MKC-NBS: Úkol č. 1

Kryptografie

Filip Paul

18.02.2022

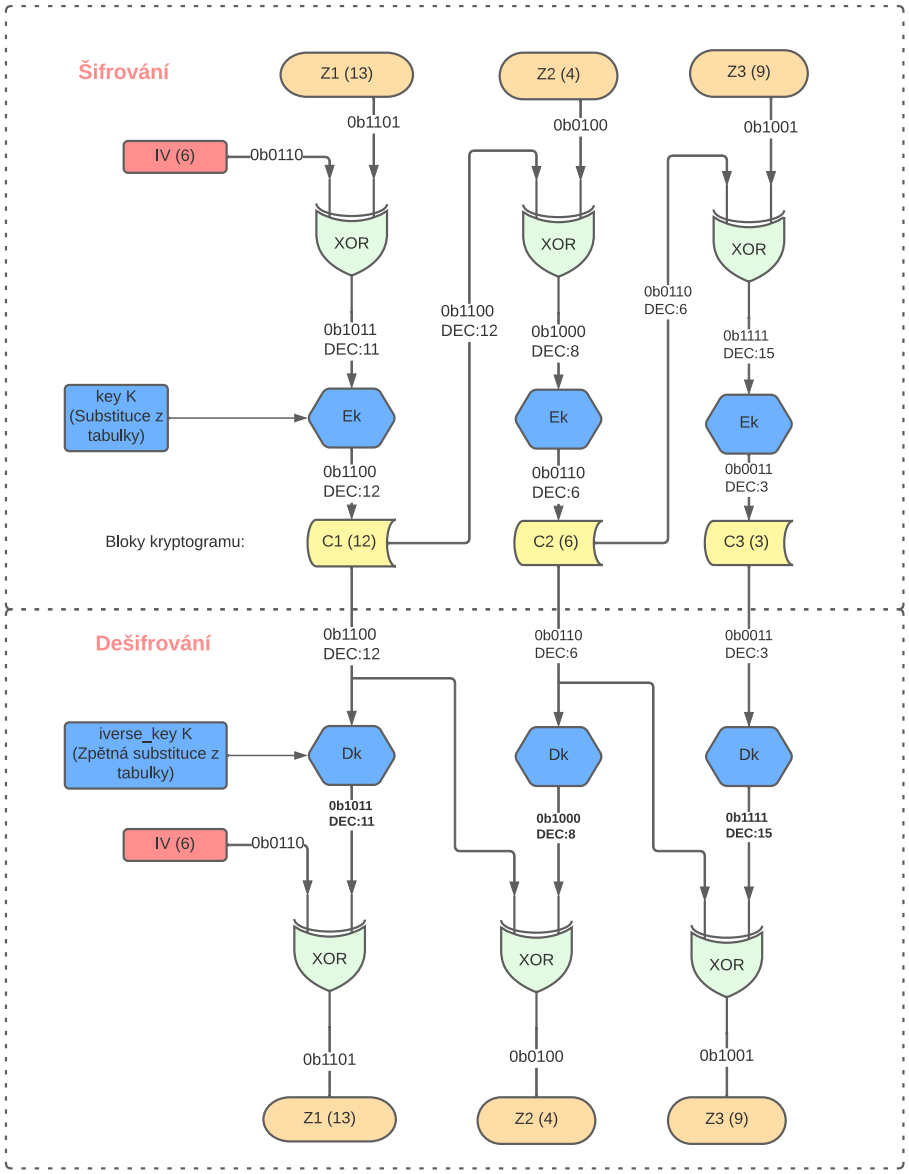
Bloková šifra v režimu CBC:

1. **Zadání:**
   Mějme zprávu Z = (13, 4, 9), kde jednotlivá čísla jsou bloky zprávy. Tuto zprávu zašifrujte v režimu CBC pro inicializační vektor IV = 6. Vypočítaný kryptogram pro kontrolu dešifrujte. Šifrování E a dešifrování D je dáno substitucemi podle tabulky 1. K provedení operací XOR si dekadická čísla převeďte na čtyřbitová čísla. Pro daný provozní režim nakreslete diagramy podle první přednáškové prezentace (snímek č. 21), přičemž v datových blocích schématu uveďte dekadicky i binárně hodnotu příslušného vstupu, či výstupu.

Table 1: Šifrovací substituce y = E(x,K)

X	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Y	4	10	9	2	13	8	0	14	6	11	1	12	7	15	5	3

2. **Vypracování:**
   Na následujícím obrázku jsou vyznačeny stavy v decimální i binární podobě pro každý "stupeň" blokové šifry. Tyto hodnoty byly vypočítány pomocí python scriptu přiloženého na konci tohoto pdf souboru. Nicméně pro lepší zobrazení scriptu můžete využít link na můj github repozitář → PYTHON CBC.py



Výpočet pečeti HMAC

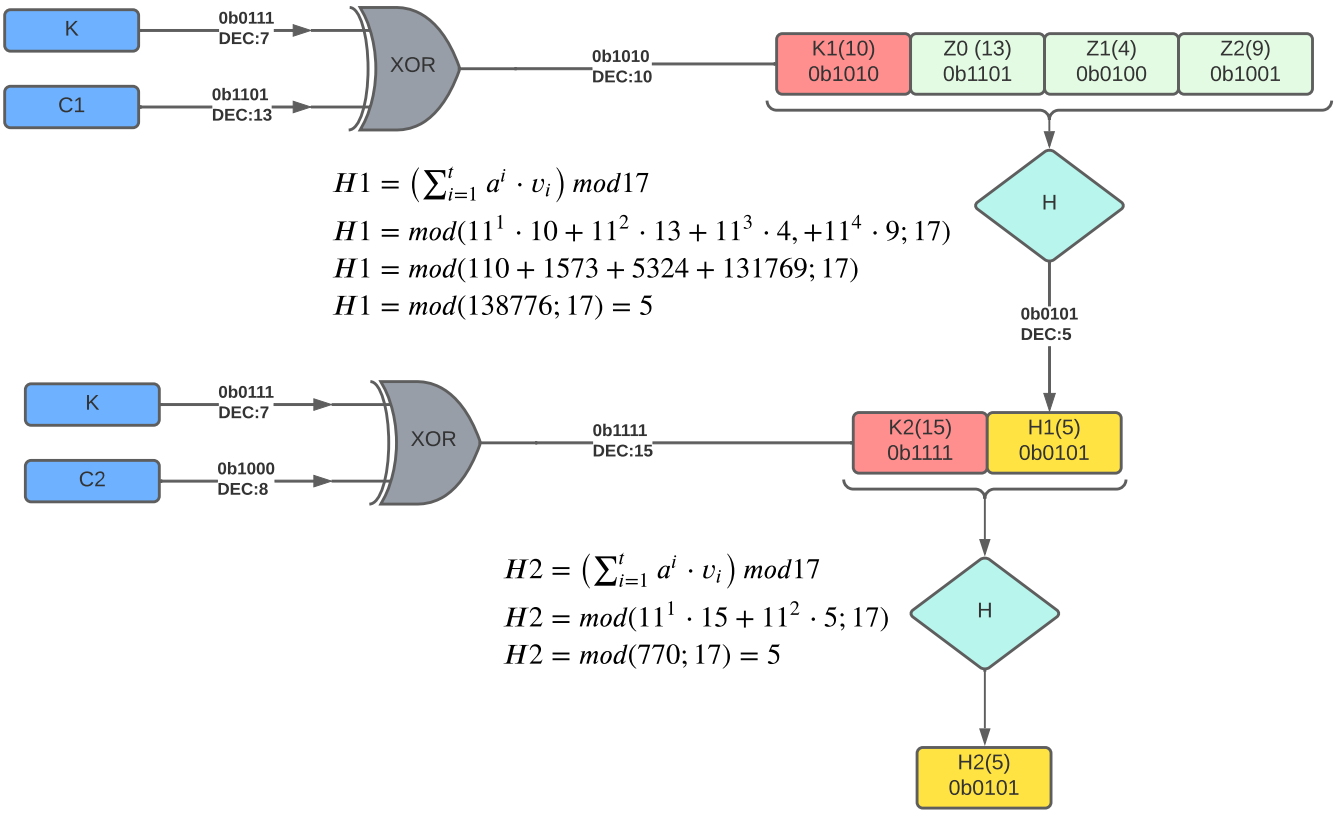
1. **Zadání:**
   Mějme zprávu Z = (13, 4, 9), kde jednotlivá čísla jsou bloky zprávy. Pro tuto zprávu vypočítejte technikou HMAC pečeť P. Pečetící klíč K = 7, konstanta $C_1 = 13$ a $C_2 = 8$. K provedení operací XOR si dekadická čísla převeďte na čtyřbitová čísla. Hešovací funkce H je definována následovně:

   $$h = \left(\sum_{i=1}^{t} a^i \cdot v_i\right) mod 17$$

   kde hešovací konstanta a = 11, v_i je i-tý blok hešovaného vstupu a t počet bloků na vstupu. V prvém hešování tedy bude t = 4, protože první blok je výsledek xorování klíče a konstanty a další bloky jsou bloky zprávy. Ve druhém hešování bude t = 2. Pečetí P je výstup z druhého hešování, tj. P = h_2.

2. **Vypracování:**

   Podobně jako v předcházejícím úkolu byla úloha řešena pomocí python scriptu. Na následujícím obrázku jsou znázorněny jednotlivé "stavy" heshovací funkce HMAC v binární i decimální podobě.



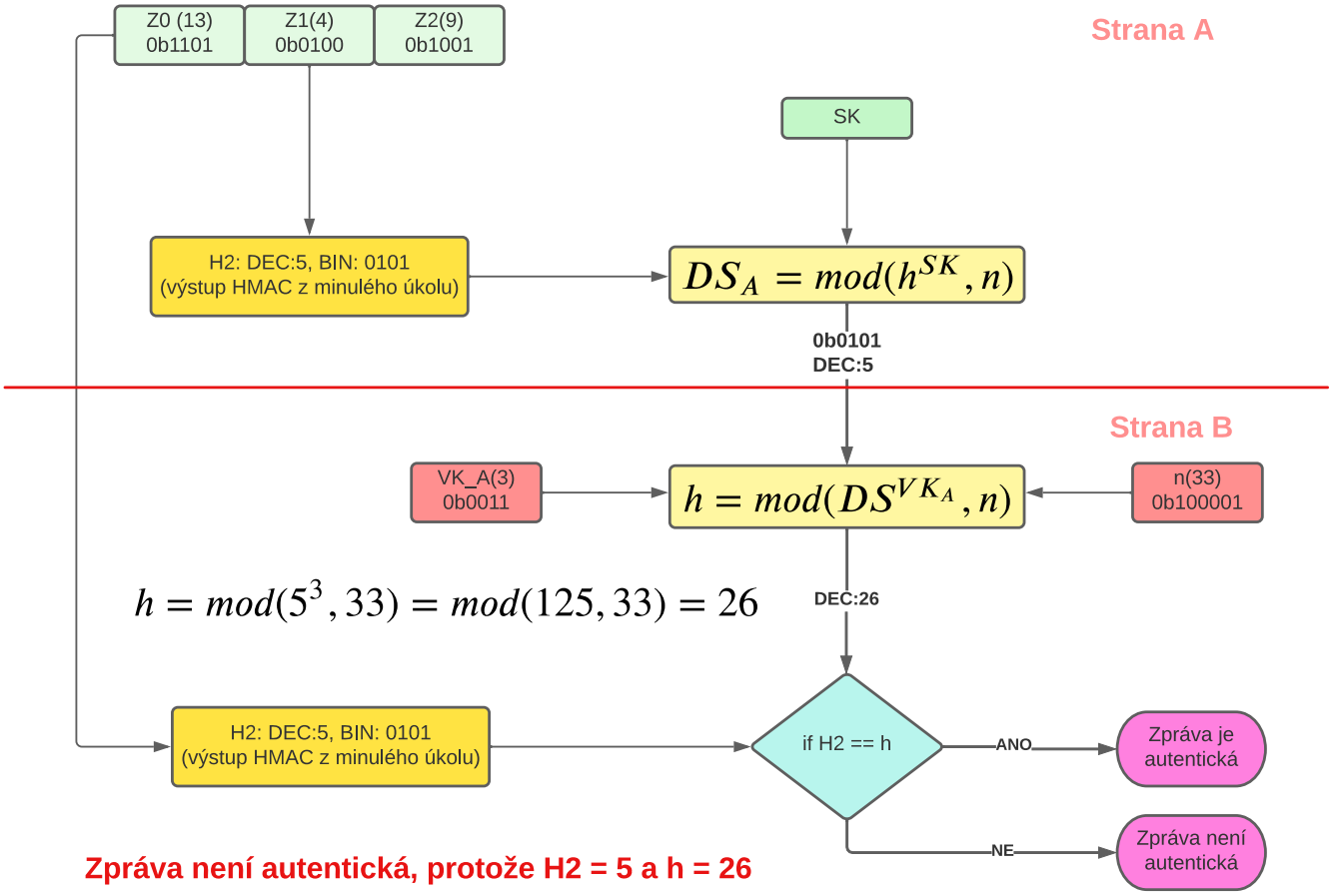
$$H1 = \left(\sum_{i=1}^{t} a^i \cdot v_i\right) mod 17$$
$$H1 = mod(11^1 \cdot 10 + 11^2 \cdot 13 + 11^3 \cdot 4, +11^4 \cdot 9; 17)$$
$$H1 = mod(110 + 1573 + 5324 + 131769; 17)$$
$$H1 = mod(138776; 17) = 5$$

$$H2 = \left(\sum_{i=1}^{t} a^i \cdot v_i\right) mod 17$$
$$H2 = mod(11^1 \cdot 15 + 11^2 \cdot 5; 17)$$
$$H2 = mod(770; 17) = 5$$

RSA podpis

1. **Zadání:**
   Byla Vám doručena zpráva Z = (13, 4, 9), jejíž RSA podpis DS = 5. Ověřte, zda je tato zpráva autentická. Znáte veřejný ověřovací klíč udávaného autora VK = 3, jeho modulus n = 33 a víte, že byla použita hešovací funkce H ze 2. příkladu.

2. **Vypracování:**
   Z následujícího obrázku je doufám patrný postup ověření podpisu. Výsledkem procesu je, že podpis nebyl přijat. Jinak řečeno zpráva je nevěrohodná.



Přiložené soubory

1. **CBC.py**

```python
def formatInputs(num)->str:
    """this function formats input number (in this example 4) into the following string:
    (DEC:4, BIN:0100)"""
    return f"(DEC:{num}, BIN:{bin(num)[2:].zfill(4)})"

message_blocks = [13,4,9] #input message blocks
table_output = [4,10,9,2,13,8,0,14,6,11,1,12,7,15,5,3] #Encryption substitution table
initial_vector = 6 #IV

########################## ENCRYPTION##############################################
output_cryptograms_list = [] #help variable to store cryptograms
input_encryption_list = [] #help variable to store inputs for encryption function (output of XOR)
i = 0
for blocks in message_blocks: #for each message block
    if i == 0:#first XOR has one input in form of initial vector, therfore this condition..
        print(f"xor_{i+1}: init_vect{formatInputs(initial_vector)} XOR Z_{i}{formatInputs(blocks)}")
        input_encryption_list.append(blocks^initial_vector)#output of XOR

    else:
        print(f"xor_{i+1}: init_vect{formatInputs(output_cryptograms_list[-1])} XOR Z_{i}{formatInputs(blocks)}")
        input_encryption_list.append(blocks^output_cryptograms_list[-1])#output of XOR

    print(f"Encription_input_{i+1}: {formatInputs(input_encryption_list[-1])}")
    #XOR output serves as pointer to the item of encryption table_output list
    output_cryptograms_list.append(table_output[input_encryption_list[-1]])
    print(f"Cryptogram_{i+1}: {formatInputs(output_cryptograms_list[-1])}")
    i+=1

########################## DECRYPTION ##############################################
i = 0
for cryptograms in output_cryptograms_list:
    print(f"Output of Dk_{i+1}: {formatInputs(table_output.index(cryptograms))}")
    if i == 0:
        print(f"Decrypted message block_{i+1}: {formatInputs(table_output.index(cryptograms)^initial_vector)}")
    else:
        print(f"Decrypted message block_{i+1}: {formatInputs(table_output.index(cryptograms)^output_cryptograms_list[i-1])}")
    i+=1
```

Output:
xor_1: init_vect(DEC:6, BIN:0110) XOR Z_0(DEC:13, BIN:1101)
Encription_input_1: (DEC:11, BIN:1011)
Cryptogram_1: (DEC:12, BIN:1100)
xor_2: init_vect(DEC:12, BIN:1100) XOR Z_1(DEC:4, BIN:0100)
Encription_input_2: (DEC:8, BIN:1000)
Cryptogram_2: (DEC:6, BIN:0110)
xor_3: init_vect(DEC:6, BIN:0110) XOR Z_2(DEC:9, BIN:1001)
Encription_input_3: (DEC:15, BIN:1111)
Cryptogram_3: (DEC:3, BIN:0011)

2. **HMAC.py**

```
1
2  def hashFunction(input):
3      a = 11#constant
4      t = int(len(input)/4) #count of blocks with length 4bits
5      sum_result = 0 #variable to sum
6      list_of_blocks_decimal = []
7      number = 0
8      for i in range(len(input)):#convert list of binary such as: [0,1,0,1,0,0,1,1]
9          #into list of 4 bit integers [0,1,0,1,0,0,1,1] -> [5,3]
10         number += input[i]* pow(2,3-((i)%4))
11         if ((i+1) % 4) == 0:
12             list_of_blocks_decimal.append(number)
13             number = 0
14
15     #computing hash sum:
16     for i in range(t):
17         sum_result += pow(a,i+1)* list_of_blocks_decimal[i]
18     result = sum_result%17 #modulo 17
19     return result
20
21 def HMAC(message_blocks,seal_key,Constatnt_C1,Constatnt_C2,block_size):
22     K_1 = Constatnt_C1^seal_key #XOR key with constant to get new keys
23     K_2 = Constatnt_C2^seal_key #XOR key with constant to get new keys
24
25     #K_1_list in for of [MSB, , ,LSB]
26     K_1_list_of_bits =  [int(i) for i in bin(K_1)[2:].zfill(4)]
27     #EXTEND message blocks to the K1 key
28     for blocks in message_blocks:
29         message_list_of_bits = [int(i) for i in bin(blocks)[2:].zfill(4)]
30         K_1_list_of_bits.extend(message_list_of_bits)
31
32     h_1 = hashFunction(K_1_list_of_bits) #HASH K1|Z(0)|Z(1)|Z(2)
33
34     #create blocks in form of K2|h1
35     h_1_list =  [int(i) for i in bin(h_1)[2:].zfill(4)]
36     K_2_list_of_bits =  [int(i) for i in bin(K_2)[2:].zfill(4)]
37     K_2_list_of_bits.extend(h_1_list)
38
39     h2 = hashFunction(K_2_list_of_bits) #HASH K2|h1
40
41     print(f"K1: (DEC:{K_1}, BIN: {bin(K_1)[2:].zfill(4)})")
42     print(f"K2: (DEC:{K_2}, BIN: {bin(K_2)[2:].zfill(4)})")
43     print(f"h1: (DEC:{h_1}, BIN: {bin(h_1)[2:].zfill(4)})")
44     print(f"h2: (DEC:{h2}, BIN: {bin(h2)[2:].zfill(4)})")
45     return h2
46
47 message_blocks = [13,4,9] #input message blocks
48 seal_key =7
49 Constatnt_C1 = 13
50 Constatnt_C2 = 8
51 block_size = 4 #constants are 4 bit long.. 13 = 0b1101
52
53 HMAC(message_blocks,seal_key,Constatnt_C1,Constatnt_C2,block_size)
```

Output:
K1: (DEC:10, BIN: 1010)
K2: (DEC:15, BIN: 1111)
h1: (DEC:13, BIN: 1101)
h2: (DEC:3, BIN: 0011)