# Lab 4: Programming Symmetric & Asymmetric Crypto

**Student Name:** *Md Robiul Islam Robin*
**Student ID:** *2020831043*

---

## Step 1: Importing Libraries

I imported necessary modules from the **PyCryptodome** library for implementing cryptographic operations.
The `time` module is used for execution time measurement, and `matplotlib` is used for performance graph plotting.

**Message:**

```
"This is a test message for cryptography"
```

The program begins by importing:

- AES (for symmetric encryption)
- RSA & PKCS1_OAEP (for asymmetric encryption)
- pkcs1_15 (for digital signature)
- SHA256 (for hashing)
- get_random_bytes (for secure random key generation)
- time, os, matplotlib

Padding and unpadding functions are also implemented to support AES ECB mode.

---

## Step 2: Key Generation

The function `generate_keys()` performs:

### AES Key Generation

- AES-128 → 16 bytes
- AES-256 → 32 bytes

Keys are saved into:

```
aes_128.key
aes_256.key
```

### RSA Key Pair Generation

- 2048-bit RSA key
- Private key saved as: `rsa_private.pem`
- Public key saved as: `rsa_public.pem`

Keys are generated only once (if files don't exist).

---

## Step 3: AES ECB Mode

Two functions are implemented:

### AES ECB Encryption

- Reads AES key from file
- Pads message manually

Encrypts using:

```
AES.new(key, AES.MODE_ECB)
```

-

Saves ciphertext into:

```
encrypted_ecb_<keysize>.bin
```

- 

### AES ECB Decryption

- Reads ciphertext
- Decrypts with the stored key
- Unpads the decrypted text
- Displays output message

ECB encrypts data in independent fixed-size blocks.

---

## Step 4: AES CFB Mode

AES CFB mode is implemented with:

### Encryption

- Reads AES key
- Generates a random 16-byte IV

Uses:

```
AES.new(key, AES.MODE_CFB, iv=iv, segment_size=128)
```

- 

Saves `IV + ciphertext` into:

```
encrypted_cfb_<keysize>.bin
```

- 

### Decryption

- Extracts the first 16 bytes as IV
- Uses the remaining bytes as ciphertext
- Decrypts data correctly

CFB works like a stream cipher and is more secure than ECB.

---

## Step 5: RSA Encryption/Decryption

### RSA Encryption

- Uses **public key**
- Applies **PKCS1_OAEP** padding
- Encrypts the message as bytes

Saves ciphertext to:

`encrypted_rsa.bin`

### RSA Decryption

- Loads **private key**
- Applies OAEP padding
- Decrypts and prints output

RSA is slower and suitable only for small messages.

---

```
RSA Encryption
Encrypted saved to: encrypted_rsa.bin
Time: 0.009794 seconds

RSA Decryption
Decrypted: This is a test message for cryptography
Time: 0.009257 seconds
```

## Step 6: RSA Signature

### Signing Process

- Computes SHA-256 hash of the message

Signs the hash using:

`pkcs1_15.new(private_key).sign(h)`

- Saves:
    - `message.txt`
    - `signature.sig`

### Verification

- Loads stored message + signature
- Computes hash
- Verifies using public key

Prints:

```
RSA Signature
Message saved to: message.txt
Signature saved to: signature.sig
Time: 0.010256 seconds

RSA Signature Verification
Signature is VALID
```

```
Signature is VALID
```

**Signature.sig**

N?��G'�V�{#��Q&�t��v�A�ïU��)6O/]���Td�□�|�gdi(�w�#qv�MB �����MQ��Q�G�����Kj�X��b���□a/��0�K�|���x=�i��uz pN_[@I:V���S�7

��8.&�����>J

*�^�Z�W?�

2$21H�8�m�1����VY��������=�\QrqQ4��UXj�=��1����t�Y �`�EJ��Qj�vDf�Lf��d���y?�K

This ensures authenticity and integrity.

---

# Step 7: SHA-256 Hashing

A simple hash function:

- Takes message

- Computes SHA-256 digest
- Prints hash value and execution time

```
SHA-256 Hashing
Message: This is a test message for cryptography
SHA-256: e0b586d890ab58e7d127ef86897b0d837aa07a87168c993b70db6084a0f974ee
Time: 0.014946 seconds
```

Hashing is essential for integrity checking.

---

## Step 8: Performance Analysis

The performance test compares AES and RSA for different key sizes.

### AES Performance

Key sizes tested:

- 128 bits
- 192 bits
- 256 bits

Results:

- All executions completed in **under 0.003 seconds**
- Key size has very small impact on AES execution time
- AES is very efficient

### RSA Performance

Key sizes tested:

- 1024 bits

- 1536 bits
- 2048 bits
- 3072 bits

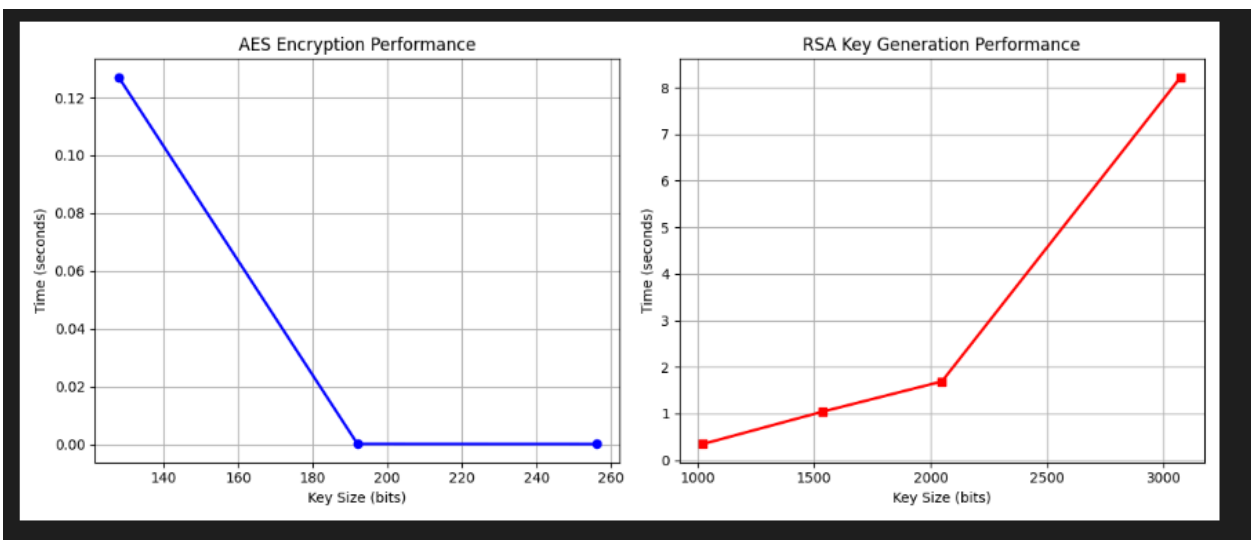
Observations:

- RSA-1024 is fastest
- RSA-3072 can take several seconds
- Execution time increases dramatically with key size
- RSA is far slower than AES

## Graphs Generated

Two graphs are saved as:

`performance_graph.png`




Showing:

1. AES encryption time vs key size

2. RSA key generation time vs key size

---

## Observations

- AES is extremely fast, secure, and scalable
- CFB provides better security compared to ECB
- RSA is slow and only suitable for encrypting small data or signing
- Hashing provides fixed-size unique fingerprints
- Digital signatures prove message authenticity

---

## Step 9: Menu System

A command-line menu allows the user to choose:


```
1. AES-128 ECB
2. AES-256 ECB
3. AES-128 CFB
```

```
4. AES-256 CFB
5. RSA Encryption/Decryption
6. RSA Signature
7. SHA-256 Hash
8. Performance Test
0. Exit
```

Each option triggers the corresponding cryptographic function.

The program loops until the user chooses **Exit**.

---

# Reference


1. https://www.laurentluce.com/posts/python-and-cryptography-with-pycrypto/
2. https://pycryptodome.readthedocs.io/en/latest/
3. https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html
4. https://pycryptodome.readthedocs.io/en/latest/src/public_key/rsa.html
5. https://realpython.com/read-write-files-python/
6. https://matplotlib.org/stable/tutorials/pyplot.html