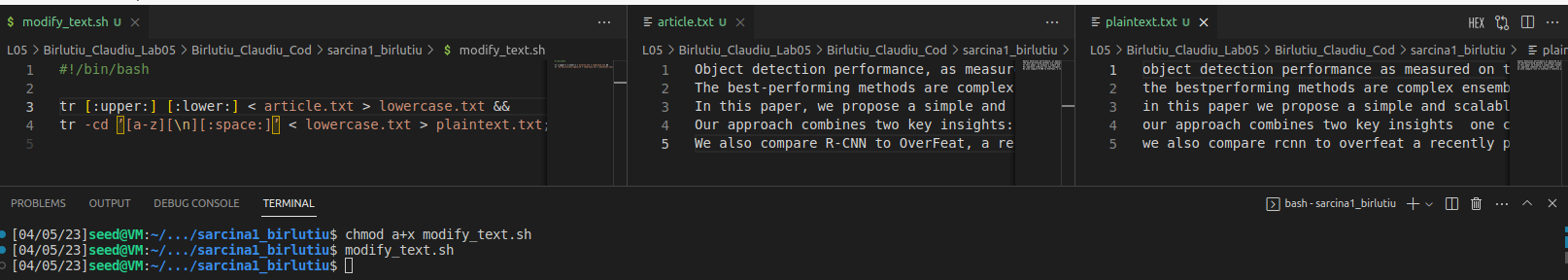
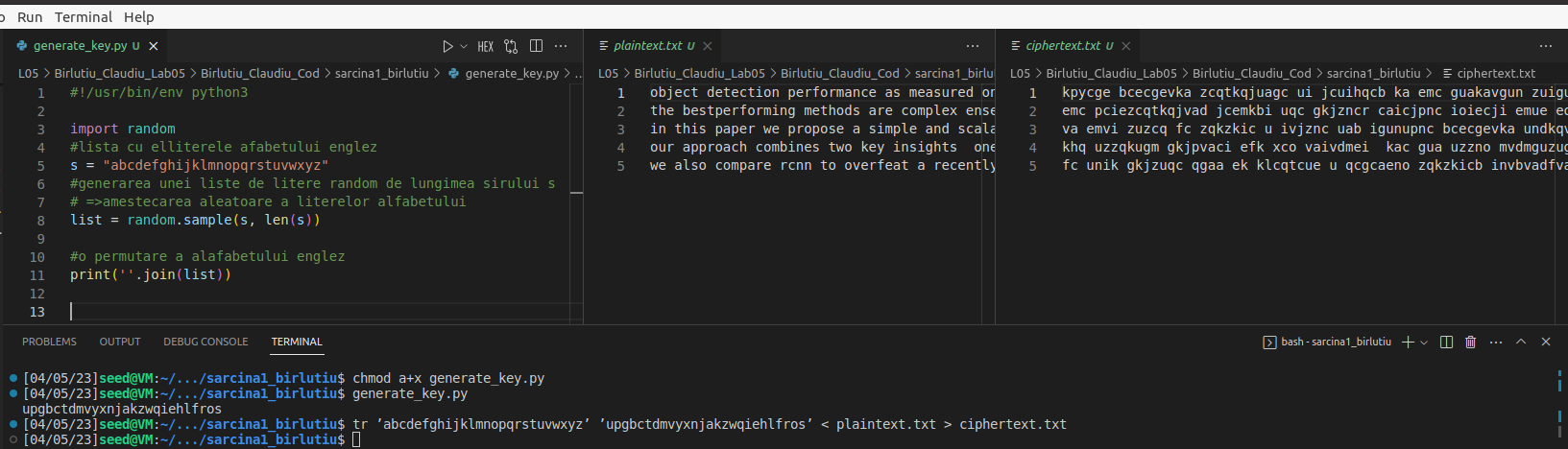
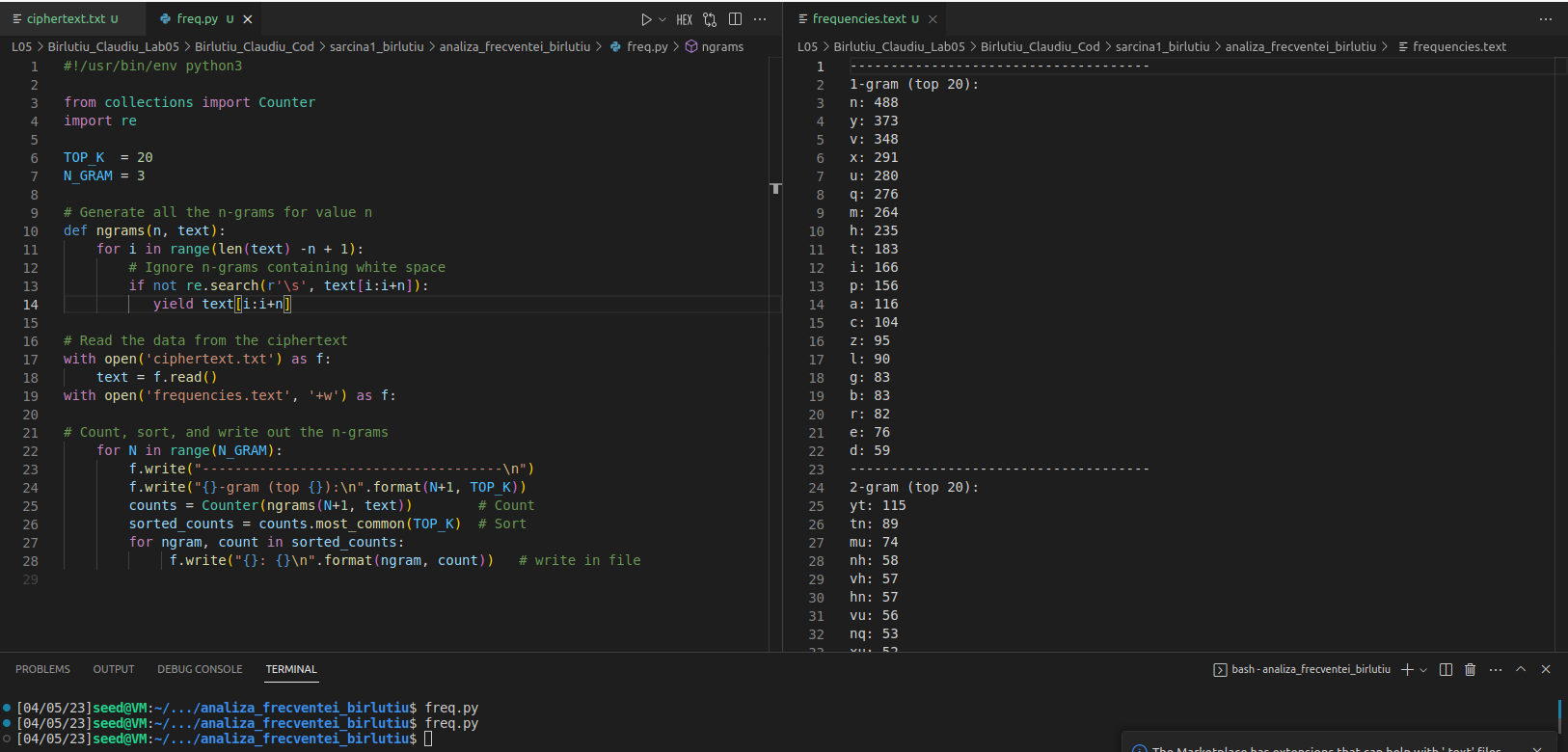
# Raport pentru lucrarea 5: Cifrarea cu cheie secretă

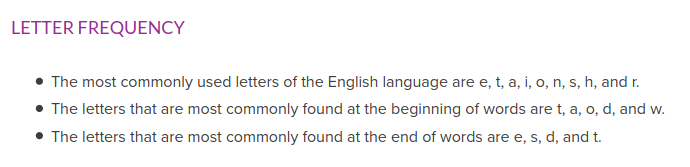
Autor: Birlutiu Claudiu-Andrei, gr 30643

## Sarcina 1: Analiza frecvenței împotriva unui cifru de substituție monoalfabetic

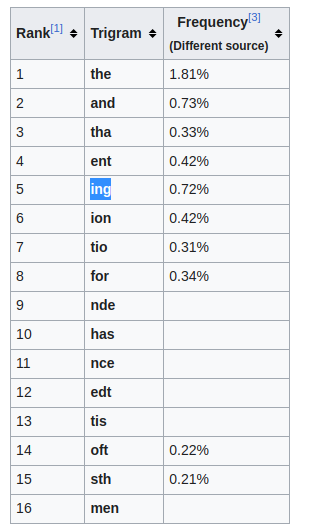
* În continuare am urmat tuorialul de criptare:
  + in prima faza am creat un script bash care ia cifrul și îl transforma în plain text (fără semene de punctuatie și litere mici) – fișierul se numește **modify\_script.sh** și i s-a dat permisiuni de execuție; se observa ca din textul article.txt s-a generat plaintext.txt care reprezintă textul fără cemne de punctuatie și lowercase
  + am creat un script python care genereaza o cheie aleatoare de mapare a literelor alfabetului la o alta litera și am făcut inlocuirile pe plaintext.txt => cipher.txt

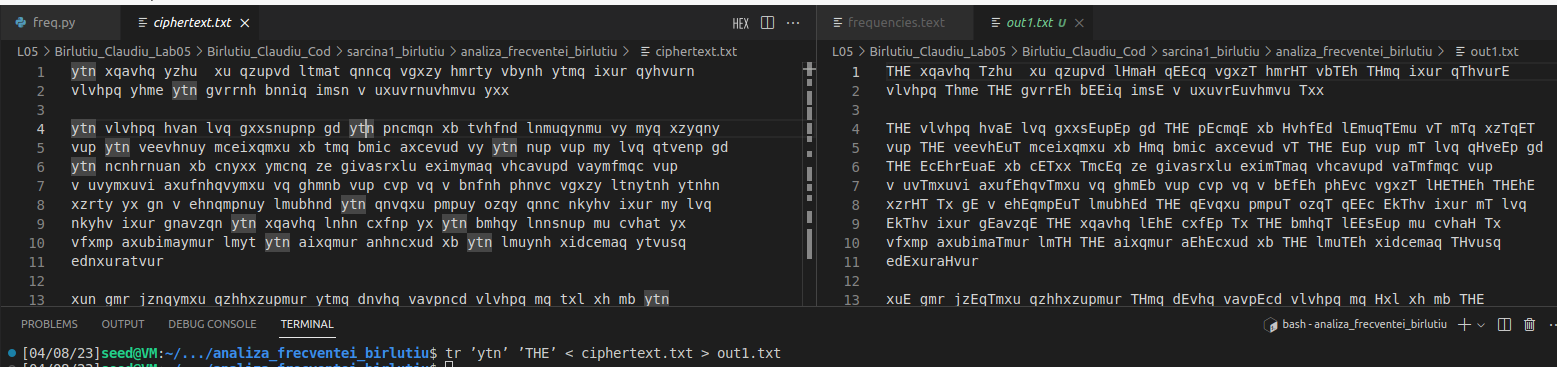


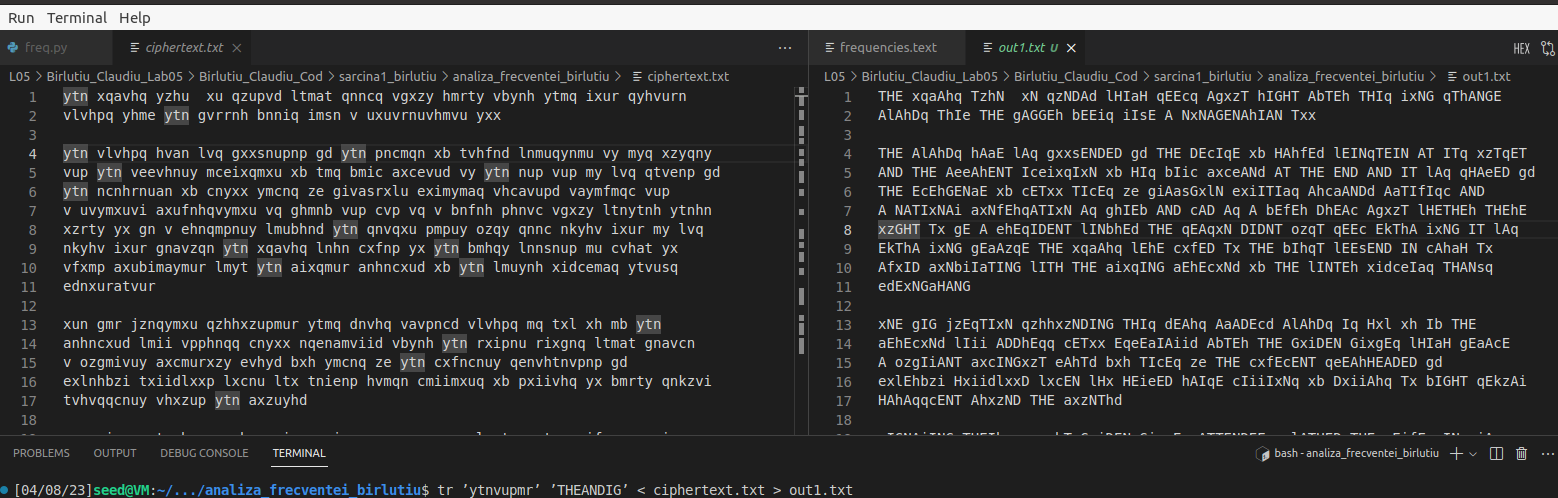
* Am trecut la descifrarea textului criptat din lucrare. Scopul a fost sa analizez frecventa literelor sau grupurilor de litere astfel încât sa găsesc cheia de criptare a mesajului
* am modificat programul python de analiza a textului sa scrie într-un fisier frecventele
* am căutat pe internet care sunt cele mai frecvente litere din limba engleza



* am încercat următoarea analiza în ceea ce privește n-gramele generate: din sursa <https://en.wikipedia.org/wiki/Frequency_analysis> înțelegem ca the – este cea mai comuna tri gram din limba engelza, th- cea mai comuna bi-gram și e este cea mai folosită litera

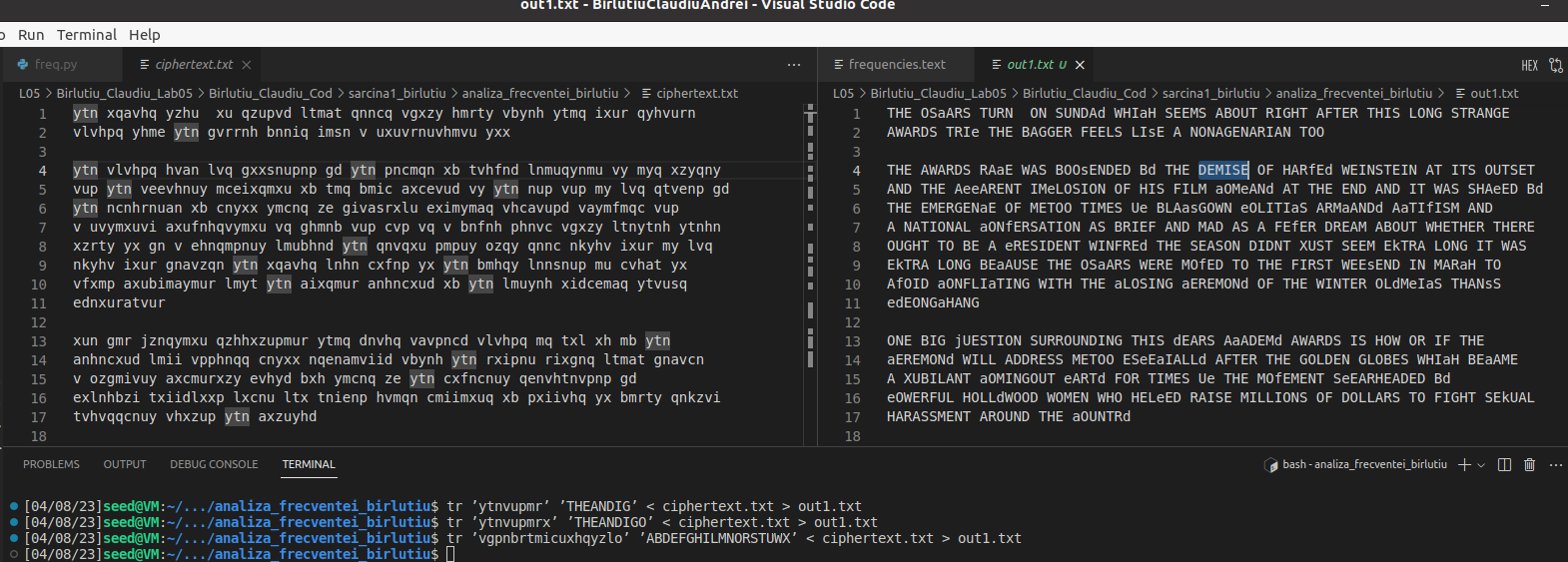


* în urma analizei a reiesit faptul ca n apare de cele mai multe ori (488); yt – apare de asemenea de cel mai multe ori și ytn – apare de asemenea de cele mai multe ori => vom inlocui astfel **y** cu **t**, **n** cu **e** și **t** cu **h**
* **and** este de asemenea o triagrama detsul de întâlnită în textul englezesc, iar la noi pe a doua poziție a triagramelor regasim vup =? încercam astfel sa inlocuim v cu a, u cu n și p cu d
  + de asemenea **ing** este o trigrama frecventa -> iar la noi **mur** (a 3-a cea mai frecventa) s-ar potrivi cu ing având în vedere supozitia anterioara (
  + v = a, u = n , p = d, m = i , r = g



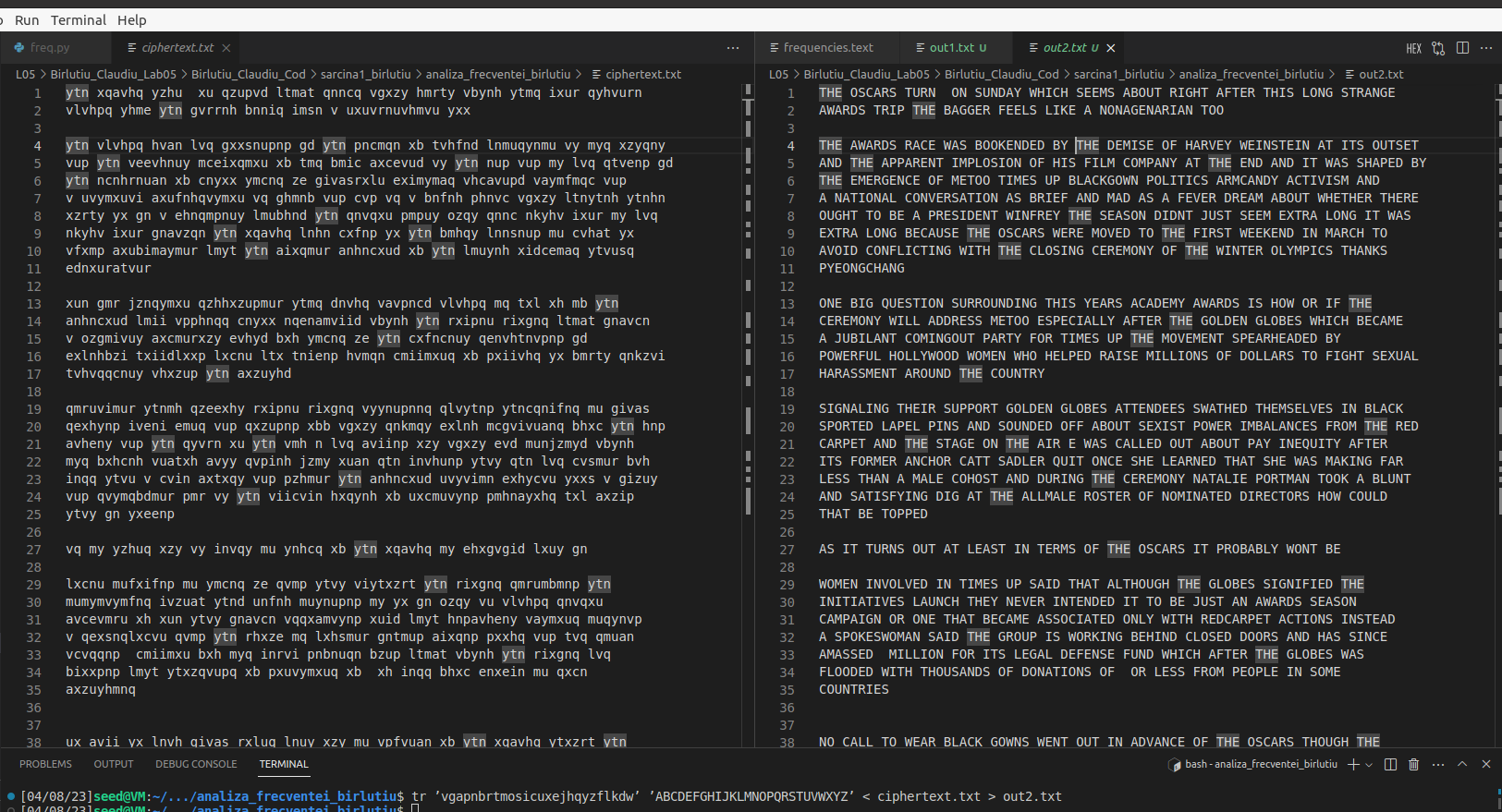
* avem:
  + ytn = the
  + vup = and
  + mur = ing
  + ynh = te\* => ter [0] **h =r**
  + xzy = \*\*t => o\*t => out [2] **z = u**
  + mxu = i\*n => [1] **x = o**
  + gnq = \*e\* => \*es [4] bes => **g =b**
  + ytv = tha
  + nqy = e\*t => [3] **q =s**
  + vii = a\*\* => [7] **i = l (**NATIONAi)
  + bxh = \*or = [5] for **b=f**
  + lvq = \*as = [6] was  **l = w**
  + nuy = ent
  + vyn = ate
  + uvy = nat
  + lmu = win
  + nvh = ear
  + cmu = \*in [8] => min **c=m** (DEcIqE = DmcISE = DEMISE)
  + tmq = his
  + vhp => ard

=>

* a b c d e f g h i j k l m n o p q r s t u v w x y z
* v g \* p n b r t m i c u x h q y z l o
* încercam cu valorile descoprite iar apoi vom identifica și pe celelalte lipsa
* abdefghilmnorstuwx (ABDEFGHILMNORSTUWX) = vgpnbrtmicuxhqyzlo
* identificam restul corespondetelor din cuvinte: OSaARS => **a = c;** TRIe => **e =p ;** LIsE => **s = k;** Bd **=> d =y;** HARfEY = > **f = v;** SEkIST=> k=x; jUESTION = > **j = q;** EkTRA = > **k =x; (modificare); (**XUST=JUST) => **o= j;** (eRIwE = PRIwE) => **w =z**

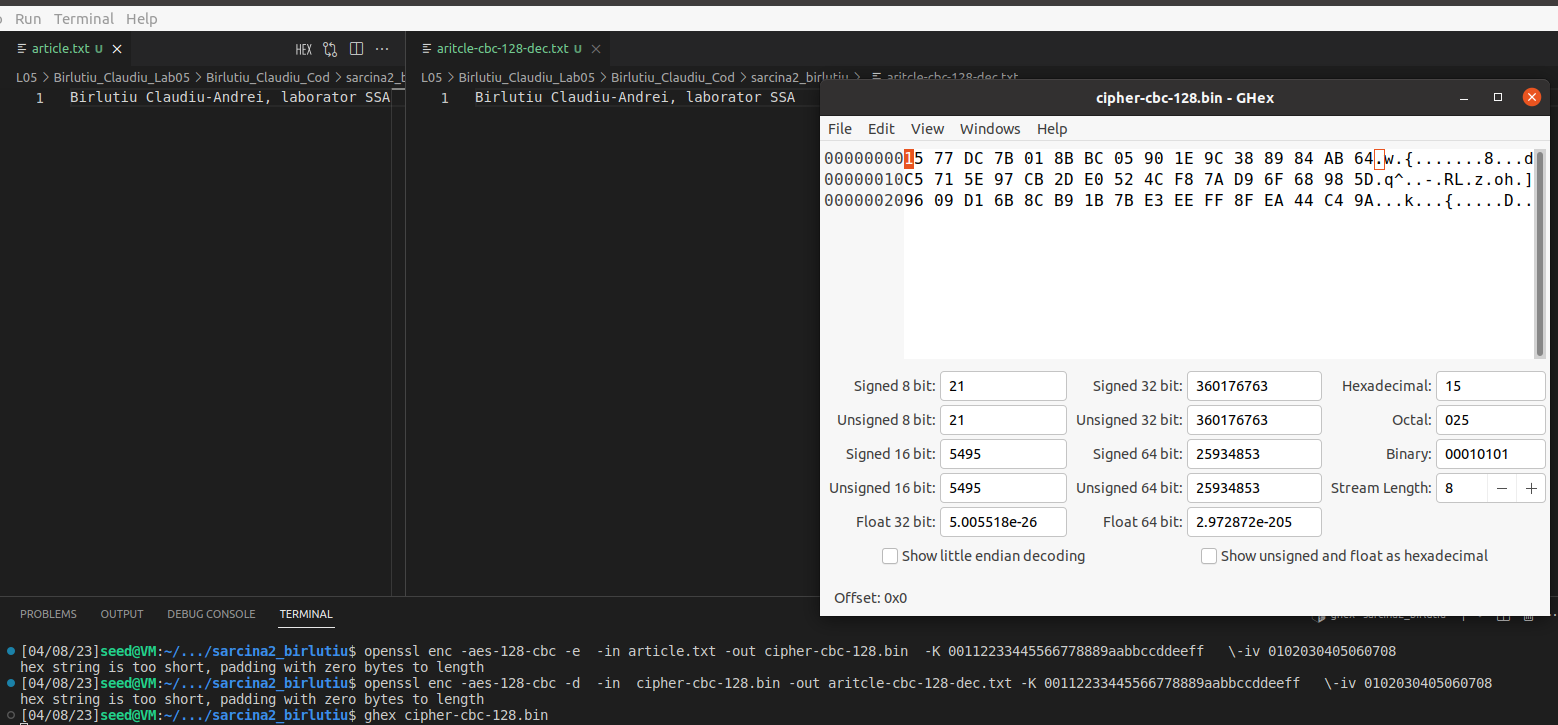
**REZULTAT:**

ab**c**defghi**jk**lmno**pq**rstu**v**wx**y**z(AB**C**DEFGHIJ**K**LMNO**PQ**RSTU**V**WX**YZ**)=vg**a**pnbrtm**os**icux**ej**hqyz**f**lk**dw**

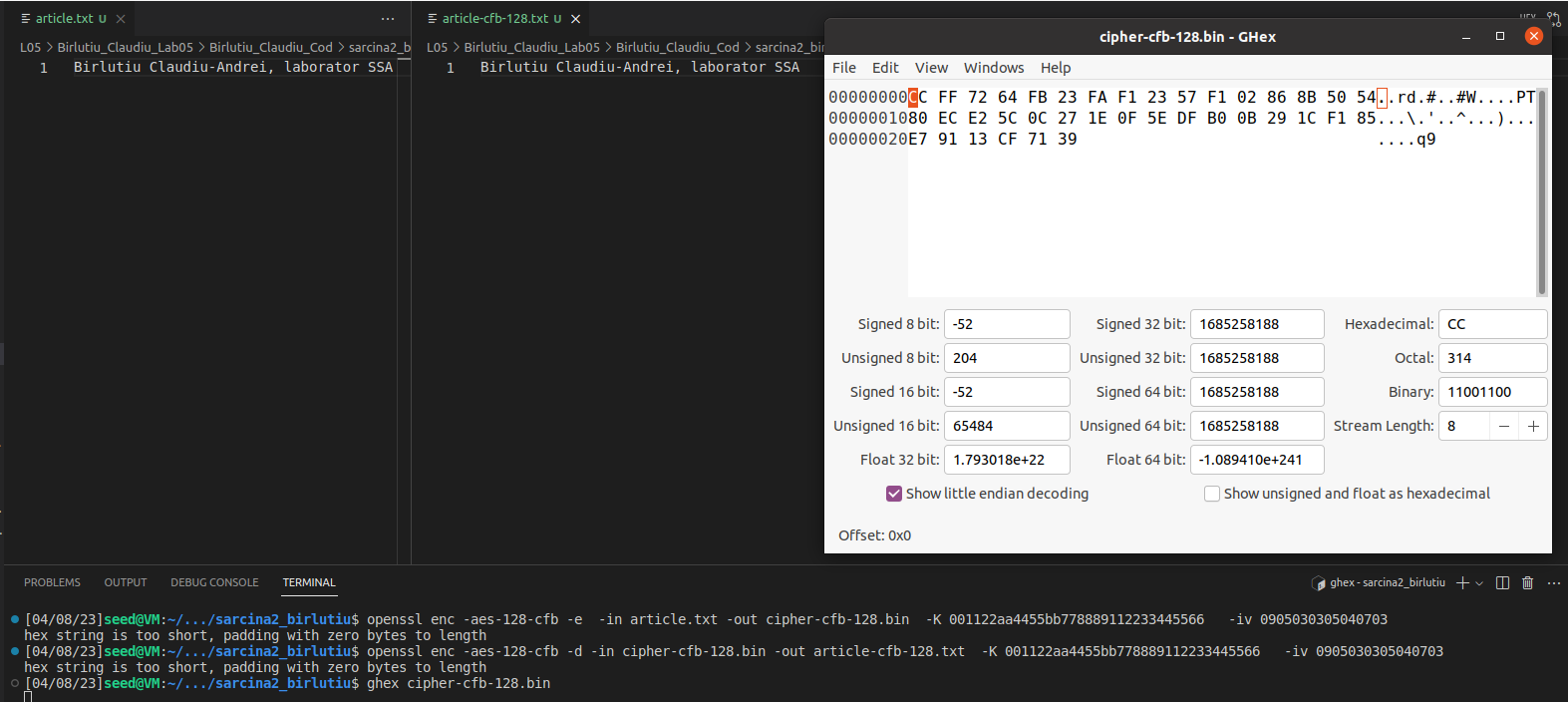


***Sarcina 2: Criptarea cu diferite cifruri și în diverse moduri***

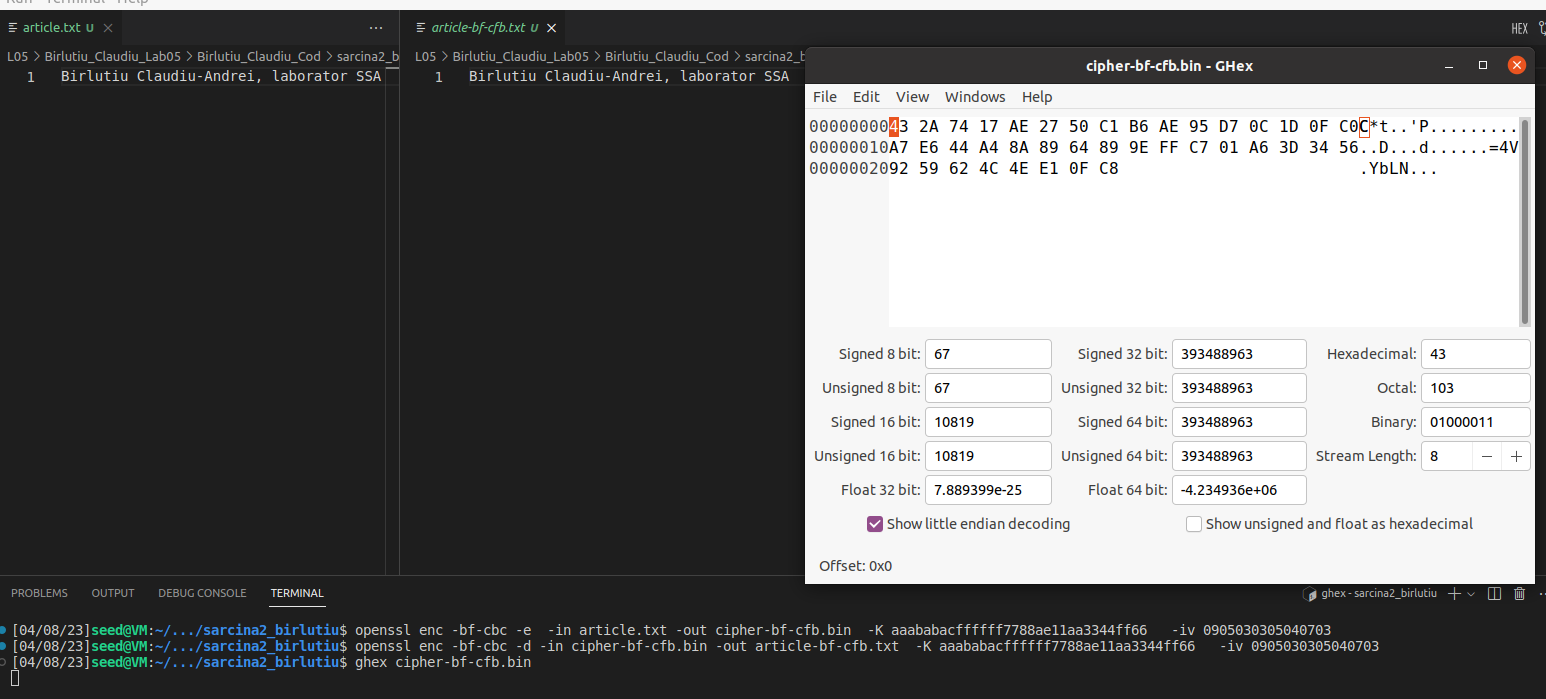
* am încercat urmatorele comenzi de cifrare a textului article.txt
  + **openssl enc -aes-128-cbc -e -in article.txt -out cipher-cbc-128.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708**
  + comanda cripteaza fisierul "article.txt" folosind cifrul AES cu o cheie de 128 de biti si modul de operare CBC, generand fisierul criptat "cipher.bin".
  + parametrul "-K" specifica cheia de criptare, in format hexazecimal. In acest caz, cheia specificata este "00112233445566778889aabbccddeeff".
  + parametrul "-iv" specifica vectorul de initializare (IV), in format hexazecimal. IV-ul este un parametru necesar pentru modurile de operare CBC si OFB. In acest caz, IV-ul specificat este "0102030405060708"



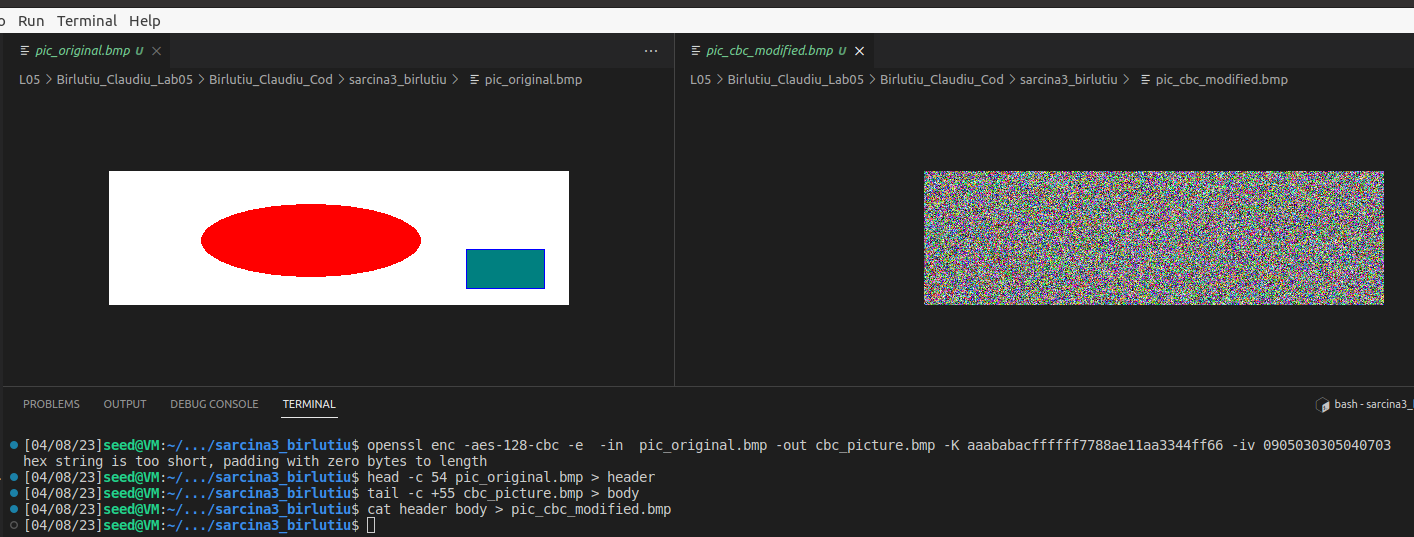
* + am decriptat de asemnea fișierul: **openssl enc -aes-128-cbc -e -in cipher-cbc-128.bin -out aritcle-cbc-128.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708**
* openssl enc **-aes-128-cfb -**e -in article.txt -out cipher-cfb-128.bin -K 001122aa4455bb778889112233445566 -iv 0905030305040703
* openssl enc **-aes-128-cfb** -d -in cipher-cfb-128.bin -out article-cfb-128.txt -K 001122aa4455bb778889112233445566 -iv 0905030305040703



* openssl enc **-bf-cbc** -e -in article.txt -out cipher-bf-cfb.bin -K aaababacffffff7788ae11aa3344ff66 -iv 0905030305040703
* openssl enc **-bf-cbc** -d -in cipher-bf-cfb.bin -out article-bf-cfb.txt -K aaababacffffff7788ae11aa3344ff66 -iv 0905030305040703



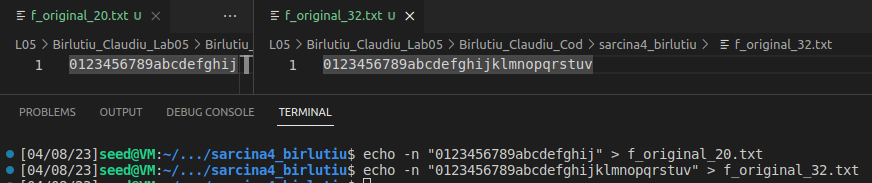
## Sarcina 3: Modul de criptare -- ECB vs. CBC

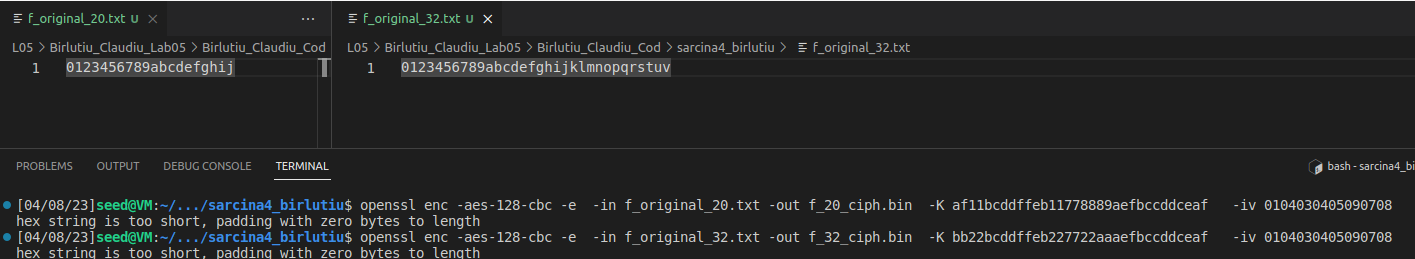
* In continuare vom encodifica o imagine cu extensia .bmp cu cele 2 modalitati de criptare ECB so CBC si vom modifica headerul imaginii criptate pentru a putea fi vizulizata cu ajutorul comenzilor afisate in alborator;
* din docmunenatre am gasit ca:
  + **modul ECB**: funcționează prin criptarea fiecărui bloc de date BMP separat, utilizând aceeași cheie de criptare pentru fiecare bloc astfle ca blocurile de date identice vor fi criptate în același mod, ceea ce poate face ca datele criptate să fie vulnerabile
  + **modul CBC:** funcționează prin criptarea fiecărui bloc BMP utilizând cheia de criptare, dar combină rezultatele criptării cu blocul BMP anterior prin aplicarea unei operații de XOR →datele criptate să fie mai sigure, deoarece blocurile identice de date vor fi criptate în moduri diferite
* openssl enc **-aes-128-ecb** -e -in pic\_original.bmp -out ebc\_picture.bmp -K aaababacffffff7788ae11aa3344ff66
  + Se vor rula urmatoarele comenzi pentru ca fisierul ebc\_picture.bmp sa fie tratat ca si un fisier bmp.
    - * head -c 54 pic\_original.bmp > header
      * tail -c +55 ebc\_picture.bmp > body
      * cat header body > pic\_ebc\_modified.bmp
    - se observa cum blocurile de date identice se cripteaza la fel si se pt distige formele obiectelor
* openssl enc -**aes-128-cbc** -e -in pic\_original.bmp -out cbc\_picture.bmp -K aaababacffffff7788ae11aa3344ff66 -iv 0905030305040703
  + Se vor rula urmatoarele comenzi pentru ca fisierul ebc\_picture.bmp sa fie tratat ca si un fisier bmp.
    - * head -c 54 pic\_original.bmp > header
      * tail -c +55 cbc\_picture.bmp > body
      * cat header body > pic\_cbc\_modified.bmp
      * in cazul encodarii cu CBC se observa ca nu se disting formele din imaginea initiala, deci gradul de securitate e mai ridicat, cee ce demonstreaza ca CBC reduce vulnerabilitatea descifrarii mesajeor

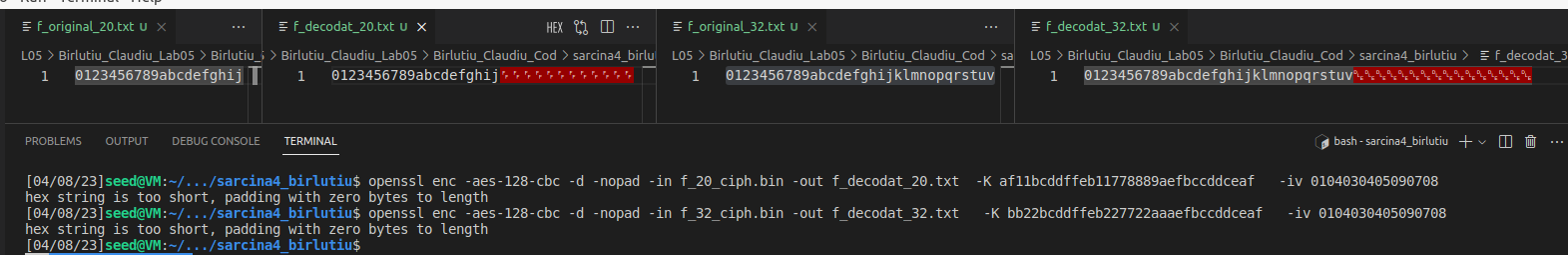
## Sarcina 4 : Caractere de completare pentru textul în clar

### Care sunt caracterele de completare în cifrarea AES atunci când lungimea textului în clar este 20 octeți și 32 octeți

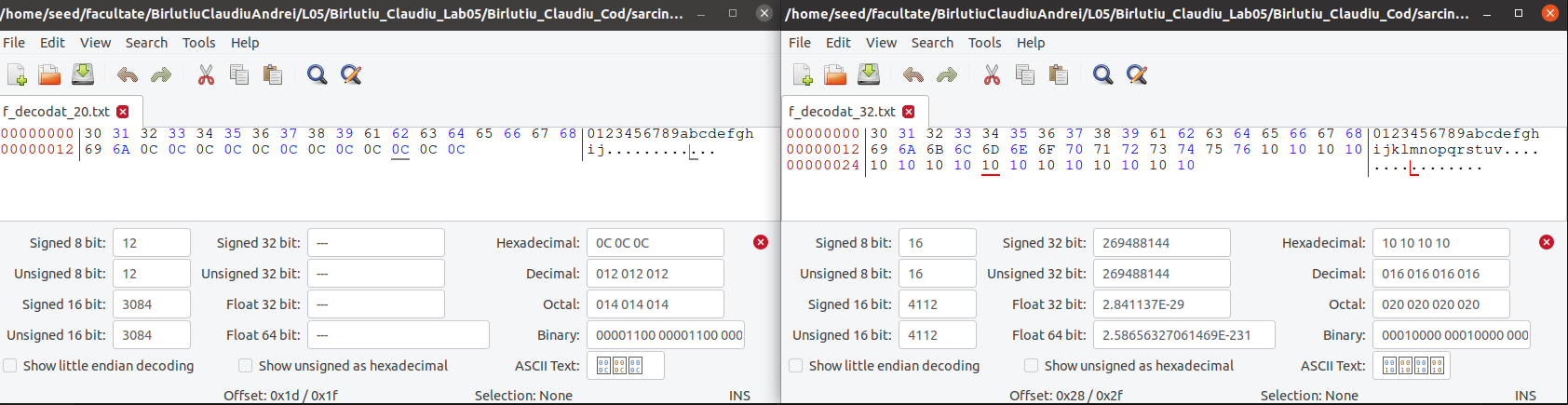
* Am ales ca și studiu de caz modul **CBC** pentru criptare, care este un mod care introduce caractere complementare și vom observa acets lucru prin executarea pasilor ce s-au menționat în lucrarea de laborator și vom folosi bless editor pentru a observa caracterele introduse în plus
* cream un fisier cu 20 de caractere și unul cu 32 de caractere (octeti)



* vom encodifica cele 2 fisiere cu cu modul cbc aes pe 128 de biti:
  + openssl enc -aes-128-cbc -e -in f\_original\_20.txt -out f\_20\_ciph.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + openssl enc -aes-128-cbc -e -in f\_original\_32.txt -out f\_32\_ciph.bin -K bb22bcddffeb227722aaaefbccddceaf -iv 0104030405090708
* vom decoda cele 2 fisier binare cu optiunea **nopad** astfel încât caracterele adaugate în plus se vor vedea în fieserele noi decodate
  + openssl enc -aes-128-cbc -d -nopad -in f\_20\_ciph.bin -out f\_decodat\_20.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + openssl enc -aes-128-cbc -d -nopad -in f\_32\_ciph.bin -out f\_decodat\_32.txt -K bb22bcddffeb227722aaaefbccddceaf -iv 0104030405090708
* în imaginea de mai jos se observa cum au fost adaugate caractere complementare la final; acestea nu sunt afisabile, dar putem sa le vizualizam cu un editor de text cum ar fi bless

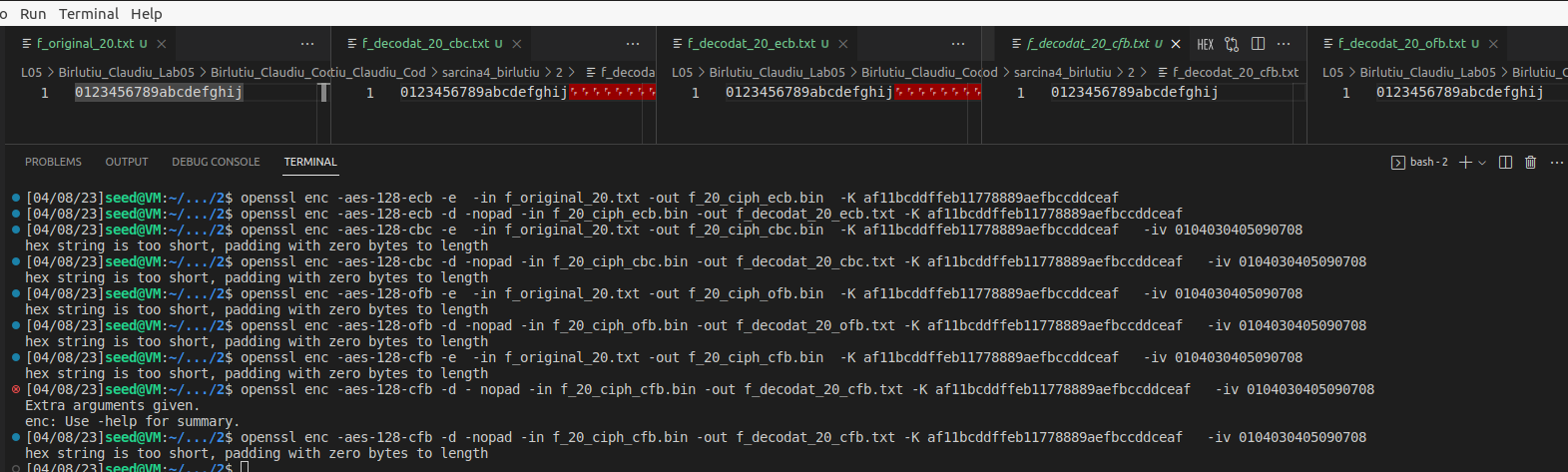


* am examinat prin intermediul tool-ului bless ce caratere s-au adaugat în plus și am observat ce în cadrul fisierului cu 20 de caractere s-a aduagat octetul 0x0C(form feed) de 12 ori => 32 de octeti, iar pentru cel de 32 de caractere s-a pus 0x10 (linefeed) de 16 ori => 48 de octeti; astfel numărul de octeti a devenit **multiplu** de 8



1. **Considerați modurile ECB, CBC, CFB, și OFB de cifrare a unui fișier. Care au caractere de completare și care nu au?**

* Din documentare am aflat ca modurile **ECB** și **CBC** folosesc o dimensiune fixă a blocului de date pentru criptare și de aceea vor utiliza caractere de completare. Datele trebuie să fie în mod explicit completate cu caractere suplimentare, astfel încât să poată fi împărțite în blocuri egale.
* În cauzl modurilor **CFB** și **OFB** permit o dimensiune variabilă a blocului de date și astfle nu este necesara completarea datelor -> aceste moduri de criptare criptează o secvență de biți și generează un flux de ieșire pentru a fi combinat cu datele de intrare în loc sa cripteze blocuri de date întregi
  + - **EBC**
* openssl enc -aes-128-ecb -e -in f\_original\_20.txt -out f\_20\_ciph\_ecb.bin -K af11bcddffeb11778889aefbccddceaf
* openssl enc -aes-128-ecb -d -nopad -in f\_20\_ciph\_ecb.bin -out f\_decodat\_20\_ecb.txt -K af11bcddffeb11778889aefbccddceaf
* **CBC**:
  + openssl enc -aes-128-cbc -e -in f\_original\_20.txt -out f\_20\_ciph\_cbc.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + openssl enc -aes-128-cbc -d -nopad -in f\_20\_ciph\_cbc.bin -out f\_decodat\_20\_cbc.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
* **OFB**
  + openssl enc -aes-128-ofb -e -in f\_original\_20.txt -out f\_20\_ciph\_ofb.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + openssl enc -aes-128-ofb -d -nopad -in f\_20\_ciph\_ofb.bin -out f\_decodat\_20\_ofb.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
* **CFB**:
  + openssl enc -aes-128-cfb -e -in f\_original\_20.txt -out f\_20\_ciph\_cfb.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + openssl enc -aes-128-cfb -d - nopad -in f\_20\_ciph\_cfb.bin -out f\_decodat\_20\_cfb.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
* se poate observa n următoarea imagine care dintre moduri aduaga caractere complementare



## Sarcina 5: Propagarea erorilor -- Text cifrat alterat

### Câtă informație puteți recupera din descifrarea fișierului alterat, dacă modul de cifrare a fost ECB, CBC, CFB, respectiv OFB?

### În czul ECB o parte desctul de semnificativa.

### Răspunsul dat înainte de efectuarea sarcinii

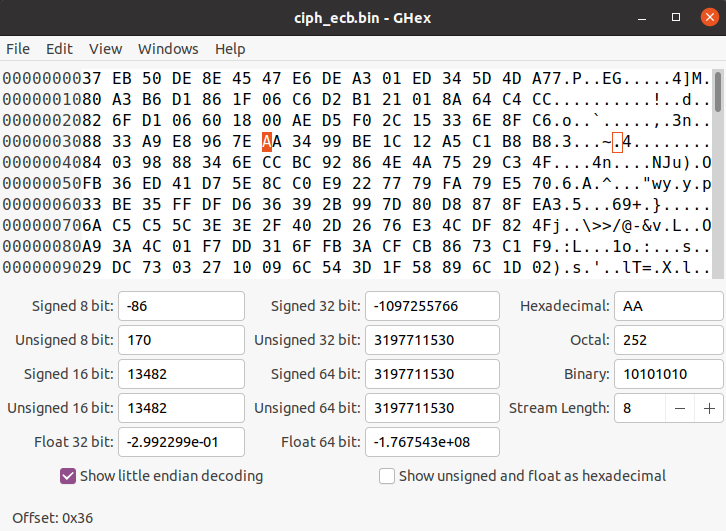
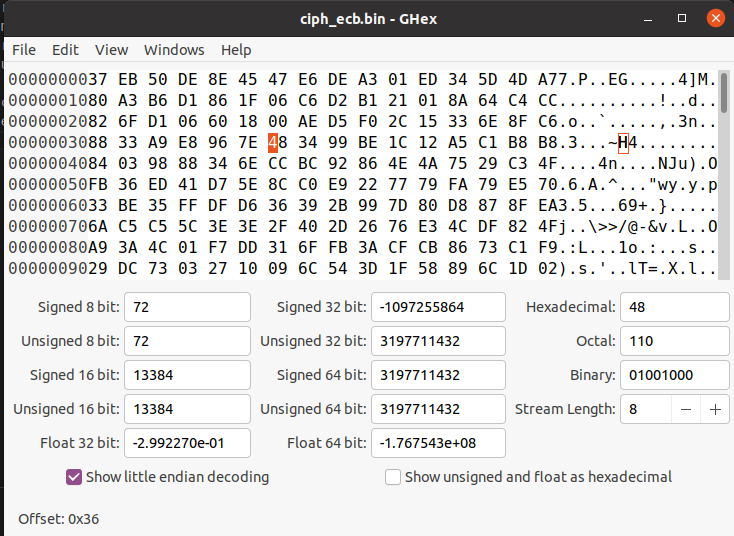
* Ma gândesc, ca doar în cadrul ECB putem recupera mare parte din informație deoarece codificare se face la nivelul fiecarui bloc și nu debinde de alte blocuri cum este în cazul celorlaltor modele de criptare. În cazul ECB acest lucru poate face ca modificările într-un bloc să nu afecteze blocurile ulterioare, ceea ce poate permite recuperarea unor părți semnificative ale informației originale din fișierul alterat. De asemenea cred ca și în cadrul OFB putem recupera date

### Răspunsul după execuția acestei sarcini.

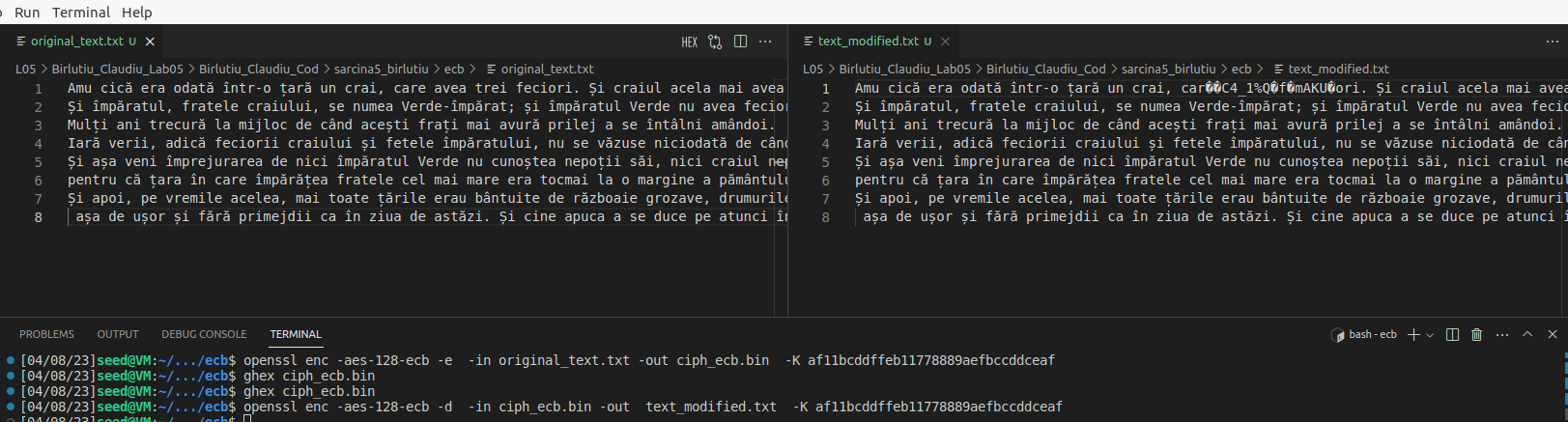
* Voi face testul pe o imagine: aceasta are mai mult de 1000 de octeti si vom observa mai bine modificarile la nivelul blocurilor
* 55 in hexa e 0x37 => trebuie moidifcat octetul de la adresa 0x36 si vom folosi tool-ul **Ghex**

### ECB:

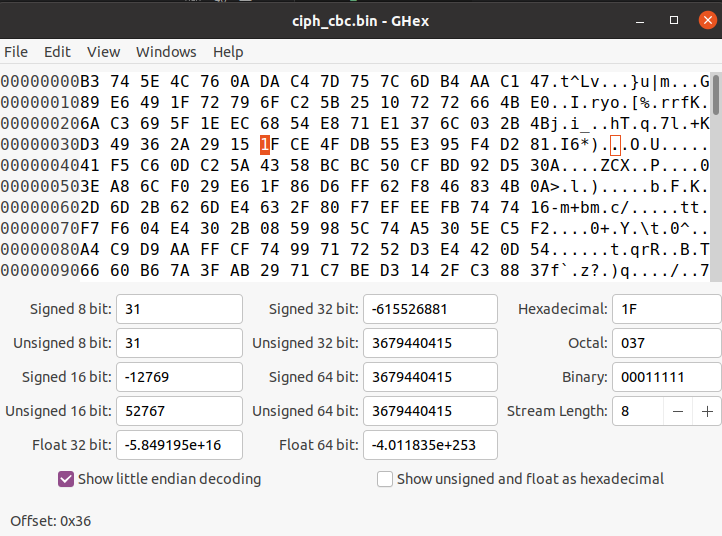
* + openssl enc -aes-128-ecb -e -in original\_text.txt -out ciph\_ecb.bin -K af11bcddffeb11778889aefbccddceaf
  + modificam un byte-ul de la adresa 0x36 din **0x48** în **0xAA**

****

* + decodificam: openssl enc -aes-128-ecb -d -in ciph\_ecb.bin -out text\_modified.txt -K af11bcddffeb11778889aefbccddceaf



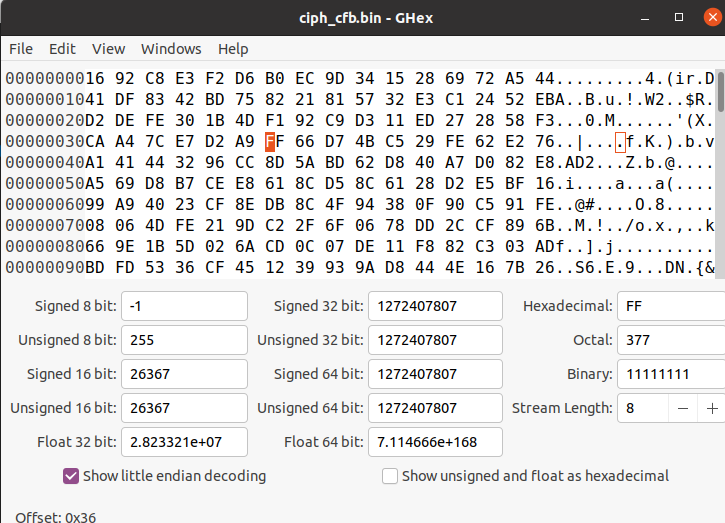
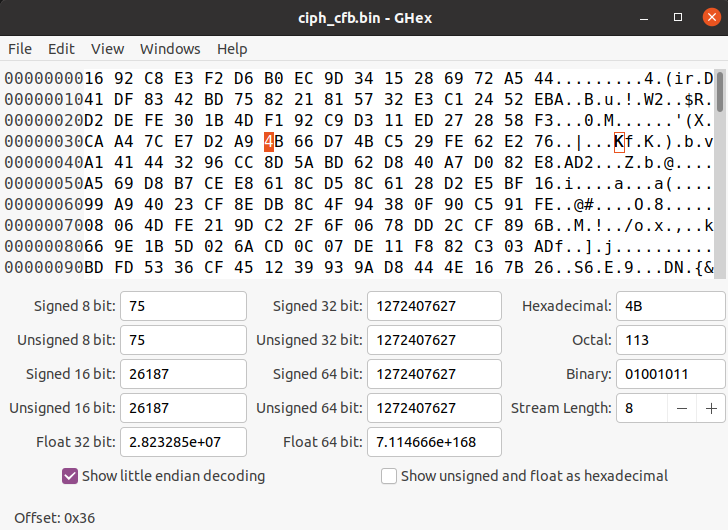
* + vom încerca descifrarea mesajului sa vedem cât de corupt e mesajul și am observat ca textul decodata nu suferă modificari majore, doar într-un singur loc foarte puține caractere au fost pierdute
* **CBC:**
  + openssl enc -aes-128-cbc -e -in original\_text.txt -out ciph\_cbc.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + modificam byte-ul de la adresa 0x36 din **0x1F** în **0x00**



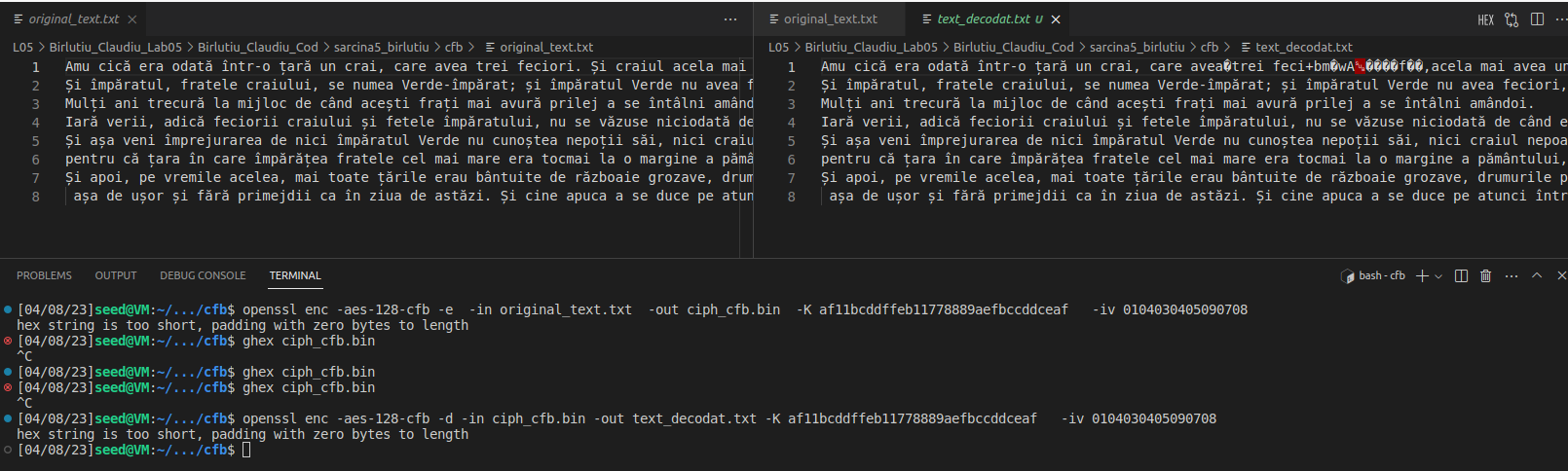
* + decodema princ comanda: openssl enc -aes-128-cbc -d -in ciph\_cbc.bin -out text\_decodat.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + observam ca numărul de caractere pierdute e mai mare



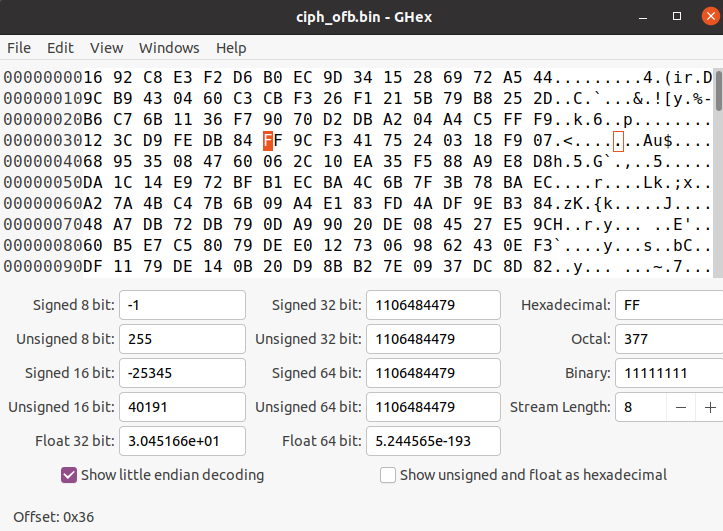
* **CFB:**
  + openssl enc -aes-128-cfb -e -in original\_text.txt -out ciph\_cfb.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + modificam byte-ul de la adresa 0x36 din **0x4B** în **0xFF**

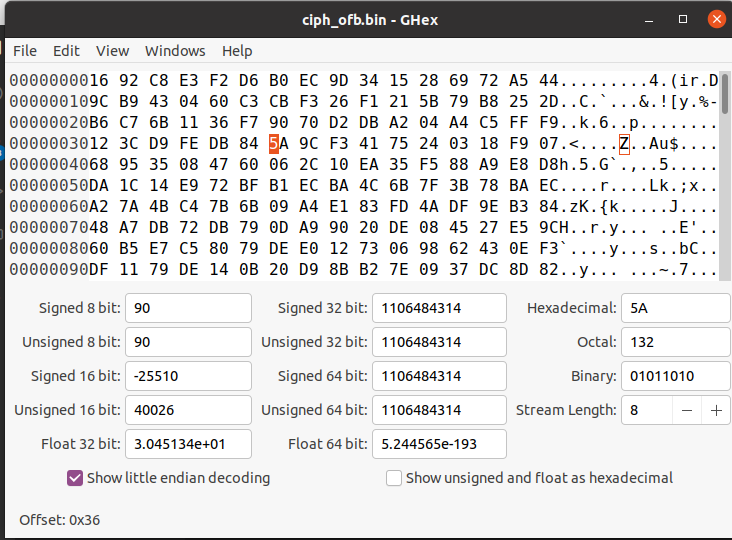


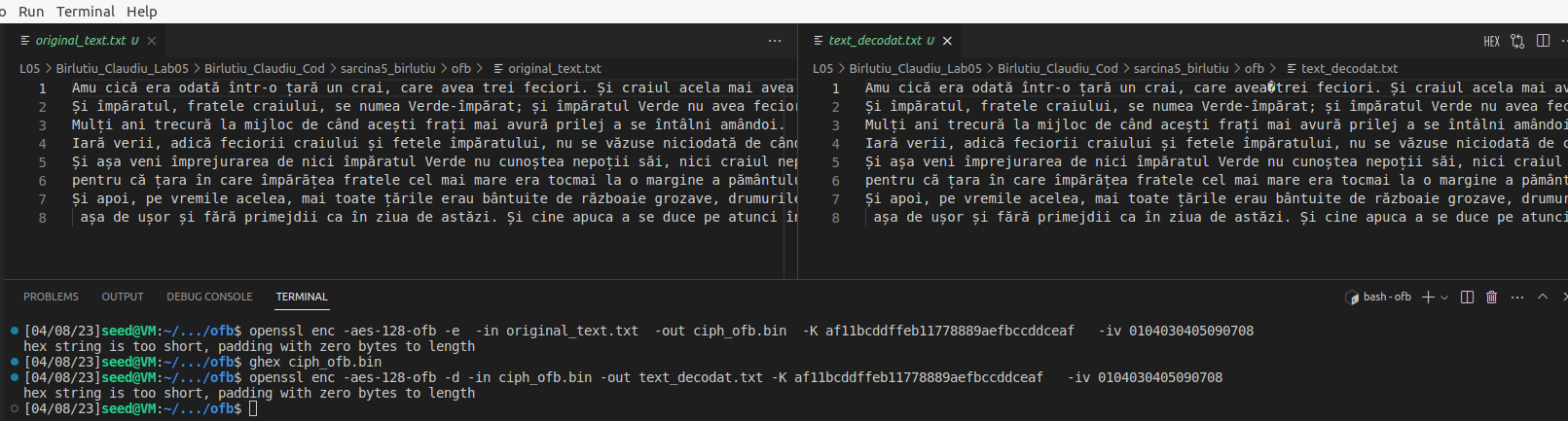
* + decodema princ comanda: openssl enc -aes-128-cfb -d -in ciph\_cfb.bin -out text\_decodat.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + obsservam și în czaul acesta sunt mai multe caractere pierdute



* **OFB:**
  + openssl enc -aes-128-ofb -e -in original\_text.txt -out ciph\_ofb.bin -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + modificam byte-ul de la adresa 0x36 din **0x5A** în **0xFF**



* + decodema prin comanda: openssl enc -aes-128-ofb -d -in ciph\_ofb.bin -out text\_decodat.txt -K af11bcddffeb11778889aefbccddceaf -iv 0104030405090708
  + observam în ca incadrul modelului OFB avem cea mai mica deterioare

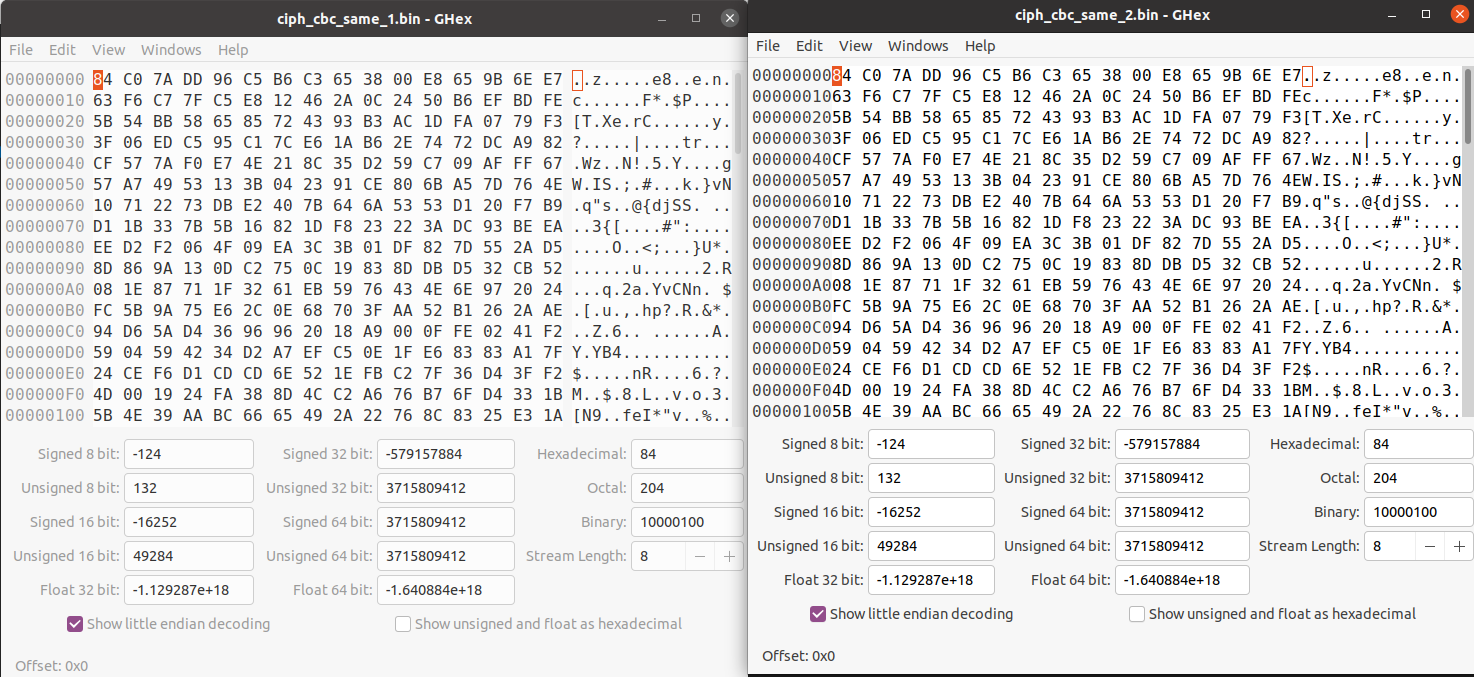


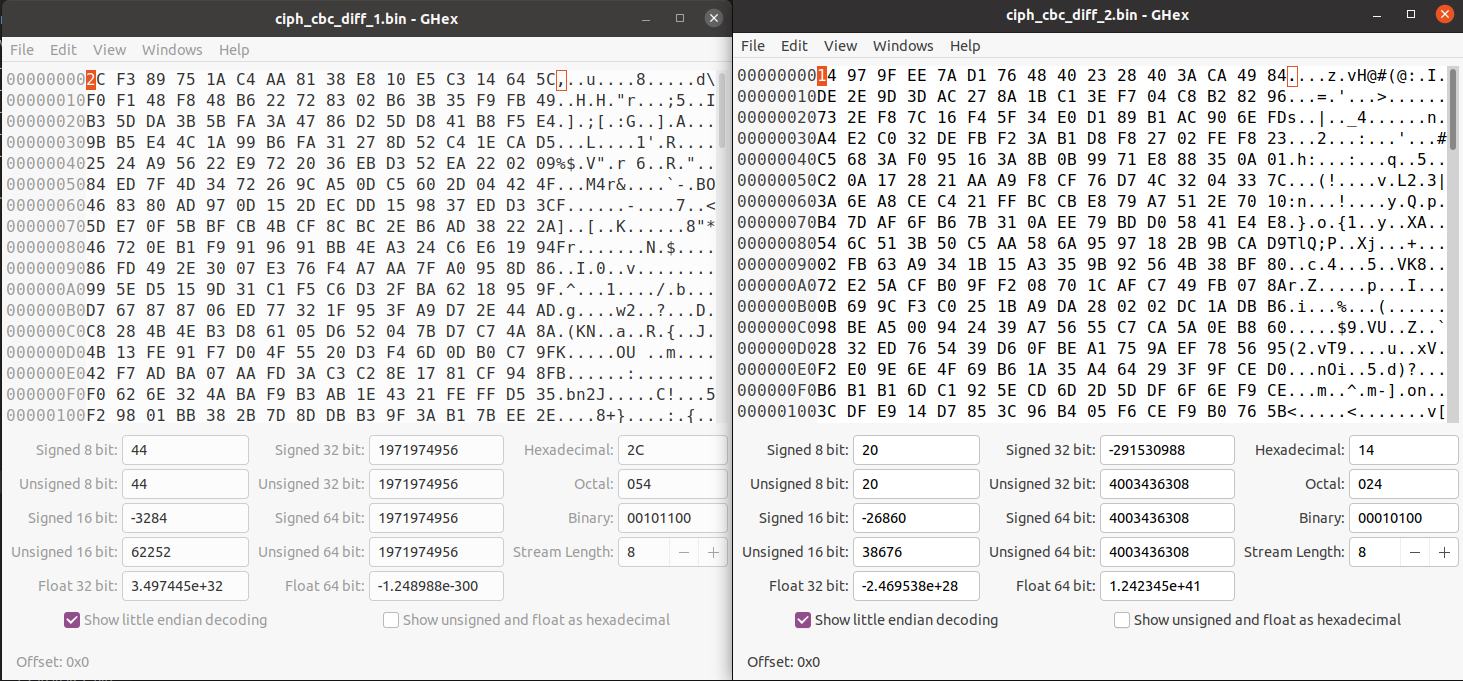
### Explicați de ce.

* In cadrul **ECB** putem recupera mare parte din informație deoarece codificare se face la nivelul fiecarui bloc și nu debinde de alte blocuri cum este în cazul celorlaltor modele de criptare. În cazul ECB acest lucru poate face ca modificările într-un bloc să nu afecteze blocurile ulterioare, ceea ce poate permite recuperarea unor părți semnificative ale informației originale din fișierul alterat.
* Pentru **CBC** blocurile de date sunt cifrate în funcție de blocurile precedente și este utilizată o valoare de inițializare pentru primul bloc -> interdependență între blocuri (modificările la un bloc afecteaza blocurile următoare). Mai precis, fiecare bloc de date este combinat cu blocul anterior înainte de a fi criptat prin intermediul unei operații XOR.
* Pentru **CFB** este textul clar este cifrat în blocuri, iar rezultatul cifrat este apoi utilizat pentru a cifra următorul bloc -> aceeași situație ca la CBC, este greu de recuperat datele.
* Pentru **OFB** va pemite recuperarea semnificativa de date doarece un bloc initial este cifrat, iar apoi utilizat pentru cifrarea urmatorului bloc fără utilizare textului clar. Biții de ieșire generați sunt independenți de blocurile de date de intrare și de biții de ieșire anteriori, ceea ce face ca OFB să fie mai rezistent la unele tipuri de atacuri criptografice.

## Sarcina 6. Vectorul inițial (IV) și erori comune

### 6.1 Sarcina 6.1. Unicitatea IV

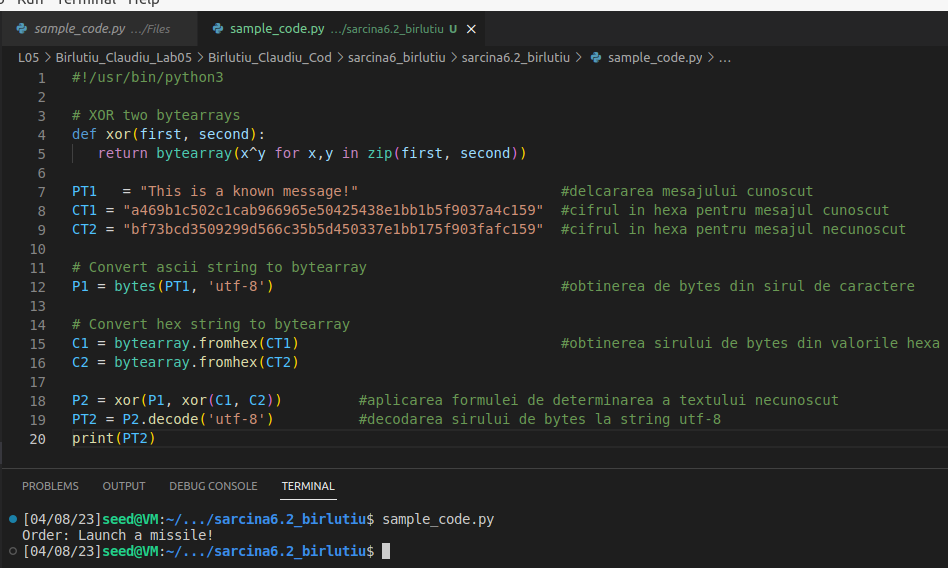
* La recomandarile din laborator vom executa in prima faza pe textul original\_text.txt ce reprezinta o parte din povestea lui Harap Alb, un algoritm de criptare AES modul CBC in 2 cazuri:
  + cifram textul cu doi IV la fel (vom executa comanda de 2 ori):
    - openssl enc -aes-128-cbc -e -in original\_text.txt -out ciph\_cbc\_smae\_x.bin -K af11bcddffeb11778889aefbccddceaf -iv **aaafafafbb090708 -** unde x va fi 1 și apoi 2
    - observam ca cele 2 fisiere binare de codificare sunt la fel
  + cifram textul cu doi IV difieri (vom executa comanda de 2 ori):
    - openssl enc -aes-128-cbc -e -in original\_text.txt -out ciph\_cbc\_diff\_x.bin -K af11bcddffeb11778889aefbccddceaf -iv **0b0bafafbb0907aa /0c0cafafbb0907ff -** unde x va fi 1 și apoi 2



* + - observam ca cele 2 fisiere cu mesajul codificat difera chiar dacă au acceasi cheie
    - această **unicitate** a IV-ului este importantă deoarece, în caz contrar, ar putea exista mai multe blocuri de date criptate care să folosească aceeași valoare de IV și aceeași cheie de criptare, ceea ce ar putea conduce la dezvăluirea informațiilor secrete din mesajul original -> un atacator ar putea să deducă blocurile de date originale din textul criptat prin analizarea valorilor repetate ale IV.

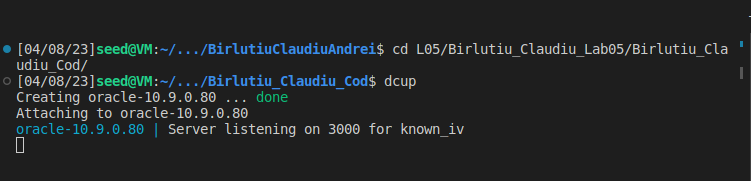
### 6.2 Sarcina 6.2. Eroare comună: folosirea aceluiași IV

* Având in vedere faptul ca cele 2 cifre au fos encodificate folosit același IV – uri înseamnă ca **Plain text 1** XOR **Plain text 2** va fi egal cu **Cipher text 1** XOR **Cipher text 2 –** deaorece având aceeași cheie de criptare și același vector de intializare =>
* PT2 = CT1 xor CT2 xor PT1
* ne vom folosi de programul sample\_code.py pentru a face operatiile de xor pe caractere

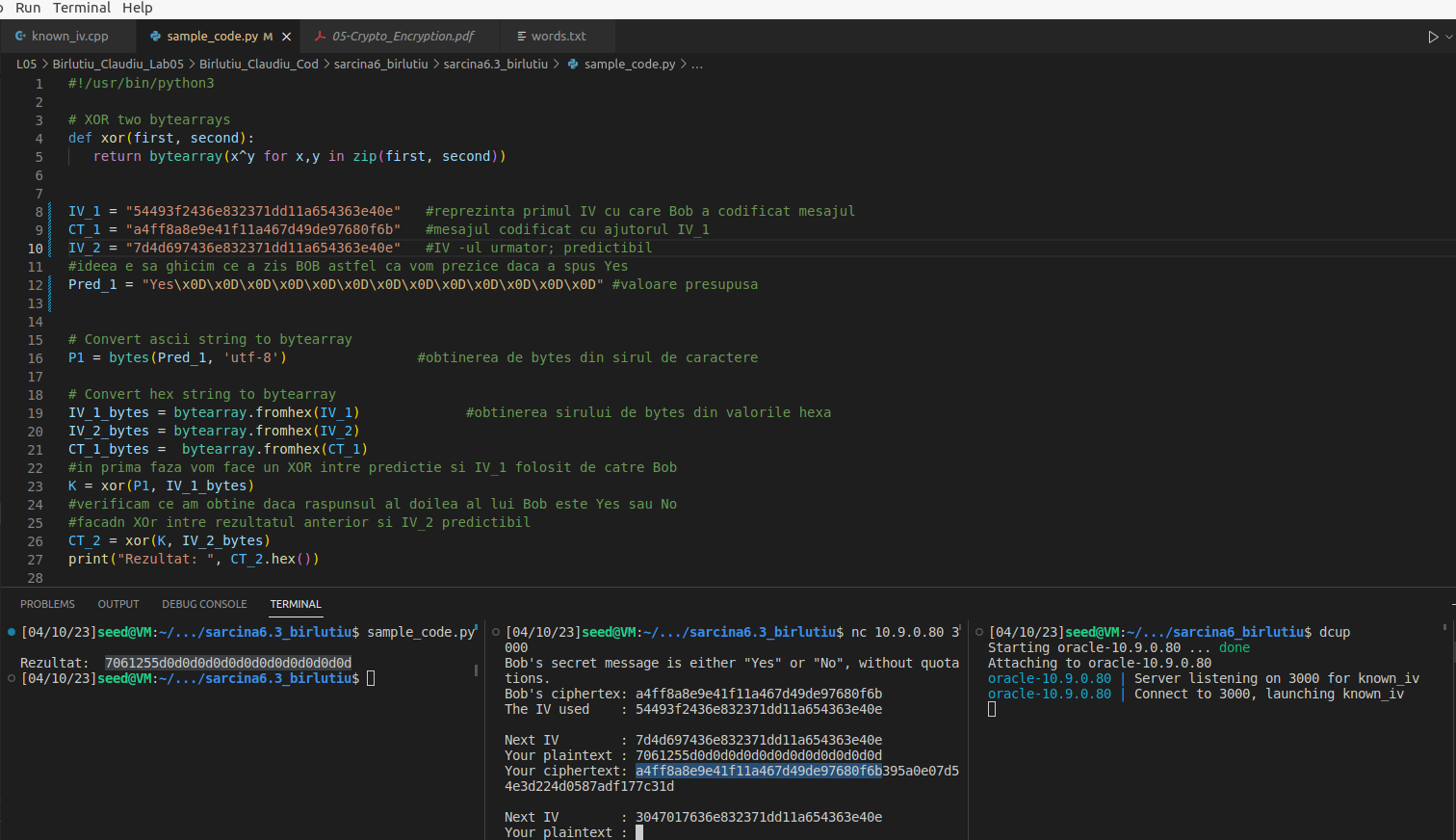


### 6.3 Sarcina 6.3. Eroare comună: folosirea unui IV predictibil

* În prima faza am porint containerul docker cu server-ul de oracle:



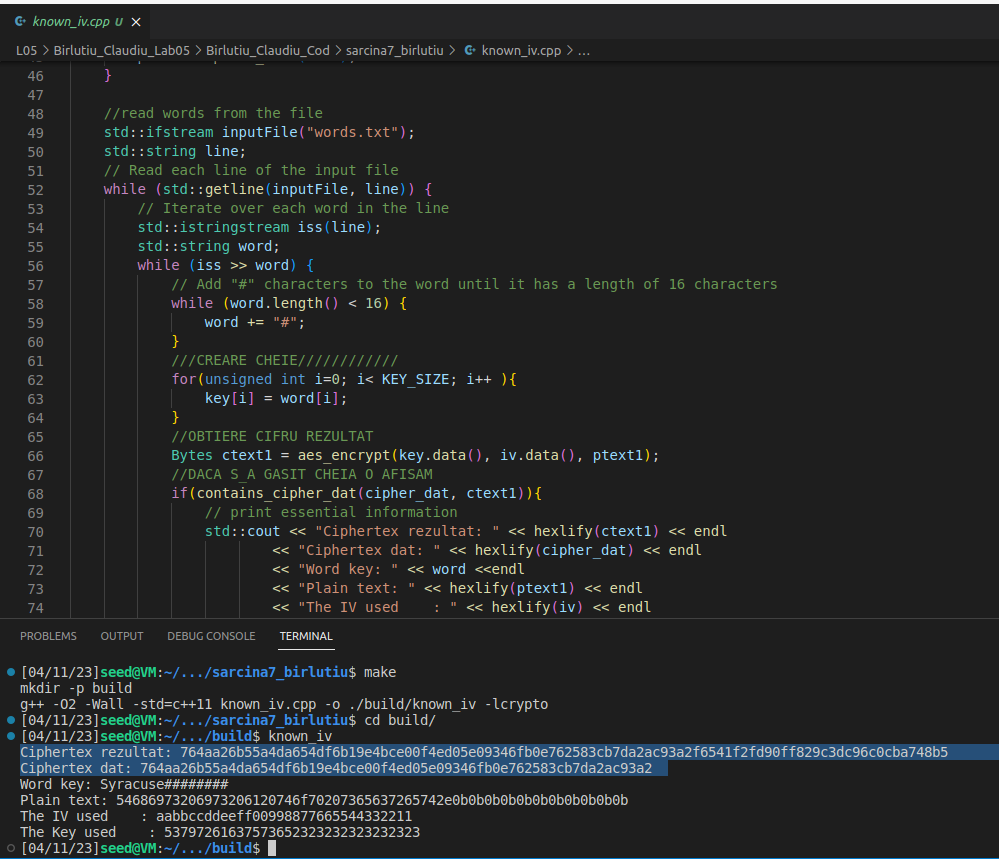
* ca și premisa avem faptul ca Eve știe ca Bob a trims un mesaj Yes or No (nu știe care dintre), da pe lângă aceasta cunoaște cifrul mesajului și IV ul folosit.Pe lângă acestea Eve mai știe și Iv-ul pe care îl va folosi Bob pentru criptarea urmatorului mesaj
* sarcina pe care am auto a fost sa construiec un mesaj P2 pe care o săi -l cerem lui Bob sa îl cifreze și sa dea textul cifrat -> aceasta ocazie am folosit pentry a detemrina dacă continutul real al mesajului lui Bob este **Yes** sau **No**
* am făcut logica de calcul în fișierul **sample\_code.py** unde am luat ca plai text P2 ca fiind codul hexa pentru „Yes”
  + am făcut xor intre codul acesta hexa (Yes) și primul IV afisat de server (Bob) iar apoi am făcut XOR cu IV-2; valorile se observa în imaginea de mai jos
  + am trimis rezultattul obținut în hexa lui Bob pentru a-l cifra și observam ca cele 2 coduri sunt identice CT-1 și CT-2 (la început) ceea ce demonstrează faptul ca mesajul trimis de Bob a fost yes
  + OBSERVATIE: a trebuit să se facă paddinng la stringul Yes pentru a avea un numar de octeti multiplu de 16. S-a ADAUGTA **0X0D**



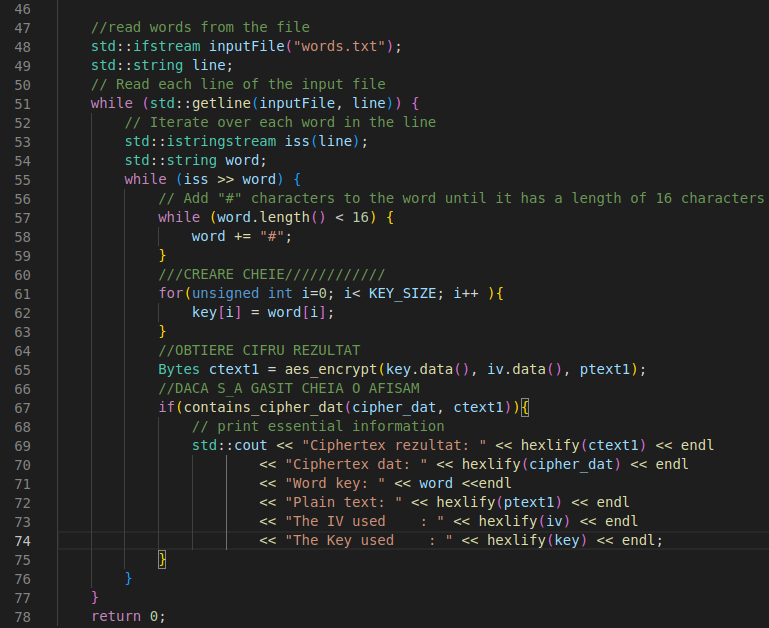
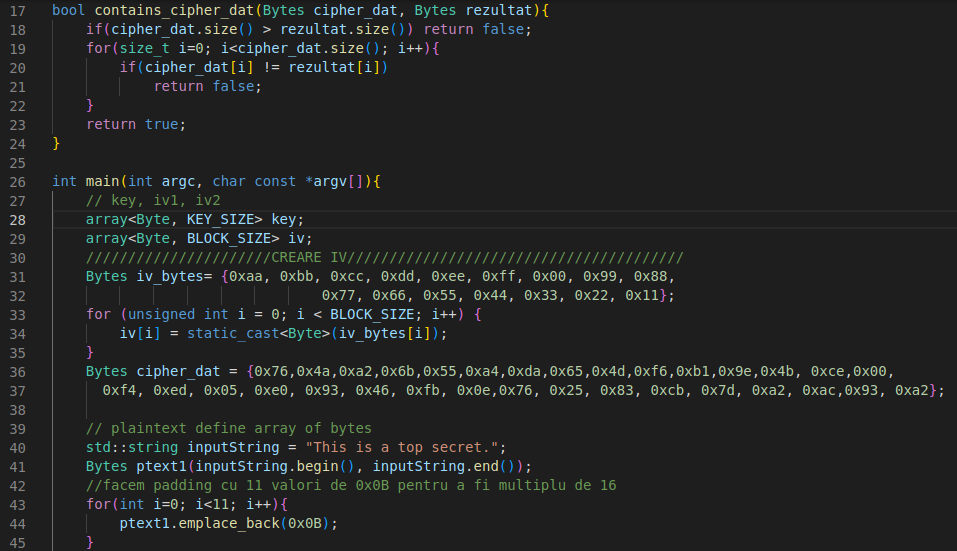
* mai sus se regaseste ss-ul cu scriptul scris

## Sarcina 7: Programarea folosind biblioteca criptografică

* În prima faza am analizat fisierele pe care le aveam în **encryption\_oracle** unde aveam **evp-encrypt.hpp** care conține cele funcții de **encrypt** și **decrypt** cu AES. Am mdoificat **known\_iv.cpp** astfel încât sa îmi citească fiecare cuvânt din word.txt și sa facă paading cu # pana la 16 caractere. Am luat aceste cuvinte și le-am pus într-un sir de bytes ce reprezintă cheia de encriptare. Pentru plain text deaorece are 21 de caractere (octeti) am făcut padding pana la 32 de octeti pentru a fi multiplu de 16 numărul. => am adaugat astfel **11** octeti de **0x0B** conform regulii PKS.
* Compilarea programului am făcut-o cu **make**
* Pentru fiecare cuvant luat ca si cheie am apelat functia de encriptare si am verificat daca cipher\_tex-ul dat e continut in cel rezultat.(am creat o functie de verificare; iar pentru acea cheie am afisat informatiile relevante



IMPORTANT: CODUL SURSA SE AFLA IN Birlutiu\_Claudiu\_Cod/sarcina7\_birlutiu



#include <array>

#include <cstdlib>

#include <iostream>

#include <fstream>

#include <algorithm>

#include <vector>

#include <openssl/rand.h>

#include "evp-encrypt.hpp"

#include "utils.hpp"

using namespace std;

/\*\*

\* Aceasta functie verifica daca cifrul dat e continut in cel rezultat

\*/

bool contains\_cipher\_dat(Bytes cipher\_dat, Bytes rezultat){

if(cipher\_dat.size() > rezultat.size()) return false;

for(size\_t i=0; i<cipher\_dat.size(); i++){

if(cipher\_dat[i] != rezultat[i])

return false;

}

return true;

}

int main(int argc, char const \*argv[]){

// key, iv1, iv2

array<Byte, KEY\_SIZE> key;

array<Byte, BLOCK\_SIZE> iv;

//////////////////////CREARE IV////////////////////////////////////////

Bytes iv\_bytes= {0xaa, 0xbb, 0xcc, 0xdd, 0xee, 0xff, 0x00, 0x99, 0x88,

0x77, 0x66, 0x55, 0x44, 0x33, 0x22, 0x11};

for (unsigned int i = 0; i < BLOCK\_SIZE; i++) {

iv[i] = static\_cast<Byte>(iv\_bytes[i]);

}

Bytes cipher\_dat = {0x76,0x4a,0xa2,0x6b,0x55,0xa4,0xda,0x65,0x4d,0xf6,0xb1,0x9e,0x4b, 0xce,0x00,

0xf4, 0xed, 0x05, 0xe0, 0x93, 0x46, 0xfb, 0x0e,0x76, 0x25, 0x83, 0xcb, 0x7d, 0xa2, 0xac,0x93, 0xa2};

// plaintext define array of bytes

std::string inputString = "This is a top secret.";

Bytes ptext1(inputString.begin(), inputString.end());

//facem padding cu 11 valori de 0x0B pentru a fi multiplu de 16

for(int i=0; i<11; i++){

ptext1.emplace\_back(0x0B);

}

//read words from the file

std::ifstream inputFile("words.txt");

std::string line;

// Read each line of the input file

while (std::getline(inputFile, line)) {

// Iterate over each word in the line

std::istringstream iss(line);

std::string word;

while (iss >> word) {

// Add "#" characters to the word until it has a length of 16 characters

while (word.length() < 16) {

word += "#";

}

///CREARE CHEIE////////////

for(unsigned int i=0; i< KEY\_SIZE; i++ ){

key[i] = word[i];

}

//OBTIERE CIFRU REZULTAT

Bytes ctext1 = aes\_encrypt(key.data(), iv.data(), ptext1);

//DACA S\_A GASIT CHEIA O AFISAM

if(contains\_cipher\_dat(cipher\_dat, ctext1)){

// print essential information

std::cout << "Ciphertex rezultat: " << hexlify(ctext1) << endl

<< "Ciphertex dat: " << hexlify(cipher\_dat) << endl

<< "Word key: " << word <<endl

<< "Plain text: " << hexlify(ptext1) << endl

<< "The IV used : " << hexlify(iv) << endl

<< "The Key used : " << hexlify(key) << endl;

}

}

}

return 0;

}