Lab 4: Programming Symmetric & Asymmetric Cryptography

The main goals of this lab are to:
- Implement symmetric (AES) and asymmetric (RSA) cryptographic operations using Python.
- Implement RSA digital signature generation and verification.
- Generate SHA-256 hashes for data integrity verification.
- Measure and analyze execution times of AES and RSA algorithms for different key lengths.
- Compare performance and study how key length affects encryption time.

Tools and Environments setup

Programming Language: Python 3

Libraries: `pycryptodome`, `matplotlib`, `hashlib`

Platform: Windows

Editor: VS Code

Libaries:pip install pycryptodome matplotlib

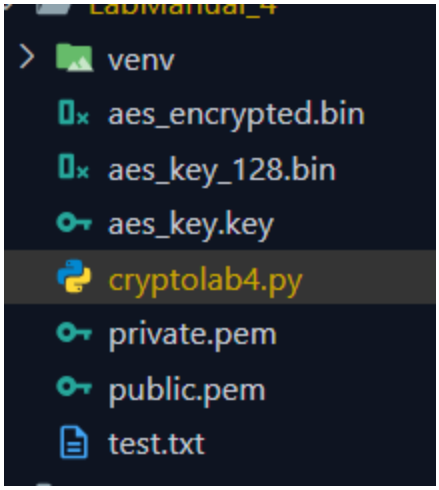
1. AES Encryption/Decryption

AES keys of **128-bit** and **256-bit** are generated and stored in `aes_key_128.bin` and `aes_key_256.bin`.

Supports **ECB** and **CFB** modes.

Encryption reads a file (`test.txt`) and writes the encrypted content to `aes_encrypted.bin`.

Decryption reads the encrypted file and AES key to retrieve the original text.



Implemented function for AES Encryption Decryption

```
def aes_encrypt():
    bits = int(input("Enter AES key size (128 or 256): "))
    mode = input("Enter mode (ECB or CFB): ").upper()
    input_file = input("Enter the file name to encrypt (e.g., test.txt): ")

    try:
        with open(input_file, "r") as f:
            plaintext = f.read()
    except FileNotFoundError:
        print(" File not found. Make sure it's in the same folder as this progra
        return

    key = generate_aes_key(bits)

    if mode == "ECB":
        cipher = AES.new(key, AES.MODE_ECB)
    elif mode == "CFB":
        iv = get_random_bytes(16)
        cipher = AES.new(key, AES.MODE_CFB, iv)
    else:
        print(" Invalid mode. Choose ECB or CFB.")
        return

    start_time = time.time()

    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))
    with open("aes_encrypted.bin", "wb") as f:
        if mode == "CFB":
            f.write(iv)
        f.write(ciphertext)

    end_time = time.time()

    print(f" AES-{bits} {mode} encryption done.")
    print(f" Execution time: {end_time - start_time:.6f} seconds\n")
```

```
def aes_decrypt():
    bits = int(input("Enter AES key size (128 or 256): "))
    mode = input("Enter mode (ECB or CFB): ").upper()
    key_file = f"aes_key_{bits}.bin"

    try:
        with open(key_file, "rb") as f:
            key = f.read()
        with open("aes_encrypted.bin", "rb") as f:
            data = f.read()
    except FileNotFoundError:
        print(" Required file(s) not found.")
        return

    start_time = time.time()

    if mode == "CFB":
        iv = data[:16]
        ciphertext = data[16:]
        cipher = AES.new(key, AES.MODE_CFB, iv)
        plaintext = cipher.decrypt(ciphertext).decode()
    else:
        cipher = AES.new(key, AES.MODE_ECB)
        plaintext = unpad(cipher.decrypt(data), AES.block_size).decode()

    end_time = time.time()

    print("\n Decrypted text:")
    print(plaintext)
    print(f"\n Execution time: {end_time - start_time:.6f} seconds\n")
```

Implemented Function For RSA Encryption/Decryption,Signature,Verify

```
def generate_rsa_keys():
    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)

    print(" RSA keys generated: private.pem & public.pem\n")
```

```python
def rsa_encrypt():
    input_file = input("Enter file name to encrypt (e.g., test.txt): ")

    try:
        with open(input_file, "r") as f:
            plaintext = f.read().encode()
        with open("public.pem", "rb") as f:
            public_key = RSA.import_key(f.read())
    except FileNotFoundError:
        print(" File not found.")
        return

    cipher = PKCS1_OAEP.new(public_key)

    if len(plaintext) > 190:  # RSA can't handle large data directly
        print(" File too large for direct RSA encryption. Use smaller text file.")
        return

    start_time = time.time()
    ciphertext = cipher.encrypt(plaintext)
    end_time = time.time()

    with open("rsa_encrypted.bin", "wb") as f:
        f.write(ciphertext)

    print(f" RSA encryption done. ⏱ Time: {end_time - start_time:.6f} seconds\n")
```

```python
def rsa_decrypt():
    try:
        with open("private.pem", "rb") as f:
            private_key = RSA.import_key(f.read())
        with open("rsa_encrypted.bin", "rb") as f:
            ciphertext = f.read()
    except FileNotFoundError:
        print(" Missing RSA key or encrypted file.")
        return

    cipher = PKCS1_OAEP.new(private_key)
    start_time = time.time()
    plaintext = cipher.decrypt(ciphertext)
    end_time = time.time()

    print(f"\n Decrypted text:\n{plaintext.decode()}")
    print(f"Execution time: {end_time - start_time:.6f} seconds\n")
```

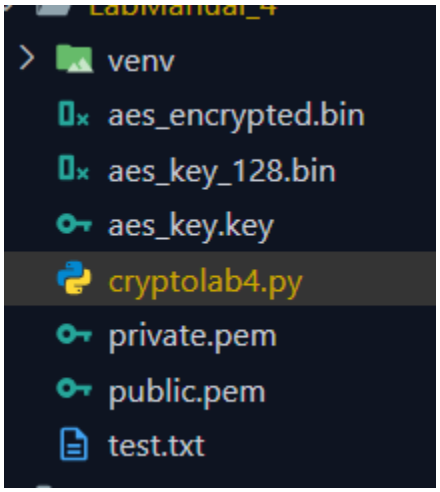
```
def rsa_signature():
    input_file = input("Enter file to sign (e.g., test.txt): ")
    try:
        with open(input_file, "rb") as f:
            message = f.read()
        with open("private.pem", "rb") as f:
            private_key = RSA.import_key(f.read())
    except FileNotFoundError:
        print(" File not found.")
        return

    h = SHA256.new(message)
    signature = pkcs1_15.new(private_key).sign(h)

    with open("signature.bin", "wb") as f:
        f.write(signature)

    print(" RSA signature created and saved to signature.bin\n")
```

```
def verify_signature():
    input_file = input("Enter file to verify (e.g., test.txt): ")
    try:
        with open(input_file, "rb") as f:
            message = f.read()
        with open("signature.bin", "rb") as f:
            signature = f.read()
        with open("public.pem", "rb") as f:
            public_key = RSA.import_key(f.read())
    except FileNotFoundError:
        print(" File(s) missing for verification.")
        return

    h = SHA256.new(message)
    try:
        pkcs1_15.new(public_key).verify(h, signature)
        print(" Signature is valid.\n")
    except (ValueError, TypeError):
        print(" Signature is invalid.\n")
```

RSA key pair (2048-bit) is generated and saved as `private.pem` and `public.pem`.

RSA encrypts small files (or text ≤ 190 bytes) and stores as `rsa_encrypted.bin`.

RSA decrypts using the private key.



SHA-256 Hash

```python
def sha256_hash():
    input_file = input("Enter file to hash (e.g., test.txt): ")
    try:
        with open(input_file, "rb") as f:
            data = f.read()
    except FileNotFoundError:
        print(" File not found.")
        return

    hash_value = hashlib.sha256(data).hexdigest()
    print(f" SHA-256 hash:\n{hash_value}\n")
```

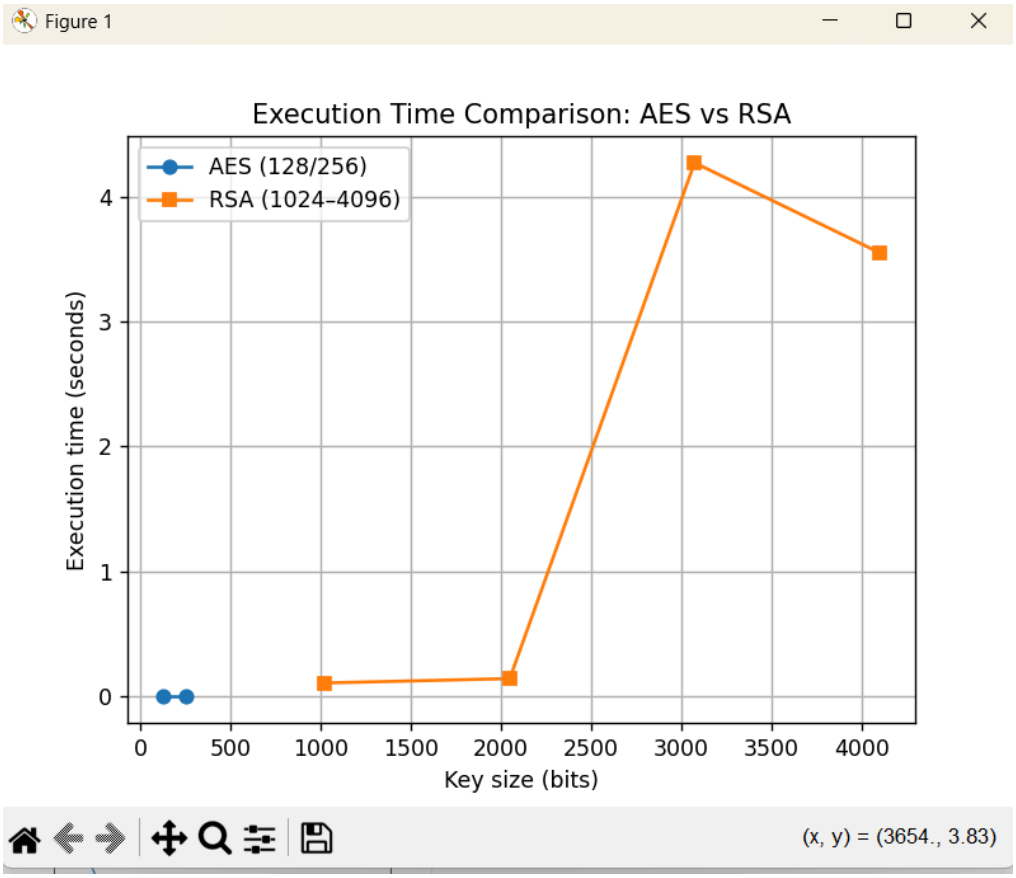
Execution Time Measurement

```
es_key_sizes = [128, 256]
sa_key_sizes = [1024, 2048, 3072, 4096]

es_times = []
sa_times = []

rint(" Measuring AES and RSA performance...")

# --- AES timing ---
or bits in aes_key_sizes:
    key = get_random_bytes(bits // 8)
    cipher = AES.new(key, AES.MODE_ECB)
    data = b"A" * 64
    start = time.time()
    cipher.encrypt(pad(data, AES.block_size))
    end = time.time()
    aes_times.append(end - start)

# --- RSA timing ---
or bits in rsa_key_sizes:
    start = time.time()
    key = RSA.generate(bits)
    end = time.time()
    rsa_times.append(end - start)
```

Measures AES (128 & 256-bit) encryption time and RSA key generation time for valid sizes.

Plots a graph comparing execution time vs key size.

RSA keys smaller than 1024 bits are skipped



Observations

1. AES is much faster than RSA for encryption/decryption.
2. CFB mode is slightly slower than ECB due to IV handling.
3. RSA key generation time increases significantly with key size.
4. SHA-256 hash computation is very fast even for large files