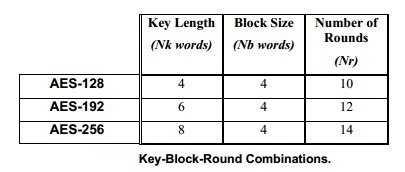
**~~Introduction~~**

~~The~~ **~~Advanced Encryption Standard (AES)~~** ~~algorithm, also known as the~~ **~~Rijndael algorithm~~** ~~is a symmetrical block cipher algorithm that takes plain text in blocks of 128 bits and converts them to ciphertext using keys of 128, 192, and 256 bits. AES encryption used in a lot of ways, including wireless security, processor security, image encryption, file encryption, and SSL/TLS. In fact, our web browser also use AES to encrypt our connection. Since the AES algorithm is considered secure, it is in the worldwide standard.The AES algorithm uses a substitution permutation, with multiple rounds to produce cipher string. The number of rounds depends on the size of key we are using. A 128-bit key size consider ten rounds, a 192-bit key size consider 12 rounds, and a 258-bit key size has 14 rounds. Each of these rounds requires a round key, but since only one key is inputted into the algorithm, this key needs to be expanded to get keys for each round, including round zero. The NSA (National Security Agency) United States Department of Defense they are using AES to encrypt Top Secret information. So thats why AES has gained the confidence of various industries. If it's good enough for Security Agency like the NSA, then it must be good enough for businesses.~~

**Algorithm Specification**

**Cipher**

For the AES algorithm, the length of the Cipher Key is 128, 192, or 256 bits. The key length is represented by Nk = 4, 6, or 8, which reflects the number of 32-bit words (number of columns) in the Cipher Key. For the AES algorithm, the number of rounds to be performed during the execution of the algorithm is dependent on the key size. The number of rounds is represented by Nr, where Nr = 10 when Nk = 4, Nr = 12 when Nk = 6, and Nr = 14 when Nk = 8. The only Key-Block-Round combinations that conform to this standard are given in Figure.

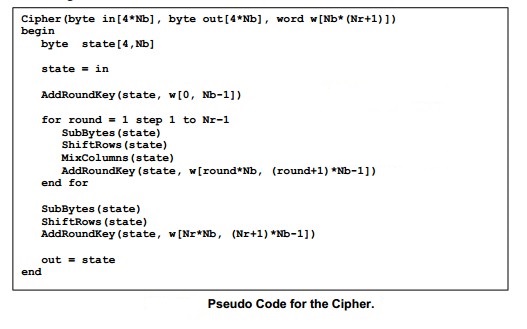


For both its Cipher and Inverse Cipher, the AES algorithm uses a round function that is composed of four different byte-oriented transformations:

1. Byte substitution using a substitution table (S-box),
2. Shifting rows of the State array by different offsets,
3. Mixing the data within each column of the State array, and
4. Adding a Round Key to the State

At the start of the Cipher, the input is copied to the State. After an initial Round Key addition, the State array is transformed by implementing a round function 10, 12, or 14 times (depending on the key length), with the final round differing slightly from the first Nr-1 rounds. The final State is then copied to the output. The round function is parameterized using a key schedule that consists of a one-dimensional array of four-byte words derived using the Key Expansion. The Cipher is described in the pseudo code.

The individual transformations SubBytes() , ShiftRows() , MixColumns() , and AddRoundKey() – process the State and are described next. All Nr rounds are identical with the exception of the final round, which does not include the MixColumns() transformation.



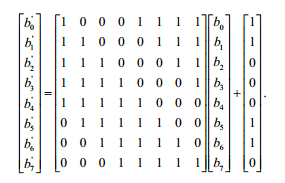
**SubBytes() Transformation**

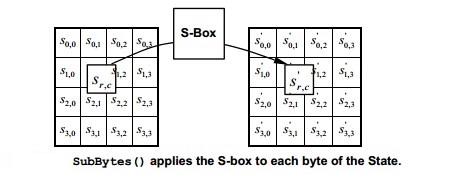
The SubBytes() transformation is a non-linear byte substitution that operates independently on each byte of the State using a substitution table. This S-box which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field (Galoi’s Field) GF(28),

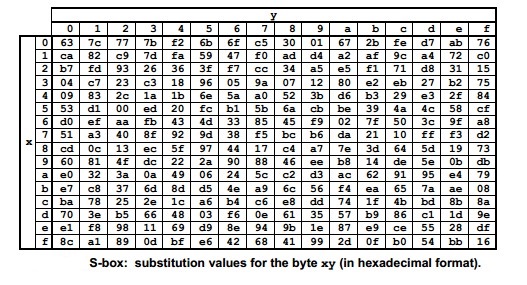
2. Apply the following affine transformation (over GF(2) ):

  
 In matrix form, the affine transformation element of the S-box can be expressed as:

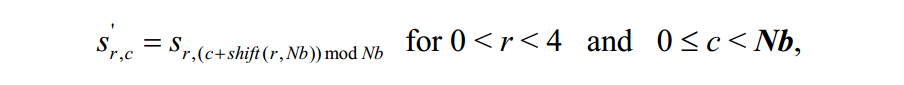




The S-box used in the SubBytes() transformation is presented in hexadecimal form in Figure. For example, if s1,1 = {53}, then the substitution value would be determined by the intersection of the row with index ‘5’ and the column with index ‘3’ in Figure. This would result in s’1,1 having a value of {ed}.

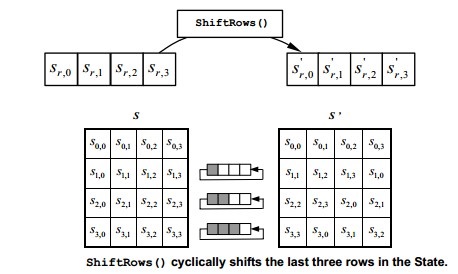


**ShiftRows() Transformation**

In the ShiftRows() transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, r = 0, is not shifted. Specifically, the ShiftRows() transformation proceeds as follows:  


where the shift value shift(r,Nb) depends on the row number, r, as follows:  
shift(1,4) = 1; shift(2,4) = 2 ; shift(3,4) = 3 .

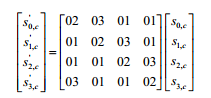
This has the effect of moving bytes to “lower” positions in the row, while the “lowest” bytes wrap around into the “top” of the. Figure illustrates the ShiftRows() transformation.



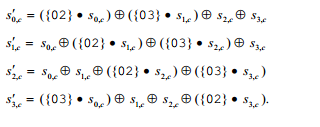
**MixColumns() Transformation**

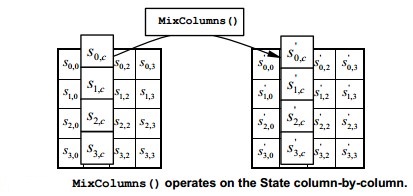
The MixColumns() transformation operates on the State column-by-column, treating each column as a four-term polynomial. The columns are considered as polynomials over GF(28) and multiplied modulo x4 + 1 with a fixed polynomial a(x), given by  
 a(x) = {03} x3 + {01}x2 + {01}x + {02} .

This can be written as a matrix multiplication. Let  
s’(x) = a(x) s (x):



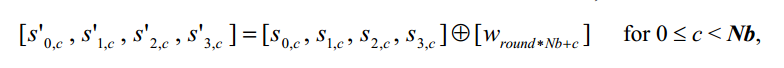
As a result of this multiplication, the four bytes in a column are replaced by the following:





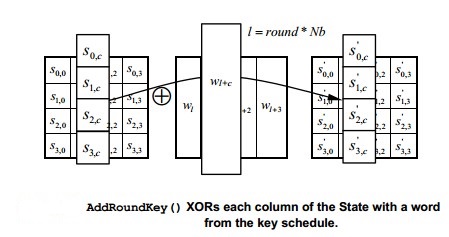
**AddRoundKey() Transformation**

In the AddRoundKey() transformation, a Round Key is added to the State by a simple bitwise XOR operation. Each Round Key consists of Nb words from the key schedule. Those Nb words are each added into the columns of the State, such that



Where [wi] are the key schedule words, and round is a value in the range 0 <= round <= Nr. In the Cipher, the initial Round Key addition occurs when round = 0, prior to the first application of the round function. The application of the AddRoundKey() transformation to the Nr rounds of the Cipher occurs when 1 <= round <= Nr.

The action of this transformation is illustrated in Figure, where l = round \* Nb



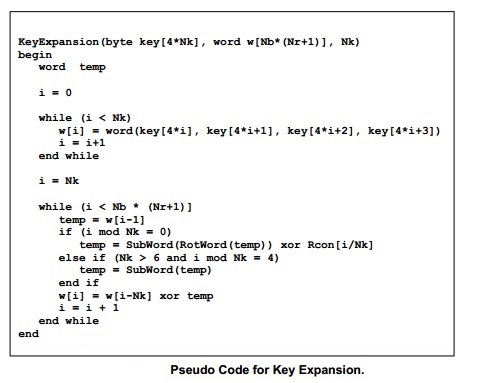
**Key Expansion**

The AES algorithm takes the Cipher Key, K, and performs a Key Expansion routine to generate a key schedule. The Key Expansion generates a total of Nb\*(Nr + 1) words: the algorithm requires an initial set of Nb words, and each of the Nr rounds requires Nb words of key data. The resulting key schedule consists of a linear array of 4-byte words, denoted [wi ], with i in the range 0 <= i < Nb\*(Nr + 1).

The expansion of the input key into the key schedule proceeds according to the pseudo code.  
SubWord() is a function that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word. The function RotWord() takes a word [a0,a1,a2,a3] as input, performs a cyclic permutation, and returns the word [a1,a2,a3,a0]. The round constant word array, Rcon[i] , contains the values given by [xi-1,{00},{00},{00}], with xi-1 being powers of x (x is denoted as {02}) in the field GF(28)

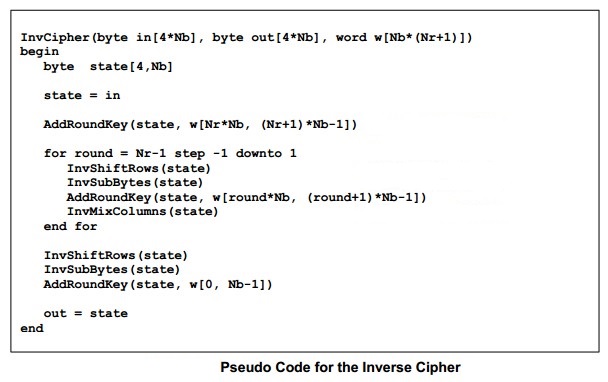
From Figure, it can be seen that the first Nk words of the expanded key are filled with the Cipher Key. Every following word, w[i], is equal to the XOR of the previous word, w[i-1], and the word Nk positions earlier, w[i-Nk]. For words in positions that are a multiple of Nk, a transformation is applied to w[i-1]prior to the XOR, followed by an XOR with a round constant, Rcon[i] . This transformation consists of a cyclic shift of the bytes in a word (RotWord() ), followed by the application of a table lookup to all four bytes of the word (SubWord() ).

It is important to note that the Key Expansion routine for 256-bit Cipher Keys (Nk = 8) is slightly different than for 128- and 192-bit Cipher Keys. If Nk = 8 and i-4 is a multiple of Nk, then SubWord() is applied to w[[i-1]] prior to the XOR.



**Inverse Cipher**

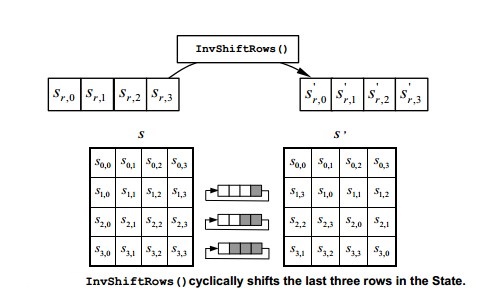
The Cipher transformations can be inverted and then implemented in reverse order to produce a straightforward Inverse Cipher for the AES algorithm. The individual transformations used in the Inverse Cipher - InvShiftRows(), InvSubBytes(), InvMixColumns(), and AddRoundKey() – process the State and are described in the following subsections. The Inverse Cipher is described in the pseudo code in Fig.



**InvShiftRows() Transformation**

InvShiftRows() is the inverse of the ShiftRows() transformation. The bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, *r* = 0, is not shifted. The bottom three rows are cyclically shifted by *Nb* - *shift*(*r*, *Nb*) bytes, where the shift value *shift(r,Nb)* depends on the row number, and is given in equation.

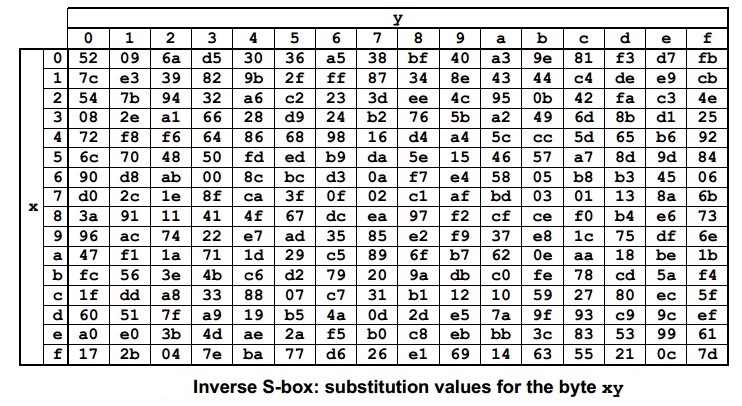




**InvSubBytes() Transformation**

InvSubBytes() is the inverse of the byte substitution transformation, in which the inverse Sbox is applied to each byte of the State. This is obtained by applying the inverse of the affine transformation (5.1) followed by taking the multiplicative inverse in GF(28).

The inverse S-box used in the InvSubBytes() transformation is presented in Fig:

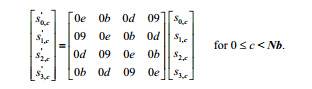


**InvMixColumns() Transformation**

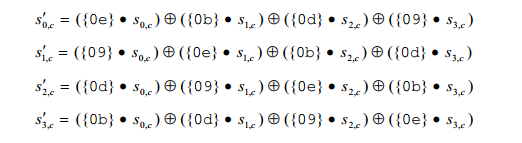
InvMixColumns() is the inverse of the MixColumns() transformation.  
InvMixColumns() operates on the State column-by-column, treating each column as a fourterm polynomial as described in Sec. 4.3. The columns are considered as polynomials over GF(28) and multiplied modulo *x*4 + 1 with a fixed polynomial *a*-1(*x*), given by

a-1 (*x*) = {0b} x3 + {0d} x2 + {09}*x* + {0e}

This can be written as a matrix multiplication. Let  
s’(x) = a-1(x) s (x):



As a result of this multiplication, the four bytes in a column are replaced by the following:



**Inverse of the AddRoundKey() Transformation**

AddRoundKey() , which was described previously is its own inverse, since it only involves an application of the XOR operation.