COMSATS University Islamabad
(Attock Campus)

Information Security

LAB TERMINAL

Instructor: Ma'am Amreen Gul

DATE: Dec 16, 2025

Submitted By:

Muhammad Talha Mehmood

SP24-BSE-041

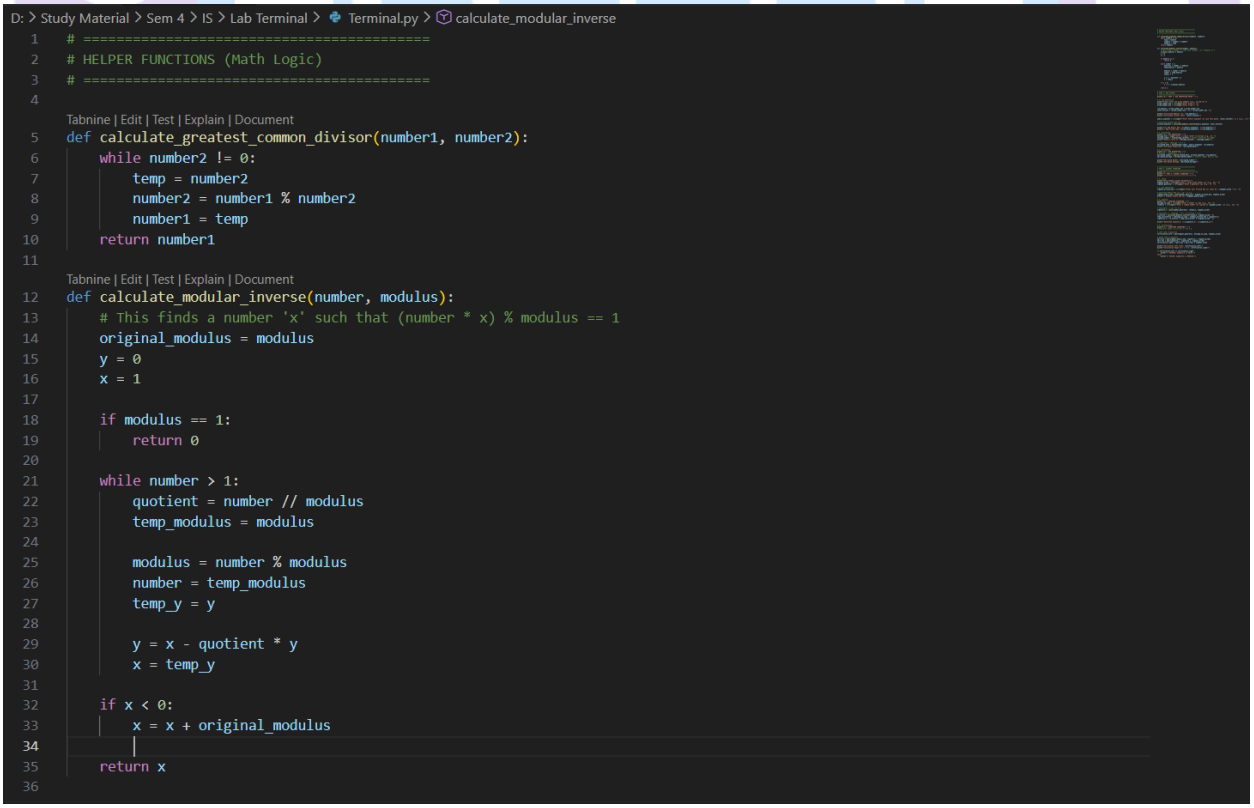
Question 1:

You are designing a secure email communication system that ensures the confidentiality, integrity, and authenticity of messages. The system uses RSA for encryption and ElGamal for signing messages.

Tasks:

1. Key Generation.
2. Encryption/Decryption using RSA.
3. Digital Signature and Verification using ElGamal.

Code Screenshots:



D: > Study Material > Sem 4 > IS > Lab Terminal > Terminal.py > calculate_modular_inverse

```
# ========================================
# HELPER FUNCTIONS (Math Logic)
# ========================================

Tabnine | Edit | Test | Explain | Document
def calculate_greatest_common_divisor(number1, number2):
    while number2 != 0:
        temp = number2
        number2 = number1 % number2
        number1 = temp
    return number1

Tabnine | Edit | Test | Explain | Document
def calculate_modular_inverse(number, modulus):
    # This finds a number 'x' such that (number * x) % modulus == 1
    original_modulus = modulus
    y = 0
    x = 1

    if modulus == 1:
        return 0

    while number > 1:
        quotient = number // modulus
        temp_modulus = modulus

        modulus = number % modulus
        number = temp_modulus
        temp_y = y

        y = x - quotient * y
        x = temp_y

    if x < 0:
        x = x + original_modulus

    return x
```

D: > Study Material > Sem 4 > IS > Lab Terminal > Terminal.py > calculate_modular_inverse

```python
# ========================================
# PART 1: RSA SYSTEM
# ========================================
print("\n--- PART 1: RSA ENCRYPTION SETUP ---")

# 1. Key Generation
print("Please enter two prime numbers (e.g., 11 and 17):")
prime_number_one = int(input("Enter Prime 1: "))
prime_number_two = int(input("Enter Prime 2: "))

rsa_modulus = prime_number_one * prime_number_two
euler_totient = (prime_number_one - 1) * (prime_number_two - 1)

print(f"Calculated Modulus (n): {rsa_modulus}")
print(f"Calculated Totient (phi): {euler_totient}")

public_exponent = int(input(f"Enter Public Exponent (e) such that gcd(e, {euler_totient}) is 1 (e.g., 3 or 7): "))

# Calculate Private Key (d)
private_exponent = calculate_modular_inverse(public_exponent, euler_totient)

print(f"\n> RSA Public Key: (e={public_exponent}, n={rsa_modulus})")
print(f"> RSA Private Key: (d={private_exponent}, n={rsa_modulus})")

# 2. Encryption
print("\n--- RSA ENCRYPTION ---")
message_text = input("Enter a single letter to encrypt (e.g., H): ")
message_number = ord(message_text[0]) # Convert character to number
print(f"Numeric value of '{message_text[0]}': {message_number}")

# Ciphertext = (Message ^ e) % n
encrypted_data = pow(message_number, public_exponent, rsa_modulus)
print(f"Encrypted Ciphertext: {encrypted_data}")

# 3. Decryption
print("\n--- RSA DECRYPTION ---")
# Message = (Ciphertext ^ d) % n
decrypted_number = pow(encrypted_data, private_exponent, rsa_modulus)
decrypted_message = chr(decrypted_number) # Convert number back to text

print(f"Decrypted Number: {decrypted_number}")
print(f"Decrypted Message: {decrypted_message}")



# ========================================
# PART 2: ELGAMAL SIGNATURE
# ========================================
print("\n-----------------------------------")
print("--- PART 2: ELGAMAL SIGNATURE ---")
print("-----------------------------------")

# 1. Setup
print("Setup ElGamal Global Parameters:")
elgamal_prime = int(input("Enter a large prime number (p) (e.g., 23): "))
elgamal_generator = int(input(f"Enter a generator (g) (e.g., 5): "))

# 2. Key Generation
elgamal_private_key = int(input(f"Enter your Private Key (x) (must be < {elgamal_prime - 1}): "))

# Calculate Public Key (y) = (g ^ x) % p
elgamal_public_key = pow(elgamal_generator, elgamal_private_key, elgamal_prime)
print(f"> ElGamal Public Key (y): {elgamal_public_key}")

# 3. Signing
print("\n--- CREATING SIGNATURE ---")
message_to_sign = int(input("Enter a number to sign (e.g., 72): "))
random_k = int(input(f"Enter a random number (k) coprime to {elgamal_prime - 1} (e.g., 13): "))

# Calculate r = (g ^ k) % p
signature_r = pow(elgamal_generator, random_k, elgamal_prime)

```

try.py | Terminal.py

D: > Study Material > Sem 4 > IS > Lab Terminal > Terminal.py > calculate_modular_inverse


```python
# 3. Decryption
print("\n--- RSA DECRYPTION ---")
# Message = (Ciphertext ^ d) % n
decrypted_number = pow(encrypted_data, private_exponent, rsa_modulus)
decrypted_message = chr(decrypted_number) # Convert number back to text

print(f"Decrypted Number: {decrypted_number}")
print(f"Decrypted Message: {decrypted_message}")



# ========================================
# PART 2: ELGAMAL SIGNATURE
# ========================================
print("\n-----------------------------------")
print("--- PART 2: ELGAMAL SIGNATURE ---")
print("-----------------------------------")

# 1. Setup
print("Setup ElGamal Global Parameters:")
elgamal_prime = int(input("Enter a large prime number (p) (e.g., 23): "))
elgamal_generator = int(input(f"Enter a generator (g) (e.g., 5): "))

# 2. Key Generation
elgamal_private_key = int(input(f"Enter your Private Key (x) (must be < {elgamal_prime - 1}): "))

# Calculate Public Key (y) = (g ^ x) % p
elgamal_public_key = pow(elgamal_generator, elgamal_private_key, elgamal_prime)
print(f"> ElGamal Public Key (y): {elgamal_public_key}")

# 3. Signing
print("\n--- CREATING SIGNATURE ---")
message_to_sign = int(input("Enter a number to sign (e.g., 72): "))
random_k = int(input(f"Enter a random number (k) coprime to {elgamal_prime - 1} (e.g., 13): "))

# Calculate r = (g ^ k) % p
signature_r = pow(elgamal_generator, random_k, elgamal_prime)

```

try.py | Terminal.py

D: > Study Material > Sem 4 > IS > Lab Terminal > Terminal.py > calculate_modular_inverse

```python
# 3. Signing
print("\n--- CREATING SIGNATURE ---")
message_to_sign = int(input("Enter a number to sign (e.g., 72): "))
random_k = int(input(f"Enter a random number (k) coprime to {elgamal_prime - 1} (e.g., 13): "))

# Calculate r = (g ^ k) % p
signature_r = pow(elgamal_generator, random_k, elgamal_prime)

# Calculate s = (message - x*r) * k_inverse % (p-1)
k_inverse = calculate_modular_inverse(random_k, elgamal_prime - 1)
temp_calculation = (message_to_sign - elgamal_private_key * signature_r)
signature_s = (k_inverse * temp_calculation) % (elgamal_prime - 1)

print(f"Generated Signature: (r={signature_r}, s={signature_s})")

# 4. Verification
print("\n--- VERIFYING SIGNATURE ---")
# Verify if (g^m) % p == (y^r * r^s) % p

# Left side of equation
verification_left = pow(elgamal_generator, message_to_sign, elgamal_prime)

# Right side of equation
part_one = pow(elgamal_public_key, signature_r, elgamal_prime)
part_two = pow(signature_r, signature_s, elgamal_prime)
verification_right = (part_one * part_two) % elgamal_prime

print(f"Calculation Left (g^m): {verification_left}")
print(f"Calculation Right (y^r * r^s): {verification_right}")

if verification_left == verification_right:
    print(">> SUCCESS: Signature is VALID.")
else:
    print(">> FAILED: Signature is INVALID.")
```

Output Screenshots:

```
PS C:\Users\HP> python -u "d:\Study Material\Sem 4\IS\Lab Terminal\Terminal.py"

--- PART 1: RSA ENCRYPTION SETUP ---
Please enter two prime numbers (e.g., 11 and 17):
Enter Prime 1: 11
Enter Prime 2: 17
Calculated Modulus (n): 187
Calculated Totient (phi): 160
Enter Public Exponent (e) such that gcd(e, 160) is 1 (e.g., 3 or 7): 3

> RSA Public Key: (e=3, n=187)
> RSA Private Key: (d=107, n=187)

--- RSA ENCRYPTION ---
Enter a single letter to encrypt (e.g., H): T
Numeric value of 'T': 84
Encrypted Ciphertext: 101

--- RSA DECRYPTION ---
Decrypted Number: 84
Decrypted Message: T

-----------------------------------
--- PART 2: ELGAMAL SIGNATURE ---
-----------------------------------
Setup ElGamal Global Parameters:
Enter a large prime number (p) (e.g., 23): 23
Enter a generator (g) (e.g., 5): 5
Enter your Private Key (x) (must be < 22): 12
> ElGamal Public Key (y): 18

--- CREATING SIGNATURE ---
Enter a number to sign (e.g., 72): 72
Enter a random number (k) coprime to 22 (e.g., 13): 13
Generated Signature: (r=21, s=20)

--- VERIFYING SIGNATURE ---
Calculation Left (g^m): 8
Calculation Right (y^r * r^s): 8
>> SUCCESS: Signature is VALID.
PS C:\Users\HP>
```

Question 2:

You have developed a project during this course as your final submission. Using that same project:

1. Justify your security method: Why is it suitable or better than other possible methods for your type of project?

Why is RSA suitable for this project?

For this project, I chose RSA because it is an asymmetric encryption standard that solves the key distribution problem. Unlike symmetric methods (like AES) where sender and receiver must secretly share the same key, my RSA implementation allows anyone to send me a message using my public key, but only I can read it with my private key. This is perfect for a secure communication system where users don't meet beforehand. Additionally, the security of RSA relies on the mathematical difficulty of factoring large prime numbers (the factoring problem), which is a well-proven standard for securing data in transit.

2. Identify one possible vulnerability or weakness in your current system. How could an attacker misuse it?

Weakness: Textbook RSA Vulnerability(Deterministic Ecryption)

A major weakness in my current implementation is that it is 'Textbook RSA' without padding (like OAEP). This means the encryption is deterministic: if you encrypt the same letter (e.g., 'H') multiple times with the same key, it always produces the exact same ciphertext number.

How an Attacker can misuse it?

An attacker who sees the ciphertext can perform a frequency analysis or a 'dictionary attack.' If they guess the message is 'HELLO', they can encrypt 'HELLO' with my public key themselves and compare their result to the intercepted message to confirm their guess, effectively breaking the confidentiality without needing the private key.

3. Suggest one realistic improvement to enhance the security of your project. Briefly explain how it would work.

Improvement: Add Padding etc

To fix the deterministic weakness, I would add random padding before encryption. Instead of encrypting just the character , I would encrypt a combination of the character and a random number (salt).

How it would work:

Before converting the character to a number, I would append a random string to it. For example, instead of encrypting 'A', I might encrypt 'A-39281'. This ensures that even if I send the letter 'A' ten times, the resulting ciphertext will look completely different each time because the random salt changes. This prevents dictionary attacks and frequency analysis.