**Network Security Assignment 1**

**Done by:**

**Aaron Agnel (2017010) and Bharath Kumar Thulasidoss (2017035)**

**Project 1: Advanced Encryption Standard**

**1) Introduction:**

We were allocated the advanced encryption standard project out of the two options. We have designed the project in such a way that any plain text can be fed into the program ( irrespective of the size of the file ) and the program generates an encrypted file and a decrypted file, with the input and the decrypted file being same, if the program was a success. We have used the 128 bit AES algorithm ( i,e input size, key size = 128 bits and number of rounds=10) in this program. We have created an AES class to house all the functions and variables required during the encryption and decryption process.

**2) Skeletal structure of the program:**

The program first asks for the input of the plain text file’s location. After that the program does the following:

1. Read the file 16 bytes at a time and store each byte as an element of an array and display the total size of the file and number of blocks generated.
2. The last block is checked to make sure that it has 16 bytes in it, else a padding (pkcs5) function is used to make grow the size of the last block to 16 bytes.
3. An AES class object is initialized with key size as 128 and the number of rounds as 10. This also initializes all the required variables like the input blocks, S-boxes, R-con, Mix column matrixes and the inverses of S-boxes and Mix column matrixes.
4. The master key of the AES class object is initialized and the key generation algorithm is run to generate 11 subkeys, each of size 16 bytes.
5. Encryption of the text is done block by block using the CBC approach with the AES encryption algorithm and stored in encrypt.txt.
6. Then decryption is done block by block using the CBC approach with the AES decryption algorithm and stored in decrypt.txt.

**3) Design and Implementation**

**a) Input:**

As described in the previous section, the function get\_blocks(f) is used to read 16-byte blocks. Then the function pad\_block\_PKCS5(blocks) is used to pad the last block.

**i) PKCS5 padding:**

If the block is not 16 bytes long then, we calculate (16 - (length of block)) and then append this value, that many times to the block. For example if the length is 9, then we append the value, 16-9 = 7, 7 times to the block.

**b) Key generation:**

After the initializing as described in section 2.1, we call key\_gen() function which does the subkey generation process and creates a total of 11 subkeys ( including the master key ). The process involves converting the master key into a 4\*4 byte array. Let’s call it A0. Now the subkey array would involve creating 10 more 4\*4 byte arrays. Let’s call them A1, A2,... .So the first column of Ai is calculated as ( 1<=i<=10 and 1<=j<=3) :

Ai,1 = S-box( Ai-1,0[1:] + Ai-1,0[:1])^R-con[i]^Ai-1,3

Whereas the remaining columns are calculated as

Ai,j = Ai-1,j^Ai,j-1

This is how the key generation is done and stored in the vector sub\_keys.

**c) Encryption :**

We have used CBC chaining in our program, with a randomly generated 16-byte initialization vector (IV). CBC involves XOR-ing the current 16-byte block with the previous 16-byte encrypted block ( or the IV, in the case of the first block ) and then use that output for the encryption process. The output from the previous line is fed into encrypt\_block(i) function, where i is the index of the current input block XORed with the previous encrypted block. This function converts the current input block to a 4\*4 byte array called the state vector, S. Now we XOR the state vector with the first subkey A0. Now we run a loop for the number of rounds we have ( i,e 10 ). In each iteration we do the following:

1. **Call sub\_bytes function:**

Here each element of the state vector is replaced by hashing into the S-Box array. For example if the S1,1 byte was equal to \x09 then we would replace it by looking at the value stored in S-Box0,9  whichhere is \x01.

1. **Call Shift\_rows function**

Here other than the first row of the state vector, the remaining rows are circularly left-rotated by 1,2 and 3 places respectively. That is:

Si=Si[i:]+Si[:i] where (0<=i<=3)

1. **Call Mix\_Col function ( except for the last round )**

Mix columns involves doing a matrix multiplication of the state vector with the mix\_col array. The matrix multiplication is done in the 28 galois field. The multiplication happens by converting each byte to its polynomial representation and then multiplying them using the distributive law and then dividing them ( XORing here ) by x8+x4+x3+x+1. But as this field obeys distributive law over addition, we can represent any number in the form 2x or 2x+1 and hence we need to only worry about multiplying a number by 2 and see if the answer is in the field, if not then we xor with the irreducible polynomial ( in this case 0x11B ) to bring it within the field. Example 6 = 3\*2 or (2+1)\*2 hence 2\*2 + 2\*1 = 2\* 2 + 1. So now all we need to do is multiply 2 with 2 in the galois field and then XOR 2 with it ( equivalent of addition ).

A sudo code to accomplish this is given here. Consider 2, 1 byte numbers b and c,

For i in range(8):

Check if c or b is not 0:

If so then check if c’s last bit ==1:

Then XOR b to the answer

Divide c by 2

Get the LSB of b as carry

Multiply b by 2

If carry == 1:

Then xor b with 0x11B

This algorithm simplifies the distributive multiplication by multiplying b with the ith

bit of c by left shifting c the same number of times as the ith bit. That is if b was to be multiplied by x2 then we shift b to the left by 2 times and then add it to the answer. If we detect an overflow in any of the steps, then xor with 0x11B as discussed above.

1. Xor state vector with subkey Acurrent\_round

After this we store the encrypted block and append it to the output string. The output string is then written to the encrypt.txt file.

**d) Decryption**

Similarly for decryption, we read the encrypt.txt file and do the same as encryption but with the following changes to the algorithm which results in decryption.

1. InvShiftRows - Does the inverse of Shift rows and cyclically shifts(right rotates) the rows 2-4 of the state vector.

State\_vector[i] = state\_vector[i][4-i:]+state\_vector[i][0:4-i] for 0<i<4

1. InvSubBytes - Converts the byte into hex value and uses the first value as row and second column to index into the inverse Rijndael S box. So for {a9} we would do state\_vector[i][j] = inv\_SBox[10][9]

1. InvMixColumns - We follow the same procedure as mixcloumns, however here we will use the inverse column matrix [[0e,0b,0d,09],[09,0e,0b,0d],[0d,09,0e,0b],[0b,0d,09,0e]]
2. Invaddroundkey - Since its only xor it is equivalent to addroundkey.

**Pseudocode - decryption**

addroundkey(number\_of\_rounds)

For round from num\_of\_round-1 to 1:

invshiftrow()

invsubbytes()

addroundkey(round)

invmixcolumns()

invshiftrow()

invsubbytes()

addroundkey(0)