IN , LASEC: Bachelor Project #1

Due on Spring 2016

Pr. Serge Vaudenay

Max Premi

Abstract

This Hill cipher is a polygraphic substitution cipher based on linear algebra , invented by Lester S. in 1929. Each letter is represented by a number modulo 26 , and break the plaintext into blocks of size d and then applies a matrix $d \times d$ to this matrix to yield ciphertext. As it's a linear encryption, it can be simply broken with Know PlainText Attacks. The author takes the previous paper about a new Ciphertext-only Attacks on Hill , and try to improve it's complexity to get a better result that $O(d13^d)$.

The goal of this project is to actually study the algorithm to get the matrix modulo 26 and then to improve the algorithm to get the key matrix modulo 2.

The project is organized as follows:

Contents

Introduction

The motivation of this project is first and foremost to improve the Linear attack on the Hill cipher , by changing the recovery of the key modulo 26 and then see the possible algorithm to improve the FFT.
Let's briefly recall how this attack works.
You get the plaintext modulo 2 , then with the aid of vectors , and bias(X)= $\varphi_X(\frac{2\pi}{p})$ in $\mathbb{Z}/26\mathbb{Z}$, we found correspondence between λ and μ (which is the same vector but for the cipher text). Wa actually get $\mu = (K^T)^{-1} \times \lambda$
Then with this formula and the approximation of all the vector μ , we get the vectors column of the key matrix.
You just need to reorder them with the correlation , you find the last one and first one easily, and you do it recursively to find all the vectors in the correct order.
All this process is described by algorithm 1 at the end of the page.

Key recovery modulo 26

So now that we have the key matrix in $\mathbb{Z}/2\mathbb{Z}$, we can have the plain text in $\mathbb{Z}/2\mathbb{Z}$ using the linearity of the cipher.
To get the key matrix in $\mathbb{Z}/26\mathbb{Z}$, we can use the Chinese Reminder Theorem , but we would get a complexity of $O(13^d)$. In the previous paper , it was believed that it's possible to get the key matrix in $\mathbb{Z}/26\mathbb{Z}$ without considering $\mathbb{Z}/13\mathbb{Z}$.
First of all , we create a hash table using long text , and search mapping between segments of reference text and plain text modulo 2
#(seg in reference) = len(reference text) - n +1 , with n the segment size.
Indeed , if you take the following text : *thisisatest* , with n = 5 , you get the following segment: $thisi, hisis, isisa, sisat, isate, sates, atest$ which is 7 segments $11 - 5 + 1 = 7$

We get the same thing for $\#(strinplain) = len(plaintext) - n + 1$, with n the string size.
Then we define the good mathcings : segments are equals before and after modulo 2 , and bad matching segment which are not equal but equal modulo 2.
We use Rnyi entropy to get the good matching and all matching as it find the collision , with the following formula :
$H_\alpha(X) = \frac{1}{1-\alpha} log_2(\sum_{i=1}^{n} Pr(X = i)^\alpha)$, then when alpha has the value 2 , we just get $-log_2(\sum_{i=1}^{n} Pr(X = i)^2)$ that gives us the probability that a segment equals another one.

For good matching , we have $E(\#goodmatching) = (\#segmentsinreference)x(\#segmentinplaintext)x2^{-H_2(X)}$, as the number of good matching is actually the collision between segment in plaintext and segment in reference text time the entropy of rnyi where two segments are the same.
Then you do the same for $E(\#allmatching)$, the difference is that you do it this way : $E(\#goodmatching) = (\#segmentsinreference) \times (\#segmentinplaintext) \times 2^{-H_2(Xmod2)}$. And indeed you understand that if 2 words modulo 2 are equals, these words are not always equals modulo 26.

For the $E(\#allmatching)$ the calculation is really simple , you must take $(\#segmentsinreference)x(\#segmentinplain$
as we do all the possibles matching.
$H_2(Xmod2) = -log_2(\sum_{i=1}^{n} Pr(X = i)^2)$, where $Pr(X = i)$ declined in $Pr(X = 0)$ and $Pr(X = 1)$
From diverse calculation we always get 0.5^n so E(# all matching) is always equals to $(\#segmentsinreference)x(\#segmentinp$
0.5^n Then to have an idea of the complexity , you do the ratio $\frac{E(\#goodmatchings)}{E(\#allmatchings)}$, you generaly found $\frac{1}{8^n}$
In the following parts , the calculation of $E(\#allmatchings)$ are done again thanks to a Java programm. But to have a better complexity , we need to increase this ratio : $E(\#allmatchings)$ can't be changed so we can only try on $E(\#goodmatchings)$, with different assumptions and calculations.

Experiment

from wiki :

Probabilite de la 1ieme lettre de l'aphabet 0.08167
Probabilite de la 2ieme lettre de l'aphabet 0.01492
Probabilite de la 3ieme lettre de l'aphabet 0.02782
Probabilite de la 4ieme lettre de l'aphabet 0.04253
Probabilite de la 5ieme lettre de l'aphabet 0.12702
Probabilite de la 6ieme lettre de l'aphabet 0.02228
Probabilite de la 7ieme lettre de l'aphabet 0.02015
Probabilite de la 8ieme lettre de l'aphabet 0.06094
Probabilite de la 9ieme lettre de l'aphabet 0.06966
Probabilite de la 10ieme lettre de l'aphabet 0.00153
Probabilite de la 11ieme lettre de l'aphabet 0.00772
Probabilite de la 12ieme lettre de l'aphabet 0.04025
Probabilite de la 13ieme lettre de l'aphabet 0.02406
Probabilite de la 14ieme lettre de l'aphabet 0.06749
Probabilite de la 15ieme lettre de l'aphabet 0.07507
Probabilite de la 16ieme lettre de l'aphabet 0.01929
Probabilite de la 17ieme lettre de l'aphabet 9.5E-4
Probabilite de la 18ieme lettre de l'aphabet 0.05987
Probabilite de la 19ieme lettre de l'aphabet 0.06327
Probabilite de la 20ieme lettre de l'aphabet 0.09056
Probabilite de la 21ieme lettre de l'aphabet 0.02758
Probabilite de la 22ieme lettre de l'aphabet 0.00978
Probabilite de la 23ieme lettre de l'aphabet 0.02361
Probabilite de la 24ieme lettre de l'aphabet 0.0015
Probabilite de la 25ieme lettre de l'aphabet 0.01974
Probabilite de la 26ieme lettre de l'aphabet 7.4E-4

proba sum = 0.9999999999999999
proba sum squared = 0.06549717159999999
proba sum squared 0 = 0.56832 0.32298762240000006
proba sum squared 1 = 0.43167999999999995 0.18634762239999997
Ration of good matching and all matchings=0.1285934407027314
Donc 7,77644

Another site :
Probabilite de la 1ieme lettre de l'aphabet 0.0808
Probabilite de la 2ieme lettre de l'aphabet 0.0167
Probabilite de la 3ieme lettre de l'aphabet 0.0318
Probabilite de la 4ieme lettre de l'aphabet 0.0399
Probabilite de la 5ieme lettre de l'aphabet 0.1256
Probabilite de la 6ieme lettre de l'aphabet 0.0217
Probabilite de la 7ieme lettre de l'aphabet 0.018
Probabilite de la 8ieme lettre de l'aphabet 0.0527

```
Probabilite de la 9ieme lettre de l'aphabet 0.0724
Probabilite de la 10ieme lettre de l'aphabet 0.0014
Probabilite de la 11ieme lettre de l'aphabet 0.0063
Probabilite de la 12ieme lettre de l'aphabet 0.0404
Probabilite de la 13ieme lettre de l'aphabet 0.026
Probabilite de la 14ieme lettre de l'aphabet 0.0738
Probabilite de la 15ieme lettre de l'aphabet 0.0747
Probabilite de la 16ieme lettre de l'aphabet 0.0191
Probabilite de la 17ieme lettre de l'aphabet 9.0E-4
Probabilite de la 18ieme lettre de l'aphabet 0.0642
Probabilite de la 19ieme lettre de l'aphabet 0.0659
Probabilite de la 20ieme lettre de l'aphabet 0.0915
Probabilite de la 21ieme lettre de l'aphabet 0.0279
Probabilite de la 22ieme lettre de l'aphabet 0.01
Probabilite de la 23ieme lettre de l'aphabet 0.0189
Probabilite de la 24ieme lettre de l'aphabet 0.0021
Probabilite de la 25ieme lettre de l'aphabet 0.0165
Probabilite de la 26ieme lettre de l'aphabet 7.0E-4


proba sum = 0.9999000000000001
proba sum squared = 0.06609151
proba sum squared 0 = 0.5657 0.32001649
proba sum squared 1 = 0.4342 0.18852964
Ration of good matching and all matchings=0.12996168115565054
```

Donc 7,69457

Enhancement good matching's ratio

This section will be to increase the ratio found which is actually $\frac{1}{8^n}$
To do so , I'll now consider instead of independent letters , independent blocks of letter. With the help of a Java programm , i'm doing an heuristic search over a very very long text , with different block size.
With the program , we clearly see that there is no way to improve it considering that they are independent.

Study of Faster Fourrier Transform for Algorithm 1

With a fast fourrier Transform (FFT) the complexity is $O(NlogN)$ for N the input size

Deteministic Sparse Fourier Approximation via Fooling Arithmetic Progressions

If we only want to have the few significant Fourrier Coefficient , we can use this.
Here if we gave a threshold τ and an oracle access to a function f , it outputs the τ-significant Fourier Coefficient. This is called SFT and runs in $log(N), \frac{1}{\tau}$
An oracle access to a function take as input x and return the f(x) of the function f.
This algorithm is robust to random noise and local (mean polynomial time)
It's based on partition of set by binary search , you have at the beginning 4 intervals , you test for the two first if the norm of f's Fourier Transform squared is equals to the set_i oracle ouput squared

Meaning more explicitly : $f(J_i)^2 = \sum_{\alpha \in J_i} |f(\alpha)|^2$ If this pass , it will output yes , and we'll be able to continue the algorithm by replacing the J and insert the J_i
The heart of the code is actually to decide which intervals potentially contain a significant Fourier coefficient.
Yes if weight on J , exceeds significant threshold τ , NO if J larger.

Nearly optimal Sparse Fourier Transform

We want here to compute the k-sparse approximation to the discrete Fourier transform of an n-dimensional signal.
There is to time in function of the number of input has at most k non-zero Fourier Coefficient.
In this case , we got $O(k.log(n))$ time , else we have $O(k.log(n).log(\frac{n}{k}))$
The basis is still the same, if a signal has a small number k of non-zero Fourier , the output of this DFT can be represented succinctly using only k coefficient.
What is required , is that the input size n is a power of 2.
This algorithm seems to restrictive and also perform the same in the worst case.

Combinatorial sub linear-Time Fourier Algorithm

Here same thinking , SFT , and getting polynomial time $(k, log(n))$

Algorithm

You hash a reference text.
You take the key matrix that you get from algorithm 1 , find plain text in $\mathbb{Z}/2\mathbb{Z}$, and create an array.
find the list of all matchings : find all pairs(seg,str) such that seg is a segment of plaintext modulo 2 and str $\in hash(seg)$ and save it in a list.
repeat
select d matching form list (you'll get a dxd key matrix)
for each of these matchings (seg_i, str_i)
extract $block_i$ from seg_i and str'_i from str_i,
then find $ciphertext_i$ such that K^{-1} x $ciphertext_i$ mod 2 = $block_i$
solve $ciphertext_i = K * str'_i$ for i=1 to d
compute $K^{-1}*$ciphertext
until it makes sense
number of iteration is $\frac{1}{ratio^{nd}} = 8^{nd}$

The following algorithm is to recover the key matrix in $\mathbb{Z}/2\mathbb{Z}$
Part1:

You require Ciphertext $Y_1, Y_2, ..., Y_n$
for all μ do compute $S_n(\mu) = \sum_{k=1}^{n} (-1)^{\mu.y} \times n_y$ where $n_y = \#\{k; Y_k = y\}$
endfor
set all μ to the d values of μ with largest $S_n(\mu) = bias(\mu.Y)$

Part2:

for all (i,i') do
compute $n_{00}(i,i') = \#\{k < n : (\mu_i.Y_k, \mu'_i.Y_{k+1}) = (0,0)\}$ endfor

set (i_d, i_1) to the first pair with lowest n_{00}

Part3:

```
for all $t = 2$ to $d - 1$ do
for all i $\notin \{i_1, i_2, .., i_{t-1}, i_d\}$ do
compute $n_{00}(i, i') = \#\{k : (\mu_{i_{t-1}}^T Y_k, \mu_i^T Y_k) = (0, 0)\}$
endfor
take i such that $n_{00}$ is minimum and set $i_t = i$
endfor
set $\mu = (\mu_{i1}, \mu_{i1}, ..., \mu_{id})$ and $K = (\mu^- 1)^T$ output K
```

Here to be faster we store n_y in a table and we do a FFT on this table to get S_n. With this operation the total complexity drop from $O(d^2 \times 2^d)$ to $O(d \times 2^d)$ But it seems with some other techniques we could do better.