Assignment 5. AES

Crypto basics 1

If by first subkey it was meant first subkey :)

sub bytes:
```
[['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16']]
```

shift rows:
```
[['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16']]
```

mix columns:
```
[['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16'],
['16', '16', '16', '16']]
```

add round key:
```
[['e9', 'e9', 'e9', 'e9'],
['e9', 'e9', 'e9', 'e9'],
['e9', 'e9', 'e9', 'e9'],
['e9', 'e9', 'e9', 'e9']
```

If initial:

subkey 1:
```
[['e8', 'e9', 'e9', 'e9'],
['17', '16', '16', '16'],
['e8', 'e9', 'e9', 'e9'],
['17', '16', '16', '16']]
```

sub bytes
```
[['63', '63', '63', '63'],
['63', '63', '63', '63'],
['63', '63', '63', '63'],
['63', '63', '63', '63']]
```

shift rows
```
[['63', '63', '63', '63'],
['63', '63', '63', '63'],
['63', '63', '63', '63'],
['63', '63', '63', '63']]
```

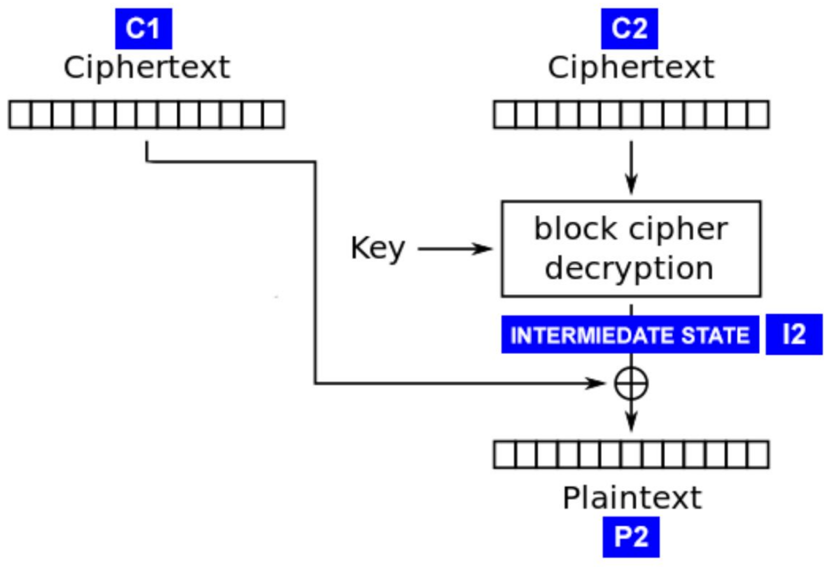
mix columns
```
 [['63', '63', '63', '63'],
 ['63', '63', '63', '63'],
 ['63', '63', '63', '63'],
 ['63', '63', '63', '63']]
```

add round key
```
[['8b', '8a', '8a', '8a'],
['74', '75', '75', '75'],
['8b', '8a', '8a', '8a'],
['74', '75', '75', '75']]
```

Crypto basics 2 [1]

“Padding Oracle Attack”

That is the scheme of CBC algorithm:

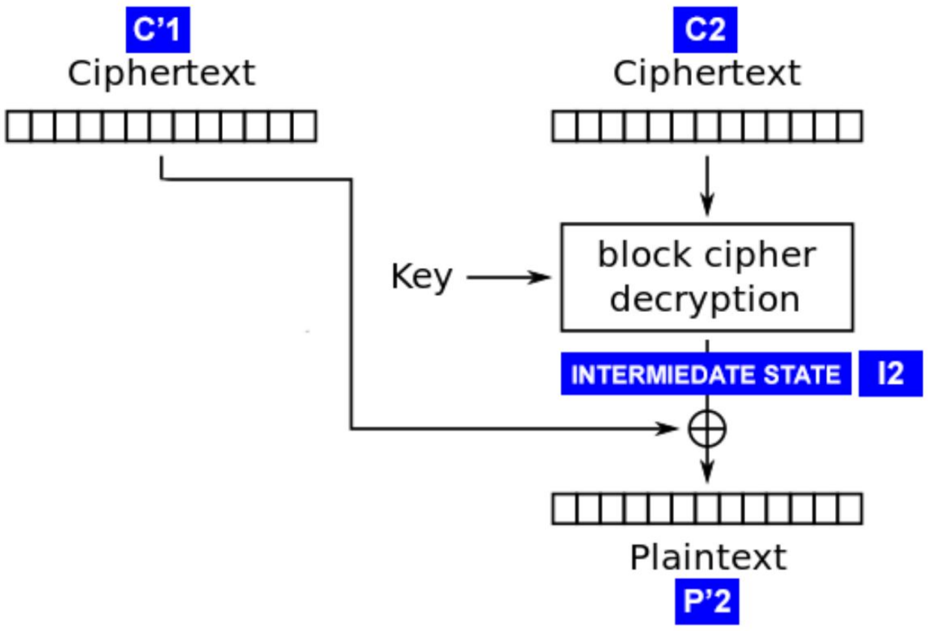


Because we operate with the blocks, our message should be a multiple of the block size, but that is uncomfortable, so we can just add padding at the end of the message.

The preferred method of padding block ciphertexts is PKCS7. In PKCS7 the value of each padded byte is the same as the number of bytes being added. So, If it is 14 characters in the message, there are 2 padding bytes with value of [02], if 15 characters, there is 1 padding byte with value of [01]. If it is 16 characters - an entire extra block of [16] * 16.

Server gets the ciphertext, encrypts it and checks if PKCS7 padding format is OK, and returns a message if it is OK or not (because if padding is not OK, maybe there are some mistakes and a client should retransmit a message)

This approach to make clients retransmit messages leads to a vulnerability. Let’s try to get the plaintext. Let’s create some random string C’1 and send C’1 + C2 as ciphertext to the server.



The server counts:
C1'[16] ^ I2[16] = P2'[16]

If P2’[16] appears to be 01 server returns “padding is correct” (because the padding format is OK).
So let’s bruteforce C1’[16] to get “padding is correct”.

After that we get I2[16] :
I2[16] = P2'[16]^C1'[16] = 01&C1’[16]

That is how we get P2[16] known I2[16] :

P2 [16]= C1[16] ^ I2[16]

What is next?

If P2[16] = 02 and P2[15]=02 server will return “padding is correct”

We can get such C1’[16] so that P2[16] = 02:
C1’[16] = 02 ^ I2[16]

Now we have to bruteforce C1’[15], if server returns “padding is correct” we found right C1’[15]

That is how we get I2[15] :
I2[15] =02^C1'[15] = 02&C1’[15]

That is how we get P2[15] known I2[15]:
P2 [15]= C1[15] ^ I2[15]

If P2[16] = 03 and P2[15]=03 and P2[14]=03 server will return “padding is correct”
C1’[16] = 03 ^ I2[16]
C1’[15] = 03 ^ I2[15]
=> Bruteforce C1’[14] until server will returns “padding is correct”, find next byte of the plaintext.

And so on, and so on, and so on.

However, there is one problem. It is true that if P2’[16] appears to be 01 server returns “padding is correct”. However, server returns “padding is correct” NOT ONLY when P2’[16] is 01, we should remember about that and consider different variants.

“CBC Byte Flipping Attack”
CBC byte flipping attack relies on the block cipher using CBC mode, where we XOR the previous ciphertext with the next plaintext.

This attack allows us to change the plaintext of a ciphertext without knowing the key (provided that, in most cases, we turn one previous block to garbage).

Suppose we need to change byte X to Y (in ascii) in some plaintext on position 2. What we need to do is to take the original byte of the ciphertext at same position but in a previous block, XOR it with X, and then with Y. After the first XOR we would have a zero byte in the target plaintext (because of how XOR works), and after the second XOR we have successfully changed X to Y!

The drawback of this is of course that we have garbled the whole plaintext for the previous block, but even this is not a problem, when we need to change the byte only in the first block, because then we alter the IV, not the ciphertext, and that results in no noticable problems with the resulting plaintext.

Crypto basics 3 1

- Compute the output of the first round of AES to the input W and the subkeys W0, . . . ,W7.

subkey 1

```
['a0', 'fa', 'fe', '17']
['88', '54', '2c', 'b1']
['23', 'a3', '39', '39']
['2a', '6c', '76', '05']
```

sub bytes

```
['e5', 'f3', '59', '47']
['34', 'e4', 'b5', '24']
['62', '68', '59', 'c4']
['01', '8a', '84', 'eb']
```

shift rows

```
['e5', 'e4', '59', 'eb']
['34', '68', '84', '47']
['62', '8a', '59', '24']
['01', 'f3', 'b5', 'c4']
```

mix_columns

```
['54', '36', '95', '44']
['13', '34', '86', '3e']
['3c', 'a2', '36', '3d']
['7d', 'fc', 'd4', 'd6']
```

add round key

```
['f4', 'cc', '6b', '53']
['9b', '60', 'aa', '8f']
['1f', '01', '0f', '04']
['57', '90', 'a2', 'd3']
```

- Compute the output of the first round of AES for the case that all input bits are zero (using the same key).

subkey key 1
```
['a0', 'fa', 'fe', '17']
['88', '54', '2c', 'b1']
['23', 'a3', '39', '39']
['2a', '6c', '76', '05']
```

sub bytes
```
['f1', 'f3', '59', '47']
['34', 'e4', 'b5', '24']
['62', '68', '59', 'c4']
['01', '8a', '84', 'eb']
```

shift rows
```
['f1', 'e4', '59', 'eb']
['34', '68', '84', '47']
['62', '8a', '59', '24']
['01', 'f3', 'b5', 'c4']
```

mix_columns
```
['7c', '22', '81', '78']
['13', '34', '86', '3e']
['3c', 'a2', '36', '3d']
['7d', 'fc', 'd4', 'd6']
```

add round key
```
['dc', 'd8', '7f', '6f']
['9b', '60', 'aa', '8f']
['1f', '01', '0f', '04']
['57', '90', 'a2', 'd3']
```

- How many output bits have changed?

  10

Crypto basics 4

The three key schedules known are for AES-128, AES-192, and AES-256.

The constants are:

* RC[8] = 0x80

  For this one, we are talking either about AES-128, or AES-192, but not about AES-256, because the latter does not use any constants with indices larger than 7.

* RC[10] = 0x36

  For this one, we are certainly talking about AES-128, because AES-256 does not use anything larger than 7, and AES-192 does not use anything larger than 8.

1) For the AES tasks this code was used https://github.com/bozhu/AES-Python (with some modifications for the pretty output)