Documentație: Tema 1, Securitatea Informației

Mediul de lucru utilizat:

Tema a fost realizată utilizând limbajul de programare Python, versiunea 3.8.3.

Pentru utilizarea modulului de algoritmului de criptare AES, a fost utilizat modulul criptografic pycrypto, care poate fi adăugat alături de orice versiunea de Python şi care pune la dispoziţie un API cu diferite funcţii criptografice.

Funcţionalitatea aplicaţiei a fost testată folosind o maşină virtuală care rulează sistemul de operare Linux, cu o arhitectură 32-bit.

Modul de rezolvare al cerinţei:

Nodul A, are comportamentul unui client care se conecteaza folosind socket-uri, atât la nodul B, cât şi la nodul KM(Key Manager). Am ales aceast comportament pentru nodul A, deoarece am considerat că el este iniţiatorul comunicării, atât cu nodul B, cât şi cu KM. Nodul B, are atât comportament de server, cât şi de client, mai exact, primeşte şi procesează mesaje de la nodul A, drept server, dar se conectează şi comunică cu nodul KM(Key Maneger), ca un client al acestuia. Nodul KM este doar un server, care aşteaptă moduri de operare de la clienţi, pentru a le putea transmite cheile necesare.

În primul pas în nodul A se cere introducerea de la tastatură a modului de operare, dar şi numele fişierului care urmează să fie criptat. Dacă modul de implementare introdus de la tastatură este unul valid(ECB sau OFB), nodul A, se conectează la cele două servere B şi KM şi le transmite acest mod de operare care va fi folosit în etapele următoare.

Toate cele 3 noduri folosesc thread-uri prin care aşteaptă primirea de mesaje de la celelalte noduri implicate în sistem.

La primirea acestui mesaj nodul KM, returnează nodului A o cheie criptată respectiv cheia K1, pentru modul de operare ECB şi cheie K2 pentru modul de operare OFB, ambele cheie de criptează folosind modul de operare corespunzător şi cheia K3, cheie pe care o cunosc toate cele 3 noduri. Cheia K1, se criptează folosind AES, K3 şi modul ECB fără se mai împărţim pe mai multe blocuri, deoarece ştim din enunţarea problemei că K1 are dimensiunea fixă, egală cu dimensiunea unui bloc din modul de criptare ECB. Analog pentru cheie K2, se foloseşte AES, K3, un vector de iniţializare cunoscut de toate nodurile şi o implementare proprie a modului de criptare OFB, pentru un singur bloc. Implementarea proprie a modului OFB poate fi descrisă astfel, criptăm folosind AES vectorul de iniţializare cu cheia K3, apoi facem operaţia de XOR

între fiecare byte din vectorul de iniţializare şi cheie K2, pe care dorim să o criptăm pentru a o trimite nodurilor care au nevoie de ea.

Aceeaşi paşi de realizează şi în urma primirii modului de operare de la nodul B.

În acest punct ambele noduri, A şi B, au o cheie criptată K1 sau K2, primită de la nodul KM.

Cele două cunosc modul de operare utilizat la criptare, K3(cheie folosită la criptare) şi respectiv vectorul de iniţializare(necesar pentru modul OFB). Cunoscând aceste lucruri pot afla cheile originale. Pentru acest lucru în fiecare nod există câte o funcţie de decriptare pentru fiecare mod. Pentru modul ECB, se face o decriptare AES pentru un singur bloc(ştim că avem chei de dimensiunea fixă a unui bloc), acelaşi lucru şi pentru OFB, criptăm vectorul de iniţializare cu cheie K3 şi apoi facem operaţia de xor, între vectorul de iniţializare şi textul criptat primit, care reprezintă criptarea pentru cheie K2.

După decriptarea cheilor nodul A, trimit mesajul de start la nodul B, din acest punct se începe trimiterea mesajelor criptate de la A la B.

Acest lucru se realizează în modul următor: în modul A se citeşte conţinutul fişierului dat se la tastatură la început. Se face padding acestuia în cazul în care dimensiunea nu este un multiplu de 16(128 biti). Padding-ul se realizează astfel, adăugat textului atâţia bytes de 0, cât sunt necesari pentru a ajunge la primul multiplu al lui 16 care urmează după lungimea curentă a textului.

```
def pad(text):
    if len(text) % 16 == 0:
        return text
    data = text + (16 - (len(text) % 16)) * b'\x00'
    return data
```

Dacă modul de criptare este ECB, se împarte textul căruia i s-a făcut padding în blocuri de câte 16 octeti(128 biti). Şi fiecare text se criptează cu AES şi este trimis nodului B, care primeşte fiecare bloc în parte şi îl decriptează tot folosind modul ECB. În implementarea curentă am ales să afişez atât cheile criptate şi decriptate, cât şi versiunea criptată a textului, doar în scop didactic, într-un sistem real am vedea doar rezultatul decriptarii în final.

```
def encrypt_text_ECB(text, key, s):
    text = pad(text)
    cipher = AES.new(key.encode('utf8'), AES.MODE_ECB)
    for bloc in range(len(text) // 16 ):
        block_enc = cipher.encrypt(text[bloc * 16:(bloc + 1) * 16])
        s.sendall(codecs.encode(block_enc, 'base64'))
```

Analog pentru modul OFB, fac pad, împart textul în câte 16 octeti(128 de biti), şi criptez folosind modul OFB. Criptez vectorul de iniţializare cu cheie K2 şi fac xor între vector şi text, noul vectorul de iniţializare astfel obtinut în urma criptării cu cheia, îl voi utiliza ca vector de iniţializare pentru blocul următor. Nodul B primeşte blocurile pe rand şi le decriptează în modul

următor: foloseşte acelaşi vector de iniţializare iniţial ca şi nodul A, îl criptează cu K2 din modul B, face xor între el şi textul criptat primit şi afişează rezultatul. Vectorul de initializare obtinut în urma criptării este folosit ca vector de iniţializare pentru blocul următor.

```
def XOR(input_bytes, key_input):
    index = 0
    output_bytes = b''
    for byte in input_bytes:
        if index >= len(key_input):
            index = 0
        output_bytes += bytes([byte ^ key_input[index]])
        index += 1
    return output_bytes

def encrypt_text_OFB(text, key, s):
    global iv
    text = pad(text)
    cipher = AES.new(key.encode('utf8'), AES.MODE_ECB)
    iv = '0' * 16
    for b in range(len(text) // 16 ):
        iv = cipher.encrypt(iv)
        enc = XOR(text[b * 16: (b + 1) * 16], iv)
        s.sendall(codecs.encode(enc, 'base64'))
```

După terminarea trimiterii mesajului, se cere iar introducerea unui mod de comunicare şi a unui fisier de intrare şi se continua comunicare folosind acelaşi procedeu descris mai sus.

Testele realizate:

Testele realizate au avut în special în vedere urmărirea criptării cu succes atât pentru fişiere de diverse dimensiuni, mai exact, texte de dimensiuni egale multiplii ai numărului 16, pentru care nu este necesar padding-ul, sau texte de dimensiuni chiar egale cu 16, care au drept urmare trimiterea unui singur bloc, şi nu în ultimul rând texte de dimensiuni care nu sunt egale cu multplii e lui 16 şi nici egale cu el, fişiere pentru care e necesar padding-ul şi separarea în blocuri.

Am urmărit să văd modul în care rezultatul final ar putea fi influenţat de padding şi împărţirea pe blocuri, urmârind mai exact modul în care arată blocurile criptate, am observat astfel că rezultatul criptării rămâne în continuare corect, chiar şi cu aceste “artificii” necesare criptarii în cele două moduri.

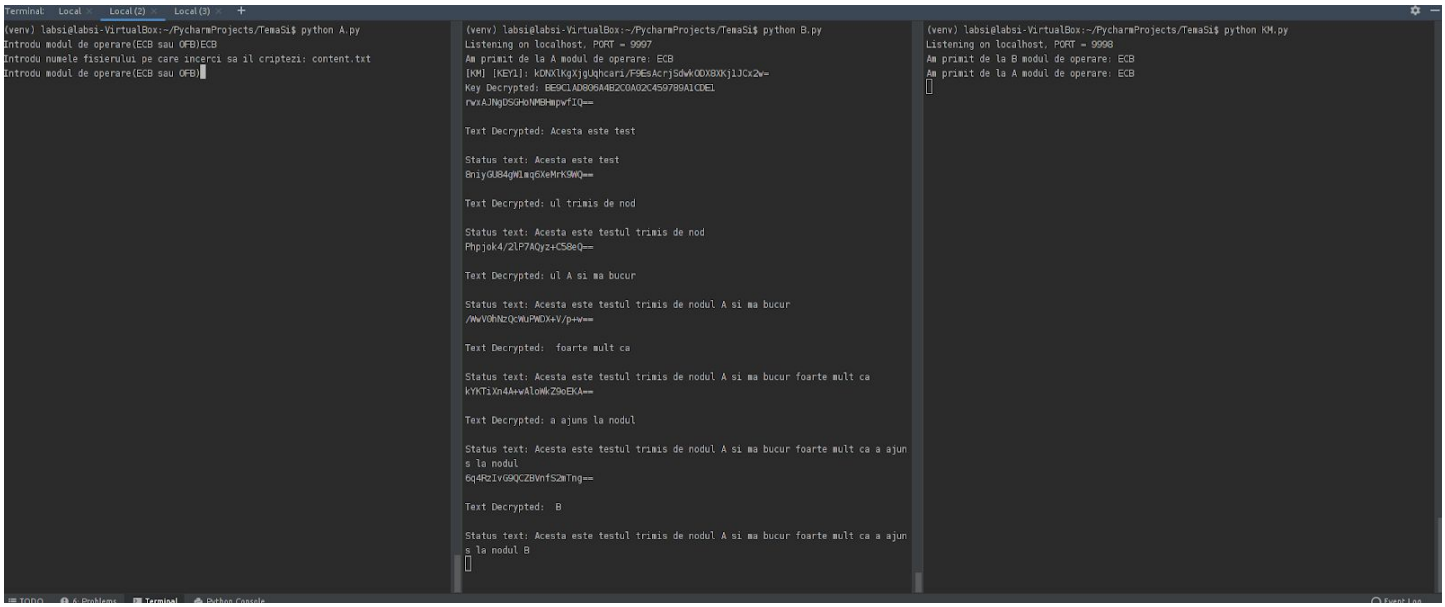

Fig 1.1 Exemplu de test ECB

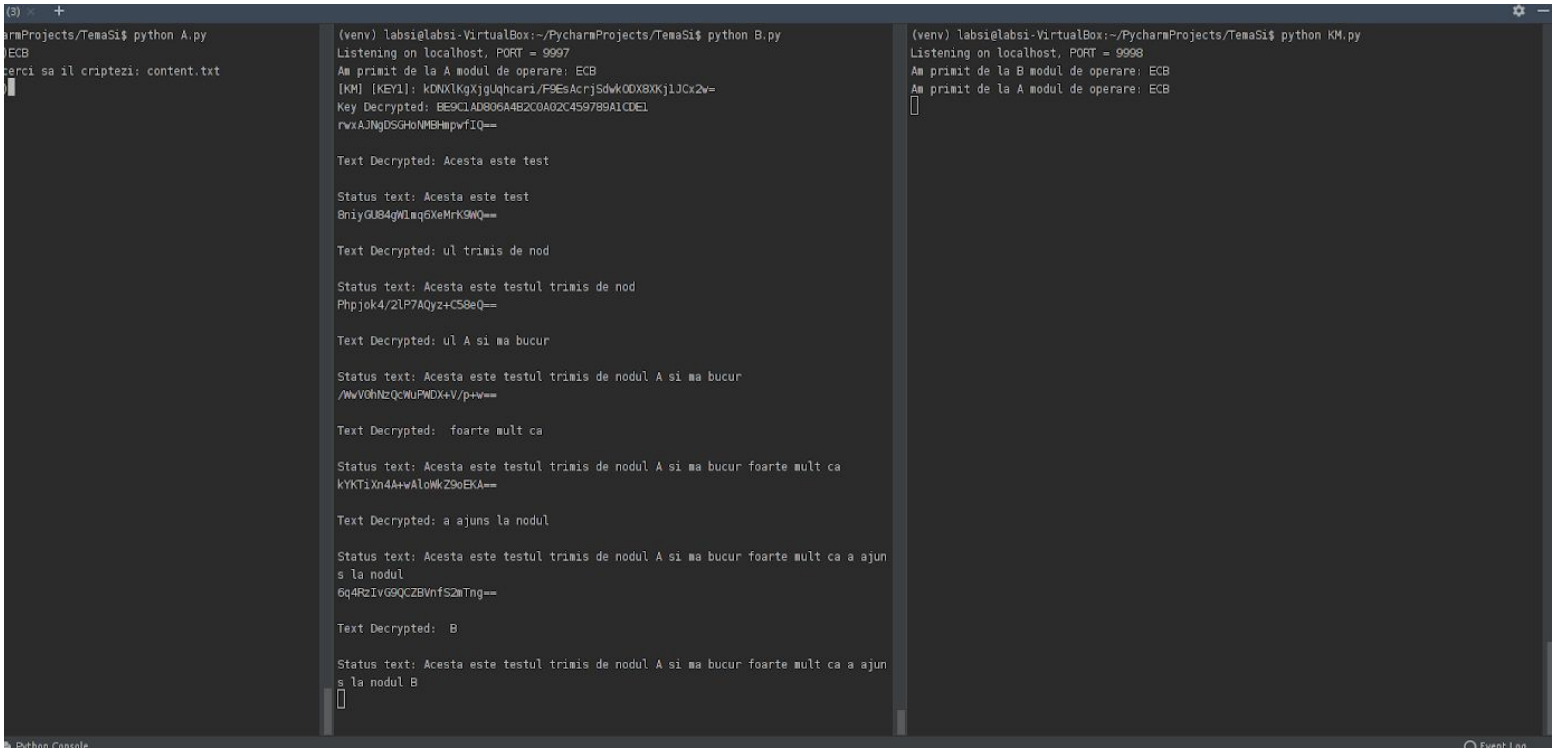

Fig 1.2 Exemplu de test OFB

Bibliografie

https://stackoverflow.com/questions/52356956/python-implement-aes-in-cbc-encryption-mode
https://gist.github.com/chrcoe/9667052
https://www.journaldev.com/15906/python-socket-programming-server-client