Lab Manual – 5: Programming Symmetric & Asymmetric Cryptography

Course: CSE-478: Introduction to Computer Security

Lab Title: Programming Symmetric & Asymmetric Cryptography

Lab No: 05

Objectives

1. Understand the fundamental concepts of **symmetric** and **asymmetric** encryption techniques.
2. Implement **AES (symmetric)** and **RSA (asymmetric)** cryptography using Python.
3. Learn key generation, message encryption, and decryption processes in both approaches.
4. Compare the performance and security aspects of the two methods.

Required Tools

- **Python 3.x**
- **Libraries:** cryptography, pycryptodome
- **Operating System:** Ubuntu / Windows / macOS
- **Editor:** VS Code

Installation Commands:

pip install cryptography pycryptodome

Theoretical Background

1. Symmetric Encryption

Symmetric cryptography uses the **same secret key** for both encryption and decryption.
It is efficient and suitable for encrypting large amounts of data.
However, secure key distribution between sender and receiver is a challenge.

Example Algorithms: AES, DES, 3DES, Blowfish.

AES (Advanced Encryption Standard)

- Operates on fixed blocks of **128 bits**.
- Supports key sizes of **128, 192, or 256 bits**.
- Performs multiple rounds of substitution and permutation to achieve confusion and diffusion.

Advantages:

- Fast and efficient.
- Ideal for bulk data encryption.

Disadvantages:

- Requires secure key sharing.

---

2. Asymmetric Encryption

Asymmetric cryptography uses a **pair of keys** — one **public** and one **private**.

- The **public key** encrypts the data.
- The **private key** decrypts it.

This approach solves the key exchange problem in symmetric systems but is computationally slower.

RSA (Rivest–Shamir–Adleman)

- Based on the mathematical difficulty of factoring large prime numbers.
- Uses key pair:
  - Public Key → (n, e)
  - Private Key → (n, d)

Advantages:

- Secure key distribution.
- Enables digital signatures.

Disadvantages:

- Slower than symmetric encryption.

- Not efficient for large data encryption.

Implementation

Part A: Symmetric Encryption using AES (via Fernet)

Program: aes_encryption.py

```
from cryptography.fernet import Fernet


# Generate secret key

key = Fernet.generate_key()

cipher = Fernet(key)

print("Generated Key:", key.decode())


# Encrypt message

message = "Hello, this is Symmetric Encryption using AES!".encode()

encrypted = cipher.encrypt(message)

print("Encrypted Message:", encrypted.decode())


# Decrypt message

decrypted = cipher.decrypt(encrypted)

print("Decrypted Message:", decrypted.decode())
```

Sample Output:

Generated Key: V1YjLr2O...==

Encrypted Message: gAAAAABl...

Decrypted Message: Hello, this is Symmetric Encryption using AES!

Explanation:

1. A random AES key is generated using Fernet.

2. The plaintext message is encrypted and stored as ciphertext.
3. Using the same key, the message is decrypted successfully.

Part B: Asymmetric Encryption using RSA

Program: rsa_encryption.py

```
from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1_OAEP


# Generate RSA key pair

key = RSA.generate(2048)

private_key = key.export_key()

public_key = key.publickey().export_key()


with open("private.pem", "wb") as f:

   f.write(private_key)

with open("public.pem", "wb") as f:

   f.write(public_key)


# Load keys

pub_key = RSA.import_key(open("public.pem").read())

priv_key = RSA.import_key(open("private.pem").read())


# Encrypt message

cipher = PKCS1_OAEP.new(pub_key)

message = b"Hello, this is Asymmetric Encryption using RSA!"

ciphertext = cipher.encrypt(message)
```

```
print("Encrypted:", ciphertext)


# Decrypt message
decipher = PKCS1_OAEP.new(priv_key)
plaintext = decipher.decrypt(ciphertext)
print("Decrypted:", plaintext.decode())
```

Sample Output:

```
Encrypted: b'\x9c\x12\xae\x98...'
Decrypted: Hello, this is Asymmetric Encryption using RSA!
```

Explanation:

1. RSA key pair is generated and stored in .pem files.
2. The message is encrypted using the **public key**.
3. Only the **private key** can decrypt the ciphertext successfully.

Comparison between Symmetric and Asymmetric Encryption

Feature	**Symmetric (AES)**	**Asymmetric (RSA)**
Key Type	Single key for encryption/decryption	Public and private key pair
Speed	Fast	Relatively slow
Security	Key must remain secret	Public key can be shared safely
Use Cases	File encryption, data storage	Key exchange, digital signatures
Efficiency	High for large data	Lower for large data

Discussion

Both encryption methods serve distinct purposes:

- **AES** is ideal for encrypting large datasets where performance matters.

- **RSA** is best suited for secure key exchange or small data where confidentiality is critical.

A hybrid system (combining AES + RSA) is often used in real-world applications — RSA to securely transmit the AES key, and AES to encrypt actual data.

Conclusion

This lab demonstrated the implementation of both **symmetric** and **asymmetric cryptographic systems** using Python.
Through AES and RSA examples, we learned how encryption and decryption work at a programmatic level.
This exercise reinforced the understanding of key management, confidentiality, and data protection — the foundation of modern secure communication systems.