Seiyun University, Yemen

Subject Name

# SPECIAL TOPICS IN

# INFORMATION SECURITY

Prepared by :

**walaa Abdaruhman**

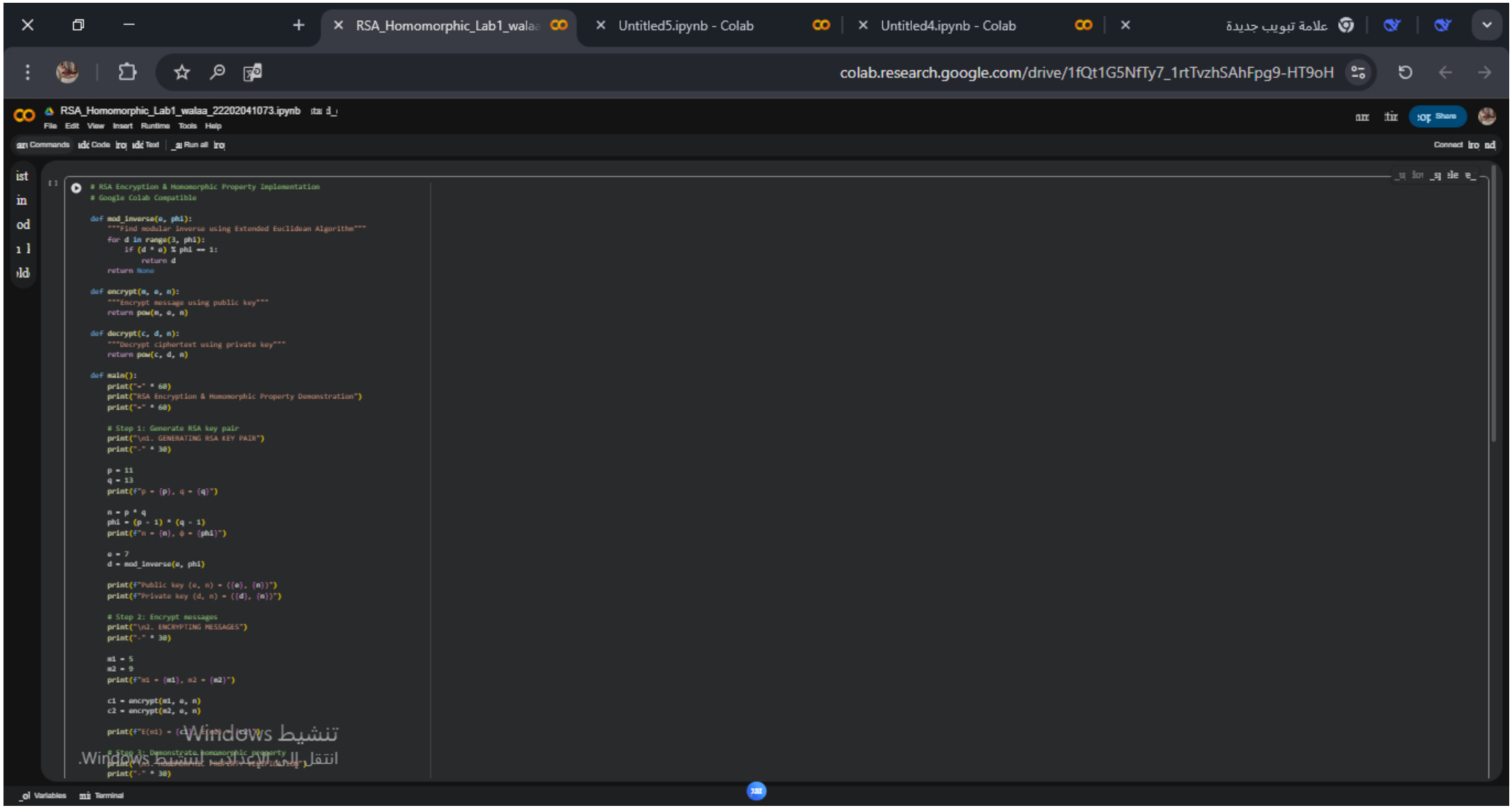
Registration number

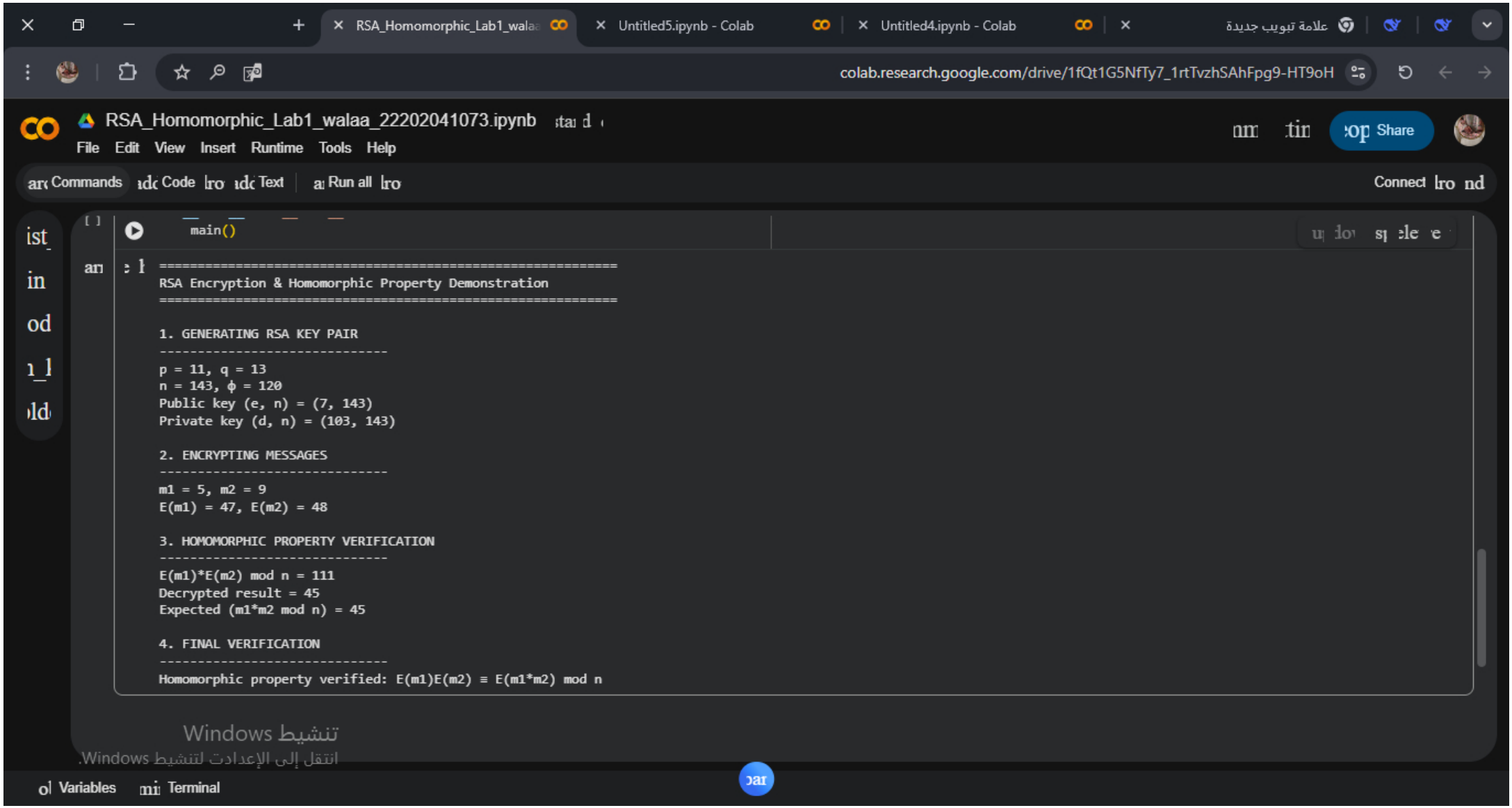
2202041073

information security

```
# RSA Encryption & Homomorphic Property Implementation
# Google Colab Compatible

def mod_inverse(e, phi):
    """Find modular inverse using Extended Euclidean Algorithm"""
    for d in range(3, phi):
        if (d * e) % phi == 1:
            return d
    return None

def encrypt(m, e, n):
    """Encrypt message using public key"""
    return pow(m, e, n)

def decrypt(c, d, n):
    """Decrypt ciphertext using private key"""
    return pow(c, d, n)

def main():
    print("=" * 60)
    print("RSA Encryption & Homomorphic Property Demonstration")
    print("=" * 60)

    # Step 1: Generate RSA key pair
    print("\n1. GENERATING RSA KEY PAIR")
    print("-" * 30)

    p = 11
    q = 13
    print(f"p = {p}, q = {q}")

    n = p * q
    phi = (p - 1) * (q - 1)
    print(f"n = {n}, ϕ = {phi}")

    e = 7
    d = mod_inverse(e, phi)

    print(f"Public key (e, n) = ({e}, {n})")
    print(f"Private key (d, n) = ({d}, {n})")

    # Step 2: Encrypt messages
    print("\n2. ENCRYPTING MESSAGES")
    print("-" * 30)

    m1 = 5
    m2 = 9
    print(f"m1 = {m1}, m2 = {m2}")

    c1 = encrypt(m1, e, n)
    c2 = encrypt(m2, e, n)

    print(f"E(m1) = {c1}, E(m2) = {c2}")

    # Step 3: Demonstrate homomorphic property
    print("\n3. HOMOMORPHIC PROPERTY VERIFICATION")
    print("-" * 30)

    ciphertext_product = (c1 * c2) % n
    print(f"E(m1)*E(m2) mod n = {ciphertext_product}")

    decrypted_result = decrypt(ciphertext_product, d, n)
    print(f"Decrypted result = {decrypted_result}")

    expected_result = (m1 * m2) % n
    print(f"Expected (m1*m2 mod n) = {expected_result}")

    # Step 4: Final verification
    print("\n4. FINAL VERIFICATION")
    print("-" * 30)

    if decrypted_result == expected_result:
        print("Homomorphic property verified: E(m1)E(m2) ≡ E(m1*m2) mod n")
    else:
        print("Homomorphic property verification failed")

# Run the demonstration
if __name__ == "__main__":
    main()
```

```
============================================================
RSA Encryption & Homomorphic Property Demonstration
============================================================

1. GENERATING RSA KEY PAIR
------------------------------
p = 11, q = 13
n = 143, ϕ = 120
Public key (e, n) = (7, 143)
Private key (d, n) = (103, 143)

2. ENCRYPTING MESSAGES
------------------------------
m1 = 5, m2 = 9
E(m1) = 47, E(m2) = 48

3. HOMOMORPHIC PROPERTY VERIFICATION
------------------------------
E(m1)*E(m2) mod n = 111
Decrypted result = 45
Expected (m1*m2 mod n) = 45

4. FINAL VERIFICATION
------------------------------
Homomorphic property verified: E(m1)E(m2) ≡ E(m1*m2) mod n
```


RSA_Homomorphic_Lab1_walaa
Untitled5.ipynb - Colab
Untitled4.ipynb - Colab
علامة تبويب جديدة
colab.research.google.com/drive/1fQt1G5NfTy7_1rtTvzhSAhFpg9-HT9oH
RSA_Homomorphic_Lab1_walaa_22202041073.ipynb
File Edit View Insert Runtime Tools Help
Share
Connect
# RSA Encryption & Homomorphic Property Implementation
# Google Colab Compatible
def mod_inverse(e, phi):
"""Find modular inverse using Extended Euclidean Algorithm"""
for d in range(3, phi):
if (d * e) % phi == 1:
return d
return None
def encrypt(m, e, n):
"""Encrypt message using public key"""
return pow(m, e, n)
def decrypt(c, d, n):
"""Decrypt ciphertext using private key"""
return pow(c, d, n)
def main():
print("=" * 60)
print("RSA Encryption & Homomorphic Property Demonstration")
print("=" * 60)
# Step 1: Generate RSA key pair
print("\n1. GENERATING RSA KEY PAIR")
print("-" * 30)
p = 11
q = 13
print(f"p = {p}, q = {q}")
n = p * q
phi = (p - 1) * (q - 1)
print(f"n = {n}, φ = {phi}")
e = 7
d = mod_inverse(e, phi)
print(f"Public key (e, n) = ({e}, {n})")
print(f"Private key (d, n) = ({d}, {n})")
# Step 2: Encrypt messages
print("\n2. ENCRYPTING MESSAGES")
print("-" * 30)
m1 = 5
m2 = 9
print(f"m1 = {m1}, m2 = {m2}")
c1 = encrypt(m1, e, n)
c2 = encrypt(m2, e, n)
print(f"E(m1) = {c1}, E(m2) = {c2}")
# Step 3: Demonstrate homomorphic property
print("-" * 30)
Variables
Terminal
تنشيط Windows
انتقل إلى الإعدادات لتنشيط Windows.


RSA_Homomorphic_Lab1_walaa
Untitled5.ipynb - Colab
Untitled4.ipynb - Colab
علامة تبويب جديدة
colab.research.google.com/drive/1fQt1G5NfTy7_1rtTvzhSAhFpg9-HT9oH
RSA_Homomorphic_Lab1_walaa_22202041073.ipynb
File Edit View Insert Runtime Tools Help
Share
Commands
Code
Text
Run all
Connect
main()
==========
RSA Encryption & Homomorphic Property Demonstration
==========
1. GENERATING RSA KEY PAIR
---
p = 11, q = 13
n = 143, ϕ = 120
Public key (e, n) = (7, 143)
Private key (d, n) = (103, 143)
2. ENCRYPTING MESSAGES
---
m1 = 5, m2 = 9
E(m1) = 47, E(m2) = 48
3. HOMOMORPHIC PROPERTY VERIFICATION
---
E(m1)*E(m2) mod n = 111
Decrypted result = 45
Expected (m1*m2 mod n) = 45
4. FINAL VERIFICATION
---
Homomorphic property verified: E(m1)E(m2) ≡ E(m1*m2) mod n
Windows تنشيط
انتقل إلى الإعدادات لتنشيط Windows.
Variables
Terminal

# Lab Assignment No. 01

Q1. Define RSA algorithm and explain each step ?

- RSA is like a secure locks and key system it use tow key one public (like a lock every one can use) and one private (like the key only you haves
- RSA is a public-key cryptosystem that enables secure data transmission. It's based on the practical difficulty of factoring large integers

- Here is how it works :-

1. Key Generation:

- Select two distinct prime numbers p and q
- Compute $n = p \times q$
- Calculate Euler's totient: $\phi(n) = (p-1)(q-1)$
- choose an integer e such that

$1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$

- Determine d where $d = e^{-1} \bmod \phi(n)$

2. Keys :- - public (e, n) , Private (d, n

3. Operations :-

- Encryption : $c = m^e \bmod n$
- Decryption : $m = c^d \bmod n$

Q2 : What is the homomorphic property--?

Answer:

-Homomorphic encryption allows specific types of computations to be performed on ciphertexts generating an encrypted result that, when decrypt- matches the result of operation performed on the plaintexts. This means we can process encrypted data without decrypting it first maintaining ~~Confiden~~ Confidentiality throughout the computation process.

---

Q3 Explain why RSA is multiplicatively homomorphic?

Answer:

RSA exhibits multiplicative homomorphism because of its mathematical structure:

$$E(m_1) \times E(m_2) = (m_1^e \bmod n) \times (m_2^e \bmod n) \Rightarrow$$
$$(m_1 \times m_2)^e \bmod n = E(m_1 \times m_2).$$

This shows that multiplying two ciphertexts together and reducing modulo n produces the same result as first multiplying the plaintexts and then encrypting the product.

Q4: What are the advantages and risks of homomorphic encryption?

- Answer:-

• Advantages:-

- Privacy Preservation: Data remains encypted during processing.
- Secure Outsourcing: Enables Computation on sensitive data in nutrusted environments
- Regulatory Compliance: Helps meet data protection requirements.

• Risks:-

- It Can Slow, especially Fully homomorphic encryption
- Attackers might manipulate ciphertexts to learn Something about the Original data
- Still a relatively new Field, So there might be unknown vulnerabilites.

Q5: Why do we use the mod n operation in RSA?

- The mod n keeps within a fixed range (0 to n-1) , which makes computations manageable and ensures everything wraps around correctly, Think of it like a clock- once you hit 12 you start back at 1 This cyclic behavior is essential for RSA's security and correctness.

---

Q6 Compute manually for a small case $p=3$, $q=11$ $e=7$, $m_1=2$, $m_2=5$?

- Let's breaks it down :-

$n = 3 \times 11 = 33$ $\quad \phi(n) = (3-1)(11-1) = 2 \times 10 = 20$

$e = 7$ $\quad$ * $d = 3$ (because $7 \times 3 = 21 \bmod 20 = 1$)

- Encrypt $m_1 = 2$: $2^7 = 128$, $128 \bmod 33 = 29$

Encrypt $m_2 = 5$: $5^7 = 78125$, $78125 \bmod 33 = 14$

- Multiply ciphertexts: $29 \times 14 = 406$, $406 \bmod 33 = 10$

- Compare with plaintext product $2 \times 5 = 10$

* it matches the homomorphic property works.

Q7 Why is key size important for security?
- larger keys are like bigger, more complex locks
if the key size is too small, attackers can factor n quickly
and breaks the encryption. ~~whi~~With larger keys (e.g 2048
bits or more), factoring becomes practically impossible with
current techology.

---

Q8 Explain what happens if e and $\phi(n)$ are not coprime?
if e and $\phi(n)$ share a factor, then e doesn't have a
modular inverse modulo $\phi(n)$. That means you can't find d
the private exponent without d you can't decrypt messages
- so the whole system breaks down.

---

Q9 Describe one real-world application of homomorphic
encryption?
A great example is secure voting system. Votes can be
encrypted and tallied without ever being decrypted.
ensuring voter privacy while still providing accurate
results. Another example is medical research, where
hospitals can share encrypted patient data for analysis
without exposing sensitive information.