20CYS205 – MODERN CRYPTOGRAPHY

RSA & ELGAMAL ENCRYPTION SCHEMES

Submitted by

KAUSHIK M – CB.EN.U4CYS22035
LOGESH R – CB.EN.U4CYS22036
LALITHA K – CB.EN.U4CYS22037
SAI TEJAS MAREDDY – CB.EN.U4CYS22038

Under the guidance of

Mr. Aravind Vishnu,
Assistant Professor,
Amrita Vishwa
Vidyapeetham
Coimbatore.

TIFAC-CORE IN CYBER SECURITY
AMRITA SCHOOL OF
ENGINEERIN

AMRITA VISHWA VIDYAPEETHAM

COIMBATORE - 641 112

Acknowledgement

First of all, we would like to express our gratitude to our Mentor, Aravind vishnu, Assistant Professors, TIFAC-CORE inCyber Security, Amrita Vishwa Vidyapeetham, Coimbatore, for their valuable suggestions and timely feedbacks during the course of this major project. It was indeed a great support from them that helped me successfully fulfill this work

I would like to thank **Dr.M.Sethumadhavan**, Professor and Head of Department, TIFAC-CORE in Cyber Security, for his constant encouragement and guidance through-out the progress of this major project.

I convey special thanks to our friends for listening to our ideas and contributing their thoughts concerning the project. All those simple doubts from their part have also made us think deeper and understand about this work.

In particular, we would like also like to extend our gratitude to all the other faculties of TIFAC-CORE in Cyber Security and all those people who have helped us in many ways for the successful completion of this project.

TABLE OF CONTENTS

Introduction 3

Computational Aspect 5

Code.. 6

Comman Modulas Attack 11

Conclusion 13

Introduction:

RSA Encryption Scheme:

Number Theory Basis:

- Security of RSA relies on the difficulty of factoring large semiprime numbers into their prime components.
- The core mathematical problem is the relationship between Euler's totient function ($\varphi(N)$) and the public and private exponents (e and d).

Public and Private Key Pairs:

- The strength of RSA lies in the generation of a public key that can be freely distributed while keeping the corresponding private key secret.
- The security is based on the challenge of deriving the private key from the public key.

Key Exchange and Digital Signatures:

- RSA is not only used for encryption but also for key exchange and digital signatures.
- Public keys are used to encrypt messages, and private keys are used to decrypt them, ensuring confidentiality.
- The roles of public and private keys can be reversed for digital signatures, ensuring authenticity.

ElGamal Encryption Scheme:

Discrete Logarithm Problem:

- The security foundation of ElGamal is rooted in the difficulty of the discrete logarithm problem.
- Given y=g^x(modp), finding x is computationally infeasible when g, y, and p are known.

Key Generation and Primitive Roots:

- The public-private key pair is generated based on a large prime p, a primitive root g modulo p, and a secret exponent x.
- The security relies on the difficulty of determining x given g, p, and y=gx(modp).

Randomness in Encryption:

- ElGamal introduces an element of randomness in the encryption process through the choice of a random value k.
- This randomness enhances the security and prevents attackers from exploiting patterns in the encryption process.

Security in Diffie-Hellman Key Exchange:

- ElGamal encryption is connected to the Diffie-Hellman key exchange, where the public keys exchanged are used to derive a shared secret.
- The security of both schemes relies on the underlying difficulty of the discrete logarithm problem.

Computational Aspects of RSA:

Key Generation:

Select two large prime numbers p and q.
Compute N=p×q.
Calculate φ(N)=(p−1)×(q−1).
Choose e such that 1<e<φ(N) and e is coprime to φ(N).
Calculate d such that d≡e−1(modφ(N)).

Encryption:

To encrypt a message M, compute)C≡Me(modN).

Decryption:

To decrypt the ciphertext C, compute M≡Cd(modN).

Computational Aspects of ElGamal:

Key Generation:

Choose a large prime p.
Select a primitive root g modulo p.
Choose a secret key x randomly from [1,p−2].
Compute y=gx(modp).

Encryption:

To encrypt a message M:
Choose a random k.
Compute C1≡g^k(modp).
Compute C2≡M×y^k(modp).
The ciphertext is (C1,C2).

Decryption:

To decrypt the ciphertext (C1,C2):
Compute s≡C1x(modp).
Compute M≡C2×^s−1(modp), where s^−1 is the modular multiplicative inverse of s modulo p.

Code:

RSA

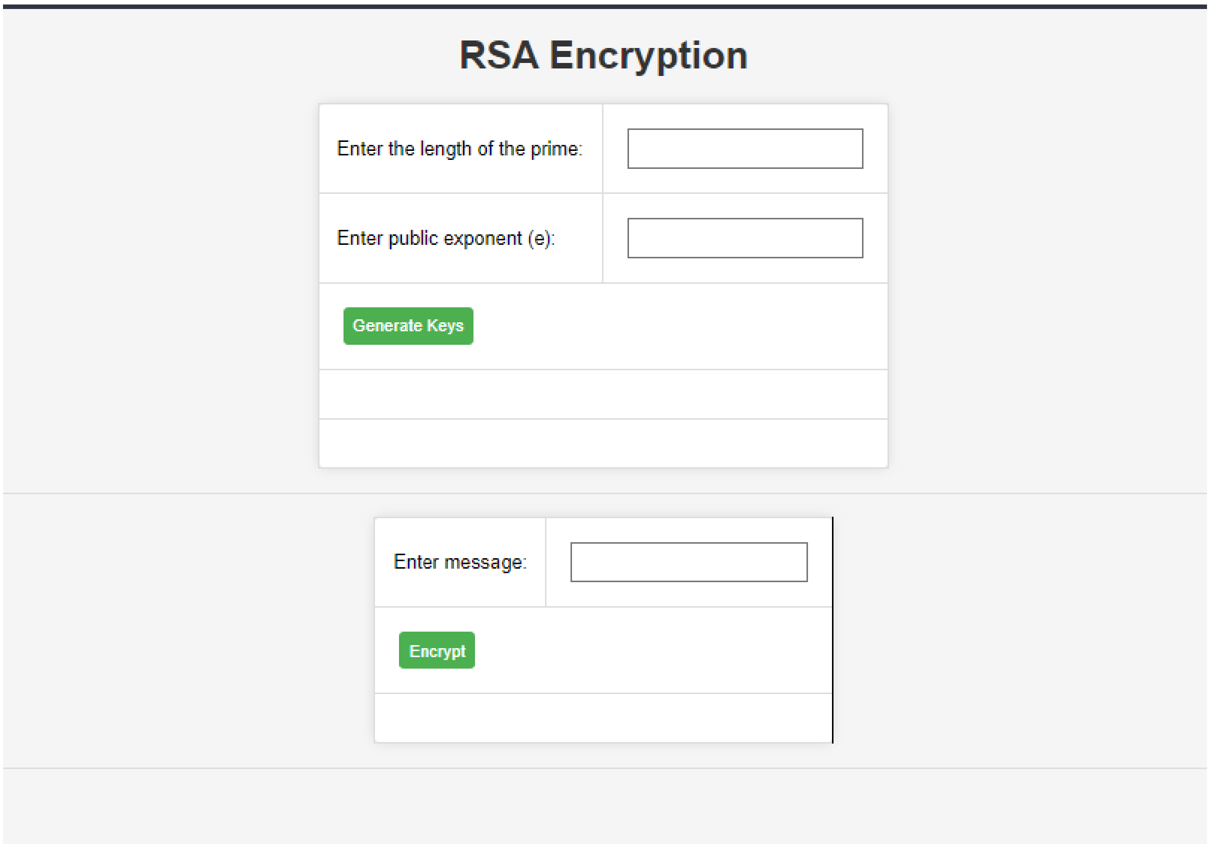
```
import random
prime_numbers = [61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109,
113, 127, 131, 137, 139, 149, 151, 157]

# Function to pick any two random numbers from the list
def pick_random_numbers():
```

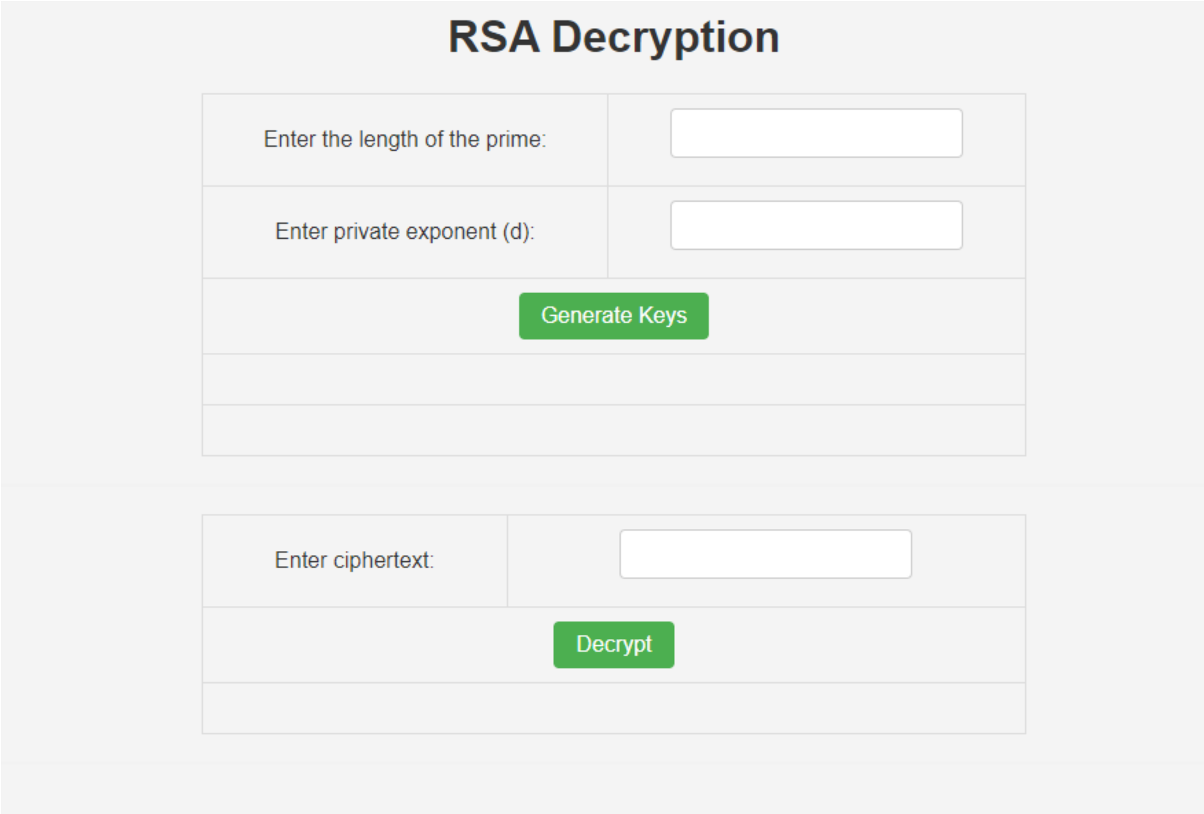
```
    return random.sample(prime_numbers,2)

# Function to calculate gcd
def gcd(a, b):
  while b:
      a, b = b, a % b
  return a

# Function to calculate modular inverse using the extended
Euclidean algorithm
def mod_inverse(a, m):
  m0, x0, x1 = m, 0, 1
  while a > 1:
      q = a // m
      m, a = a % m, m
      x0, x1 = x1 - q * x0, x0
  return x1 + m0 if x1 < 0 else x1

# Function to encrypt a message using RSA
def encrypt(message, public_key):
  e, n = public_key
  cipher_text = [pow(ord(char), e, n) for char in message]
  return cipher_text

# Function to decrypt a cipher text using RSA
def decrypt(cipher_text, private_key):
  d, n = private_key
  decrypted_message = [chr(pow(char, d, n)) for char in
cipher_text]
  return ''.join(decrypted_message)

# Input prime numbers p and q
primes=pick_random_numbers()
p = primes[0]
q = primes[1]
print("p= ",p)
print("q= ",q)
# Calculate n and phi(n)
n = p * q
phi_of_n = (p - 1) * (q - 1)
```

```
print("n:",n)
print("phi(n):",phi_of_n)

# Input public exponent e
e = int(input("Enter e: "))
while e < phi_of_n:
  if gcd(e, phi_of_n) == 1:
      break
  else:
      e = int(input("Enter a valid e: "))

# Calculate private exponent d (modular multiplicative inverse of e
modulo phi_of_n)
d = mod_inverse(e, phi_of_n)

# Generate public and private keys
public_key = (e, n)
private_key = (d, n)

print(f"Public Key: {public_key}")
print(f"Private Key: {private_key}")

# Input message for encryption
message = input("Enter the message to encrypt: ")

# Encrypt using the public key
cipher_text = encrypt(message, public_key)
print(f"Encrypted Message: {cipher_text}")

# Decrypt using the private key
decrypted_message = decrypt(cipher_text, private_key)
print(f"Decrypted Message: {decrypted_message}")
```

```
p=  97
q=  113
n: 10961
phi(n): 10752
Enter e:
77
Enter a valid e:
13
Public Key: (13, 10961)
Private Key: (9925, 10961)
Enter the message to encrypt:
hello
Encrypted Message: [4597, 7174, 3444, 3444, 1187]
Decrypted Message: hello
```

UI:





ELGAMAL

```
import random
from math import pow
from math import sqrt
def gcd(a, b):
    if a < b:
        return gcd(b, a)
    elif a % b == 0:
        return b
    else:
        return gcd(b, a % b)
def is_prime(num):
    if num < 2:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True
def power(a, b, c):
```

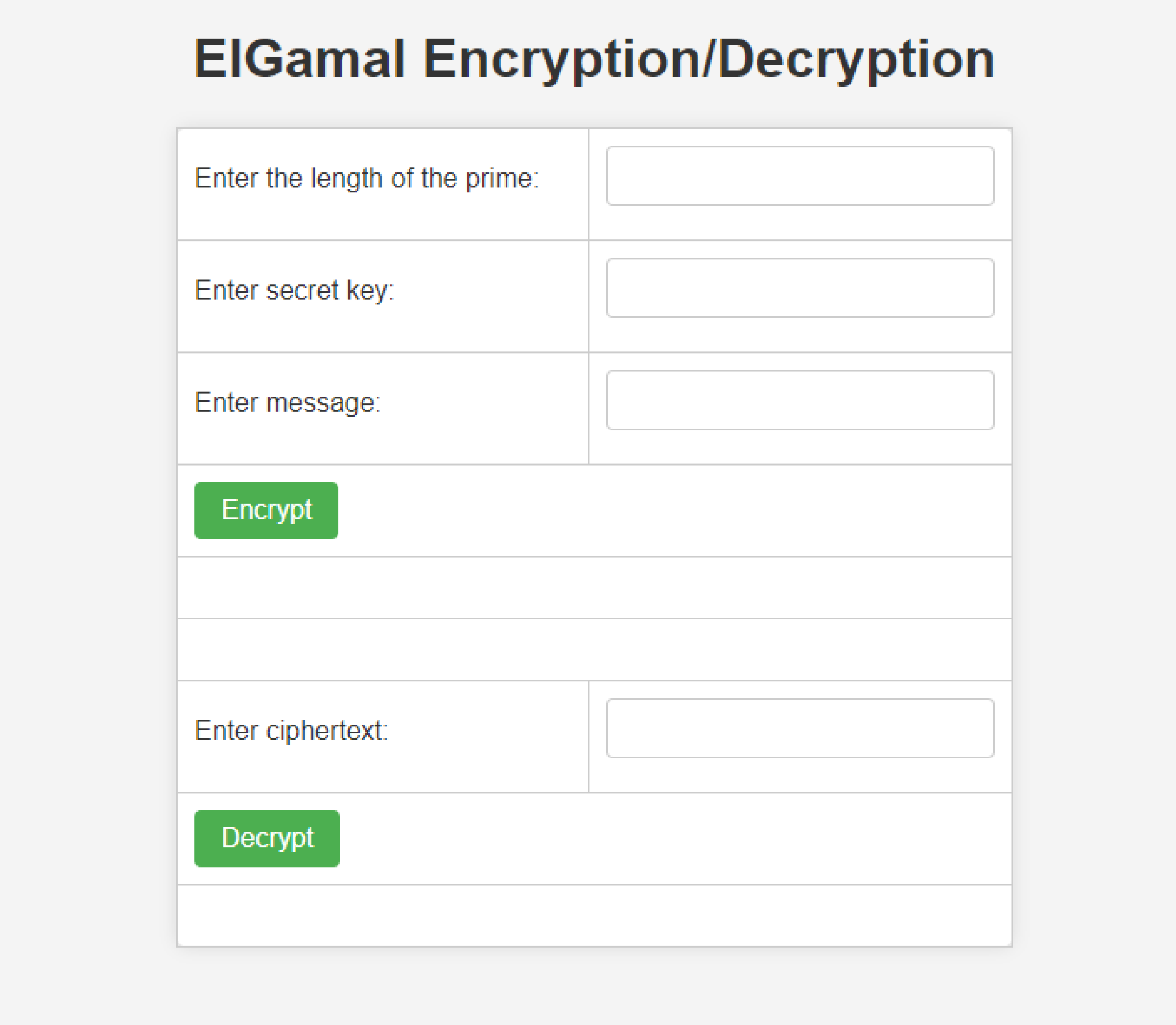
```
    x = 1
    y = a
    while b > 0:
        if b % 2 == 1:
            x = (x * y) % c
        y = (y * y) % c
        b = int(b / 2)
    return x % c
def findPrimefactors(s, n) :
    while (n % 2 == 0) :
        s.add(2)
        n = n // 2
    for i in range(3, int(sqrt(n)), 2):
        while (n % i == 0) :
            s.add(i)
            n = n // i
    if (n > 2) :
        s.add(n)
def gen_key(n):
    key = random.randint(pow(2, n), pow(2,n+2))
    while is_prime(n)==False:
        key = random.randint(pow(2, n), pow(2,n+2))
    return key
def findPrimitive(n) :
    s = set()
    phi = n - 1
    findPrimefactors(s, phi)
    for r in range(2, phi + 1):
        flag = False
        for it in s:
            if (power(r, phi // it, n) == 1):
                flag = True
                break
        if (flag == False):
            return r
```

```
        break

def mod_inverse(A, M):

    for X in range(1, M):
        if (((A % M) * (X % M)) % M == 1):
            return X
    return -1

def encryption(msg, q, h, g):
    ct = []
    k = gen_key(q)
    s = power(h, k, q)
    p = power(g, k, q)
    for i in range(0, len(msg)):
        ct.append(s * ord(msg[i]) % q)
    return ct, p

def decryption(ct, p, key, q):
    pt = []
    h = power(p, key, q)
    for i in range(0, len(ct)):
        pt.append(chr(int(ct[i] * pow(h, -1, q) % q)))
    return pt

length=int(input("Enter the length of the prime"))
msg = int(input("Enter message: "))
p=gen_key(length)
g = findPrimitive(p)
key = int(input("secret key"))
h = power(g, key, p)
print("g used=", g)
print("g^a used=", h)
rand=random.randint(2,p-1)
c1=power(g,rand,p)
```

```
c2=(msg*power(h,rand,p))%p
ct=[c1,c2]
print("Original Message=", msg)
print("Encrypted Message=", ct)
temp=mod_inverse(c1,p)
pt=(c2*power(temp,key,p))%p
print("Decrypted Messagge=", pt)
```

```
Enter the length of the prime
11
Enter message:
45
secret key
17
g used= 2
g^a used= 1624
Original Message= 45
Encrypted Message= [76, 3812]
Decrypted Messagge= 2072
```

UI:



Common Modulus Attack on RSA:

Overview:

The Common Modulus Attack targets RSA cryptosystems where multiple public exponents share the same modulus n. This attack exploits the common modulus to recover plaintext from distinct ciphertexts encrypted with different public keys.

Key Concepts:

Public and Private Keys:

RSA public keys include a modulus n and an exponent e.

Private keys comprise the modulus n and a secret exponent d.

Common Modulus Attack:

In certain scenarios, different public exponents e1 and e2 may use the same modulus n.

The attack relies on finding integers x and y such that x.e1+y.e2=1 (i.e., e1 and e2 are coprime).

Extended Euclidean Algorithm:

Finds x and y using the Extended Euclidean Algorithm, ensuring g=1.

If g is not 1, the attack is not feasible.

Calculating Private Keys:

For g=1, calculates private keys

2d1=xmode2

1d2=ymode1.

Recovering Plaintext:

Obtains plaintexts m1 and m2 from ciphertexts c1 and c2 using the respective private keys.

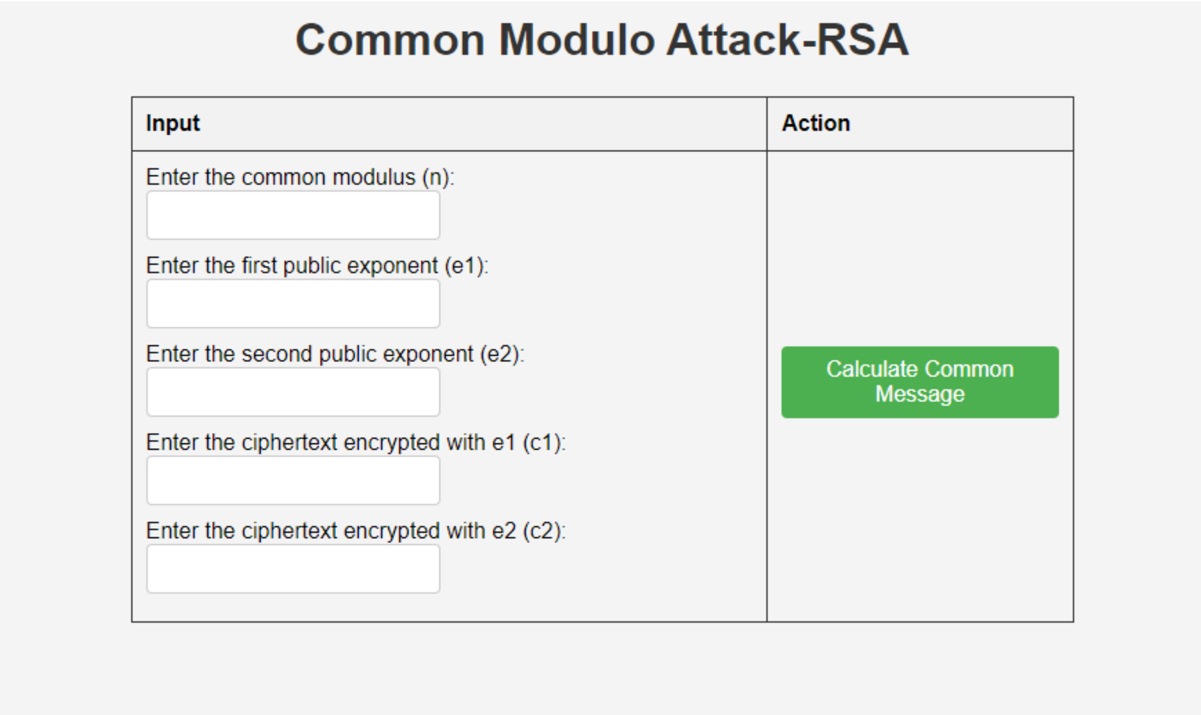
The common plaintext is m=(m1.m2)modn.

Limitations:

Effective when public exponents are not coprime.

Proper RSA key generation with coprime public exponents mitigates this attack.

UI:



Conclusion:

In conclusion, both RSA and ElGamal encryption schemes are fundamental cryptographic algorithms widely used to secure communications and protect sensitive information. The provided Python code snippets offer insights into the implementation of ElGamal encryption and certain aspects of RSA. Let's summarize the key points for your project report:
RSA Encryption:

Strengths:

- RSA is based on the difficulty of factoring large semiprime numbers, providing a robust foundation for secure communication.
- The security of RSA relies on the challenge of factoring the product of two large prime numbers.

Key Generation:

- Key generation involves selecting two large prime numbers and computing the public and private keys.
- The encryption and decryption processes require modular exponentiation.

Implementation Insights (from the provided code):

- The code includes functions for key generation, modular exponentiation, and encryption/decryption processes.
- A potential improvement could be the addition of padding schemes to enhance security.

ElGamal Encryption:

Strengths:

- ElGamal offers semantic security against chosen plaintext attacks.
- The security of ElGamal relies on the difficulty of the discrete logarithm problem.

Key Generation:

- Key generation involves choosing a large prime and a primitive root modulo the prime.
- Public and private keys are then generated based on these choices.

Implementation Insights (from the provided code):

- The code illustrates the key generation process, modular exponentiation, and encryption/decryption steps in the ElGamal cryptosystem.
- The inclusion of random values and primitive roots contributes to the security of the scheme.