

CES571S - L2 - Crypto Encryption

467261 - Yifu Wang

2018 - 09 - 25

1 Substitution Cipher

I didn't reinvent the crypto algorithm for substitution cipher. I did some research in this field and use the substitution solver to solve this online. The following steps is the path I will take to implement a solver by myself.

1. Scan the ciphertext, record the frequency of each letter, record the frequency of each 1-letter, 2-letter, 3-letter words, record the pattern of two consequent same letter.
2. The frequency of what we recorded in step one in plaintext of English should be

letter: {ETAOIN/RSHDL}>0.04, {CU/FMPGWY}>0.02, {BV/KXJQZ}<0.02

1-letter word: a, i

2-letter word: of, to, in, it, is, be, as, at, so, we, he, by, or, on, no...

3-letter word: the, and...

consequent pattern: ss, ee, tt, ff, ll, mm, oo...

3. Sort the a-z letters by the frequency we recorded and use this as an initial key for our algorithm. This algorithm could be Hill Climbing, Simulated Annealing or Genetic Algorithm. In order to find the best solution which make our translated text fit the frequency table listed in step 2 best. We need a algorithm to measure the fitness of current solution. This algorithm can be as simple as measure how close a 'out-of-order' sequence to a 'well-ordered' sequence.
4. After we get the solution from step 3. The key we obtained might be still not right. Then we exchange the letter in the same group in the **letter** listed in step 2 by exhaustive method. Quit the algorithm once the grading of current solution exceed the pre-set threshold.

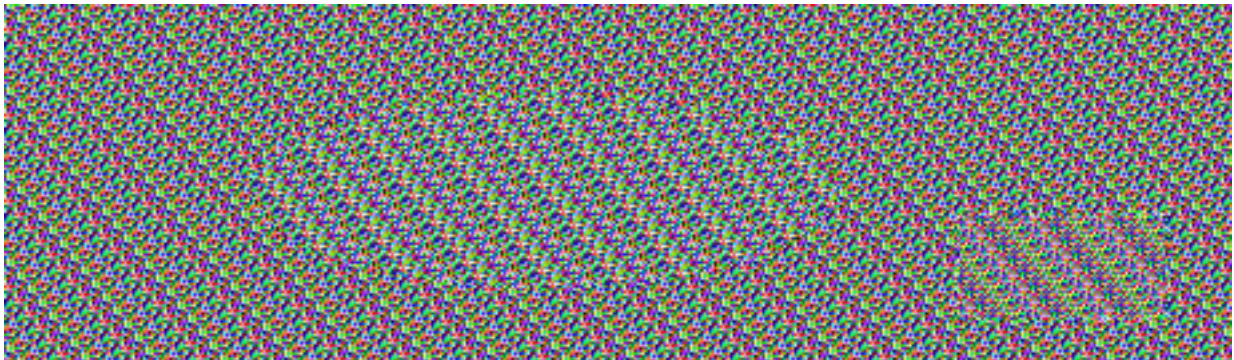
Here is the [solution](#) I obtained and the [solver](#) I used.

2 Encryption Mode - ECB vs. CBC

This task is somehow related to another course I take, CSE554T, where we learn to process 2D or 3D img data using Mathematica language. It's a interactive programming language. I choose use it for this task and I upload the source code to [github](#). The following is a simple explanation of the code.

1. KeyGen generate a L-byte-long random key.
2. The block cipher used in ECB and CBC are BitXor for simple.
3. The IV used in CBC is randomly generated, since we didn't really need to decrypt the image.

The following 3 images are origin, ECB and CBC.



You can see that the encrypted img produced by ECB still preserve the shape of original img. That is because when using an ECB same plaintext will be encrypted into same ciphertext. By applying CBC you can eliminate such disadvantage.

Following 3 images reproduce this phenomenon using another original image. [original](#) [ECB](#) [CBC](#)

3 Error Propagation - Corrupted Cipher Text

To answer the question first.

- **ECB,OFB** will only affect the block where the corruption happend.
- **CBC,CFB** will affect the block where the corruption happend and the block after it.

In order to observe the corruption easily. The plaintext I choose will be character '1' repeated 1008 times. I used Javascript with the package 'aes-js' to perform this task. I encountered several trouble by doing so.

- I have tried another famouse npm package for crypto, 'crypto-js', but I couldn't corrupt the ciphertext produced by it. It seems that 'crypto-js' do extra work to make sure ciphertext is untouched.

Here is the [source code](#). And the results: (the block refers to the block located at corrupt point)

- **ECB** only the block breaks.
- **CBC** and **CFB** only the block and the block after it.
- **OFB** every block break, totally unreadable.

The result of OFB is unexpected. Which I don't understand. From the definition in [this picture](#). And from the mathematical definition

$$\begin{aligned}C_i &= P_i \oplus O_i \\P_i &= C_i \oplus O_i \\O_i &= E_k(O_{i-1}) \\O_0 &= IV\end{aligned}$$

It seems that the single byte corruption won't cause any propagation.

4 Initial Vector

4.1

The Javascript source code is uploaded to [github](#). The plaintext is

'The quick brown fox jumps over the lazy dog. '

key, iv1 and iv2 are randomly generated. e1 and e2 using iv1 as IV, e3 using iv2. e1 and e2 are exactly the same. Here is the [screen shot](#). Thus since using same IV will cause the same plaintext encrypted into the same ciphertext. It's vulnerable under strong attacking.

4.2

Yes we can. Since we have

$$\begin{aligned}C_i &= P_i \oplus O_i \\P_i &= C_i \oplus O_i \\O_i &= E_k(O_{i-1}) \\O_0 &= IV\end{aligned}$$

thus $O_i = C_i \oplus P_i$. Furthermore if IV remains the same, all O_i will remain the same. By running [this algorithm](#). We have

P2 = 'Order: Launch a missile!'

If we replace OFB with CFB. We will at least reveal the first block of P2. Since by the definition

$$\begin{aligned}C_i &= E_K(C_{i-1}) \oplus P_i \\P_i &= E_K(C_{i-1}) \oplus C_i \\C_0 &= IV\end{aligned}$$

only C_0 will remain unchanged between two cipher processes.

4.3

The definition of CBC is (only one block)

$$\begin{aligned}C1 &= E_K(IV \oplus P1) \\C2 &= E_K(IV \oplus P2)\end{aligned}$$

Thus if we construct

$$P2 = IV2 \oplus IV1 \oplus P1$$

we shall get $C1 = C2$. The Javascript source code is uploaded to github. And from the debug view, we can see $P1 = \text{Yes}$.

When conduct this attacking. I encountered a trouble. That is when the plaintext is not a multiple of 16 (key size in byte). aes-js package will throw a error and refuse to continue any calculation. And I don't know what has been append to the P1 by default. So I have to using the key (known only to Bob) to decrypt the C1 and found out that ' ' have been added to make a 16 length P1. And the code of ' ' is 13. That's the reason of line 11.