Anna Krawczuk
Gabriel Skowron-Rodriguez

AES cipher – software implementation with ECB and CBC ciphering modes

1. Theoretical introduction

The AES cipher

AES is a symmetric block cipher that is intended to replace DES as the approved standard for a wide range of applications. Compared to public-key ciphers such as RSA, the structure of AES and most symmetric ciphers is quite complex. AES does not use a Feistel structure. Instead, each full round consists of four separate functions: byte substitution, permutation, arithmetic operations over a finite field, and XOR with a key.

The cipher takes a plaintext block size of 128 bits, or 16 bytes. The key length can be 16, 24, or 32 bytes (128, 192, or 256 bits). The algorithm is referred to as AES-128, AES-192, or AES-256, depending on the key length. The key size used for an AES cipher specifies the number of transformation rounds that convert the input, called the plaintext, into the final output, called the ciphertext.

The input to the encryption and decryption algorithms is a single 128-bit block. In FIPS PUB 197, this block is depicted as a 4x4 square matrix of bytes. This block is copied into the State array, which is modified at each stage of encryption or decryption. After the final stage, State is copied to an output matrix.

Similarly, the key is depicted as a square matrix of bytes. This key is then expanded into an array of Round Keys. The key expansion function generates N + 1 round keys, each of which is a distinct 4x4 matrix. The length of the expanded key depends on the length of the original key.

The cipher consists of N rounds, where the number of rounds depends on the key length . Below is a table of dependancies between the key length and other parameters:

Table 1 AES Parameters

Key Size (words/bytes/bits)	4/16/128	6/24/192	8/32/256
Plaintext Block Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Number of Rounds	10	12	14
Round Key Size (words/bytes/bits)	4/16/128	4/16/128	4/16/128
Expanded Key Size (words/bytes)	44/176	52/208	60/240

And following is a graph that shows the structure of the cipher:

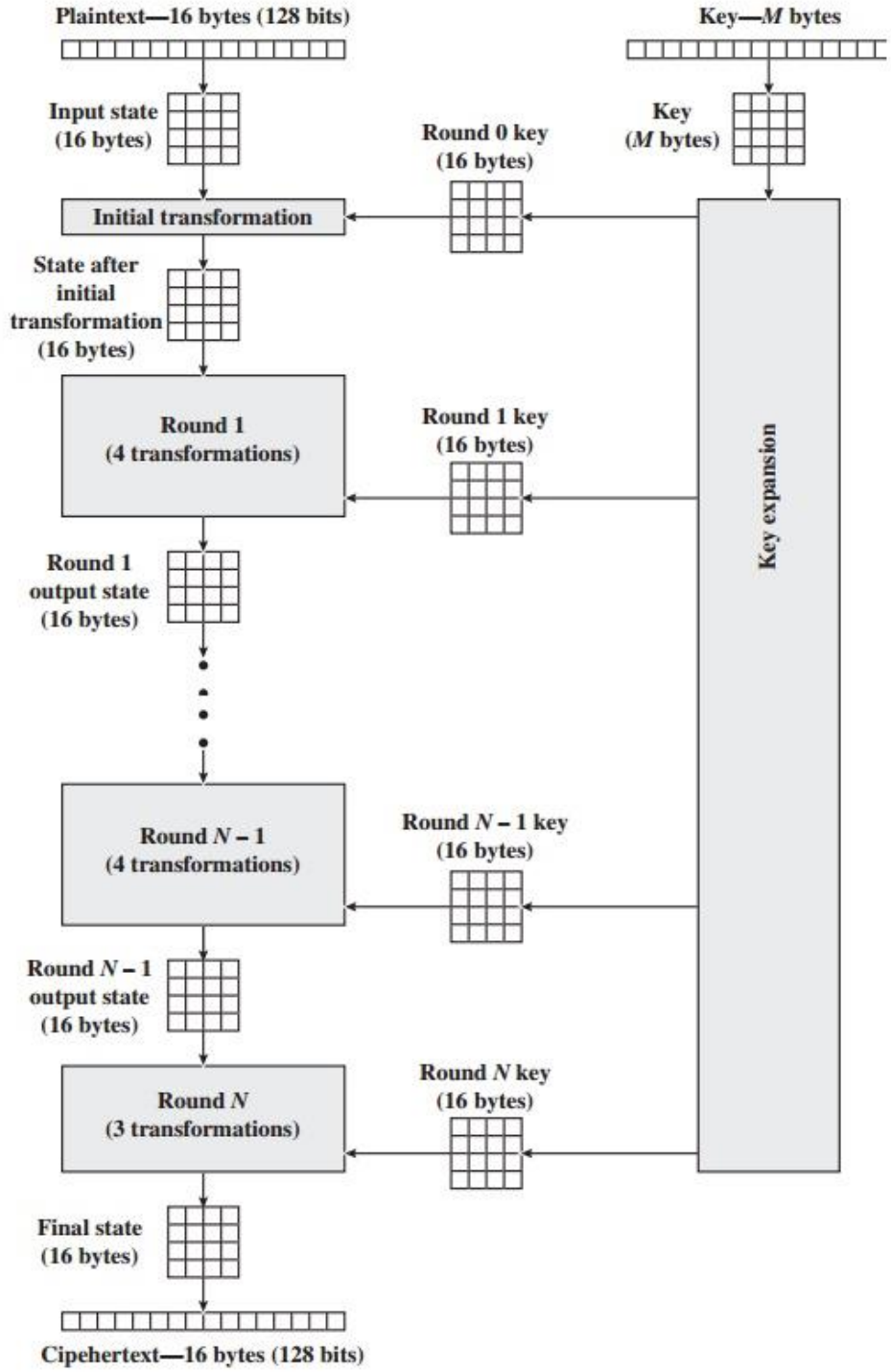


Figure 1 AES Encryption Process

The first N - 1 rounds consist of four distinct transformation functions: **SubBytes, ShiftRows, MixColumns, and AddRoundKey**, which are described below. The final round contains only three transformations (MixColumns is not performed), and there is a initial single transformation (AddRoundKey) before the first round, which can be considered Round 0. Each transformation takes one or more 4x4 matrices as input and produces a 4x4 matrix as output. Each round key generated form the expansion serve as one of the inputs to the AddRoundKey transformation in each round.

The 4 transformation functions are one of permutation and three of substitution:

- Substitute bytes: Uses an S-box to perform a byte-by-byte substitution of the block
- ShiftRows: A simple permutation
- MixColumns: A substitution that makes use of arithmetic over GF(2^8)
- AddRoundKey: A simple bitwise XOR of the current block with a portion of the expanded key

The decryption algorithm follows the same structure. However, the transformations inside the round occur in different order, and the transformation functions (except for AddRoundKey) appear in their inverted variations. Below is a graph that shows the differences:

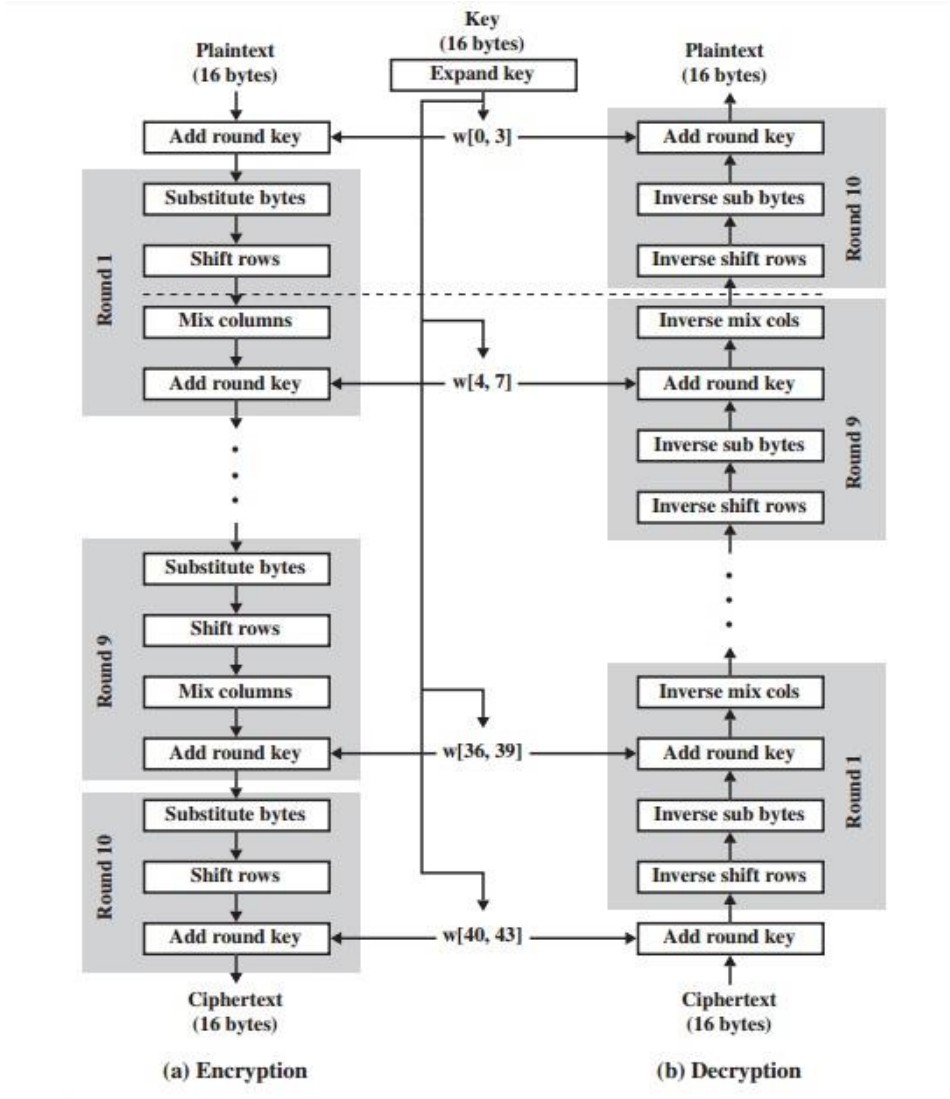


Figure 2 AES Encryption and Decryption

Following are short descriptions of the 4 transformation functions, then the key expansion function:

a) SubBytes

In the SubBytes step, each byte in the state array is replaced with a SubByte using an 8-bit substitution box (s-box) defined by the FIPS PUB 197, in the following way: the leftmost 4 bits of the byte are used as a row value and the rightmost 4 bits are used as a column value. The inverted version of this function, used in decryption algorithm, makes use of the inverse s-box. The vaules in both s-boxes are hexadecimal.

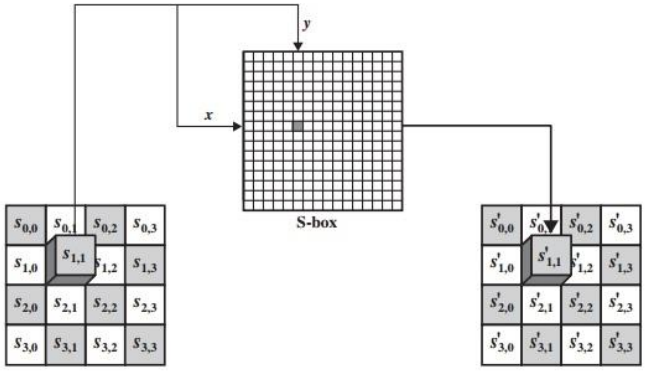
Table 2 AES S-Boxes

x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

(a) S-box

x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

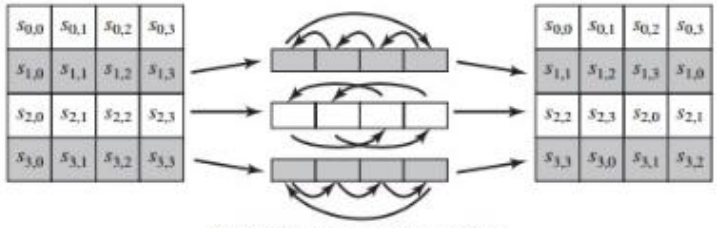
(b) Inverse S-box



(a) Substitute byte transformation

b) ShiftRows

The ShiftRows step operates on the rows of the state; it cyclically shifts the bytes in each row by a certain offset. For AES, the first row is left unchanged. Each byte of the second row is shifted one to the left. Similarly, the third and fourth rows are shifted by offsets of two and three respectively. In this way, each column of the output state of the ShiftRows step is composed of bytes from each column of the input state.

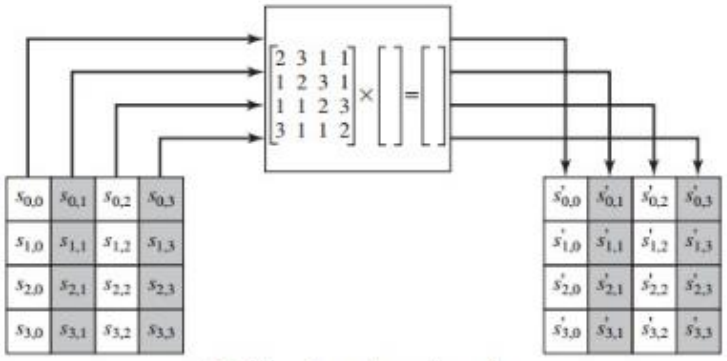


(a) Shift row transformation

c) MixColumns

In the MixColumns step, the four bytes of each column of the state are combined using an invertible linear transformation. The MixColumns function takes four bytes as input and outputs four bytes, where each input byte affects all four output bytes.

During this operation, each column is transformed using a fixed matrix (matrix left-multiplied by column gives new value of column in the state):



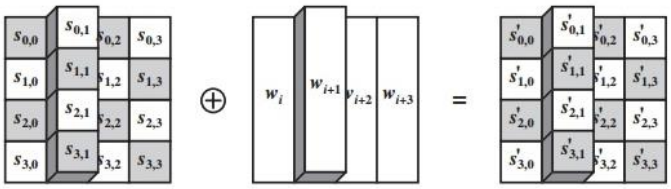
(b) Mix column transformation

Matrix multiplication is composed of multiplication and addition of the entries. Entries are bytes treated as coefficients of polynomial of order x^7. Addition is simply XOR. Multiplication is modulo irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. If processed bit by bit, then, after shifting, a conditional XOR with $1B_{16}$ should be performed if the shifted value is larger than FF_{16} (overflow must be corrected by subtraction of generating polynomial). These are special cases of the usual multiplication in $GF(2^8)$. The MixColumns step can also be viewed as a multiplication by the shown particular MDS matrix in the finite field $GF(2^8)$. When decrypting, a different matrix is used (shown on the right):

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

d) AddRoundKey

In the AddRoundKey step, the subkey is added by combining each byte of the state with the corresponding byte of the subkey using bitwise XOR.



(b) Add round key transformation

e) Key expansion

This section is explained on the example of a 16-byte key, but the structure is the same regardless of length.

The AES key expansion algorithm takes as input a four-word (16-byte) key and produces a linear array of 44 words (176 bytes). This is sufficient to provide a four-word round key for the initial AddRoundKey stage and each of the 10 rounds of the cipher. The key is copied into the first four words of the expanded key. The remainder of the expanded key is filled in four words at a time. Each added word w[i] depends on the immediately preceding word, w[i - 1], and the word four positions back, w[i - 4]. In three out of four cases, a simple XOR is used. For a word whose position in the w array is a multiple of 4, a more complex function **g** is used. The function **g** consists of the following subfunctions:

1. RotWord performs a one-byte circular left shift on a word.
2. SubWord performs a byte substitution on each byte of its input word, using the S-box.
3. The result of steps 1 and 2 is XORed with a round constant, Rcon[j].

The values of RC[j] in hexadecimal are:

j	1	2	3	4	5	6	7	8	9	10
RC[j]	01	02	04	08	10	20	40	80	1B	36

The following is a graph of the structure of key expansion:

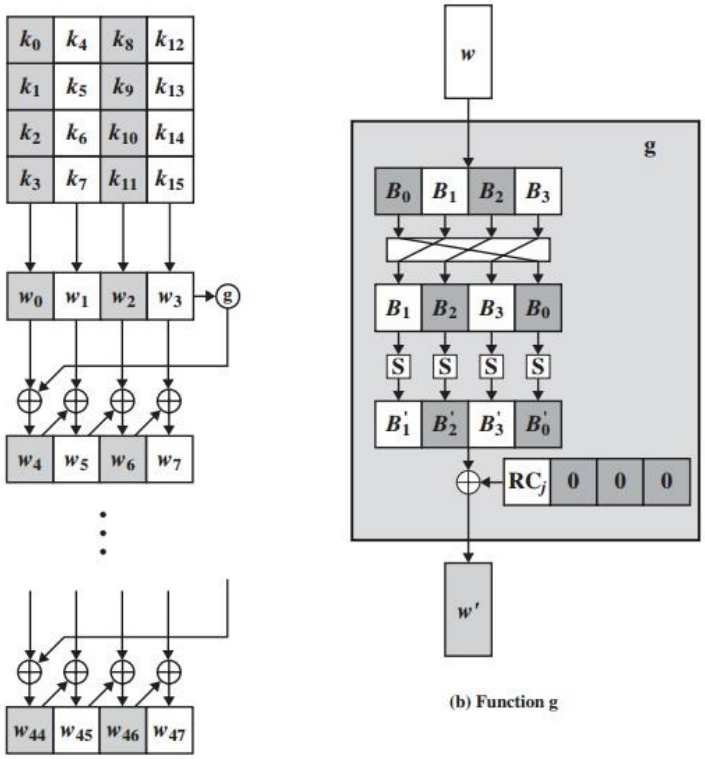


Figure 5 AES Key Expansion

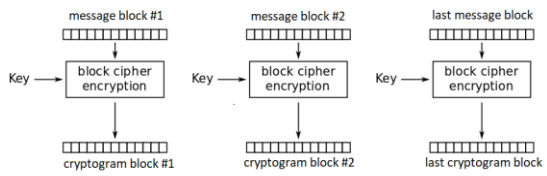
ECB and CBC ciphering modes

Now that we know how the AES ciphering/deciphering unit operates, we can introduce ciphering modes.

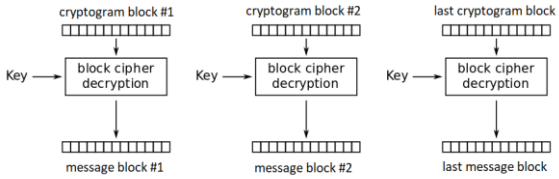
Since most plaintext messages are longer than 128 bits (which translates to 16 characters in utf-8 encoding), we need to be ready to split the message into 128-bit long blocks.

The concept Electronic Codebook (ECB) ciphering mode is:

Encrypt all message blocks in parallel and construct the cryptogram by concatenating subsequent cryptogram blocks in the original order. We decode the cryptogram in an analogous manner:



Electronic Codebook (ECB) mode encryption



Electronic Codebook (ECB) mode decryption

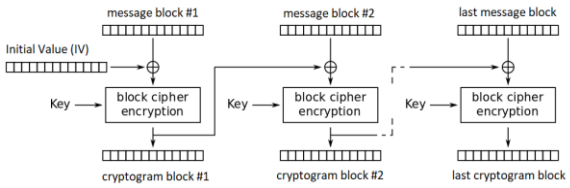
As we can see on the pictures above, the ECB mode is very simple to implement, but has one major security flaw: identical message blocks within a message are always encrypted into identical cryptogram blocks.

On the other hand, ECB operations can be parallelized – significantly reducing time needed to perform encryption/decryption.

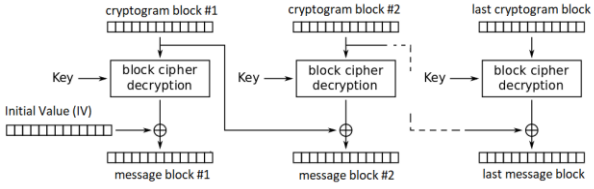
The Cipher Block Chaining technique is a different approach:

Encrypt message blocks one by one and take the previous blocks into consideration.

In order to decrypt the message, we need to perform a similar approach:



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

This means that any two identical message blocks are very unlikely to be represented by identical cryptogram blocks. Furthermore, since we need to provide some 128-bit Initial Value for our encryption/decryption processes, there is one more variable a potential attacker must consider.

However, the longer the message, the slower the CBC mode is: we need to encrypt the message block-by-block.

Block Padding

Messages' length is rarely a multiple of 16 characters. This is why we need to perform padding after "cutting" the message into blocks.

We will be performing PKCS#7 padding on the message bytes, which consists of the following steps:

1. Let **paddingSize** be the number of bytes needed to be appended to the message in order for it to be a multiple of block size (if the message's byte representation's length is already a multiple of block size – choose **paddingSize** == block size)
2. Append **paddingSize** number or special bytes to the message. Every single byte has an integer value equal to **paddingSize**
3. The obtained message's length is now a multiple of block size > proceed with encryption

When performing decryption, the padding is removed from the obtained (final) message bytes in the following manner:

1. Read the last byte as **paddingSize**
2. Discard **paddingSize** last bytes from the decrypted message
3. The original message's bytes have been recovered (by decoding using the selected codec – in our case: UTF-8)

Differences between the AES theory and implementation

The functions SubBytes and MixColumns use different, but equivalent methods of obtaining the same result, than were described in this theoretical section. The changes were done mainly in order to make the functions easier to program and do not change or impede the functionality of the code. The specific changes are explained in the comments in the code when relevant.

2. Functional description of the application (input data format, outputs, etc.)

Our application is a simple “test runner”:
One can add a mode, key, message, and optionally Initial Value to the tests array and observe the encrypted message.

Input data format:

To add a new test:

1. Find the comment: “#TESTS“
2. To the found dictionary, add a new test in the following manner:

```
"TEST NAME": {
            "mode": "CBC",
            "key": "VALID_SECRET_KEY",
            "initialValue": "16_CHAR_LONG_IV_",
            "message": "Some message to encrypt",
            "expectedCryptogram": None, # optional:
    #visit “# trusted source” to quickly obtain a valid expected cryptogram
        },
```

When setting up a CBC test, remember to include the “initialValue”.

If you want a ECB encryption to be performed, replace the “mode” with “ECB”

Running the program:

To run the program, simply execute the python file by:

a) running the command “py AES_Program.py” in the command line.

b) open the code file in an IDE of your choice and run/debug it

Output format:

After a brief greeting:

```
Hello, this is our AES implementation in python:
```

Each test result is described by a following set of lines:

```
Encrypting the following message in ECB mode, (key :bytearray(b'SuperSecret1234512345678')):
123456789ABCDEF123456789ABCDEF123456789AB
Result:
2a6e8a3df1847ab182d035e3ef65b203bcaa495bafff5f75827329311e32d25e0a108e8bfbe29c32ab6e8aca6e97224f
Correct!
Decrypting...
Decrypted message:
123456789ABCDEF123456789ABCDEF123456789AB
Correct!
```

Where we can see the key used, message and cryptogram.

If no reference value is specified, only decryption of cryptogram will be evaluated:

```
Encrypting the following message in CBC mode, (key :bytearray(b'VALID_SECRET_KEY')):
Some message to encrypt
Using Initial Variable: bytearray(b'16_CHAR_LONG_IV_')
Result:
a4cbc2b68a7bc646681b6ee8f073d0351edc052d84761cb87d499038ed7b27c8
Decrypting...
Decrypted message:
Some message to encrypt
Correct!
```

Last output line:

```
All tests passed.
```

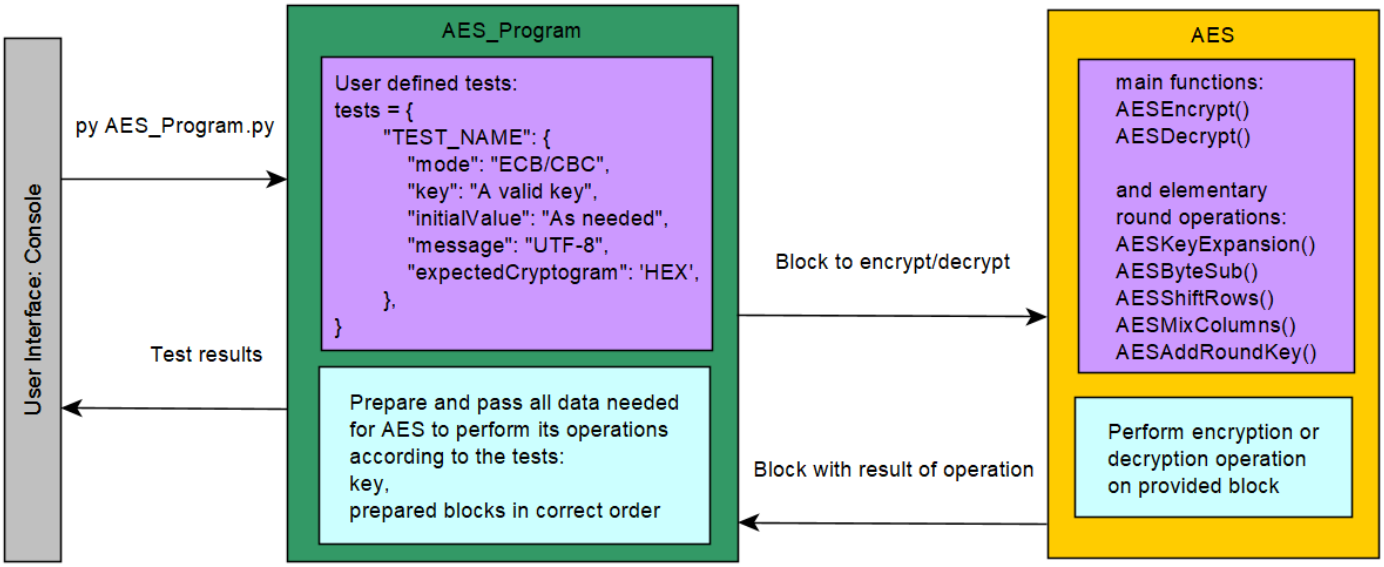
This message will be shown if all tests have been successful.

Whereas if any tests should fail, you will see a line like this:

```
Tests Failed: 1
```

Indicating the total number of failed tests.

3. Description of designed code structure:



Utilizing a console, user may run the program (AES_Program.py), which will perform all defined tests.

The "AES_Program" class implements specific ciphering modes by:

- Converting messages into AES-compatible blocks (according the each mode's specifications)
- Feeding data blocks into the AES class
- Receiving results of operations, concatenating them into cryptograms/deciphered messages and printing test results into the console

The "AES" class's sole purpose is to be a cipher/decipher unit:

- It receives data blocks and a key and kind of requested operation
- Encrypts/decrypts provided data blocks (as described in Theoretical Introduction)
- Outputs the result of requested operations.

4. Performed tests:

The tests were specified in a way shown in the Functional Description section. All reference values were generated using the website: https://www.javainuse.com/aesgenerator

The interface there is very clean and simple. The only thing to keep in mind is to change the Output Text Format from "Base64" to "Hex" before generating a reference value and copying it into a test definition.

Now, let's see how the tests went:

Test 1:

```
#TESTS
# trusted source: https://www.javainuse.com/aesgenerator  (expectecCryptograms are in hex format):
    tests = {
        "ECB": {
            "mode": "ECB",
            "key": "SuperSecret1234512345678",
            "initialValue": None,
            "message": "123456789ABCDEF123456789ABCDEF123456789AB",
            "expectedCryptogram":
            '2a6e8a3df1847ab182d035e3ef65b203bcaa495bafff5f75827329311e32d25e0a108e8bfbe29c32ab6e8aca6e97224f',
        },
```

Test 1 result:

```
Hello, this is our AES implementation in python:

Encrypting the following message in ECB mode, (key :bytearray(b'SuperSecret1234512345678')):
123456789ABCDEF123456789ABCDEF123456789AB
Result:
2a6e8a3df1847ab182d035e3ef65b203bcaa495bafff5f75827329311e32d25e0a108e8bfbe29c32ab6e8aca6e97224f
Correct!
Decrypting...
Decrypted message:
123456789ABCDEF123456789ABCDEF123456789AB
Correct!
```

As we can see, the ECB mode with a 24 bit key was successful. While the message being 41 characters long, the cryptogram is 48 bytes long. This means that PKCS#7 padding is working as intended.

Test 2:

```
        "CBC": {
            "mode": "CBC",
            "key": "SuperSecret1234512345678",
            "initialValue": "InitVarOLength16",
            "message": "123456789ABCDEF123456789ABCDEF123456789AB",
            "expectedCryptogram":
            '0966b37a583dcd2a6713ed3cd894301be1d8443f9ab2db2bc1e9677203a72beeb9d6b933b28724410e8740999b90cd10',
        },
```

Test 2 result:

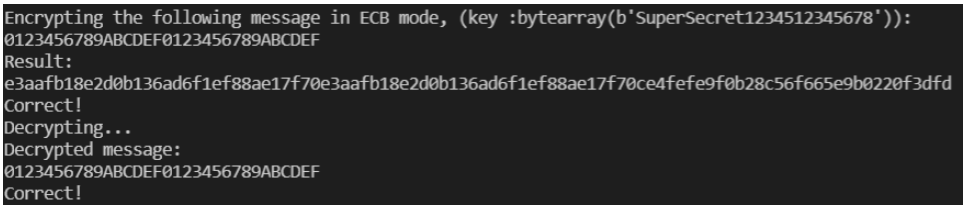
```
Encrypting the following message in CBC mode, (key :bytearray(b'SuperSecret1234512345678')):
123456789ABCDEF123456789ABCDEF123456789AB
Using Initial Variable: bytearray(b'InitVarOLength16')
Result:
0966b37a583dcd2a6713ed3cd894301be1d8443f9ab2db2bc1e9677203a72beeb9d6b933b28724410e8740999b90cd10
Correct!
Decrypting...
Decrypted message:
123456789ABCDEF123456789ABCDEF123456789AB
Correct!
```

CBC ciphering of the same message (and with the same key) yielded a different but correct cryptogram. The cryptogram is also of length 48 (proving padding works for CBC mode as well)

Test 3:

```
    "ECB2": {
        "mode": "ECB",
        "key": "SuperSecret1234512345678",
        "initialValue": None,
        "message": "0123456789ABCDEF0123456789ABCDEF",
        "expectedCryptogram":
        'e3aafb18e2d0b136ad6f1ef88ae17f70e3aafb18e2d0b136ad6f1ef88ae17f70ce4fefe9f0b28c56f665e9b0220f3dfd',
    },
```

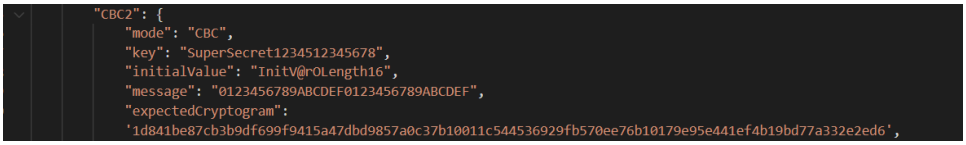
Test 3 result:



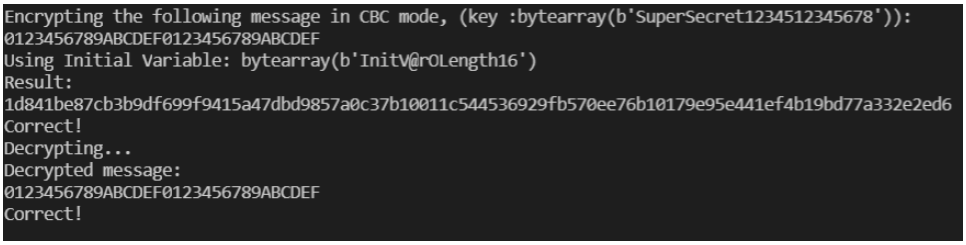
```
Encrypting the following message in ECB mode, (key :bytearray(b'SuperSecret1234512345678')):
0123456789ABCDEF0123456789ABCDEF
Result:
e3aafb18e2d0b136ad6f1ef88ae17f70e3aafb18e2d0b136ad6f1ef88ae17f70ce4fefe9f0b28c56f665e9b0220f3dfd
Correct!
Decrypting...
Decrypted message:
0123456789ABCDEF0123456789ABCDEF
Correct!
```

The message's length is a multiple of 16 characters (32 to be exact), and we can clearly see that the cryptogram is of length 48 bytes, which proves that padding works as intended (even such specific messages)

Test 4:



```
    "CBC2": {
        "mode": "CBC",
        "key": "SuperSecret1234512345678",
        "initialValue": "InitV@rOLength16",
        "message": "0123456789ABCDEF0123456789ABCDEF",
        "expectedCryptogram":
        '1d841be87cb3b9df699f9415a47dbd9857a0c37b10011c544536929fb570ee76b10179e95e441ef4b19bd77a332e2ed6',
```



```
Encrypting the following message in CBC mode, (key :bytearray(b'SuperSecret1234512345678')):
0123456789ABCDEF0123456789ABCDEF
Using Initial Variable: bytearray(b'InitV@rOLength16')
Result:
1d841be87cb3b9df699f9415a47dbd9857a0c37b10011c544536929fb570ee76b10179e95e441ef4b19bd77a332e2ed6
Correct!
Decrypting...
Decrypted message:
0123456789ABCDEF0123456789ABCDEF
Correct!
```

The same message was processed using CBC mode, with success.

Test 5:

```
"ECB3": {
    "mode": "ECB",
    "key": "SuperSecret12345",
    "initialValue": None,
    "message": "Some secretive text that needs to be encrypted",
    "expectedCryptogram":
    '6c3e288541bca3a5a20056a4653d57470a3a9eb620116af6d46b6081cc8adbb1ff163ddf6f0a23202542b2c059dab5b4',
},
```

```
Encrypting the following message in ECB mode, (key :bytearray(b'SuperSecret12345')):
Some secretive text that needs to be encrypted
Result:
6c3e288541bca3a5a20056a4653d57470a3a9eb620116af6d46b6081cc8adbb1ff163ddf6f0a23202542b2c059dab5b4
Correct!
Decrypting...
Decrypted message:
Some secretive text that needs to be encrypted
Correct!
```

A CBC AES-128 encryption was performed successfully. The cryptogram was the same as the expected one, and the decryption worked perfectly as well.

5. Bibliography:

AES specification: FIPS PUB 197: Advanced Encryption Standard (AES)

AES images: "CRYPTOGRAPHY AND NETWORK SECURITY PRINCIPLES AND PRACTICE" by William Stallings published on brainkart.com

ECB and CBC figures adapted from:
https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_codebook_(ECB)