Cryptanalysis of a class of ciphers based on known plaintext match

Introduction

There is only one member in my team who do all works.

Test1 is based on shifting on five known plaintext, our way don’ t need to guess key. The most import information is that key in shorter than 24. With this fact, we present three way for different types of crypto from simple to complex.

Test2 is based on a list of known words, which formed plaintext and then encrypt. We need to Reconstruction the plaintext based on cipher text. Here we assume there is a cyclic scheduling algorithm.

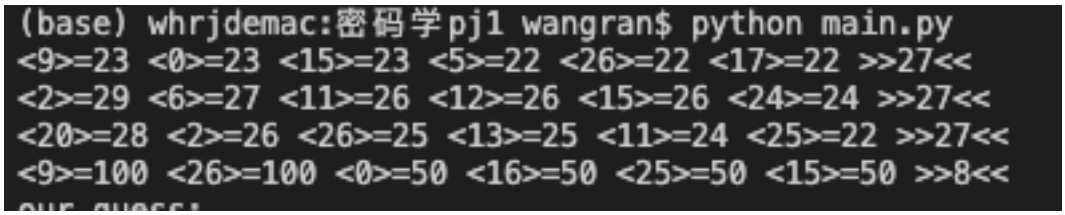
Informal explanation

TEST1

We present three ways to break test1, from the simplest situation to the most complexed situation.

First way for no random character ciphertext: Known plaintext match

We know the max key length is under 24, so there are only 24 possible shift step. We simply calculate distance between every plaintext and ciphertext. If there are more than 24 different distances, it can't be plaintext. Based on our experiment, wrong plaintext match ciphertext will always produce 27 different distance.



For example, the last line is the right plaintext with key lenth is 10, there is only 8 different discance.

Second, consider a ciphertext with some random character andcyclic schedule algorithm: Repeated Pattern

To eliminate the influence of random character, we move the ciphertext forward one character for L times to obtain L different ciphertext. for example:

ciphertext

iphertextc

phertextci

hertextcip

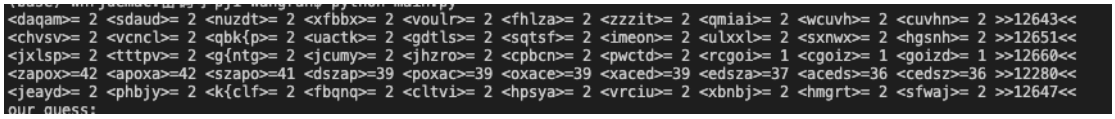
...

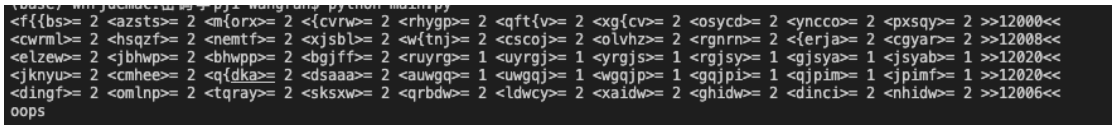
L is less than 24，because the key is no longer than 24.

We use one of plaintexts to align at one of the L ciphertext and calculate the distaInce. If it is the right plaintext, some cyclic pattern will appear. If it is not the right plaintext, distance will distribute randomly.

For example, if our key is [1,2,3,4,5], then in the distance array, 1,2,3,4,5 will appear several times.

In the worst case, we have to calculate 5 plaintext with 24 moved ciphertext, and calculate every repeated pattern ，which need much more memory than first way.





For example, with key length=10, there are far more repeated pattern than wrong plaintext. But if we use a random schedule algorithm, no repeated pattern can be found.

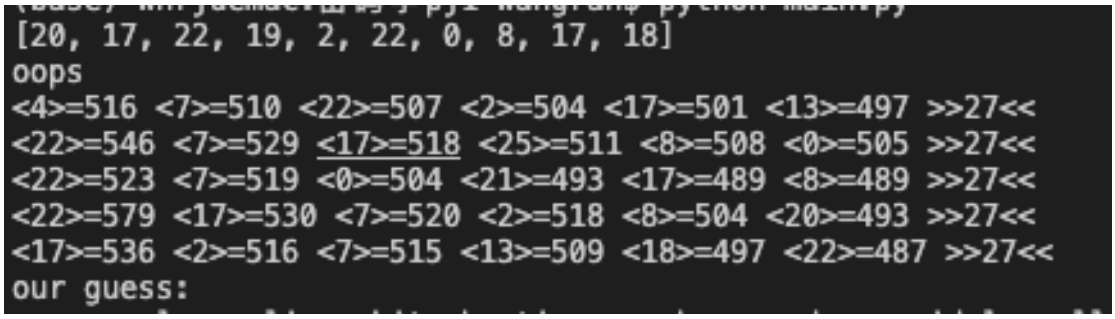
Third, consider a ciphertext with some random character and no cyclic schedule algorithm: Distance frequency analyse

Since there is no cyclic pattern, second way is useless and won’ t produce and repeated pattern. We back to basic frequency analyze. We use the same way to eliminate the influence like second way. If the right plaintext matches the ciphertext, at least 500 correct characters match each other, and they produce no more than 24 different distances. but if wrong plaintext match the ciphertext, distance will distribute randomly.

For example, if my key is [1,4,7,11,19,3], in the statistic of distance distribute, there must be more 1,4,7,11,19,3 than any other distance like 23,17,16,8. We sum up top 6 frequencies for each plaintext and chose the biggest sum as our guess.

With larger random character, there will be more and more noise, because the proportion of 500 right matches is smaller.

This way might be quicker than Second way, but second way is much more precise if there do exist repeated pattern.



For example, top line is our key, and right plaintext is the fourth one. Every key appear on the stastics.

TEST2: known plaintext attack

Test2 is far more difficult than test1, so we make some basic assumptions to make things easier: scheduler is cyclic, and no random character in first few words. Violate these assumptions will greatly reduce the accuracy.

We choose a word from form word list to match the ciphertext to generate the key and try to use this key to decrypt the ciphertext. Every time we decrypt the ciphertext by move key forward only one digit to reduce the influence of random character. If we use the wrong key or key is not aligned, we obtain meaningless messy code. But in the right cases, we may get meaningful words divided by space. We use SequenceMatcher from python difflib to calculate similarity between word from decrypted text and original word list. Based on our experiment, the right key with right aligned produces much more high ratio than wrong key or wrong aligned.

Now we have the key, and we shall use that key to reconstruct plaintext. We still use the key to decrypt ciphertext and calculate the SequenceMatcher ratio with every word on the list. If this ratio > 0.7 ， we return it as a word in guessed plaintext.

Rigorous description

Test1

First way:

For each p in plaintext and c in ciphertext:

    D=Distance(p,c)

Distance_dict[D]= Distance_dict[D]+1

If keys of Distance_dict <27 :

    Plaintext is correct

Second way:

Num_rand_char=len(ciphertext)-len(plaintext)

Shifted_round=Num_rand_char but no more than 24

Ciphertexts=Get_shifted_text(Shifted_round,ciphertext)

For ct in Ciphertexts:

    For p in plaintexts:

        Distances=distance(ct,p)

For each 5_step in Distances:

    If 5_step in Distances:

        5_step_dict[5_step]= 5_step_dict[5_step]+1

Find max_freq in 5_step_dict for each plaintext

If max(max_freq) larger than min(max_freq):

    Max_freq is correct plaintext

Third way:

Num_rand_char=len(ciphertext)-len(plaintext)

Num_rand_char is no more than 24

Ciphertexts=Get_shifted_text(Num_rand_char,ciphertext)

For each character c in Ciphertexts:

    For each character p in plaintexts:

        Distances.append(distance(c,p))

Sum (Find_top_6_frequent distance from Distances)

maximal sum is correct plaintext.

Test2:

Ciphertexts=Get_shifted_text(Shifted_round,ciphertext)

For w in words:

    Key=guesskey(w,ciphertext)

```
        Decryptedtext= Decrypt_ciphertext(key,ciphertext)
        For word in Decryptedtext:
            For w in words:
                Ratio= difflib.SequenceMatcher(word,w)
                Ratios.append(Ratio)
        Score=Ratios that larger than 0.6
Find maximal score as key
For every position in ciphertext
        For w in words:
            Decrypted_word=decryp(key,ciphtext)
            Ratio= difflib.SequenceMatcher(word,w)
            If ratio>0.7:
                Plaintext=plaintext+w
```