f\\\hline\end{array}$$

$$3 \qquad \begin{array}{|c|c|c|c|}\hline aa&61&82&68\\\hline 8f&dd&d2&32\\\hline 5f&e3&4a&46\\\hline 03&ef&d2&9a\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline ac&ef&13&45\\\hline 73&c1&b5&23\\\hline cf&11&d6&5a\\\hline 7b&df&b5&b8\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline ac&ef&13&45\\\hline c1&b5&23&73\\\hline d6&5a&cf&11\\\hline b8&7b&df&b5\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 75&20&53&bb\\\hline ec&0b&c0&25\\\hline 09&63&cf&d0\\\hline 93&33&7c&dc\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline 3d&47&1e&6d\\\hline 80&16&23&7a\\\hline 47&fe&7e&88\\\hline 7d&3e&44&3b\\\hline\end{array}$$

$$4 \qquad \begin{array}{|c|c|c|c|}\hline 48&67&4d&d6\\\hline 6c&1d&e3&5f\\\hline 4e&9d&b1&58\\\hline ee&0d&38&e7\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 52&85&e3&f6\\\hline 50&a4&11&cf\\\hline 2f&5e&c8&6a\\\hline 28&d7&07&94\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 52&85&e3&f6\\\hline a4&11&cf&50\\\hline c8&6a&2f&5e\\\hline 94&28&d7&07\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 0f&60&6f&5e\\\hline d6&31&c0&b3\\\hline da&38&10&13\\\hline a9&bf&6b&01\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline ef&a8&b6&db\\\hline 44&52&71&0b\\\hline a5&5b&25&ad\\\hline 4a&7f&3b&00\\\hline\end{array}$$

$$5 \qquad \begin{array}{|c|c|c|c|}\hline e0&c8&d9&85\\\hline 92&63&b1&b8\\\hline 7f&63&35&be\\\hline e8&c0&50&01\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline e1&e8&35&97\\\hline 4f&fb&c8&6c\\\hline d2&fb&96&ae\\\hline 9b&ba&53&7c\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline e1&e8&35&97\\\hline fb&c8&6c&4f\\\hline 96&ae&d2&fb\\\hline 7c&9b&ba&53\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 25&bd&b6&4c\\\hline d1&11&3a&4c\\\hline a9&d1&33&c0\\\hline ad&68&8e&b0\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline d4&7c&ca&11\\\hline d1&8d&f2&f9\\\hline c6&9d&b8&15\\\hline f8&87&bc&bc\\\hline\end{array}$$

$$6 \qquad \begin{array}{|c|c|c|c|}\hline f1&c1&7c&5d\\\hline 00&92&c8&b5\\\hline 6f&4c&8b&d5\\\hline 55&ef&32&0c\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline a1&78&10&4c\\\hline 63&4f&e8&d5\\\hline a8&29&3d&03\\\hline fc&df&23&fe\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline a1&78&10&4c\\\hline 4f&e8&d5&63\\\hline 3d&03&a8&29\\\hline fe&fc&df&23\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 4b&2c&33&37\\\hline 86&4a&9d&d2\\\hline 8d&89&f4&18\\\hline 6d&80&e8&d8\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline 6d&11&db&ca\\\hline 88&0b&f9&00\\\hline a3&3e&86&93\\\hline 7a&fd&41&fd\\\hline\end{array}$$

$$7 \qquad \begin{array}{|c|c|c|c|}\hline 26&3d&e8&fd\\\hline 0e&41&64&d2\\\hline 2e&b7&72&8b\\\hline 17&7d&a9&25\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline f7&27&9b&54\\\hline ab&83&43&b5\\\hline 31&a9&40&3d\\\hline f0&ff&d3&3f\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline f7&27&9b&54\\\hline 83&43&b5&ab\\\hline 40&3d&31&a9\\\hline 3f&f0&ff&d3\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 14&46&27&34\\\hline 15&16&46&2a\\\hline b5&15&56&d8\\\hline bf&ec&d7&43\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline 4e&5f&84&4e\\\hline 54&5f&a6&a6\\\hline f7&c9&4f&dc\\\hline 0e&f3&b2&4f\\\hline\end{array}$$

$$8 \qquad \begin{array}{|c|c|c|c|}\hline 5a&19&a3&7a\\\hline 41&49&e0&8c\\\hline 42&dc&19&04\\\hline b1&1f&65&0c\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline be&d4&0a&da\\\hline 83&3b&e1&64\\\hline 2c&86&d4&f2\\\hline c8&c0&4d&fe\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline be&d4&0a&da\\\hline 3b&e1&64&83\\\hline d4&f2&2c&86\\\hline fe&c8&c0&4d\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 00&b1&54&fa\\\hline 51&c8&76&1b\\\hline 2f&89&6d&99\\\hline d1&ff&cd&ea\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline ea&b5&31&7f\\\hline d2&8d&2b&8d\\\hline 73&ba&f5&29\\\hline 21&d2&60&2f\\\hline\end{array}$$

$$9 \qquad \begin{array}{|c|c|c|c|}\hline ea&04&65&85\\\hline 83&45&5d&96\\\hline 5c&33&98&b0\\\hline f0&2d&ad&c5\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 87&f2&4d&97\\\hline ec&6e&4c&90\\\hline 4a&c3&46&e7\\\hline 8c&d8&95&a6\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 87&f2&4d&97\\\hline 6e&4c&90&ec\\\hline 46&e7&4a&c3\\\hline a6&8c&d8&95\\\hline\end{array} \quad \begin{array}{|c|c|c|c|}\hline 47&40&a3&4c\\\hline 37&d4&70&9f\\\hline 94&e4&3a&42\\\hline ed&a5&a6&bc\\\hline\end{array} \quad \oplus \quad \begin{array}{|c|c|c|c|}\hline ac&19&28&57\\\hline 77&fa&d1&5c\\\hline 66&dc&29&00\\\hline f3&21&41&6e\\\hline\end{array}$$

$$\begin{array}{cccccc}
10 & \begin{array}{|c|c|c|c|}\hline eb & 59 & 8b & 1b \\ \hline 40 & 2e & a1 & c3 \\ \hline f2 & 38 & 13 & 42 \\ \hline 1e & 84 & e7 & d2 \\ \hline \end{array} & \begin{array}{|c|c|c|c|}\hline e9 & cb & 3d & af \\ \hline 09 & 31 & 32 & e2 \\ \hline 89 & 07 & 7d & 2c \\ \hline 72 & 5f & 94 & b5 \\ \hline \end{array} & \begin{array}{|c|c|c|c|}\hline e9 & cb & 3d & af \\ \hline 31 & 32 & e2 & 09 \\ \hline 7d & 2c & 89 & 07 \\ \hline b5 & 72 & 5f & 94 \\ \hline \end{array} & \begin{array}{|c|c|c|c|}\hline & & & \\ \hline & & & \\ \hline & & & \\ \hline & & & \\ \hline \end{array} \quad \oplus & \begin{array}{|c|c|c|c|}\hline d0 & c9 & e1 & b6 \\ \hline 14 & ee & 3f & 63 \\ \hline f9 & 25 & 0c & 0c \\ \hline a8 & 89 & c8 & a6 \\ \hline \end{array} \\
\text{output} & \begin{array}{|c|c|c|c|}\hline 39 & 02 & dc & 19 \\ \hline 25 & dc & 11 & 6a \\ \hline 84 & 09 & 85 & 0b \\ \hline 1d & fb & 97 & 32 \\ \hline \end{array} & & & &
\end{array}$$

Table 4: *AES-128 Scrambling on 16 byte packet*

Table 5 is a test case for the CBC-mode scrambling performed on a TS-packet. The highlighted bytes are left in the clear, due to the scrambling only working with packets consisting of 16 bytes.

Clear Packet	$\boxed{\text{47 60 80 11}}$ 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
Scrambled Packet	$\boxed{\text{47 60 80 11}}$ 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 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 5: *TS packet scrambled 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

```
Output key:  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
             62  63  63  63  62  63  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  aa
             90  97  34  50  69  6c  cf  fa  f2  f4  57  33  0b  0f  ac  99
             ee  06  da  7b  87  6a  15  81  75  9e  42  b2  7e  91  ee  2b
             7f  2e  2b  88  f8  44  3e  09  8d  da  7c  bb  f3  4b  92  90
             ec  61  4b  85  14  25  75  8c  99  ff  09  37  6a  b4  9b  a7
             21  75  17  87  35  50  62  0b  ac  af  6b  3c  c6  1b  f0  9b
             0e  f9  03  33  3b  a9  61  38  97  06  0a  04  51  1d  fa  9f
             b1  d4  d8  e2  8a  7d  b9  da  1d  7b  b3  de  4c  66  49  41
             b4  ef  5b  cb  3e  92  e2  11  23  e9  51  cf  6f  8f  18  8e
```

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  ff
             e8  e9  e9  e9  17  16  16  16  e8  e9  e9  e9  17  16  16  16
             ad  ae  ae  19  ba  b8  b8  0f  52  51  51  e6  45  47  47  f0
             09  0e  22  77  b3  b6  9a  78  e1  e7  cb  9e  a4  a0  8c  6e
             e1  6a  bd  3e  52  dc  27  46  b3  3b  ec  d8  17  9b  60  b6
             e5  ba  f3  ce  b7  66  d4  88  04  5d  38  50  13  c6  58  e6
             71  d0  7d  b3  c6  b6  a9  3b  c2  eb  91  6b  d1  2d  c9  8d
             e9  0d  20  8d  2f  bb  89  b6  ed  50  18  dd  3c  7d  d1  50
             96  33  73  66  b9  88  fa  d0  54  d8  e2  0d  68  a5  33  5d
             8b  f0  3f  23  32  78  c5  f3  66  a0  27  fe  0e  05  14  a3
             d6  0a  35  88  e4  72  f0  7b  82  d2  d7  85  8c  d7  c3  26
```

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  0f
             d6  aa  74  fd  d2  af  72  fa  da  a6  78  f1  d6  ab  76  fe
             b6  92  cf  0b  64  3d  bd  f1  be  9b  c5  00  68  30  b3  fe
             b6  ff  74  4e  d2  c2  c9  bf  6c  59  0c  bf  04  69  bf  41
             47  f7  f7  bc  95  35  3e  03  f9  6c  32  bc  fd  05  8d  fd
             3c  aa  a3  e8  a9  9f  9d  eb  50  f3  af  57  ad  f6  22  aa
             5e  39  0f  7d  f7  a6  92  96  a7  55  3d  c1  0a  a3  1f  6b
             14  f9  70  1a  e3  5f  e2  8c  44  0a  df  4d  4e  a9  c0  26
             47  43  87  35  a4  1c  65  b9  e0  16  ba  f4  ae  bf  7a  d2
             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  bb
```

Examples

CBC-mode calculations

The ciphertext is obtained through the following equation where C_0 is the IV, and the XOR-operation 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

$$C_i = E_k(P_i \oplus C_{i-1}) \tag{5}$$

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

The inverse of the XOR-operation the XOR-operation.

This gives us:

$$D_k(C_i) = P_i \oplus C_{i-1} \tag{6}$$

which gives us

$$P_i = D_k(C_i) \oplus C_{i-1} \tag{7}$$

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

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 19.

[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. 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. 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. Accessed: 10 Feb 2014. Cited on pages 12, 13, 14, 23, 24, and 25.

[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 19 and 20.

[8] Mr Internet. Mixcolumns step for aes. *Empty*, January 2014. URL http://www.angelfire.com/biz7/atleast/mix_columns.pdf. Accessed: 28 jan 2014. Cited on page 28.

[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 19 and 20.

[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. Accessed: 4 March 2014. Cited on pages 8, 9, 25, and 75.

[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. Cited on page 9.

[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 41, 53, 54, and 55.

[13] Bruce Schneier and Niels Fergusson. *Practical Cryptography*. Wiley Publishing, Inc., first edition, 2003. Cited on pages 11, 15, 17, 18, and 20.

[14] G.J. Schrijen. Use case: Control word protection, May 2011. URL http://www.hisinitiative.org/_lib/img/Intrinsic-ID_CWProtection_May_25.pdf. Accessed: 18 Feb, 2014. Cited on page 20.

[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 pages 17 and 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 21.

[18] Douglas R. Stinson. *Cryptography : Theory and practice*. Chapman & Hall / CRC, third edition, 2006. Cited on pages 17, 18, 27, 28, and 29.

[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`. Accessed: 3 Feb 2014. Cited on pages 21 and 22.

[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`. 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 20.

[22] Unknown Wikipedia. Cbc encryption, 2014. URL `http://upload.wikimedia.org/wikipedia/commons/thumb/8/80/CBC_encryption.svg/2000px-CBC_encryption.svg.png`. Accessed: 7 Feb 2014. Cited on pages 51 and 75.

[23] Unknown Wikipedia. Cipher-taxanomy, 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 75.

[24] Wikipedia Jr Wikipedia. Rijndael's key schedule. *Empty*, January 2014. URL `http://en.wikipedia.org/wiki/Rijndael_key_schedule`. Accessed: 28 jan 2014. Cited on page 30.

[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 21.

[26] Xilinx. Fast aes xts/cbc (helion), December 2007. URL `http://www.xilinx.com/products/intellectual-property/Fast_AES_XTS_CBC.htm`. Accessed 3 June 2014. Cited on page 42.