

Cryptography Final Project

Manasi Paste

16th December 2019

Contents

1	Introduction	2
2	Cryptomath Library	3
2.1	Greatest Common Divisor	3
2.1.1	Example	3
2.2	Extended GCD	4
2.2.1	Example	4
2.3	Finding Modular Inverse	4
2.3.1	Example	5
2.4	Power Modulus	5
2.5	Modular Square root	6
2.6	Primality Check	6
2.7	Random Prime	8
2.8	Factorization	8
2.8.1	Pollard Rho	9
2.8.2	Fermat's Method	10
2.8.3	Pollard P-1 Method	10
2.8.4	Shanks Square Form	11
2.8.5	Factorize Function	13
2.9	Testing CryptoMath Library	14
3	Classical Cryptosystems	14
3.1	Affine Cipher	14
3.1.1	Example	14
3.1.2	Code	15
3.1.3	Weaknesses	15
3.1.4	Attack	16
3.2	Vigenere Cipher	16
3.2.1	Example	16
3.2.2	Code	17
3.2.3	Weakness	18
3.2.4	Attack	18
3.3	ADFGX Cipher	19

3.3.1	Example	20
3.3.2	Code	21
3.3.3	Weakness	22
3.4	Frequency Calculation	22
4	DES	22
4.1	Simplified DES Algorithm	23
4.1.1	Permutation Tables and S Boxes	23
4.1.2	S-DES Diagram	24
4.1.3	Key Generation Code	24
4.1.4	Encryption Code	26
4.1.5	S Box look up:	26
4.1.6	Expansion function	27
4.1.7	Feistel's function	27
4.2	64-DES	28
4.2.1	Key Generation Code	29
4.2.2	f Function	31
4.2.3	S Box Look Up	32
4.2.4	Encryption	32
4.2.5	DES Weakness	33
4.3	Differential Cryptanalysis for Three Rounds	33
5	RSA	35
5.1	RSA Key Generation	35
5.2	RSA Encryption and Decryption	36
6	DES and RSA Hybrid	36
7	Sources	37

1 Introduction

This document contains description of different functions and their code, which are relevant in the field of cryptography. These functions are solved according to what was learned in the course CSC 412 for SDSMT. Most of the algorithms in this project are from "Introduction to Cryptography with Coding Theory" by Wade Trappe and Laurence C. Washington textbook. Since I was very interested into cryptography I chose DES and RSA hybrid(atleast that what I call it) and I have added it to this program too. The document tries to describe all the functions and algorithms that are coded on Github:

<https://github.com/manasi1401/cryptography>. I have made an attempt to explain the algorithm either via step wise explanations, examples, diagrams or simply the code.

2 Cryptomath Library

2.1 Greatest Common Divisor

To calculate GCD between two numbers there are two ways to do it. One we can find all prime factors and find a set of intersecting factors. Another way is Euclidean Algorithm. Algorithm:

- x should bigger than y
- x is equal to y and y is equal to $x\%y$
- repeat till y is 0
- x will have the value of gcd between x and y

Time complexity of this algorithm is $O(\lg(a))$ where a is minimum between a and b.

```
def gcd(x, y):  
    """  
    Calculate greatest common divisor between two numbers  
    :param x: first number  
    :param y: second number  
    :return: gcd(x,y)  
    """  
    if x < y:  
        gcd(y, x)  
  
    while y != 0:  
        x, y = y, x % y  
  
    return x
```

2.1.1 Example

Let x = 50 and Y =15

X	Y
50	15
15	5
5	0

Therefore $\text{gcd}(50,15) = 5$

2.2 Extended GCD

Extended Euclidean Algorithm attempts to find $\text{GCD}(a,b)$ by finding x and y in the following equation.

$$ax + by = \text{gcd}(a,b)$$

```
def extendedgcd( a, b):
    """
    Calculated GCD using extended eucledian method
    :param a: first number
    :param b: second number
    :return: egcd(a,b)
    """
    if b == 0:
        return a, 1, 0
    else:
        g, x, y = extendedgcd(b, a % b)
        return g, y, x - (a//b)*y
```

2.2.1 Example

Let's use same example $a = 50$ and $b = 15$

G	X	Y
	50	15
	15	5
	5	0
5	1	0
5	0	$1 - (15/5) * 0 = 1$
5	1	$0 - 50/15 * 1 = -3$

2.3 Finding Modular Inverse

To find Modular Inverse for $a\%n$ we need to find x in $a * x + n * y = 1\%n$. We add the term $n*y$ since we know the modular n of that term will be zero. We can calculate x by using extended GCD function. We return $x\%n$ to ensure x is smaller than n .

```
def findModInverse(a, n):
    """
    Find  $a^{-1} \text{ mod } n$ 
    :param a: base number
    :param n: number that takes the mod
    :return:  $a^{-1} \text{ mod } n$ 
    """
    g, x, y = extendedgcd(a, n)
```

```

if g != 1:
    return -1
else:
    return x % n

```

2.3.1 Example

Let $n = 26$ and $a = 5$

$$g, x, y = \text{extendedgcd}(5, 26) = 1, -5, 1$$

$$5x + ny = g$$

Taking mod n

$$5x \% n + 0 = 1 \% n$$

$$x \% n = 5^{-1} \% n = -5 \% 26 = 21 \% 26$$

2.4 Power Modulus

One way to calculate $a^{d \% n}$ is to multiply a with itself d times and then take a modulus. But when numbers are large such computations may not be handled easily by the program. So this functions loops only till the biggest 1 bit in the d .

- if last bit of d is 1 then update $r = r * a \% n$
- shift d right by 1 bit
- update $a = a * a \% n$
- repeat this until d is zero

```

def powerModulus(a, d, n):
    """
    Calculate a^d mod n
    :param a: base
    :param d: power
    :param n: number to take a mod with
    :return: a^d mod n
    """
    r = 1
    a = a % n
    while d > 0:
        if d & 1:
            r = (r*a) % n
        d = d >> 1
        a = (a*a) % n
    return r

```

2.5 Modular Square root

This function finds the x such that $x^2 = a \bmod n$. The algorithm works as following:

- make sure $a \in \mathbb{Z}_n$ by $a = a \bmod n$
- calculate $a^{(n+1)/4} \bmod n$
- if x is not the root then check $-x$ or $(n-x)$

```
def compute_sqr_root(a,n):
    """
    Helper function to compute square root of a mod n
    :param a: base
    :param n: modular n
    :return: x where  $x^2 = a \bmod n$ 
    """
    # p = 3 mod 4
    if n % 4 != 3:
        return 0

    # make sure a < n
    a = a % n
    #  $x = a^{(n+1)/4} \bmod n$ 
    x = powerModulus(a, int((n+1)/4), n)
    if (x*x) % n == a:
        return x
    # if x has no roots then -x has roots
    x = n - x
    if (x*x) % n == a:
        return x

    return 0
```

2.6 Primality Check

To check if a number is prime or not, I used Miller-Rabin primality test. Miller Rabin Algorithm is as follows:

- Express $n-1$ as $2^k m$
- Pick a random $1 < a < n-1$
- Compute $b_0 = a^m \bmod n$
- if $b_0 = +1$ or $-1 \bmod n$ then its a prime; exit if not
- Compute $b_1 = b_0^2 \bmod n$ if $b_1 = 1 \bmod n$ then it is a composite; exit

- if $b1 = -1 \bmod n$ it is probably a prime; exit
- if not continue $b2 = b1^2 \bmod n$ and keep testing

```
def is_prime(n):
    """
    Miller Rabin primality test
    :param n: number to be tested
    :return: True or false
    """
    # check its not 1 or even or 3, 5, 7
    if n == 1:
        return False
    if n == 2 or n == 3 or n == 5 or n == 7:
        return True

    if n % 2 == 0:
        return False
    # n-1
    m = n - 1
    k = 0
    while m % 2 == 0:
        k = k+1
        m = m/2.0
    # n-1 = 2^k * m
    # Testing multiple times
    for i in range(5):
        if computeMiller(int(m), k, n) is False:
            return False

    return True

def computeMiller(m, k, n):
    """
    perform Millers test to check for Primality
    :param m: m from n-1 = 2^k * m
    :param k: k from n-1 = 2^k * m
    :param n: Number to be tested
    :return:
    """
    a = random.randint(1, n-1)
    # b = a^m mod n
    b = powerModulus(a, m, n)
    # if b = 1 or -1 mod n then its a prime
    if b == 1 or b == n - 1:
```

```

        return True

# Testing it K time
while k >= 0:
    # b = b^2 mod n
    b = (b * b) % n
    # b = 1 mod n it is composite
    if b == 1:
        return False
    # b = -1 mod n it is probably a prime
    if b == n - 1:
        return True
    k = k-1
return False

```

2.7 Random Prime

This function was written specifically for RSA key generation. It takes an input of b that is length of the bit string. Then the function picks a random integer between $2^b - 1$ and $2^{b+1} - 1$. It runs the primality check on it. If its true then it returns that value. If not then it finds another random integer and runs the check on it.

```

def random_prime(b):
    """
    random prime of bit string of length b
    :param b: number of bits
    :return: random prime between  $2^b-1$  and  $2^{(b+1)}-1$ 
    """
    s = 2**b-1
    e = 2**(b+1) -1

    randNum = random.randint(s, e)
    while is_prime(randNum) is False:
        randNum = random.randint(s, e)

    return randNum

```

2.8 Factorization

There are multiple factorization methods out there in the world of cryptography. In my project I have implemented Pollard Rho, Fermat's Method, Pollard's P1 and Shank's Square Form Method.

2.8.1 Pollard Rho

Pollard Rho method was proposed by John Pollard in 1975. Its algorithm is as follows:

- Pick X to be a random integer between 2 to n
- let $Y = X$
- Pick C to be a random integer between 2 to n
- Pick a polynomial equation in this case $f(x) = (x^2 + c) \bmod n$
- Let d equal to 1
- $x = f(x)$, $y = f(f(y))$, $d = \gcd(x - y, n)$
- Repeat until $d \neq 1$
- As long as $d \neq n$, you have your factor

Code for Pollard's Rho

```
def pollard_rho(n):
    """
    Calculate a factor of n using pollard rho method
    :param n: number to factorized
    :return: factor of n
    """
    if n == 1: # if no prime divisors for 1
        return n

    if n % 2 == 0:
        return 2

    x = random.randint(2, n) # range of divisors from 2 to N
    y = x
    c = random.randint(1, n)

    d = 1 # divisor
    # f(x) = (x^2 + c) mod n
    while d == 1:
        # x = f(x)
        x = ((x*x) % n + c) % n
        # y = f(f(y))
        y = ((y*y) % n + c) % n
        y = ((y*y) % n + c) % n
        d = gcd(abs(x-y), n)
    if d == n: # no factor found
        return n
    return d
```

2.8.2 Fermat's Method

Fermat's Method attempts to express a given integer as difference between two integers. Let's say if $n = x^2 - y^2$ then we can express n as $(x + y)(x - y)$. It starts with $x = \text{sqrt}(n)$. Then it incrementally increases x by 1 and calculates $y = x^2 - n$. If y is a perfect square it returns the smaller factor (x-y).

Code for Fermat's method

```
def fermats_method(n):
    """
    Calculate factor of n using Fermat's Method
    :param n: number to factorized
    :return: factor of n
    """
    if is_prime(n):
        return n
    x = ceil(sqrt(n))
    y = x*x - n
    while (int(sqrt(y))**2 != y):
        x = x + 1
        y = x*x - n
    return int(x - sqrt(y))
```

2.8.3 Pollard P-1 Method

Algorithm:

- Pick b1 and b2 as lower and upper bounds
- Compute all primes smaller than b2
- Pick a as a random number between 2, n
- Compute $g = \text{gcd}(a, n)$ if g is not 1 or n then we found a factor.
- If not compute power of a prime to be closest to b1.
- Compute $a^{\text{power}} \bmod n$
- compute $g = \text{gcd}(a-1, n)$ if g is not 1 or n then we have found non trivial factor
- Compute increment $b1 = b1*2$

Code for Pollard's p-1

```
def pollard_p_1(n):
    """
    Calculate factor of n using Pollards P1 Method
    :param n: number to factorized
```

```

: return: factor of n
"""
b1 = 1 # lower bound
b2 = 1000 # upper bound
# compute all primes less than b2
primes = all_primes(b2)
while b1 <= b2:
    a = random.randint(2, n)
    g = gcd(a, n)
    if 1 < g < n:
        return g

    for p in primes:
        # ignore if p is greater than b1
        if p < b1:
            power = 1
            while p * power <= b1:
                power *= p
            a = powerModulus(a, power, n)
            g = gcd(a - 1, n)
            if 1 < g < n:
                return g

    b1 = b1 * 2
return 1

```

2.8.4 Shanks Square Form

This algorithm was out forth by Daniel Shanks and it was meant to improve on Fermat's method. According to Fermat's Method, $x^2 - y^2 = n$ that means there exists $x^2 = y^2 \bmod n$, Shank's method helps us find these pairs.

Initialize n is the integer to be factored and k is a multiplier.

$$P0 = PPrev = \text{floor}(\sqrt{kn})$$

$$Q0 = 1$$

$$Q1 = kn - P0^2$$

$$P1 = 0$$

Loop until Q1 is perfect square

$$b = \text{floor}\left(\frac{P0 + PPrev}{Q1}\right)$$

$$P1 = bQ1 - PPrev$$

$$Q0 = Q1$$

$$Q1 = Q1 + b(PPrev - P)$$

$$PPrev = P1$$

Initialize

$$b = \text{floor}(\frac{\sqrt{kn} - PPREV}{\sqrt{Q1}})$$

$$P0 = P1 = PPrev = b\sqrt{Q1} + PPrev$$

$$Q0 = \sqrt{Q1}$$

$$Q1 = \frac{kn - P0^2}{Q0}$$

Loop until P1 != PPrev

$$b = \text{floor}(\frac{\sqrt{kn} + PPrev}{Q1})$$

$$P1 = bQ1 - PPrev$$

$$Q0 = Q1$$

$$Q1 = Q1 + b(PPrev - P1)$$

$$PPrev = P1$$

Code for Shanks square form

```
k = [1, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]
def shanks_sq(n,k):
    if is_prime(n) or is_square(n):
        return n
    # initialize
    p0 = floor(sqrt(k*n))
    p_prev = p0
    q0 = 1
    q1 = k*n - p0*p0
    p1 = 0
    i = 2
    # loop till Q1 is perfect square
    while is_square(q1) is False:
        temp = q1
        b = floor((p0+p_prev)/q1)
        p1 = b*q1-p_prev
        q1 = q0+b*(p_prev - p1)
        i += 1
        q0 = temp
        p_prev = p1
```

```

        if i > sqrt(n):
            break
    # reinitialize
    b = floor((p0-p1)/sqrt(q1))
    p0 = b*sqrt(q1) + p1
    p1 = p_prev = p0
    q0 = sqrt(q1)
    q1 = (k*n - p_prev*p_prev)/q0
    i = 0
    # loop until p1 != p_prev
    while True:
        b = floor((p0 + p1)/q1)
        p_prev = p1
        temp = q1
        p1 = b*q1 - p1
        q1 = q0 + b * (p_prev - p1)
        i += 1
        q0 = temp
        if p1 == p_prev:
            break
        if i > sqrt(n):
            break
    f = gcd(n, p1)
    f = int(f)
    if f is not 1 and f is not n:
        return f
    return 0

def factorize(n):
    for i in k:
        f = shanks_sq(n,i)
        print(i)
        if f is not 0:
            return f, n/f
    return 1, n

```

2.8.5 Factorize Function

This function lets you chose what method you want to use for factorization.

```

def factorize(n, m):
    """
    Lets you choose what method you want to use for factorizing.
    1 - Fermat's Method
    2 - Pollard Rho
    3 - Pollard P-1

```

```

4 - Shank's Form
:param n:
:param m:
:return:
"""
if m == 1:
    return fermats_method(n)
if m == 2:
    return pollard_rho(n)
if m == 3:
    return pollard_p_1(n)
if m == 4:
    return ShanksSquareForm(n)

```

2.9 Testing CryptoMath Library

To enable you to test all the functions in the cryptomath library I have a file called *basictest.py*. On running it, you will see a menu and you can choose whatever function you want to run and get answers for. This program will ask for input values from the user.

3 Classical Cryptosystems

3.1 Affine Cipher

Affine Cipher is substitution cipher. Each alphabet in the cipher can be mapped to a specific number or alphabet. The same mathematical formula lets you map the encrypted value back to the alphabet. The key for this encryption is two values (a, b) such that $\gcd(a, 26) = 1$ and $0 \leq b < 26$.

$$\text{encrypt}(x) = ax + b \% 26$$

$$\text{decrypt}(x) = a^{-1}(x - b) \% 26$$

$$\text{decrypt}(\text{encrypt}(x)) = a^{-1}(\text{encrypt}(x) - b) \% 26$$

$$\text{decrypt}(\text{encrypt}(x)) = a^{-1}(ax + b - b) \% 26$$

$$\text{decrypt}(\text{encrypt}(x)) = a^{-1}ax \% 26 = x \% 26$$

3.1.1 Example

Example for $a = 5$ and $b = 3$

```

7404855@T-SMD1027771 MIN
$ python affine.py
Enter value of a: 5
Enter value of b: 3
Enter plain text: HELLO
a inverse: 21
MXGGV
HELLO

```

Figure 1: Affine Program Output

Plaintext	H	E	L	L	O
$5x+3$	38	23	58	58	73
$c = 5x+3 \% 26$	12	23	6	6	21
cipher	M	X	G	G	V
$5^{-1}(c-3)$	189	420	63	63	378
$21(c-3)\%26$	7	4	11	11	14
Decrypted Messgae	H	E	L	L	O

3.1.2 Code

```

def encrypt(a, b, pt, n):
    cipher = ""
    for c in pt:
        ci = (a*(ord(c) - 65)+b) % n
        cipher += str(chr(ci + 65))
    return cipher

def decrypt(a, b, ct, n):
    a_inv = findModInverse(a, n)
    decipherd = ""
    for c in ct:
        pi = (a_inv*((ord(c)- 65) - b)) % n
        decipherd += str(chr(pi + 65))
    return decipherd

```

3.1.3 Weaknesses

Affine encryption is nearly a linear shift and all the possible pair of key are $12 \times 26 = 312$ keys. This number is small enough to be brute forced. Frequency analysis will also help us break this cipher.

3.1.4 Attack

I was able to implement Brute Force method Affine cipher. We just have 312 possible keys. So we try all of them and check what the message.

```
def brute_force(cipher, n):
    possible_a = []
    possible_b = range(1, n)
    for i in range(1, n):
        if gcd(i, n) == 1:
            possible_a.append(i)
    for i in possible_a:
        for j in possible_b:
            print(decrypt(i, j, cipher, n))
```

Another way to attack is if you have large amount of data, you can run frequency analysis. 'e' and 't' are two most common letters in the english text. We identify which cipher characters have the highest and 2nd highest frequency. Then you map them to 'e' and 't' respectively. This will give you two equations to work with.

```
def attack_affine(e, t, n):
    e_cipher = ord(e) - 65
    t_cipher = ord(t) - 65
    # a(4) + b = e_cipher
    # a(19) + b = t_cipher
    # a = (e_cipher - t_cipher) / (-15)
    a = (e_cipher - t_cipher) / (-15.0)
    a = int(a) % n
    while a < 0:
        a += n
    b = (e_cipher - 4*a) % n
    return a, b
```

3.2 Vignere Cipher

Vignere Cipher is another type of substitution cipher but unlike Affine it poly-alphabetic.

$$\text{encrypt}(xi) = xi + Ki \% 26$$

$$\text{decrypt}(xi) = xi - ki \% 26$$

3.2.1 Example

To understand this algorithm lets look at an example. Lets pick a key "APLE"

Plaintext	C	R	Y	P	T	O	I	S	F	U	N
Key	A	P	L	E	A	P	L	E	A	P	L
Cipher	C	G	J	T	T	D	T	W	F	J	Y

3.2.2 Code

Here the encryption and decryption code:

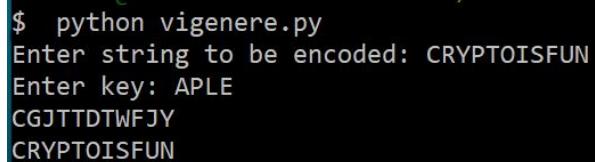
```
def vigenere_encode(str , key):
    """
    To encode using vigenere algorithm add each character from
    the plain text to the corresponding character from the key.
    :param str: string to be encoded
    :param key: key to be used for encoding
    :return: encrypted cipher
    """
    cipher = ""
    for i in range(len(str)):
        val = (ord(str[i]) + ord(key[i% len(key)]) - 130) % 26
        cipher += chr(val + 65)

    return cipher

def vigenere_decode(cistr , key):
    """
    To decode subtract every character from the cipher with
    the corresponding character from the key
    :param cistr: cipher string
    :param key: key to be used to decode
    :return: original message
    """
    orig = ""

    for i in range(len(cistr)):
        val = (ord(cistr[i]) - ord(key[i% len(key)]) + 26) % 26
        orig += chr(val + 65)

    return orig
```

A terminal window with a black background and white text. The first line shows a shell prompt '\$' followed by the command 'python vigenere.py'. The next two lines are prompts 'Enter string to be encoded:' and 'Enter key:', followed by the user input 'CRYPTOISFUN' and 'APLE' respectively. The final two lines show the program's output: 'CGJTDTWFJY' and 'CRYPTOISFUN'.

```
$ python vigenere.py
Enter string to be encoded: CRYPTOISFUN
Enter key: APLE
CGJTDTWFJY
CRYPTOISFUN
```

Figure 2: Vigenere Program output

3.2.3 Weakness

Guessing Key length is very easy by looking at the repeated patterns in the cipher. If you know some characters from the plain text the code is not hard to break.

3.2.4 Attack

Guessing the Key Length To be able to guess the key length we start out by shifting the cipher by let's k which is one of our possible lengths. We count the number of times the character's match. We record the counts for each possible length. We see a certain periodicity/frequency in the peaks that give us the key length.

```
def key_length(cipher):
    possible_lengths = range(1, 100)
    counts = []

    for length in possible_lengths:
        count = 0
        for i in range(length, len(cipher)):
            if cipher[i] == cipher[i-length]:
                count +=1
        counts.append(count)
    plt.plot(possible_lengths, counts)
    print(counts)
    plt.show()
```

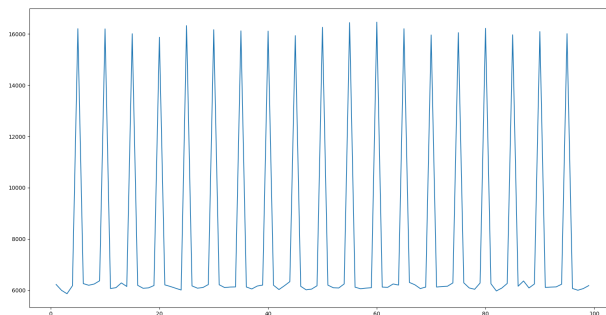


Figure 3: Possible Key length Coincidence for A Christmas Carol text file with key "LEMON"

We can conclude that the Key length is 5 which matches with our actual key "LEMON". Now we perform frequency analysis. We input that length

and text along with a list of frequencies of English alphabet(a) in to the find key function. Then we look at all character offset by the length of the key and analyze its frequency to build W matrix. Say if the length of the key is 5 then we look at all 1st, 6th, 11th ...and so and build our W for the first possible key character.

Then we find the dot product of $W \cdot a_i$ and check which character has the highest frequency. Currently due to some undetected bug the key prediction fails for 'A'. I will continue to work on this project and fix it later.

```
def find_key(text, length, a):
    W = [] # all W matrices

    for j in range(length):
        temp = ""
        for i in range(len(text)):
            if i % length == 0 and i + j < len(text):
                temp += text[i + j] # append all i mod n characters
        w = analyze(temp.upper()) # analyze their frequency
        W.append(w) # Append it to W matrix

    possibleKey = []
    for j in range(length):
        value = 0 #initialize
        k = 0
        for i in range(1, 26): # try all possible shifts
            rotate_i = np.array(a[i:]).tolist() + np.array(a[:i]).tolist()
            dot = np.dot(np.array(W[j][:]), np.array(rotate_i)) # dot product
            if dot > value: # find the max
                value = dot
                k = 26 - i
        if k < 8:
            possibleKey.append(alphabets[k]) # append the key chars
        else:
            possibleKey.append(alphabets[k-1])
    print(possibleKey)
```

3.3 ADFGX Cipher

ADFGX cipher was used by German Army during World War I. It is a fractionating transposition cipher. Cipher is named ADFGX because they are very different letters in the Morse Code and would reduce errors while communicating. Polybuis Square for my program looks like this

```

$ python vigenere.py
Enter filename from your folder: christmas.txt
Enter key: leson
['L', 'E', 'S', 'O', 'N']

7404855@T-SMD1027771 MINGW64 ~/Documents/cryptogr
$ python vigenere.py
Enter filename from your folder: christmas.txt
Enter key: trete
['T', 'R', 'E', 'T', 'E']

```

Figure 4: Key Predictions for Vigenere Cipher

	A	D	F	G	X
A	p	g	c	e	n
D	b	q	o	z	r
F	s	l	a	f	t
G	m	d	v	i	w
X	k	u	y	x	h

3.3.1 Example

Lets walk through an example.

Plaintext	C	R	Y	P	T	O	I	S	F	U	N
Substitution	AF	DX	XF	AA	FX	DF	GG	FA	FG	XD	AX

L	E	M	O	N
A	F	D	X	X
F	A	A	F	X
D	F	G	G	F
A	F	G	X	D
A	X			

Sorting

E	L	M	N	O
F	A	D	X	X
A	F	A	X	F
F	D	G	F	G
F	A	G	D	X
X	A			

Cipher text : FAFFX AFDAAG DAGG XXFD XFGX

```

$ python adfgx.py
c : A F
r : D X
y : X F
p : A A
t : F X
o : D F
i : G G
s : F A
f : F G
u : X D
n : A X
cipher = FAFFXAFDAADAGGXXFDXFGX
decrypted = cryptoisfun

```

Figure 5: ADFGX program output

3.3.2 Code

Code for encryption:

```

def encrypt(pt, key):
    # string to store substitution cipher
    subs_cipher = ""
    # convert it lower case
    pt = pt.lower()

    for i in range(len(pt)):
        # calculate row by dividing by 5
        r = floor(alphaMatrix.index(pt[i])/5)
        # calculating col by modulus 5
        c = alphaMatrix.index(pt[i]) % 5
        subs_cipher += adfgx[r] + adfgx[c]

    # if not even then add X
    if len(subs_cipher) % 2 != 0:
        subs_cipher += "X"

    keyLen = len(key)
    # sort just the indexes
    sortedKey = sorted(range(keyLen), key = lambda i: key[i])
    s = len(subs_cipher)
    # string to store cipher text
    cipher = ""

```

```

    for i in sortedKey:
        for j in range(i, s, keyLen):
            cipher += subs_cipher[j]

    return cipher

```

3.3.3 Weakness

ADFGX was cracked by Lieutenant Painvin and he used frequency analysis multiple times to unscramble and decipher the message. Today this cipher is considered to be insecure.

3.4 Frequency Calculation

To analyze frequency this function takes a 26 long array and keeps a count for each character it notices in the text. There is also a helper function called print analysis.

```

def analyze(text):
    freq = np.zeros(26)
    for c in text:
        index = ord(c) - 65
        if 0 <= index < 26:
            freq[index] += 1
    sum = freq.sum()
    for i in range(26):
        freq[i] /= sum
    return freq

def print_analysis(freq):
    for i in range(26):
        print(chr(i+65), " : ", freq[i])
    sortedFreq = np.argsort(freq)
    print("Max #1 : ", chr(sortedFreq[-1] + 65), " ", freq[sortedFreq[-1]])
    print("Max #2 : ", chr(sortedFreq[-2] + 65), " ", freq[sortedFreq[-2]])
    return chr(sortedFreq[-1] + 65), chr(sortedFreq[-2] + 65)
# return freq[sortedFreq[-1]], freq[sortedFreq[-2]], freq[sortedFreq[-3]]

```

4 DES

DES or Data Encryption Standards is symmetric key algorithm. It is also block cipher. It was developed by IBM in 1975. It uses 56 bit keys and 64 bit block sizes. Encryption and Decryption in DES are both similar and the only difference is that the keys are fed in reverse order for decryption.

```

$ python frequency.py
Enter filename from your folder: christmas.txt
A : 0.07567480719794345
B : 0.016220203318532366
C : 0.026751285347043702
D : 0.04542533302173405
E : 0.12227594064033653
F : 0.020704311755082964
G : 0.024786749240476746
H : 0.06530439354989484
I : 0.06910931292358027
J : 0.0014825309651787801
K : 0.008544636597335826
L : 0.03690990885720963
M : 0.023494099088572095
N : 0.06581561112409441
O : 0.08078698293993924
P : 0.018798200514138816
Q : 0.0007887356859079224
R : 0.06011918672587053
S : 0.0637707408272961
T : 0.09104784996494508
U : 0.028138875905585416
V : 0.008413180649684505
W : 0.02461147464360832
X : 0.0011611942042533301
Y : 0.019221780789904185
Z : 0.0006426735218508997
Max #1 : E 0.12227594064033653
Max #2 : T 0.09104784996494508

```

Figure 6: Frequency Analysis of program on a text file

4.1 Simplified DES Algorithm

Simplified DES algorithm was generated for educational purpose. It uses 10 bits long key and 8 bit long input.

4.1.1 Permutation Tables and S Boxes

```

# Permutation Tables
# 1) IP

```

```

IP = [3, 5, 2, 7, 4, 10, 1, 9, 8, 6]
# 2) Permute 8 bits by removing the 1st 2 bits from 10 bits
Perm10_8 =[6, 3, 7, 4, 8, 5, 10, 9]
# 3) Permute 8 bits
P8 = [2, 6, 3, 1, 4, 8, 5, 7]
# 4) IP inverse
IP_inv = [4, 1, 3, 5, 7, 2, 8, 6]
# 5) Expansion Table
EP = [4, 1, 2, 3, 2, 3, 4, 1]
# 6) Permute 4 bits
P4 =[2, 4, 3, 1]
# Sbox 1
s1 =[[1, 0, 3, 2], [3, 2, 1, 0], [0, 2, 1, 3],
      [0, 2, 1, 3], [3, 1, 3, 2]]
# SBox 2
s2 =[[0, 1, 2, 3], [2, 0, 1, 3], [3, 0, 1, 0],
      [2, 1, 0, 3]]

```

4.1.2 S-DES Diagram

This diagram attempts at explaining how my S-DES is implemented.

4.1.3 Key Generation Code

```

def generateKeys(key):
    """
    Given an integer key generate 4 sub keys.
    Stores the sub keys in the KEY table
    :param key: integer key
    :return: nothing.
    """
    key_string = get_bin(key, 10)
    p = permute(key_string, IP)
    key_left = p[0:5]
    key_right = p[-5:]
    rKey_left, rKey_right = round_shift(key_left, key_right)
    keyOne = permute(rKey_left+rKey_right, Perm10_8)

    rKey_left, rKey_right = round_shift(rKey_left, rKey_right)
    keyTwo = permute(rKey_left+rKey_right, Perm10_8)

    rKey_left, rKey_right = round_shift(rKey_left, rKey_right)
    rKey_left, rKey_right = round_shift(rKey_left, rKey_right)
    keyThree = permute(rKey_left+rKey_right, Perm10_8)

```

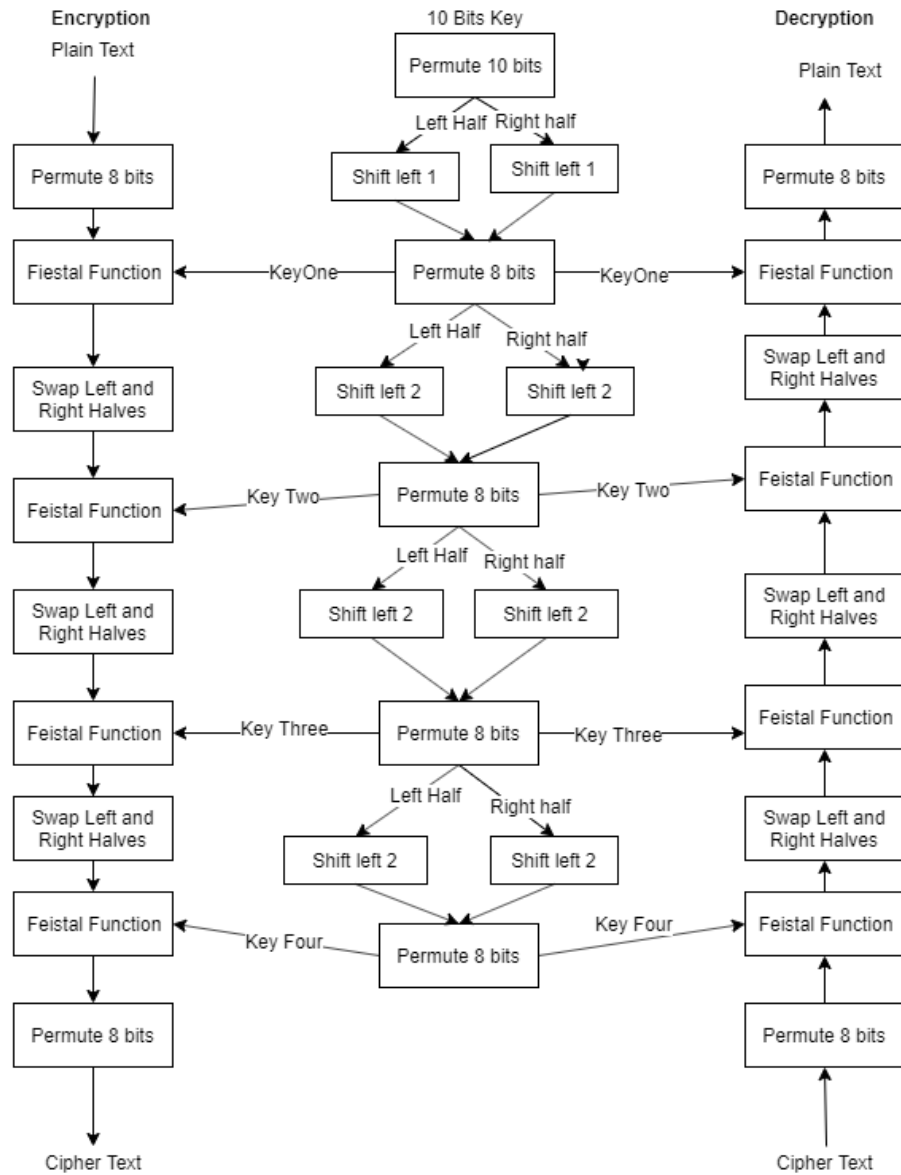



Figure 7: Diagram for S-DES Key generation, Encryption and Decryption

```

rKey_left , rKey_right = round_shift(rKey_left , rKey_right)
rKey_left , rKey_right = round_shift(rKey_left , rKey_right)
keyFour = permute(rKey_left+rKey_right , Perm10_8)

```

```
KEYS = [keyOne, keyTwo, keyThree, keyFour]
return KEYS
```

4.1.4 Encryption Code

The function takes in an integer and converts it to 8 bit string. Performs initial permute and then splits it into two halves. These are put through 4 rounds. The output from these rounds is put through inverse permutation. The output from the permutation is the cipher.

Decryption is very similar to decryption except the order of keys is reversed.

```
def encrypt_des(plain_text):
    """
    Encrypt the given integer
    :param plain_text: integer to be encrypted
    :return: cipher
    """
    pt_bits = get_bin(plain_text, 8)
    perm_bits = permute(pt_bits, P8)
    left = perm_bits[:4]
    right = perm_bits[-4:]

    # Round 1
    st1L, st1R = feistel(left, right, KEYS[0])
    # Round 2
    st2L, st2R = feistel(st1L, st1R, KEYS[1])
    # Round 3
    st3L, st3R = feistel(st2L, st2R, KEYS[2])
    # Round 4
    st4L, st4R = feistel(st3L, st3R, KEYS[3])

    cipher = permute(st4R+st4L, IP_inv)
    return int(cipher, 2)
```

4.1.5 S Box look up:

Looks up given S Box. The bit string is of length 4. First and last bit give the row number. 2nd and 3rd bits gives the column number.

```
def look_up_stable(bits, table):
    """
    Looks up given S Box.
    :param bits: 4 bit string
    :param table: Sbox that needs to be looked up
    :return: 2 bit binary string
    """
```

```

r = int(bits[0]+bits[3], 2)
c = int(bits[1]+bits[2], 2)
return get_bin(table[r][c], 2)

```

4.1.6 Expansion function

This function XORs right half and the sub key. First half of the output feeds into SBox 1 and 2nd half feeds into the SBOX 2. The output combined and permuted before returning.

```

# Expansion Function and Permute
def expand_perm(r, k):
    """
    Xors right half and the key. First of the output feeds
    into SBox 1 and 2nd half feeds into the SBOX 2. The output
    combined and permuted before returning.
    :param r: right half
    :param k: subkey
    :return: bit string
    """
    expand_r = permute(r, EP)
    expand_r_xor_k = xor(expand_r, k)
    e_l = expand_r_xor_k[:4]
    e_r = expand_r_xor_k[-4:]

    sleft = look_up_stable(e_l, s1)
    sright = look_up_stable(e_r, s2)
    perm4 = permute(sleft + sright, P4)
    return perm4

```

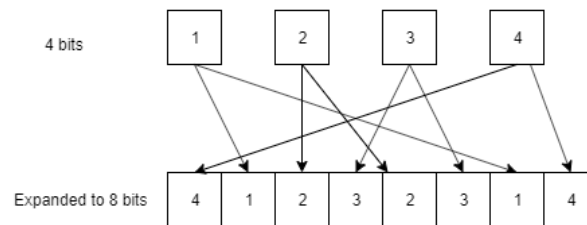


Figure 8: Key Expansion for S-DES

4.1.7 Feistel's function

Left half is XORed with the output from the Expansion function. This function that swaps the right and left while returning.

```

def feistel(left , right , k):
    """
    The left half and output from the f function is
    XORed and swapped before returning
    :param left: left half
    :param right: right half
    :param k: subkey
    :return: right half, left key
    """
    l_xor_perm = xor(left , expand_perm(right , k))
    return right , l_xor_perm

```

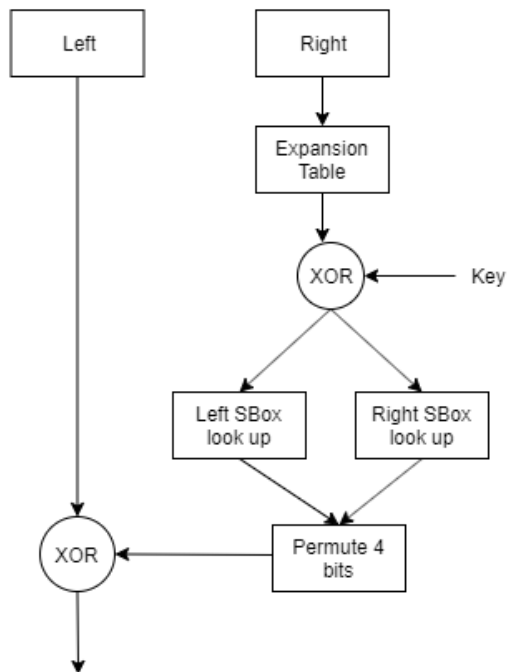


Figure 9: Feistel's function for S-DES

4.2 64-DES

For 64 Bit DES I have used all the S boxes and permutation tables from the textbook. I also followed step by step instructions from the textbook. To understand what my code is doing we need to look at the diagram from the textbook.

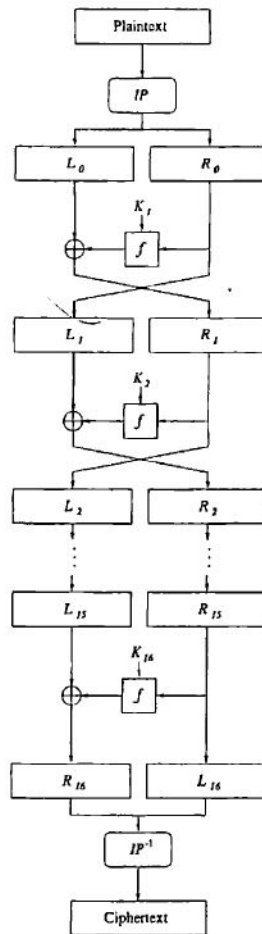


Figure 4.4: The DES Algorithm.

Figure 10: Encryption of 64 DES from the textbook

4.2.1 Key Generation Code

We generate 16 sub keys from a given key. For each key the bit string is split in half. Each half is rotated a specific shifts. Then those two halves are combined and reduced to 48 bits. I am storing each rotated half in to the list C(left) and D(right).

```

shifts = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
def generate_keys(key):
    """

```

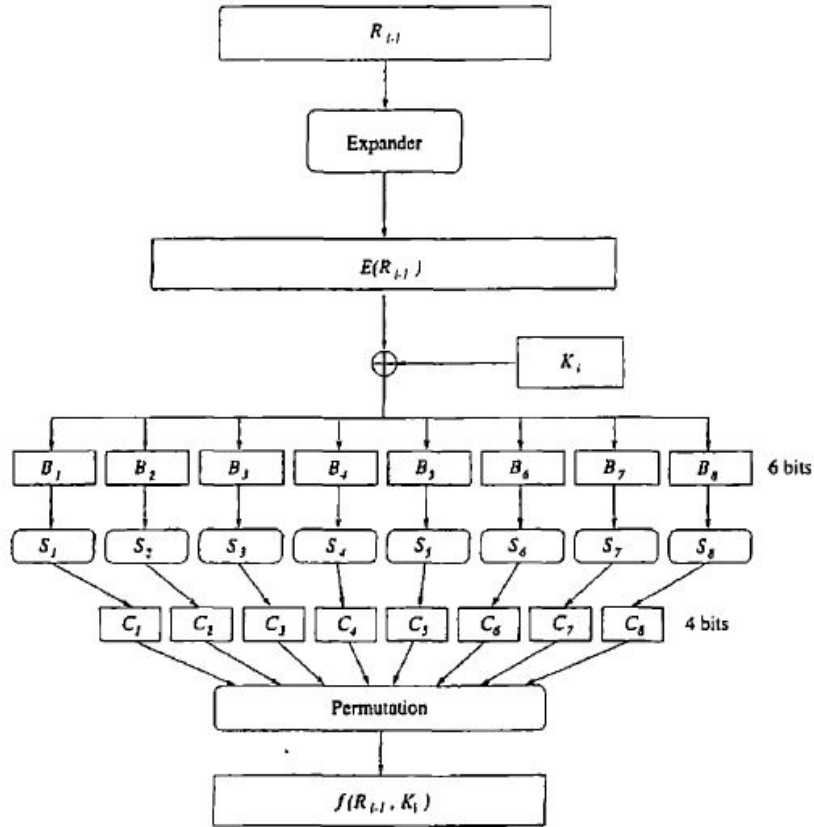


Figure 4.5: The DES Function $f(R_{i-1}, K_i)$.

Figure 11: DES 64 f function from the textbook

```

Given an integer key generate 16 sub keys.
Stores the sub keys in the KEY table
:param key: integer key
:return: nothing.
"""
key_string = get_bin(key, 64)
#print(key_string)
key_56 = permute(key_string, Key56Table)
#print("Key 56: ", key_56)
KEYS.append(key_56)
C.append(key_56[:28])

```

```

D.append(key_56[-28:])
for i in range(1, 17):
    C0 = C[i - 1]
    D0 = D[i - 1]
    #print(i, shifts[i-1])
    if shifts[i - 1] == 1:
        Ci, Di = round_shift(C0, D0)
    elif shifts[i - 1] == 2:
        Ci, Di = round_shift(C0, D0)
        Ci, Di = round_shift(Ci, Di)

    #print("C: ", Ci)
    #print("D: ", Di)
    C.append(Ci)
    D.append(Di)
    Ki = permute(Ci + Di, Key48Table)
    KEYS.append(Ki)

```

4.2.2 f Function

f function takes the right half and the subkey. The right half is expanded to 48 bits and XORed with the sub key. The output is then sliced in to 6 bit strings where each string is an input for S box look up. All the outputs from the S box are combined to form 32 bit string. It is permuted before returned.

```

def f(right, keyi):
    """
    f function which takes the right half and subkey.
    The right half is expanded to 48 bits and XORed with
    the sub key. The output is then sliced in to 6 bit strings
    where each string is an input for S box look up. All the outputs
    from the S box are combined to form 32 bit string. It is permuted
    before returned
    :param right: right half
    :param keyi: subkey
    :return: bit string
    """
    expanded_right = permute(right, EPTable)
    expRXORKey = xor(expanded_right, keyi)
    C = ""
    for i in range(0, 8):
        bits6 = expRXORKey[i * 6:(i + 1) * 6]
        c = SBox_lookup(bits6, S[i])
        C = C + c
    perm_C = permute(C, Permuted32)
    return perm_C

```

4.2.3 S Box Look Up

This function is to look up into a S box. The bit string is of length 6. First and last bit give the row number. 2nd, 3rd, 4th and 5th bits give the column number. These values are used to index into the Sbox and return the 4 bit result..

```
def SBox_lookup(bits , table):
    """
    Looks up given S Box.
    :param bits: 6 bit string
    :param table: Sbox that needs to be looked up
    :return: 4 bit binary string
    """
    row = int(bits[0] + bits[5], 2)
    col = int(bits[1] + bits[2] + bits[3] + bits[4], 2)
    index = row * 16 + col
    return get_bin(table[index], 4)
```

4.2.4 Encryption

For encryption first the input is converted to 64 bits long binary string. Initial permutation is performed before splitting the string in two 32 bits string. The 16 rounds are performed and its output is put through inverse IP before returning the cipher.

Decryption occurs in exactly same manner but the only difference is that keys are fed in reverse order. That is the loop goes from 16 to 1 instead of 1 to 16.

```
def encrypt(pt):
    """
    Encrypts given input
    :param pt: Integer input to be encrypted string.
    :return: encrypted cipher
    """
    plaintext = get_bin(pt, 64)
    # initial permutation
    pt_init_perm = permute(plaintext, IP)
    # Left and right halves
    left = pt_init_perm[:32]
    right = pt_init_perm[-32:]
    # 16 rounds
    for i in range(1, 17):
        left, right = F(left, right, KEYS[i])
    # Final Permutation
    cipher = permute(right + left, IPinv)
    return hex(int(cipher, 2))[2:]
```


4.2.5 DES Weakness

So far there have been many researches on attempts to find weakness in DES. Brute-Force Attack is simply trying all possible keys and the length of the key determines if it is computationally possible. Differential and Linear Cryptanalysis are able to break all 16 rounds of DES.

Over the years new algorithms have replaced DES which are more secure AES. Triple DES was later developed which used DES 3 times with different keys. It is sufficiently secure but computationally heavy.

4.3 Differential Cryptanalysis for Three Rounds

Differential Crypt-analysis helps us break DES. The algorithm is as follows:

- Pick $L1R1$ and $L1 \oplus R1^*$ such that $R1 = R1^*$
- Compute $L4R4$ and $L4 \oplus R4^*$ using same key
- Compute $L4' = L4 \oplus l4^*$
- Compute $E(L4')$, expansion function -> Input for SBox
- Compute $R4' \oplus L1'$ -> Output of SBox
- find pairs of such that their Sbox input and output match what was computed.
- Run these steps on multiple inputs and eliminate the possible pairs until only one is remaining.
- The pairs from SBox 1 will give us Left 4 bits
- The pairs from SBox 2 will give us Right 4 bits

Code for Differential Cryptanalysis

```
def get_round_14(bits , key):
    left = bits[:4]
    right = bits[-4:]
    KEYS = generateKeys(key)

    # Round 1
    #st1L , st1R = feistal(left , right , keyOne)
    # Round 2
    st2L , st2R = feistal(left , right , KEYS[1])
    # Round 3
    st3L , st3R = feistal(st2L , st2R , KEYS[2])
    # Round 4
    st4L , st4R = feistal(st3L , st3R , KEYS[3])
```

```

    return left , right , st4L , st4R

def differential_cryptanalysis(str , str_star , key):
    l1 , r1 , l4 , r4 = get_round_14(str , key)
    l1_s , r1_s , l4_s , r4_s = get_round_14(str_star , key)
    print(r4 , r4_s)
    l1_diff = xor(l1 , l1_s)
    r1_diff = xor(r1 , r1_s)
    l4_diff = xor(l4 , l4_s)
    r4_diff = xor(r4 , r4_s)
    # r4' xor l1'
    r4l1d = xor(r4_diff , l1_diff)
    # E(l4')
    el4d = permute(l4_diff , EP)
    # Sbox 1 input
    s1_input = el4d[:4]
    # Sbox 1 output
    s1_output = r4l1d[:3]
    # Sbox 2 input
    s2_input = el4d[-4:]
    # Sbox 2 output
    s2_output = r4l1d[-3:]
    # contain possible K4L
    pairs_s1 = []
    # find pairs s1
    for i in range(0 , 16):
        for j in range(0 , 16):
            x = get_bin(i , 4)
            y = get_bin(j , 4)
            if xor(x , y) == s1_input:
                s1_x = look_up_stable(x , s1)
                s1_y = look_up_stable(y , s1)
                if xor(s1_x , s1_y) == s1_output:
                    pairs_s1.append(xor(x , el4_left))

    # contain possible K4R
    pairs_s2 = []
    # find pairs s2
    for i in range(0 , 16):
        for j in range(0 , 16):
            x = get_bin(i , 4)
            y = get_bin(j , 4)
            if xor(x , y) == s2_input:
                s2_x = look_up_stable(x , s2)
                s2_y = look_up_stable(y , s2)
                if xor(s2_x , s2_y) == s2_output:

```

```

pairs_s2.append(xor(y, el4_right))

# needs to be completed...

```

5 RSA

RSA or Rivest- Shamir - Adleman was one of the first asymmetric encryption. It has two keys one public that is available for any one to find and be able to use for sending information to its owner. Private key on the other hand is closely guarded secret on the host. In asymmetric key encryption when one key is used encrypt you require the other key decrypt and vice-versa. This algorithm stems from simple modular mathematics.

Key Generation Algorithm:

- Generate two large prime numbers p and q
- Compute n such that $n = pq$
- Find e such that $\gcd(e, \phi(n)) = 1$
- your public key will be (e, n)
- Compute $d = \text{inverse of } e \text{ mod } n$ your private key will be (d, n)

RSA Communication Algorithm If you want to send a message m to an user with public key (e, n) and private key (d, n)

- compute $m^e \text{ mod } n$ and send it to the user
- Receiver will decrypt it using private key $(m^e)^d \text{ mod } n = m$

5.1 RSA Key Generation

```

def generate_keypair():
    p = random_prime(10)
    q = random_prime(10)
    # make sure p and q are not same
    while p == q:
        q = random_prime(10)
    # n = pq
    n = p*q
    # phi(n) = p-1*q-1
    phi = (p-1)*(q-1)

    # choosing e
    e = random.randint(1, phi)
    while gcd(e, phi) != 1:

```

```

e = random.randint(1, phi)

# finding modular inverse of for private key
d = findModInverse(e, phi)
# return e, n , d
return e, n, d

```

5.2 RSA Encryption and Decryption

$$E(x) = x^e \% n$$

$$D(x) = x^d \% n$$

```

def encrypt_rsa(publicK, plaintext):
    e, n = publicK
    # make it uppercase
    plaintext = plaintext.upper()
    cipher = []
    for ch in plaintext:
        c = powerModulus(ord(ch), e, n)
        cipher.append(c)
    return cipher

def decrypt_rsa(privateK, cipher):
    d, n = privateK
    plain = ""
    for ch in cipher:
        c = powerModulus(ch, d, n)
        plain = plain + str(chr(c))

    return plain

```

6 DES and RSA Hybrid

For my AOA course project, I had coded DES and RSA Hybrid to verify user, send DES key securely and then encrypt message. DES is symmetric, requires DES key to be communicated ahead of time but is faster. While RSA is asymmetric, requires no key communication but its slower. So I used RSA to verify the user and communicate the DES. While I used DES to encrypt the large amount of data that needed to be communicated. Apparently, this same principle is used in SSL Certification (Secure Socket Layer). Asymmetric encryption is used to verify the client to the server. Then it communicates the Session Key to the server. Symmetric encryption is used to encrypt information with session key as the key. The Server is able to decrypt using the session key it acquired

earlier. You can test the code in the file hybrid.py.

Secure Communication Algorithm:

- Sender sends message encrypted using its RSA secret key to the user
- Receiver decrypts it using Sender's RSA public key and verifies the sender
- Sender then sends DES key encrypted to the Receiver using Receiver's RSA public key
- Receiver decrypts and acquires the key using its own RSA private key
- Sender then sends the DES encrypted message and receiver decrypts using the DES key it acquired in previous step.

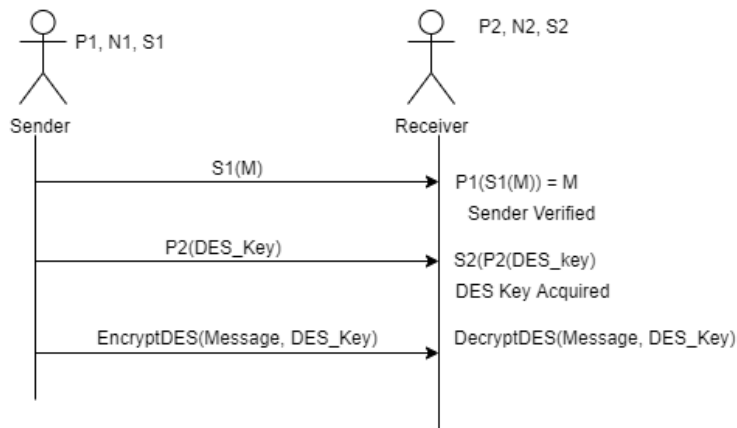


Figure 12: Communication between two users

7 Sources

- “Introduction to Cryptography with Coding Theory” by Wade Trappe and Laurence C. Washington
- Shanks Square Form: <https://resources.saylor.org/wwwresources/archived/site/wp-content/uploads/2012/07/Shanks-square-forms-factorization.pdf>
- S-DES: <https://www.c-sharpcorner.com/article/s-des-or-simplified-data-encryption-standard/>