ADVANCED ENCRYPTION STANDARD (AES)

A TERM PROJECT

Submitted by

ARNAB DAS (CRS2109)
SUTIRTHA GHOSH (CRS2114)

Under the supervision of

Prof. Subhamoy Maitra, ISI Kolkata

CRYPTOLOGY AND SECURITY
INDIAN STATISTICAL INSTITUTE
203 B.T Road, Kolkata 700108

JUNE, 2022

ACKNOWLEDGEMENT

We wish to express our sincerest gratitude to Dr Subhamoy Maitra for his continuous guidance and mentorship that he provided us during the project. He showed us the path to achieve our targets by explaining all the tasks to be done and explained to us the importance of this project as well as its industrial relevance. He was always ready to help us and clear our doubts regarding any hurdles in this project. Without his constant support and motivation, this project would not have been successful.

Place: Kolkata — Arnab Das

Date: 25.06.2022 — Sutirtha Ghosh

Contents

Acknowledgement **i**

Content **ii**

1 INTRODUCTION **1**
1.1 History of AES 1

2 Internal Structure of AES **2**
2.1 Key length of AES 2
2.2 Layers of AES 2

3 AES Encrypton **4**
3.1 Single Round AES 4
3.1.1 *Byte Substitution Layer* 5
3.1.2 Diffusion layer 6
3.1.3 Key Schedule 7

4 DECRYPTION **11**
4.1 Inverse Mix Column Sublayer 12
4.2 Inverse ShiftRows Sublayer 13
4.3 Inverse Byte Substitution Layer 13
4.4 Decryption Key Schedule 13

5 AES IMPLEMENTATION **14**

6 CONCLUSION **30**

Chapter 1

INTRODUCTION

1.1 History of AES

In 1997 NIST called for proposals for a new Advanced Encryption Standard(AES).Unlike the DES development, the selection of the algorithm for AES was an open process administered by NIST. In three subsequent AES evaluation rounds, NIST and the international scientific community discussed the advantages and disadvantages of the submitted ciphers and narrowed down the number of potential candidates. In 2001, NIST declared the block cipher Rijndael as the new AES and published it as a final standard (FIPS PUB 197). Rijndael was designed by two young Belgian cryptographers.

The invitation for submitting suitable algorithms and the subsequent evaluation of the successor of DES was a public process. A compact chronology of the AES selection process is given here:

- The need for a new block cipher was announced on January 2, 1997, by NIST.
- A formal call for AES was announced on September 12, 1997.
- Fifteen candidate algorithms were submitted by researchers from several coun- tries by August 20, 1998.
- On August 9, 1999, five finalist algorithms were announced.
- On October 2, 2000, NIST announced that it had chosen Rijndael as the AES.
- On November 26, 2001, AES was formally approved as a US federal standard.

It is expected that AES will be the dominant symmetric-key algorithm for many commercial applications for the next few decades. It is also remarkable that in 2003 the US National Security Agency (NSA) announced that it allows AES to encrypt classified documents up to the level SECRET for all key lengths, and up to the TOP SECRET level for key lengths of either 192 or 256 bits. Prior to that date, only non-public algorithms had been used for the encryption of classified documents.[**?**]

Chapter 2

Internal Structure of AES

2.1 Key length of AES

The AES cipher is almost identical to the block cipher Rijndael. The Rijndael block and key size vary between 128, 192 and 256 bits. However, the AES standard only calls for a block size of 128 bits. The number of internal rounds of the cipher is a function of the key length according to Table 1.1

Table 2.1: key length and round table

key length	**# of rounds**
128 bit	10
192 bit	12
256	14

2.2 Layers of AES

AES consists of so-called layers. Each layer manipulates all 128 bits of the data path. There are only three different types of layers. Each round, with the exception of the first, consists of all three layers as shown in Fig. 2.1: the plaintext is denoted as x, the ciphertext as y and the number of rounds as nr. Moreover, the last round nr does not make use of the *MixColumn* transformation, which makes the encryption and decryption scheme symmetric.
This is a brief description of layers:

Key Addition Layer A 128-bit round key, or subkey, which has been derived from the main key in the key schedule, is XORed to the state.

Byte Substitution Layer(S-Box) Each element of the state is nonlinearly transformed using lookup tables with special mathematical properties. This introduces confusion to the data, i.e., it assures that changes in individual state bits propagate quickly across the data path.

Diffusion layer It provides diffusion over all state bits. It consists of two sublayers, both of which perform linear operations:

- The *ShiftRows* layer permutes the data on a byte level.
- The *MixColumn* layer is a matrix operation which combines (mixes) blocks of four bytes.

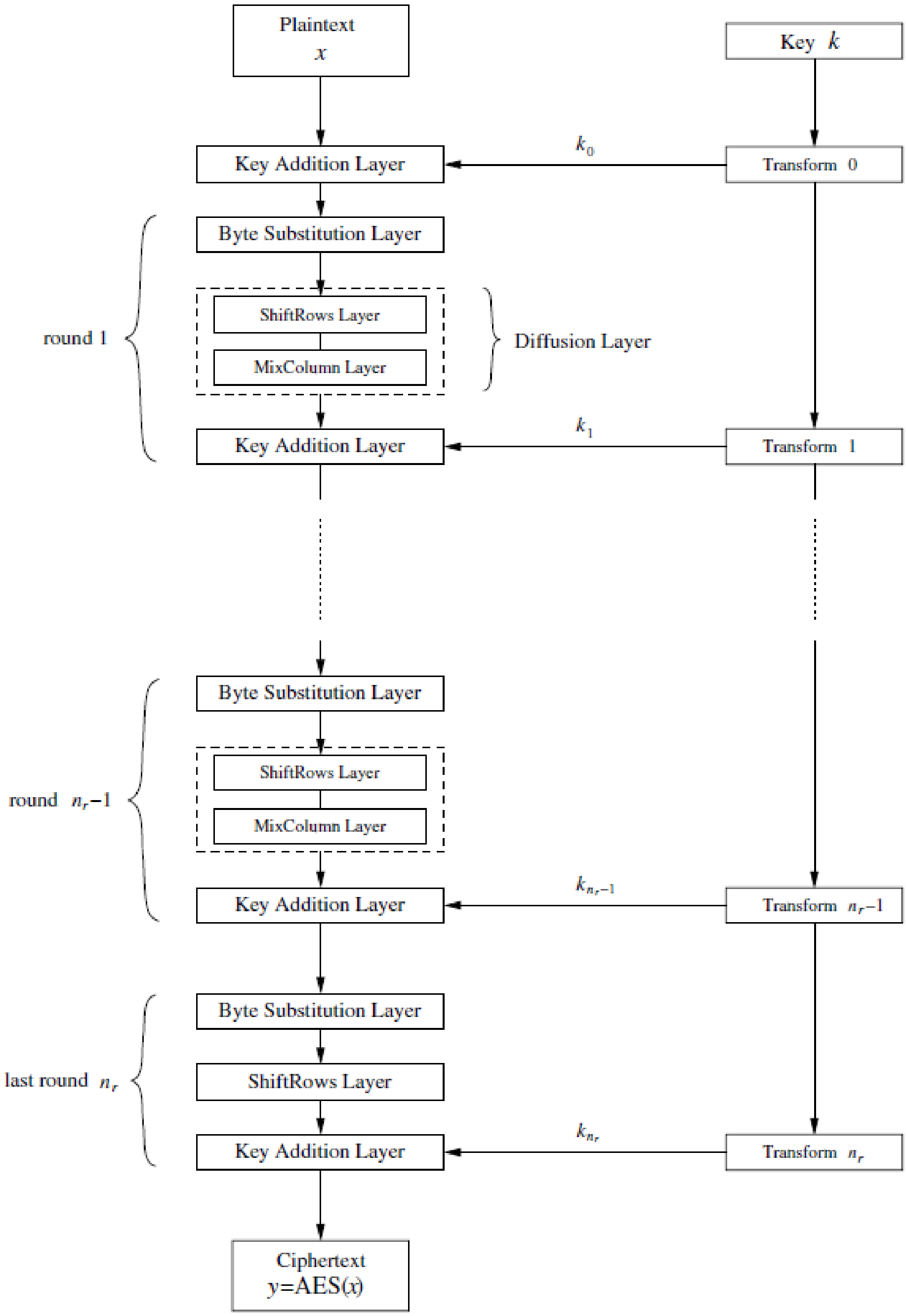

Figure 2.1: AES encryption block diagram

Chapter 3

AES Encrypton

3.1 Single Round AES

In AES size of input block size is 128 bits and it is divided into 16 bytes $A_0, A_1, ..., A_{15}$ $16 * 8 = 128 bits$ and that input is fed byte-wise in S-Box as in Fig 3.1. In order to understand how the data moves through AES, we first imagine that the state A (i.e., the 128-bit data path) consisting of 16 bytes $A_0, A_1, ..., A_15$ is arranged in a four-by-four byte matrix:

$$\begin{bmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{bmatrix}$$

AES operates on elements, columns or rows of the current state matrix. Similarly, the key bytes are arranged into a matrix with four rows and four (128-bit key), six (192-bit key) or eight (256-bit key) columns.

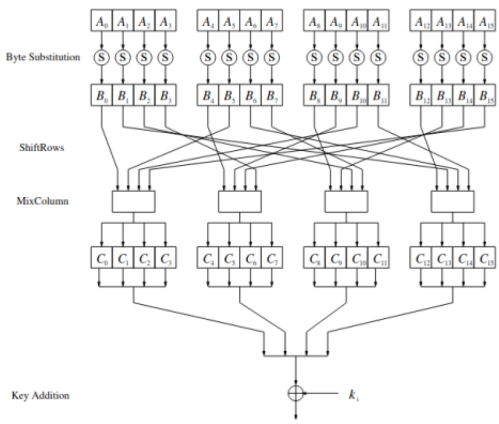

Figure 3.1: AES round function for rounds $1, 2, ..., n_r - 1$

3.1.1 *Byte Substitution Layer*

The first layer in each round is the *Byte Substitution layer*.In the layer, each state byte A_i is replaced, i.e., substituted, by another byte B_i: $S(A_i) = B_i$.
Each S-Box in AES is identical and in software implementations the S-Box is usually realized as a 256-by-8 bit lookup table with fixed entries, as given in Figure 3.2

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

Figure 3.2: AES S-Box: Substitution values in hexadecimal notation for input byte (xy)

As an example if our 8 bits input is $(11000011)_2$ we will divide into two 4-bits number and convert it into hexadecimal number. $S(11000011)_2 = S(C2)_{hex} = (25)_{hex} = (00100101)_2$. Row wise we will check at C and column wise at *2* .

Analysis of S-Box: An AES S-Box can be viewed as a two-step mathematical transformation(Fig. 3.3).

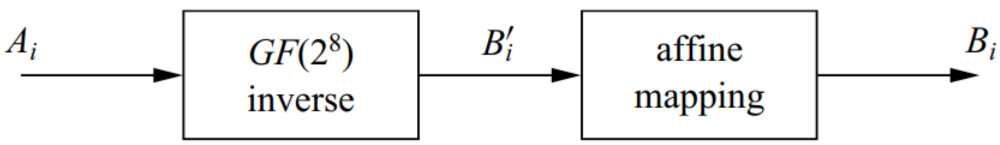

Figure 3.3: The two operations within the AES S-Box which computes the function $B_i = S(A_i)$

The first part of the substitution is a Galois field inversion, the mathematics. For each input element A_i, the inverse is computed: $B'_i = A_i^{-1}$ where both A_i and B'_i are considered elements in the field $GF(2^8)$ with the fixed irreducible polynomial $P(x) = x^8+x^4+x^3+x+1$ In our example , $A_i = (11000010)_2$, corresponding polynomial is $A_i(x) = x^7 + x^6 + x$ and since $B'_i = A_i^{-1}$ so $B'_i(x) = x^5+x^3+x^2+x+1 = A_i^{-1}$ as $A_i(x).B'_i = 1$ mod P(x) in $GF(2^8)$.

In the second part of the substitution, each byte B'_i is multiplied by a constant bit-matrix followed by the addition of a constant 8-bit vector. The operation is described by:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \bmod 2.$$

We now apply the $B_i^{'}$ bit vector as input to the affine transformation. Note that the least significant bit (lsb) b_0 of $B_i^{'}$ is at the rightmost position.After applying the affine transformation we have $B_i = (00100101)_2 = (25)_{hex}$ which is exactly as $(25)_{hex}$ given in fig 3.2.

3.1.2 Diffusion layer

In AES, the Diffusion layer consists of two sublayers, the ShiftRows transformation and the MixColumn transformation. We recall that diffusion is the spreading of the influence of individual bits over the entire state.

ShiftRows Sublayer

If the input of the ShiftRows sublayer is given as a state matrix $B = (B_0, B_1, ..., B_15)$:

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

the output is the new state:

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B_5 & B_9 & B_{13} & b1 \\ B_{10} & B_{14} & B_2 & B_6 \\ B_{15} & B_3 & B_7 & B_{11} \end{bmatrix}$$

MixColumn Sublayer The MixColumn step is a linear transformation which mixes each column of the state matrix. Since every input byte influences four output bytes, the MixColumn operation is the major diffusion element in AES. The combination of the ShiftRows and MixColumn layer makes it possible that after only three rounds every byte of the state matrix depends on all 16 plaintext bytes. In the following, we denote the 16-byte input state by B and the 16-byte output state by C:

$$MixColumn(B) = C. \tag{3.1}$$

$$[\begin{bmatrix} C_0 \\ C1 \\ C_2 \\ C_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 02 & 01 & 01 & 03 \end{bmatrix} \begin{bmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{bmatrix}]$$

The second column of output bytes (C4,C5,C6,C7) is computed by multiplying the four input bytes (B4, B9, B14, B3) by the same constant matrix, and so on. Figure 4.3 shows which input bytes are used in each of the four MixColumn operations. Here 01 means the element 1 of the Galois field and 02 means to the polynomial x;03 means to the polynomial x+1.

3.1.3 Key Schedule

The key schedule takes the original input key (of length 128, 192 or 256 bit) and derives the subkeys used in AES. Note that an XOR addition of a subkey is used both at the input and output of AES. This process is sometimes referred to as key whitening. The number of subkeys is equal to the number of rounds plus one, due to the key needed for key whitening in the first key addition layer, cf. Fig. 3.4. Thus, for the key length of 128 bits, the number of rounds is $n_r = 10$, and there are 11 subkeys, each of 128 bits. The AES with a 192-bit key requires 13 subkeys of length 128 bits, and AES with a 256-bit key has 15 subkeys. The AES subkeys are computed recursively, i.e., in order to derive subkey k_i, subkey $k_i 1$ must be known, etc.

The AES key schedule is word-oriented, where 1 word = 32 bits. Subkeys are stored in a key expansion array W that consists of words. There are different key schedules for the three different AES key sizes of 128, 192 and 256 bit, which are all fairly similar. We introduce the three key schedules in the following.

Key Schedule for 128-Bit Key AES

The ll subkeys are stored in a key expansion array with the elements W[0],...,W[43]. The subkeys are computed as depicted in Fig. 4.5. The elements $K_0, ..., K_{15}$ denote the bytes of the original AES key. First, we note that the first subkey k0 is the original AES key, i.e., the key is copied into the first four elements of the key array W. The other array elements are computed as follows. As can be seen in the figure, the leftmost word of a subkey W[4i], where i = 1,...,10, is computed as:

$$W[4i] = W[4(i-1)] + g(W[4i-1]) \tag{3.2}$$

Here g() is a nonlinear function with a four-byte input and output. The remaining three words of a subkey are computed recursively as:

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j] \tag{3.3}$$

where i = 1,...,10 and j = 1,2,3. The function g() rotates its four input bytes, performs a byte-wise S-Box substitution, and adds a round coefficient RC to it. The round coefficient is an element of the Galois field GF(28), i.e, an 8-bit value. It is only added to the leftmost byte in the function g(). The round coefficients vary from round to round according to the following rule:

$$RC[1] = x^0 = (00000001)_2, \tag{3.4}$$

$$RC[2] = x^1 = (00000010)_2, \tag{3.5}$$

$$RC[3] = x^2 = (00000100)_2, \tag{3.6}$$

$$. \tag{3.7}$$

$$. \tag{3.8}$$

$$. \tag{3.9}$$

$$RC[10] = x^9 = (00110110)_2 \tag{3.10}$$

The function g() has two purposes. First, it adds nonlinearity to the key schedule. Second, it removes symmetry in AES. Both properties are necessary to thwart certain block cipher attacks.

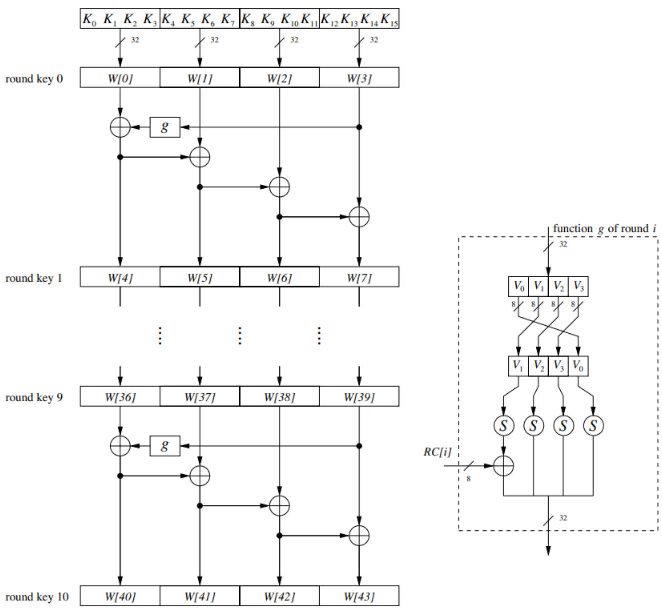

Figure 3.4: Key Schedule for 128-Bit Key AES

Key Schedule for 192-Bit Key AES

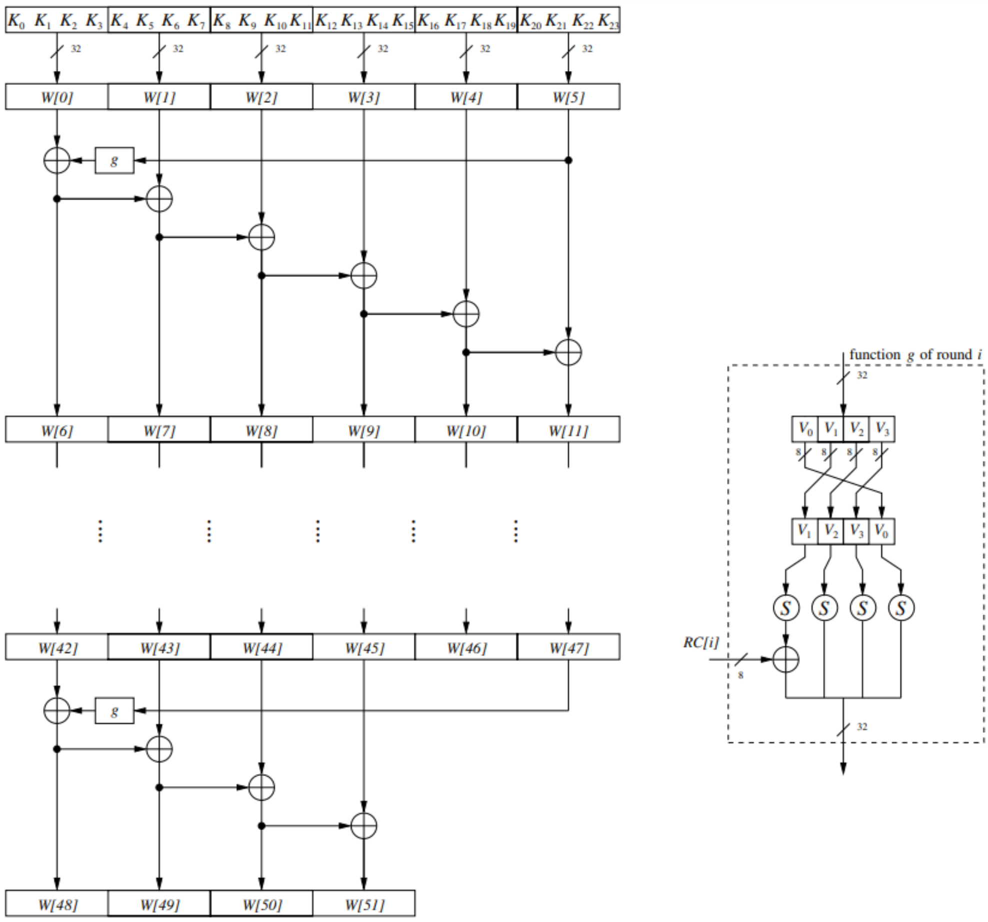

Figure 3.5: AES key shedule for 192-bit key size

Key Schedule for 256-Bit Key AES

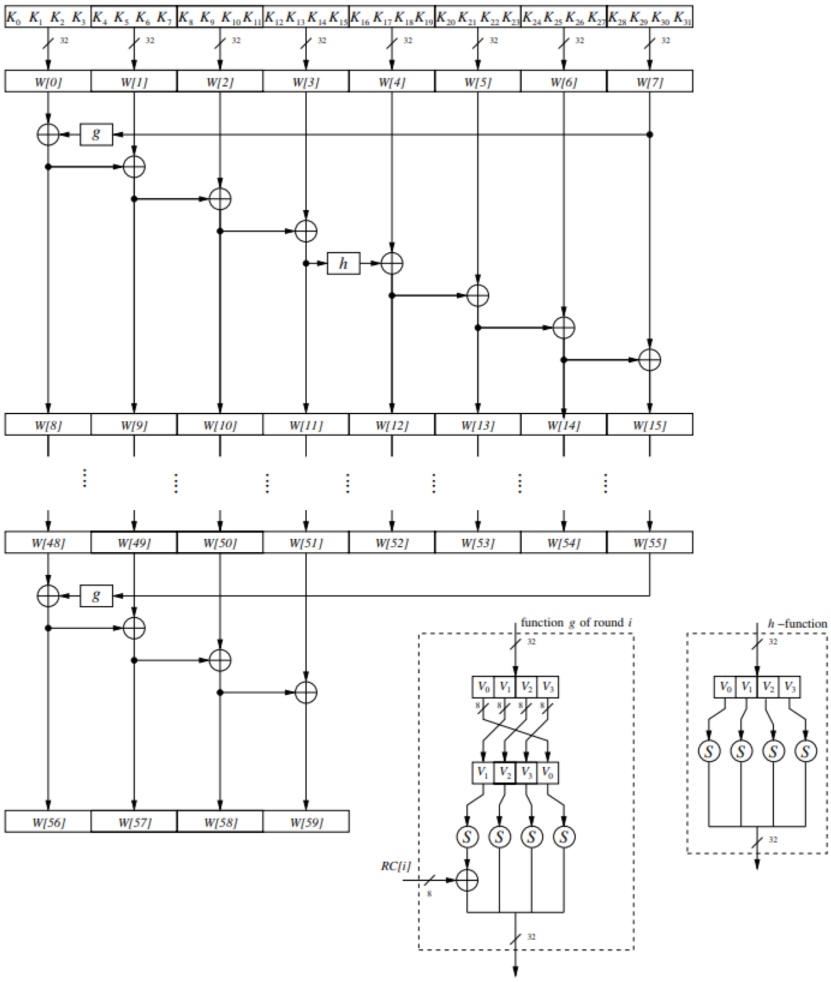

Figure 3.6: AES key shedule for 256-bit key size

Chapter 4

DECRYPTION

A block diagram of the decryption function is shown in Fig. 4.8.

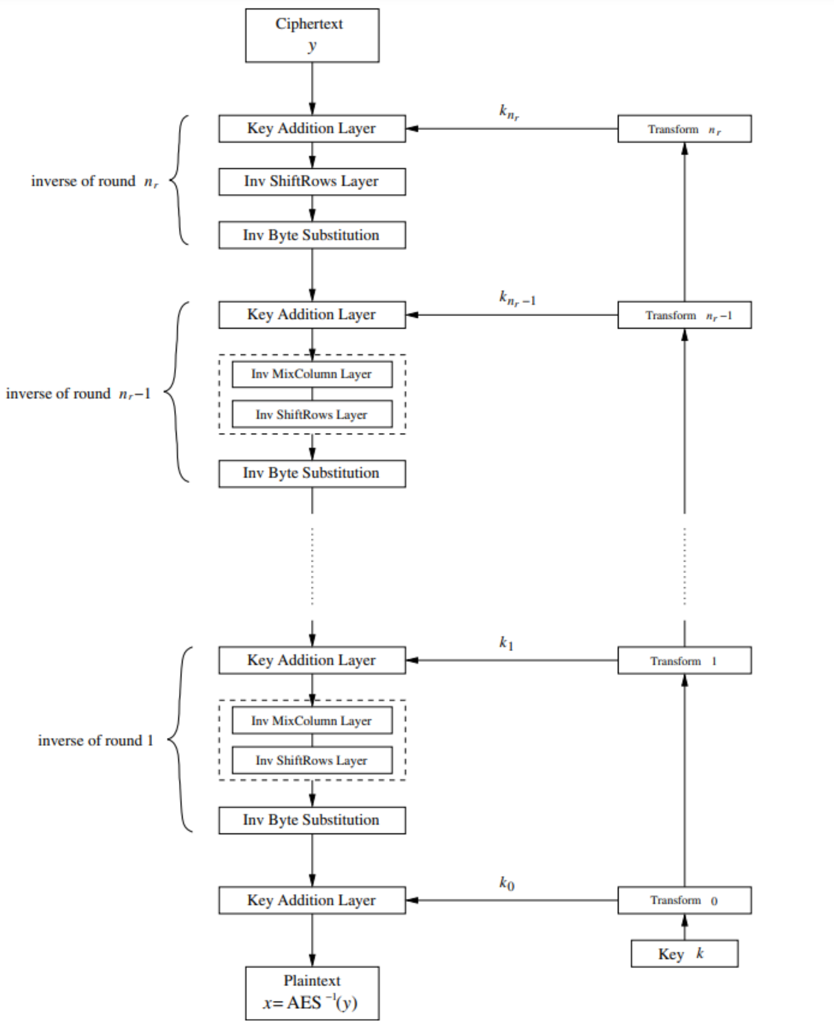

Figure 4.1: AES decryption block diagram

Since the last encryption round does not perform the MixColum operation, the first decryption round also does not contain the corresponding inverse layer. All other decryption rounds, however, contain all AES layers. In the following, we discuss the inverse layers of the general AES decryption round (Fig. 4.9). Since theXOR operation is its own inverse, the key addition layer in the decryption mode is the same as in the encryption mode: it consists of a row of plain XOR gates.

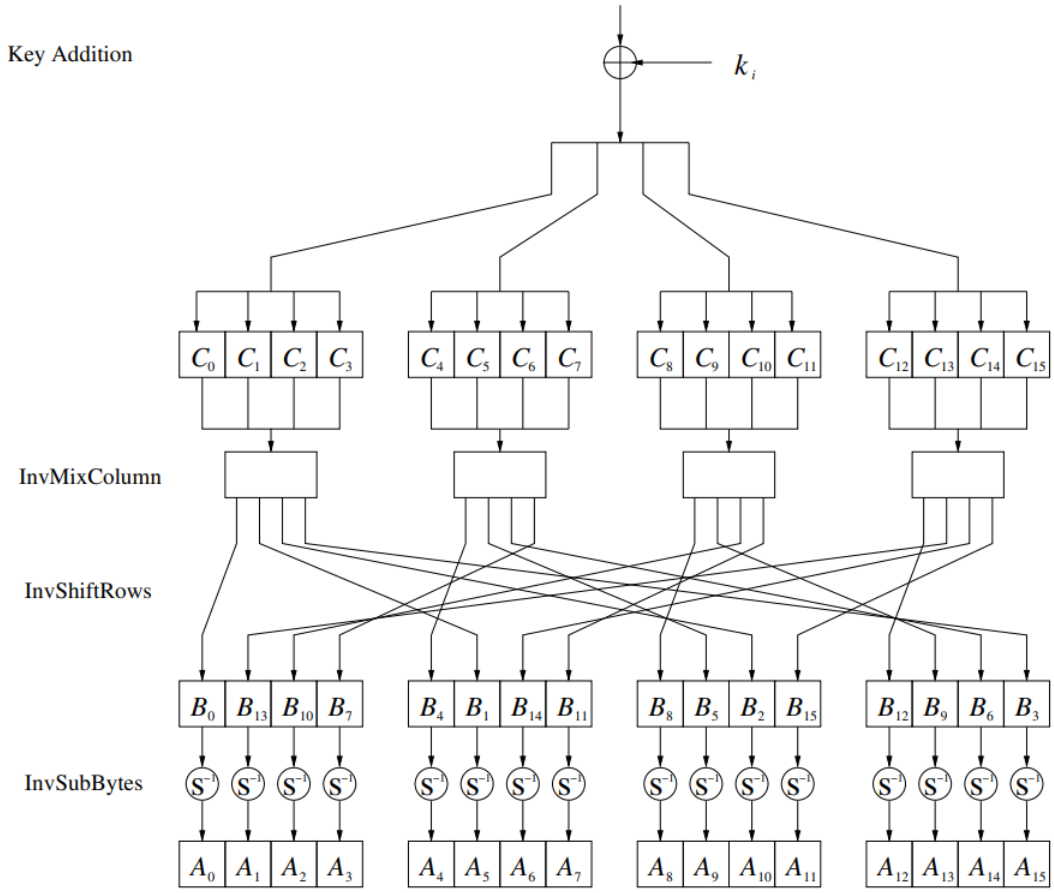

Figure 4.2: AES decryption round function $1, 2, ..., n_r - 1$

4.1 Inverse Mix Column Sublayer

The input is a 4-byte column of the State C which is multiplied by the inverse 4×4 matrix. The matrix contains constant entries. Multiplication and addition of the coefficients is done in GF(28).

$$[\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}]$$

The second column of output bytes (B_4, B_5, B_6, B_7) is computed by multiplying the four input bytes (C_4, C_5, C_6, C_7) by the same constant matrix, and so on.

4.2 Inverse ShiftRows Sublayer

In order to reverse the ShiftRows operation of the encryption algorithm, we must shift the rows of the state matrix in the opposite direction. The first row is not changed by the inverse ShiftRows transformation. If the input of the ShiftRows sublayer is given as a state matrix B = $(B_0, B_1, ..., B_{15})$:

$$\begin{bmatrix} B_0 & B_4 & B_8 & B_{12} \\ B1 & B_5 & B_9 & B_{13} \\ B_2 & B_6 & B_{10} & B_{14} \\ B_3 & B_7 & B_{11} & B_{15} \end{bmatrix}$$

4.3 Inverse Byte Substitution Layer

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

Figure 4.3: Inverse AES S-Box

4.4 Decryption Key Schedule

Since decryption round one needs the last subkey, the second decryption round needs the second-to-last subkey and so on, we need the subkey in reversed order as shown in Fig. 4.8. In practice this is mainly achieved by computing the entire key schedule first and storing all 11, 13 or 15 subkeys, depending on the number orrounds AES is using (which in turn depends on the three key lengths supported by AES). This precomputation adds usually a small latency to the decryption operation relative to encryption.

Chapter 5

AES IMPLEMENTATION

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "aes.h"

// substitute box
// size 16x16
const unsigned int sBox[16][16] = {
 { 0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76 },
   { 0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0 },
   { 0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15 },
   { 0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75 },
   { 0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84 },
   { 0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf },
   { 0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8 },
   { 0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2 },
   { 0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73 },
   { 0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb },
   { 0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79 },
   { 0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08 },
   { 0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a },
   { 0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e },
   { 0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf },
   { 0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }
 };

// reverse substitute box
```

```
// size 16x16
const unsigned int rsBox[16][16] = {
    { 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb },
    { 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb },
    { 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e },
    { 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25 },
    { 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92 },
    { 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84 },
    { 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06 },
    { 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b },
    { 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73 },
    { 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e },
    { 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b },
    { 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4 },
    { 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f },
    { 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef },
    { 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61 },
    { 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d }
    };

// round constants to find round keys
const unsigned int rCon[11] = { 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36 };

// GF(2^8) multiplication constants to do mix columns
const unsigned int mCon[BLOCK_SIZE][BLOCK_SIZE] = {
    { 0x02, 0x03, 0x01, 0x01 },
    { 0x01, 0x02, 0x03, 0x01 },
    { 0x01, 0x01, 0x02, 0x03 },
    { 0x03, 0x01, 0x01, 0x02 }
    };

// GF(2^8) multiplication constants to reverse mix columns
const unsigned int rmCon[BLOCK_SIZE][BLOCK_SIZE] = {
    { 0x0e, 0x0b, 0x0d, 0x09 },
    { 0x09, 0x0e, 0x0b, 0x0d },
    { 0x0d, 0x09, 0x0e, 0x0b },
    { 0x0b, 0x0d, 0x09, 0x0e }
    };

// this function produces all round keys
void key_schedule(unsigned int w[], unsigned int key[][BLOCK_SIZE]){
```

```
unsigned int row, column; // row and column for lookup
unsigned int temp[4];  // temporary holds the results
int i, j, k;    // counters for the loops and used as array "pointers"

// the first round key is the given key
// we store it to the first 4 words
// w0 w1 w2 w3 || w[0] ... w[15]
printf("Round keys:\n");
  for(i = 0; i < BLOCK_SIZE; i++){
   printf("w[%d] = ", i);
   for(j = 0; j < BLOCK_SIZE; j++){
    w[(i * 4) + j] = key[j][i];
     printf("%x ", w[(i * 4) + j]);
 }
 printf("\t");
  }

// all other round keys are found from the previous round keys
// start for 4, because we calculated the 4 words before
// 4 is the block size and 10 is the number of rounds
  for(i = 4; i < (4 * (10 + 1)); i++){
 // k is "pointer" to find wi-1
 k = (i - 1) * 4;
 // temp = w-1
     temp[0] = w[k + 0];
     temp[1] = w[k + 1];
     temp[2] = w[k + 2];
     temp[3] = w[k + 3];

     if(i % 4 == 0){
        // rot_word function
         k = temp[0];
         temp[0] = temp[1];
         temp[1] = temp[2];
         temp[2] = temp[3];
         temp[3] = k;

   // sub_word function
   for(j = 0; j < 4; j++){
    get2Bytes(temp[j], &row, &column);
    temp[j] = sub_byte(row, column);
   }

   // temp = sub_word(rot_word(temp)) XOR RCi/4
        temp[0] = temp[0] ^ rCon[i/4];
     }

 // j is "pointer" to find wi
     j = i * 4;
     // k is "pointer" to find wi-4
 k = (i - 4) * 4;

 // wi = wi-4 XOR temp
     w[j + 0] = w[k + 0] ^ temp[0];
     w[j + 1] = w[k + 1] ^ temp[1];
     w[j + 2] = w[k + 2] ^ temp[2];
     w[j + 3] = w[k + 3] ^ temp[3];
}
```

```
 // print round keys 1 - 10
 // round key 0(given key) printed before
 for(i = 4; i < 44; i++){
    printf("%s", (i % 4 == 0) ? "\n" : "\t");
  printf("w[%d] = ", i);
    for(j = 0; j < 4; j++){
     printf("%x ", w[(i * 4) + j]);
  }
   }
 printf("\n\n");

return;
}

// this function do result = a XOR b
void add_roundkey(unsigned int result[][BLOCK_SIZE], unsigned int a[][BLOCK_SIZE], unsigned int b[][BLOCK_SIZE]){
 int i, j; // counters for the loops

 for(i = 0; i < BLOCK_SIZE; i++)
  for(j = 0; j < BLOCK_SIZE; j++)
   result[i][j] = a[i][j] ^ b[i][j];

return;
}

// this function lookup in sBox table and return the new value
unsigned int sub_byte(unsigned int row, unsigned int column){
return sBox[row][column];
}

// this function shifts the rows to the left
// each row is shifted differently
void shift_rows(unsigned int state[][BLOCK_SIZE]){
 unsigned int temp; // temporary variables to do the shifts

 // first row is not shifted

 // second row is shifted left 1 time
 temp = state[1][0];
 state[1][0] = state[1][1];
 state[1][1] = state[1][2];
 state[1][2] = state[1][3];
 state[1][3] = temp;

 // third row is shifted left 2 times
 temp = state[2][0];
 state[2][0] = state[2][2];
 state[2][2] = temp;

 temp = state[2][1];
 state[2][1] = state[2][3];
 state[2][3] = temp;

 // fourth row is shifted left 3 times
 temp = state[3][0];
 state[3][0] = state[3][3];
```

```
 state[3][3] = state[3][2];
 state[3][2] = state[3][1];
 state[3][1] = temp;

return;
}

// this function mix the columns
// state column multiplied with mCon row
// instead of normal multiplication here we do GF(2^8) multiply
// and instead of addition here we do XOR operation
void mix_columns(unsigned int result[][BLOCK_SIZE], unsigned int state[][
    BLOCK_SIZE]){
 unsigned int sum = 0; // we hold the sum here
 int i, j, k;   // counters for the loops

    for(i = 0; i < BLOCK_SIZE; i++){
     for(j = 0; j < BLOCK_SIZE; j++){
         for(k = 0; k < BLOCK_SIZE; k++){
             sum ^= gfMul(state[k][i], mCon[j][k]);
            }
            result[j][i] = sum;
            sum = 0;
        }
    }

return;
}

// this function encrypts the plaintext
void encryption(unsigned int state[][BLOCK_SIZE], unsigned int w[]){
 unsigned int roundKey[BLOCK_SIZE][BLOCK_SIZE]; // round key
 unsigned int table[BLOCK_SIZE][BLOCK_SIZE];  // temp array to hold results

 unsigned int row, column; // row and column for sub_bytes lookup
 int i, j, k;    // counters for the loops

 // round 0
   // 1. add_roundkey
   getRoundKey(w, roundKey, 0);
   add_roundkey(state, state, roundKey);

 printf("add_roundkey:\n");
 printArray(state);

   // round 1-9
   // 1. sub_bytes
   // 2. shift_rows
   // 3. mix_columns
   // 4. add_roundkey
   for(k = 1; k < 10; k++){
  for(i = 0; i < BLOCK_SIZE; i++){
   for(j = 0; j < BLOCK_SIZE; j++){
    get2Bytes(state[i][j], &row, &column);
    state[i][j] = sub_byte(row, column);
   }
  }
  printf("sub_bytes:\n");
```

```
  printArray(state);

  shift_rows(state);

  printf("shift_rows:\n");
  printArray(state);

  mix_columns(table, state);

  printf("mix_columns:\n");
  printArray(table);

  getRoundKey(w, roundKey, k);
      add_roundkey(state, table, roundKey);

      printf("add_roundkey:\n");
      printArray(state);
 }

 // round 10
 // 1. sub_bytes
 // 2. shift_rows
 // 3. add_roundkey
   for(i = 0; i < BLOCK_SIZE; i++){
  for(j = 0; j < BLOCK_SIZE; j++){
   get2Bytes(state[i][j], &row, &column);
   state[i][j] = sub_byte(row, column);
  }
 }

 printf("sub_bytes:\n");
 printArray(state);

 shift_rows(state);

 printf("shift_rows:\n");
 printArray(state);

 getRoundKey(w, roundKey, 10);
    add_roundkey(state, state, roundKey);

 printf("add_roundkey:\n");
 printArray(state);

return;
}

// this function lookup in rsBox table and return the new value
unsigned int inv_sub_byte(unsigned int row, unsigned int column){
return rsBox[row][column];
}

// this function shifts the rows to the right
// each row is shifted differently
void inv_shift_rows(unsigned int state[][BLOCK_SIZE]){
 unsigned int temp; // temporary variable to do the shifts

 // first row is not shifted
```

```
	// second row is shifted right 1 time
	temp = state[1][3];
	state[1][3] = state[1][2];
	state[1][2] = state[1][1];
	state[1][1] = state[1][0];
	state[1][0] = temp;

	// third row is shifted right 2 times
	temp = state[2][0];
	state[2][0] = state[2][2];
	state[2][2] = temp;

	temp = state[2][1];
	state[2][1] = state[2][3];
	state[2][3] = temp;

	// fourth row is shifted right 3 times
	temp = state[3][0];
	state[3][0] = state[3][1];
	state[3][1] = state[3][2];
	state[3][2] = state[3][3];
	state[3][3] = temp;

return;
}

void inv_mix_columns(unsigned int result[][BLOCK_SIZE], unsigned int state[][BLOCK_SIZE]){
	unsigned int sum = 0; // we hold the sum here
	int i, j, k;    // counters for the loops
	unsigned int temp;  // temporary variable to hold the results

    for(i = 0; i < BLOCK_SIZE; i++){
        for(j = 0; j < BLOCK_SIZE; j++){
            for(k = 0; k < BLOCK_SIZE; k++){
             switch(rmCon[j][k]){
              // x = state[k][i]
              case 9:
               temp = gfMul(state[k][i], 2); // x * 2
               temp = gfMul(temp, 2);    // (x * 2) * 2
               temp = gfMul(temp ,2);    // ((x * 2) * 2) * 2
               temp ^= state[k][i];    // (((x * 2) * 2) * 2) + x = x * 9
      break;

              case 11:
               temp = gfMul(state[k][i], 2); // x * 2
               temp = gfMul(temp, 2);    // (x * 2) * 2
               temp ^= state[k][i];    // ((x * 2) * 2) + x
               temp = gfMul(temp, 2);    // (((x * 2) * 2) + x) * 2
               temp ^= state[k][i];    // ((((x * 2) * 2) + x) * 2) + x = x * 11
               break;

              case 13:
               temp = gfMul(state[k][i], 2); // x * 2
               temp ^= state[k][i];    // (x * 2) + x
               temp = gfMul(temp, 2);    // ((x * 2) + x) * 2
```

```c
                temp = gfMul(temp, 2);    // (((x * 2) + x) * 2) * 2
                temp ^= state[k][i];    // ((((x * 2) + x) * 2) * 2) + x = x * 13
                break;

              case 14:
                temp = gfMul(state[k][i], 2); // x * 2
                temp ^= state[k][i];    // (x * 2) + x
                temp = gfMul(temp, 2);    // ((x * 2) + x) * 2
                temp ^= state[k][i];    // (((x * 2) + x) * 2) + x
                temp = gfMul(temp, 2);    // ((((x * 2) + x) * 2) + x) * 2 = x * 14
                break;
    }
             sum ^= temp;
            }
            result[j][i] = sum;
            sum = 0;
        }
    }

return;
}

// this function encrypts the plaintext
void decryption(unsigned int state[][BLOCK_SIZE], unsigned int w[]){
 unsigned int roundKey[BLOCK_SIZE][BLOCK_SIZE]; // round key
 unsigned int table[BLOCK_SIZE][BLOCK_SIZE];  // temp array to hold results

 unsigned int row, column; // row and column for sub_bytes lookup
 int i, j, k, i1, i2;  // counters for the loops

 // round 0
 // 1. add_roundkey
 // 2. inv_shift_rows
 // 3. inv_sub_bytes
 getRoundKey(w, roundKey, 10);
    add_roundkey(state, state, roundKey);

 printf("\nadd_roundkey:\n");
 printArray(state);

 inv_shift_rows(state);

 printf("inv_shift_rows:\n");
 printArray(state);

   for(i = 0; i < BLOCK_SIZE; i++){
  for(j = 0; j < BLOCK_SIZE; j++){
   get2Bytes(state[i][j], &row, &column);
   state[i][j] = inv_sub_byte(row, column);
  }
 }

 printf("inv_sub_bytes:\n");
 printArray(state);

 // round 1-9
```

```
    // 1. add_roundkey
    // 2. inv_mix_columns
    // 3. inv_shift_rows
    // 4. inv_sub_bytes
    for(k = 9; k > 0; k--){
     getRoundKey(w, roundKey, k);
      add_roundkey(state, state, roundKey);

      printf("add_roundkey:\n");
      printArray(state);

      inv_mix_columns(table, state);

   printf("inv_mix_columns:\n");
   printArray(table);

      for(i1 = 0; i1 < BLOCK_SIZE; i1++){
       for(i2 = 0; i2 < BLOCK_SIZE; i2++){
        state[i1][i2] = table[i1][i2];
    }
   }

      inv_shift_rows(state);

   printf("inv_shift_rows:\n");
   printArray(state);

   for(i = 0; i < BLOCK_SIZE; i++){
    for(j = 0; j < BLOCK_SIZE; j++){
     get2Bytes(state[i][j], &row, &column);
     state[i][j] = inv_sub_byte(row, column);
    }
   }

   printf("inv_sub_bytes:\n");
   printArray(state);
  }

  // round 10
    // 1. add_roundkey
    getRoundKey(w, roundKey, 0);
    add_roundkey(state, state, roundKey);

  printf("add_roundkey:\n");
  printArray(state);

return;
}

// this function converts hexadecimal character to integer
int hexCharToDec(char hex){
 if(hex >= 48 && hex <= 57){ // ascii code for character 1-9
  return (hex - '0');  // as it happens, the ascii value of the characters 1-9 is greater than the value of '0'
 }else{
  switch(hex){
   case 'a':   // a hexadecimal is a number 10 to decimal. Similar for the rest
```

```
        return 10;

      case 'b':
        return 11;

      case 'c':
        return 12;

      case 'd':
        return 13;

      case 'e':
        return 14;

      case 'f':
        return 15;
    }
  }
}

// this function returns the integer value of the 2 bytes hex input
// input: xy || row = x and column = y
void get2Bytes(unsigned int a, unsigned int *row, unsigned int *column){
  // we need 2 bytes for the hexadecimal value and 1 more for '\0' character
  unsigned char temp[3];

  // convert the number to string
  sprintf(temp, "%x", a);

  if(strlen(temp) == 1){ // if number is smaller than 15, c saving 1 digit instead 2 digits
          // e.g. (14)dec = (0x0e)hex | C saving only e instead of 0e
    // add the '\0' character to 2nd position, because we need 1 slot for the hexadecimal number
    temp[1] = '\0';

    // find the row for the lookup
    *row = 0; // row will be 0, because number will have form like this: 0x0..
    //printf("(%c)hex = (%d)dec\n", temp[0], *row);

    // find the column for the lookup
    *column = hexCharToDec(temp[0]);
    //printf("(%c)hex = (%d)dec\n", temp[1], *column);
  }else{
    // add the '\0' character to 3rd position, because we need 2 slots for the hexadecimal number
    temp[2] = '\0';

    // find the row for the lookup
    *row = hexCharToDec(temp[0]);
    //printf("(%c)hex = (%d)dec\n", temp[0], *row);

    // find the column for the lookup
    *column = hexCharToDec(temp[1]);
    //printf("(%c)hex = (%d)dec\n", temp[1], *column);
  }

```

```c
return;
}

// this function return the 16bytes round key
void getRoundKey(unsigned int w[], unsigned int roundKey[][BLOCK_SIZE], int
    round){
 int i, j; // counters for the loops

 for(i = (round * 4); i < ((round * 4) + 4); i++){
    for(j = 0; j < 4; j++){
   roundKey[j][i - (round * 4)] = w[(i * 4) + j];
  }
   }

return;
}

// multiply two numbers in the GF(2^8)
// polynomial: x^8 + x^4 + x^3 + x + 1
// binary: 00011011  || hex: 0x1b
unsigned int gfMul(unsigned int a, unsigned int b){
 unsigned int r = 0;   // result
 unsigned int hi_bit_set; // high bit (leftmost)
 int i;      // counter for the loop

 for(i = 0; i < 8; i++) {
     if(b & 1)
         r ^= a;
     hi_bit_set = (a & 0x80);
     a <<= 1;

  if(hi_bit_set)
         a ^= 0x1b; // x^8 + x^4 + x^3 + x + 1
     b >>= 1;
 }

 // if result legnth is more than 8 bits
 if(r > 255 && r < 512)
  return r - 256;

 if(r > 511)
  return r - 512;

 // if result is max 8 bits
 return r;
}

// this function converts string to unsigned int array 4x4
void convertStringToBlock(char string[BYTES+1], unsigned int block[][
   BLOCK_SIZE]){
 int i, j; // counters for the loops

 for(i = 0; i < BLOCK_SIZE; i++){
  for(j = 0; j < BLOCK_SIZE; j++){
   block[j][i] = string[BLOCK_SIZE * i + j];
  }
 }

```

```c
return;
}

// this function converts unsigned int array 4x4 to string
void convertBlockToString(unsigned int block[][BLOCK_SIZE], char string[BYTES+1]){
 int i, j; // counters for the loops

 for(i = 0; i < BLOCK_SIZE; i++){
  for(j = 0; j < BLOCK_SIZE; j++){
   string[BLOCK_SIZE * i + j] = block[j][i];
  }
 }

return;
}

// this function prints the array
void printArray(unsigned int array[][BLOCK_SIZE]){
 int i, j; // counters for the loops

 for(i = 0; i < BLOCK_SIZE; i++){
  for(j = 0; j < BLOCK_SIZE; j++){
   printf("%x\t", array[i][j]);
  }
  printf("\n");
 }
 printf("\n");

return;
}

/* Run the encryption and decryption
 * Plaintext:  Two One Nine Two = (54 77 6F 20 4F 6E 65 20 4E 69 6E 65 20 54 77 6F)hex
 * Key:   Thats my Kung Fu = (54 68 61 74 73 20 6D 79 20 4B 75 6E 67 20 46 75)hex
 * we decrypting the output of encryption
 */
void test(){
 unsigned int state[BLOCK_SIZE][BLOCK_SIZE] = { // input
  { 0x54, 0x4f, 0x4e, 0x20 },
  { 0x77, 0x6e, 0x69, 0x54 },
  { 0x6f, 0x65, 0x6e, 0x77 },
  { 0x20, 0x20, 0x65, 0x6f }
  };

 unsigned int key[BLOCK_SIZE][BLOCK_SIZE] = { // encryption key
  { 0x54, 0x73, 0x20, 0x67 },
  { 0x68, 0x20, 0x4b, 0x20 },
  { 0x61, 0x6d, 0x75, 0x46 },
  { 0x74, 0x79, 0x6e, 0x75 }
  };

 // we have 10 round, so we need 40 words in array plus 4 for the given key
 // each word have 4 bytes, so we need 44 * 4 = 176
 unsigned int w[176];  // all round keys

```

```
 /****************************/

 // print the plaintext and key
 printf("Plaintext:\n");
 printArray(state);

 printf("Key:\n");
 printArray(key);

 // we calculate all round keys 1-10
 key_schedule(w, key);

 encryption(state, w);

 // print the result (ciphertext)
 printf("\nCiphertext:\n");
 printArray(state);

 printf("\n\n--------------------------------\n\n\n");

 decryption(state, w);

 // print the result (plaintext)
 printf("\nPlaintext:\n");
 printArray(state);

return;
}
// AES.H
#ifndef AES_H
#define AES_H

// AES 128bit = 16 bytes
#define BYTES 16

// block size is 128 bits = 16 bytes
// so we need 4x4 blocks
#define BLOCK_SIZE 4

// aes functions prototypes
void key_schedule(unsigned int [], unsigned int [][BLOCK_SIZE]);
void add_roundkey(unsigned int [][BLOCK_SIZE], unsigned int [][BLOCK_SIZE],
    unsigned int [][BLOCK_SIZE]);

unsigned int sub_byte(unsigned int, unsigned int);
void shift_rows(unsigned int [][BLOCK_SIZE]);
void mix_columns(unsigned int [][BLOCK_SIZE], unsigned int [][BLOCK_SIZE]);
void encryption(unsigned int [][BLOCK_SIZE], unsigned int []);

unsigned int inv_sub_byte(unsigned int, unsigned int);
void inv_shift_rows(unsigned int [][BLOCK_SIZE]);
void inv_mix_columns(unsigned int [][BLOCK_SIZE], unsigned int [][
   BLOCK_SIZE]);
void decryption(unsigned int [][BLOCK_SIZE], unsigned int []);

// secondary functions
int hexCharToDec(char);
void get2Bytes(unsigned int, unsigned int *, unsigned int *);
```

```c
void getRoundKey(unsigned int [], unsigned int [][BLOCK_SIZE], int);
unsigned int gfMul(unsigned int, unsigned int);
void convertStringToBlock(char [], unsigned int [][BLOCK_SIZE]);
void convertBlockToString(unsigned int [][BLOCK_SIZE], char []);

void printArray(unsigned int [][BLOCK_SIZE]);

void test();

#endif //AES_H


///////////        MAIN.C //////////////////////////////////////
#include <stdio.h>

#include "read.c"
#include "aes.c"

int main(){
 // last extra byte is for '\0' character
 unsigned char plainText[BYTES+1]; // plaintext
 unsigned char cipherText[BYTES+1]; // ciphertext
 unsigned char key[BYTES+1];    // encryption key
 unsigned char output[BYTES+1];  // string to prints the results

 int answer;  // user answer

 unsigned int state4x4[BLOCK_SIZE][BLOCK_SIZE]; // input
 unsigned int key4x4[BLOCK_SIZE][BLOCK_SIZE]; // encryption or decryption key

 // we have 10 round, so we need 40 words in array plus 4 for the given key
 // each word have 4 bytes, so we need 44 * 4 = 176
 unsigned int w[176];  // all round keys

 // print the menu to user and read the answer
 printf("--------------------------\n1. Encryption\n2. Decryption\n3. Encrypt and Decrypt\n4. Run example\n--------------------------\n");
 do{
  printf("Choose action: ");
  scanf("%d", &answer);

 }while(answer < 1 || answer > 4);

 switch(answer){
  case 1:
   // read the plaintext
   printf("Text: ");
   readInput(plainText);

   // read the key
   printf("Key: ");
   readInput(key);

   // convert plaintext and key to 4x4 blocks
   convertStringToBlock(plainText, state4x4);
   convertStringToBlock(key, key4x4);
```

```
    // we calculate all round keys 1-10
    key_schedule(w, key4x4);

    // encrypt the plaintext
    encryption(state4x4, w);

    // print the output to user as string
    convertBlockToString(state4x4, output);
    output[BYTES] = '\0';
    printf("Ciphertext: %s", output);
    break;

  case 2:
    // read the ciphertext
    printf("Text: ");
    readInput(cipherText);

    // read the key
    printf("Key: ");
    readInput(key);

    // convert ciphertext and key to 4x4 blocks
    convertStringToBlock(cipherText, state4x4);
    convertStringToBlock(key, key4x4);

    // we calculate all round keys 1-10
    key_schedule(w, key4x4);

    // decrypt the ciphertext
    decryption(state4x4, w);

    // print the output to user as string
    convertBlockToString(state4x4, output);
    output[BYTES] = '\0';
    printf("Plaintext: %s", output);
    break;

  case 3:
    // read the plaintext
    printf("Text: ");
    readInput(plainText);

    // read the key
    printf("Key: ");
    readInput(key);

    // convert plaintext and key to 4x4 blocks
    convertStringToBlock(plainText, state4x4);
    convertStringToBlock(key, key4x4);

    // we calculate all round keys 1-10
    key_schedule(w, key4x4);

    printf("\nENCRYPTION\n");
    // encrypt the plaintext
    encryption(state4x4, w);

```

```c
    // print the output to user as string
    convertBlockToString(state4x4, output);
    output[BYTES] = '\0';
    printf("Ciphertext: %s\n", output);
    printf("END OF ENCRYPTION\n");

    printf("\n\nDECRYPTION\n");
    // decrypt the ciphertext
    decryption(state4x4, w);

    // print the output to user as string
    convertBlockToString(state4x4, output);
    output[BYTES] = '\0';
    printf("Plaintext: %s\n", output);
    printf("END OF DECRYPTION\n");
    break;

   case 4:
    test();
    break;
 }

return 0;
}
//////////////////////      READ.C //////////////////////////////////////////////
#include <stdio.h>
#include <string.h>

#include "aes.h"

void readInput(char []);

// this function reads the user input
void readInput(char input[BYTES+1]){
 unsigned short int sizeCheck; // temporary saving the length of given input

 do{
  // read the input
  gets(input);

  // clear the input buffer
  fflush(stdin);

  // save the length of input
  sizeCheck = strlen(input);
 }while(sizeCheck != BYTES);

return;
}
```

Chapter 6

CONCLUSION

- AES is a modern block cipher which supports three key lengths of 128, 192 and 256 bit. It provides excellent long-term security against brute-force attacks.
- AES has been studied intensively since the late 1990s and no attacks have been found that are better than brute-force.
- AES is not based on Feistel networks. Its basic operations use Galois field arithmetic and provide strong diffusion and confusion.
- AES is part of numerous open standards such as IPsec or TLS, in addition to being the mandatory encryption algorithm for US government applications. It seems likely that the cipher will be the dominant encryption algorithm for many years to come.
- AES is efficient in software and hardware.

Bibliography

[1] Christof Paar, Jan Pelzl., "Understanding Cryptography A Textbook for Students and Practitioners", *Springer Berlin Heidelberg*

[2] $https://en.wikipedia.org/wiki/AdvancedEncryptionStandard$