AES 128 Documentation

Prabal Das

Supervisor: Dr. Sabyasachi Karati

Date: September 16, 2024

Abstract

This AES 128 documentation is based on what we implemented in our class at ISI, Kolkata. In our class, we have implemented the whole AES 128 encryption and decryption, along with five modes of operation. In this documentation, we have discussed all of the above along with some basic introduction in what is AES, how the mathematical operations are done in AES, how it is implemented in basic C language. We have also produced the pseudocode corresponding to all the code we have done in our class. At last, we have given a basic comparison of all these modes with respect to how much time it takes to encrypt and decrypt a message.

Contents

0.1 Introduction

The **Advanced Encryption Standard (AES)**, also known by its original name **Rijndael** is a specification for the encryption of electronic data established by the U.S. National Institute of Standards and Technology (NIST) in 2001.
AES is a variant of the Rijndael block cipher developed by two Belgian cryptographers, **Joan Daemen** and **Vincent Rijmen**, who submitted a proposal to NIST during the AES selection process. AES was announced by the NIST as **U.S. FIPS PUB 197 (FIPS 197)** on November 26, 2001.
AES is a symmetric block cipher that can process data blocks of **128 bits**, using cipher keys with lengths of **128**, **192**, and **256 bits** and therefore these different **flavors** also referred as **AES-128**, **AES-192**, and **AES-256**.
This documentation is based on **AES-128** and we will see how the algorithms work along with its mathematical background.

0.2 Notation and Conventions

0.2.1 Inputs and Outputs

The input and output for the AES algorithm each consist of sequences of 128 bits (digits with values of 0 or 1). These sequences will sometimes be referred to as **Blocks** and the number of bits they contain will be referred to as their **length**. The key for AES-128 is 128 bits long.

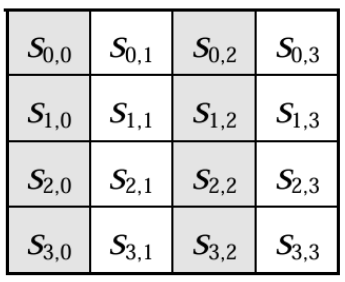
0.2.2 Bytes

The basic unit for processing in the AES algorithm is a **byte**, a sequence of eight bits treated as a single entity. So, the input, output and key all are **16 bytes** long. Each bytes are also represented in **Hexadecimal** representation, like the element **{01100011}** can be represented as **{63}**.

0.2.3 The State

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the **State**. This state is a 4-by-4 matrix, each cell is of **1 Bytes** i.e. each state matrix holds **16 bytes** data.

State array



$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

This is a visualization of how state matrix looks like, where each cell $S_{i,j}$ is one byte.

0.2.4 Word

The four bytes in each column of the State array form **32-bit *Words***, where the row number **j** provides an index for the four bytes within each word. This data structure is very useful in many operations, which we will see shortly.
Each state matrix can be viewed as one-dimensional array of four *Words* W_0, W_1, W_2, W_3, where $W_j = S_{0,j}S_{1,j}S_{2,j}S_{3,j}$

0.3 Mathematical Preliminaries

All byte values in the AES algorithm is an element of finite field $GF(2^8)$ and can be represented in polynomial forms. Like, $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$ can be represented as

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$$

For example, $\{01100011\}$ identifies the specific finite field element $x^6 + x^5 + x + 1$.

0.3.1 Addition

The addition of two elements in a finite field is achieved by **adding** the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the **XOR** operation of the binary rep of two elements.
For example, the following expressions are equivalent to one another:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 \text{ (poly. rep)}$$
$$\{01010111\} \oplus \{10000011\} = \{11010100\} \text{ (binary rep)}$$
$$\{57\} \oplus \{83\} = \{d4\} \text{ (Hex. rep)}$$

0.3.2 Multiplication

In the polynomial representation, multiplication in $GF(2^8)$ (denoted by $\bullet$) corresponds with the multiplication of polynomials modulo an **irreducible polynomial of degree 8**. For the AES algorithm, this irreducible polynomial is

$$m(x) = x^8 + x^4 + x^3 + x + 1 \text{ or } \{01\}\{1b\} \text{ (in Hex)}$$

The modular reduction by $m(x)$ ensures that the result will be a binary polynomial of degree less than 8, and thus can be represented by a byte. Unlike addition, there is no simple operation at the byte level that corresponds to this multiplication. That's why we will first define `xtime()`, then the multiplication of two bytes.

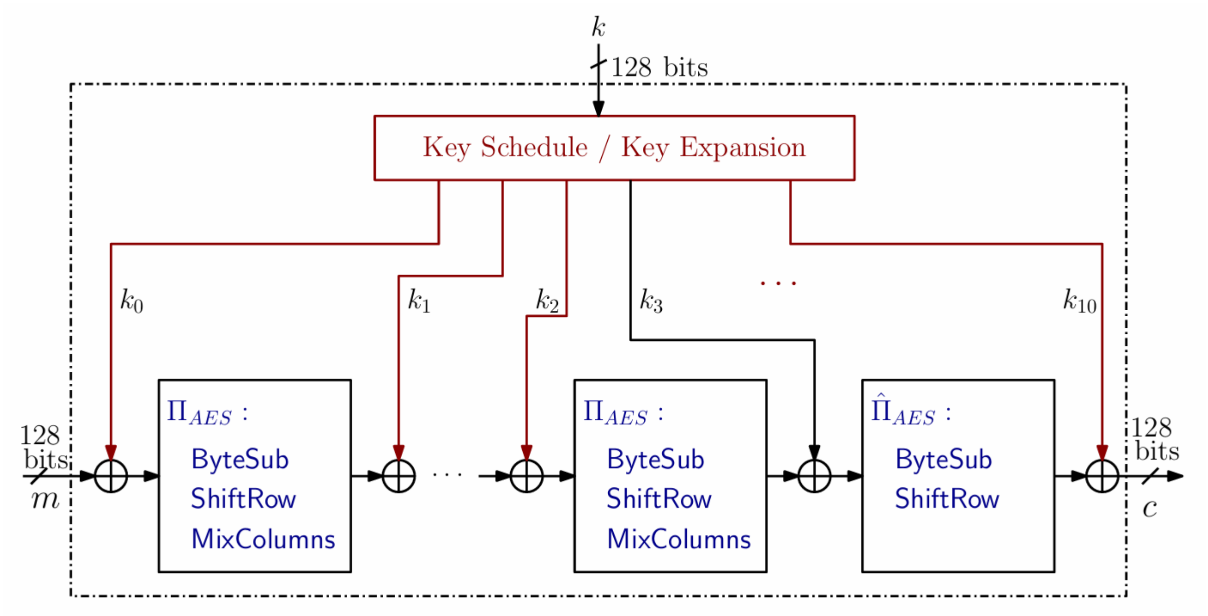
- <u>`xtime()`</u>: Let, **b** be a byte and it's polynomial representation is $b(x) = b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0$. `xtime()` is an algorithm which on input **b** returns $x \bullet b(x)$, which another element in the field.
  If $b_7 = 0$, then $x \bullet b(x)$ is nothing but left shift of 1 bit of b, no reduction is needed. As an example, if $b = \{01011011\}$, then $x \bullet b(x)$ will be $b = \{10110110\}$.
  If $b_7 = 1$, then $x \bullet b(x)$ will be of 8 degree polynomial, which need to be reduced. So, we do 1 bit left shift then xor with $\{1b\}$, to get the multiplication. As an example, if $b = \{11011011\}$, then $x \bullet b(x)$ will be $b = \{10101101\}$.

- ByteMultiply(,): `ByteMultiply(,)` is an algorithm which on input **a, b** returns $a(x) \bullet b(x)$ and this is done by iteratively calling `xtime()`. For i-th bit b_i, `xtime()` is called i-th time and if $b_i = 1$, then reduction is done. Although this can be done more efficiently, instead of calling `xtime()` i-th for each i, we can compute `xtime()` one time on the previous iteration result for each iteration.
  For example, let a = {57}, b = {13}, then $b_i = 1$ at $i = 0, 1, 4$. So, we will compute `xtime()` for each iteration based on previous iteration output.

$$\begin{aligned}
\{57\}\bullet\{02\} &= \texttt{xtime(\{57\})} = \{\text{ae}\}\\
\{57\}\bullet\{04\} &= \texttt{xtime(\{ae\})} = \{47\}\\
\{57\}\bullet\{08\} &= \texttt{xtime(\{47\})} = \{8\text{e}\}\\
\{57\}\bullet\{10\} &= \texttt{xtime(\{8e\})} = \{07\}\\
\text{Now, } \{57\}\bullet\{13\} = \{57\}\bullet(\{01\} \oplus \{02\} \oplus \{10\}) &= \{57\} \oplus \{ae\} \oplus \{07\} = \{fe\}
\end{aligned}$$

0.4 Algorithms

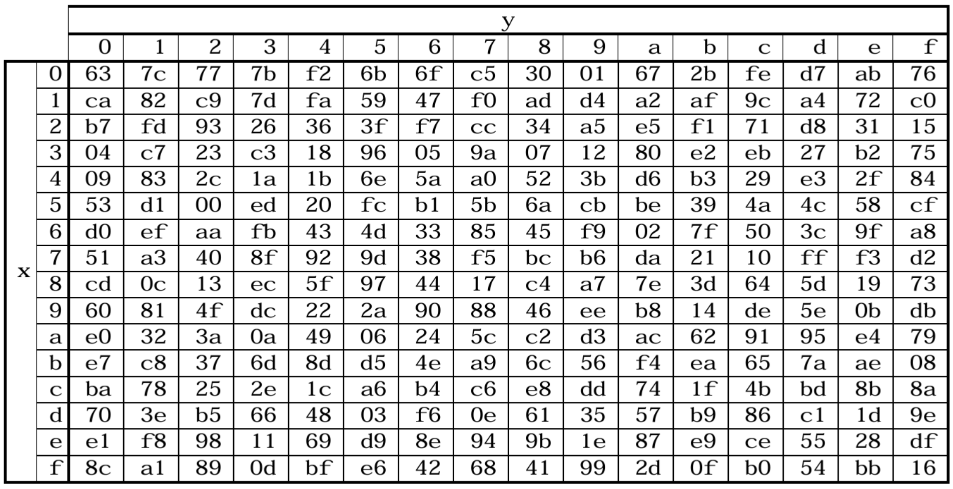
In AES-128, the plain text, cipher text and keys, all are 128 bit long. In this block cipher, each cipher consists of three function `SBox(), ShiftRows(), MixColumns()` and we iterate this cipher with some keys, which are generated by `KeyExpansionFunction()` function.



AES 128

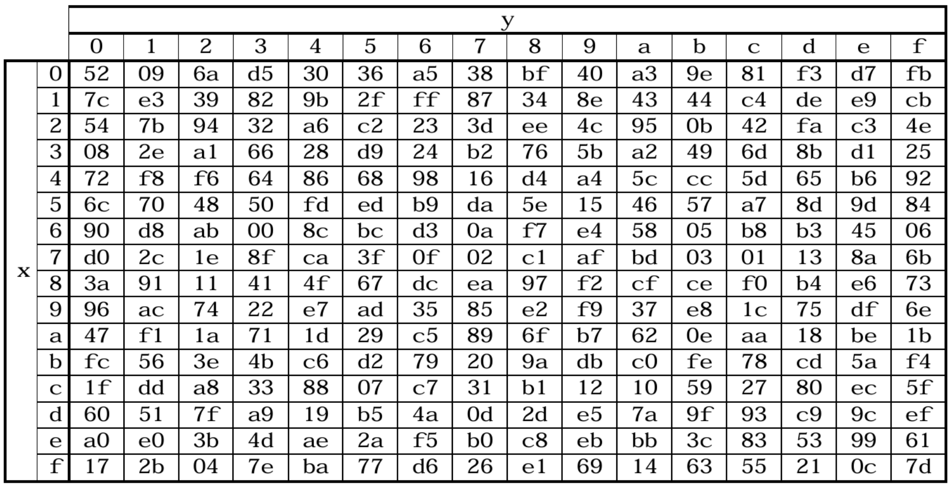
In the above diagram, m, k all are 128 bit input and c is 128 bit output and π_{AES} is the cipher which is iterated 10 times. The `KeyExpansionFunction()` function on input k, generates 11 keys, say $k_0, k_1, \cdots, K_{10}$, which is xor-ed with the state blocks.
Notice, in the iterations, the last cipher namely π_{AES} will not contain the `MixColumns()` operation. So, this is the overall view of how AES-128 cipher works, now we will discuss functionalities of each algorithms.

0.4.1 Components



		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
x	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

S-box: substitution values for the byte xy **(in hexadecimal format).**



		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

Inverse S-box: substitution values for the byte xy **(in hexadecimal format).**

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Figure 1: mix_columns_matrix

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

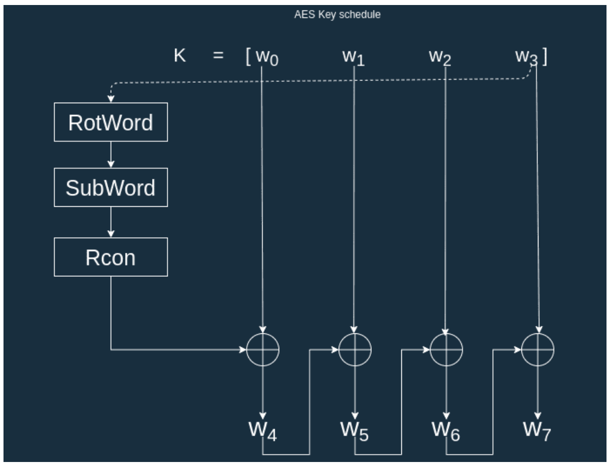
Figure 2: inv_mix_columns_matrix

Table 1: round constants in Hex

i	1	2	3	4	5	6	7	8	9	10
round_constants[i]	01	02	04	08	10	20	40	80	1B	36

0.4.2 `KeyExpansionFunction()`

As we discussed previously, in AES-128, each 128 bit key can we viewed as **4 *Words***, say W_0, W_1, W_2, W_3, each of 32 bits and the below diagram shows how with those *Words*, we can generate another 4 *Words*, which will form another key. For encryption/decryption we need 11 keys i.e. 44 *Words* and it will be generated by `KeyExpansionFunction()` algorithm.



In our implementation, we take a *Word* array, *KeyWords[44]*, which will store all the 44 *Words* and components of each *Word*, W_i is *bytes[j]*, j = 0,1,2,3.

Algorithm 1 `KeyExpansionFunction(Word * KeyWords[44])`

```
1: $i \leftarrow 4$
2: $j$
3: Word temp
4: while $i < 44$ do
5:     for $j = 0$ to 3 do
6:         temp.bytes[j] ← KeyWords[i-1]->bytes[j]
7:     end for
8:     if $i \mod 4 = 0$ then
9:         RotWord(temp)
10:        SubWord(temp)
11:        Rcon(temp.bytes[0])
12:    end if
13:    KeyWords[i] ← xorWords(KeyWords[i-4], temp, KeyWords[i])
14:    $i \leftarrow i + 1$
15: end while
16: return KeyWords
```

This is the pseudocode which will give all the 44 round keys. Now we will see the in between functions `RotWord()`, `SubWord()`, `Rcon()` and `xorWords()`.

- `RotWord()`: This function will rotate the bytes of `temp` cyclically. The corresponding pseudocode is as follows

Algorithm 2 `RotWord(Word * temp)`

```
1: procedure RotWord(temp)
2:     t ← temp.bytes[0]
3:     temp.bytes[0] ← temp.bytes[1]
4:     temp.bytes[1] ← temp.bytes[2]
5:     temp.bytes[2] ← temp.bytes[3]
6:     temp.bytes[3] ← t
7: end procedure
```

- `SubWord()`: This function will map each bytes of the Word `temp` to the s_box. The pseudocode is as follows.

Algorithm 3 `SubWord(Word * temp)`

```
1: for $j = 0$ to 3 do
2:     temp.bytes[j] ← s_box[(temp.bytes[j] >> 4) & 0x0F][temp.bytes[j] & 0x0F]
3: end for
```

- `Rcon()`: This function will do the xor of first byte with the value given by `round_constant` table. The pseudocode is as follows.

Algorithm 4 `Rcons(Word * temp.bytes[0])`

```
1: temp.bytes[0] ← temp.bytes[0] ^ round_constants[(i/4)-1]
```

- `xorWords()`: This function will do the xor of two Words. The pseudocode is as follows.

Algorithm 5 `xorWords(Word * a, Word * b, Word * result)`

```
1: procedure xorWords($a$, $b$, $result$)
2:     for $j = 0$ to 3 do
3:         $result.bytes[i] \leftarrow a.bytes[i] \oplus b.bytes[i]$
4:     end for
5:     return $result$
6: end procedure
```

This is all about how `KeyExpansionFunction()` generates all the keys for encryption and decryption.

0.4.3 `Encryption()`

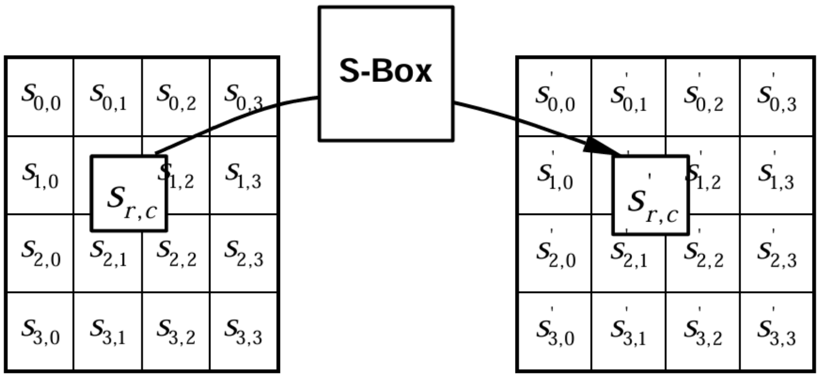
As we discussed earlier, AES-128 encryption is done by iteratively running the round function and followed by key xoring. The below pseudocode tells how the encryption is done on `msg[4][4]` and we get the cipher text `cipher[4][4]`.

Algorithm 5 `Encryption(unsigned char msg[4][4], unsigned char cipher[4][4], Word * KeyWords[])`

```
 1: procedure ENCRYPTION(msg[4][4], cipher[4][4], KeyWords[11])
 2:     state[4][4] ← msg
 3:     for $i \leftarrow 0$ to 3 do
 4:         for $j \leftarrow 0$ to 3 do
 5:             state[$i$][$j$] ← state[$i$][$j$] ⊕ KeyWords[$j$] → bytes[$i$]        ▷ Key Xor
 6:         end for
 7:     end for
 8:     for round ← 1 to 9 do
 9:         SBox(state)
10:         ShiftRows(state)
11:         MixColumns(state)
12:         for $i \leftarrow 0$ to 3 do
13:             for $j \leftarrow 0$ to 3 do
14:                 state[$i$][$j$] ← state[$i$][$j$] ⊕ KeyWords[$4 \times \text{round} + j$] → bytes[$i$]
15:             end for
16:         end for
17:     end for                                                        ▷ For last round
18:     SBox(state)
19:     ShiftRows(state)
20:     for $i \leftarrow 0$ to 3 do
21:         for $j \leftarrow 0$ to 3 do
22:             state[$i$][$j$] ← state[$i$][$j$] ⊕ KeyWords[$40 + j$] → bytes[$i$]
23:         end for
24:     end for
25:     for $i \leftarrow 0$ to 3 do
26:         for $j \leftarrow 0$ to 3 do
27:             cipher[$i$][$j$] ← state[$i$][$j$]                      ▷ Copying to the cipher matrix
28:         end for
29:     end for
30: end procedure
```

Since, we have seen how the AES-128 encryption is done, now we will see the functions `SBox()`, `ShiftRows()` and `MixColumns()`, how it works and its pseudocode.

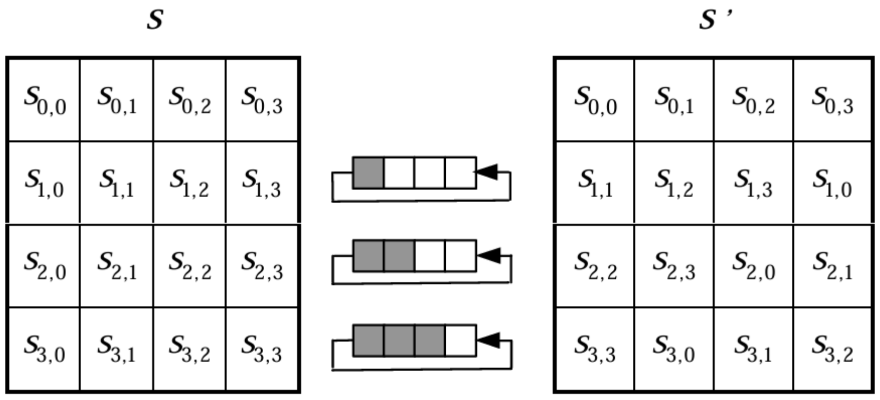
- SBox(): This function maps a state to another state by the s_box. The below figure shows how this is done and then we will see the pseudocode.


S-Box
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,2}$ $S_{1,3}$
$S_{r,c}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S'_{0,0}$ $S'_{0,1}$ $S'_{0,2}$ $S'_{0,3}$
$S'_{1,0}$ $S'_{1,2}$ $S'_{1,3}$
$S'_{r,c}$
$S'_{2,0}$ $S'_{2,1}$ $S'_{2,2}$ $S'_{2,3}$
$S'_{3,0}$ $S'_{3,1}$ $S'_{3,2}$ $S'_{3,3}$

Algorithm 6 SBox(unsigned char state[4][4])

```
1: procedure SBox(state[4][4])
2:     for $i \leftarrow 0$ to 3 do
3:         for $j \leftarrow 0$ to 3 do
4:             state[$i$][$j$] $\leftarrow$ s_box[(state[$i$][$j$] $\gg$ 4)&0$x$0$F$][state[$i$][$j$]&0$x$0$F$]
5:         end for
6:     end for
7: end procedure
```

- ShiftRows(): In the ShiftRows() transformation, the bytes in the last three rows of the State are cyclically left shifted over different numbers of bytes i.e. i-th row is shifted to i-many cells in the state matrix. The following figure shows how it is done.


S
S'
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,0}$ $S_{1,1}$ $S_{1,2}$ $S_{1,3}$
$S_{2,0}$ $S_{2,1}$ $S_{2,2}$ $S_{2,3}$
$S_{3,0}$ $S_{3,1}$ $S_{3,2}$ $S_{3,3}$
$S_{0,0}$ $S_{0,1}$ $S_{0,2}$ $S_{0,3}$
$S_{1,1}$ $S_{1,2}$ $S_{1,3}$ $S_{1,0}$
$S_{2,2}$ $S_{2,3}$ $S_{2,0}$ $S_{2,1}$
$S_{3,3}$ $S_{3,0}$ $S_{3,1}$ $S_{3,2}$

Now we will also see the corresponding pseudocode which we did in our implementation.

Algorithm 7 `ShiftRows(unsigned char state[4][4])`

```
procedure ShiftRows(state[4][4])
    temp: unsigned char                                ▷ Temporary variable for shifting
    for $i \leftarrow 1$ to 3 do
        for $j \leftarrow 0$ to $i - 1$ do
            temp ← state[$i$][0]
            for $k \leftarrow 0$ to 2 do
                state[$i$][$k$] ← state[$i$][$k+1$]
            end for
            state[$i$][3] ← temp
        end for
    end for
end procedure
```

- `MixColumns()`: In this transformation, we multiply the state matrix with the **mix_columns_matrix** to get another state matrix. The following figure tells the procedure and the pseudocode which we did in our implementation.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

Algorithm 8 `MixColumns(unsigned char state[4][4])`

```
procedure MixColumns(state[4][4])
    unsigned char temp_state[4][4] ← {0}               ▷ Temporary state to hold the result
    for $j \leftarrow 0$ to 3 do
        for $i \leftarrow 0$ to 3 do
            temp_state[$i$][$j$] ← ByteMultiply(mix_columns_matrix[$i$][0], state[0][$j$]) $\oplus$
                ByteMultiply(mix_columns_matrix[$i$][1], state[1][$j$]) $\oplus$
                ByteMultiply(mix_columns_matrix[$i$][2], state[2][$j$]) $\oplus$
                ByteMultiply(mix_columns_matrix[$i$][3], state[3][$j$])
        end for
    end for
    for $i \leftarrow 0$ to 3 do
        for $j \leftarrow 0$ to 3 do
            state[$i$][$j$] ← temp_state[$i$][$j$]          ▷ Copying to the state matrix
        end for
    end for
end procedure
```

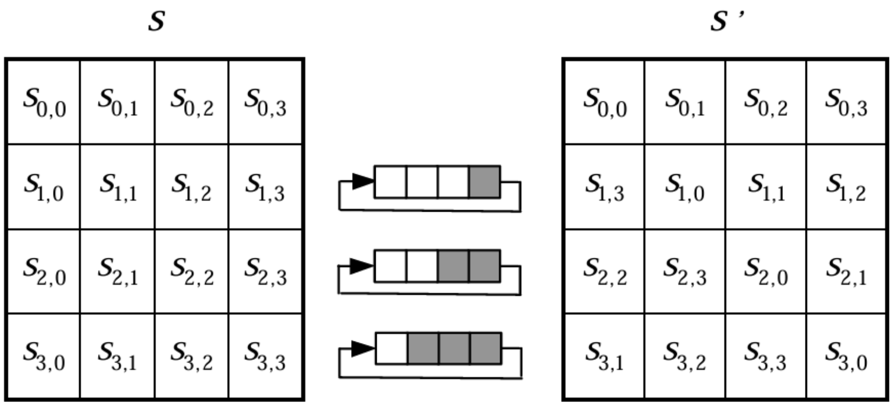
0.4.4 `Decryption()`

Algorithm 9 `Decryption(unsigned char cipher[4][4], unsigned char msg[4][4], Word * KeyWords[])`

```
procedure DECRYPTION(cipher[4][4], msg[4][4], KeyWords[11])
    unsigned char state[4][4]                                        ▷ Temporary variable
    int $i, j,$ round $\leftarrow 9$
    for $i \leftarrow 0$ to 3 do
        for $j \leftarrow 0$ to 3 do
            state[$i$][$j$] $\leftarrow$ cipher[$i$][$j$]                          ▷ Copying to state matrix
        end for
    end for
    for $i \leftarrow 0$ to 3 do
        for $j \leftarrow 0$ to 3 do
            state[$i$][$j$] $\leftarrow$ state[$i$][$j$] $\oplus$ KeyWords[$40 + j$] $\rightarrow$ bytes[$i$]    ▷ Add round key
        end for
    end for
    for round $\leftarrow 9$ downto 1 do
        InvShiftRows(state)
        InvSBox(state)
        for $i \leftarrow 0$ to 3 do
            for $j \leftarrow 0$ to 3 do
                state[$i$][$j$] $\leftarrow$ state[$i$][$j$] $\oplus$ KeyWords[$4 \times$ round $+ j$] $\rightarrow$ bytes[$i$]
            end for
        end for
        InvMixColumns(state)
    end for
    InvShiftRows(state)                                              ▷ Last round
    InvSBox(state)
    for $i \leftarrow 0$ to 3 do
        for $j \leftarrow 0$ to 3 do
            state[$i$][$j$] $\leftarrow$ state[$i$][$j$] $\oplus$ KeyWords[$j$] $\rightarrow$ bytes[$i$]         ▷ Add Round Key
        end for
    end for
    for $i \leftarrow 0$ to 3 do
        for $j \leftarrow 0$ to 3 do
            msg[$i$][$j$] $\leftarrow$ state[$i$][$j$]     ▷ Copy the final result back to the message space
        end for
    end for
end procedure
```

Like AES-128 encryption, AES-128 decryption is also done by iteratively running the round function, followed by key xoring and the decryption circuit almost same as encryption circuit. The above pseudocode tells how the decryption is done on `cipher[4][4]` and we get the cipher text `msg[4][4]`. Like encryption, in decryption there are three transformations e.g. `InvShiftRows()`, `InvSBox()` and `InvMixColumns()`, which are essentially same as the transformation used in encryption. So, we will briefly discuss all these three transformations.

- <u>`InvShiftRows()`</u>: This transformation is same as `ShiftRows()` transformation, except it cyclically rotates to right. The below diagram says how it is done.



S
S '
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1,2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2,3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,3}$ $s_{1,0}$ $s_{1,1}$ $s_{1,2}$
$s_{2,2}$ $s_{2,3}$ $s_{2,0}$ $s_{2,1}$
$s_{3,1}$ $s_{3,2}$ $s_{3,3}$ $s_{3,0}$

- <u>`InvSBox()`</u>: This transformation is the inverse of the byte substitution transformation, in which the inverse S-box is applied to each byte of the State.

- <u>`InvMixColumns()`</u>:This transformation is same as the transformation `MixColumns()`, except the mix_columns_matrix is replaced by inv_mix_columns_matrix. The following diagram shows how it is done.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

This is how the AES-128 decryption works.
So, we have seen that how the whole AES-128 block cipher works, how the `KeyExpansionFunction()` algorithm generates all the round keys from the master key and it is used in `Encryption()` function to encrypt a message as well as how `Decryption()` algorithm decrypts the cipher text to get back the plain text. But the plain texts are of 128 bit long and we want to encrypt arbitrary long message. For that we will use this block cipher in different **Modes of Operations** to do our task.

0.5 Modes of Operations

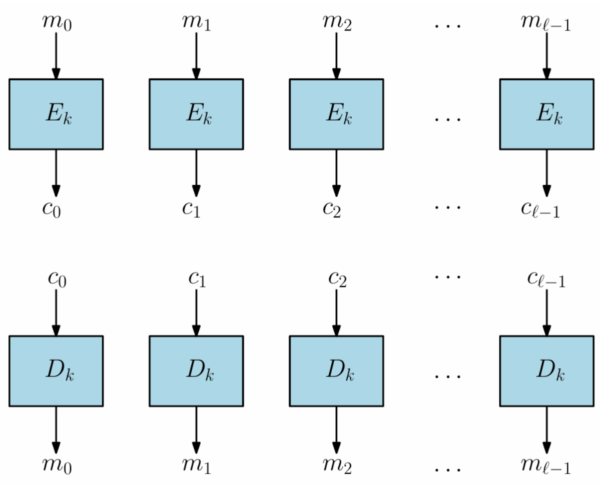
There are different types of modes of operations and in our implementation we have done five modes of operations namely,

- Electronic Code Book mode (ECB mode)
- Cipher Block Chaining mode (CBC mode)
- Output FeedBack mode (OFB mode)
- Cipher FeedBack mode (CFB mode) and
- Counter mode (CTR mode)

Let's discuss each modes one by one.

0.5.1 Electronic Code Book mode

The simplest of the encryption modes is the electronic codebook (ECB) mode. The message is divided into blocks, and each block is encrypted separately.



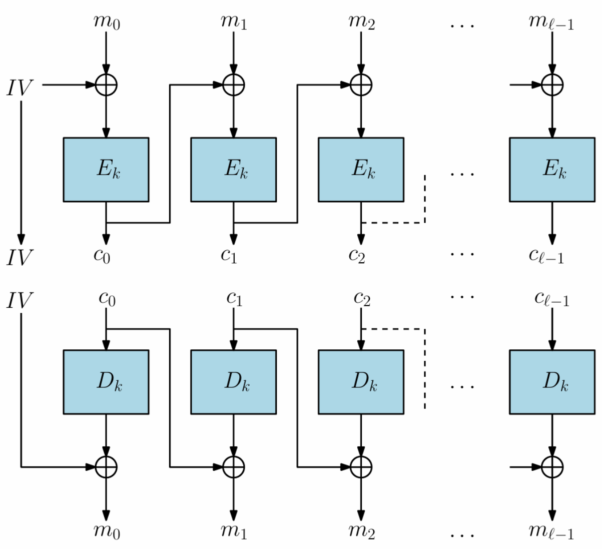
The above diagram shows how ECB mode looks like and below pseudocode is the encryption function for ECB mode, which has the arguments `filePointer, KeyWords`, which are the plain text file and set of all keys resp. The file will be encrypted by those keys using AES-128 encryption function `Encryption()`. The decryption function will also be the same with some changes like input file will be a cipher text file, paddiing will not be needed and rest same, which we left for the reader.

Algorithm 10 `ECBEncryption(FILE * filePointer, Word * KeyWords[])`

```
procedure ECBENCRYPTION(filePointer, KeyWords[11])
    unsigned char buffer[16], state[4][4], cipher[4][4]
    int $i, j$, count
    filePointer ← fopen("input.txt", "r")                ▷ Open "input.txt" file
    OutfilePointer ← fopen("output_ecb.txt", "w")        ▷ Open "output_ecb.txt" file
    while not feof(filePointer) do
        count ← fread(buffer, 1, 16, filePointer)
        if count $\neq$ 16 then
            $k \leftarrow 16 - \text{count}$
            for $i$ ← count to 15 do
                buffer[$i$] ← unsigned char($k$)          ▷ padding PKCS#7
            end for
        end if
        for $i \leftarrow 0$ to 3 do
            for $j \leftarrow 0$ to 3 do
                state[$i$][$j$] ← buffer[$i \times 4 + j$]       ▷ Creating state matrix from buffer
            end for
        end for
        cipher ← Encryption(state, cipher, KeyWords)     ▷ Perform AES Encryption
        for $i \leftarrow 0$ to 3 do
            for $j \leftarrow 0$ to 3 do
                fprintf(OutfilePointer, "%$c$", cipher[$i$][$j$])   ▷ Write in output file
            end for
        end for
    end while
end procedure
```

0.5.2 Cipher Block Chaining mode

In CBC mode, each block of plaintext is XORed with the previous ciphertext block before being encrypted.



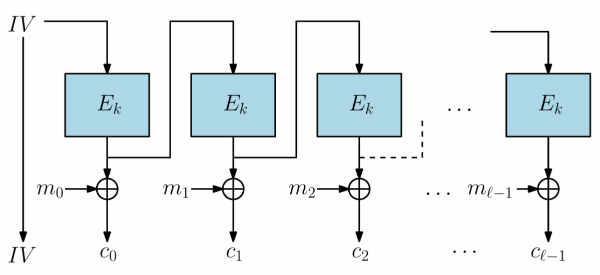
Algorithm 11 `CBCEncryption(FILE * filePointer, Word * KeyWords[])`

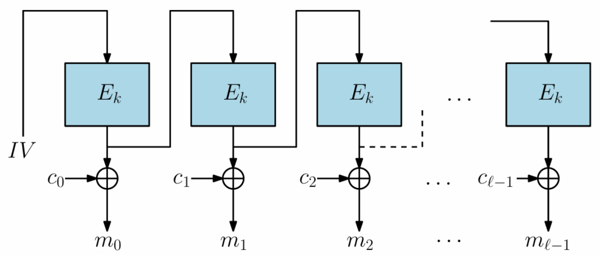
```
procedure CBCEncryption(filePointer, KeyWords[11])
    unsigned char buffer[16], state[4][4], cipher[4][4], temp[16], IV[16]
    int $i, j$, count, index
    filePointer ← fopen("input.txt", "r")                          ▷ Open plain text file
    OutfilePointer ← fopen("output_cbc.txt", "w")                  ▷ Open cipher text file
    print IV in OutfilePointer
    temp ← IV
    while not feof(filePointer) do
        count ← fread(buffer, 1, 16, filePointer)
        if count $> 0$ then
            if count $\neq 16$ then
                $k \leftarrow 16 -$ count
                for $i \leftarrow$ count to 15 do
                    buffer[$i$] ← $k$                                ▷ Padding with PKCS#7
                end for
            end if
            for $i \leftarrow 0$ to 15 do
                buffer[$i$] ← buffer[$i$] $\oplus$ temp[$i$]
            end for
            for $i \leftarrow 0$ to 3 do
                for $j \leftarrow 0$ to 3 do
                    state[$i$][$j$] ← buffer[$i \times 4 + j$]      ▷ Creating state matrix from buffer
                end for
            end for
            cipher ← Encryption(state, cipher, KeyWords)
            index ← 0
            for $i \leftarrow 0$ to 3 do
                for $j \leftarrow 0$ to 3 do
                    fprintf(OutfilePointer, "%c", cipher[$i$][$j$])
                    temp[index] ← cipher[$i$][$j$]
                    index ← index + 1
                end for
            end for
        end if
    end while
end procedure
```

This above pseudocode is the CBC mode encryption which we did in our implementation. The decryption is also the same instead we need to perform the operations from bottom to top, which we left for the reader.

0.5.3 Output FeedBack mode

The output feedback (OFB) mode makes a block cipher into a synchronous stream cipher by generating keystream blocks, which are then XORed with the plaintext blocks to get the ciphertext. The below diagram and the pseudocode says how this is done.





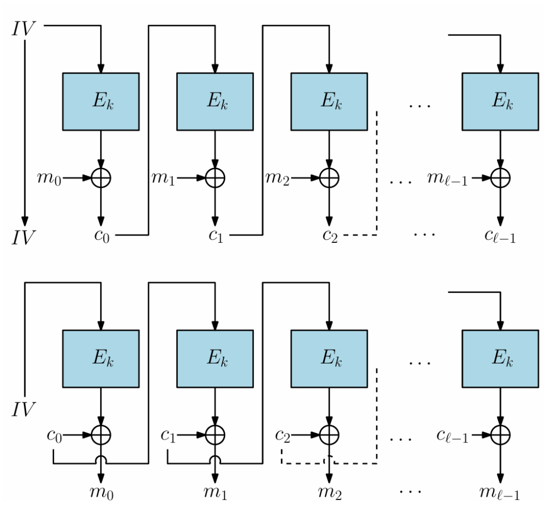
Algorithm 12 `OFBEncryption(FILE * filePointer, Word * KeyWords[])`

```
procedure OFBEncryption(filePointer, KeyWords[11])
    unsigned char buffer[16], state[4][4], cipher[4][4], temp[16], IV[16]
    int $i, j$, count, index
    filePointer ← fopen("input.txt", "r")                ▷ Open plain text file
    OutfilePointer ← fopen("output_ofb.txt", "w")        ▷ Open cipher text file
    print IV in OutfilePointer
    temp ← IV
    while (count = fread(buffer, 1, 16, filePointer)) > 0 do
        Padding with PKCS#7
        for $i \leftarrow 0$ to 3 do
            for $j \leftarrow 0$ to 3 do
                state[$i$][$j$] ← temp[$i \times 4 + j$]     ▷ Creating state matrix from temp
            end for
        end for
        cipher ← Encryption(state, cipher, KeyWords)     ▷ Perform AES Encryption
        index ← 0
        for $i \leftarrow 0$ to 3 do
            for $j \leftarrow 0$ to 3 do
                temp[index] ← cipher[$i$][$j$]
                fprintf(OutfilePointer, "%c", temp[index] $\oplus$ buffer[index])
                index ← index + 1
            end for
        end for
    end while
end procedure
```

Decryption follows the same approach, notice that during decryption, we will also use the `Encryption()` function.

0.5.4 Cipher FeedBack mode

The cipher feedback (CFB) mode, in its simplest form uses the entire output of the block cipher. In this variation, it is very similar to CBC, turning a block cipher into a stream cipher. CFB decryption in this variation is almost identical to CBC encryption performed in reverse.

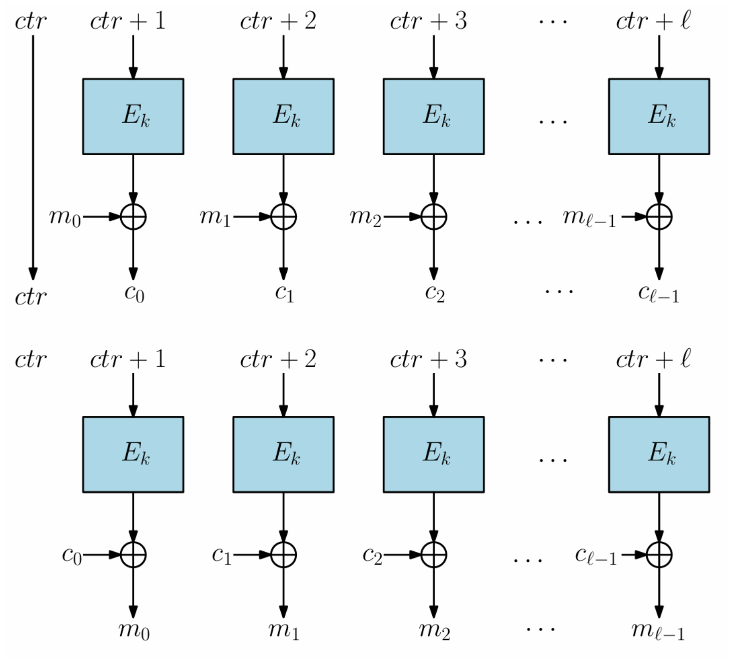


Algorithm 13 `CFBEncryption(FILE * filePointer, Word * KeyWords[])`

```
 1: procedure CFBENCRYPTION(filePointer, KeyWords[11])
 2:     unsigned char buffer[16], state[4][4], cipher[4][4], temp[16], IV[16]
 3:     int $i, j$, count, index
 4:     filePointer ← fopen(”input.txt”, ”r”)                 ▷ Open plain text file
 5:     OutfilePointer ← fopen(”output_cfb.txt”, ”w”)        ▷ Open cipher text file
 6:     print IV in OutfilePointer
 7:     temp ← IV
 8:     while (count = fread(buffer, 1, 16, filePointer)) > 0 do
 9:         Padding with PKCS#7
10:         for $i$ ← 0 to 3 do
11:             for $j$ ← 0 to 3 do
12:                 state[$i$][$j$] ← temp[$i \times 4 + j$]      ▷ Creating state matrix from temp
13:             end for
14:         end for
15:         cipher ← Encryption(state, cipher, KeyWords)      ▷ Perform AES Encryption
16:         index ← 0
17:         for $i$ ← 0 to 3 do
18:             for $j$ ← 0 to 3 do
19:                 temp[index] ← buffer[index] ⊕ cipher[$i$][$j$]
20:                 fprintf(OutfilePointer, ”%c”, temp[index])
21:                 index ← index + 1
22:             end for
23:         end for
24:     end while
25: end procedure
```

0.5.5 Counter mode

Like OFB, counter mode turns a block cipher into a stream cipher. It generates the next keystream block by encrypting successive values of a **counter**. The counter can be any function which produces a sequence which is guaranteed not to repeat for a long time, although an actual increment-by-one counter is the simplest and most popular.



In our implementation, we have taken **counter** to be $0x00$, but it can be taken arbitrarily. The decryption is same as encryption, and here also notice that, during decryption, `Encryption()` function is used instead of `Decryption()` function.

Algorithm 14 `CTREncryption(FILE * filePointer, Word * KeyWords[])`

```
 1: procedure CTRENCRYPTION(filePointer, KeyWords[11])
 2:     unsigned char temp[16], buffer[16], ctr[16] = {$0x00$}, state[4][4], cipher[4][4]
 3:     int $i, j$, count, $n = 1$, index
 4:     filePointer ← fopen("input.txt", "r")                    ▷ Open plain text file
 5:     OutfilePointer ← fopen("output_ctr.txt", "w")            ▷ Open cipher text file
 6:     while not eof(filePointer) do
 7:         count ← fread(buffer, 1, 16, filePointer)
 8:         Padding with PKCS#7
 9:         ctr[15] ← ctr[15] + $n$
10:         n ← n + 1
11:         for $i$ ← 0 to 3 do
12:             for $j$ ← 0 to 3 do
13:                 state[$i$][$j$] ← ctr[$i \times 4 + j$]      ▷ Creating state matrix from ctr
14:             end for
15:         end for
16:         cipher ← Encryption(state, cipher, KeyWords)
17:         index ← 0
18:         for $i$ ← 0 to 3 do
19:             for $j$ ← 0 to 3 do
20:                 temp[index] ← cipher[$i$][$j$] $\oplus$ buffer[index]
21:                 index ← index + 1
22:             end for
23:         end for
24:         for $i$ ← 0 to 15 do
25:             fprintf(OutfilePointer, "%$c$", temp[$i$])
26:         end for
27:     end while
28: end procedure
```

0.6 Our Implementation

As said earlier, we have implemented the whole AES-128 encryption and decryption, along with those five modes of operations. We have used minimal no of c libraries in our implementation, which makes it interesting.
There are two c files namely, **AES_128_source.c** and **AES_128.h**, where the first file contains all the source codes like `Encryption(), Decryption(), KeyExpansionFunction()` etc and the second file contains all the definitions of the functions defined in source file.
There are other files which are the codes of modes of operations like, **ECB_Encrypt.c, ECB_Decrypt.c, CBC_Encrypt.c, CBC_Decrypt.c, CFB_Encrypt.c, CFB_Decrypt.c, OFB_Encrypt.c, OFB_Decrypt.c, CTR_Encrypt.c, CTR_Decrypt.c**.

0.6.1 One Observation

We did a small experiment on our encryption and decryption scheme like we have taken a text **Hello World!!** and executed encryption and decryption in all the modes to compare the time taken by them. The below table gives the result which we implemented in a windows x64-based processor.

Modes	**Encryption(in sec.)**	**Decryption(in sec.)**
ECB	0.000897	0.001046
CBC	0.001511	0.001201
CFB	0.001080	0.000818
OFB	0.000917	0.001022
CTR	0.001259	0.000897

Table 2: Time comparison table

0.7 References

NIST documenttion on AES nist.fips.197
Original AES proposal The Rijndael Block cipher
Class notes