




DECEMBER 12, 2017

## CSC 412 PORTFOLIO

NAIK, SOHAM G.  
SOUTH DAKOTA SCHOOL OF MINES AND TECHNOLOGY  
BACHELOR OF COMPUTER SCIENCE



## Abstract

This portfolio summarizes what I had learned in my CSC 412 (Cryptography) class in the fall 2017 semester. It is divided into four parts, which are, Classical Cryptosystems, Basic Number Theory, Data Encryption Standard and the RSA algorithm. Each part explains the corresponding algorithms with theorems and examples from the book.

I chose Python 2 to write all the modules because I found the syntax to be very simple as compared to the other programming languages I know. Cryptography is definitely a field I would like to explore in the near future as every day scientists and mathematicians come up with new algorithms as people find ways to crack the previous ones.

Note that I have borrowed a lot of information from our textbook while describing most of the algorithms.

## Table of Contents

<b>Abstract .....</b>	<b>1</b>
<b>1 Classical Cryptosystems .....</b>	<b>3</b>
1.1 Frequency Calculation .....	3
1.2 Affine Cipher.....	3
1.3 Affine Cipher Attacks .....	4
1.4 Vigenere Cipher .....	4
1.5 Vigenere Cipher Attacks .....	4
1.6 Block Cipher .....	5
<b>2 Basic Number Theory .....</b>	<b>6</b>
2.1 GCD.....	6
2.2 Extended GCD.....	6
2.3 Modular Inverse .....	6
2.4 Primitive Root.....	6
2.5 Miller-Rabin Primality Test.....	7
2.6 Random Prime Generator .....	7
2.7 Factoring Algorithms .....	7
2.7.1 Fermat's Method.....	7
2.7.2 Pollard Rho .....	8
2.7.3 Pollard P-1 .....	8
2.8 Shanks Square Forms .....	8
2.9 Sieve of Sundaram.....	9
<b>3 Data Encryption Standard .....</b>	<b>9</b>
3.1 Simplified DES System .....	9
3.2 Differential Cryptanalysis for Three rounds .....	10
3.3 64 DES.....	10
<b>4 RSA Algorithm .....</b>	<b>11</b>
4.1 RSA algorithm.....	11
<b>References .....</b>	<b>12</b>

# 1 Classical Cryptosystems

Cryptography is an essential component of cybersecurity. The need to protect sensitive information has risen. In this section, I have covered some of the older cryptosystems that were used before the advent of the computer. These are not used any more as they can easily be solved by hand or can be broken with modern technology.

Before I start with, I would like to mention some of the conventions I have followed while writing the code

- *Plaintext* is the text to be encoded and CIPHERTEXT is the encoded text.
- *Plaintext* is written in lowercase letters and CIPHERTEXT is written in uppercase letters (except in computer problems).
- For simplicity and calculations purposes, punctuations and special characters are omitted. The programs also specifically only encrypt and decrypt messages which only contain letters.
- The letters of the alphabet are assigned numbers as follows:  
a:0, b:1, c:2, d:3, e:4, f:5, g:6, h:7, i:8, j:9, k:10, l:11, m:12, n:13, o:14, p:15, q:16, r:17, s:18, t:19, u:20, v:21, w:22, x:23, y:24, z:25  
Note that I start with a = 0 as this helps me make my calculations easier.

I have implemented the following modules and they can be found in *cryptfunc.py*.

## 1.1 Frequency Calculations

In most of the classical cryptosystems, frequency analysis of the message was an integral part of the decryption process as it helped determine the frequency of each letter in the ciphertext and compare it to the frequencies of the letters in the English language. If it was an encryption algorithm which encrypted each letter individually, then one could conclude that 'e' was encrypted to the most occurring letter in the ciphertext.

## 1.2 Affine Ciphers

This was one of the simplest ciphers used back in the days. One chooses two integers  $\alpha$  and  $\beta$ , with the greatest common divisor of  $\alpha$  and 26 equal to 1. Consider the function (called the affine function)

$$X = \alpha x + \beta \pmod{26}$$

For example, let  $\alpha = 9$  and  $\beta = 2$ , and so we are working with  $9x + 2$ . Take a plaintext c (=2). It's encrypted to  $9*2 + 2 \equiv 20 \pmod{26}$ , which is the letter U. Using the same function, we obtain

$$\text{affine} \rightarrow \text{CVVWPM}$$

Ciphertext is decrypted using the same equation, and after rearrangement, it looks like this:

$$x = (1/\alpha) * (X - \beta) \pmod{26}$$

Here, one would have to compute the multiplicative inverse of  $\alpha$ . In this example,  $\alpha = 9$ , and  $9 * 3 \equiv 27 \equiv 1 \pmod{26}$  and so 3 is the multiplicative inverse of 9 (mod 26). If we try to decrypt  $V (=21)$  using the above  $\alpha$  and  $\beta$  values, we get  $3 * 21 + 20 \equiv 83 \equiv 5 \pmod{26}$  which is the letter f.

### 1.3 Affine Cipher Attacks

I have only implemented two different types of attacks in my program, and they are:

- Ciphertext only: One basically does an exhaustive search through all of the 312 keys and tries each one of them on the text and keeps the one which provides a meaningful text.
- Chosen Plaintext: Choose *ab* as the plaintext. The first character of the ciphertext will be  $\alpha * 0 + \beta = \beta$ , and the second will be  $\alpha + \beta$ . The key can be computed from this.

### 1.4 Vigenere Cipher

The key for this encryption is a vector whose entries are integers from 0 to 25, for example  $k = (8, 13, 3)$ . Often, the key corresponds to a word is *end*. Each letter in the plaintext is shifted forward by a certain value in the key. The security of the system depends on the fact that neither the keyword nor its length is known. Let's consider an example.

(plaintext): *h e l l o h o w l s y o u r d a y g o l n g*

(key):        8 13 3 8 13 3 8 13 3 8 13 3 8 13 3 8 13 3 8

(ciphertext): P R O T B K W J L A L R C E G I L J W V Q O

As we can see, *h* (=7) is shifted 8, which yields 15 (=P), and so on. Decryption follows a similar process wherein each letter is shifted backwards by a certain value in the key. Frequency analysis cannot be done on this cipher because each letter is shifted by a different value and so often times, frequencies of the least occurring letters may be higher than the most occurring ones.

### 1.5 Vigenere Cipher Attacks

One has to follow two steps while decrypting a message encrypted using this cipher.

- First step: Find Key Length  
Take the ciphertext and write it on a paper, and on another strip write the same message, but each letter displaced to the right by a number. Mark a \* each time a letter and the one below it are the same, and count the total number of coincidences. The displacement with the most number of coincidences will usually yield the key length. One should try to decrypt using this key, and if no meaningful text is received, then another key whose displacement is also high should be used.

- Second step: Finding the Key

In my program, I have implemented the second method from the book. I'll be working with an example given in the book, and my function in the program also uses this text to explain the algorithm. Note that key length was 5 in the book. To decrypt:

To find the first element of the key, count occurrences of the letters in the 6<sup>th</sup>, 11<sup>th</sup> .. positions, as before, and put them in a vector. We get this:

$$V = (0, 0, 7, 1, 1, 2, 9, 0, 1, 8, 8, 0, 0, 3, 0, 4, 5, 2, 0, 3, 6, 5, 1, 0, 1, 0)$$

The first entry gives the number of occurrences of A, the second gives the number of occurrences of B, etc. Then divide all elements in this vector by the total number of letters counted, in this case it was 67, and so after dividing one gets.

$$W = (0, 0, 0.1045, 0.0149 \dots 0.0149, 0)$$

Since we know that the key length is 5, the 1<sup>st</sup>, 6<sup>th</sup>, 11<sup>th</sup>, .. letters in the ciphertext were all shifted by the same amount, and so, they represent a random sample of letters. Their frequencies are given by vector  $W$ , should approximate the  $A_i$  where  $i$  is the shift caused by the first element in the key. Now  $i$  has to be determined. Recall that  $A_i * A$  is largest when  $i = j$ , and that  $W$  approximates  $A_i$ . If we compute,  $W * A_j$  for  $0 \leq j \leq 25$ , the maximum value occurs when  $j = i$ . Here the dot products are:

$$0.025, 0.039, 0.0713, 0.0388 \dots 0.0349$$

The largest value is the third, which equals  $W * A_2$ . There the guess is that the first shift is 2.

Now calculate a new  $W$  by finding frequencies for 3<sup>rd</sup>, 8<sup>th</sup>, 13<sup>th</sup> .. letters and repeat this process. You will find that the maximum value is the 4<sup>th</sup> one, which is 0.0624, which equals  $W * A_3$ . Therefore, the best guess is that the shift is 3.

Repeat this five (key length) times, and you will get the key as *c, o, d, e, s*.

## 1.6 Block Cipher

Like the name says, this cipher encrypts a blocks of letters from a message at a time. This way, it does not fall for frequency analysis. It basically works like this:

First choose an integer  $n$  and construct a  $n \times n$  matrix. Then make groups of  $n$  letters of the message. If the last group does not have enough letters, then append  $x$  until it has enough. Next, multiply each group of letters with the  $n \times n$  matrix and the resulting matrix is your ciphertext.

To decrypt, first check if the greatest common divisor of the determinant of  $m$  and 26 is 1 or not. If it is, then compute the inverse of  $n \times n$ . Simply multiply the groups of letters with this matrix and the resulting groups of matrices mod 26 will give you numbers of the letters.

My function uses a fixed number  $n = 3$  and the  $n \times n$  matrix =  $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 11 & 9 & 8 \end{bmatrix}$ .

$\text{GCD}(\det(m), 26) = 1$ . Lets try encrypting *blockcipher*. This becomes:

*b l o c k c i p h e r x*

1 11 14 2 10 2 8 15 7 4 17 23

Note that an x was added to fill in the extra space. This gives the cipher text

RBZMUEPYONOM

## 2 Basic Number Theory

This section contains mathematical algorithms used in various functions in the program. I have implemented these modules in the file *cryptomath.py*.

### 2.1 GCD

Stands for Greatest Common Divisor, and commonly denoted by *gcd*. This is the largest number that divides two numbers, *a* and *b*.

$$\text{gcd}(9, 3) = 3, \text{gcd}(20, 21) = 1$$

### 2.2 Extended GCD

Extended Euclidean is an extension to the Euclidean Algorithm, and computes, in addition to the GCD of integers *a* and *b*, also the coefficients of Bezout's identity, which are integers *x* and *y* such that  $ax + by = \text{gcd}(a, b)$

$$\text{extendedgcd}(30, 20) = 10, -1, 1$$

where  $\text{gcd}(30, 20) = 10$  and -1 and 1 satisfy Bezout's identity.

### 2.3 Modular Inverse

Inverse modulo to a number *a* to a base *n* is a number *b* such that  $a*b \equiv 1 \pmod{n}$

For example: inverse of 3 mod 26 is 9 since  $3*9 \equiv 27 \equiv 1 \pmod{26}$

### 2.4 Primitive Root

In general, when *p* is a prime, a primitive root mod *p* is a number whose powers yield every non-zero value mod *p*. For example 3 is a primitive root modulo 7, but not modulo 11 because

$$3^0 \equiv 1, 3^1 \equiv 3, 3^2 \equiv 2, 3^3 \equiv 6, 3^4 \equiv 4, 3^5 \equiv 5, 3^6 \equiv 1 \pmod{7}$$

All possible values were obtained. Whereas modulo 11

$$3^0 \equiv 1 \equiv 3^5 \pmod{11}$$

## 2.5 Miller –Rabin Primality Test

*Let  $n > 1$  be an odd integer. Write  $n - 1 \equiv 2^k m$  with  $m$  odd. Choose a random integer  $a$  with  $1 < a < n - 1$ . Compute  $b_0 \equiv a^m \pmod{n}$ . If  $b_0 \not\equiv \pm 1 \pmod{n}$ , then stop and declare that  $n$  is probably prime. Otherwise, let  $b_1 \equiv b_0^2 \pmod{n}$ . If  $b_1 \equiv 1 \pmod{n}$ , then  $n$  is composite (and  $\gcd(b_0 - 1, n)$  gives a non-trivial factor of  $n$ ). If  $b_1 \equiv -1 \pmod{n}$ , then stop and declare that  $n$  is probably prime. Otherwise, let  $b_2 \equiv b_1^2 \pmod{n}$ . If  $b_2 \equiv 1 \pmod{n}$ , then  $n$  is composite. If  $b_2 \equiv -1 \pmod{n}$ , then stop and declare that  $n$  is probably prime. Continue in this way until stopping or reaching  $b_{k-1}$ . If  $b_{k-1} \not\equiv -1 \pmod{n}$ , then  $n$  is composite.*

This algorithm is generally used for large numbers. My function accepts two arguments,  $n$  and  $r$ .  $n$  being the number to be tested and  $r$  is the number of rounds. The probability of the test being correct is  $\frac{3}{4}$ , and so that the probability of success increases, I provide the 'number of rounds' argument, and that's the number of times this algorithm will be performed. For example, if  $r = 4$ , probability of success will be  $(\frac{3}{4})^4 > \frac{3}{4}$ .

This function returns 0 for composite, 1 probably prime and 2 for definitely prime. I have hardcoded some small primes so that it does not conduct the test on them.

`is_prime(4, 4)` returns 0 since 4 is not a prime

`is_prime(32416190071, 4)` returns 1 since it may be a prime

`is_prime(5, 4)` returns 2 since 5 is in the list of hardcoded primes

## 2.6 Random Prime Generator

This generates a random prime number. My function first creates a random number between  $2^{b+1}$  and  $2^{b-1}$  and then runs it through the Miller-Rabin Primality test. It does this until a prime number is obtained or it tries one thousand times. The time constraint is put so that it does not go into an infinite loop.

Random number generators have various applications, RSA being a very important one, wherein two big prime numbers are generated, and later an encrypting exponent.

`random_prime(40) = 1993319282051`

## 2.7 Factoring Algorithms

Factorizing is extremely important in Mathematics as it allows us to solve various types of problems like polynomial inequalities and quadratic equations. I have implemented three types of factoring algorithms in my program which are *Fermat's Method*, *Pollard Rho* and *Pollard P-1*.

### 2.7.1 Fermat's Method

Here, the idea is to express  $n$  as a difference of two squares:  $n = x^2 - y^2$ . Then  $n = (x+y)(x-y)$ . For example, let's consider  $n = 295927$ . Compute  $n + 1^2$ ,  $n + 2^2$  .. until we find a square. In this case,



$295927 + 3^2 = 295936 = 544^2$ . Therefore,  $295927 = (544 + 3)(544 - 3) = 547 \cdot 541$ .

Note that this method only works for  $n=p \cdot q$  if  $p$  and  $q$  are very close to each other.

$\text{fermat}(456789) = [3541, 129]$

## 2.7.2 Pollard Rho

This algorithm uses a small amount of space, and its running time is proportional to the square root of the size of the smallest prime factor of the composite number being factorized. Here is the pseudo code for the algorithm,

if  $n$  is prime: return  $[1, n]$

$x \leftarrow 2; y \leftarrow 2, d \leftarrow 1$

while  $d = 1$ :

$x \leftarrow g(x)$

$y \leftarrow g(x)$

$d \leftarrow \gcd(|x-y|, n)$

if  $d=n$ :

return failure

else:

return  $d$

$g(x) = (x^2 + 1) \bmod n$

If a failure is reached, one can re-run the algorithm by using a different  $g(x)$  function.

$\text{pollard\_rho}(56789345) = [5, 11357869]$

## 2.7.3 Pollard P-1

*Choose an integer  $a > 1$ . Often  $a = 2$  is used. Choose a bound  $B$ . Compute  $b \equiv a^{B!} \pmod{n}$  as follows. Let  $b_1 \equiv a \pmod{n}$  and  $b_j \equiv b_{j-1}^i \pmod{n}$ . Then  $b_B \equiv b \pmod{n}$ . Let  $d = \gcd(b-1, n)$ . If  $1 < d < n$ , we have found a non-trivial factor.*

Bound  $B$  is chosen such that the algorithm is not too slow and it has a decent chance of being successful.

$\text{pollard\_p1}(123456789) = [3, 41152263]$

## 2.8 Shanks Square Forms

I have pasted a link to the pdf file I referred to as the explanation was to the point.

<https://www.saylor.org/site/wp-content/uploads/2012/07/Shanks-square-forms-factorization.pdf>

$\text{shanks}(11111111, 1) = [1111, 10001]$

## 2.9 Sieve of Sundaram

This is a simple deterministic algorithm which produces all prime numbers smaller than or equal to  $n$ . One first starts with a list of 1 to  $n$ , and removes all numbers of the form  $i + j + 2*i*j$ , where  $i$  and  $j$  are integers and  $1 \leq i \leq j$ . The remaining numbers are doubled and incremented by one, giving a list of the odd prime numbers below  $2n+2$ . It sieves out the composite numbers, but even numbers are not considered. It crosses out  $i + j(2i+1)$  for  $1 \leq j \leq \text{floor}(k/2)$ .

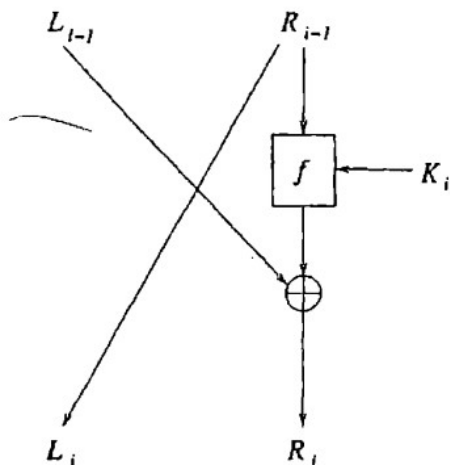
`sieve_sundaram(2) = [2, 3, 5, 7, 11, 13, 19]`

## 3 Data Encryption Standard

The DES is a block cipher, namely, it breaks the plaintext into blocks of 64 bits, and encrypts each block separately. It is widely used for the encryption of electronic data. It was one of the most influential ciphers in the history of cryptography. I have implemented this in the file *des.py*.

### 3.1 Simplified DES

I have implemented a four round mini version of the DES which encrypts 12 bits. Each round of this algorithm looks something like this.



$L_i = R_{i-1}$ ,  $R_i = L_{i-1} \text{ XOR } f(R_{i-1}, K_i)$ , where  $L_i$  is the left 6 bits for the  $i^{\text{th}}$  round, and  $K_i$  is the 8 bit key generated for the  $i^{\text{th}}$  round generated from the original key  $K$ .

The  $f$  function accepts a 6 bit and 8 bit number and returns a 6 bit encoded message which is merged with the 6 bits obtained from the previous round. It first expands the

6 bit input to 8 bits and XOR's it with the other number ( $K_i$ ). It then passes the first 4 bits of this number to S - Box 1 and the last 4 bits to S - Box 2 and attaches both the messages together. This is the result it gives out.

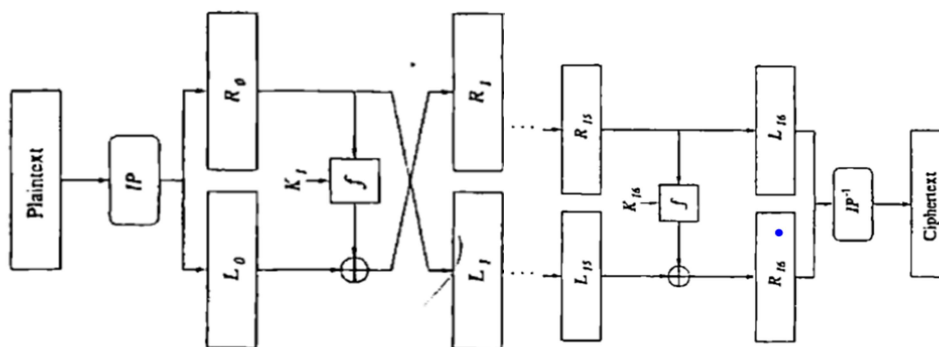
## 3.2 Differential Cryptanalysis for Three Rounds

This is a general form of cryptanalysis applicable to block ciphers. It uses different inputs and analyzes all the corresponding outputs to compute the key. A thorough explanation is given here : [https://en.wikipedia.org/wiki/Differential\\_cryptanalysis](https://en.wikipedia.org/wiki/Differential_cryptanalysis)

## 3.3 64 - DES

The 64 - DES accepts a block of 64 bits. The key has 56 bits, but is expressed as a 64 bit string. It basically consists of 3 steps. Let the message be  $m$ :

- 1) The bits of the  $m$  are permuted by a fixed initial permutation to obtain  $m_0 = IP(m)$ . Then write  $m_0 = L_0R_0$  where  $L_0$  is the first 32 bits and  $R_0$  are the last 32 bits.
- 2) For  $1 \leq i \leq 16$ , perform the following:  
 $L_i = R_{i-1}$ ,  $R_i = L_{i-1} \text{ XOR } f(R_{i-1}, K_i)$ , where  $K_i$  is a string of 48 bits obtained from the key  $K$ .
- 3) Switch left and right to obtained  $R_{16}L_{16}$ , then apply inverse of the initial permutation to get ciphertext  $c = IP^{-1}(R_{16}L_{16})$



The implementation of the full 64\_DES is done in this program. A thorough explanation can be found here:

[https://en.wikipedia.org/wiki/Data\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Data_Encryption_Standard)

## 4 RSA

RSA is an algorithm used by modern computers to encrypt and decrypt messages. It is an asymmetric cryptographic algorithm and easy to understand. There are two different keys, and one of them made public, thus, this is also called public key cryptography. I have implemented this in the file *rsa.py*.

### 4.1 RSA Algorithm

This algorithm consists of the following:

- 1) Choose two primes  $p$  and  $q$  and compute  $n = p * q$
- 2) Compute  $e$  such that  $\gcd(e, (p-1)(q-1)) = 1$
- 3) Compute  $d$  such that  $d * e \equiv 1 \pmod{(p-1)*(q-1)}$
- 4) Make  $n$  and  $e$  public, and keep  $p, q, d$  secret.
- 5) Encrypt  $m$  as  $c \equiv m^e \pmod{n}$  and send  $c$  back.
- 6) Decrypt by computing  $m \equiv c^d \pmod{n}$

A sample run of my function for  $m = \text{"soham"}$  would produce an output similar to this:

```
p: 2024192596229
q: 1921533971251
n: 388955483008782337012479
(p-1)(q-1): 3889554838004836610445000
e: 44963
d: 3122818358470985065652027
encrypted msg: 65888498382633727783004
```

Note that my program will not produce the same numbers on a second run with this message because  $p$  and  $q$  is randomly generated in the function.

## References

Trappe, Wade, and Lawrence C. Washington. Introduction to Cryptography: with Coding Theory. 2nd ed., Pearson Prentice Hall, 2006.