Advanced Encryption Standard

Keane J. Moraes & Nadeem Ahamed

History

The Advanced Encryption Standard (just called AES) is the standard for encryption of electronic data established by the National Institute of Standards and Technology (NIST). This name is just the official name for the Rjindael encryption algorithm that was the winner of the AES Selection Process. The AES specifications demanded that any standard adopted be available free of royalties anywhere in the world. The choosing of a Belgian model signalled the NIST's commitment to the internationalization of AES.

Rjindael

Rjindael is a family of block cipher encryption schemes devised by 2 Belgian cryptographers, Vincent Rjimen and Joan Daemen. The family of ciphers contained variants for 128, 192 and 256 bit key lengths but had a fixed block size of 128 bits. (From the 'Encryption' section onwards, the names Rjindael and AES will be used interchangeably)

AES Selection Process

Beginning in 1997, NIST worked with industry and the cryptographic community to develop an Advanced Encryption Standard (AES). The overall goal was to develop a Federal Information Processing Standard (FIPS) specifying an encryption algorithm capable of protecting sensitive government information well into the 21st century.

In September 1997, NIST made a formal call for algorithms for a symmetric key block ciphers. In August 1998 and March 1999, NIST held two of conferences where they finalized 5 algorithms from the 15 proposals. The five finalists were MARS, RC6, Rijndael, Serpent, and Twofish. Next, the algorithms were subject to a more in-depth review 'Round 2'. Until the end of Round 2 in May 2000, NIST solicited public comments on the remaining algorithms. Comments and analysis were actively sought by NIST on any aspect of the candidate algorithms. After the Third AES Conference (AES3), submitters of the AES finalists were invited to attend and engage in discussions regarding comments on their algorithms. All papers proposed for AES3 were considered as official Round 2 public comments.

In October 2000, NIST declared that Rjindael was finalized as the AES FIPS. In Novmeber 2001, NIST published the FIPS 197 as the official publication for the Advanced Encryption Standard.[1]

Modern Usage

AES is the international accepted standard for encrypting any electronic data in the 21st century. It succeeded DES, which was showing its age by the late 90s. AES is not only used as a symmetric key block cipher but also in message authentication constructions. Intel and AMD both have several subroutines for speeding up AES encryption and decryption[2]. However, with the dawn of quantum computers, AES-128 faces a unique threat.[3]. That being said, AES-192 and AES-256 are secure from this threat. A point to note however is that in the real world, solely AES is not used to encrypt messages. There are several other "add-ons" that must be applied onto the block cipher to make it usable in the real world. We will see exactly what these are in the sections 'Semantic Security' and 'Authenticated Encryption'.

[1] To read the official publication use this link NIST FIPS PUB197

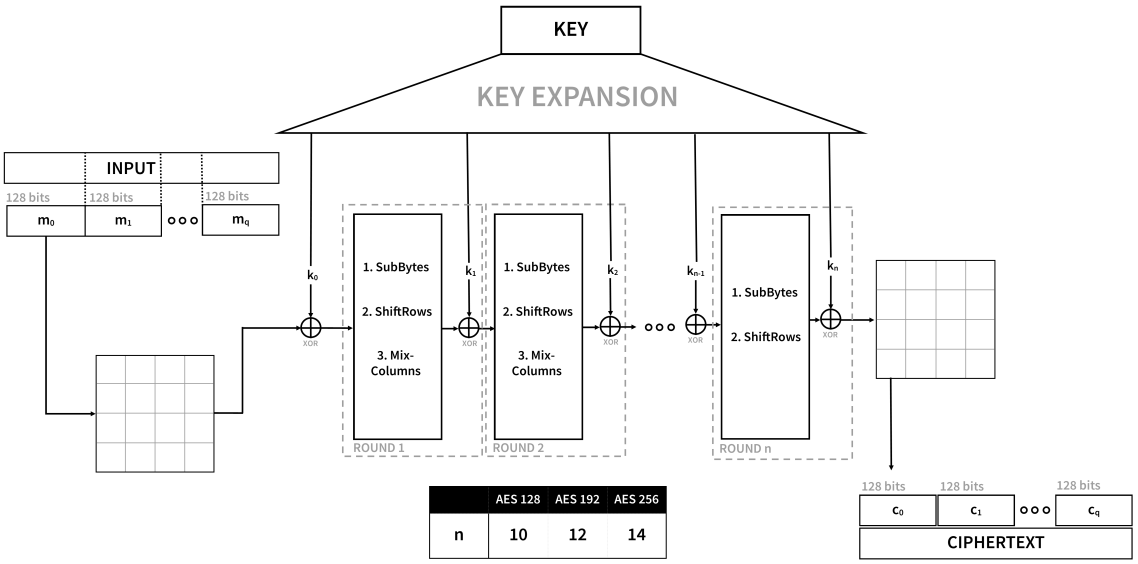
[2] To read about these specific instruction sets, use this link for Intel

[3] Refer to Cryptanalysis for more information

Encryption and Decryption Algorithms

Encryption

Rjindael is based on a substitution-permutation network with a block size of 128 bits irrespective of the key size. In Rjindael, all the 'work' occurs on a 4×4 matrix. Each element of the matrix is 8 bits (1 byte) and thus with 16 elements, we have 128 bits as required. Being a substitution-permutation network, every round, a round function substitutes and permutes the elements of the matrix while XORing with a 4×4 key matrix. A high level view of AES is given below.



The two keys features to describe at length are :

1. Round Function
2. Key Generation and Expansion

This will be followed by a brief explanation on Padding.

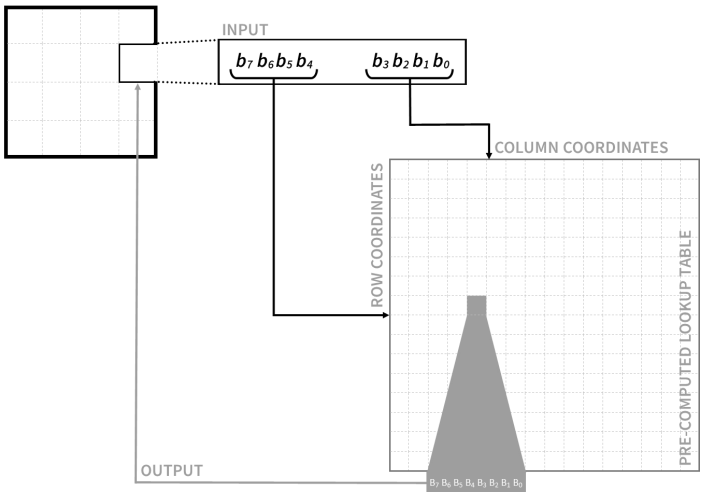
Round Function

The Round Function in AES is comprised of just 3 'sub-functions' that are applied every round except the last round where the 3^{rd} function `MixColumns` is not applied ∴ it is a permutation step and does not add to the security of the scheme since we're only shifting bits around. The functions are :

1. SubBytes
2. ShiftRows
3. MixColumns

SubBytes

This is a simple substitution step that relies on a pre-computed lookup table. The element that it substitutes acts gives the row and column coordinates. The table is a 16 ×16 hexadecimal table. The following diagram illustrates the process of substitution.



Each element is 8 bits in the main matrix. The first 4 bits (nibble) is the row coordinate and the last nibble is the column coordinate. This new element from the lookup table is put into the matrix.

> **How is the lookup table generated?**
> This lookup table is generated using polynomials in a finite field. Blah blah blah.....

Code Implementation

This SubBytes function is implemented through the class *SBox.java*. The S-Box for encryption is the `forwardSBox`. The real magic happens in the `performSubstitution()` function.

```
int row = Integer.parseInt(DecimalTo8BitBinary(number).substring(0, 4), 2);
```

Since we store all the values in an `int` matrix, when we accept the integer value that we want substituted we have to convert it into 8-bit binary. That is what the `DecimalTo8BitBinary()` function does (and it returns a `String`). Now out of the 8-bit binary `String` we have to take the first 4 bits and so `substring(0, 4)`. Now we have the 4 bits for the row coordinate but to use it to access the array, it must be an `int`. So we use the in-built function `parseInt()` from the `Integer` class. It converts the provided `String` into an `int`. The 2 at the end specifies that the `String` is of base 2 (i.e it's binary).

```
int column = Integer.parseInt(DecimalTo8BitBinary(number).substring(4, 8), 2);
```

The same analysis from the row coordinate applies. the only difference being that we take the last 4 bits so `substring(4, 8)`. This is then inputed in `forwardSBox` and that value is then returned.

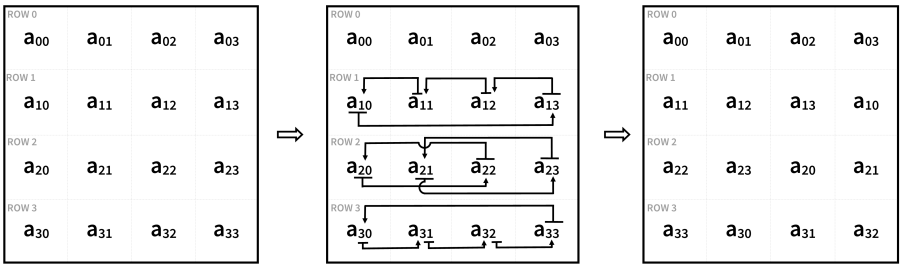
This is all coordinated in the *AESRoundFunction.java* class in the `subBytes` function.

```
for (int i = 0; i < matrix.length; i++)
  for (int j = 0; j < matrix[i].length; j++)
    matrix[i][j] = S.performSubstitution(matrix[i][j]);
```

This is simply itering through the whole matrix and substituting each element.

ShiftRows

This step is a permutation step that 'rotates' the rows of the matrix. Each row is shifted to the left and wrapped around by its index no.



As depicted above, Row 0 is not shifted at all. Row 1 is shifted to the left by 1. Row 2 is shifted to the left by 2 and Row 3 is shifted to the left by the 3. Any elements 'exceeding' the matrix are 'wrapped' around.

Code Implementation

This is done in the *AESRoundFunction.java* under the `shiftRows()` function. A temporary array `tempArray` is created to hold the shifted row before it is put back into the matrix.

```
for (int n = 0; n < 4; n++){...}
```

This outer loop itres through the 4 rows in the matrix. n represents the row index.

```
tempArray = new int[4];
```

The array declaration is put inside the loop because we need a new `null` array every row.

```
for (int i = 0; i < 4; i++)
  tempArray[i] = matrix[n][(i+n)%4];
```

Now we itre through all the elements in the row. Each time, the element is placed in the $(i+n)^{th}$ spot, where i is the element's index and n is the row index. The reader is encouraged to find out how this is true.

MixColumns

This is also a permuatation step and is the most complex of the three. Here, the column that is inputed is mulitplied by a 4×4 predefined matrix. The matrix multiplication is a polynomial multiplication over a finite

field. Each columnis treated as a polynomial $p(x) = k_{3a} \cdot x^3 + k_{2a} \cdot x^2 + k_{1a} \cdot x + k_{0a}$ where k_{ia} is the i^{th} element of the column a. This polynomial is an element in the field $\mathbb{GF}(2^8)$. This matrix multiplication is represented below :

$$\begin{bmatrix} k'_{0a} \\ k'_{1a} \\ k'_{2a} \\ k'_{3a} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} k_{0a} \\ k_{1a} \\ k_{2a} \\ k_{3a} \end{bmatrix}$$

where the $\cdot$ operation denotes multiplication (usual polynomial multiplication when between polynomials, and multiplication over $\mathbb{GF}(2^8)$ for the coefficients).

Code Implementation

This is a tricky one to write in code, however, several documents explain the code implementation without having to do 'actual' matrix multiiplication. The documents refer are Understanding AES Mix-Columns Transformation Calculation and the crypto.stackexchange post - How to solve MixColumns. In summary, these two articles provide a computation that can easily be translated into code that give us the same result as the matrix equation above. The computation is as follows :

To compute *02* $\cdot$ $\texttt{d4}_{16}$ ($\texttt{d4}_{16}$ is a random hexadecmal and has no specific purpose) :

1. Convert the hexadecimal to 8 bit binary $\therefore \texttt{d4}_{16} = 1101\ 0100_2$. Call this the intermediate
2. Left shift the bits of the intermediate by 1. Now intermediate $= 1010\ 1000_2$
3. XOR the intermediate with $0001\ 1011_2$ if the first bit was 1 **before** the shift, so intermediate $= 1010\ 1000_2 \oplus 0001\ 1011_2$
4. This is the result $(1011\ 0011_2)$

To compute *03* $\cdot$ $\texttt{d4}_{16}$:

1. Compute *02* $\cdot$ $\texttt{d4}_{16}$
2. XOR this result with $\texttt{d4}_{16}$ in binary

To compute *01* $\cdot$ $\texttt{d4}_{16}$: Leave it as is

These values are then XORed t ogether (in place of the addition that occurs in regular matrix multiplication). This computation is handled by the `dotProduct()` function in the *AESRoundFunction.java* class. The function takes 2 parameters, the first one being either *02* or *03* (the operator), the second being the value (the operand). Luckily for us, both XOR and Bit Shift is provided by Java's in-built operators.

```
case 2:
  result = b<<1>=256 ?(b<<1)^256:b<<1;
  result = b>=128?result^27:result;
```

In Java, the `a<<b` shifts `a` by `b` bits to the left. The ternary operator `condition?outcomePositive : outcomeNegative` is a miniaturized version of an if statement fit into 1 line. Since we have an 8-bit normalized binary string, if the left bit shift has a 1 in the 9^{th} place, we have to eliminate it. Any integer whose binary counterpart has a 1 in the 9^{th} place is ≥ 256. So we set the condition to check whether the integer is ≥ 256. If it is, the `^256` will eliminate the 1 in the position. This is assigned to result. (Note that `b` itself is untouched.). Now we check if the leading bit of the operand `b` is 1. This is easily tested by checking if the number is ≥ 128. The reader may check that this is true (it applies the same logic as the 256 condition). This takes care of the *02* case.

```
case 3:
  result = dotProduct(2,b)^b
```

For the *03* case, we simply compute the *02* case and then XOR with the operand `b`. Now this is tied togehter by the `mixColumns()` function. Each element of the new column k'_{ia} can be represented as an equation based on the matrix multiplication. So itering through each column of our 4×4 matrix, we compute the result

```
for (int i = 0; i < 4; i++) {
  matrix[0][i] = dotProduct(2,matrix[0][i]) ^ dotProduct(3,matrix[1][i]) ^ matrix[2][i] ^ matrix[3][i];
  matrix[1][i] = matrix[0][i] ^ dotProduct(2,matrix[1][i]) ^ dotProduct(3,matrix[2][i]) ^ matrix[3][i];
  matrix[2][i] = matrix[0][i] ^ matrix[1][i] ^ dotProduct(2,matrix[2][i]) ^ dotProduct(3,matrix[3][i]);
  matrix[3][i] = dotProduct(3,matrix[0][i]) ^ matrix[1][i] ^ matrix[2][i] ^ dotProduct(2,matrix[3][i]);
}//for loop - i
```

Key Generation and Expansion

Key Generation

In Rjindael, an **explicit** key generation is not given, because its is expected that the user feed it a high-entropy pseudorandom key of the necessary length. Consequently, (as a precaution) the user is not allowed to type in their own key for encryption. They can type in a 'seed' and a 'salt'.

A **seed** (in our context) is an input that is used to initialize a function which produces a unique pseudorandom key as output (i.e. there is a one-to-one correlation between seeds and keys when using this function). This seed is essential to generating a pseudo random key.

A **salt** is an additional piece of data that facilitates 'randomness' in key generation. (There exists an alternate definition with regards to hashing and passwords, but that is not so important in this context).[4]

Why must there be a salt?
Although not essential to key generation, it helps with ensuring that an attacker cannot easily guess our key if they somehow get a hold of our salt and had access to the same function we used to generate the key. Furthermore, since we use a PBKDF (Password Based Key Derivation Function), the Java class implementing this **insists** on having a salt (and for lack of a better alternative for generating key), we are stuck with having to use a salt.

How do you generate a salt?
We can simply get the salt from the user. Now, as a bonus feature, there's an option to generate a pseudorandom salt. This doesn't add SIGNIFICANTLY to the strength of the key. Its is simply to give the user more options. The salt is size normalized to 16 bytes. If the user leaves it blank or types in less than 16 characters, we generate pseudorandom hex characters until it is 16 bytes.

```
1  if (salt.length() < 16) {
2    Random rand = new Random();
3    while (16 - salt.length() > 0)
4      salt += Integer.toHexString(rand.nextInt(0XF));
```

This is the snippet of code where the pseudorandom salt is generated. `Random` is a class in the *java.util* package specially designed for random number generation and more. The loop itres until the length of the salt's length is 16. At every step we generate a random integer upto 15 (0XF is the hexadecimal representation of 15). This integer is then converted back into Hex and appended to the salt.

It is strongly advised that the reader treat the code implementation of the pseudorandom key generation as a black box that simply returns a pseudorandom key as output when fed a seed and a salt. So, kindly skip STEP1 of the constructor in the *Keys.java* class. Take for granted that after STEP1 we have a pseudorandom key.

[4] To understand the differences between these, refer to What are the differences between an encryption seed and salt?

But why should I? ...
Be satisfied treating it as a black box. This is for the readers who aren't satisfied with treating it as a black box abstraction and want to get their hands dirty with code. The following 11 lines is where the magic takes place :

```
1    //STEP 1 : GENERATE A PSEUDORANDOM KEY BASED ON THE USER'S INPUT
2    SecretKey key = null;
3    try {
4      KeySpec keyGen = new PBEKeySpec(seed.toCharArray(),salt.getBytes(),65536,algorithm);
5      key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1").generateSecret(keyGen);
6    }//try block
7    catch (NoSuchAlgorithmException | InvalidKeySpecException ex) {
8      System.out.println("ERROR");
9      System.exit(0);
10   }//catch block
11   initialKeyState = ByteArrayToHexadecimal(key.getEncoded());
```

A line by line analysis follows :

```
2    SecretKey key = null;
```

This initializes the *SecretKey* interface from the *javax.crypto* package with a null. We cannot omit the `= null`, since the compiler returns a 'not initialized' error.

```
4    KeySpec keyGen = new PBEKeySpec(seed.toCharArray(),salt.getBytes(),65536,algorithm);
```

KeySpec interface is implemented using the *PBEKeySpec* class. This is where all the necessary data is put into position. The constructor takes the char array of the seed (*seed.toCharArray*), the byte array of the salt (*salt.getBytes()*), an iteration count (*65536*), the key size it needs to generate (*algorithm*) in order.

```
5    key = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1").generateSecret(keyGen);
```

This is where our pseudorandom key is born. The *SecretKeyFactory* implements the *SecretKey* interface using a PBKDF as mentioned earlier. This class access the data from the *KeySpec* object.

```
3    try(...){...}

7    catch(...){...}
```

The try-catch block is necessary as the above 2 lines of code may throw a *NoSuchAlgorithmException* or a *InvalidKeySpecException*. Since there is no `throws` in the function declaration, we neatly wrap these exceptions into one block of code (a try-catch block).

```
11   initialKeyState = ByteArrayToHexadecimal(key.getEncoded());
```

The pseudorandom key that was generated exists as a byte array. This is printable but not feasible. So we convert this byte array into a hexadecimal string. Since a hexadecimal character is a nibble (4 bits ∴ half a byte) the length of the hexadecimal string will be twice that of the byte array. So, a 128 bit key (16 bytes) will have 32 hexadecimal characters, a 192 bit key (24 bytes) will have 48 hexadecimal characters and so on.

These 11 lines of code achieve the following for us :-

1. Generate a **unique, pseudorandom key** of desired length based on a seed.
2. Used a salt to enhance randomness of the generated key.
3. Got a copy of the key in an 'outputable' format.

In Rjindael, the key is fit into a 4×4, 4×6 or 4×8 matrix for the 128, 192 or 256 bit key variants respectively-

```
1  private static int[][] currentKeyMatrix;
2  ...
3  case 128:
4    currentKeyMatrix = new int[4][4];
5    ...
6
7  case 192:
8    currentKeyMatrix = new int[4][6];
9    ...
10
11 case 256:
12   currentKeyMatrix = new int[4][8];
```

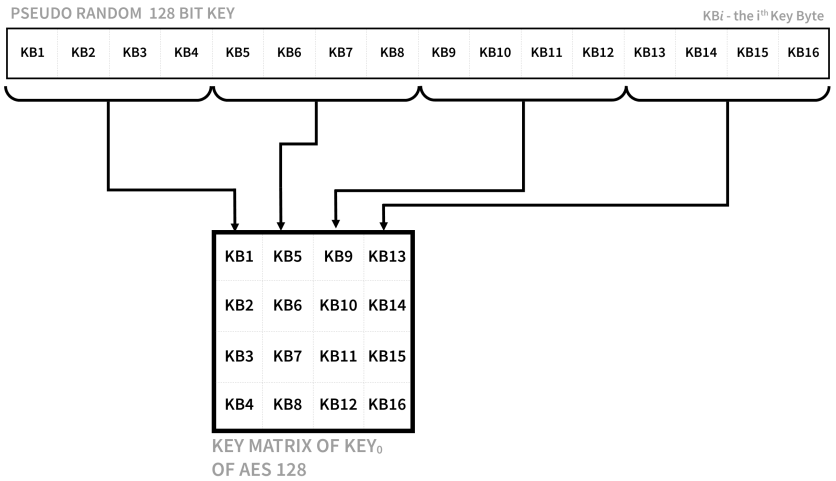
The key matrix (`currentKeyMatrix`) is where all the action takes place. The `initialKeyState` global variable only stores the initially generated pseudorandom key. Given that we have set up the key matrix, we must now put our generated key in the matrix **column-wise**. The reader might have also noted that out matrix is an `int` matrix while our key is hexadecimal (`String`), so we must convert from hex to int the respective integer before putting it in the matrix.`code`

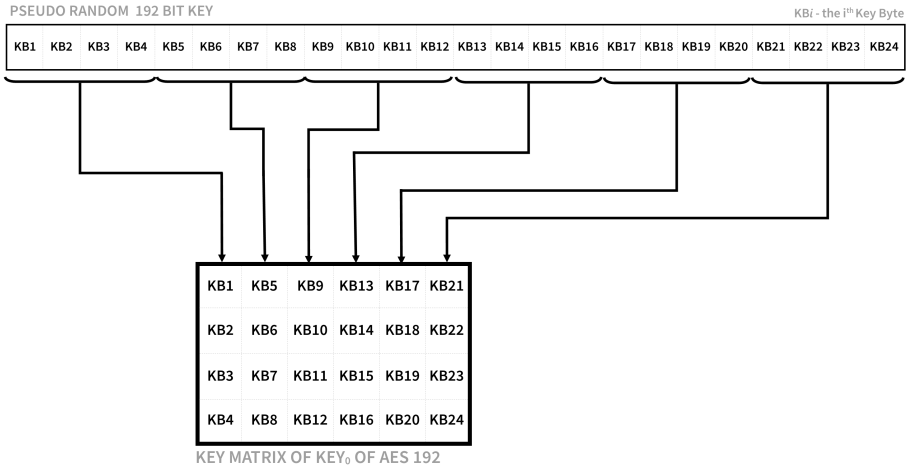
Why an integer matrix?

The reasons behind making the key matrix (and the main matrix) integers are as follows:

1. Size constraints (`int` matrices occupy much less space that `String` matrices).
2. XOR is a big part of AES. Since Java provides a bitwise operator for XORing integers (^), it would simplify our work substantially if we worked with integers (since we don't have to define a special function just to XOR.)

However, we only place the key in the matrix when the *AES.java* class calls for the k_0 (initial key). As displayed in the AES Overview, the k_0 matrix is XORed before the rounds start. This is when the key is put in the `currentKeyMatrix`. This is STEP1 of the `generateKeyMatrix()` function. It substrings 2 hexadecimals characters at a time, converts it into it its respective decimal for, and assigns it to its respective matrix position. An overview of this process (albeit omitting the conversion from Hex to Decimal) is given for 128 and 256 key variants.





Key Expansion

Now we elaborate the key expansion which unlike Key Generation is part of the Rjindael algorithm. This will be done in 2 phases simultaneously. First the key expansion scheme is explained and illustrated as it is given in the NIST Publication. After this, the second phases explains the code implementation of it.

What is the criteria for the Key Expansion :-

1. **Efficiency**
   a. Working Memory - Memory utilized should be minimized
   b. Performance - It should have high performance on multiple processors
2. **Symmetry elimination** - Round counstants will be used to eliminate symmetries.
3. **Diffusion** - It should have an efficient diffusion of cipher key differences into the expanded key.
4. **Non-linearity** - It should exhibit enough non-linearity to prohibit the full determination of differences in the expanded key from cipher key differences only

For more criteria refer to the "The Design of Rjindael" Section 5.1 by Joan Daemen and Vincent Rijmen.

RCON Values

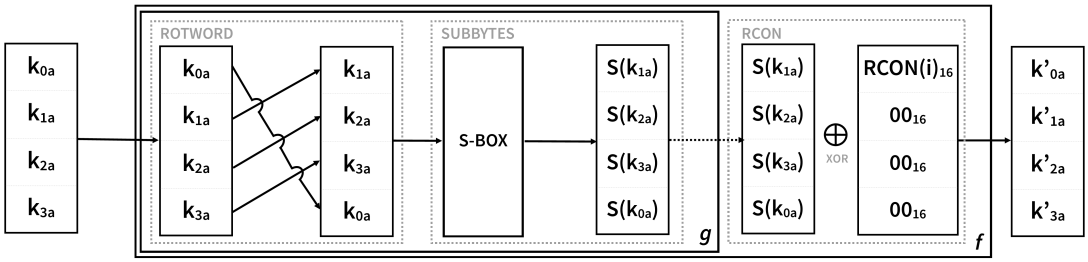
These RCON values are used in ***f*** functions to break symmetry. It is stored as a table. Depending on the round no., a certain value from the table is XORed with the first element of the column. Use this link to find the RCON values : AES Key Schedule

S-Box

This is a precomputed lookup table that is used in the Key Generation and in the SubBytes step in Round Function. The procedure for substituting a value in th lookup table is specified in the SubBytes step in the Round Function.

Key Generation Functions

There are 2 functions that are used in the key expansion for Rjindael.The ***g*** function and the ***f*** function. These functions take columns as input and return columns as output. The functions are described below. The ***f*** function is an extension of the ***g*** function and utilizes the RCON values. $RCON(i)_{16}$ means that the values are in hexadecimal and i is the index no for the specific RCON value needed for that round.



Code Implementation of *f* and *g* Functions

This is achieve by the private functions `functionG` and `functionF`.

```
// STEP 1: ROTWORD
int[] resultantKeyColumn = new int[4];
resultantKeyColumn[3] = column[0];
for (int i = 1; i < 4; i++)
  resultantKeyColumn[i - 1] = column[i];
```

ROTWORD: We first start with defining a temporary array location `resultantKeyColumn` that acts as a workstation and will be returned. Now the last element of this array is the first element of the column. The `for`-loop itres through the remaining elements and places them accoriding to the diagram above.

```
// STEP 2 : SUBBYTES
SBox S = new SBox('e');
for (int i = 0; i < 4; i++)
  resultantKeyColumn[i] = S.performSubstitution(resultantKeyColumn[i]);
```

SUBBYTES: This works exactluy like the SubBytes from the Round Function.
This completes the ***g*** function. The ***f*** function requires the XORing of the RCON values.

```
resultantKeyColumn = functionG(resultantKeyColumn);
// STEP 3 : RCON
resultantKeyColumn[0] ^= RCON[roundNo - 1];
```

RCON: The ***f*** function is an extension of ***g***, so we can assign `resultantKeyColumn` to the output of the ***g*** function. Now we XOR only the first element with the RCON value based on the round number.

Henceforth we will call each column of a matrix a 'word'. Quite obviously, each word is 32 bytes and is written as KW_i, which represents the i^{th} word of the key expansion. The functions $\boldsymbol{g}$ and $\boldsymbol{f}$ takes 1 word as input and gives 1 word as output. Before we get to the key expansion, we need to place the generated key (`intialKeyState` - stored as a hex string) into the `currentKeyMatrix`. We can do this simultaneously with returning k_0 matrix. This is a universal step and thus does not come under the switch case for the different variants. This is listed as STEP1 of the code.

```
// STEP 1 : TAKE CARE OF THE KEY0 STATE FOR ALL POSSIBLE KEY LENGTHS
if (roundNo == 0) {
  initialKeyState += " ";
  for (int i = 0; i < initialKeyState.length() - 1; i += 2) {
    String extract = initialKeyState.substring(i, i + 2);
    currentKeyMatrix[i / 2 % 4][i / 8] = Integer.parseInt(extract, 16);
  }// for loop - i
  for (int i = 0; i < 4; i++)
    System.arraycopy(currentKeyMatrix[i], 0, outputMatrix[i], 0, 4);
  return outputMatrix;
} // if statement - INITIAL ROUND
```

As shown above, `initialKeyState` is appended with a " ". The reason for this will become apparent in the next line. We itre through the hex string incremeneting by 2 (Why 2? Remember it takes 2 hex characters to form a byte). These 2 'extracted' hex characters are placed into the temporary variable `extract`.

```
currentKeyMatrix[i / 2 % 4][i / 8] = Integer.parseInt(extract, 16);
```

This is a formula for placing the byte in the matrix column wise as depicted in the diagrams. This is a universally applicable and fills up 4×4. 4×6, 4×8.

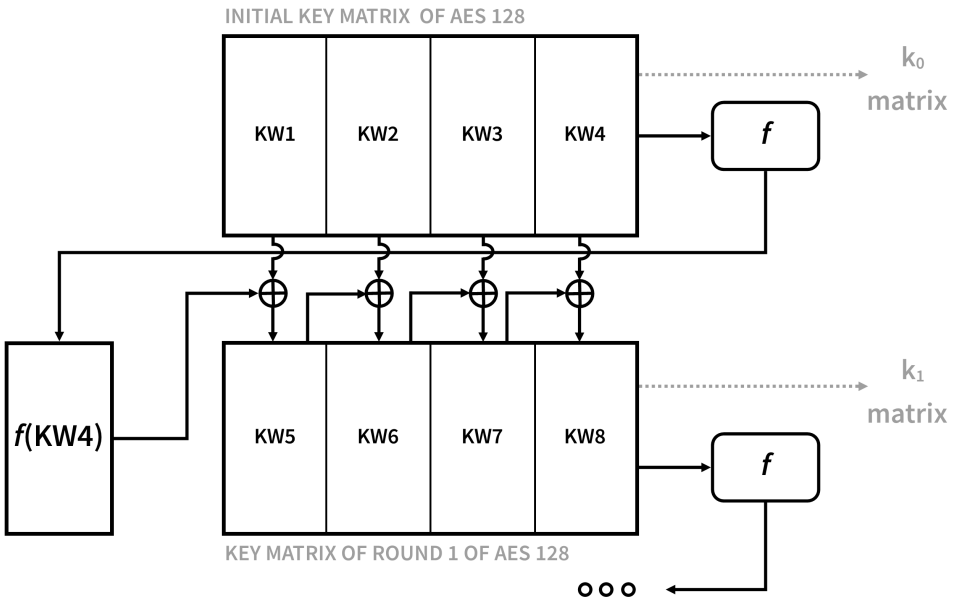
```
for (int i = 0; i < 4; i++)
  System.arraycopy(currentKeyMatrix[i], 0, outputMatrix[i], 0, 4);
return outputMatrix;
```

This last `for` loop is to put the first 4 columns of the `currentKeyMatrix` into `outputMatrix`.

The key expansions for the the 3 variants are given below. These are STEP2 of the `generateKeyMatrix()` function.

128 Bit Key

AES128 only has 10 rounds and every round a different RCON value is used every round. A total of 10 RCON values are used. The words of the initial key matrix (k_0 matrix) are KW_1, KW_2, KW_3, KW_4. The diagram explaining illustrates the key generation for the k_1 matrix. This is repeated until the k_{10} matrix.:



In the k_0 matrix, we take the last word, pass it through the $\boldsymbol{f}$ function and XOR the output of that function with the first word of the k_0 matrix. This resultant word is the first word of the k_1 matrix and is named KW_5. KW_5 is then XORed with KW_2 to give KW_7 and this continues until we complete the k_1 matrix.

Code Implementation

In the `generateKeyMatrix()` of the *Keys.java* class, this is STEP2.1. The `int` array `column` is created so that it can be the temporary workstation which the function acts on so that the `currentKeyMatrix`'s state is preserved. So, in order we perform the computation.

```
1     for (int i = 0; i < 4; i++)
2       column[i] = currentKeyMatrix[i][3];
```

First, the last column of the matrix is copied into the `column` array.

```
1     column = functionF(column, roundNo);
```

This is passed into the ***f*** function and the result is stored in `column`.

```
1     for (int i = 0; i < 4; i++)
2       currentKeyMatrix[i][0] ^= column[i];
```

This column is then XORed with the first column of the matrix and the result is the first column of the `currentKeyMatrix`. This is why the `^=` operator is used.

```
1     for (int i = 1; i <= 3; i++)
2       for (int j = 0; j < 4; j++)
3         currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
```

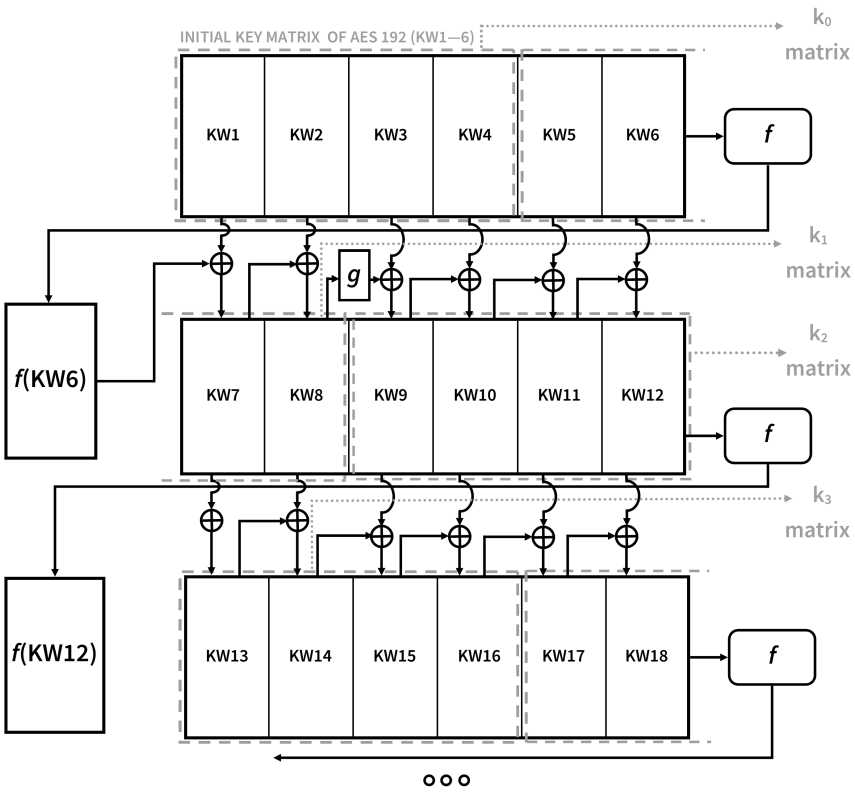
The first column will now generate the second by XOR. The second generates the third and so on. The outer loop is to itre through the second to the fourth columns. The inner loop is to itre through each element in the column.

```
1     for (int i = 0; i < 4; i++)
2       System.arraycopy(currentKeyMatrix[i], 0, outputMatrix[i], 0, 4);
```

Now, the `outputMatrix` is a size-normalized matrix (4×4) that acts as a common output 'valve'. Depsite the size of the `currentKeyMatrix`, the outputed matrix that will be XORed with the main matrix is **ALWAYS** a 4×4 matrix. Hence this does not come under STEP2.1, rather at the end of the function because it serves as a common output. The `System.arrayCopy(...)` is a default array copying function provided by Java that takes the following parameters in order : the array to be copied **from**, the start position in that array, the array to be copied **to**, the start position of that array, the no. of elements that must be copied (with the start position inclusive).

192 Bit Key

AES192 has 12 rounds and the initial key matrix is comprised of $KW_1, KW_2, KW_3, KW_4, KW_5, KW_6$. A different RCON value is **NOT** used every round. A total of 8 RCON values are used. To understand key generation of AES192 better, we split them up into cases. A diagram illustrates the generation of the first 4 keys. This is then generalized in the cases that follow :



The 3 different types of cases are listed below.

(i) **case 0 :**

This is the case which takes the first 4 columns as the key matrix. If we continue the diagram, it will be evident that the $k_0, k_3, k_6, k_9, k_{12}$ matrices are generated this way. To generate these matrices, an **RCON value is needed** (other than for k_0). A common pattern for identifying this case is that it outputs all keys whose round number is 0 mod 3.

Code Implementation

To code for this case, we implement a switch case that takes the round number and computes its value modulo 3. A line by line analysis of STEP 2.2.0 follows

```
case 0:
  // STEP 2.2.0.1 : ASSIGN THE LAST COLUMN (COLUMN 5) TO A TEMPORARY VARIABLE.
  for (int i = 0; i < 4; i++)
    column[i] = currentKeyMatrix[i][5];
```

A temporary variable `column` stores the last column of the matrix.

```
    // STEP 2.2.0.2 : GENERATE THE FIRST COLUMN OF THE NEXT 4X6 KEY MATRIX
    column = functionF(column, (int) (Math.floorDiv(roundNo, 2) + Math.floorDiv(roundNo,
     7) + Math.floorDiv(roundNo, 10)));
    for (int i = 0; i < 4; i++)
      currentKeyMatrix[i][0] ^= column[i];
```

The column is then passed through the $\boldsymbol{f}$ function. The RCON number for that is a calculated by an ad hoc formula that I devised. The result of this is stored in column. The 0^{th} column of the next key matrix is generated by XORing the `column` array with the 0th column of `currentKeyMatrix`. This gives us the 0^{th} column of the next key matrix.

```
    // STEP 2.2.0.3 : USE THE 0th COLUMN TO GENERATE COLUMNS THE NEXT 3 COLUMNS
    for (int i = 1; i <= 3; i++)
      for (int j = 0; j < 4; j++)
        currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
    break;
```

The 0th column of the next key matrix is used to generate the next 3 columns. The common output handles its placement into `outputMatrix`.

(ii) **case 1 :**

This is the case which takes the last 2 columns of the matrix and generate the next two columns to form the key matrix. If we continue the diagram, it will be evident that the k_1, k_4, k_7, k_10 matrices are generated this way. To generate these matrices, an **RCON value is needed** (other than for k_0). A common pattern for identifying this case is that it outputs all keys whose round number is 1 mod 3.

Code Implementation

This is case 1 of the same switch case. A line by line analysis of STEP 2.2.1 follows

```
// STEP 2.2.1.1 : GENERATE THE COLUMNS 4 & 5
if (roundNo != 1) {
  for (int i = 4; i < 6; i++)
    for (int j = 0; j < 4; j++)
      currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
}// if statement - EXCEPT ROUND 1
```

In the k_1 matrix we don't have to generate the 4^{th} and 5^{th} columns because its already in there due to STEP1. So if the round number is 1 we can skip this step. Otherwise, we perform the operations depicted above.

```
// STEP 2.2.1.2 : ASSIGN THE LAST COLUMN TO A TEMPORARY VARIABLE.
for (int i = 0; i < 4; i++)
  column[i] = currentKeyMatrix[i][5];
// STEP 2.2.1.3 : GENERATE THE FIRST COLUMN OF THE NEXT KEY 4X6 MATRIX
column = functionF(column, (int) (Math.floorDiv(roundNo, 2) + Math.floorDiv(roundNo,
  7) + Math.floorDiv(roundNo, 10)));
for (int i = 0; i < 4; i++)
  currentKeyMatrix[i][0] ^= column[i];
```

The last column is stored in the temporary variable `column` and then passed into the ***f*** function. Then, the 0^{th} column of the `currentKeyMatrix` is XORed with `column`.

```
 // STEP 2.2.1.3 : USE THE 0th COLUMN TO GENERATE COLUMN 1
  for (int j = 0; j < 4; j++)
    currentKeyMatrix[j][1] ^= currentKeyMatrix[j][0];
```

Lastly, to get the final column of our key matrix (the 1^{st} of `currentKeyMatrix`) we XOR with the 0^{th} column.

```
// STEP 2.2.1.4 : ISOLATE THE COLUMNS 4,5,0,1 INTO A 4x4 PART TO RETURN
for (int i = 4; i <= 7; i++)
  for (int j = 0; j < 4; j++)
    outputMatrix[j][i] = currentKeyMatrix[j][i % 6];
return outputMatrix;
```

Now the common output cannot handle this case because it was designed to extract columns 0,1,2,3 and place them into `outputMatrix`. We have to take columns 4,5,0,1 and place them into `outputMatrix`. So our `for` loop itres from 4 to 7 inclusive and the column of `currentKeyMatrix` is held by i mod 6. So as we take columns 4 & 5, we get columns 4 & 5, but as soon as we go to 6 & 7, the modulo corrects it to 0 & 1 thus giving us the required columns.

(iii) **case 2 :**

This is the case which takes the last 4 columns as the key matrix. If we continue the diagram, it will be evident that the k_2, k_5, k_8, k_{11} matrices are generated this way. To generate these matrices, an **RCON value is NOT needed** (other than for k_0). A common pattern for identifying this case is that it outputs all keys whose round number is 2 mod 3.

Code Implementation

This is case 2 of the same switch case. A line by line analysis of STEP 2.2.2 follows

```
// STEP 2.2.2.1 : ASSIGN COLUMN 1 TO A TEMPORARY VARIABLE.
for (int i = 0; i < 4; i++)
  column[i] = currentKeyMatrix[i][3];
// STEP 2.2.2.2 : GENERATE COLUMN 2 USING COLUMN 1
column = functionG(column);
for (int i = 0; i < 4; i++)
  currentKeyMatrix[i][4] ^= column[i];
```

Column 1 of `currentKeyMatrix` is assigned to the temporary variable `column`. This is then passed through the g function as depicted above. The result is XORed with column 2 of `currentKeyMatrix`.

```
// STEP 2.2.2.3 : USE COLUMN 2 TO GENERATE COLUMNS 3,4,5
for (int i = 3; i <= 5; i++)
  for (int j = 0; j < 4; j++)
    currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
```

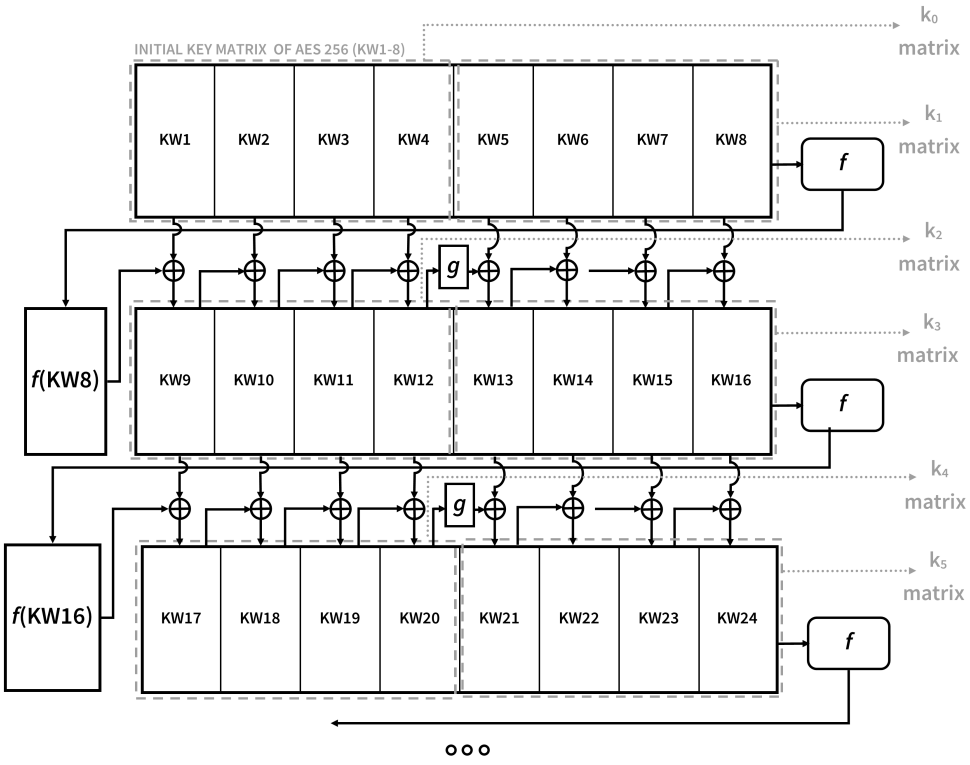
Now column 2 is used to generate column 3,4 & 5.

```
// STEP 2.2.2.4 : ISOLATE COLUMNS 2,3,4,5 INTO A 4x4 PART
for (int i = 2; i <= 5; i++)
  for (int j = 0; j < 4; j++)
    outputMatrix[j][i - 2] = currentKeyMatrix[j][i];
return outputMatrix;
```

The common output cannot be used because it handles the 4×4 for columns 0,1,2,3 however we need a 4×4 for columns 2,3,4,5. This piece of code handles the output.

256 Bit Key

AES256 has 14 rounds and the initial key matrix is comprised of $KW_1, KW_2, KW_3, KW_4, KW_5, KW_6$. A different RCON value is **NOT** used every round. A total of 7 RCON values are used. Just like AES192, we split them up into cases. A diagram illustrates the generation of the first 4 keys. This is then generalized in the cases that follow :



The 2 different types of cases are listed below.

(i) **case 0 :**

This is the case which takes the first 4 columns as the key matrix. If we continue the diagram, it will be evident that the $k_0, k_2, k_4, k_6, k_8, k_{10}, k_{12}, k_{14}$ matrices are generated this way. To generate these matrices, an **RCON value is needed** (other than for k_0). A common pattern for identifying this case is that it outputs all keys whose round number is 0 mod 2.

Code Implementation

To code for this case, we implement a switch case that takes the round number and computes its value modulo 3. A line by line analysis of STEP 2.3.0 follows

```
// STEP 2.3.0.1 : ASSIGN THE LAST COLUMN (COLUMN 7) TO A TEMPORARY VARIABLE.
for (int i = 0; i < 4; i++)
  column[i] = currentKeyMatrix[i][7];
```

The last column of the the matrix is stored in a temporary variable `column`.

```
// STEP 2.3.0.2 : GENERATE THE FIRST COLUMN OF THE NEXT KEY MATRIX
column = functionF(column, roundNo / 2 - 1);
for (int i = 0; i < 4; i++)
  currentKeyMatrix[i][0] ^= column[i];
```

Then `column` is passed through the $\boldsymbol{f}$ function and XORed with the column 0 of `currentKeyMatrix`.

```
// STEP 2.3.0.3 : USE THE FIRST COLUMN TO GENERATE THE NEXT COLUMNS
for (int i = 1; i <= 3; i++)
  for (int j = 0; j < 4; j++)
    currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
break;
```

Now column 0 is used to generate columns 1,2 & 3 as depicted above. The common output handles its placement into `outputMatrix`.

(ii) **case 1 :**

This is the case which takes the last 4 columns as the key matrix. If we continue the diagram, it will be evident that the $k_1, k_3, k_5, k_7, k_9, k_{11}, k_{13}$ matrices are generated this way. To generate these matrices, an **RCON value is NOT needed** (other than for k_0). A common pattern for identifying this case is that it outputs all keys whose round number is $1 \mod 2$.

This is case 1 of the same switch case. A line by line analysis of STEP 2.3.1 follows

```
// STEP 2.3.1.1 : ASSIGN COLUMN 3 TO A TEMPORARY VARIABLE.
for (int i = 0; i < 4; i++)
  column[i] = currentKeyMatrix[i][3];
```

First column 3 of `currentKeyMatrix` is assigned to the temporary variable `column`.

```
// STEP 2.3.1.2 : GENERATE COLUMN 4 USING COLUMN 3.
if (roundNo != 1) {
  column = functionG(column);
  for (int i = 0; i < 4; i++)
    currentKeyMatrix[i][4] ^= column[i];

  // STEP 2.3.1.3 : USE THE COLUMN 4 TO GENERATE COLUMNS 5,6,7
  for (int i = 5; i <= 7; i++)
    for (int j = 0; j < 4; j++)
      currentKeyMatrix[j][i] ^= currentKeyMatrix[j][i - 1];
}
```

Now, we need to generate the column 4 of `currentKeyMatrix` using the output of the $\boldsymbol{g}$ function as depicted above. However, this is not done for k_1 because the key words for k_1 are already present when we placed `initialKeyState` in `currentKeyMatrix`. The standard procedure is followed. Pass `column` through the $\boldsymbol{g}$ function. XOR the result with column 4. Use column 4 to generate columns 5, 6, 7 sequentially as depicted above.

```
// STEP 2.3.1.4 : ISOLATE THE COLUMNS 4,5,6,7 INTO A 4x4 PART TO RETURN
for (int i = 4; i <= 7; i++)
  for (int j = 0; j < 4; j++)
    outputMatrix[j][i - 4] = currentKeyMatrix[j][i];
return outputMatrix;
```

The common output cannot be used because it handles the 4×4 for columns 0,1,2,3 however we need a 4×4 for columns 4,5,6,7. This piece of code handles the output.

This completes the long and meticulous description of the encryption and decryption algorithms of Rjindael (AES). What follows is a detour into the mathematical realm of block ciphers and securrity. These concpets are not crucial to learning AES but it is a useful corollary to know for future reference.

Semantic Security

We shall proceed to define Semantic Security of a block cipher, followed by an aside on Chosen Plaintext Attacks. To make this notion formal we leverage our understanding of Advantage from 'Security of PRGs' in One Time Pad.[5] Furthermore, every cipher $(\mathbb{E}, \mathbb{D})$ hereafter in this documentation is going to refered to a pseudorandom function $\mathbb{F}$ and will use the terms cipher and PRF interchangeably. Do not bother too much about what this means, the important thing to understand is that when we say 'we query $\mathbb{F}$', we simply request the encryption

[5] If the reader is not familiar with this topic, it is highly recommended that they refer back to the OTP documentation before proceeding.

or decryption of a provided input depending on the mode it runs on.

Security Experiment

Let us begin by defining an experiment. We have 2 parties in this experiment - ATK(attacker) and MHN(machine). MHN is simply an AES engine. It does not verify the user or question the user - it function is to receive a plaintext, randomly pick a key and return the cipher text. Now we will define ATK's 'abilities'.

What can ATK do?

He can query MHN with any messages he likes (so long as they belong in the message space).

What is ATK's goal?

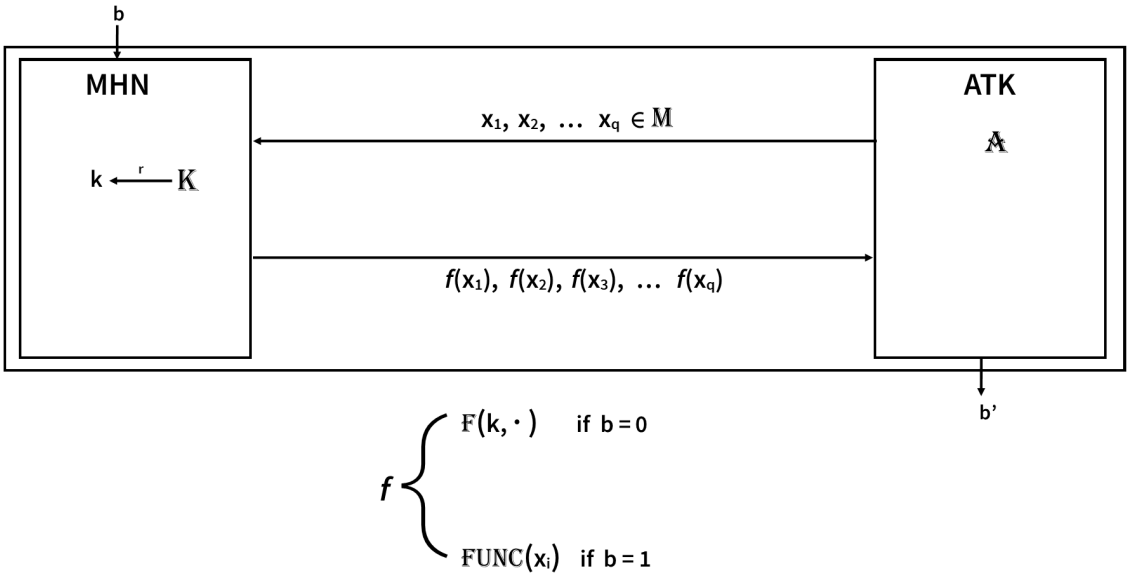
His goal is to break the encryption scheme by determining what message gets encrypted to what ciphertext. If he knows the ciphertext to a corresponding message, everyone implementing this encryption scheme is at risk.[6]

Let b be a variable. I flip a coin and if I get heads I set the value of b to 0, if I get tails I set the value of b to 1. In the experiment, ATK can give MHN up to q different queries. Each query must belong to the message space. MHN will respond differently based on the which experiment it is in which is determined by the value of b.

If $\mathbb{EXP}(0)$, MHN will query $\mathbb{F}$ for the encryption of the inputs.

If $\mathbb{EXP}(1)$, MHN will choose any function $\mathbb{FUNC}$ that will map the input to any **random** element of the ciphertext space (i.e. $\mathbb{FUNC}(m) = c$ where $c \xrightarrow{r} \mathbb{C}$).

The core idea behind establishing security in this experiment is that ATK should not find out which experiment he is in with high probability.



ATK will then use some statistical test $\mathbb{A}$ to determine whether the output he got was from a random function or our PRF. This statistical test's output is assigned to b'. Intuitively, if $b' = b$ in all the times we perform the experiment then we say that the ATK using $\mathbb{A}$ **broke** our PRF $\mathbb{F}$.

Security Definition

The security of our PRF $\mathbb{F}$ is now defined as :

$$\mathbb{ADV}_{PRF}(\mathbb{A}, \mathbb{F}) = \left| P\Big[\mathbb{EXP}(0) = 1\Big] \; - \; P\Big[\mathbb{EXP}(1) = 1\Big] \right|$$

PRF is secure if $\forall$ 'efficient' tests $\mathbb{A}$, $\mathbb{ADV}_{PRF}(\mathbb{A}, \mathbb{F}) <$ negligible

Alternatively, it can also be defined it terms of computational indistinguishability :

$$\{\mathbb{F}(k, \cdot) \mid k \xrightarrow{r} \mathbb{K}\} \overset{\text{poly}}{\approx} \text{uniform}(\mathbb{C})$$

What do these mean?

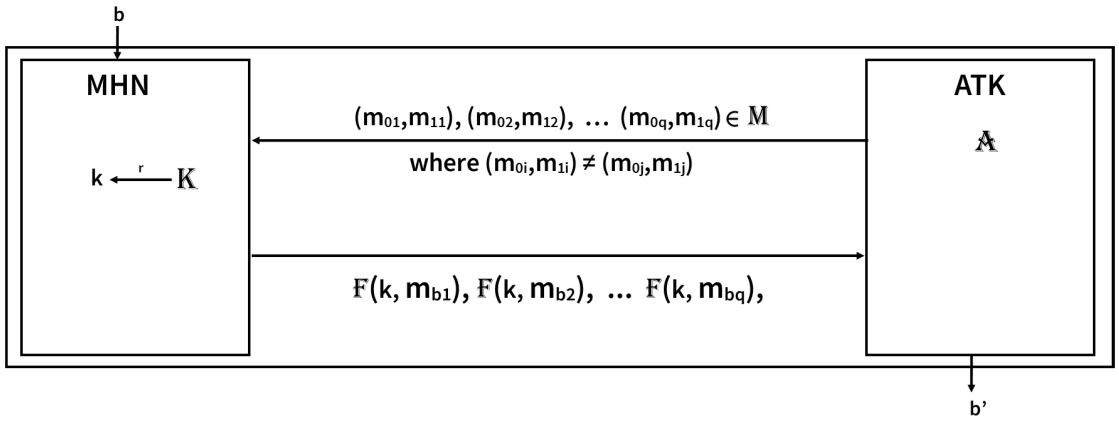
Both these security theorems mean the same thing, they are simply two different methods of writing it. They are 2 different ways of mathematically describing how a PRF $\mathbb{F}$'s output is **indistinguishable** from that of just picking some random ciphertext. So for any ATK, they don't know if they got some arbitrarily chosen ciphertext or the encryption of his message.

[6] We will see why this is true later.

An Apt Analogy
Suppose you had a blindfold on and were given 2 sets of objects. One is of them is a real tree bark and the other is some synthetic plastic one. Your job is to differentiate them solely based on touch alone. You cannot alter its properties in any way (for e.g., set both on fire and check which turns to ash). Based on a coin flip, you will be provided either with the natural tree bark or the synthetic one.
Consider yourself analogous ATK and your sense of touch and your brain analogous to the statistical test $\mathbb{A}$. Consider the machine synthesizing these artificial tree barks analogous to the secure PRF $\mathbb{F}$. Its job is to emulate the natural tree bark as much as possible and consequently fool you into misidentifying. This is analogous to the PRF $\mathbb{F}$ producing ciphertext(s) (synthetic bark(s)) indistinguishable from any random ciphertext (natural bark). The security theorems put this 'tree bark identification' into more mathematically formal notation.

Chosen Plaintext Attack

Having set up the necessary security parameter requisites we can go on to exhibit a chosen plaintext attack. First we need to alter the experiment for this to be demonstrated. In this experiment we will be querying a secure PRF $\mathbb{F}$, so there is no need for the randomizing function from $\mathbb{EXP}$. Instead we allow ATK to submit q distinct message pair queries. Based on the value of b, we return either the 0^{th} element or the 1^{st} element (i.e. either m_{0i} or m_{1i} for the i^{th} message pair).
ATK upon receiving these must decide using some algorithm $\mathbb{A}$ whether he received the encryption of the 0^{th} element or the 1^{st} element. He will output this into b'.



Will our security parameters hold? They **should**, since we chose a secure PRF so ATK cannot learn the ciphertext of any message. But ATK has a trick up his sleeve. Take a minute or so to ponder on what this could be before reading on.
ATK's Clever Trick
ATK will set both elements of the first message pair to some M_0, thus $(m_{01},\ m_{11}) = (M_0, M_0)$. Next he will set the 0^{th} element of the second message pair to the same M_0 and the the first element to another message M_1, so $(m_{02},\ m_{12}) = (M_0, M_1)$.

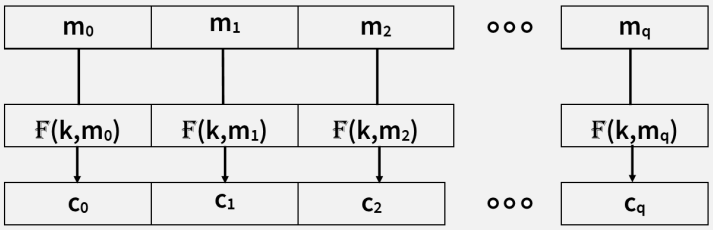
He will query MHN with **only** these 2 message pairs. Based on the value of b, MHN will give the output of 0^{th} element or the 1^{st} element. In the first message pair, since the elements are the same, ATK learns the ciphertext of M_0 which is $\mathbb{F}(k, M_0)$. MHN next outputs the result of the second message pair.
If $b = 0$, MHN outputs $\mathbb{F}(k, M_0)$, else it outputs $\mathbb{F}(k, M_1)$. Once ATK receives this, he can compare it to the ciphertext of the first message pair. If the ciphertexts are the same, he will know that he got the encryption of the zeroth element and will set $b' = 0$. Else he will set $b' = 1$. Thus,

$$\mathbb{ADV}_{PRF}(\mathbb{A}, \mathbb{F}) = 1$$

This is the best possible advantage. It means that **ATK broke our PRF** $\mathbb{F}$. Even though our PRF is secure, it cannot hold up against Chosen Plaintext Attacks. It must be apparent that **we cannot use a determinisitic encryption algorithm** $\mathbb{E}$. The cipher must output ciphertexts in such a way that no two ciphertexts of the same message are the same with very high probability (i.e given many ciphertexts of the same message, we should not find any repetitions 'easily'). Below is a small divergence into a real-world problem with using a CPA-Insecure Scheme.

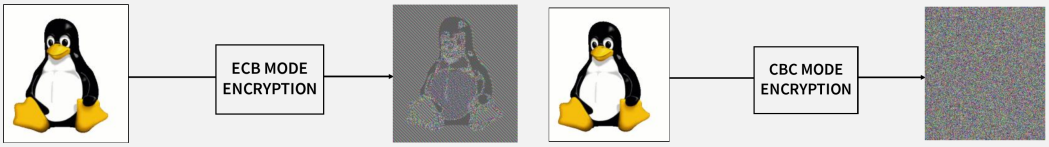
FAULTS IN A CPA-INSECURE SCHEME

This is not an 'attack' on an implementation of AES, rather a reminder for the reader that a deterministic encryption scheme cannot be secure. We will look at this through a visual example. The Electronic Code Book (ECB) mode of encryption is a multiple block encryption mode using a block cipher. The encryption is quite straightforward and is described below. Notice that ECB is entirely deterministic and as shown before will be vulnerable to a CPA. Let us say that we use the AES block cipher as the PRF $\mathbb{F}$ in this example.



.

Immediately, it becomes apparent that if $m_i = m_j$ then $c_i = c_j$ when $i \neq j$. So if 2 message blocks are the same, the cipher texts are the same. But what information does this leak? Below are 2 encryptions of an image using ECB (a CPA insecure scheme) and using CBC (a CPA secure scheme).[a]



This should make it clear that a **DETERMINISTIC ENCRYPTION SCHEME SHOULD NEVER BE USED IN PRACTICE.**

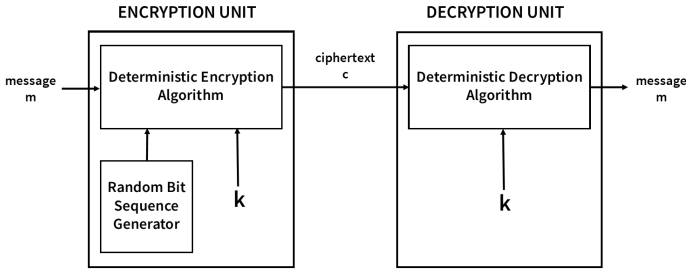
[a]This example is borrowed from Modes of Operation

CPA-Secure Methods

For guaranteeing message security, a CPA-Secure method must be used. How do we build such a scheme?

Randomized Encryption

This method is one where the encryption algorithm $\mathbb{E}$ is randomized but the decryption algorithm $\mathbb{D}$ is purely deterministic. So the encryption function can map a message onto any set of ciphertexts $\mathbb{E}(m) = \{c_1, c_2 \dots c_n\}$ where any ciphertext c_i has a certain probability of being the output. However the decryption algorithm will map a cipher text to its corresponding message $\mathbb{D}(c) = m$. There are several ingenious ways to achieve randomized encryption. They are not specified here but the interested reader can learn more about such schemes from the Further Reading section. The overall outline for a randomized encryption process is given below.



Nonce Based Encryption

Nonce-Based Encryption uses a unique non-repeating value (nonce) to encrypt the data. The nonce need not be random - the only constraint is that the (key, nonce) pair be used once and only once. Now our encryption

and decryption functions have alternate definitions. The only addition is that of a nonce space $\mathbb{N}$.

$$\mathbb{E} : \mathbb{K} \times \mathbb{N} \times \mathbb{M} \to \mathbb{C}$$

$$\mathbb{D} : \mathbb{K} \times \mathbb{N} \times \mathbb{C} \to \mathbb{M}$$

Building Nonce Based Semantically Secure Schemes
In order to build a nonce-based encryption scheme, we need a secure method to come up with a nonce. Below we list some methods for nonce generation.

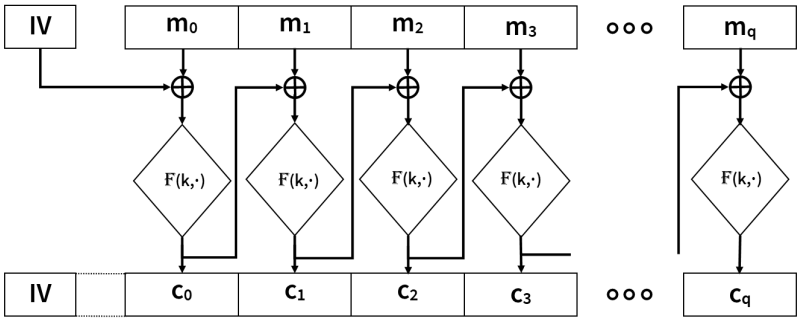
1. **Packet Counter**
   The nonce would be the counter for every packet encrypted. If the 'encryptor' and the 'decryptor' keep the same state at the start of the end-to-end transmission, then there would be no need to send the nonce in the packet since the nonce would be in the state of the machine.
2. **Random Nonce**
   The nonce can be randomly generated from a set $\mathbb{R}$ given that $|\mathbb{R}|$ is sufficiently large. Often these random nonces are called IV's (Initialization Vectors).

Constructions

The constructions below are the most used and are a blend of randomized and nonce based schemes.

1. **Cipher Block Chaining**
   This is one of the most popular and widely-used CPA-secure schemes. The advantages to using CBC are that using only slight modification it is a good authentication measure and is resistant to many published cryptanalytic methods. Further, if a block of cipher text is damaged in transmission, only that corresponding message block and the one next will be corrupted. The rest of the message blocks are untouched. The disadvantage to using it is that cannot be parallelized. There are two versions of CBC - one with a random IV (depicted below) and the other is a Nonce-Based Scheme[7]

   Encryption



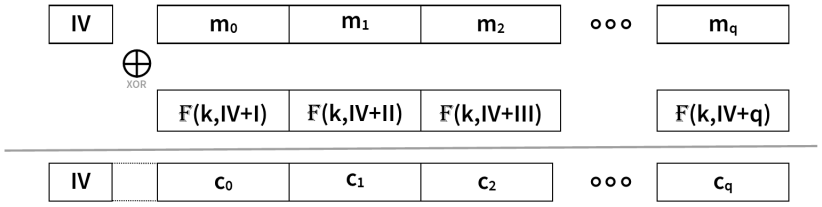
2. **Randomized Counter Mode**
   This is another nonce-based encryption scheme. The reader may notice that this is very similar to the structure of a Stream Cipher. The advantage this method has over the others is that it is parallelizable[8]. Thus, one can use multiple AES engines to compute the ciphertext. Furthermore, another advantage to using this scheme is that to decrypt the i'th block, only the i'th ciphertext is needed. This property is called **random access**. This can be particularly useful when a only one message block is changed in an $n-$block message. Suppose in a given message, we want to replace m_i with m_i', but we have already performed the encryption of the whole block. To alter this we simply apply the following formula to get the new ciphertext block at the i^{th} position (c_i'),

   $$c_i' = c_i \oplus m_i \oplus m_i'$$

   This gives it an advantage over CBC (where if you wanted to change a ciphertext at the i^{th} position, you'd have to recompute the all the ciphertext blocks from i to the end.)

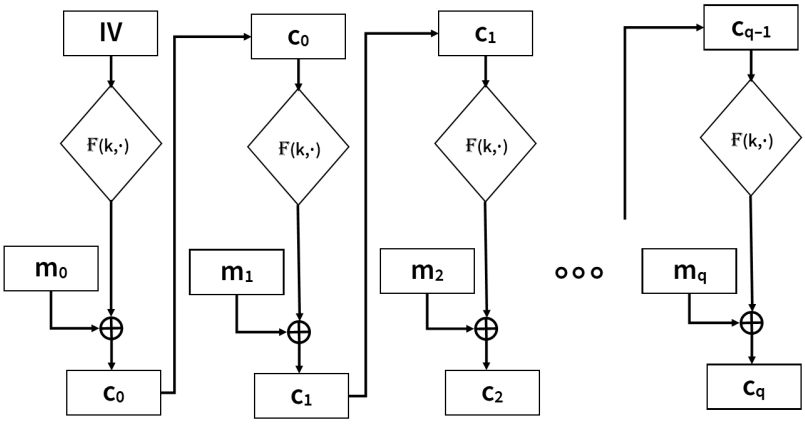
   Encryption

[7] Use these links to learn more about CBCs with noncesNonce-Based CBC
[8] This means that the encryption process can be run in multiple instances or processes



3. **Cipher Feedback Mode**
   The CFB mode is similar to the CBC mode described above. The main difference is that one should encrypt ciphertext data from the previous round (so not the plaintext block) and then add the output to the plaintext bits. It does not affect the cipher security but it results in the fact that the same encryption algorithm (as was used for encrypting plaintext data) should be used during the decryption process. One disadvantage to this is that if one block of plaintext is damaged, after encryption all the cipher texts after that block will be damaged too.

Encryption



4. **Output Feedback Mode**

Authenticated Encryption

In the real world, semantic security is enough to guarantee the security of the message but not enough to guarantee the 'integrity' of the message. In real world application, message integrity is as important as security and if a ciphertext can be manipulated or forged (even without complete knowledge of its contents), this can comprise the whole system. Before showing why this is true, a brief introduction into message authentication is necessary.

Message Authentication Codes (MACs)

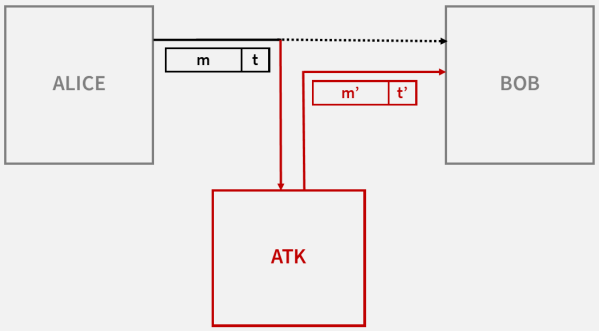
Message Authentication Codes (MACs for short) are a pair of functions $(\mathbb{S}, \mathbb{V})$ whose role is to 'sign' and 'verify' the message respectively.

$$\mathbb{S} : \mathbb{K} \times \mathbb{M} \to \mathbb{T} \qquad \mathbb{V} : \mathbb{K} \times \mathbb{M} \times \mathbb{T} \to \{0, 1\}$$

$$\mathbb{S}(k, m) = t \qquad \mathbb{V}(k, m, t) = \begin{cases} 1 & \text{if } \mathbb{S}(k, m) = t \\ 0 & \text{otherwise} \end{cases}$$

where $\mathbb{T}$ is the 'tag-space' (a set of all possible message tags). Usually, the tag is appended to the ciphertext and sent out into the clear.

Why must there be a key?
The secret key between Alice and Bob is a necessity. Additionally, the secret key for MACs should be different from the secret key for encryption. Consider a system without a shared secret key for message authentication.[a] Given below is a transmission where Alice and Bob are communicating and their MAC scheme does not use a secret shared key.



Any attacker would simply block the communication between Alice and Bob, replace Alice's messages with his/her own(m') and generate a MAC tag on that message (t'). This would be sent to Bob who will have no idea whether it is Alice's actual message. This completely compromises the MAC scheme.

[a]Read about why that is true on this stackexchange post - Why can't I use the same key for encryption and MAC?

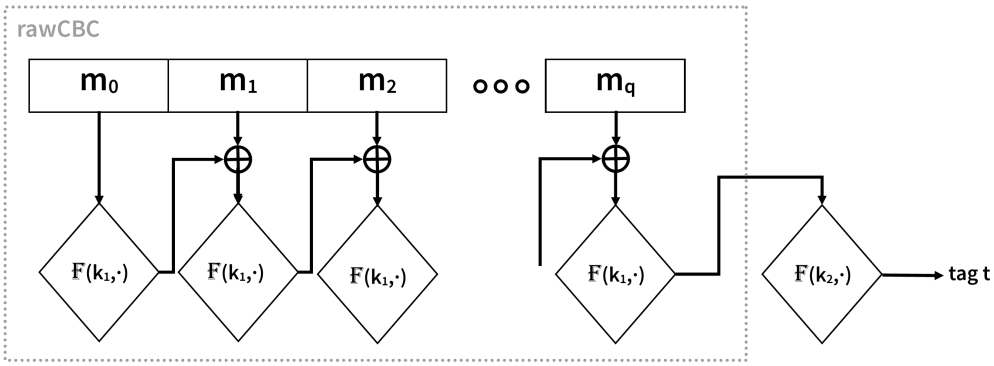
MAC Security
The security of MACs is quantified in its resistance to forgery. Given a message-tag pair (m, t), an attacker cannot forge a new tag t' such that $\mathbb{V}(k, m, t') = 1$ when $t \neq t'$. We say that a MAC is secure if an attacker cannot generate this alternate tag with high probability. If an attacker can generate such an alternate tag, we say that they have achieved **existential forgery** and that the MAC is insecure.

MACs from PRFs
We can construct secure MACs from secure PRFs.
Given a secure PRF $\mathbb{F} : \mathbb{K} \times \mathbb{M} \to \mathbb{C}$, we can define a MAC $(\mathbb{S}, \mathbb{V})$ such that

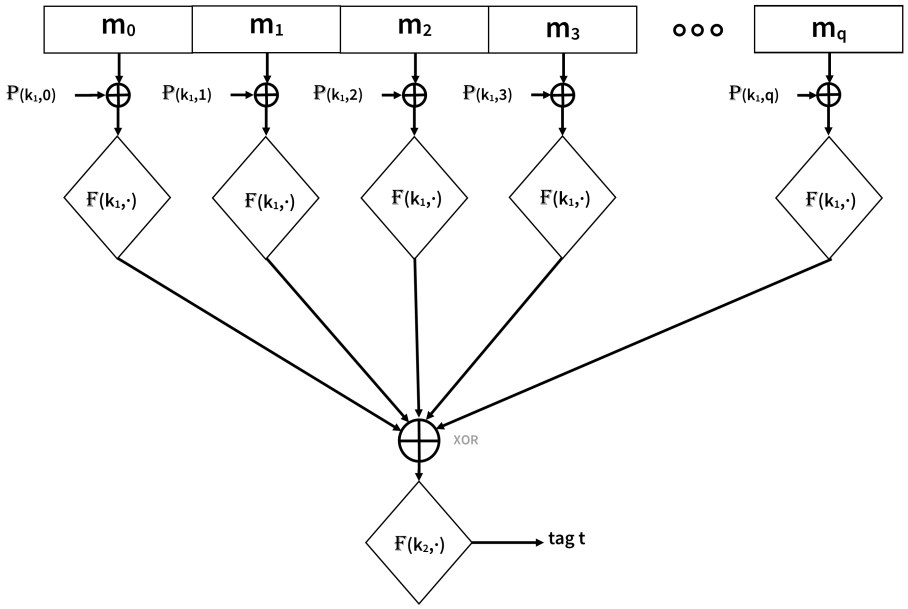
$$\mathbb{S}(k, m) = \mathbb{F}(k, m) \qquad \mathbb{V}(k, m, t) = \begin{cases} 1 & \text{if } \mathbb{F}(k, m) = t \\ 0 & \text{otherwise} \end{cases}$$

From here, we only need to show that $\mathbb{F}$ is a secure PRF and that $\frac{1}{|\mathbb{C}|}$ is negligible (i.e $\leq \frac{1}{2^{80}}$). Now we can construct a one-block MAC. To generalize this to $n-$blocks, several constructions exist. Some of these are depicted below.

1. **CBC-MAC**
   This MAC scheme is used widely in AE schemes. It uses 2 keys, one for the chained encryption and the last step uses a seperate key. If the last step is omitted, the scheme is called **rawCBC** (which is extremely vulnerable). The last step is vital to the security of this MAC scheme.



2. **PMAC**
   This MAC scheme is especially popular because it is parallelizable.

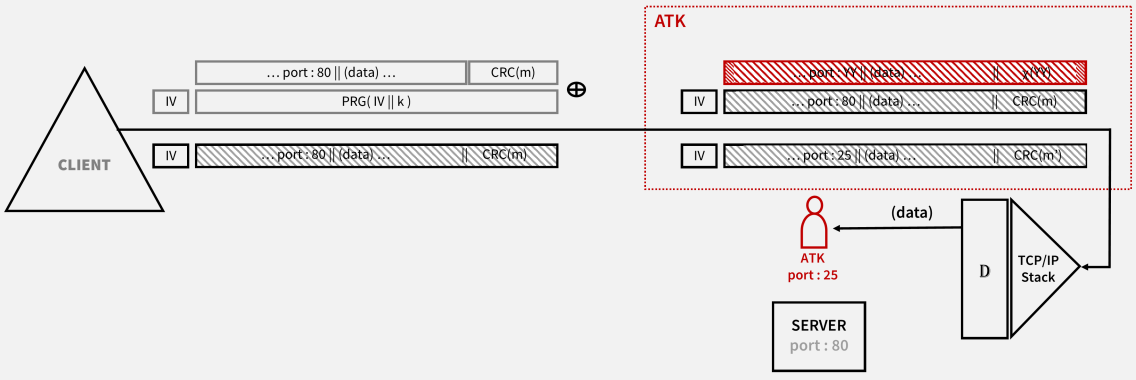


The $\mathbb{P}$ function is especially important because if it wasn't in the scheme, we could swap 2 blocks and still have the same tag leading to an existential forgery. However, $\mathbb{P}$ is easy to compute so it does not add drastically to the MAC run time.

REAL WORLD MAC FORGERY

As mentioned earlier in the OTP Documentation, WEP was a disaster and often used as an example of how **not to do** things in cryptography. It should never be used in application. We already saw one flaw in its implementation (short IVs) that led to a TTPA. Another flaw is its message authentication scheme. In WEP, even the CRC (a checksum which authenticates the data) is insecure because of a linearity property. The CRC is linear, i.e $\forall m, p \;\; CRC(m \oplus p) = CRC(m) \oplus \chi(p)$ where χ is a well known public function. This can be exploited to alter the message **AND** the MAC tag.

Consider a system where an attacker (ATK) is trying to break the WEP encryption between a client and a server at port80. ATK is connected at port25. All the messages feed into a TCP/IP Stack that will decrypt the message with the shared key, verify the message and then send to the port that it was addressed to. Since ATK knows the destination port, he will set up a replacement (YY) to be XORed with the ciphertext. He knows that if can manipulate the bits of the ciphertext at the exact position, the message will decrypt to port25 instead of port80. So if $80 \oplus k_1k_2 = c_1c_2$ then $(80 \oplus k_1k_2) \oplus (YY) = 25 \oplus k_1k_2$. Thus he sets $YY = 80 \oplus 25$.
However, even if he can manipulate the message, he must also change the MAC tag otherwise the verification will fail. So he exploits the CRC's weakness. He XORs the ciphertext's CRC position with $\chi(YY)$. He knows that $CRC(m \oplus YY) = CRC(m) \oplus \chi(YY)$. Having manipulated both the message and the MAC tag in the right way, the TCP/IP Stack will decrypt the message and verify it and send to port25 instead of port80. **ATK successfully broke the scheme.**

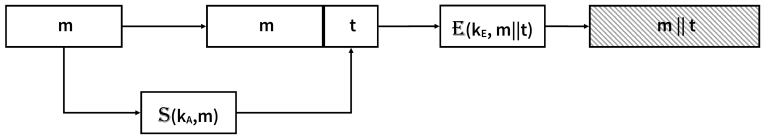


Building Authenticated Encryption Schemes

Given that AE schemes are a must for real-world communication, how do we combine semantically secure ciphers

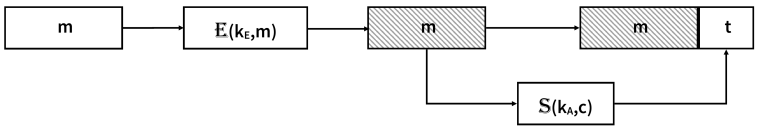
and MACs in the right way to achieve this? We have 3 options -

1. **MAC then Encrypt** (SSL)



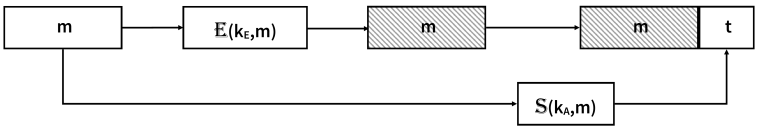
This option is the second best on the list (first being Option 2). It may also be vulnerable to Chosen Ciphertext Attacks. This is AE-Secure when the PRF $\mathbb{F}$ is either CBC or RANDCTR.

2. **Encrypt then MAC V1** (IPSec)



This is the best method for building an AE Scheme. It reinforces the AEAD (Authenticated Encryption with Associated Data) Principle. This principle states that for real-world communication, **not all end-to-end transmission should be encrypted but all of it should be authenticated.** The reader may find this counter-intuitive, but one should convince oneself that this is in fact true.

3. **Encrypt then MAC V2** (SSH)



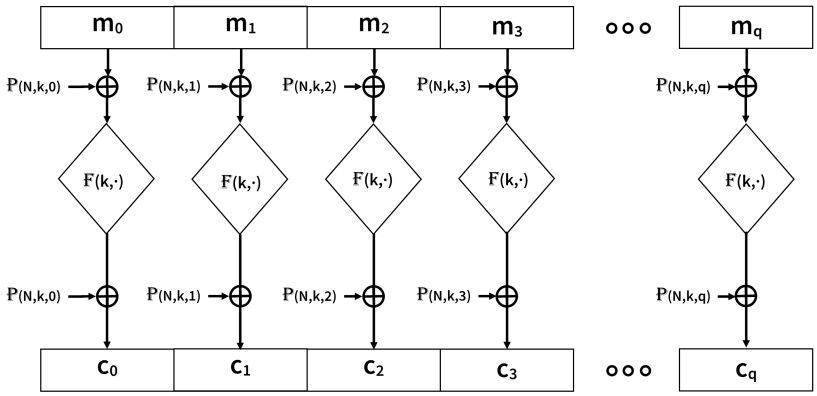
This option is not recommended. This is because MACs are built to prevent forgery not to ensure security, hence a MAC tag might leak bits of the message if it was sent in the clear. **This option should be avoided**.

Constructions

These constructions are widely adopted and implemented AE schemes.

1. **Offset Code Book**
   OCB is a direct construction from a block cipher. It is a departure from the paradigm of blending semantically secure ciphers and MACs to build AE schemes. The advantage to using this is that we do not have to evaluate the PRF twice (once for encryption and once for MACing). Furthermore, it is parallelizable and the block cipher is evaluated only once per block. This makes it an extremely fast and efficient AE Scheme.

Encryption



2. **Galois Counter Mode**
   This is one of the NIST Standards for Authenticated Encryption schemes. It blends RANDCTR for encryption with a Carter-Wegman MAC[9]. Nearly every chip today has acceleration for AES-GCM. GCM is used all over the place - TLS 1.2 IPSec standards, etc.

3. **Counter CBC-MAC**
   As the name suggests, this scheme mixes a CBC-MAC with RANDCTR encryption. It uses only AES and thus can be implemented using relatively little code. The wireless standard 802.11i used CCM.

GCM and CCM also abide by the AEAD Principle. Usually, given a message containing header||data, the scheme will authenticate (generate a tag) on the whole message, but will only encrypt the data.

Cryptanalysis

Many documented attacks exist against not only the Advanced Encryption Standard but to block ciphers in general. We shall examine a few here.

Quantum Key Search

This attack spawns out of a search algorithm named Grover's Algorithm. In a classical computer, the time complexity for running a search on a unsorted list is $O(n)$. However, a quantum computer running Grover's algorithm can a search the same unsorted list in $O(\sqrt{n})$ time. Let us see how this helps. Let f be function defined as follows.

$$f : \mathbb{X} \to \{0, 1\}$$

Suppose we know that only one element in $\mathbb{X}$ returns a 1 when passed through the function. The task is to find some $x \in \mathbb{X}$ such that $f(x) = 1$. A classical computer would be able to find this element in $O(|\mathbb{X}|)$ time. A quantum computer running Grover's algorithm could do this in $O(|\mathbb{X}|^{\frac{1}{2}})$. Using this quantum search advantage, we can elaborate on how the function f is computed -

Given a message-ciphertext pair (m, c), then $\forall k \in \mathbb{K}$

$$f(k) = \begin{cases} 1 & \text{when } \mathbb{E}(k, m) = c \\ 0 & \text{otherwise} \end{cases}$$

So we can say that a quantum computer can find this key k in time $O(|\mathbb{K}|^{\frac{1}{2}})$.
For DES, this would take $\approx 2^{28}$ time. For AES-128, this would take $\approx 2^{64}$ time. For AES-192 and AES-256, this would take $\approx 2^{96}$ and $\approx 2^{128}$ time respectively. (Recall that the threshold for intractability is time $\geq 2^{90}$). Thus, this proposed key search would endanger AES-128 but not AES-192 and AES-256.

Side Channel Attacks

Further Reading

Message Authentication Codes

Read more about MAC Constructions here - Message Authentication Code (MAC)

[9] Read about One-Time MACs and Carter Wegman MACs from the articles stated in the Further Reading Section