Symmetric Key Cryptography
CBC mode with AES-32 Encryption

EE720, Assignment - 2

Rathod Harekrissna Upendra, 17D070001
Sakshee Pimpale, 17D070021

November 15, 2020

1 Introduction

Advanced Encryption Standard (AES) is a symmetric key cryptography algorithm used as an encryption function in block ciphers. There are 4 modes in which a block cipher can be used: ECB, CBC, OFB, CTR; We will use the CBC i.e. Cipher Block Chaining Mode for this assignment. AES is a standardized encryption algorithm and hence it's security is not scrutinized till date. It was developed after several successful attacks on DES (Data Encryption Standard) arising due to increase in computation power. The general key length used in AES is 128, 192 or 256 bits for an input block length of 128 bits for all. These are named as AES-128, AES-192 and AES-256 respectively.

2 Input Blocks and CBC chaining

A block in a block cipher requires fixed input size e.g. 128 in AES. The length of plaintext that we want to encrypt need not necessarily a multiple of block size. To make it a multiple of block size, various padding schemes are used e.g For a block length of 8 , add 0x80 followed by zeros to make it a multiple of block size. This enables us to read the output plaintext after decryption unabiguously. Now it is split into N parts each of size blocksize. The CBC chaining mode proceeds as follows:

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

$$B_i = D_k(C_i) \oplus C_{i-1} \tag{2}$$

where E_k and D_k are encryption and decryption functions respectively. C_0 is a random initialization vector(IV) which is also sent along with the ciphertext. It helps deal with known-plaintext attacks as the output is different with different C_0 even with the same key. C_i for i = 1 to N is the ciphertext output of each block and B_i is the input to be encrypted for i^{th} block.

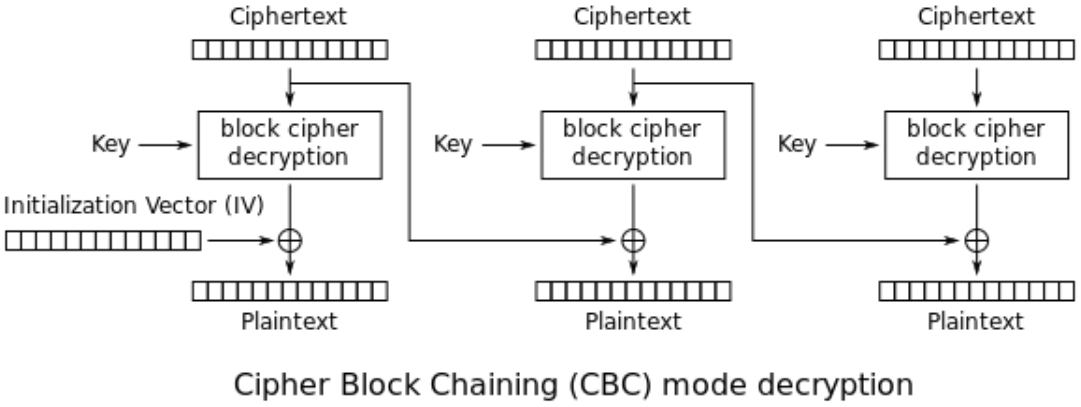

Figure 1: Source: `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation`

3 Encryption in AES-128

This section describes how ciphertext is generated for each block in AES-128.

The following figure gives an overview of the overall AES structure. AES-128 has $N_r = 10$. Thus there are 10 rounds in total. The first 9 rounds are identical with SubBytes, ShiftRows, MixColumns and AddRoundKey steps. The last column has only SubBytes, ShiftRows and AddRoundKey steps. One AddRoundKey transformation is done to the input before sending to the 1^{st} round. The Key Expansion algorithm produces 11 keys from the original key, each of which is used in each AddRoundKey step.

The decryption in AES involves these steps in exact reverse order with SubBytes replaced by InvSubBytes, ShiftRows by InvShiftRows, MixColumns by InvMixColumns and AddRoundKey is same.

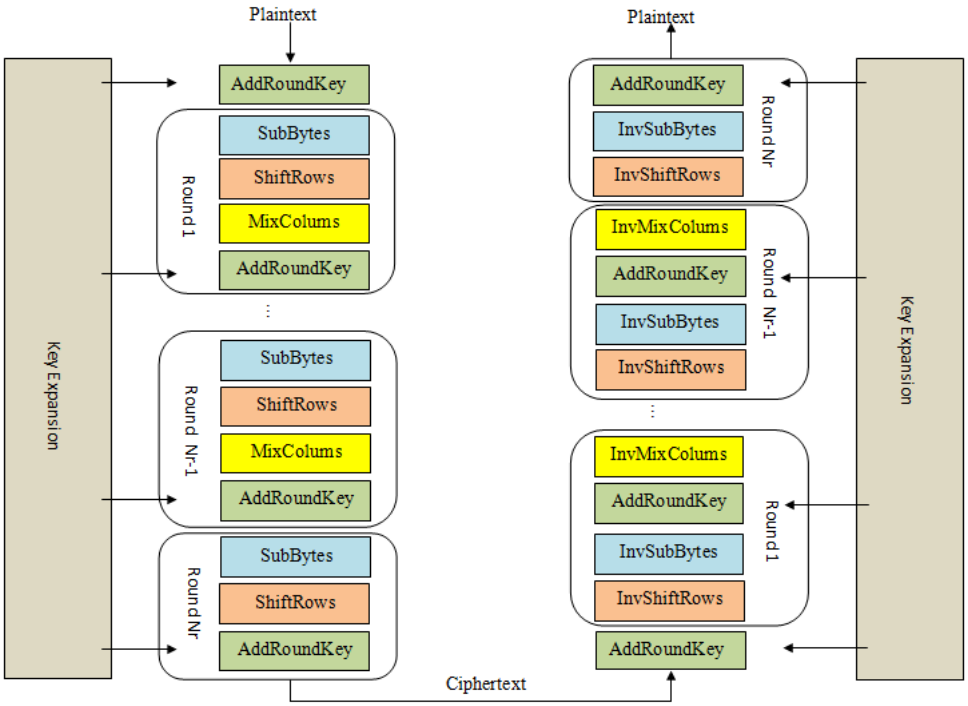

Figure 2: AES Scheme ;Source: `https://www.commonlounge.com/discussion/e32fdd267aaa4240a4464723bc74d0a5`

3.1 Pre-Processing of Plaintext

AES requires input to be in the form of a matrix with hexadecimal entries. For a 128 bit input, we have 16 bytes, represent each byte as a hexadecimal number and form a 4*4 matrix by arrangement of the 16 entries such that we cover first row then move to second and so on.

3.2 Key Expansion Algorithm

Considering AES-128, we have a 128 bit key. It consists of 16 bytes. Each byte converted to hexadecimal representation and arranged in a matrix columnwise forms our key matrix. This key matrix is put into an expansion algorithm to create 11 key matrices for 11 AddRoundKey Steps. These are called round keys. The diagram shown explains the algorithm. Note that word is a group of 32 bits. A key matrix consists of 4 words. The algorithm has to output 11 key matrices and thus 44 words.

w_0, w_1, w_2 and w_3 are the initial words formed from each column of the input key. The outputs are w_4 to w_{47}. Each w_i is computed using w_{i-1} and w_{i-4}. Note that w_4 to w_{47}, such 44 w_is only are ultimately used in encryption and decryption.

$$w_i = \begin{cases} w_{i-1} \oplus g(w_{i-4}), & \text{when } i = 0 \mod 4 \\ w_{i-1} \oplus w_{i-4}, & \text{otherwise} \end{cases}$$

The function g performs-

1. Left shift by one byte in a round fashion.
2. Bytewise substitution with S-box.
3. Bitwise XOR with a different pre-defined constant for every round. The constants are named as $rcon_j$. The first 8 bits of $rcon_j$ are set as given in the table by RC_js. The remaining 24 bits are set to 0. For AES-128, 10 RC_js are required.

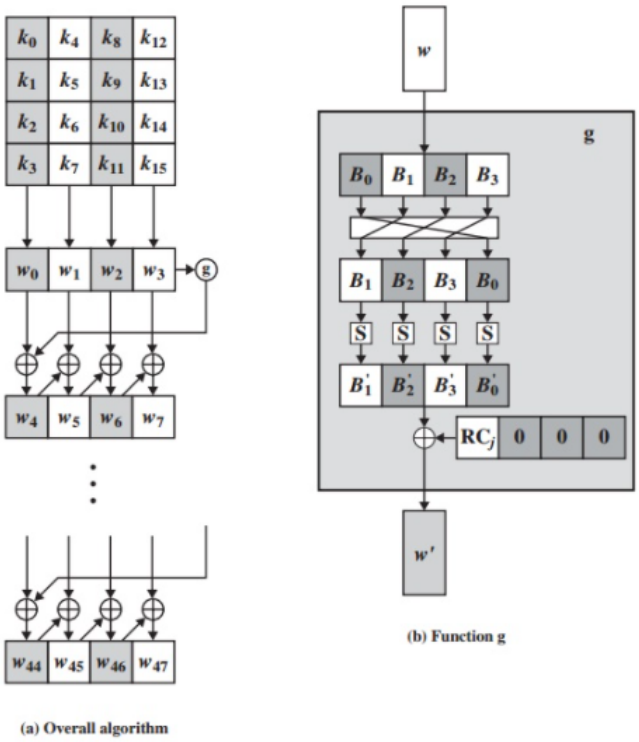

Figure 3: Key Expansion Algorithm; Source: `http://www.brainkart.com/article/AES-Key-Expansion_8410/`

Values of rc_i in hexadecimal

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

Figure 4: RC_i values; Source: Wikipedia

3.3 SubBytes or Substitution Via S-box

The s-box is a one-one mapping in the 8 bit space. The s-box is used as a look table for substitution and has been very carefully designed to resist attacks through cryptanalysis. This step introduces "confusion" in our encryption. The most significant nibble is used to identify row and the least significant one for identifying column Thus, an 8 bit value is substituted by another 8 bit value. The following table has some examples of S-box mappings of some 8-bit inputs.

Input Byte	Output Byte
0x6b	0x7f
0x65	0x4d
0x85	0x97

AES S-box

	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

The column is determined by the least significant nibble, and the row by the most significant nibble. For example, the value $9a_{16}$ is converted into $b8_{16}$.

3.4 ShiftRows

In ShiftRows step, given 4*4 matrix, shift first row by 0 bytes, second row by 1 byte to left, third row by 2 bytes to the left, and 4th row by 3 bytes to the left. Note that shift is in circular fashion and shift of 1 byte corresponds to shift of 8 bits.

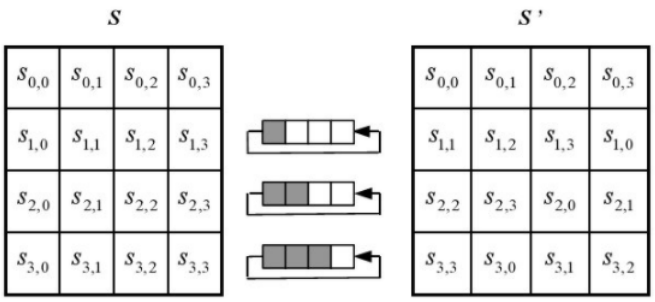

Figure 5: ShiftRows Step; Source: `https://www.researchgate.net/publication/272912836_A_High_Speed_and_Low_Power_Image_Encryption_with_128-Bit_AES_Algorithm`

3.5 MixColumns

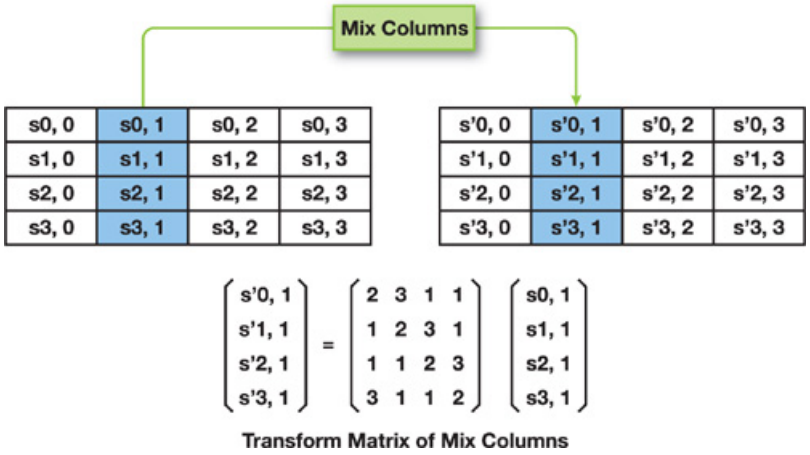

Figure 6: Source: `https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-36-aes-encryption-and-decryption-gpu`

MixColumns is a matrix multiplication step(pre-multiplication) with an invertible pre-specified matrix. The multiplication happens in GF(2^8). It is called MixColumns since given an input row, every entry of output row is nothing but some combination of each column entry of the input row. The pre-specified matrix used is found in the figure.

3.6 AddRoundKey

The key expansion creates different keys to be used for different applications of AddRoundKey. AES-128 has 11 applications of AddRoundKey step. The 11 generated keys after key expansion are used sequentially for these steps. The output of this step is a bitwise XOR of input with the supplied key.

3.7 Encryption Rounds

The number of rounds required depends on key length. Standard requirements for AES-128, AES-192 and AES-256 are as follows:

Key Length	Rounds
128	10
192	12
256	14

This table can be easily calculated by the standard equation:
N_round = Key_length (in bits)/32 + 6 This formula can be derived from the method by which we do key expansion which is explained above in detail.

4 Decryption in AES-128

The various steps involved are as descrubed below:

4.1 InvSubBytes or Inverse Substitution Via Inverse S-box

A different table which is an exact inverse of the S-box is used. e.g. xx is subsituted with yy with S-box and yy is substituted back to xx with inverse S-box for all xx.

Inverse S-box

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
10	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
20	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
30	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
40	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
50	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
60	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
70	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
80	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
90	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
a0	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
b0	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
c0	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
d0	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
e0	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
f0	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 7: Source: `Wikipedia`

4.2 InvShiftRows

This is also the exact inverse of ShiftRows. The first row is left as it is. Second row is shifted to the right by 1 byte, third row by 2 bytes and fourth by 3.

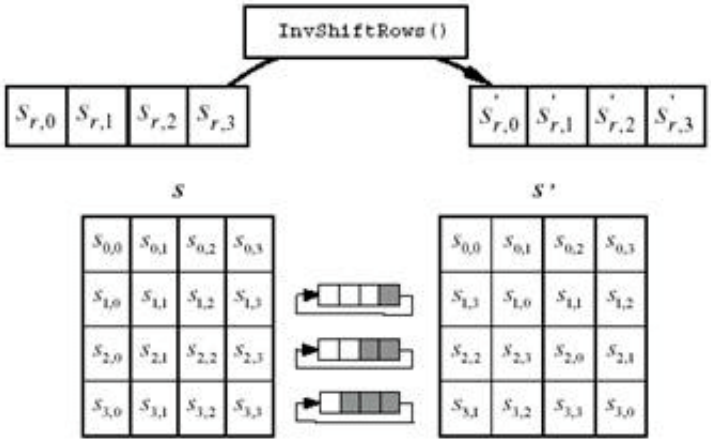

Figure 8: Source: `https://www.researchgate.net/publication/312277403_An_Improved_AES_Encryption_of_Audio_Wave_Files/figures?lo=1&utm_source=google&utm_medium=organic`

4.3 InvMixColumns

Since MixColumns step was nothing but matrix multiplication with a fixed invertible pre-defined matrix, the inverse of the matrix is used for InverseMixColumns. The inverse matrix is pre-multiplied with the input to this block to obtain InverseMixColumns output.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \quad \text{Inverse} \quad \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

$$C \qquad\qquad C^{-1}$$

4.4 InvAddRoundKey

Repeated XOR with the same bit is equivalent to identity operation. Thus, AddRoundKey is its own inverse. This happens since, $1 \oplus 1 = 0$, $0 \oplus 0 = 0$ and thus $a \oplus (k \oplus k) = a \oplus 0 = a$. The only difference is that, the keys generated after key expansion are supplied in reverse order. The keys remain same but 11^{th} key is supplied first, 10^{th} is supplied second and so on.

5 Attacks, Security and Future

The security of a block cipher requires a good amount of "confusion" and "diffusion". Confusion is obtained by the multiple rounds of substitution. The s-box is designed such that even a change in a single bit of plaintext causes a drastic change in the subsituted output. ShiftRows and MixColumns is nothing but permutation operations. A permutation operation does diffusion i.e. the changed bits caused by subsitution get dispersed throughout the ciphertext. ShiftRows diffuses horizontally while MixColumns diffuses vertically. Thus since multiple such rounds are used, even a change in a single bit causes a drastic change in the ciphertext output. This also makes it extremely difficult for a man in the middle to modify the ciphertext to get the desired plaintext since resultant plaintext changes drastically even with a single bit change in cipher text.
The large keylength of 128(2^{128} searches) and above causes brute force search impractical (astronomical time required).

The confusion and difffusion steps in round key generation also leads to a drastic change in ciphertext with even a single bit change in key. This makes it even harder to break the encryption
The AES cipher hasn't been broken since 18 years now. Some version with 8 round or versions with improper implementations have been broken. Only side-channel attacks can occur. Brute force is impossible with current computational power. In future, it can be easily made more secure by increasing key length and adding more rounds.

6 Boolean Model of reduced cipher

We first explain how different components of AES model get modified for a smaller model.

6.1 SubBytes and InvSubBytes

This is identical in our reduced model since our model also uses byte wise operations.

6.2 ShiftRows and InvShftRows

The first row is not shifted while the second row is shifted left by one byte.
In inverse, the second row is shifted by 1 bye i.e 8 bits towars right.

6.3 MixColumn and InvMixColumn

We pre-multiply with $\begin{matrix} 2 & 3 \\ 1 & 2 \end{matrix}$ for the mix column step. Its inverse in GF(2^8) is used in InvMixColumns step. We compute it with inbuilt python function for inverse in a galva field.

6.4 Key Expansion

Since we now use 7 rounds and thus 8 AddRoundKey steps, 8 keys are computed given the input key. Since the key is 32 bit, a 2 x 2 is matrix is obtained by arranging each byte as an entry. We redefine word to mean a group of 16 bits. Our new equations become,

$$w_i = \begin{cases} w_{i-1} \oplus g(w_{i-2}), & \text{when } i = 0 \mod 2 \\ w_{i-1} \oplus w_{i-2}, & \text{otherwise} \end{cases}$$

and i ranges from 0 to 17. w_0 and w_1 are obtained by combining entries of each column in the input key and are not used in AddRoundKey steps. Thus we obtain 18-2 =16 words which form 8 keys that are required.

6.5 AddRoundKey

It is exact same as for the larger model. The only difference is the key lengths and input lengths which are now 32 bits, i.e. a 2x2 matrix. The inverse of AddRoundKey remains same as AddRoundKey as properties of XOR remain same.

6.6 Overall Model

This section builds our boolean model. We can see that in reduced size model, the blocks will become equal to key length i.e. 4 bytes instead of 16 bytes. Thus we changed the CBC algorithm to fit blocks of 4 bytes. The various functions of CBC mode can be described in a boolean expression which summarizes the overall encryption process. In this section we derive the expression step by step.

Initially, looking at the CBC mode, we can easily write the expression: $C_i = E_K(B_i \oplus C_{i-1})$, where $C_i = E_K(B_i)$ and each B_i consists of blocks of 4 bytes or 32-bits, equal to the key length.

Now the encryption model E_K is defined by the AES process. For AES-32, we define the Boolean model of the Function E_K as follows: Let the element wise substitution function be S, which implements the look-up table for a given 2x2 matrix (or any size matrix if need be). Let R be the function implementing the shift-rows function on the given 2x2 matrix. Similarly, if M is the pre-multiplication matrix for Mix-columns operation, then for a round i of AES we can state the Boolean model as: $F_i(I) = M.dot(R(S(I))) \oplus K_i$ where I is the input 2x2 matrix to that round and F_i is the overall transformation for that round. Now if $K_i = W_{i1}W_{i2}$ (4 Bytes = 2 words of 2 bytes each), then from the key-expansion procedure, we can write the series of equations and obtain:

$K_i = [W_{(i-1)1} \oplus S(P(W_{(i-1)2})) \oplus (RC_j^{0-padded}), W_{(i)1} \oplus W_{(i-1)2}]$
With K_0 is the master key provided by the user. Also P is the permutation function of bytes (predefined). In this way, we can get keys for all the 7 rounds of the encryption. Hence, for overall Boolean expression of AES-32, we can write the formula:
$C_i = R(S(F_6(F_5(F_4(F_3(F_2(F_1(B_i \oplus K_0))))))) \oplus K_7$**, where** $F_i(I) = M.dot(R(S(I))) \oplus K_i$

Therefore including CBC, we can write the overall encryption formula (where C_0 is the IV):
$C_i = R(S(F_6(F_5(F_4(F_3(F_2(F_1(B_i \oplus C_{i-1} \oplus K_0))))))) \oplus K_7$**, where** $F_i(I) = M.dot(R(S(I))) \oplus K_i$
& $K_i = [W_{(i-1)1} \oplus S(P(W_{(i-1)2})) \oplus (RC_j^{0-padded}), W_{(i)1} \oplus W_{(i-1)2}]$, K_0 is the initial Key

6.7 Encryption Process of AES-32

Consider the input I1 I2 I3 I4 to an AES encryption block where I1, I2, I3 and I4 are 8 bit long.

1. We obtain input matrix as $\begin{pmatrix} I1 & I3 \\ I2 & I4 \end{pmatrix}$.
2. The first AddRoundKey operation on it gives output $\begin{pmatrix} I1 \oplus k1 & I3 \oplus k3 \\ I2 \oplus k2 & I4 \oplus k4 \end{pmatrix}$. Where k1 is W_{21}, k2 is W_{22}, k3 is W_{31}, k4 is W_{32}. W_{i1} represents the 8 most significant bits of W and W_{i2} represents the 8 least significant bits of W. Note that W is 16 bit in our reduced model.In our notation, W_0 and W_1 represent two halves of our original key. Hence we start from W_2 and go upto W_{17} for AddRoundKey operations.
3. This becomes the input of round 1. SubBytes step gives $\begin{pmatrix} S(I1 \oplus k1) & S(I3 \oplus k3) \\ S(I2 \oplus k2) & S(I4 \oplus k4) \end{pmatrix}$.
4. The Shift row operation gives $\begin{pmatrix} S(I1 \oplus k1) & S(I3 \oplus k3) \\ S(I4 \oplus k4) & S(I2 \oplus k2) \end{pmatrix}$.
5. Mix column operation results in output of

   $$\begin{pmatrix} 2 * S(I1 \oplus k1) + 3 * S(I4 \oplus k4) & 2 * (I3 \oplus k3) + 3 * S(I2 \oplus k2) \\ 1 * S(I1 \oplus k1) + 2 * S(I4 \oplus k4) & 1 * (I3 \oplus k3) + 2 * S(I2 \oplus k2) \end{pmatrix}$$

6. AddRoundKey operation gives
   $\begin{pmatrix} 2 * S(I1 \oplus k1) + 3 * S(I4 \oplus k4) \oplus k5 & 2 * (I3 \oplus k3) + 3 * S(I2 \oplus k2) \oplus k7 \\ 1 * S(I1 \oplus k1) + 2 * S(I4 \oplus k4) \oplus k6 & 1 * (I3 \oplus k3) + 2 * S(I2 \oplus k2) \oplus k8 \end{pmatrix}$

This continues 6 more times and we finish 7 rounds. The last round involves the steps 1,2,3,4 and 6 only. There is no MixColumn Operation. This generates our ciphertext for this block of input.

Note that in the code we have implemented, we have taken the input matrix as row wise filled instead of column wise filled as described above

6.8 Decryption Process of AES-32

Consider the input C1 C2 C3 C4 to an AES decryption block where C1, C2, C3 and C4 are 8 bit long.

1. We obtain input matrix as $\begin{pmatrix} C1 & C3 \\ C2 & C4 \end{pmatrix}$.
2. The first AddRoundKey operation on it gives output $\begin{pmatrix} C1 \oplus k1 & C3 \oplus k3 \\ C2 \oplus k2 & C4 \oplus k4 \end{pmatrix}$. Where k1 is $W_{(16)1}$, k2 is $W_{(16)2}$, k3 is $W_{(17)1}$, k4 is $W_{(17)2}$. W_{i1} represents the 8 most significant bits of W and W_{i2} represents the 8 least significant bits of W. Note that W is 16 bit in our reduced model.In our notation, W_0 and W_1 represent two halves of our original key. Hence we start from W_2 and go upto W_{17} for AddRoundKey operations.
3. This becomes the input of round 1 of decryption model. Inverse shift row step gives $\begin{pmatrix} C1 \oplus k1 & C3 \oplus k3 \\ C4 \oplus k4 & C2 \oplus k2 \end{pmatrix}$
4. The InvSubBytes step gives $\begin{pmatrix} S_{in}((C1 \oplus k1) & S_{in}(C3 \oplus k3) \\ S_{in}(C4 \oplus k4) & S_{in}(C2 \oplus k2) \end{pmatrix}$ where S_{in} is the inverse substitution matrix.
5. AddRoundKey operation gives $\begin{pmatrix} S_{in}((C1 \oplus k1) \oplus W_{(14)1} & S_{in}(C3 \oplus k3) \oplus W_{(15)2} \\ S_{in}(C4 \oplus k4) \oplus W_{(14)2} & S_{in}(C2 \oplus k2) \oplus W_{(15)2} \end{pmatrix}$
6. InvMixColumn operation gives

$$\begin{pmatrix} m1 * S_{in}(C1 \oplus k1) \oplus m3 * S_{in}(C4 \oplus k4) \oplus W_{(14)2} & m1 * S_{in}(C3 \oplus k3) \oplus W_{(15)2} + m3 * S_{in}(C2 \oplus k2) \oplus W_{(15)2} \\ m2 * S_{in}((C1 \oplus k1) + m4 * S_{in}(C4 \oplus k4) \oplus W_{(14)2} & m2 * S_{in}(C3 \oplus k3) \oplus W_{(15)2} + m4 * S_{in}(C2 \oplus k2) \oplus W_{(15)2} \end{pmatrix}$$

where $\begin{pmatrix} 2 & FD \\ FF & 2 \end{pmatrix}$ i.e. $\begin{pmatrix} m1 & m3 \\ m2 & m4 \end{pmatrix}$ is the InvMixColumn matrix.

This is repeated for 6 more identical rounds and the 8th and last round excludes the InvMixColumns step. Thus we obtain the plain text back. Sometimes padding has to be removed which is unambiguous by using good padding techniques.

The overall encryption and decryption process for CBC remains exactly same as original version just the input blocks become 4 bytes instead of 16 bytes

7 References used in this report

- https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf
- https://csrc.nist.gov/csrc/media/projects/cryptographic-standards-and-guidelines/documents/aes-development/rijndael-ammended.pdf#page=1
- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
- https://en.wikipedia.org/wiki/Advanced_Encryption_Standard_process
- https://en.wikipedia.org/wiki/AES_key_schedule(forkeyschedule)
- https://www.solarwindsmsp.com/blog/aes-256-encryption-algorithm#:~:text=AES%20256%20is%20virtu 20impenetrable,encryption%20system%20is%20entirely%20secure.
- https://www.cryptosys.net/pki/manpki/pki_paddingschemes.html#:~:text=To%20perform%20encryption% 20with%20a,16%20bytes%20(128%20bits). For padding schemes

This is the CBC, AES-32 function implemented as a class. Sbox is the substitution matrix, InvSbox is the Inv Substituion matrix. Rest algo is as described in the report. Inspiration of code taken from : https://github.com/bozhu/AES-Python (https://github.com/bozhu/AES-Python) and https://github.com/boppreh/aes/blob/master/tests.py (https://github.com/boppreh/aes/blob/master/tests.py)

In [8]:
```
import numpy as np
Sbox = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0x
FE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x
9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x
71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0x
EB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x
29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x
4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x
50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x
10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x
64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0x
DE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x
91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x
65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x
4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x
86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0x
CE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0x
B0, 0x54, 0xBB, 0x16,
)

InvSbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x
81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43, 0x44, 0x
C4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x
42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x
6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x
5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46, 0x57, 0x
A7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58, 0x05, 0x
B8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x
01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0x
F0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8, 0x
```

```
1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62, 0x0E, 0x
AA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x
78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x
27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x
93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB, 0x3C, 0x
83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14, 0x63, 0x
55, 0x21, 0x0C, 0x7D,
)

Rcon = (
    0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
    0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
    0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
    0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
)


def text2matrix(text):
    matrix = []
    for i in range(4):
        byte = (text >> (8 * (3 - i))) & 0xFF
        if i % 2 == 0:
            matrix.append([byte])
        else:
            matrix[int(i/2)].append(byte)
    return matrix

def matrix2text(matrix):
    text = 0
    for i in range(2):
        for j in range(2):
            text |= (matrix[i][j] << (24 - 8 * (2 * i + j)))
    return text


class AES_32:
    def __init__(self, master_key):
        self.change_key(master_key)

    def change_key(self, master_key):
        self.round_keys = text2matrix(master_key)

        for i in range(2, 2 * 8):
            self.round_keys.append([])
            if i % 2 == 0:
                byte = self.round_keys[i - 2][0]        \
                     ^ Sbox[self.round_keys[i - 1][1]]  \
                     ^ Rcon[int(i /2)]
                self.round_keys[i].append(byte)

                for j in range(1, 2):
```

```
                    byte = self.round_keys[i - 2][j]    \
                         ^ Sbox[self.round_keys[i - 1][(j + 1) % 2]]
                    self.round_keys[i].append(byte)
            else:
                for j in range(2):
                    byte = self.round_keys[i - 2][j]    \
                         ^ self.round_keys[i - 1][j]
                    self.round_keys[i].append(byte)

    def encrypt(self, inp_msg):
        self.plain_state = text2matrix(inp_msg)

        self.__add_round_key(self.plain_state, self.round_keys[:2])

        for i in range(1, 7):
            self.__round_encrypt(self.plain_state, self.round_keys[2 * i : 2 *
(i + 1)])

        self.__sub_bytes(self.plain_state)
        self.__shift_rows(self.plain_state)
        self.__add_round_key(self.plain_state, self.round_keys[14:])

        return matrix2text(self.plain_state)

    def decrypt(self, ciphertext):
        self.cipher_state = text2matrix(ciphertext)

        self.__add_round_key(self.cipher_state, self.round_keys[14:])

        for i in range(6, 0, -1):
            self.__round_decrypt(self.cipher_state, self.round_keys[2 * i : 2
* (i + 1)])

        self.__inv_shift_rows(self.cipher_state)
        self.__inv_sub_bytes(self.cipher_state)
        self.__add_round_key(self.cipher_state, self.round_keys[:2])

        return matrix2text(self.cipher_state)

    def __add_round_key(self, s, k):
        for i in range(2):
            for j in range(2):
                s[i][j] ^= k[i][j]


    def __round_encrypt(self, state_matrix, key_matrix):
        self.__sub_bytes(state_matrix)
        self.__shift_rows(state_matrix)
        self.__mix_columns(state_matrix)
        self.__add_round_key(state_matrix, key_matrix)


    def __round_decrypt(self, state_matrix, key_matrix):
        self.__inv_shift_rows(state_matrix)
        self.__inv_sub_bytes(state_matrix)
        self.__add_round_key(state_matrix, key_matrix)
        self.__inv_mix_columns(state_matrix)
```

```
    def __sub_bytes(self, s):
        for i in range(2):
            for j in range(2):
                s[i][j] = Sbox[s[i][j]]


    def __inv_sub_bytes(self, s):
        for i in range(2):
            for j in range(2):
                s[i][j] = InvSbox[s[i][j]]


    def __shift_rows(self, s):
        s[1][0], s[1][1] = s[1][1], s[1][0]


    def __inv_shift_rows(self, s):
        s[1][0], s[1][1] = s[1][1], s[1][0]



    def __mix_columns(self, s):
        s = np.matrix(s)
        M = np.matrix(([2,3],[1,2]))
        return M.dot(s)


    def __inv_mix_columns(self, s):
        s = np.matrix(s)
        M = np.matrix(([2,253],[255,2]))
        return M.dot(s)

        self.__mix_columns(s)

class AES_TEST_32():
    def setUp(self, master_key):
        self.AES_32 = AES_32(master_key)

    def test_encryption(self, inp_msg):
        encrypted = self.AES_32.encrypt(inp_msg)
        return encrypted

    def test_decryption(self, ciphertext):
        decrypted = self.AES_32.decrypt(ciphertext)
        return decrypted

def pad(plaintext):
    padding_len = 4 - (len(plaintext) % 4)
    padding = bytes([padding_len] * padding_len)
    return plaintext + padding

def split_blocks(message, block_size=4, require_padding=True):
    return [message[i:i+4] for i in range(0, len(message), block_size)]

def unpad(plaintext):
    padding_len = plaintext[-1]
```

```
    message, padding = plaintext[:-padding_len], plaintext[-padding_len:]
    return message

def xor_bytes(a, b):
    return bytes(i^j for i, j in zip(a, b))

def CBC_enc(plaintxt, key, iv):
  aes_32 = AES_TEST_32()
  aes_32.setUp(master_key = key)
  blocks = []
  previous = iv
  i = 0
  for plaintext_block in split_blocks(plaintxt):
    plaintext_block = pad(plaintext_block)
    test = int.from_bytes((xor_bytes(plaintext_block, previous)), byteorder =
'big')
    block = aes_32.test_encryption(inp_msg = test).to_bytes(4, 'big')
    blocks.append(block)
    previous = block
    i = i+1
  return b''.join(blocks)

def CBC_dec(ciphertxt, key, iv):
  aes_32 = AES_TEST_32()
  aes_32.setUp(master_key = key)
  blocks = []
  previous = iv
  i = 0
  for ciphertext_block in split_blocks(ciphertxt):
    block = aes_32.test_decryption(ciphertext = int.from_bytes(ciphertext_bloc
k, 'big')).to_bytes(4, 'big')
    block = xor_bytes(block, previous)
    blocks.append(block)
    previous = ciphertext_block
    i = i+1
  str_out = b''.join(blocks)
  str_out = unpad(str_out)
  return str_out

def Check_CBC_enc_dec(plaintext, key, iv):
  ciphertxt = CBC_enc(plaintext, key, iv)
  print("Cipher text: ", ciphertxt)
  dec_txt = CBC_dec(ciphertxt, key, iv)
  print("Decrypted text: ", dec_txt)
  if(dec_txt == plaintext):
    print("Algorithm Working Fine")
```

In [9]:

```
Check_CBC_enc_dec(b'This is my message to be encrypted, you can type anything
 instead of this', key = 0x2b7e1516, iv = b'love')
```

```
Cipher text:  b'\xd3\x83\xce.4R\xf7\xf4\xda9\x0b\xc8\x92r\xd2\xe8Z@\xe1\x8d\x
11Z\xecSq\xc3\t\xb0t\x06\xd3\x82\x9d\xad\xa1\xff\xeb\x90z\xe5\x01\xf9\xb0-\xf
c<h\x04K\xf0x\xfa\xcbkn5\x9b\xf5t\xd9\xc2K\x0653\xcdm\xbe\x9a\xea\xee\xe9\x85
\xe5k\x99'
Decrypted text:  b'This is my message to be encrypted, you can type anything
instead of this'
Algorithm Working Fine
```

In [ ]: