CMPE 494 ASSIGNMENT 1 REPORT
Gözde Ünver 2018400309
Sevde Sarıkaya 2017400081
Ahmet Necip Görgülü 2017400090

HOW AES WORKS

AES (Advanced Encryption Algorithm) is an encryption algorithm that puts the original input through multiple methods for 10 rounds(128 bit case) in order to conceal the original information. The methods used in the AES are:

Key Expansion: Initially determined key, through a series of operations we obtain 11 round keys out of that one key. Whether we are encrypting or decrypting, 11 round keys are generated from the initially given key to be used in rounds.

AddRoundKey: In this method the current state matrix gets XOR'ed with round key.

SubBytes: Every value in the current state matrix gets replaced with their corresponding values from S-BOX(Substitution Box). This operation provides the non-linearity in the algorithm. The S-Box used in this method is derived deliberately from the multiplicative inverse over Galois Field which is known to have good non-linear properties. S-Box is also a derangement in order to avoid attackers. While decrypting, Inverse S-Box is used instead of S-Box.

ShiftRows: Each row in the current state matrix gets shifted towards left by a certain offset. First doesn't get shifted at all. Second row is shifted 1, third row is shifted 2 and the fourth row is shifted 3 towards left.

MixColumns: In this method each column of the current state matrix is multiplied by a fixed polynomial. Together with the ShiftRows function it creates diffusion for the Advanced Encryption System.

Encryption:

First Round: AddRoundKey
After first round for 9 rounds: SubBytes, ShiftRows,MixColumns,AddRoundKey (respectively)
Final Round: SubBytes, ShiftRows, AddRoundKey (respectively)

Decryption: For decryption, the inner workings of methods have to be reversed as well as the order of the methods. Therefore in order to decrypt an already encrypted message, decryption algorithms for each method have to be implemented separately.

First Round: AddRoundKey, ShiftRows, SubBytes (respectively)
After first round for 9 rounds: AddRoundKey, MixColumns, ShiftRows, SubBytes (respectively)
Final Round: AddRoundKey

HOW WE IMPLEMENTED

We wrote a main function to call other functions and functions for key expansion, shift rows, sub bytes, mix columns. Functions for key expansion, shift rows, sub bytes and mix columns work both for encryption and decryption.

MB/sec: 0.000235 (10 lines)

Encryption/Decryption speed decreases as the size of your input decreases since the overhead becomes more dominant. This value was obtained from taking the average of 10 line inputs (160 bytes).

Main

In our main function we read the input and key files in order to fill up our matState and key 2d arrays. According to the option input we determine the "dec" boolean value. We use the keyExpansion method to obtain 11 round keys. After that in a while loop we take a line of input, put it through our encipher method, write the resulting matrix properly to the output file then repeat until there are no more lines in the inputFile.

Built-in arrays

We initialized some arrays as static built-in arrays to directly use them in our code.
S-Box: It is a 16x16 matrix that we use in SubBytes operation. Even though it was meant to be a 16x16 byte matrix we turned it into a single dimensional array with 256 elements for ease of implementation.
Inverse S-Box: Inverse of S-Box that we use in SubBytes operation while doing decryption.
Rijndaelmatrix: It is a 4x4 two dimensional array. It stores the array whose rows are used for multiplication with the columns of the input block in the encryption mixColumn operation. The cells are either 2, 3 or 1.
invMixColumnMatrix: It is a 4x4 two dimensional array. It stores the array whose rows are used for multiplication with the columns of the input block in the decryption mixColumn operation. The cells are either 9, 11, 13 or 14.
multiplyBy9, multiplyBy11, multiplyBy13, multiplyBy14: They are one dimensional arrays of length 256. They are used for the decryption operation in mixColumn. k'th cell of them stores the multiplication result of the value "k" and 9, 11, 13 or 14.
rconst: It is a one dimensional array with length 10. It stores round constant to be used in Key Expansion.

subBytes(int [][]state,boolean dec)

This method takes the state matrix and a boolean value "dec" as parameters. Boolean value "dec" determines whether we are going to use S-Box or Inverse S-Box(whether we are encrypting or decrypting). According to the dec value, we replace the values in state matrix with the corresponding values in S-Box or Inverse S-Box. After the replacements state matrix is returned as a result.

mixColumns(int [][]state,boolean dec)

Parameter "dec" is true if the user typed the decryption option, false if the user typed encryption option. This function can execute both of them according to the value of "dec". "State" array is a 2 dimensional input block to shift its rows. This operation is a matrix multiplication. When a row is multiplied with a column, the result is stored in a new array called "temp" of size 4x4. This temp array is bpth used for encryption and decryption. For each row of the Rijndaelmatrix and each column of the state array, each corresponding cell is multiplied and each separate multiplications are xor'ed in the end and stored at the

temp cell. Multiplication is different from a regular multiplication. If the cell in the row of a Rijndaelmatrix is 1, then its multiplication with the cell in the state array gives the cell in the state array so nothing changes. This state value is xor'ed with the destination temp cell. While traversing each cell of the Rijndaelmatrix's row and state array's column, the results are always xor'ed with the correct cell of the temp array. If the cell in the row of a Rijndaelmatrix is 2, then multiplyBy2 helper function is called and the result is xor'ed with the temp cell. If the cell in the row of a Rijndaelmatrix is 3, then first multiplication with 2 is executed and then multiplication by 1 is executed. It is like we are separating 3 into 2+1 After that the result obtained from multiplication by 2 and the initial state cell value are both xor'ed with the cell of the temp array because 2+1 means xor the results of 2 and 1.
For the decryption part, the invMixColumnMatrix is used instead of Rijndaelmatrix. We used built-in result matrices of numbers multiplied by 9, 11, 13, or 14 for the inverse mix column operation. For each row of the invMixColumnMatrix and each column of the state array, each cell multiplication is checked. If the state cell is multiplied with 9 in the invMixColumnMatrix, then the result is obtained by indexing multiplyBy9[state cell value]. The result is xor'ed with the cell of the temp array. Similarly, multiplications with 11, 13 and 14 are done in the same way using their own arrays.
In the end, temp array is returned for both encryption and decryption options to update the state array in the main function.

shiftRows(int[][] state,boolean dec)

Parameter "dec" is true if the user typed the decryption option, false if the user typed encryption option. This function can execute both of them according to the value of "dec". "State" array is a 2 dimensional input block to shift its rows. For the encryption, except for the first row, all the rows are shifted to the left by (their row numbers-1) places. For the shifting operation, an array of size 4 and named "temp" stores the shifted versions of a row and after all the cells of a row are stored at their new places in the temp array via a for loop, temp is cloned to the correct row of the state array and then temp array is reused for the next row. For the second row, a cell at the place "i" in the second row of the state array is stored at the (i-1+4)%4 place of the temp array. For the third row cells are stored at the (i-2+4)%4 place and the fourth row cells are at the (i-3+4)%4 place of the temp array. If we say that the initial row is row 1, the last row is row 4. General rule is that a cell at row (k+1) in the state array will appear at (i-k+4)%4. Before taking the modulo 4 of the value, 4 is added because shifting is a cyclic operation and the cells at the beginning of the array are shifted to the end of the array so the addition of 4 provides that. In order to prevent indexing out of bounds, modulo 4 makes sure indexes are less than 4.
Similarly, the decryption shifts the rows to right by (their row numbers-1) places. Again an array of size 4 and named "temp" is created. A cell at the row (k+1) is shifted right by this formula: (i+k)%4. Modulo 4 provides that the index is not bigger than or equal to the length of the array. "Temp" array stores the shifted versions of the cells of a row and at the end of storing all of the cells for that row, temp is cloned to the state array at the exact row.
Finally, this function returns the updated version of the state array for both encryption and decryption options to update the state array in the main function.

multiplyBy2(int num)

This function is used for the multiplication by 2 statements in the mix column operation. Normally this is a polynomial multiplication but for the coding implementations there is a coding convention and it is simplified to this algorithm: When a cell of the input matrix is going to be multiplied with 2, this means

that the number is first going to be shifted to the left by 1 place. If originally the left most bit of the number is 1, then the shifted number will also be xor'ed with 0x1b. This function does these operations on the given number. First the number is shifted to the left by 1 and the number is bit wise &'ed with 255 to only store the first 8 bits of the integer number and the result is stored in the "shiftedNum". Then the first bit of the original "num" is checked by bit wise & with (1<<7). The result of this operation only returns the value in the seventh bit and the rest are all zeros. If the result is (1<<7) then it means that the seventh bot of num is 1 so the shiftedNum is xor'ed with 0x1b. Finally the resulting shiftedNum is returned.

KeyExpansion(int[][] key)

It takes an initial key with 4-words (16 byte). This key is read from keyFile which includes one line of 32 hex characters. After reading this file, it is transformed to a 4x4 array. Each cell has 2 hex chars (1 byte). Key Expansion takes this input key and generates 44 words (176 bytes). So, at the end we have 176/16 = 11 keys which are used during each step of AES in the order.

N = length of the key -> 4 words for AES-128
K_i = The ith word in the initial key
W_i = The ith 32-bit word of the expanded key.
The function used to generate keys:

$$W_i = \begin{cases} K_i & \text{if } i < N \\ W_{i-N} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus rcon_{i/N} & \text{if } i \geq N \text{ and } i \equiv 0 \pmod N \\ W_{i-N} \oplus \text{SubWord}(W_{i-1}) & \text{if } i \geq N, N > 6, \text{ and } i \equiv 4 \pmod N \\ W_{i-N} \oplus W_{i-1} & \text{otherwise.} \end{cases}$$

Resource: Wikipedia (https://en.wikipedia.org/wiki/AES_key_schedule)

So, this is what the keyExpansion method does. It copies the initial key exactly to the first round key. Then if the index of the round key is bigger than 4 and it is divisible by 4, it takes the previous round word, rotates and substitutes it. Then it XORs this value with the N previous round word and i/4'th round constant. Else, it just XORs N previous and the last word. We don't need to have the third case written on the function above i%4 cannot be 4. It can be 0,1,2,3.

subWord(int [] word):

It takes a word (4 length of array) and applies AES S-Box to each byte of the word. AES S-Box is the Rijndael S-box (substitution box). It is a lookup table.

rotWord(int[] word): it rotates the bytes of the given word. (b0 b1 b2 b3) becomes (b1 b2 b3 b0)

addRoundKey(int[][] state, int index): index represents the index of the round key. So, the function gets the corresponding round key and gets XOR of each byte of the round key and the state matrix. For example the cell in (x, y) in the state matrix is XORed with the cell in (x,y) in the round key.

Helper functions:

getRoundKeyMatrix(int index): It returns the round key matrix in the given index.
getColumn(int[][] matrix, int column): returns the column of the matrix in the given index.
printFunc(int [][] matState, int row, int col): it prints the matrix by changing values to hex.
printFile(int [][] matState): It prints the matrix to a file by changing its values to hex.