Information Systems Security
Mandatory TP 1 - AES

September 29th, 2021

Submit on **Moodle** your Python 3 file(s) **.py**, before **Tuesday, October 19th, 2021 at 11:59 pm (23h59)**.

Your code needs to be **commented**.

Goal

The goal of this TP is simple : you will create an AES block cipher, and use it to encode messages with ECB (Electronic Codebook) mode.

AES Encryption Box

The AES Box will be used with 128 bits plaintext/ciphertext blocks, and with keys of either 128, 192 or 256 bits. We have one initial step, and we then add 10 rounds for 128 bit keys, 12 rounds for 192 bit keys, or 14 rounds for 256 bit keys.

Whatever the size of the key, we will always use 128 bit blocks, and create sub-keys of 128 bits.

IMPORTANT : All calculations are made considering the 128 bits block as a 4x4 matrix of bytes (8 bits), where each byte is as an element of $GF(2^8)$, which is the group of polynomials of degree 7 (0 to 7, that's eight coefficients) with elements being either 0 or 1.
For instance, 00110101 represents the polynomial $x^5 + x^4 + x^2 + 1$.
We will then use addition and multiplication for polynomials. In this case, the addition becomes a simple XOR, since we only work with 0 and 1 modulo 2.

VERY IMPORTANT : When slicing the 128 bit message as a 4x4 matrix, they are ordered column by column :

$$\begin{pmatrix} m(1,8) & m(33,40) & m(65,72) & m(97,104) \\ m(9,16) & m(41,48) & m(73,80) & m(105,112) \\ m(17,24) & m(49,56) & m(81,88) & m(113,120) \\ m(25,32) & m(57,64) & m(89,96) & m(121,128) \end{pmatrix}$$

Where m(x,y) represents the bits x to y from the message block (starting with 1 in this case). Each 128 sub-key is also organised as a matrix, in the exact same way (if you do either row by row, your results will obviously be wrong in some of the operations).

- **Key expansion** : First, we need a key expansion system, which will create a number of keys equal to the number of rounds plus one (which we will need for the initial step).

  We define multiple 32 bits constants, named $rcon_1$ to $rcon_{10}$ (which are created following some rules, but it's easier to just give the table) :

  $$rcon_i = [rc_i\ (00)_{16}\ (00)_{16}\ (00)_{16}]$$

  So the last 24 bits are zeros for all these constants. Only the first 8 bits, rc_i, will differ, as follows (for ex, $rcon_9 = (1B000000)_{16}$) :

i	1	2	3	4	5	6	7	8	9	10
rc_i	$(01)_{16}$	$(02)_{16}$	$(04)_{16}$	$(08)_{16}$	$(10)_{16}$	$(20)_{16}$	$(40)_{16}$	$(80)_{16}$	$(1B)_{16}$	$(36)_{16}$

  We also define :

  – N as the number of 32-bit words of the key (4, 6 or 8 depending if the key is 128, 192 or 256 bits),
  – K_0, K_1, ..., K_{N-1} the 32-bit words of the original key,
  – R the number of keys needed (11, 13 or 15),
  – W_0,W_1,...,W_{4R-1} the 32-bit words of the expanded key (this is the result we're looking for).
  – Rotation($[b_0,b_1,b_2,b_3]$) = $[b_1,b_2,b_3,b_0]$ as a one byte left circular shift,
  – SBox($[b_0,b_1,b_2,b_3]$) = $[S(b_0),S(b_1),S(b_2),S(b_3)]$ the application of the AES S-Box on each of the four bytes.

  And we can finally compute the whole expanded key as :

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

  So the first 4 words (W_0 to W_3) are the first 128 bit key (for the initial step), then the next four ones (W_4 to W_7) are the 128 bit key for the first round, and so on to the last four ones (W_{4R-4} to W_{4R-1}) for the last round.

- **initial step** : The initial step is an xor with the first sub-key (The W_0 to W_3 in the previous definition),
- **Rounds** : Now, we will do 10/12/14 rounds as follows :
  1. **ByteSub** : ByteSub is a non linear permutation, applied byte by byte, following this given table :

R,C	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

Fig. 3. S-box lookup table

Figure 1: The AES S-Box

     The four first bits (the bigger ones) of the byte give the row, and the last four (the weaker ones) give the column. For example, $(27)_{16}$ is replaced by $(cc)_{16}$.
  2. **ShiftRow** : It is a simple operation in which the elements of each row of the matrix are shifted.
     - The first row is left intact,
     - the second row is shifted one byte on the left,
     - the third row two bytes on the left,
     - and the last one by three bytes on the left.

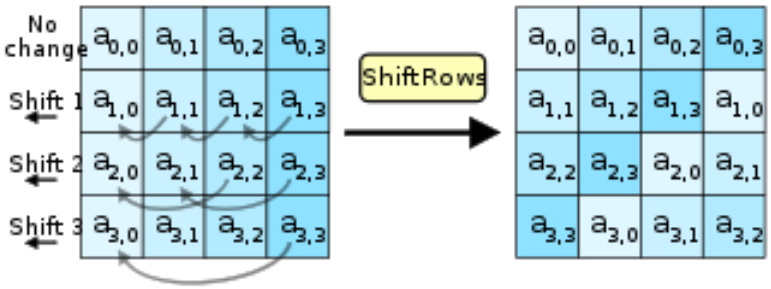

Figure 2: Shift Rows step for AES

3. **MixColumn** : Each column of 4 bytes b_0, b_1, b_2, b_3 is considered as a polynomial of degree 3 with coefficients being the bytes, which means elements of $GF(2^8)$.
   These columns are then multiplied by a fixed polynomial modulo another fixed polynomial (Details can be found easily on google), but it is equivalent (proof can be found easily too, even on wikipedia) to say the new bytes d_0, d_1, d_2 and d_3 are equal to :

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} (02)_{16} & (03)_{16} & (01)_{16} & (01)_{16} \\ (01)_{16} & (02)_{16} & (03)_{16} & (01)_{16} \\ (01)_{16} & (01)_{16} & (02)_{16} & (03)_{16} \\ (03)_{16} & (01)_{16} & (01)_{16} & (02)_{16} \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad (1)$$

   Attention : Remember that this matrix multiplication applies to bytes which are elements in $GF(2^8)$. Which means all operations are in $GF(2^8)$, so addition is a XOR, and multiplication is a polynomial multiplication modulo 2 (we'll come back on this operation later),
4. **AddRoundKey** : And a very easy step to finish : a simple xor with the sub-key for the round.

- **Warning** : The last of the 10/12/14 rounds does not apply the MixColumn step.

AES Decryption

Decryption goes through the same process, you have to create the same keys, just use them in reverse order, and apply each round in reverse order with inverse operations, which means :

- Creating the keys with the original S-box,
- The ByteSub step is done with the inverted S-box :

left (high-order) nibble \ right (low-order) nibble	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

Figure 3: AES S-Box Inverse

- Shifting rows right instead of left,
- An inverted MixColumn matrix :

$$\begin{pmatrix} (0E)_{16} & (0B)_{16} & (0D)_{16} & (09)_{16} \\ (09)_{16} & (0E)_{16} & (0B)_{16} & (0D)_{16} \\ (0D)_{16} & (09)_{16} & (0E)_{16} & (0B)_{16} \\ (0B)_{16} & (0D)_{16} & (09)_{16} & (0E)_{16} \end{pmatrix}$$

- And the inverse of AddRoundKey is itself, since it's a xor.
- And of course, you need to reverse the order of operations in each round, and finish with the reverse initial step.

Reminder : Polynomial Operations

In $GF(2^8)$, addition is just a bitwise XOR.

Multiplication is a more complicated matter though. Let's say we have p_1, p_2 two polynomials. Their product in $GF(2^8)$ is computed as their standard product, modulus the irreducible polynomial used to define $GF(2^8)$ (in this case, it is Rijndael's finite field, defined by $r = x^8 + x^4 + x^3 + x + 1$).

So all you have to do is multiply p_1 and p_2, and then reduce it modulo r to a polynomial of degree strictly inferior to 8. We can do it by observing $x^8 + x^4 + x^3 + x + 1 \equiv 0$ means $-x^8 \equiv x^8 \equiv x^4 + x^3 + x^1 + 1$ (the first equivalence is because we work modulo two since we're in $GF(2^8)$). And of course, you have to remember that we're working modulo 2. For example, let's say $p_1 = x^6 + x^4 + x^3 + 1$ and $p_2 = x^5 + x^4 + x^2 + x$:

$$p_1 \cdot p_2 = x^6 \cdot p_2 + x^4 \cdot p_2 + x^3 \cdot p_2 + p_2$$
$$= (x^{11}+x^{10}+x^8+x^7)+(x^9+x^8+x^6+x^5)+(x^8+x^7+x^5+x^4)+(x^5+x^4+x^2+x)$$
$$= x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^2 + x$$

Now, we still have to apply the modulo :

$$= (x^3 + x^2 + x^1 + 1) \cdot (x^4 + x^3 + x + 1) + x^6 + x^5 + x^2 + x$$
$$= (x^7 + x^4 + x^3 + 1) + x^6 + x^5 + x^2 + x = x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1$$

Which means $p_1 = (01011001)_2$ and $p_2 = (00110110)_2$ have the product $p = p_1 \cdot p_2 = (11111111)_2$.

Padding, slicing and ECB

Now that we have an AES Box fully functional for encryption and decryption, we just need to format any message into 128 bit blocks, i.e. 16 bytes blocks.

For that, we will start by applying the following padding, then slicing the obtained padded message into 128 bit blocks (16 bytes), and then encrypting each one separately with the AES Box (and we obviously end by concatenating all cipher blocks to get the final cipher).

Padding

We add padding for every message (even if the message already has a size that's a multiple of 16 bytes), in the following way :
Let L be the length of our message in bytes (for example, "Hello" has length L=5 if we consider characters encoded on 8 bits). Padding is done by adding X bytes of padding with value X, such that $16 \geq X > 0$, and that $L+X = 16 \cdot k$, $k \in \mathbb{N}^*$.

Examples :

"Hello" will be followed by 11 bytes of value 11 (i.e. 00001011). We then have only one block to encrypt.
"Can you smell what the Rock is cooking?" (39 bytes) would be followed by 9 bytes of value 9 (00001001). We then have 48 bytes in total, which is three blocks of 16 bytes each.
"You can't see me" is exactly 16 bytes. We then add 16 bytes of padding (as we want to always pad) with value 16 (00010000). We'll end up with 32 bytes as two blocks of 16 bytes.

ECB mode and decryption

After padding, there's not much more to do : slice the obtained padded message into blocks of 128 bits (16 bytes), run each one separately in the AES Box, and you'll get the cipher (with as many 16 bytes blocks).

Decryption with ECB is simple, since each block is encrypted independently : just slice the ciphertext into 16 bytes block, apply the AES decryption Box on each, and then you have the padded message, which you just have to unpad (check the last byte, and if its value is X, get rid of the last X bytes) :

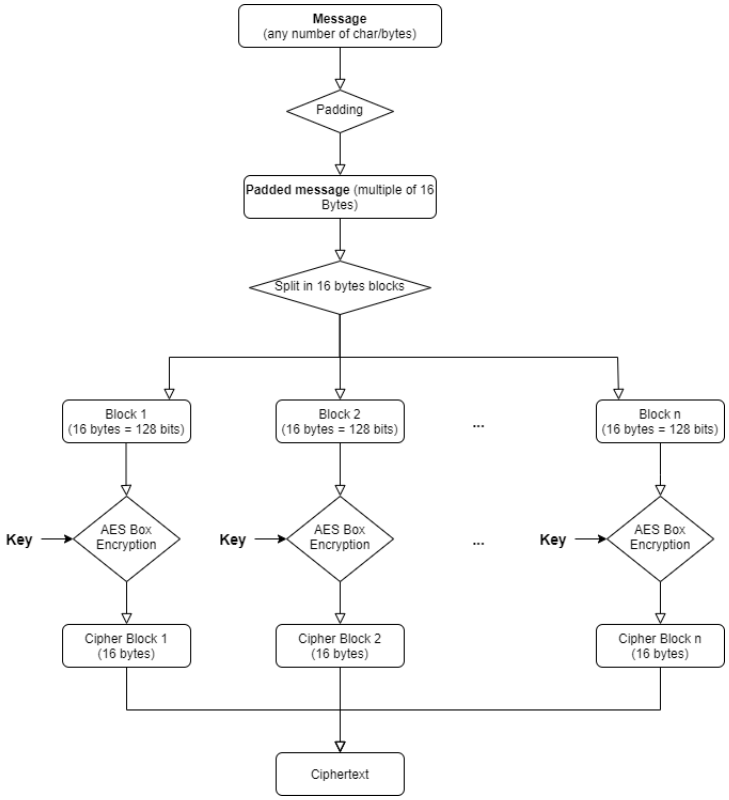

Figure 4: Electronic Codebook (ECB) encryption mode, with AES

TP : Implement AES Encryption and Decryption

You have all the information you need to implement a real version of AES with ECB mode with Python 3. Note that a file named **"Sboxes.py"** is on Moodle and contains the S-box and S-box inverse already, as two big tuples (you can obviously modify them if you prefer to work with another type of structure).

In short, here's what you need to do for encryption :

- Taking a message (as a string) in entry, and pad it as explained earlier.
- Slice it into 16 bytes blocks.
- Encrypt each block separately with AES. For that, you will need to create the whole AES Box described in this TP, including the sub-key generation, and the possibility to choose the length of the key (either 128, 192 or 256 bits).
- And then put cipher blocks back together to form the complete ciphertext.

For the decryption,same thing in reverse : slice the cipher in 16 bytes blocks, decrypt each one separately with your AES decryption Box (The dedicated section about decryption details how to change the AES box to decrypt instead of encrypt. It's mainly about reversing the order of operations and changing the matrices). Then put the plaintexts together to get the padded message. And finally unpad the message.

Help : A good debugging example

To debug your AES box, you can use this very useful document : `https://kavaliro.com/wp-content/uploads/2014/03/AES.pdf`.
It provides an complete example of what happens in the Box, with a 128 bit message and a 128 bit key, all in hexadecimal representation, what is the final ciphertext, and every one of the results after each step of the calculation.
Be careful though : that's just the encryption for one block : that is not what you would find if you encrypted this message (as it would be padded with a full block of padding, and thus would have a second block). Try with this message and key to see if your AES box if fully functional before testing for full messages after padding. And if your box doesn't work at first, this document should help in finding where the mistake is.

Finally, here's an example to check if the whole AES with ECB mode works :
If you encrypt "Can you smell what the Rock is cooking?",
With the key "You can't see me",
You should find the following results, given here in hexadecimal :
D69E09957672BB537F137948E9755D12
EA924C80079DA5B141A576D0142ED4C0

5C26547ACB217669F3C0291966BAFBE4
(The actual string that is printed by Python may change depending on the encoding, hence the hexadecimal verification).