Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Cryptography And Security

Laboratory work #5

Block Ciphers. DES Algorithm.

Author:
Alexandru Rudoi
std. gr. FAF-231

Verified:
Maia Zaica
asist. univ.

Chișinău 2025

Contents

Objective

The goal of this laboratory work is to implement and demonstrate the functionality of three fundamental cryptographic algorithms: RSA, ElGamal, and Diffie-Hellman. Through the implementation of these algorithms, the principles of **public-key cryptography**, asymmetric encryption/decryption mechanisms, and **secure key exchange** processes will be explored.

The specific objectives include:

- Implementing the RSA algorithm with at least 2048-bit keys.
- Implementing the ElGamal algorithm with specified parameters.
- Demonstrating the Diffie-Hellman key exchange to generate an AES-256 key.

Task Description

The laboratory work consists of the following tasks:

- **Task 2.1 - RSA Algorithm**: Generate RSA keys with at least 2048 bits and perform encryption and decryption for the message ‘m = ”Nume Prenume”‘.
- **Task 2.2 - ElGamal Algorithm**: Generate ElGamal keys using specified parameters ‘p‘ and ‘g‘, and perform encryption and decryption for the message ‘m = ”Nume Prenume”‘.
- **Task 3 - Diffie-Hellman Key Exchange**: Perform Diffie-Hellman key exchange between Alice and Bob, using the algorithm to derive a 256-bit AES key.

For tasks 2.1 and 2.2, the message will be represented numerically using its ASCII hexadecimal representation.

Theoretical Background

RSA Algorithm

RSA is an asymmetric encryption algorithm based on the difficulty of factoring large prime numbers. It provides both confidentiality (encryption) and authenticity (digital signatures).

Key Generation:

1. Choose two large prime numbers p and q.
2. Compute $n = p \times q$ and $\phi(n) = (p-1)(q-1)$.

3. Choose an encryption exponent e such that $\gcd(e, \phi(n)) = 1$.
4. Compute the decryption exponent d, which is the modular inverse of e modulo $\phi(n)$.
5. Public key: (n, e), Private key: (n, d).

Encryption and Decryption:

1. Encryption: $c = m^e \mod n$.
2. Decryption: $m = c^d \mod n$, where m is the message and c is the ciphertext.

Security: The security of RSA is based on the assumption that factoring large numbers is computationally infeasible.

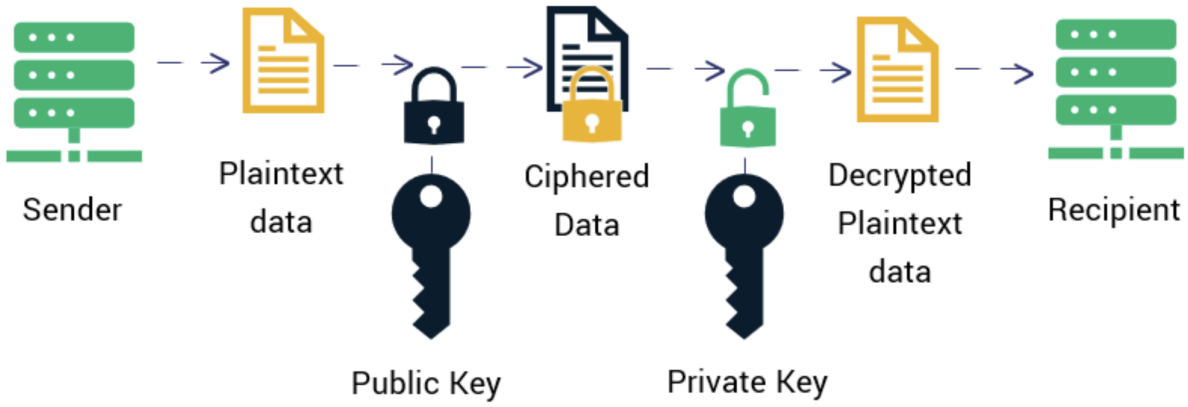

Figure 1: RSA Key Generation and Encryption/Decryption Process

ElGamal Algorithm

ElGamal is an asymmetric encryption algorithm based on the difficulty of computing discrete logarithms in a finite field.

Key Generation:

1. Choose a large prime p and a generator g.
2. Choose a private key x and compute the public key $y = g^x \mod p$.
3. Public key: (p, g, y), Private key: x.

Encryption and Decryption:

1. Choose a random integer k.
2. Compute ciphertext as $c_1 = g^k \mod p$ and $c_2 = m \times y^k \mod p$.
3. Decrypt the ciphertext by calculating $s = c_1^x \mod p$ and then $m = c_2 \times s^{-1} \mod p$.

Security: The security of ElGamal is based on the difficulty of the discrete logarithm problem.

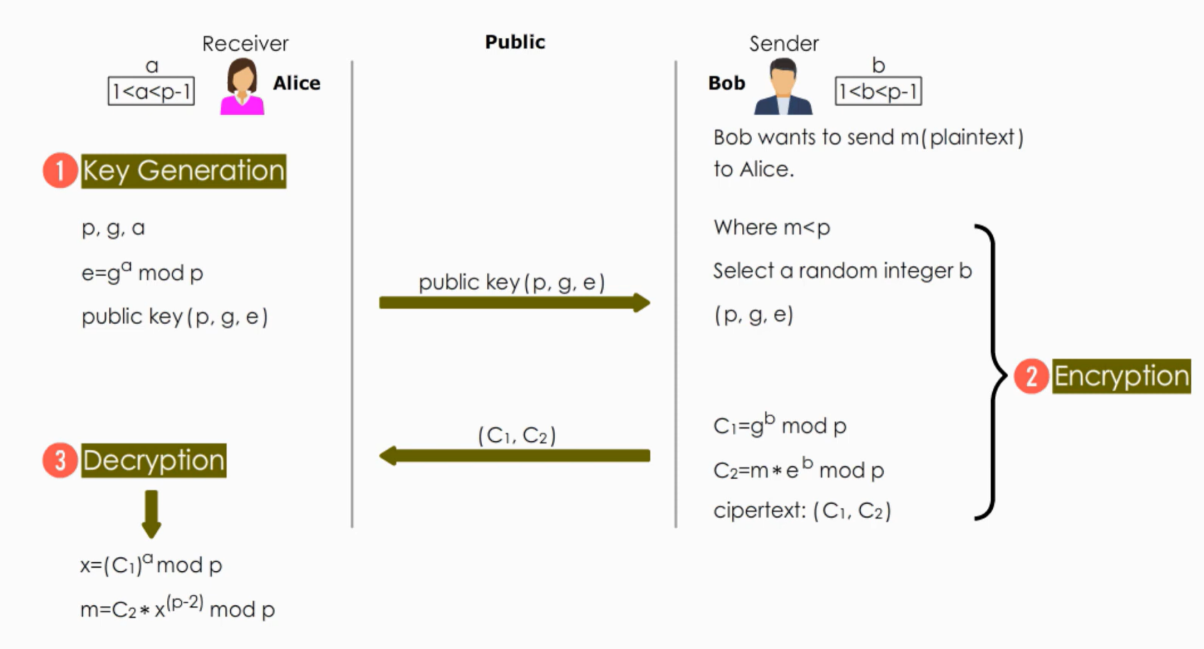

Figure 2: ElGamal Encryption and Decryption Process

Diffie-Hellman Key Exchange

Diffie-Hellman is a key exchange protocol that allows two parties to securely establish a shared secret over an insecure channel.

Key Exchange Process:

1. Choose a large prime p and a generator g.
2. Alice and Bob each select a private key a and b, respectively.
3. Alice computes $A = g^a \mod p$, and Bob computes $B = g^b \mod p$.
4. Alice and Bob exchange A and B, respectively.
5. Alice computes the shared key $K_A = B^a \mod p$, and Bob computes $K_B = A^b \mod p$.
6. The shared keys K_A and K_B are the same.

Security: The security of Diffie-Hellman relies on the difficulty of computing discrete logarithms.

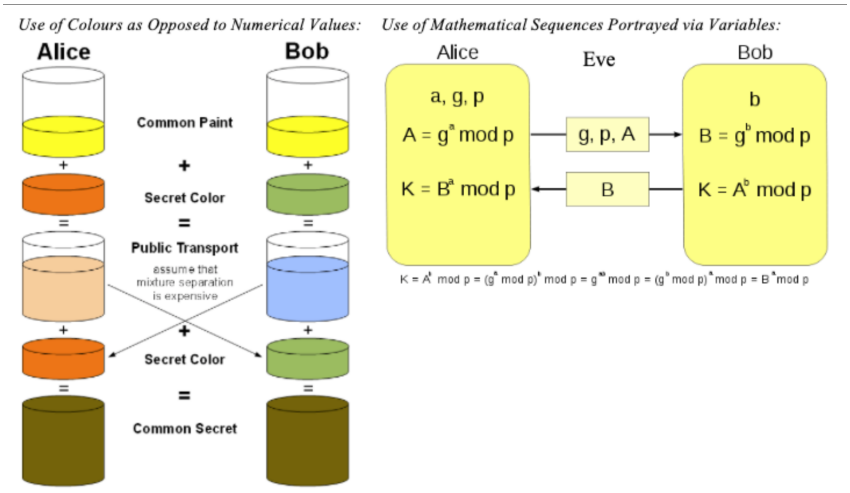

Figure 3: Diffie-Hellman Key Exchange Process

Technical Implementation

In this section, we describe the implementation of the **RSA**, **ElGamal**, and **Diffie-Hellman** algorithms. For each algorithm, we will provide the **key generation**, **encryption**, and **decryption** steps along with the corresponding Python code snippets.

RSA Algorithm Implementation

The RSA algorithm is based on the difficulty of factoring large prime numbers and relies on the use of a public and private key pair for encryption and decryption.

1. Key Generation

In RSA, we first generate two large prime numbers, p and q, and compute the modulus $n = p \times q$. We also calculate the totient function $\phi(n) = (p-1)(q-1)$ and choose a public exponent e. The private exponent d is calculated as the modular inverse of e modulo $\phi(n)$.

```python
from sympy import randprime, gcd, mod_inverse

def generate_rsa_keys(bits=2048):
    p = randprime(2**(bits//2 - 1), 2**(bits//2))
    q = randprime(2**(bits//2 - 1), 2**(bits//2))
    n = p * q
    phi_n = (p - 1) * (q - 1)
    e = 65537
    while gcd(e, phi_n) != 1:
```

```
        e += 2
    d = mod_inverse(e, phi_n)
    return {'public_key': (n, e), 'private_key': (n, d), 'p': p, 'q': q}
```

Listing 1: RSA Key Generation

Explanation:

1. Generate two large prime numbers, p and q.
2. Compute $n = p \times q$ and $\phi(n) = (p-1)(q-1)$.
3. Choose the public exponent e (commonly $e = 65537$).
4. Compute the private exponent d using the modular inverse of e modulo $\phi(n)$.

2. Encryption

To encrypt a message m using the RSA algorithm, we use the public key (n, e) and compute the ciphertext $c = m^e \mod n$.

```
def rsa_encrypt(message, public_key):
    n, e = public_key
    m = message_to_number(message) # Convert message to number
    if m >= n:
        raise ValueError("Message is too large for the RSA key")
    c = pow(m, e, n) # c = m^e mod n
    return c
```

Listing 2: RSA Encryption

Explanation:

1. Convert the message m to its numeric representation using ASCII encoding.
2. Compute the ciphertext $c = m^e \mod n$ using the public key.

3. Decryption

Decryption is done using the private key (n, d) and the ciphertext c. The message m is recovered by calculating $m = c^d \mod n$.

```
def rsa_decrypt(ciphertext, private_key):
    n, d = private_key
    m = pow(ciphertext, d, n) # m = c^d mod n
    message = number_to_message(m) # Convert number back to message
    return message
```

Listing 3: RSA Decryption

Explanation:

1. Decrypt the ciphertext by computing $m = c^d \mod n$ using the private key.
2. Convert the numeric message back to its string representation.

—

ElGamal Algorithm Implementation

The ElGamal algorithm is based on the difficulty of computing discrete logarithms in a finite field.

1. Key Generation

In ElGamal, a large prime p and a generator g are chosen. A private key x is selected, and the corresponding public key y is computed as $y = g^x \mod p$.

```python
import random

def generate_elgamal_keys():
    p = 323170060713110073001535134778251633624880571334890751745884341392 6 # large prime p
    g = 2 # generator g
    x = random.randint(2, p - 2) # private key
    y = pow(g, x, p) # public key y = g^x mod p
    return {'public_key': (p, g, y), 'private_key': x}
```

Listing 4: ElGamal Key Generation

Explanation:

1. Select a large prime p and a generator g.
2. Choose a private key x and compute the corresponding public key $y = g^x \mod p$.

2. Encryption

To encrypt a message m, a random integer k is chosen, and the ciphertext is computed as $c_1 = g^k \mod p$ and $c_2 = m \times y^k \mod p$.

```python
def elgamal_encrypt(message, public_key):
    p, g, y = public_key
```

```
    m = message_to_number(message) # Convert message to number
    k = random.randint(2, p - 2) # choose random k
    c1 = pow(g, k, p) # g^k mod p
    c2 = (m * pow(y, k, p)) % p # m * y^k mod p
    return (c1, c2)
```

Listing 5: ElGamal Encryption

Explanation:

1. Select a random integer k such that $2 \leq k \leq p-2$.
2. Compute the ciphertext components $c_1 = g^k \mod p$ and $c_2 = m \times y^k \mod p$.

3. Decryption

The ciphertext (c_1, c_2) is decrypted using the private key x by calculating $s = c_1^x \mod p$ and recovering the message $m = c_2 \times s^{-1} \mod p$.

```
def elgamal_decrypt(ciphertext, private_key, p):
    c1, c2 = ciphertext
    x = private_key
    s = pow(c1, x, p) # s = c1^x mod p
    s_inv = mod_inverse(s, p) # modular inverse of s
    m = (c2 * s_inv) % p # m = c2 * s_inv mod p
    message = number_to_message(m)
    return message
```

Listing 6: ElGamal Decryption

Explanation:

1. Calculate $s = c_1^x \mod p$, where x is the private key.
2. Compute the inverse s^{-1} modulo p, and recover the message by calculating $m = c_2 \times s^{-1} \mod p$.

—

Diffie-Hellman Key Exchange Implementation

The Diffie-Hellman algorithm allows two parties to exchange a shared secret key over an insecure channel.

1. Key Exchange

Alice and Bob each select private keys a and b, respectively. They compute their corresponding public values $A = g^a \mod p$ and $B = g^b \mod p$, exchange them, and compute the shared secret key.

```python
def diffie_hellman_exchange():
    p = 32317006071311007300153513477825163362488057133489075174588434139 26 # prime p
    g = 2 # generator g
    a = random.randint(2, p - 2) # Alice's private key
    b = random.randint(2, p - 2) # Bob's private key
    A = pow(g, a, p) # Alice computes A = g^a mod p
    B = pow(g, b, p) # Bob computes B = g^b mod p
    shared_key_alice = pow(B, a, p) # Alice computes shared key
    shared_key_bob = pow(A, b, p) # Bob computes shared key
    assert shared_key_alice == shared_key_bob, "Keys do not match!"
    return {'shared_key': shared_key_alice}
```

Listing 7: Diffie-Hellman Key Exchange

Explanation:

1. Alice and Bob select their private keys, a and b.
2. They compute their corresponding public values $A = g^a \mod p$ and $B = g^b \mod p$.
3. Alice and Bob compute the shared secret key independently using the public values.

Results

In this section, the results of the RSA, ElGamal, and Diffie-Hellman implementations are shown through the corresponding screenshots. Each screenshot is accompanied by a brief explanation of the operations and outcomes.

RSA Algorithm Results

- **Screenshot 1: RSA Key Generation** In this screenshot, the RSA algorithm

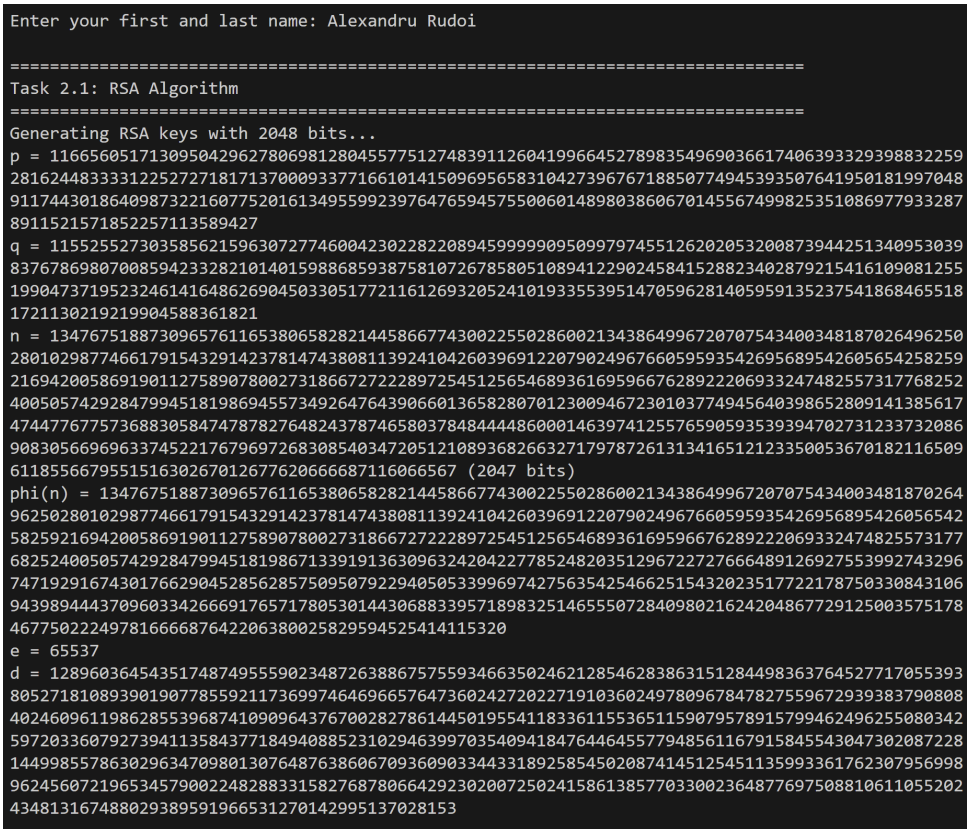

Figure 4: RSA Key Generation and Display of Public/Private Keys

has successfully generated the **public and private keys**. The public key consists of ‘n‘ and ‘e‘, and the private key contains ‘n‘ and ‘d‘. The process involves selecting two large primes, computing ‘n‘, and determining the corresponding private exponent ‘d‘.

- **Screenshot 2: RSA Encryption and Decryption** This screenshot shows the

Figure 5: RSA Encryption and Decryption Process

encryption of the message '"Nume Prenume"' using the public key, resulting in an encrypted ciphertext. Following this, the ciphertext is decrypted back to the original message using the private key. The correct decryption confirms the functionality of the RSA algorithm.

ElGamal Algorithm Results

- **Screenshot 3: ElGamal Key Generation** This screenshot demonstrates the

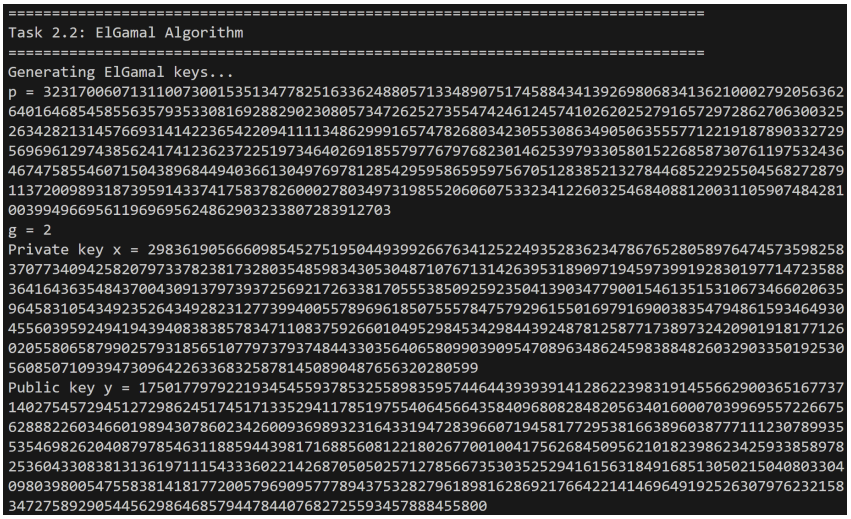

Figure 6: ElGamal Key Generation and Public Key Display

ElGamal key generation, where a **private key** ‘x‘ is randomly chosen and the corresponding **public key** ‘y‘ is computed. The parameters ‘p‘ and ‘g‘ are predefined, and ‘y‘ is computed as $g^x \mod p$.

- **Screenshot 4: ElGamal Encryption and Decryption** The screenshot shows

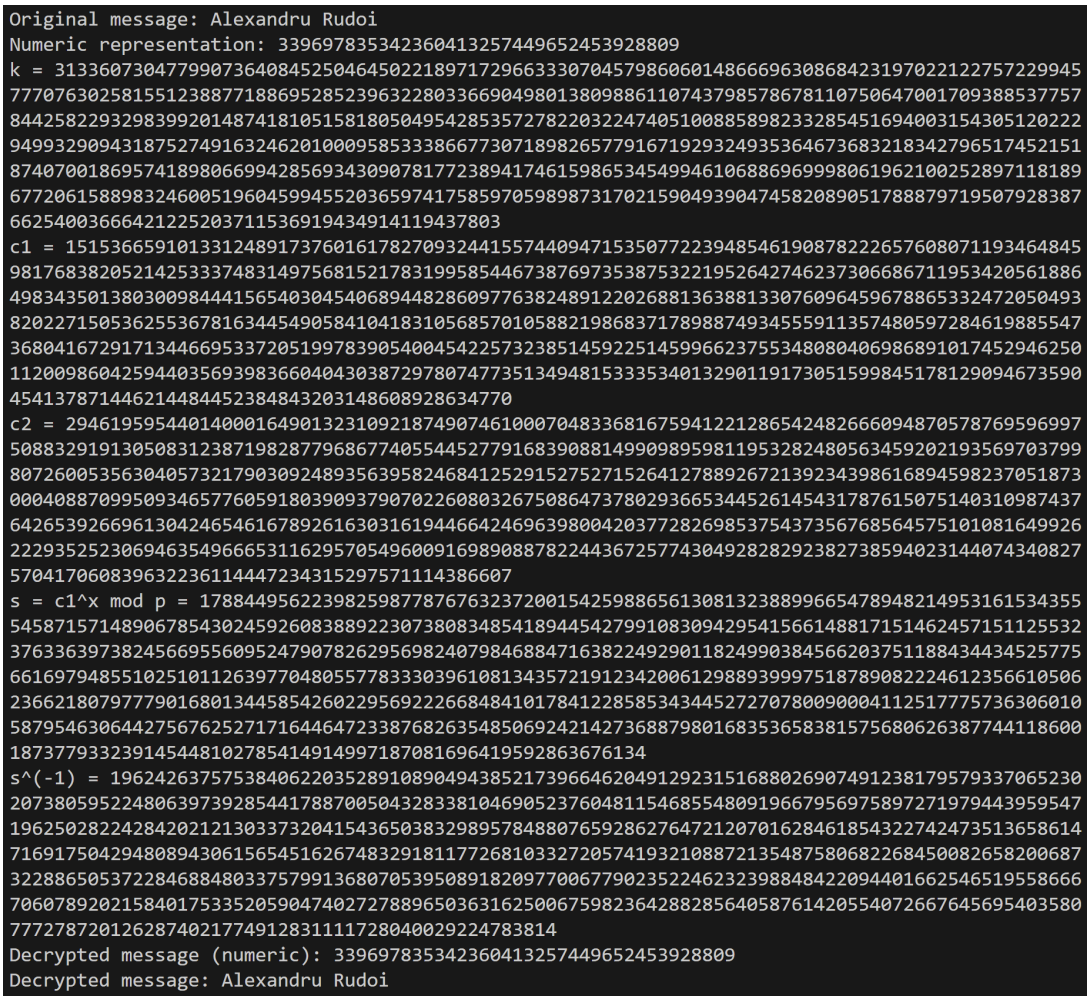

Figure 7: ElGamal Encryption and Decryption Process

the encryption of the message ‘"Nume Prenume"‘ using the ElGamal public key. The message is split into two ciphertext components, c_1 and c_2. These components are then decrypted back into the original message using the private key, successfully restoring the plaintext.

Diffie-Hellman Key Exchange Results

- **Screenshot 5: Diffie-Hellman Key Exchange Process** This screenshot illustrates the **Diffie-Hellman key exchange** between Alice and Bob. Both parties select private keys, compute their corresponding public values, exchange them, and

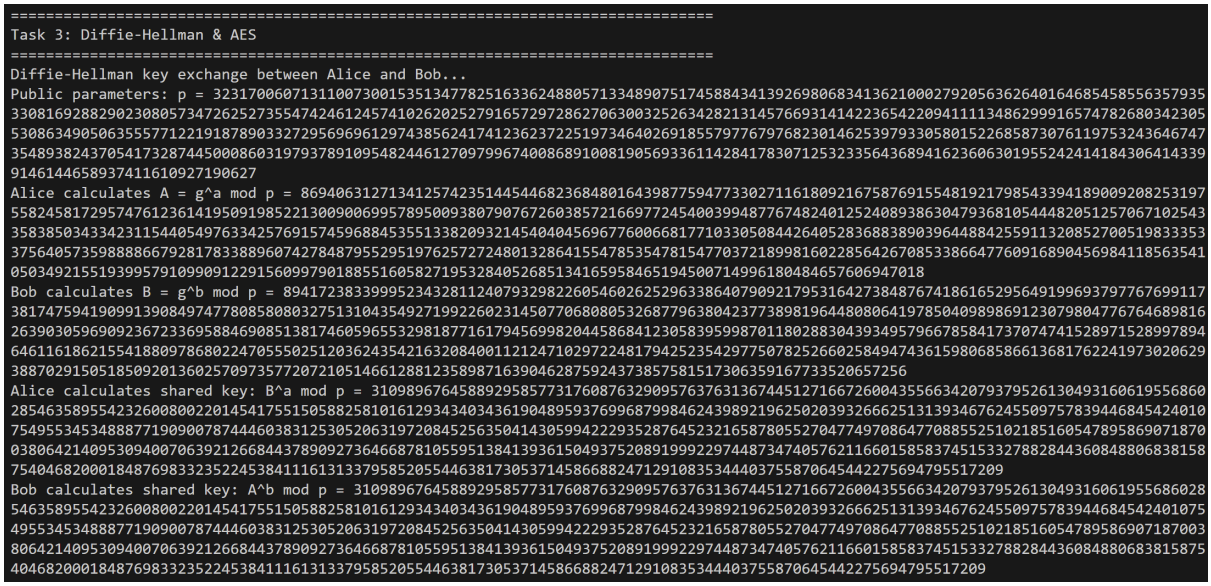

Figure 8: Diffie-Hellman Key Exchange and Shared Secret Generation

then compute the shared secret key. The shared secret is derived independently by both Alice and Bob, ensuring they end up with the same key.

- **Screenshot 6: AES Key Derivation and Encryption** The final screenshot

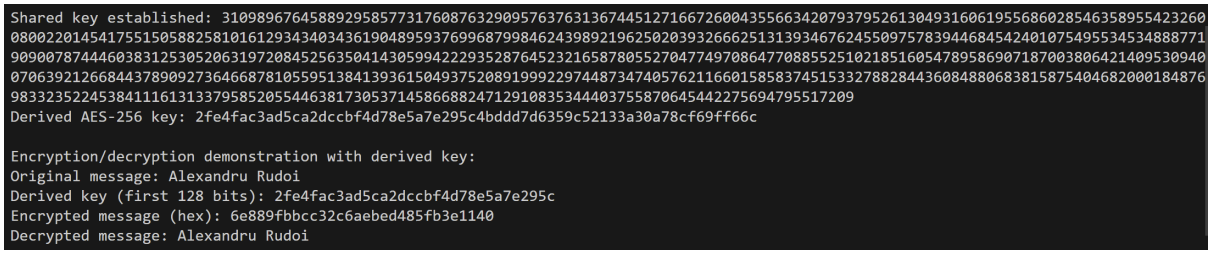

Figure 9: AES Key Derivation from Diffie-Hellman Shared Secret

demonstrates the **AES-256 key derivation** from the shared secret established by Diffie-Hellman. The shared key is hashed using SHA-256 to generate the final AES key, which can then be used for symmetric encryption in secure communication.

Conclusion

In this laboratory work, I implemented and tested three core cryptographic algorithms—RSA, ElGamal, and Diffie-Hellman—demonstrating key principles of public-key cryptography.

The RSA algorithm, based on large prime factorization, securely handles both encryption and digital signatures through key pairs. The implementation reaffirmed the importance of prime numbers and modular arithmetic in ensuring the security of cryptographic systems.

The ElGamal encryption system, which relies on the discrete logarithm problem, provided another layer of asymmetric cryptography. By using random values for key generation and encryption, it effectively ensures message security and confidentiality.

Finally, the Diffie-Hellman key exchange allowed us to securely establish a shared key over an insecure channel, which was then used to derive an AES-256 key for secure communication. This task highlighted the significance of modular exponentiation and secure key exchange protocols.

Through the successful implementation of these algorithms, I gained a deeper understanding of how public-key cryptography works to secure communications in real-world applications. The results underscore the importance of selecting secure parameters and random values in cryptographic systems to defend against potential attacks.

Git repository: `https://github.com/AlexandruRudoi/CS_Labs/tree/main/Lab_5`

Bibliography

1. Course materials: *Cryptography and Security*, UTM–FCIM, 2025.

2. ElGamal, T. (1985). *A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*. IEEE Transactions on Information Theory, 31(4), 469-472.
   This paper presents the ElGamal encryption system, which is based on the difficulty of computing discrete logarithms in a finite field.

3. Schneier, B. (2007). *Cryptography Engineering: Design Principles and Practical Applications*. Wiley.
   This book covers the implementation of various cryptographic algorithms, including RSA, ElGamal, Diffie-Hellman, and AES. It provides in-depth discussions on the design principles behind modern cryptography.

4. Stinson, D. R. (2006). *Cryptography: Theory and Practice* (3rd ed.). CRC Press.
   A comprehensive textbook covering the theoretical background of cryptographic algorithms and their practical applications, including RSA, ElGamal, Diffie-Hellman, and AES.

5. NIST (National Institute of Standards and Technology). (2015). *Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (SP 800-56A Rev. 3)*. NIST Special Publication 800-56A.
   This NIST document outlines recommended practices for the implementation of key exchange schemes like Diffie-Hellman.