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, from $A = 0$ to $Z = 25$. The algorithm breaks the plaintext into blocks of size d and then applies a matrix $d \times d$ to these blocks to yield ciphertext blocks. 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 key matrix modulo 2 and then to improve the algorithm to get the key matrix modulo 26.

The project report is organized as follows: Section1 presents the Hill cipher and the work done in the previous report. In section2, the author studies the complexity and try to improve the algorithm to get the key matrix modulo 26 . Section 3 presents the possible enhancement of the FFT of algorithm 1. Section 4 present the possible improvement of the algorithm to recover the matrix key modulo 26. Experimental results and algorithm are presented at the end.

Contents

Introduction

The motivation of this project is, first and foremost, to improve the Linear attack on the Hill cipher, by changing the recovering of the key modulo 26 and then see the possible algorithm to improve the FFT in recovery of matrix key modulo 2.

Indeed, it is known that a brute force attack can be done on the Hill cipher, as it is a Linear cipher, but to have a better complexity and less restrictive resources, improvement have been made.
It is now possible to get a matrix key with minimum length required on ciphertext of $n = 8.96d^2 - O(\log d)$.
This method has been then improved [2] using the divide-and-conquer technique, and eliminating repeated calculation while doing matrix multiplication, and have led to a ciphertext required length of $n = 8,96d^2$.
Eventually, using the Chinese Reminder Theorem [2], the length has been brought to $n = 12.5d^2$, and the complexity to $O(d13^d)$.
By this same Chinese Reminder theorem, it is believed that we can find the key matrix modulo 2 first and then recover the matrix modulo 26 with a lower complexity [1].
It is shown in the previous paper that this matrix modulo 2 can be found in $O(d2^d)$.

Let's briefly describe how this attack works:
If you consider X a random vector constituted of d letters, you can pick a fixed vector $\lambda \in \{0,1\}^*$ and consider the dot-product $\lambda.X$ in $\mathbb{Z}/2\mathbb{Z}$.
Then with the aid of the bias(X)= $\varphi_X(\frac{2\pi}{p})$ in $\mathbb{Z}/26\mathbb{Z}$, we found correspondence between λ and μ (the last is the same vector but for the cipher text). We are searching $bias(\lambda.X)$ and we found $\mu = (K^T)^{-1} \times \lambda$.
Then with this formula and the approximation of all the vector μ, we get the vectors column of the key matrix in $Z/2\mathbb{Z}$.
You just need to reorder them with the correlation, to find the last one and first one easily, and then recursively find all the vectors in the correct order.
All this process is described by algorithm 1 in the Annexe, and is done in a time $O(d2^d)$

This project will present possible improvement of this algorithm to get a lower complexity than the one mentioned before, with the help of Sparse Fourier Transform. Then, a possible enhancement to get the key matrix in modulo 26 will be discussed, as the one presented in the previous paper runs in $O(8^{nd})$.

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 proportional to $O(13^d)$. 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 $\#(seginplain) = len(plaintext) - n + 1$, with n the segment size.
Let a, b be random segment of length n and X the plaintext. We use Rényi entropy, with the following formula:

$$H_\alpha(X) = \frac{1}{1-\alpha}\log_2(\sum_{i=1}^{n} \Pr(X=i)^\alpha)$$

When alpha has the value 2, we just get the following:

$$-\log_2(\sum_{i=1}^{n} \Pr(X=i)^2)$$

that gives us the probability that a segment equals another one as $\sum_{i=1}^{n} \Pr(X=i)^2 = \sum \Pr(a=b)^2$. Rényi entropy represent more generally the quantity of information in the probability of colision of a random variable.

Then we define the good matching : segments are equals before and after modulo 2, and bad matching segment which are not equal but equal modulo 2.

For good matching, we have $E(\#goodmatching) = (\#segmentsinreference) \times (\#segmentinplaintext) \times 2^{-H_2(X)}$, as the number of good matching is actually the collision between segment in plaintext and segment in reference text multiplied by the entropy of rényi of this segment (which represents the rate of collision for a given block X).
Then you do the same for $E(\#allmatching)$, the difference is that you do it this way : $E(\#goodmatching) = (\#segmentsinreference) \times (\#segmentinplaintext) \times 2^{-H_2(X \bmod 2)}$. 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 $(\#segmentsinreference) \times (\#segmentinplaintext) \times 2^{-H_2(X \bmod 2)}$ as we do all the possibles matching.
$H_2(X \bmod 2) = -\log_2(\sum_{i=0}^{1} \Pr(X=i)^2)$, where $\Pr(X \bmod 2 = i)$ declined in $\Pr(X \bmod 2 = 0)$ and $\Pr(X \bmod 2 = 1)$
From the experiment, we always get 0.5^n for $X \bmod 2$ so E(# all matching) is always equals to $(\#segmentsinreference) \times (\#segmentinplaintext) \times 0.5^n$.
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 program. To have a better complexity, we need to increase the ratio of good matching as $E(\#allmatchings)$ can't be changed so we can only try different assumptions and calculations.

The one that is interesting is to consider blocks of letters as being independent from each others, and look at the evolution of the ratio through the growing block size. This is done in Experiment 2.
We finally conclude that it's possible to have a correct ratio for large size block, but the actual algorithm depends too much on the blocksize, and it's therefore impossible to get a correct complexity with the found curve that looks more like

Study of Algorithm to get K_{26}

We are going to try to improve the complexity of algorithm 2 to find another complexity than $O(8^{nd})$.
We want to turn the problem in another way, meaning instead of looking at all possible matching and do all the decryption, try to found the number of good matching we need so that an algorithm can find all this matching and find only one matrix possible.
So we can see the problem as a set of possible equation(which represent the vector column of the matrix), with size n and you want to find m good equation that represent the good vector column of the key matrix. The question is how many m do you need in n so that you can find only one matrix.
Pr(to have at least m good equation)= $\sum_{i=m}^{n} \binom{n}{i}(\frac{1}{13^d})^i(1-\frac{1}{13^d})^{n-i}$
With this formula we actually find that m must be close to $2d$. It is exactly the same condition described in the Berlekamp-Massey algorithm.

Study of Faster Fourier Transform for Algorithm 1

With a fast Fourier Transform (FFT) the complexity is $O(N \log N)$ for N the input size.
But generally, most of the coefficient of a FFT are small or equal to zero, meaning the output of the FFT is sparse. If a signal has a small number k of non-zero Fourier coefficients the output of the Fourier transform can be represented succinctly using only k coefficients. Hence, we can find Fourier Transform algorithm whose run time is sub-linear in the signal size n.
what we want is to enhance the possible FFT on a table called n_y which contains the number of times k where each cipher y appears. So it is a table containing numbers $\in \mathbb{N}$.
The actual probability that all vectors are different is:

$$\prod_{n=0}^{k-1} (26^{-d})(26^d - n)$$

If we suppose that they actually are 26^d ciphers.
But it's not the case, so from our experiment on independent block, we also computed the probability that a block equals another one. We just need to take this value for all the different blocksize, and then, as we assumed all the blocks where independent, put it at the power number of blocks.
Which gives us:

$$(1 - \Pr(a = b))^{\#ciphers}$$

With this formula the bigger the size of block is, the more ciphers we can get and are different for sure. You can see Experiment 3 to look the probability with some blocksize.

Simple and practical algorithm for sparse Fourier transform

This algorithm considers a complex vector x of length n.
It computes the k-sparse Fourier transform in $O(\sqrt{kn} \log^{3/2} n)$, if x is sparse then you find it in exactly $O(klog^2 n)$, but in general estimate x is approximately $O(\sqrt{nk})$
So this algorithm is better if the ratio $\frac{n}{k} \in [2 \times 10^3, 10^6]$, but it's clearly not the best one as recently found are supposed to find it in a lower complexity $(k \log(n))$.
But it can still be used and gives correct results. So we will assume all ciphers are different, and we get all non null component in n_y from the result in experiment 2, of size n, meaning n given ciphers. We find that the Fourier transform only get high picks at the extremities.
So if we consider that the middle are null, we can reduce the complexity.
For example, for blocksize $= 10$ and 6200 ciphertext , the probability that they are all different is $0.99999683297^{6200} = 0.9805$. So we assume they are all different, and find the following FFT:
put the DDT from matlab here
If we actually consider the value from to being zero, we could get a better complexity.

Deterministic Sparse Fourier Approximation via Fooling Arithmetic Progressions

If we only want to have the few significant Fourier Coefficient, we can use this.
Here if we gave a threshold $\tau \in (0, 1]$ 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 (meaning it runs in 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 Fourier Transform squared is equals to the set_i oracle output 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.
The threshold τ can be chosen, with the fact that a α is a $\tau - significant$ Fourier coefficient iff $|\hat{f}|^2 \geq \tau ||f||_2^2$ where $\hat{f} =< f, X_\alpha >$ and $X_\alpha = e^{2\pi i \alpha x/N}$.
If you consider the table n_y of size n with all entries=1, we get $\tau ||f||_2^2 = \tau \times n$
And as $\hat{f}$ got lot of small coefficient and large one on the extremities, they will be some coefficient that will satisfy this equation for small τ. So the complexity will depends on $\frac{1}{\tau}$ and it'll clearly be too big.
Let's take the same example as the previous subsection, the FFT

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

You have a vector A of length $n \gg k$ you identify the k largest frequencies of the transform of A, getting polynomial time $(k, \log(n))$ for the algorithm.

Experiment

Experiment 1:Probability of independent English letters

From the frequency letter given by Wikipédia, in english we got the following result :
Proba sum = 0.9999999999999999
Sum of probability squared = 0.06549717159999999, which corresponds to $(\sum_{i=0}^{25} \Pr(i = y)^2)^n, y \in \{alphabet\}$
Sum of probability that gives 0 modulo 2 squared = 0.32298762240000006 which corresponds to $(\sum_{i=0}^{25} \Pr(i = 0)^2)^n, i \in \{alphabet \bmod 2\}$
Sum of probability that gives 1 modulo 2 squared = 0.18634762239999997 which corresponds to $(\sum_{i=0}^{26} \Pr(i = 1)^2)^n, i \in \{alphabet \bmod 2\}$
Ration of good matching and all matching=0.1285934407027314^n
So $\frac{1}{7{,}77644^n}$.

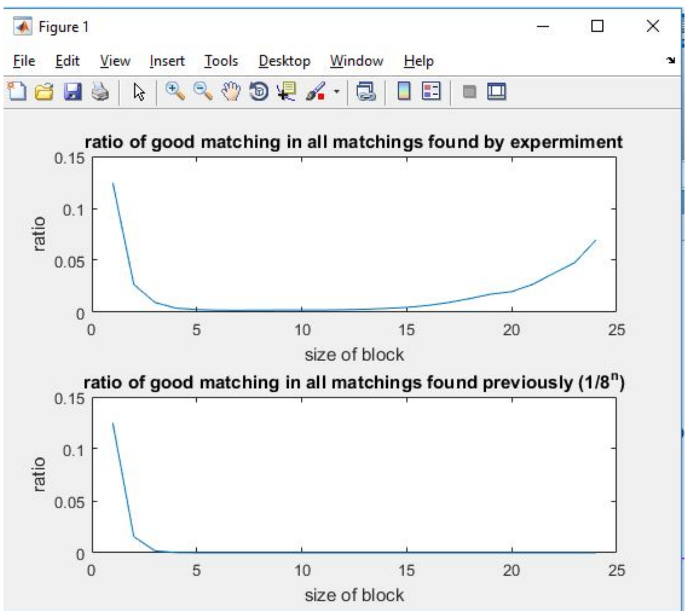
Another site [8], with a total number of 100000 letters composed with texts from Edgar Allan Poe, Arthur Conan Doyle, and 4 articles from encyclopedia Encarta 95:

proba sum = 0.9999000000000001
sum of probability squared = 0.06609151 which corresponds to $(\sum_{i=0}^{25} \Pr(i = y)^2)^n, y \in \{alphabet\}$

sum of probability that gives 0 modulo 2 squared = 0.32001649 which corresponds to $(\sum_{i=0}^{25} \Pr(i = 0)^2)^n, i \in \{alphabet \bmod 2\}$
sum of probability that gives 1 modulo 2 squared = 0.18852964 which corresponds to $(\sum_{i=0}^{25} \Pr(i = 1)^2)^n, i \in \{alphabet \bmod 2\}$
Ration of good matching and all matching=0.12996168115565054^n
So $\frac{1}{7{,}69457^n}$.

Experiment 2:Probability if you consider blocks of size d

So calculation are done on a text of approximately 860000 characters to see the evolution of the ratio good matching/bad matching.
A program is ran to see the evolution for a block size between 1 and 25, and give the ratio, thanks to the probability that a block appears. It is completely heuristic as it's just counting the number of block that appears and do some manipulation with it. So the basic is to choose a block size, then it'll count every different blocks that appears modulo 26 and modulo 2. Then it'll compute the probability that a good matching happen with the following : $\sum_{X \in block}(\frac{\#X-1}{\#block-1})^2$
You do the exact same thing with X in modulo 2, to get the probability of all matching, and then you compute the ratio $\frac{E(\#goodmatchings)}{E(\#allmatchings)}$
With this, the evolution of the ratio in function of the block size looks like this:



So we can see that the ratio follow the $\frac{1}{8^n}$ until blocksize 15, then it goes up again to almost match 1/8 for blocksize 24. With the current algorithm and this size of block the number of iteration would be 8^d and as $d = 24$ is really large it's still not effective.
Even if the ratio do not behave as previously thought, the complexity stay too high for reasonable blocksize between 8 and 14, and if you go in higher blocksize, the ratio is good but the complexity depends on a ratio power the blocksize, so it will not be good enough to be taken in account.

Experiment 3:Blocksize and diffence of block in n ciphers

blocksize	Probability that 2 blocks are different	# maximal of ciphers for Pr > 0.95
1	(0.9367250929)	less than 1
2	(0.99290120378)	7
3	(0.99868328054)	38
4	(0.99971599715)	180
7	(0.99998010694)	2578
10	(0.99999683297)	16195
12	(0.99999899833)	51207
14	(0.99999959261)	125907
16	(0.99999985551)	354995

Indeed, it's obvious that the bigger the block is, the lower is the chance that you have another one with the same value. For blocksize 1, you cannot hop to have a good probability to get another block different as there are only 26 possibilities. You just have to resolve the following:$Probability^{\#ciphers} > 0.95 = n < \frac{\log(0.95}{\log(Probability)}$.

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 matching: 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 $d \times d$ key matrix)
  for each of these matchings $(seg_i,\ str_i)$ **do**
    extract $block_i$ from seg_i and str'_i from str_i,
    then find $ciphertext_i$ such that K^{-1} x $ciphertext_i$ mod $2 = block_i$
  end for
  solve $ciphertext_i = K * str'_i$ for i=1 to d
  compute $K^{-1}*$ciphertext
until decryption make 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:
Require: Ciphertext $Y_1, Y_2, ..., Y_n$
Ensure: K(mod2)
for all μ **do**
  compute $S_n(\mu) = \sum_{k=1}^{n} (-1)^{\mu.y} \times n_y$ where $n_y = \#\{k; Y_k = y\}$
end for
set all μ to the d values of μ with largest $S_n(\mu) = bias(\mu.Y)$
Part2:
for all (i, i') **do**
  compute $n_{00}(i, i') = \#\{k < n : (\mu_i.Y_k, \mu'_i.Y_{k+1}) = (0, 0)\}$
end for
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 $n_{00}(i, i') = \#\{k : (\mu^T_{i_{t-1}} Y_k, \mu^T_i Y_k) = (0, 0)\}$
  end for
  take i such that n_{00} is minimum and set $i_t = i$
end for
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.

References

[1] Alina, Matyukhina. *Cryptanalysis of the Hill Cipher.*

[2] S. Shazaei, S. Ahmadi. *Ciphertext- only attack on* $d \times d$ *Hill in* $O(d13^d)$.

[3] Akavia, A. *Deterministic Sparse Fourier Approximation via Fooling Arithmectic Progressions.*

[4] Akavia, A., Goldwasser, S., Safra, S. *Proving Hard-Core Predicates Using List Decoding.*

[5] Hassanieh, H., Indyk, P., Katabi, D., Price, E. *Nearly optimal sparse Fourier transform.*

[6] Hassanieh, H., Indyk, P., Katabi, D., Price, E. *Simple and practical algorithm for sparse Fourier transform.*

[7] Iwen, M.A. *Combinatorial Sublinear-Time Fourier Algorithms.*

[8] http://www.nymphomath.ch/crypto/stat/anglais.html