

ASSIGNMENT

Course Code	19CSC315A
Course Name	Information Security and Protection
Programme	B.Tech.
Department	Computer Science and Engineering
Faculty	Faculty of Engineering and Technology

Name of the Student	Subhendu Maji
Reg. No	18ETCS002121
Semester/Year	6 TH /2018
Course Leader/s	Vaishali R. Kulkarni

Declaration Sheet			
Student Name	Subhendu Maji		
Reg. No	18ETCS002121		
Programme	B.Tech.	Semester/Year	6 th / 2018
Course Code	19CSC315A		
Course Title	Information Security and Protection		
Course Date		to	
Course Leader	Vaishali R. Kulkarni		
<p>Declaration</p> <p>The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.</p>			
Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			
Signature of the Course Leader and date		Signature of the Reviewer and date	

Declaration Sheet	ii
Contents	iii
Question No. 1	4
1.1 Algorithm developed and Keys used	5
1.2 Python Program	13
1.3 Validation using Test Cases	16
1.4 Conclusions	19

Question No. 1

Solution to Question No. 1:

Symmetric Key Cryptography also known as Symmetric Encryption is when a secret key is leveraged for both encryption and decryption functions. This method is the opposite of Asymmetric Encryption where one key is used to encrypt and another is used to decrypt. During this process, data is converted to a format that cannot be read or inspected by anyone who does not have the secret key that was used to encrypt it.

The success of this approach depends on the strength of the random number generator that is used to create the secret key. Symmetric Key Cryptography is widely used in today's Internet and primarily consists of two types of algorithms, Block and Stream. Some common encryption algorithms include the Advanced Encryption Standard (AES) and the Data Encryption Standard (DES). This form of encryption is traditionally much faster than Asymmetric however it requires both the sender and the recipient of the data to have the secret key.

Before going into AES algorithm, we need to understand some terms.

Encoding is not Encrypting

Encryption can be defined as the process of transforming information or data into a code in order to conceal its original content. Therefore, decryption is the process of transforming this encrypted code back to its original state. For it to be considered encryption and not encoding, a secret key is necessary.

Let's make an analogy with a lock. If we just close the lock, without locking in with its key, it gives the appearance of being locked but in reality, anybody could open it. In order to secure the lock, we must lock it with our key, so it can't be opened without the key we locked it with. Much like just closing the lock, encoded information gives the impression of being encrypted but in reality, anybody could transform it back to its original state. This is not so much the case with encryption.

1.1 Algorithm developed and Keys used

Modes of Operation

AES is a block cipher: it will receive 128 bits of text which will be transformed to obtain a different 128 bits of encrypted data. But 128 bits or 16 characters most probably won't be enough to fit all the data we wish to encrypt, so AES does Divide and conquer.

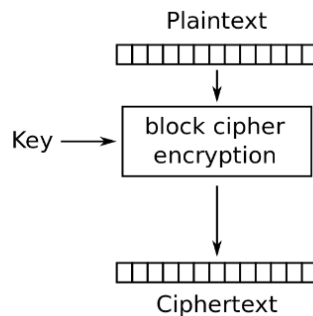


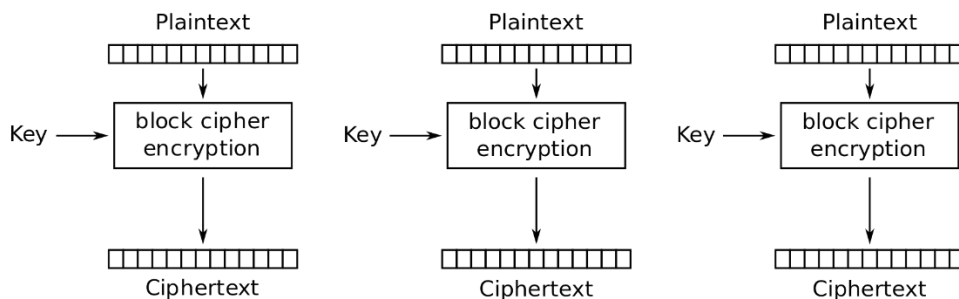
Figure 1 Block cipher encryption

AES splits the data we wish to encrypt in 128-bit sized blocks and proceeds to encrypt them. If the input data isn't a multiple of 128, we must extend the length of the message for it to be so. This process is called padding, and in its simplest form it will just add 0s to the end of the message till it is a multiple of 128 bits.

There are several more modes of encryption, ECB and CBC are the most didactic ones.

- **ECB**

The simplest one, ECB, encrypts each block on its own and displays the resultant encrypted blocks one after another.

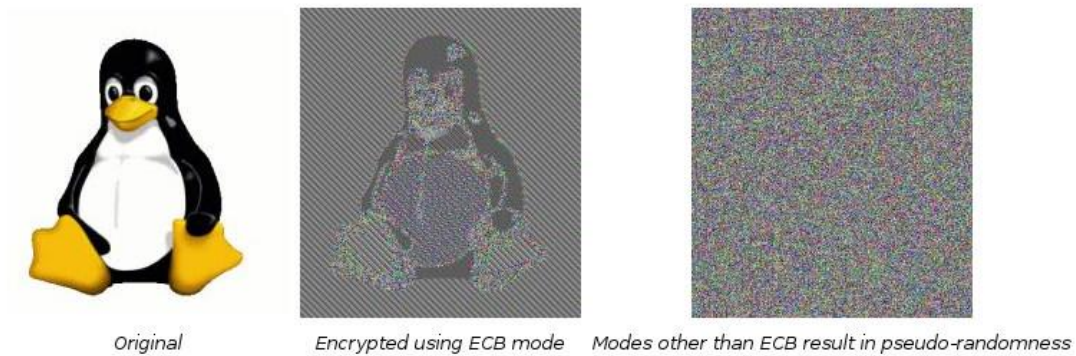


Electronic Codebook (ECB) mode encryption

Figure 2 ECB mode encryption

But there is weakness associated with ECB, as it fails to properly conceal repeated information.

For e.g.



Despite some loss of information, the resulting encrypted image it's still recognizable and far off from appearing random. For this, we should avoid using ECB mode and instead use more complex modes, such as CBC.

- **CBC**

CBC inputs the AES cipher the result of the XOR operation between the current plain text block and the previous encrypted block. For the first plain text block, as there is not a previous encrypted block, an initialization vector is required to perform the XOR operation. This IV will have the same size as our blocks, 128 bits or 16 characters.

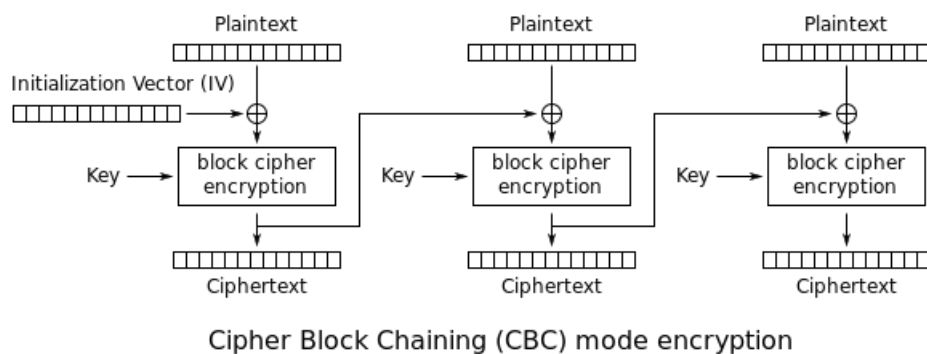


Figure 3 CBC mode encryption

It is considered a good practice to generate a new IV each time you encrypt in order to cause more randomness in the output. But in order to decrypt in CBC mode we must also know the IV used for encryption, so it should be placed before the encrypted blocks and be part of the encryption output. With this approach when we want to decrypt, we know that the first block of our encrypted message it's the IV.

AES Encryption

We know how AES can encrypt large inputs by dividing them into blocks of 128 bits. We know that AES can have keys of three sizes, 128, 192, 256 bits and that the longer the key, the stronger the encryption.

To understand this, we must know that when AES encrypts, it applies the same algorithm encryption algorithm for a certain number of rounds, each round using a different sub key generated from our initial key in a process called key expansion. The longer the size of our key, the more sub keys AES is able to generate and so more rounds can be performed of the encryption algorithm, resulting in a stronger encryption.

This algorithm is composed of 4 operations: **Byte Sub**, **Shift Rows**, **Mix Columns** and **Add Round Key**. For visual representation, our input text block of 128 bits will be represented on a 4 by 4 matrix, each position representing 8 bits from our input block.

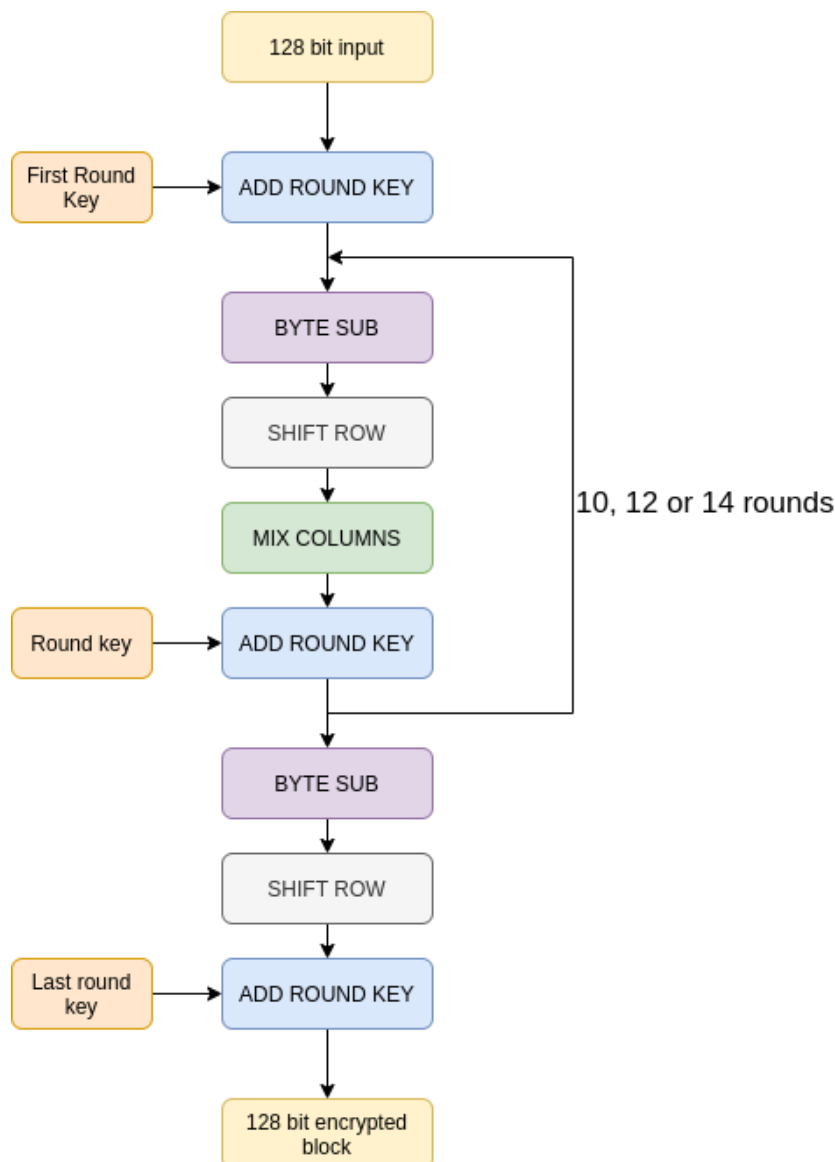


Figure 4 AES encryption flowchart

a. Byte Sub

Byte Sub will substitute each cell of 8 bits with another 8 bits. Those 8 bits will be chosen from a predefined look up table, constant throughout the rounds. The first four bits of each cell will decide the row and the last four the column, and that is the position of the look up table where we can find the 8 bits that will be replacing that cell.

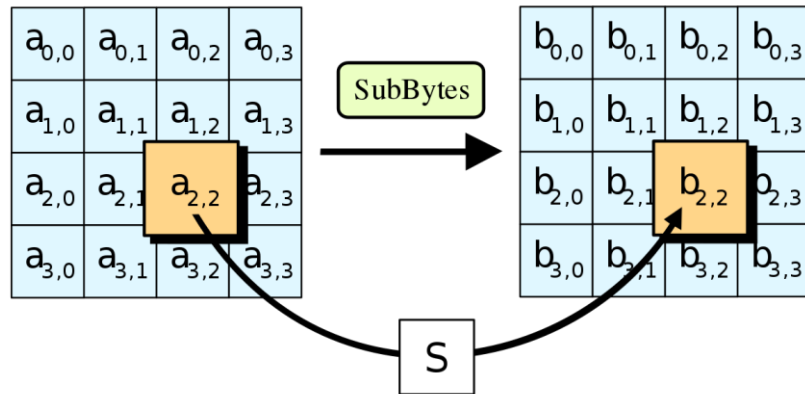


Figure 5 Byte sub visualization

Byte Sub look up table is shown in Figure 6.

	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

Figure 6 Byte sub look up table

The values of the look up table are coded in hexadecimal; each digit corresponds to 4 bits. So, if in each cell there are two digits, each cell is worth 8 bits of space, just as our input matrix.

For example, Let consider we have a cell with the value 10100101 or A5. It will be replaced for the cell in row A or 10 and column 10 in the look up table, 06 or 00000110.

b. Shift Rows

In shift rows we shift each row of our matrix to the left. The first row won't be shifted at all, the second one will shift one position, the third two positions and the fourth and last will shift 3 positions.

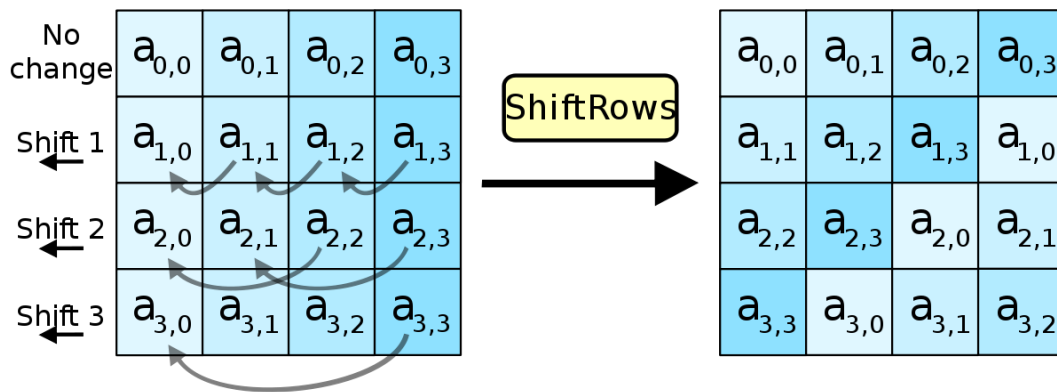


Figure 7 Shift rows visualization

c. Mix Columns

In Mix Columns we will perform a matrix multiplication between our current matrix and a predefined given matrix, constant throughout the rounds. But it's a slight trickier matrix multiplication, as the sum operation is substituted by XOR and multiplication for AND.

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \times \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Figure 8 mix columns visualization

d. Add Round Key

Up to here, we haven't encrypted the information at all. All these transformations could be undone by anyone, as we haven't yet introduced our key into the equation. Each round key is 128 bits long, so we can represent it the same way as our input block, in a matrix. Now we shall perform XOR operation between each block of our input block which its respective block from the key matrix.

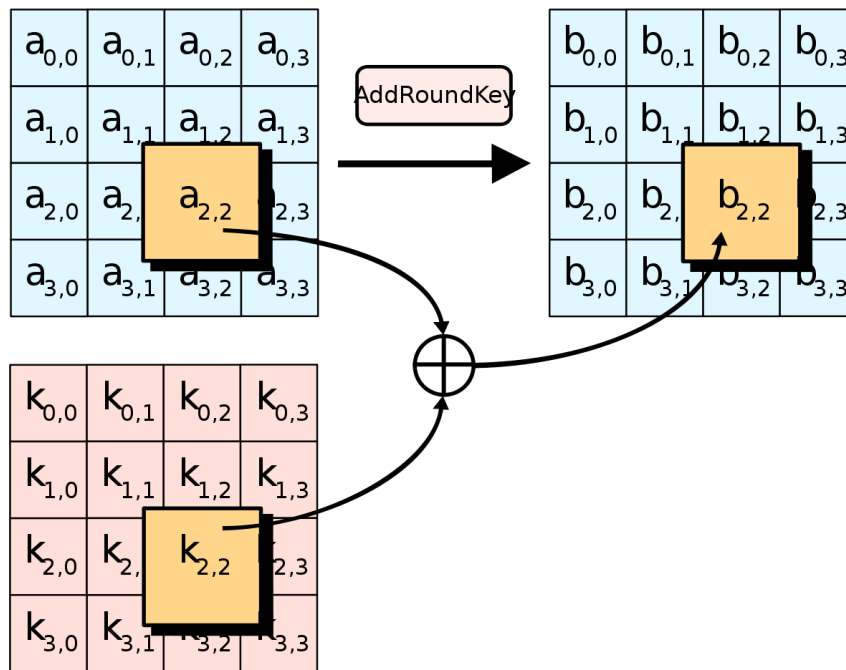


Figure 9 add round key visualization

And that's how AES encryption work.

AES Decryption

AES is symmetric cipher which means, much as our lock from before, the same key used to encrypt data is necessary to decrypt that data. In order to undo the encryption, as one may initially think, we must reverse the encryption process. We are going to perform the same operations, just in the inverse order. So, contrary to encryption, the decryption algorithm is shown in Figure 10.

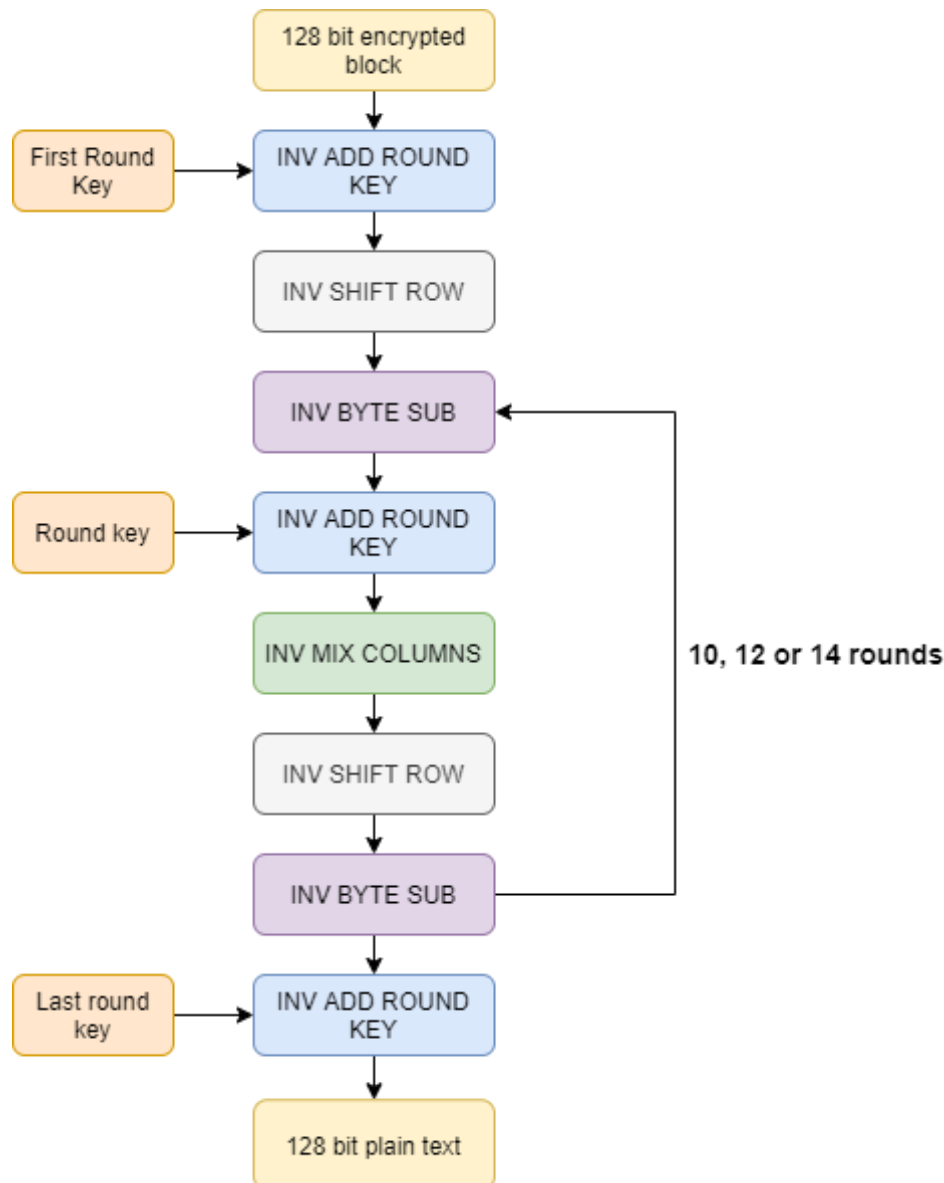


Figure 10 AES decryption flowchart

We can notice that all the operations start with INV, which symbolizes the inverse operations. For example, Inverse Mix Column is the inverse operation of Mix Columns. It means that in order to reverse the Mix Columns operation, we must apply Inverse Mix Column.

a. Inverse Add Round Key

Inverse Add Round Key is its own inverse, if we applied twice Add Round Key, you are going to be left with what we started. Because of this property, in order to undo Add Round Key we must apply it once more. For this, Inverse Add Round Key and Add Round Key are the same operation.

b. Inverse Shift Row

We know shift rows only shifted each row a certain number of positions to the left. So, in order to reverse, we must move each row the same number of positions but to the right.

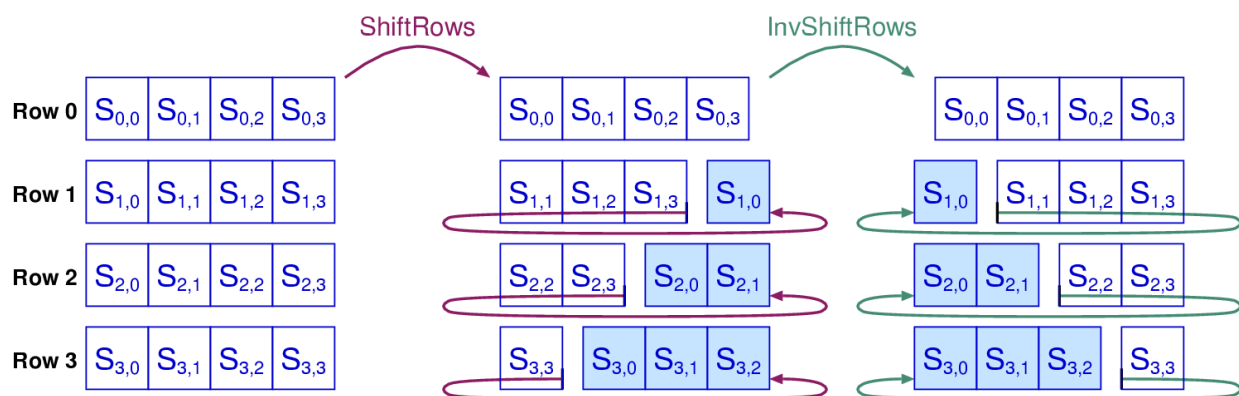


Figure 11 Inverse Shift Row

c. Inverse Sub Bytes

Inverse Sub Bytes is in essence the same operation as Sub Bytes: it will take each block in our matrix and will swap it with another from a predefined matrix. The only thing that changes between Inverse Sub Bytes and Sub Bytes is this table. The table for Inverse Sub Bytes is shown in Figure 12.

	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
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 12 Inverse sub byte lookup table

d. Inverse Mix Columns

If we multiply a matrix C with its inverse, C^{-1} , we get the identity as a result.

$$C \cdot C^{-1} = I_n$$

So, let's call our matrix of data X .

In Mix Columns, we set $X = X \cdot C$. In order to undo this multiplication, we must multiply by the inverse of C .

$$X \cdot C \cdot C^{-1} = X \cdot I_n = X$$

We showed the C matrix in Mix Columns, so here we shall show the inverse of C :

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

$C \qquad C^{-1}$

Figure 13 Inverse Mix Columns

And that's how decryption is done.

AES keys

AES keys are a string of bits (1s and 0s) and can take three different sizes: 128, 192 or 256 bits. In order to set keys in a more human way, they can be established with Unicode characters (letters, numbers and some symbols) each one taking 8 bits. The longer the key, the stronger the encryption process.

1.2 Python Program

Python3 Source Code

```
#!/usr/bin/env python3

Sbox = (
    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, 0x9A, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

InvSbox = (
    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, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCD, 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,
)

def xtime(a): return (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)

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(16):
        byte = (text >> (8 * (15 - i))) & 0xFF
        if i % 4 == 0:
            matrix.append([byte])
        else:
            matrix[i // 4].append(byte)
    return matrix

def matrix2text(matrix):
    text = 0
    for i in range(4):
        for j in range(4):
            text |= (matrix[i][j] << (120 - 8 * (4 * i + j)))
    return text

class AES:
    def __init__(self, master_key):
        if master_key < 0:
            raise Exception('Key can not be < 0')
        self.change_key(master_key)

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

        for i in range(4, 4 * 11):
            self.round_keys.append([])
            if i % 4 == 0:
                byte = self.round_keys[i - 4][0] \
                    ^ Sbox[self.round_keys[i - 1][1]] \
                    ^ Rcon[i // 4]
                self.round_keys[i].append(byte)

                for j in range(1, 4):
                    byte = self.round_keys[i - 4][j] \
                        ^ Sbox[self.round_keys[i - 1][(j + 1) % 4]]
                    self.round_keys[i].append(byte)
            else:
                for j in range(4):
                    byte = self.round_keys[i - 4][j] \
                        ^ self.round_keys[i - 1][j]
                    self.round_keys[i].append(byte)

```

```

def encrypt(self, plaintext):
    if plaintext < 0:
        raise Exception('Plaintext can not be < 0')

    self.plain_state = text2matrix(plaintext)
    self.__add_round_key(self.plain_state, self.round_keys[:4])

    for i in range(1, 10):
        self.__round_encrypt(
            self.plain_state, self.round_keys[4 * i: 4 * (i + 1)])

    self.__sub_bytes(self.plain_state)
    self.__shift_rows(self.plain_state)
    self.__add_round_key(self.plain_state, self.round_keys[40:])

    return matrix2text(self.plain_state)

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

    self.__add_round_key(self.cipher_state, self.round_keys[40:])
    self.__inv_shift_rows(self.cipher_state)
    self.__inv_sub_bytes(self.cipher_state)

    for i in range(9, 0, -1):
        self.__round_decrypt(
            self.cipher_state, self.round_keys[4 * i: 4 * (i + 1)])

    self.__add_round_key(self.cipher_state, self.round_keys[:4])

    return matrix2text(self.cipher_state)

def __add_round_key(self, s, k):
    for i in range(4):
        for j in range(4):
            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.__add_round_key(state_matrix, key_matrix)
    self.__inv_mix_columns(state_matrix)
    self.__inv_shift_rows(state_matrix)
    self.__inv_sub_bytes(state_matrix)

def __sub_bytes(self, s):
    for i in range(4):
        for j in range(4):
            s[i][j] = Sbox[s[i][j]]

def __inv_sub_bytes(self, s):
    for i in range(4):
        for j in range(4):
            s[i][j] = InvSbox[s[i][j]]

def __shift_rows(self, s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]

def __inv_shift_rows(self, s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]

def __mix_single_column(self, a):
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
    u = a[0]
    a[0] ^= t ^ xtime(a[0] ^ a[1])
    a[1] ^= t ^ xtime(a[1] ^ a[2])
    a[2] ^= t ^ xtime(a[2] ^ a[3])
    a[3] ^= t ^ xtime(a[3] ^ u)

def __mix_columns(self, s):
    for i in range(4):
        self.__mix_single_column(s[i])

def __inv_mix_columns(self, s):
    for i in range(4):
        u = xtime(xtime(s[i][0] ^ s[i][2]))
        v = xtime(xtime(s[i][1] ^ s[i][3]))
        s[i][0] ^= u
        s[i][1] ^= v
        s[i][2] ^= u
        s[i][3] ^= v

    self.__mix_columns(s)

```

1.3 Validation using Test Cases

We are using a python3 package unittest for unit-testing our AES program.

Test Case 1: Testing with 128-bit message

```
1 import unittest
2 from aes import AES
3
4
5 class AES_TEST(unittest.TestCase):
6     def setUp(self):
7         master_key = 0x2b7e151628aed2a6abf7158809cf4f3c
8         self.AES = AES(master_key)
9
10    def test_encryption(self):
11        plain_text = 0x3243f6a8885a308d313198a2e0370734
12        encrypted = self.AES.encrypt(plain_text)
13        self.assertEqual(encrypted, 0x3925841d02dc09fdbc118597196a0b32)
14
15    def test_decryption(self):
16        cipher_text = 0x3925841D02DC09FBDC118597196A0B32
17        decrypted = self.AES.decrypt(cipher_text)
18        self.assertEqual(decrypted, 0x3243f6a8885a308d313198a2e0370734)
19
20
21 if __name__ == "__main__":
22     unittest.main()
23
```

Figure 14 Test 1

Master key	0x2b7e151628aed2a6abf7158809cf4f3c
Plain Text	0x3243f6a8885a308d313198a2e0370734
Encrypted Text	0x3925841D02DC09FBDC118597196A0B32
Decrypted Text	0x3243f6a8885a308d313198a2e0370734

We can notice that key, plain_text, encrypted_text and decrypted_text is 128-bit data (hexa-decimal of length 32).

Output

```
/mnt/d/RUAS-sem-06/Information Security/Assignment 2 main*
> python3 test1.py
..
-----
Ran 2 tests in 0.001s

OK

/mnt/d/RUAS-sem-06/Information Security/Assignment 2 main*
> 
```

Figure 15 output of test 1

Test Case 2: Fuzz Testing

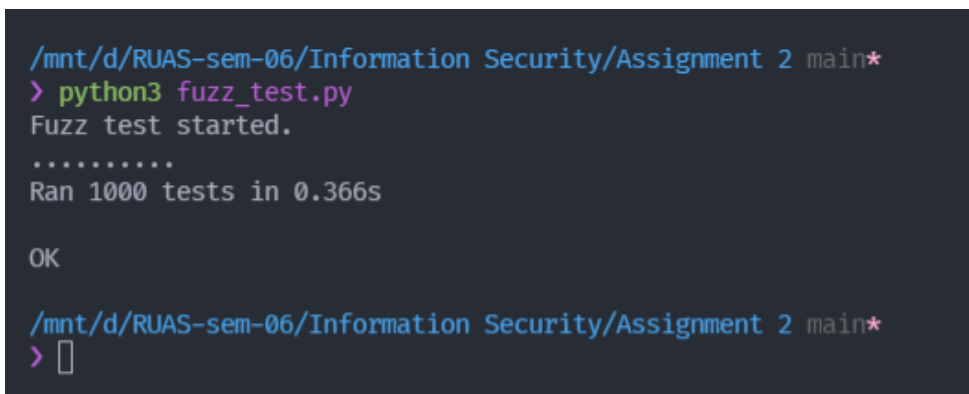
```
1 import time
2 from fuzzingbook.Fuzzer import RandomFuzzer
3 from aes import AES
4
5
6 def test_system(key, data):
7     """
8     Encrypt and decrypt @key and @data
9     """
10    a = AES(key)
11
12    plaintext = data
13    encrypted = a.encrypt(plaintext)
14    decrypted = a.decrypt(encrypted)
15
16    # check if data is encrypted
17    assert encrypted != decrypted, 'Data is not encrypted!'
18
19    return decrypted
20
21
22 def fuzz_testing(num_of_tests=10):
23     """
24     Do tests
25     """
26     ok = True
27     start = time.time()
28     errors = []
29
30     # generate integers. Max possible length is 38
31     r = RandomFuzzer(min_length=1, char_start=48, char_range=9, max_length=38)
32     dot = num_of_tests // 10 # decoration stuff
33
34     for i in range(0, num_of_tests):
35         if i % dot == 0:
36             print('.', end='')
37
38         # generate test case
39         key = int(r.fuzz())
40         data = int(r.fuzz())
41
42         # analyze output
43         try:
44             result = test_system(key, data)
45             if data != result:
46                 errors.append('TEST ' + str(i) + ' FAILED\nKey = "' +
47                               str(key) + '"\nData = "' + str(data) + '"')
48                 errors.append('Result = "' + str(result) + '"')
49                 errors.append(
50                     'Reason: Error in encryption - decryption process\n')
51                 ok = False
52             except AssertionError:
53                 errors.append('TEST ' + str(i) + ' FAILED\nKey = "' +
54                               str(key) + '"\nData = "' + str(data) + '"')
55                 errors.append('Reason: Data was not encrypted\n')
56                 ok = False
57
58     # print testing results
59     elapsed = time.time()
60     elapsed = elapsed - start
61     print()
62     for i in errors:
63         print(i)
64     print('Ran ' + str(num_of_tests) + ' tests in ' + str(round(elapsed, 3)) + 's')
65     print()
66     if ok:
67         print('OK')
68     else:
69         print('FAILED')
70
71
72 if __name__ == "__main__":
73     print('Fuzz test started.')
74     fuzz_testing(1000)
```

Figure 16 Fuzz-testing AES Source Code

In Fuzz testing (Figure 16), we are creating a random key and data. Then passing the data to test_system function which call the AES program, here, first sending data to encrypt and then sending the returned encrypted message to decrypt and finally returning the decrypted message.

Then we are checking if the decrypted message is same as the data, we sent or not. If not same then, throwing a Failed case with its key and data.

We are also monitoring the time taken for the whole operation to run 1000 times and output is shown in Figure 17.

A terminal window with a dark background and light blue text. The prompt is '/mnt/d/RUAS-sem-06/Information Security/Assignment 2 main*'. The user enters 'python3 fuzz_test.py'. The output shows 'Fuzz test started.', followed by a series of dots, then 'Ran 1000 tests in 0.366s', and finally 'OK'. The prompt is shown again at the bottom.

```
/mnt/d/RUAS-sem-06/Information Security/Assignment 2 main*  
> python3 fuzz_test.py  
Fuzz test started.  
.....  
Ran 1000 tests in 0.366s  
  
OK  
  
/mnt/d/RUAS-sem-06/Information Security/Assignment 2 main*  
> 
```

Figure 17 Output of fuzz testing

As we can see in Figure 17 the implemented AES is successfully encrypting and decrypting 1000 random messages in 0.366 sec (approx.).

1.4 Conclusions

Till date is not any evidence to crack this algorithm. AES is unbroken. It means that, without knowing the key, there hasn't been found a faster way to decrypt AES-encrypted data than by brute forcing its key i.e., checking all possible keys until we find the one that was used for encryption. AES keys are a string of bits (1s and 0s) and can take three different sizes: 128, 192 or 256 bits. So, to brute force AES, we will possibly have to go through up to 2256 different keys before we find the right one.

Going back to our analogy, it would be like having a bag full of keys and checking one by one if it opened the lock. But finding the right key in AES is slightly more subtle. If you decrypt an encrypted message with a different key than used for its encryption, the result is going to be a random set of characters, making as much sense as the encrypted message i.e., none.

This algorithm enables faster than DES and 3DES algorithms to encrypt and decrypt data. Furthermore, it is used in many cryptography protocols such as Socket Security Layer (SSL) and Transport Security Layer protocol to provide much more communications security between client and server over the internet. Before AES algorithm released both of protocols to encrypt and decrypt data relied on DES algorithm but after appearing some vulnerable of this algorithm the Internet Engineering Task Force (IETF) decided to replace DES to AES algorithm.

AES can also be found in most modern applications and devices that need encryption functionality such as WhatsApp, Facebook Messenger and Intel and AMD processor and Cisco devices like router, switch, etc. Using internet and network are increasing rapidly. Everyday a lot of digital data have been exchanging among users. Some of data is sensitive that need to protect from intruders. Encryption algorithms play vital roles to protect original data from unauthorized access. Various kind of algorithms are existing to encrypt data. Advanced encryption standard (AES) algorithm is one of the efficient algorithms and it is widely supported and adopted on hardware and software.

1. https://www.researchgate.net/publication/317615794_Advanced_Encryption_Standard_AES_Algorithm_to_Encrypt_and_Decrypt_Data
2. https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
3. <https://medium.com/quick-code/understanding-the-advanced-encryption-standard-7d7884277e7>