Criptarea imaginilor

Voinea Andrei, grupa 341

1. Descrierea task-ului

În această lucrare, ne propunem să criptăm și decriptăm la loc, în python, orice format de imagine (JPG, JPEG, PNG, etc.), folosindu-ne de două librării criptografice, anume „cryptography” și „secrets”, care ne pun la dispoziție algoritmul de criptare simetrică **AES**, respectiv, un generator de numere pseudoaleatoare. Se va pune în evidență și diferența dintre modurile de criptare ECB și CBC.

2. Motivația lucrării (Atenție! Exprimare informală!)

În fiecare zi, ne „lovim” de imagini, fotografii, fie că acestea sunt salvate pe device-urile noastre, fie că le vedem pe Internet sau că le primim de la alte persoane. Câteodată, aceste imagini conțin date sensibile, pe care nu vrem să le păstrăm într-un format înțeles de oricine sau pur și simplu nu vrem să împărtășim aceste imagini cu toată lumea. Din acest motiv, criptarea este soluția potrivită, datorită simplității, vitezei, dar în același timp și a eficienței (cifrurile folosite fiind sigure).

3. Librăriile folosite

Librăria „**cryptography**”[3] conține atât interfețe de nivel scăzut pentru algoritmi criptografici comuni, precum cifruri simetrice, funcții pentru derivarea cheilor, etc., cât și unele de nivel înalt, sigure și ușor de folosit, care necesită puține (spre deloc) alegeri de făcut de către dezvoltatori în legătură cu configurarea.

Librăria „**secrets**”[2] a fost introdusă în python 3.6, cu scopul de a genera numere pseudoaleatoare robuste și sigure, adecvate pentru gestionarea datelor, precum parolele, autentificarea conturilor, tokeni de securitate, etc. În particular, este de preferat să se folosească „secrets” în locul generatorului de numere pseudoaleatoare din cadrul modulului „random”, din cauză că cel din urmă a fost conceput pentru simulare și modelare, nu pentru securitate și criptografie.

4. Metoda abordată

Vom genera cheia, cât și IV-ul pentru o criptare AES (CBC) de 128 de biți (16 bytes). Cum o criptare nu are sens fără decriptare, le vom salva pe acestea într-un fișier de tip „.txt”, pentru a le folosi în viitor, când vom avea nevoie de ele.

```
secret_key = secrets.randbits(128)
IV = secrets.randbits(128)

f = open(settings_str, "w")
f.write(str(secret_key) + "\n")
f.write(str(IV) + "\n")

secret_key = secret_key.to_bytes(16, 'big')
IV = IV.to_bytes(16, 'big')
```

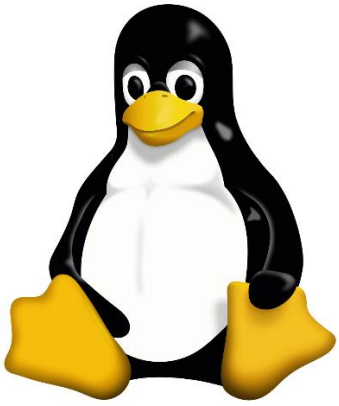
Figură 1. Generarea cheii secrete și a IV-ului

Cum AES este o criptare pe blocuri, am ales să grupez pixelii în grupuri de câte 5 (3 bytes fiecare, datorită canalelor RGB), rezultând astfel în 15 bytes. Cum nu putem aplica AES doar pe 15, vom adăuga un padding de 1 byte, astfel ajungând la 16 bytes. În cazul în care lungimea pozei nu este un multiplu de 5, ultima grupare va fi formată din pixelii rămași, adăugând la final un padding de oricâți bytes va fi nevoie pentru a ajunge la 16. Odată ce avem cei 16 bytes, putem să criptăm blocul. Din cei 16 bytes noi obținuți, vom avea nevoie doar de numărul de bytes pe care i-am grupat (înainte de a aplica padding-ul). Cum pentru decriptare, vom avea nevoie și de ultimul byte (sau ultimii bytes) pe care nu-l folosim în crearea imaginii criptate, vom păstra valoarea acestora în fișierul „.txt" pe care l-am folosit și pentru salvearea cheii și a IV-ului.

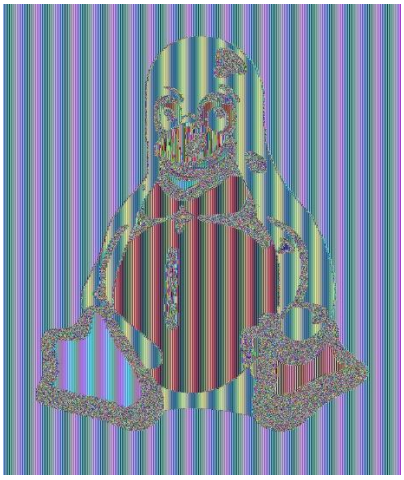
```
padding_img = [] # lista pentru salvarea padding-ului
for i in range(lin): # parcurgem linie cu linie
    for j in range(0, col, 5): # parcurgem coloana cu un salt de 5
        padder = padding.PKCS7(128).padder() # padder din libraria cryptography, va adauga x biti pana ajunge la 128

        pixels = [] # lista pentru pastrarea valorilor RGB a celor n pixeli
        for k in range(5): # luam doar 5 pixeli
            if j + k < col: # verificam sa nu trecem de sfarsitul imaginii
                pixels.append(img[i][j + k][0])
                pixels.append(img[i][j + k][1])
                pixels.append(img[i][j + k][2])
            else:
                break

        len_pad = 16 - len(pixels) # lungimea paddingului
        pixels = [int(x).to_bytes(1, 'big') for x in pixels] # convertim fiecare valoarea in byte

        block = b''
        for pixel in pixels:
            block += pixel # concatenam valorile

        block = padder.update(block) + padder.finalize() # aplicam padding
        block_crypt = encryptor.update(block) # criptam

        for k in range(len_pad - 1, -1, -1):
            padding_img.append(block_crypt[15 - k]) # conteaza ordinea in care salvam paddingul

        for k in range(5):
            if j + k < col:
                img_enc[i][j + k][0] = block_crypt[k * 3] # inlocuim in imaginea criptata
                img_enc[i][j + k][1] = block_crypt[k * 3 + 1]
                img_enc[i][j + k][2] = block_crypt[k * 3 + 2]
            else:
                break
```

Figură 2. Procesul de criptare

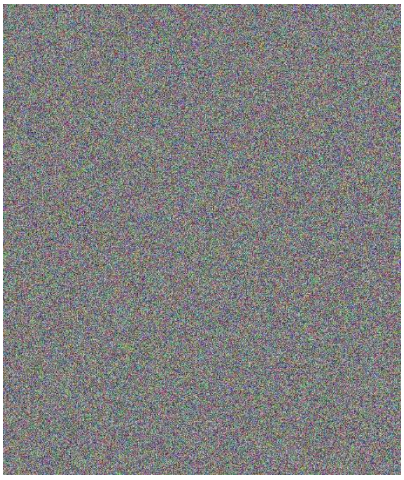
5. Rezultate



Figură 3. Linux mascot



Figură 4. AES-128-ECB



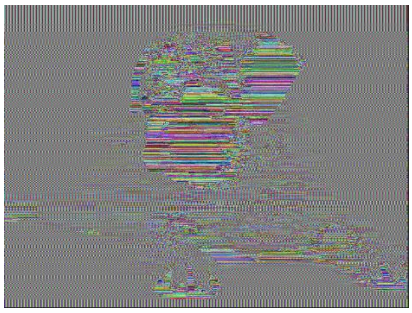
Figură 5. AES-128-CBC

6. Probleme întâlnite

Deși am folosit AES din librăria cryptography, inițial îmi propusem ca pentru criptare să folosesc XOR bitwise. Pentru simularea unui CBC, am generat la fel un IV, iar operația era relativ simplă. Xoram pixelul curent cu IV-ul și cu cheia (lucram doar cu un pixel), după care IV-ul lua valorea pixelului criptat pentru următoarea operație. Problema este că pentru o porțiune mare, în care vom avea doar pixeli albi, xorarea dintre pixel și cheie va da o valoare constantă. Caz în care putem reduce formula la xorarea dintre o constantă (constanta fiind xorarea dintre pixelii albi și cheie) și IV. Cum pentru următoarea operație IV-ul ia valoarea pixelului criptat, vom calcula xorarea dintre constantă și pixelul criptat care va da IV-ul anterior folosit, ceea ce înseamnă că pixelii se vor repeta din doi în doi, atât timp cât pixelul din plaintext nu se schimbă, ceea ce nu ne dorim. (a ^ b = c ⇔ a ^ c = b)

Figură 6. Cute dog

Figură 7 Cute dog ECB



Figură 8 Cute dog CBC

[Codul se găsește aici.]

Referințe:

1- Cryptography – site-ul librăriei „cryptography”
2- Secrets – site-ul librăriei „secrets”