Palestine Technical University-Kadoorie

Faculty of Engineering and Technology

Computer Systems Engineering Department

Network Security & Cryptography

AES

ADVANCED ENCRYPTION STANDARD
[S-Box, ShiftRows, MixColumns]

Prepared by:

Mohammad Abohasan

Supervisor:

Dr. Anas Melhem

26 August 2022

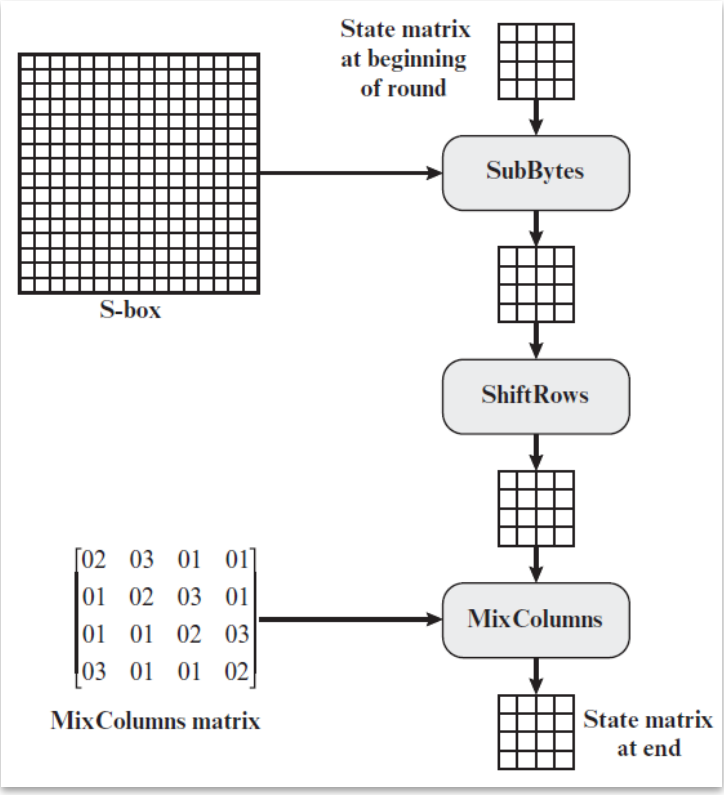
TABLE OF CONTENTS

I. Task definition

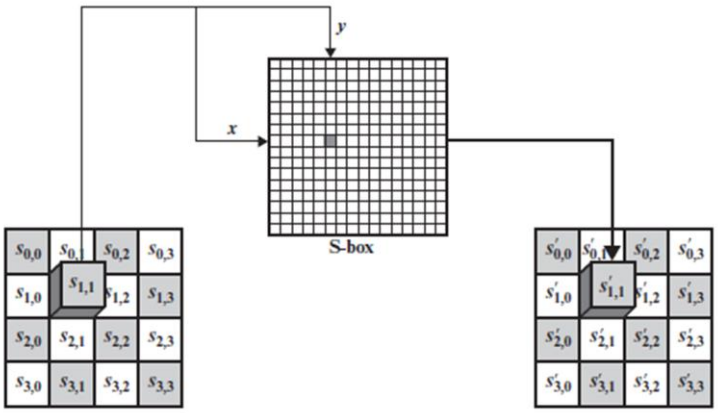
In this task, I implement and test Advanced Encryption Standard – 128bits modules (S-Box, ShiftRows, and MixColumns).
These modules are part of the AES.
I used Java to implement the algorithm because it is my favorite language.

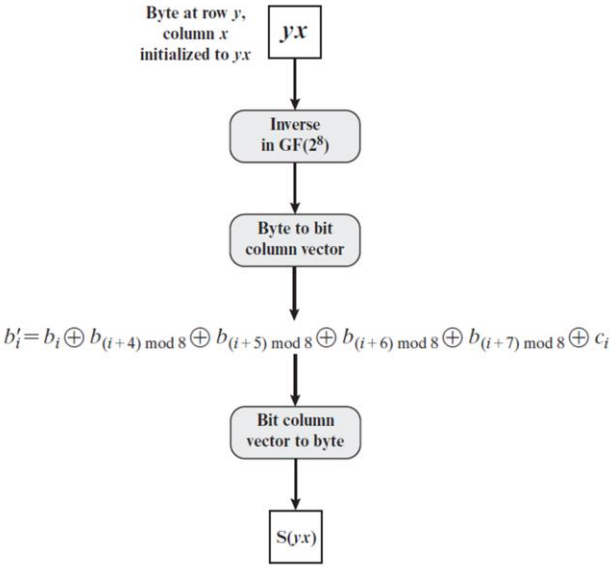
II. brief definition of an algorithm and the module to be implemented

Being divided into 3 modules, each module can work independently, and the output of the first module can also be linked to the second and the second to the third. The three modules have input in the form of four words, and each word contains four bytes, and each byte contains eight bits. So we use AES-128 in this task, but there are also AES-192 and AES-256. For the output, it is the same size and format as the input, and what happens inside each component will be detailed separately.



called Byte Substitution (SubBytes). The 16 input bytes are replaced by a fixed table lookup (Table 1.1). The result is a matrix of four rows and four columns. which is about finding a multiplcative inverse and then introducing it to an affine transformation.






```
private int[] multiplicativeInverse = new int[9];
private String[][] S_boxTable = new String[16][16];
```

`multiplicativeInverse`: Here Multiplicative Inverse is stored.

`S_boxTable`: Here, Table 1.1 is stored after its generation to reduce the complexity that is required in its calculation each time.

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

Table 1.1 AES S-Box

```
public S_Box() {
    GenerateS_boxTable();
}

public S_Box(String[] input) {
    this();
    for (int i = 0; i < input.length; i++) {
        char h1 = input[i].toUpperCase().charAt(0),
             h2 = input[i].toUpperCase().charAt(1);
        output[i] = S_boxTable[h1 >= '0' && h1 <= '9' ? h1 - '0' : h1 - 'A' + 10]
                              [h2 >= '0' && h2 <= '9' ? h2 - '0' : h2 - 'A' + 10];
    }
}
```

There are two constructors, both of which are called GenerateS_boxTable(), which generates the Table 1.1.
The first constructor to create the object I used to test the module, and the other constructor through which the input is passed in the form of a string array consisting of 16 elements, is 1 byte, i.e., 8 bits represented by two hexadecimal characters. To access its value in Table 1.1 I take each character separately and convert it from 0 to 15. The value is stored in the string array output, which also consists of 16 elements and contains the output of this module.

```
private void GenerateS_boxTable() {
    String L, R;
    for (int i = 0; i <= 15; i++) {
        for (int j = 0; j <= 15; j++) {
            L = "" + (char) (i >= 0 && i <= 9 ? i + '0' : (i - 10) + 'A') + "";
            R = "" + (char) (j >= 0 && j <= 9 ? j + '0' : (j - 10) + 'A') + "";
            S_boxTable[i][j] = Generator(L + R);
        }
    }
}
```

Generate Table 1.1 completely, and here I needed to convert numbers to 0–9 or A–F.

```
private String Generator(String input) {
    ExtendedEuclidAlgorithm(HexToBinary(input));
    return BinaryToHex(Transformation());
}


private int[] Transformation() {
    int[] B = new int[8]; // (b0, b1, b2, b3, b4, b5, b6, b7)
    for (int i = 0; i < B.length; i++) {
        B[i] = multiplicativeInverse[i];
    }
    int[] c = {1, 1, 0, 0, 0, 1, 1, 0}; // (c0c1c2c3c4c5c6c7) = (11000110) = 36
    int[] bPrime = new int[8];
    for (int i = 0; i < bPrime.length; i++) {
        bPrime[i] = B[i] ^ B[(i + 4) % 8] ^ B[(i + 5) % 8] ^
                    B[(i + 6) % 8] ^ B[(i + 7) % 8] ^ c[i];
    }
    return bPrime;
}
```

Converting the input from hexadecimal to binary (to treat it as a polynomial) using the HexToBinary() method, which I will explain later, and then giving the output to the

ExtendedEuclidAlgorithm() method to calculate the Multiplicative Inverse, and we give the output to Transformation() method that implements this equation for each bit

$$b`_i = b_i \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \quad \textbf{(1.1)}$$

and eventually convert the binary number to hexadecimal using the BinaryToHex() method.

```
public void MultiplicativeInverse() {
    System.out.println(Arrays.toString(multiplicativeInverse));
}
```

To print the multiplicativeInverse array.

```
private void ExtendedEuclidAlgorithm(int[] b) {
    int[] A1 = {1, 0, 0, 0, 0, 0, 0, 0, 0};
    int[] A2 = {0, 0, 0, 0, 0, 0, 0, 0, 0};
    int[] A3 = new int[9];

    int[] B1 = {0, 0, 0, 0, 0, 0, 0, 0, 0};
    int[] B2 = {1, 0, 0, 0, 0, 0, 0, 0, 0};
    int[] B3 = new int[9];

    int[] T1 = new int[9];
    int[] T2 = new int[9];

    System.arraycopy(mX, 0, A3, 0, mX.length);
    System.arraycopy(b, 0, B3, 0, b.length);
    while (Degree(B3) > 0) {
        Clear(Q);
        Division(A3, B3);
        System.arraycopy(B3, 0, A3, 0, B3.length);
        System.arraycopy(rem, 0, B3, 0, rem.length);

        Multiplication(Q, B1);
        int[] QB1 = new int[9];
        System.arraycopy(rem, 0, QB1, 0, rem.length);
        Multiplication(Q, B2);
        int[] QB2 = new int[9];
        System.arraycopy(rem, 0, QB2, 0, rem.length);

        for (int i = 0; i < QB1.length; i++) {
            T1[i] = A1[i] - QB1[i];
            T1[i] = Math.abs(T1[i] % 2);
        }
        for (int i = 0; i < QB2.length; i++) {
            T2[i] = A2[i] - QB2[i];
            T2[i] = Math.abs(T2[i] % 2);
        }
        System.arraycopy(B1, 0, A1, 0, B1.length);
        System.arraycopy(B2, 0, A2, 0, B2.length);
        System.arraycopy(T1, 0, B1, 0, T1.length);
        System.arraycopy(T2, 0, B2, 0, T2.length);
    }
    Clear(multiplicativeInverse);
    if (Degree(B3) == 0) {
        System.arraycopy(B2, 0, multiplicativeInverse, 0, B2.length);
    }
}
```

Applying the ExtendedEuclid algorithm, you can understand the algorithm from here: https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
and then look at my implementation. It is very clear and simple, but I will explain the working sequence of this method. First, the value of A1 = 1, i.e., it is equal to x 0 = 1, Now I'll explain how I represent polynomials in the array: index 0 means x^0, index 1 means x^1, and index i means x^i. If the value at index k is 0, there is no x^k because its coefficient is zero, but if it is 1, there is x^k. There are no other possibilities because we're working under GF (2^8) and also no k greater than 7 assuming that k is the power of x except for m(x),A irreducible polynomial is used to ensure that there is always a multiplicative inverse, and this link explains what GF(2^8) is: https://en.wikipedia.org/wiki/Finite_field

And the value of A2 = 0, A3 = m(x) at the beginning, B1 = 0, B2 = 1, and B3 = input, We will execute the algorithm until become the value of B3 = 1, meaning that the value of the GCD between m(x) and the input is 1, Therefore, the value of B2 is the multiplicative inverse. All the methods used here will be explained in detail later.

The S-Box contains attributes and methods that are present in the MixColumns, and in order to apply the principle of inheritance in the oop, I collected these attributes and methods and put them in the class coll, and then inherited both the S-Box and the MixColumns from the parent class. I will now explain what is inside it:

Class coll

```
//                              0  1  2  3  4  5  6  7  8
    protected final int[] mX = {1, 1, 0, 1, 1, 0, 0, 0, 1}; // irreducible polynomial
    protected int[] Q = new int[9];
    protected int[] rem = new int[9];
    protected String[] output = new String[16];
    private final String[] hexToBinary = {
        "0000", "0001", "0010", "0011",
        "0100", "0101", "0110", "0111",
        "1000", "1001", "1010", "1011",
        "1100", "1101", "1110", "1111"
    };
```

Here there is m(x) that I explained earlier and there is Q, which is the result of each division we do. rem is the remainder of the division we do, and the output is the array in which I will store the final value of the operation used, whether S-Box or MixColumns. hexToBinary In this fixed array, the index value of 0 is stored with 0000 for example, its value in binary and 1 in 0001 and so on. As for 10 to 15, they are meant from A to F, and this was taken into account and shown in the upcoming Methods.

```
protected String BinaryToHex(int[] a) {
    String binNum = "";
    for (int i = 0; i < 8; i++) {
        binNum += a[8 - i - 1];
    }
    String b1 = binNum.substring(0, 4), b2 = binNum.substring(4);
    String hexNum, h1 = "", h2 = "";
```

```
    for (int i = 0; i < hexToBinary.length; i++) {
        if (b1.equals(hexToBinary[i])) {
            h1 = (char) (i >= 0 && i <= 9 ? i + '0' : (i - 10) + 'A') + "";
        }
        if (b2.equals(hexToBinary[i])) {
            h2 = (char) (i >= 0 && i <= 9 ? i + '0' : (i - 10) + 'A') + "";
        }
    }
    hexNum = h1 + h2;
    hexNum = hexNum.toLowerCase();
    return hexNum;
}

protected int[] HexToBinary(String s) {
    String hexNum = s.toUpperCase();
    char h1 = hexNum.charAt(0), h2 = hexNum.charAt(1);
    String binNum = hexToBinary[h1 >= '0' && h1 <= '9' ? h1 - '0' : h1 - 'A' + 10]
                  + hexToBinary[h2 >= '0' && h2 <= '9' ? h2 - '0' : h2 - 'A' + 10];
    int[] arr = new int[8];
    for (int i = 0; i < arr.length; i++) {
        arr[i] = binNum.charAt(8 - i - 1) - '0';
    }
    return arr;
}
```

Here in BinaryToHex I'm storing the binary number represented in the array as string and then converting each 4 bits to hexadecimal separately.
In HexToBinary, I take each letter separately, convert it to a Binary, and store it as an array.

```
protected int Degree(int[] a) {
    int deg = -1;
    for (int i = a.length - 1; i >= 0; i--) {
        if (a[i] == 1) {
            deg = i;
            break;
        }
    }
    return deg;
}
```

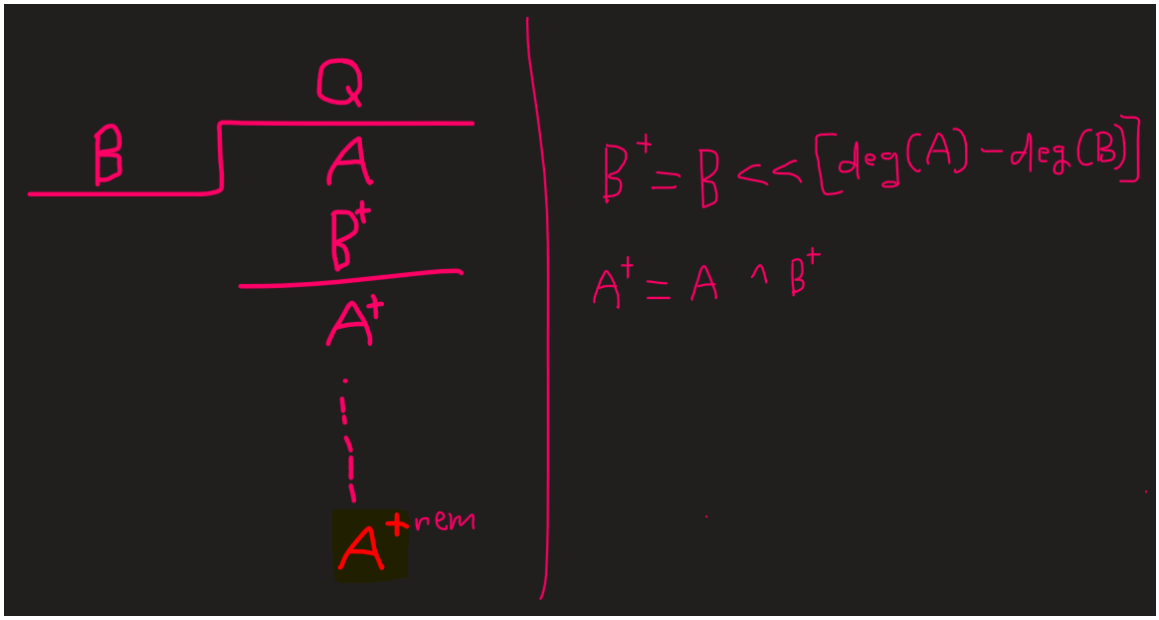
As for this method, I have used it a lot and it is very simple, to know the greatest degree of this polynomial.

```
protected void Division(int[] a, int[] b) {
    if (Degree(b) == -1) {
        System.arraycopy(b, 0, rem, 0, rem.length);
        return;
    }
    int diffDeg = Degree(a) - Degree(b);
    if (diffDeg < 0) {
        System.arraycopy(a, 0, rem, 0, rem.length);
        return;
    }
    Q[diffDeg] = 1;
    int[] tempA = new int[a.length];
    System.arraycopy(a, 0, tempA, 0, a.length);
    int[] tempB = new int[b.length];
    System.arraycopy(b, 0, tempB, 0, b.length);
    while (diffDeg-- > 0) {
```

```
        for (int i = tempB.length - 1; i > 0; i--) {
            tempB[i] = tempB[i - 1];
        }
        tempB[0] = 0;
    }
    for (int i = 0; i < tempA.length; i++) {
        tempA[i] ^= tempB[i];
    }
    Division(tempA, b);
}
```

Finally, we come to the division part. In the beginning, there was a special case, which is if B3 = 0, then this means that there is no multiplicative inverse, but here it was considered that the multiplicative inverse is 0. I will mention this at the end of the report at the part of the conclusion, of course the long division The polynomial is well known, and here is an explanation for it:
https://en.wikipedia.org/wiki/Polynomial_long_division

This method was used to complete the division process and store the remainder of the division and the result of the division:




```
protected void Multiplication(int[] a, int[] b) {
    int[] M = new int[9];
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < b.length; j++) {
            if (a[i] == 0 || b[j] == 0) {
                continue;
            }
            M[i + j] += a[i] * b[j];
        }
    }
    for (int i = 0; i < M.length; i++) {
        M[i] %= 2;
    }
    Division(M, mX);
}
```

And the multiplication procedure is straightforward: as long as the base is equal, we add the exponents, and finally we take mod 2 and mod m(x), because, as previously stated, we are working under GF (2^8).

```
protected void Clear(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = 0;
    }
}
```

A method is used to delete the contents of an array so that we can use it again.

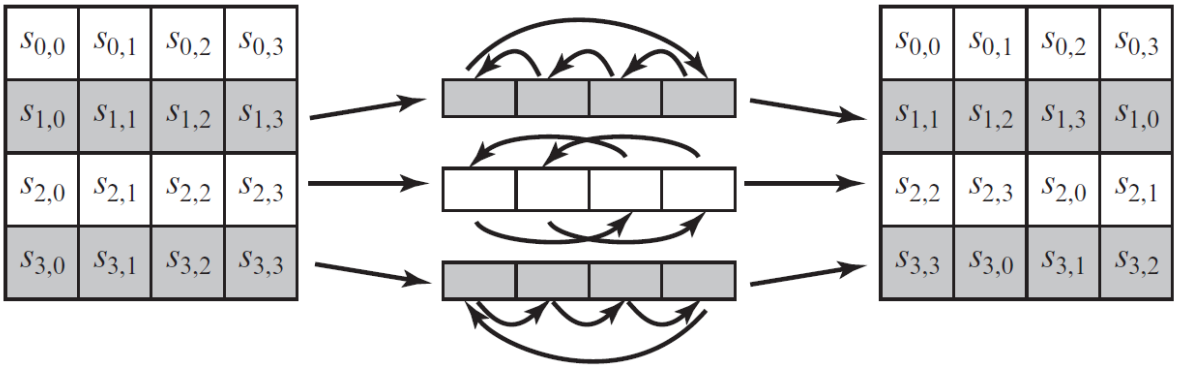
```
public String[] getOutput() {
    return output;
}
```

The method returns an output array.

CHAPTER 2 – ShiftRows

Each of the four rows of the matrix is shifted to the left. Any entries that 'fall off' are re-inserted on the right side of the row. shift is carried out as follows:

- The first row is not shifted.
- The second row is shifted one (byte) position to the left.
- The third row is shifted two positions to the left.
- The fourth row is shifted three positions to the left.
- The result is a new matrix consisting of the same 16 bytes but shifted with respect to each other.




```
private String[] output = new String[16];

public Shift_rows(String[] input) {
    String[] row1 = new String[4], row2 = new String[4], row3 = new String[4];
    System.arraycopy(input, 4 * 0, output, 0, 4);

    System.arraycopy(input, 4 * 1, row1, 0, 4);
    System.arraycopy(input, 4 * 2, row2, 0, 4);
    System.arraycopy(input, 4 * 3, row3, 0, 4);
    rotateToLeft(row1, 1);
    rotateToLeft(row2, 2);
    rotateToLeft(row3, 3);
    System.arraycopy(row1, 0, output, 4 * 1, 4);
    System.arraycopy(row2, 0, output, 4 * 2, 4);
    System.arraycopy(row3, 0, output, 4 * 3, 4);
}

private void rotateToLeft(String[] a, int n) {
    String temp;
    while (n-- > 0) {
        temp = a[0];
        for (int i = 0; i < a.length - 1; i++) {
            a[i] = a[i + 1];
        }
        a[a.length - 1] = temp;
    }
}
```

```
public String[] getOutput() {
    return output;
}
```

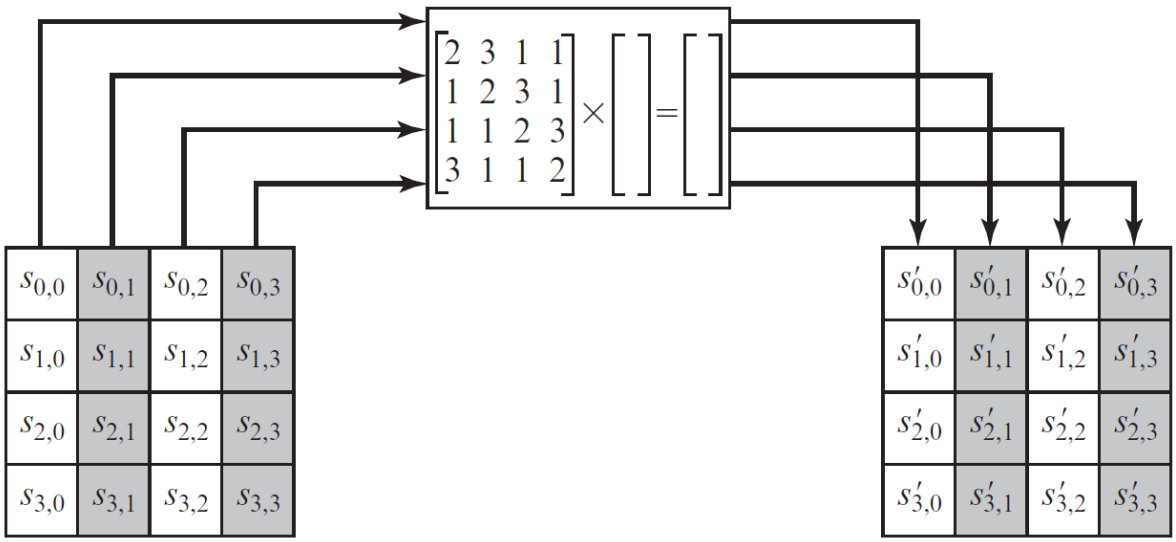
output, getOutput() is already explained in the S-Box.

rotateToLeft() by which we rotate the elements to the left the number of times required by each row.

Shift_rows() I stored each row of the input array alone and then made the required shift(rotate) for each one, and then copied the values of these hashed arrays onto the output array. Simply, this is all the ShiftRows and it's the easiest part of this task.

CHAPTER 3 – MixColumns

Each column of four bytes is now transformed using a special mathematical function. This function takes as input the four bytes of one column and outputs four completely new bytes that replace the original column. The result is another new matrix consisting of 16 new bytes.



As I said earlier, this part is shared with the S-Box in some of the attributes and methods, and I have assembled them in a class and inherited them from it. To know what this class contains, open page 7.
Now I will show and explain what the MixColumns contain independently.

```
private final String[] mixColTransformation = { "02", "03", "01", "01",
                                                "01", "02", "03", "01",
                                                "01", "01", "02", "03",
                                                "03", "01", "01", "02"
};

public Mix_columns(String[] input) {
    int[] out = new int[8];
    for (int i = 0; i < input.length; i++) {
        Clear(out);
        for (int j = (i / 4) * 4, k = i % 4, cnt = 0; cnt < 4; j++, k += 4, cnt++) {
            Multiplication(HexToBinary(mixColTransformation[j]), HexToBinary(input[k]));
            for (int r = 0; r < out.length; r++) {
                out[r] += rem[r];
            }
        }
        for (int q = 0; q < out.length; q++) {
            out[q] %= 2;
        }
        output[i] = BinaryToHex(out);
    }
}
```

mixColTransformation This is a static array into which we multiply the input; it is represented here by the multiplication of two matrices, 2D * 1D, and in the multiplication process implicitly mod m(x), and after the multiplication, we also do mod 2 because we are working under GF (2^8).

```java
import java.util.Scanner;

/**
 *
 * @author Mohammad_AboHasan
 */
public class AES {

    public static void main(String[] args) {

        // checker S-Box
        Scanner sc = new Scanner(System.in);
        String[] inputS_Box = new String[16];
        for (int i = 0; i < inputS_Box.length; i++) {
            inputS_Box[i] = sc.next();
        }
        S_Box sBox = new S_Box(inputS_Box);
        String[] outputS_Box = sBox.getOutput();
        System.out.println("================================");
        System.out.println("\tS-Box");
        System.out.println("================================");
        System.out.println("Input :");
        for (int i = 0; i < inputS_Box.length; i++) {
            System.out.print(inputS_Box[i]
                    + (i != inputS_Box.length - 1 ? ", " : "")
                    + ((i + 1) % 4 == 0 ? "\n" : ""));
        }
        System.out.println("============");
        System.out.println("Output :");
        for (int i = 0; i < outputS_Box.length; i++) {
            System.out.print(outputS_Box[i]
                    + (i != outputS_Box.length - 1 ? ", " : "")
                    + ((i + 1) % 4 == 0 ? "\n" : ""));
        }

        // checker Shift rows
        Shift_rows shiftRows = new Shift_rows(outputS_Box);
        String[] output_shiftRows = shiftRows.getOutput();
        System.out.println("==================================");
        System.out.println("\tShift rows");
        System.out.println("==================================");
        System.out.println("Input :");
        for (int i = 0; i < outputS_Box.length; i++) {
            System.out.print(outputS_Box[i]
                    + (i != outputS_Box.length - 1 ? ", " : "")
                    + ((i + 1) % 4 == 0 ? "\n" : ""));
        }
        System.out.println("============");
        System.out.println("Output :");
        for (int i = 0; i < output_shiftRows.length; i++) {
            System.out.print(output_shiftRows[i]
                    + (i != output_shiftRows.length - 1 ? ", " : "")
                    + ((i + 1) % 4 == 0 ? "\n" : ""));
        }
```

```
        // checker Mix columns
        Mix_columns mixColumns = new Mix_columns(output_shiftRows);
        String[] output_mixColumns = mixColumns.getOutput();
        System.out.println("==================================");
        System.out.println("\tMix columns");
        System.out.println("==================================");
        System.out.println("Input :");
        for (int i = 0; i < output_shiftRows.length; i++) {
            System.out.print(output_shiftRows[i]
                    + (i != output_shiftRows.length - 1 ? ", " : "")
                    + ((i + 1) % 4 == 0 ? "\n" : ""));
        }
        System.out.println("===========");
        System.out.println("Output :");
        for (int i = 0; i < output_mixColumns.length; i++) {
            System.out.print(output_mixColumns[i]
                     + (i != output_mixColumns.length - 1 ? ", " : "")
                     + ((i + 1) % 4 == 0 ? "\n" : ""));
        }

    }

}
```

Example	0e ce f2 d9 36 72 6b 2b 34 25 17 55 ae b6 4e 88	65 0f c0 4d 74 c7 e8 d0 70 ff e8 2a 75 3f ca 9c	f8 b4 0c 4c 67 37 24 ff ae a5 c1 ea e8 21 97 bc	72 ba cb 04 1e 06 d4 fa b2 20 bc 65 00 6d e7 4e	0a 89 c1 85 d9 f9 c5 e5 d8 f7 f7 fb 56 7b 11 14	db a1 f8 77 18 6d 8b ba a8 30 08 4e ff d5 d7 aa
After SubBytes	ab 8b 89 35 05 40 7f f1 18 3f f0 fc e4 4e 2f c4	4d 76 ba e3 92 c6 9b 70 51 16 9b e5 9d 75 74 de	41 8d fe 29 85 9a 36 16 e4 06 78 87 9b fd 88 65	40 f4 1f f2 72 6f 48 2d 37 b7 65 4d 63 3c 94 2f	67 a7 78 97 35 99 a6 d9 61 68 68 0f b1 21 82 fa	b9 32 41 f5 ad 3c 3d f4 c2 04 30 2f 16 03 0e ac
After ShiftRows	ab 8b 89 35 40 7f f1 05 f0 fc 18 3f c4 e4 4e 2f	4d 76 ba e3 c6 9b 70 92 9b e5 51 16 de 9d 75 74	41 8d fe 29 9a 36 16 85 78 87 e4 06 65 9b fd 88	40 f4 1f f2 6f 48 2d 72 65 4d 37 b7 2f 63 3c 94	67 a7 78 97 99 a6 d9 35 68 0f 61 68 fa b1 21 82	b9 32 41 f5 3c 3d f4 ad 30 2f c2 04 ac 16 03 0e
After MixColumns	b9 94 57 75 e4 8e 16 51 47 20 9a 3f c5 d6 f5 3b	8e 22 db 12 b2 f2 dc 92 df 80 f7 c1 2d c5 1e 52	2a 47 c4 48 83 e8 18 ba 84 18 27 23 eb 10 0a f3	7b 05 42 4a 1e d0 20 40 94 83 18 52 94 c4 43 fb	ec 1a c0 80 0c 50 53 c7 3b d7 00 ef b7 22 72 e0	b1 1a 44 17 3d 2f ec b6 0a 6b 2f 42 9f 68 f3 b1

Table 4.1 Test Cases

```
run:
0e ce f2 d9 36 72 6b 2b 34 25 17 55 ae b6 4e 88
================================
                S-Box
================================
Input :
0e, ce, f2, d9,
36, 72, 6b, 2b,
34, 25, 17, 55,
ae, b6, 4e, 88
===========
Output :
ab, 8b, 89, 35,
05, 40, 7f, f1,
18, 3f, f0, fc,
e4, 4e, 2f, c4
==================================
              Shift rows
==================================
Input :
ab, 8b, 89, 35,
05, 40, 7f, f1,
18, 3f, f0, fc,
e4, 4e, 2f, c4
===========
Output :
ab, 8b, 89, 35,
40, 7f, f1, 05,
f0, fc, 18, 3f,
c4, e4, 4e, 2f
==================================
              Mix columns
==================================
Input :
ab, 8b, 89, 35,
40, 7f, f1, 05,
f0, fc, 18, 3f,
c4, e4, 4e, 2f
===========
Output :
b9, 94, 57, 75,
e4, 8e, 16, 51,
47, 20, 9a, 3f,
c5, d6, f5, 3b
BUILD SUCCESSFUL (total time: 1 second)
```

```
run:
65 0f c0 4d 74 c7 e8 d0 70 ff e8 2a 75 3f ca 9c
================================
                S-Box
================================
Input :
65, 0f, c0, 4d,
74, c7, e8, d0,
70, ff, e8, 2a,
75, 3f, ca, 9c
===========
Output :
4d, 76, ba, e3,
92, c6, 9b, 70,
51, 16, 9b, e5,
9d, 75, 74, de
==================================
              Shift rows
==================================
Input :
4d, 76, ba, e3,
92, c6, 9b, 70,
51, 16, 9b, e5,
9d, 75, 74, de
===========
Output :
4d, 76, ba, e3,
c6, 9b, 70, 92,
9b, e5, 51, 16,
de, 9d, 75, 74
==================================
              Mix columns
==================================
Input :
4d, 76, ba, e3,
c6, 9b, 70, 92,
9b, e5, 51, 16,
de, 9d, 75, 74
===========
Output :
8e, 22, db, 12,
b2, f2, dc, 92,
df, 80, f7, c1,
2d, c5, 1e, 52
BUILD SUCCESSFUL (total time: 2 seconds)
```

```
run:
f8 b4 0c 4c 67 37 24 ff ae a5 c1 ea e8 21 97 bc
================================
                S-Box
================================
Input :
f8, b4, 0c, 4c,
67, 37, 24, ff,
ae, a5, c1, ea,
e8, 21, 97, bc
===========
Output :
41, 8d, fe, 29,
85, 9a, 36, 16,
e4, 06, 78, 87,
9b, fd, 88, 65
==================================
              Shift rows
==================================
Input :
41, 8d, fe, 29,
85, 9a, 36, 16,
e4, 06, 78, 87,
9b, fd, 88, 65
===========
Output :
41, 8d, fe, 29,
9a, 36, 16, 85,
78, 87, e4, 06,
65, 9b, fd, 88
==================================
              Mix columns
==================================
Input :
41, 8d, fe, 29,
9a, 36, 16, 85,
78, 87, e4, 06,
65, 9b, fd, 88
===========
Output :
2a, 47, c4, 48,
83, e8, 18, ba,
84, 18, 27, 23,
eb, 10, 0a, f3
BUILD SUCCESSFUL (total time: 1 second)
```

```
72 ba cb 04 1e 06 d4 fa b2 20 bc 65 00 6d e7 4e
================================
                S-Box
================================
Input :
72, ba, cb, 04,
1e, 06, d4, fa,
b2, 20, bc, 65,
00, 6d, e7, 4e
===========
Output :
40, f4, 1f, f2,
72, 6f, 48, 2d,
37, b7, 65, 4d,
63, 3c, 94, 2f
==================================
              Shift rows
==================================
Input :
40, f4, 1f, f2,
72, 6f, 48, 2d,
37, b7, 65, 4d,
63, 3c, 94, 2f
===========
Output :
40, f4, 1f, f2,
6f, 48, 2d, 72,
65, 4d, 37, b7,
2f, 63, 3c, 94
==================================
              Mix columns
==================================
Input :
40, f4, 1f, f2,
6f, 48, 2d, 72,
65, 4d, 37, b7,
2f, 63, 3c, 94
===========
Output :
7b, 05, 42, 4a,
1e, d0, 20, 40,
94, 83, 18, 52,
94, c4, 43, fb
BUILD SUCCESSFUL (total time: 1 second)
```

```
0a 89 c1 85 d9 f9 c5 e5 d8 f7 f7 fb 56 7b 11 14
================================
                S-Box
================================
Input :
0a, 89, c1, 85,
d9, f9, c5, e5,
d8, f7, f7, fb,
56, 7b, 11, 14
===========
Output :
67, a7, 78, 97,
35, 99, a6, d9,
61, 68, 68, 0f,
b1, 21, 82, fa
==================================
              Shift rows
==================================
Input :
67, a7, 78, 97,
35, 99, a6, d9,
61, 68, 68, 0f,
b1, 21, 82, fa
===========
Output :
67, a7, 78, 97,
99, a6, d9, 35,
68, 0f, 61, 68,
fa, b1, 21, 82
==================================
              Mix columns
==================================
Input :
67, a7, 78, 97,
99, a6, d9, 35,
68, 0f, 61, 68,
fa, b1, 21, 82
===========
Output :
ec, 1a, c0, 80,
0c, 50, 53, c7,
3b, d7, 00, ef,
b7, 22, 72, e0
BUILD SUCCESSFUL (total time: 1 second)
```

```
db a1 f8 77 18 6d 8b ba a8 30 08 4e ff d5 d7 aa
================================
                S-Box
================================
Input :
db, a1, f8, 77,
18, 6d, 8b, ba,
a8, 30, 08, 4e,
ff, d5, d7, aa
===========
Output :
b9, 32, 41, f5,
ad, 3c, 3d, f4,
c2, 04, 30, 2f,
16, 03, 0e, ac
==================================
              Shift rows
==================================
Input :
b9, 32, 41, f5,
ad, 3c, 3d, f4,
c2, 04, 30, 2f,
16, 03, 0e, ac
===========
Output :
b9, 32, 41, f5,
3c, 3d, f4, ad,
30, 2f, c2, 04,
ac, 16, 03, 0e
==================================
              Mix columns
==================================
Input :
b9, 32, 41, f5,
3c, 3d, f4, ad,
30, 2f, c2, 04,
ac, 16, 03, 0e
===========
Output :
b1, 1a, 44, 17,
3d, 2f, ec, b6,
0a, 6b, 2f, 42,
9f, 68, f3, b1
BUILD SUCCESSFUL (total time: 1 second)
```

CHAPTER 5 – Experiment results and discussion

AES is an iterative rather than a Feistel cipher. It is based on 'substitution-permutation', and is easy to implement in software .

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged into four columns and four rows for processing as a matrix.

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key. Looking at the results of applying the code in practice, I see that it is the same as the theoretical solution to the algorithm, so the operation is successful.

CHAPTER 6 – Conclusion

I concluded that the multiplicative inverse of the number zero in the S-Box is zero; it's a special case; and I also concluded that it is fantastic that the algorithm that we are studying theoretically can be represented programmatically, implying that we have completely mastered the algorithm.

REFERENCES

* https://www.youtube.com/watch?v=ApIFldw-Hpw&ab_channel=DG

* *Cryptography and Network Security Principles and Practice*, 7th Ed, **William Stallings**