SIL 765

AES

Harsimrat Singh
2015CS50284

Sudeep Agarwal
2015CS50295

What is AES?

AES (Advanced Encryption Standard) is a symmetric key cipher based on multiple rounds of substitution and permutation, developed by Belgian scientists Vincent Riimen and Joan Daemen. It is published as FIPS 197 by NIST in 2001. As it is a symmetric key cipher, it uses single key for encryption and decryption. Till today, there is no possible attack that could decrypt ciphertext without the knowledge of key. Usually, key size of 128 bits, 256 bits, ... are used in this encryption scheme.

How is AES implemented?

The overall structure of AES encryption of N rounds is as shown in figure 1. We give plaintext as input to this function, given a key, and we get ciphertext as output in each round. Then we give this output as input to next round and we again get another ciphertext (Intermediate ciphertext). We continue doing this till N rounds and we get out final ciphertext. Similarly, we can decrypt any ciphertext by performing reverse AES algorithm for N rounds.

Each round of AES consists of 4 steps : Substitution bytes, Shift rows, Mix Columns, Add round keys. Along with this we also have to produce different keys for different rounds. During encryption, N keys are used for N rounds while during decryption same keys are used but in reverse order.

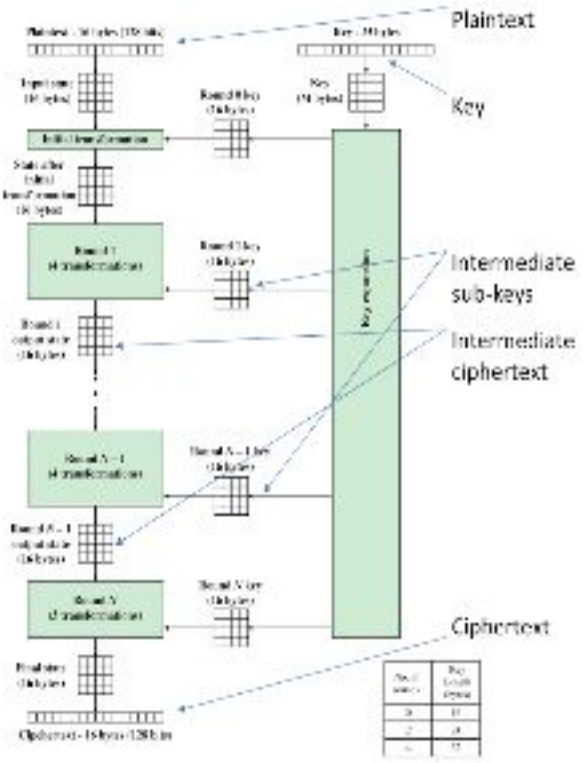
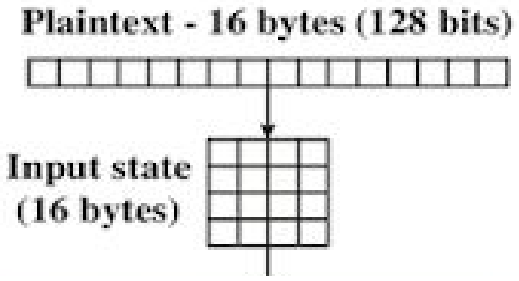
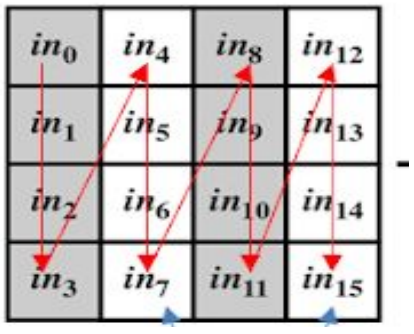


Figure 1

Figure 2





As we can see in figure 2, we modify plaintext a little bit so that we can perform AES operations. Basically we write plaintext in form of matrix as shown and we perform operations on this matrix. So the final output is again a matrix and we write it in scalar form and we get our ciphertext.

We have 6 files in which we implemented 4 steps of AES and generation of keys for each round

1.) Subtitution_box.js and gf_inverse.js
2.) Shiftrows.js
3.) Mix_column.js
4.) Key_expansion.js
5.) Core.js

Subtituion_Box.js and gf_inverse.js

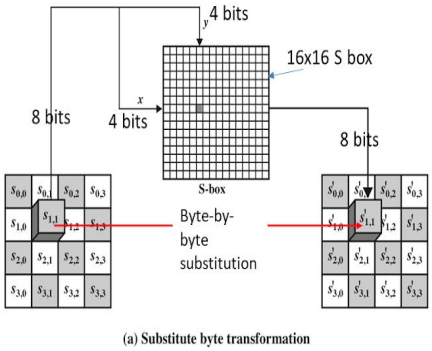


Figure 3

In substitution_box.js, we have a **function substitution** which takes a matrix of 4x4 and returns matrix of same dimension. It substitutes values of input matrix byte-by-byte to new value that **function sbox** gives. Sbox takes a hexadecimal value, basically an element of 4x4 matrix, and follows and algorithm given in figure 4 and replaces old value with this new value in that input matrix. What the algorithm does is take 8 bits of an element of 4x4 matrix and calculates its multiplicative inverse in GF(2^8) and that's where **gf_inverse.js** file comes into play. In this file, we have 3 functions that computes inverse, division and multiplication of 2 matrices or 1 matrix and 1 vector in GF(2^8). So when we have our multiplicative inverse, we modify it to a column vector and multiply it with a matrix and add a vector to it (all operations in GF(2^8)). Then change column vector to byte and this byte is our substitution for the element. So function **sbox** return this value to function substitution. This procedure is followed during encryption. For decryption, we have to compute inverse for this value, so we follow all steps of S-box in reverse order and that's what we are doing in function

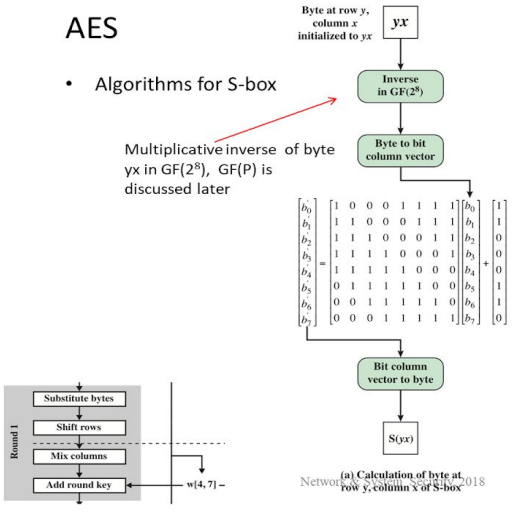


Figure 4

inverseSbox. Function **inverseSubstitute** calls inverseSbox and replaces matrix element with value computed from **inverseSbox** value.

Shift_row.js

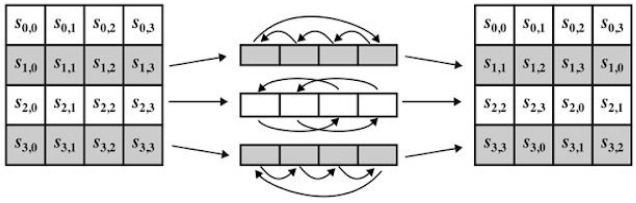


Figure 5

In this we have 4 functions, 2 for decryption and 2 for encryption. During encryption, we just shift rows left by their row number in cyclic order as shown in figure 4. This is done by **shiftRow** function with the help of **rotateLeft**. Similarly during decryption, we shift rows towards right which is done by **inverseShiftRow** function with the help of **rotateRight**.

Mix_column.js

Int this, we multiply our input with a fix matrix which is indivisible in that field which is GF(2^8).

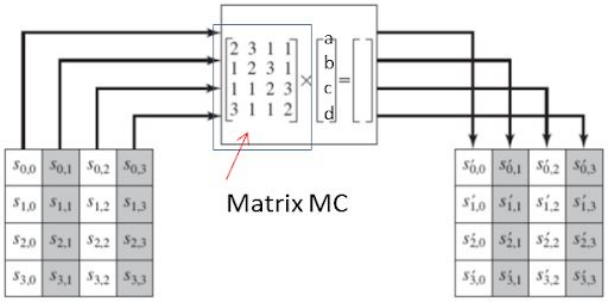


Figure 6

During decryption, we multiply input by inverse of that matrix in GF(2^8).

Key_expansion.js

We first change representation of key from scalar to matrix form as we had done with plaintext by passing it through **hexToBinaryString** and then **stringToMatrix**. Then we will change it as shown in figure 7. w0, w1, w2… have 16 bits in function **matrixToWs**. For first round we will use original key as it is without changing it. But for other rounds we have to modify the key so

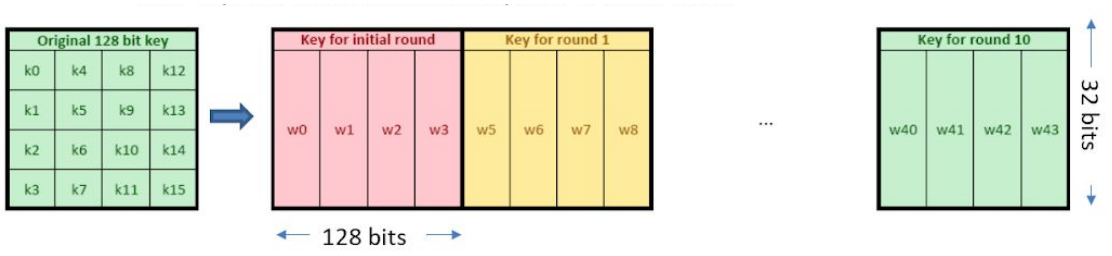


Figure 7

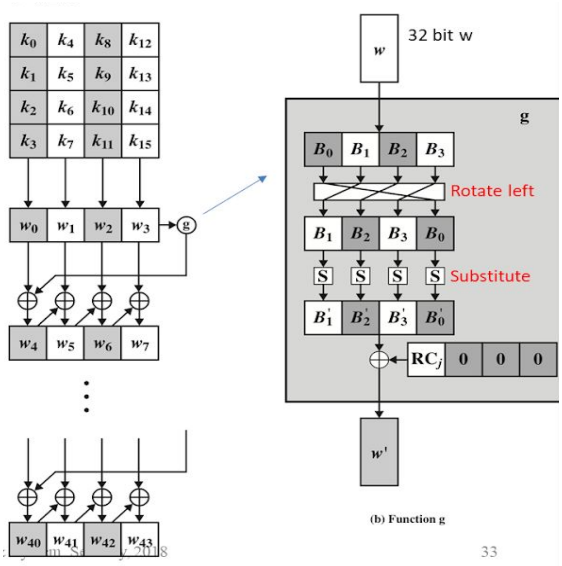


Figure 8

that any bit in original key is changed all of next key bits are affected. So we go through the procedure as shown in figure 8. Let's say we want to get key for next round given key for current round. To get first byte of next key we take xor of first byte of current key with last byte of current key but with small manipulations. By manipulation, we mean that we pass last byte of current key through a function called g. We implement it in function **gBox**. In gbox, we send whole w i.e. 16 bits. First we rotate left bytes by 1 in circular way. Then we replace value of each byte by value we get from passing them through sBox. Then we take xor of first byte with RC_j, where RC_j is given by,

$RC_j = 2 * RC_j$ in $GF(2^8)$, where j is number of current round

and $RC_1 = 1$

So what we get from gBox we take xor of it with 4 byte of current key and we get first 4 byte or w_4 for next key. To get next 16 bytes we take xor of next 16 bytes of current key with previous 16 bytes of next key. And hence we will get our next key for next round. We keep doing this and we will get key corresponding to that round.

Core.js

In this file, we have 2 major functions - one for decryption **aesDecryption** and other for encryption **aesEncryption.** In these functions, we combine all steps for AES together to encrypt or decrypt the data.