**Cryptography**

### -Lab report



Submitted By: Deshant Devkota

Submitted To: Suresh Thapal (Lecturer, Cryptography)

Vedas College, Lalitpur, Nepal

**Tribhuvan University**

**Department of Bsc Csit**

Submitted Date: Thursday, June 24, 2021

Submitted Lab Reports:

1. To implement Diffie Hellman Key Exchange and check if generated keys for two parties match.
2. To implement RSA encryption and use it to encrypt and decrypt a message.

## LAB 10

Diffie Hellman Key Exchange

**1. Objectives:**

In this lab we were to implement Diffie Hellman Key Exchange and check if generated keys for two parties match.

**2.Introduction:**

Diffie–Hellman key exchange is a method of securely exchanging cryptographic keys over a public channel and was one of the first public-key protocols as conceived by Ralph Merkle and named after Whitfield Diffie and Martin Hellman. DH is one of the earliest practical examples of public key exchange implemented within the field of cryptography. Published in 1976 by Diffie and Hellman, this is the earliest publicly known work that proposed the idea of a private key and a corresponding public key. -Wikipedia

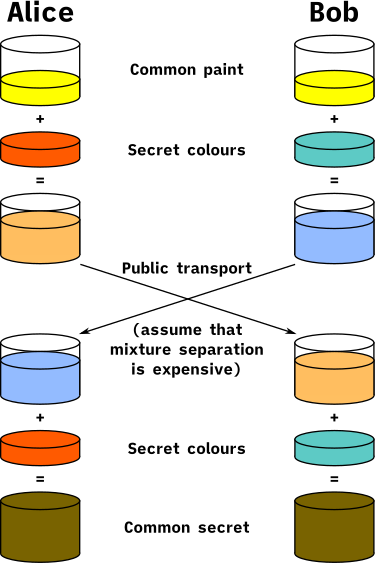
src: Wikipedia

Fig: Illustration of the concept behind Diffie–Hellman key exchange

DH Key Exchange:

1. ‘p’ and ‘g’ are public elements where ‘p’ is a prime number and ‘0<=g<=p-2’ is a primitive root of ‘p’.
2. A user selects a random key ‘x’ where 0<x<p as his private key.
3. He will then calculate his public key ‘y’ with help of his private key as: y = (g^x) mod p
4. Public key is shared between two parties and a secret key ‘k’ is formed which is secret between two of them. This is used to encrypt and decrypt messages using symmetric encryption. k = (y2)^x mod p, where y2 is the public key of another party.

**3.Code:**

//Implementing DIffie-Hellman Key Exchange

#include<iostream>

#include<math.h>

using namespace std;

//global variables

int p = 7, g = 3;

class user

{

private:

int x;

public:

int y, k;

void setxy(){

cout<<"\nEnter your private key value: ";

cin>>x;

int extra = pow(g ,x);

y = extra % p;

cout<<"Your Public Key is: "<<y<<endl;

}

void calck(int y2){

int extra = pow(y2, x);

k = extra % p;

}

void checkkey(int k2){

if (k == k2)

cout<<"Success"<<endl;

else

cout<<"Failure"<<endl;

}

};

int main()

{

cout<<"The global variables p and g are: "<<p<<" and "<<g<<endl;;

user a, b;

cout<<"\nFor user A:";

a.setxy();

cout<<"\nFor user B:";

b.setxy();

cout<<"\nCalculating Keys for both users: \n";

a.calck(b.y);

b.calck(a.y);

cout<<"\nChecking if generated keys for both parties match: ";

a.checkkey(b.k);

}

**4.Output:**



## LAB 11

RSA

.

**1. Objectives:**

In this lab we were to implement RSA encryption and use it to encrypt and decrypt a message.

**2.Introduction:**

RSA is an asymmetric key encryption scheme which is based on Euler's theorem for encryption and decryption. It’s strength is derived from hardness of factorization for a large number which is a multiple of two primes, making it’s factorization extremely hard.

Let ‘p’ and ‘q’ are two large prime numbers.

Then we calculate: ‘n’ = ‘p’ \* ‘q’

Since ‘n’ is a multiple of two prime numbers it is hard to factorize.

From Euler’s theorem we know

m^( ϕ(n) ) ≅ 1 (mod n) or m^( ϕ(n) + 1 ) ≅ m ( mod n )

Now since we know ‘p’ and ‘q’ we can calculate its Euler’s totient as:

ϕ(n) = p - 1 \* q - 1,

Given n it is hard to find ϕ(n) for a significantly large number and also hard to factorize it.

Let ‘e’ is relatively prime to the Totient and we calculate ‘d’ such that:

e \* d ≅ 1 ( mod ϕ(n) ), which can be solved as d = ( k \* ϕ(n) + 1 ) / e

Then our public key = (n, e) and a message, ‘ m ‘ is encrypted using this key to produce cipherkey ‘c’ as:

c = m^e mod n

Trying to decrypt the message using c, e and n is a hard problem as it requires us to solve:

c = x^e mod n, solving for x is hard.

But since ‘e’ \* ‘d’ = k \* ϕ(n) + 1, we can easily decrypt message using private key = (n, d) as

m = c^d mod n, {because, m = m^ed mod n, since ed ≅ 1}

This is how we do encryption and decryption in RSA.

**3. Code:**

def extendedEuclid(m, b):

a1, a2, a3 = 1, 0, m

b1, b2, b3 = 0, 1, b

while True:

q = int(a3/b3)

t1 = a1-q\*b1

t2 = a2-q\*b2

t3 = a3-q\*b3

a1 = b1

a2 = b2

a3 = b3

b1 = t1

b2 = t2

b3 = t3

if b3 == 0 or b3 == 1:

break

if b3 == 0:

print("\nInverse Doesn't Exist")

exit(1)

else:

return b1

def gcd(a, b):

if b == 0:

return a

else:

while b != 0:

a, b = b, a % b

return a

def eulartotient(n):

totientnumber=0

for i in range(1,n):

if gcd(n, i) == 1:

totientnumber += 1

return totientnumber

# let two prime numbers p and q

p, q = 7, 13

# calculate n

# 91

n = p \* q

# calculate Euler's totient of n

t = eulartotient(n)

# let e be a relatively prime number to t

# calculate d such that e \* d = 1 (mod t)

e = 5

d = extendedEuclid(e, t)

# Encryption

m = input("\nEnter a message to be encrypted: ")

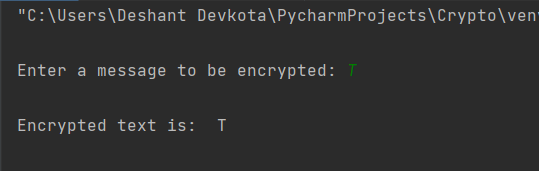
c = pow(ord(m), e) % n

# Decryption

m2 = pow(c, d) % n

print("\nEncrypted text is: ", chr(m2))

**4. Output:**



END