Lab 4 Report: Programming Symmetric & Asymmetric Crypto

Program Overview

The program (`crypto_program.py`) implements the following functionalities:

1. **AES Encryption/Decryption** (5 marks) - Supports 128-bit and 256-bit keys with ECB and CFB modes
2. **RSA Encryption/Decryption** (4 marks) - Asymmetric encryption and decryption
3. **RSA Signature** (4 marks) - Digital signature generation and verification
4. **SHA-256 Hashing** (3 marks) - Message digest generation
5. **Execution Time Measurement** (4 marks) - Performance analysis for different key sizes

---

Implementation Details

Technology Stack

- **Programming Language**: Python 3
- **Cryptography Library**: `cryptography` (hazmat primitives)
- **Visualization**: `matplotlib` for timing graphs
- **Key Storage**: Base64-encoded files for AES keys, PEM format for RSA keys

Libraries Used

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, mo
des
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.backends import default_backend
```

```
from cryptography.exceptions import InvalidSignature
import matplotlib.pyplot as plt
```

Resources and References

The implementation was based on the following resources:

1. **Python Cryptography Library Documentation**: https://cryptography.io/en/latest/
    - Used for understanding the hazmat primitives API
    - Reference for AES, RSA, and hashing implementations
2. **Cryptography Library GitHub**: https://github.com/pyca/cryptography
    - Reference for best practices and examples
3. **Python Official Documentation**: https://docs.python.org/3/
    - Used for file I/O operations and base64 encoding

Task 1: AES Encryption/Decryption (5 Marks)

Implementation

The program supports AES encryption and decryption with:

- **Key Sizes**: 128 bits and 256 bits
- **Modes**: ECB (Electronic Codebook) and CFB (Cipher Feedback)

Key Features

1. **Key Generation**: Keys are generated using `os.urandom()` for cryptographically secure random bytes
2. **Key Storage**: Keys are saved in Base64-encoded format with filenames like `aes_128_ECB.key` or `aes_256_CFB.key`
3. **IV Handling**: CFB mode uses a random 16-byte IV that is prepended to the encrypted data
4. **Padding**: Data is padded to 16-byte blocks for ECB mode (AES block size requirement)

Code Structure

Key Generation:

```
def generate_aes_key(key_size=128):
    if key_size == 128:
        return os.urandom(16)
    elif key_size == 256:
        return os.urandom(32)
```

Encryption:

```
def aes_encrypt(data, key, mode='ECB'):
    if mode == 'CFB':
        iv = os.urandom(16)
        cipher = Cipher(algorithms.AES(key), modes.CFB(iv), backend=default_backend())
    else:
        cipher = Cipher(algorithms.AES(key), modes.ECB(), backend=default_backend())
    # ... encryption logic
```

Usage Example

1. **Encrypt a file:**
    - Select option 1 from main menu
    - Choose "Encrypt file"
    - Enter input file: `sample.txt`
    - Enter output file: `aes_128_ecb.bin`
    - Enter key size: `128`
    - Enter mode: `ECB`
2. **Decrypt a file:**
    - Select option 1 from main menu
    - Choose "Decrypt file"
    - Enter encrypted file: `aes_128_ecb.bin`
    - Enter output file: `decrypted.txt`

- Enter key size: `128`
- Enter mode: `ECB`
- Decrypted content is displayed on console

Test Results

Test File: `sample.txt` (27 bytes)

```
this is a simple plain text
```

AES-128-ECB Encryption:

- Input: `aes_128_ecb_plain.txt` (27 bytes)
- Output: `aes_128_ecb.bin` (32 bytes - padded to block size)
- Key stored in: `aes_128_ECB.key`
- Decryption successfully recovered original text

AES-128-CFB Encryption:

- Key stored in: `aes_128_CFB.key`
- IV is prepended to encrypted data (16 bytes + encrypted data)

AES-256-ECB/CFB:

- Keys stored in: `aes_256_ECB.key` and `aes_256_CFB.key`
- 32-byte keys for 256-bit encryption

Task 2: RSA Encryption/Decryption (4 Marks)

Implementation

RSA encryption and decryption using OAEP padding with SHA-256.

Key Features

1. **Key Generation**: RSA key pairs are generated using 2048-bit keys by default
2. **Key Storage**: Keys are stored in PEM format:
    - Private key: `rsa_private.pem`

- Public key: `rsa_public.pem`

3. **Padding**: OAEP (Optimal Asymmetric Encryption Padding) with SHA-256
4. **Key Size Limitation**: RSA can only encrypt small amounts of data (limited by key size)

Code Structure

Key Generation:

```
def generate_rsa_keys(key_size=2048):
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=key_size,
        backend=default_backend()
    )
    public_key = private_key.public_key()
    return private_key, public_key
```

Encryption:

```
def rsa_encrypt(data, public_key):
    encrypted = public_key.encrypt(
        data,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None        )
    )
    return encrypted
```

Usage Example

1. **Encrypt a file:**
    - Select option 2 from main menu
    - Choose "Encrypt file"
    - Enter input file: `rsa_plain.txt`
    - Enter output file: `rsa_encrypted.bin`

- Keys are automatically loaded from `rsa_private.pem` and `rsa_public.pem`

2. **Decrypt a file:**
   - Select option 2 from main menu
   - Choose “Decrypt file”
   - Enter encrypted file: `rsa_encrypted.bin`
   - Enter output file: `rsa_decrypted.txt`
   - Decrypted content is displayed on console

Test Results

Test File: `rsa_plain.txt` (27 bytes)

```
this is a simple plain text
```

RSA Encryption:

- Input: `rsa_plain.txt` (27 bytes)
- Output: `rsa_encrypted.bin` (256 bytes - RSA 2048-bit output size)
- Keys: `rsa_private.pem` and `rsa_public.pem`
- Decryption successfully recovered original text

Task 3: RSA Signature (4 Marks)

Implementation

RSA digital signature generation and verification using PSS padding with SHA-256.

Key Features

1. **Signature Generation**: Uses private key to sign file content
2. **Signature Storage**: Signatures are stored in binary files (e.g., `sign.sig`)
3. **Verification**: Uses public key to verify signature against original file
4. **Padding**: PSS (Probabilistic Signature Scheme) with SHA-256

Code Structure

Signature Generation:

```
def rsa_sign_file(input_file, signature_file):
    private_key, public_key = load_rsa_keys()
    with open(input_file, 'rb') as f:
        data = f.read()
    signature = private_key.sign(
        data,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    with open(signature_file, 'wb') as f:
        f.write(signature)
```

Signature Verification:

```
def rsa_verify_signature(input_file, signature_file):
    public_key.verify(
        signature,
        data,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
```

Usage Example

1. **Generate signature:**
    - Select option 3 from main menu
    - Choose “Generate signature”
    - Enter file to sign: `sample.txt`
    - Enter signature file: `sign.sig`

- Signature is generated and saved

2. **Verify signature:**
    - Select option 3 from main menu
    - Choose “Verify signature”
    - Enter original file: `sample.txt`
    - Enter signature file: `sign.sig`
    - Program displays “Signature is VALID” or “Signature is INVALID”

Test Results

Signature Generation:

- Input file: `sample.txt`
- Signature file: `sign.sig` (256 bytes - RSA 2048-bit signature size)
- Signature successfully generated

Signature Verification:

- Original file: `sample.txt`
- Signature file: `sign.sig`
- Verification result: **VALID**

Task 4: SHA-256 Hashing (3 Marks)

Implementation

SHA-256 message digest generation for file integrity verification.

Key Features

1. **Hash Generation**: Computes SHA-256 hash of file content
2. **Output Format**: Hexadecimal representation displayed on console
3. **Fast Operation**: Efficient hashing for files of any size

Code Structure

```
def sha256_hash_file(input_file):
    with open(input_file, 'rb') as f:
        data = f.read()
    digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
    digest.update(data)
    hash_result = digest.finalize()
    print(f"SHA-256 Hash: {hash_result.hex()}")
```

Usage Example

1. **Generate hash:**
    - Select option 4 from main menu
    - Enter file to hash: `sample.txt`
    - SHA-256 hash is displayed on console in hexadecimal format

Test Results

Hash Generation:

- Input file: `sample.txt` (27 bytes)
- SHA-256 Hash: `[64-character hexadecimal string]`
- Hash is deterministic - same file always produces same hash

---

Task 5: Execution Time Measurement (4 Marks)

Implementation

The program measures execution time for cryptographic operations as a function of key size.

Key Features

1. **AES Timing**: Measures encryption/decryption time for 128-bit and 256-bit keys with ECB and CFB modes
2. **RSA Timing**: Measures encryption/decryption time for 1024-bit, 2048-bit, 3072-bit, and 4096-bit keys
3. **Graphical Visualization**: Generates plots showing timing results

4. **Multiple Measurements**: Tests each configuration multiple times for accuracy

Code Structure

AES Timing Measurement:

```
def measure_aes_timing():
    test_data = b"This is a test file for timing measurement." * 100  # ~4KB
key_sizes = [128, 256]
    modes = ['ECB', 'CFB']
    # Measure encryption and decryption times
```

RSA Timing Measurement:

```
def measure_rsa_timing():
    key_sizes = [1024, 2048, 3072, 4096]
    # Generate keys and measure encryption/decryption times
```

Graph Generation:

```
def plot_timing_results(aes_results, rsa_results):
    # Create bar chart for AES results    # Create line plot for RSA results    pl
t.savefig('timing_results.png')
```

Usage Example

1. **Measure timing:**
    - Select option 5 from main menu
    - Program automatically:
        - Measures AES timing for 128-bit and 256-bit keys (ECB and CFB)
        - Measures RSA timing for 1024, 2048, 3072, and 4096-bit keys
        - Displays results in console
        - Generates `timing_results.png` graph

Timing Results

AES Timing Results:

Key Size	Mode	Encrypt Time (s)	Decrypt Time (s)
128-bit	ECB	~0.0001	~0.0001
128-bit	CFB	~0.0001	~0.0001
256-bit	ECB	~0.0001	~0.0001
256-bit	CFB	~0.0001	~0.0001

Observations:

- AES encryption/decryption is very fast (microseconds)
- Key size (128 vs 256) has minimal impact on performance
- Mode (ECB vs CFB) has minimal impact on performance
- Both operations are symmetric in terms of time

RSA Timing Results:

Key Size	Encrypt Time (s)	Decrypt Time (s)
1024-bit	~0.001	~0.001
2048-bit	~0.002	~0.010
3072-bit	~0.003	~0.030
4096-bit	~0.005	~0.100

Observations:

- RSA decryption is significantly slower than encryption
- Execution time increases exponentially with key size
- 4096-bit keys are approximately 10x slower than 2048-bit keys for decryption
- Encryption time increases linearly, decryption time increases exponentially

Graph Analysis

The generated `timing_results.png` shows:

1. **AES Bar Chart**:
    - Minimal difference between key sizes and modes
    - Both encryption and decryption are equally fast
2. **RSA Line Plot**:

- Exponential growth in decryption time with key size
- Linear growth in encryption time with key size
- Clear demonstration of asymmetric cryptography's computational cost

Implications

1. **AES**: Suitable for bulk data encryption due to fast performance regardless of key size
2. **RSA**: Better suited for small data (like symmetric keys) due to computational cost
3. **Hybrid Approach**: Common practice is to use RSA to encrypt AES keys, then use AES for bulk data encryption
4. **Key Size Trade-off**: Larger RSA keys provide better security but significantly slower performance

---

Program Execution Guide

Prerequisites

1. **Python 3.x** installed on the system
2. **Required Python packages:**

```
pip install cryptography matplotlib
```

Running the Program

1. **Start the program:**

```
python crypto_program.py
```

2. **Main Menu Options:**

```
==================================================
         CRYPTOGRAPHY PROGRAM
==================================================
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
```

```
3. RSA Signature
4. SHA-256 Hashing
5. Measure Execution Time
6. Exit
==================================================
```

3. **Navigation:**
    - Enter the number corresponding to your desired operation
    - Follow the prompts for each operation
    - Each sub-menu has a "Back to main menu" option

Key Management

AES Keys:

- Automatically generated on first encryption operation
- Stored in files: `aes_[key_size]_[mode].key`
- Base64-encoded format
- Automatically loaded for decryption

RSA Keys:

- Automatically generated on first use if not present
- Stored in PEM format:
    - `rsa_private.pem` - Private key
    - `rsa_public.pem` - Public key
- Automatically loaded for all RSA operations

File Structure

After running the program, the following files may be created:

```
Lab4/
├── crypto_program.py        # Main program
├── sample.txt             # Test file
├── aes_128_ECB.key          # AES 128-bit ECB key
├── aes_128_CFB.key          # AES 128-bit CFB key
├── aes_256_ECB.key          # AES 256-bit ECB key
```

```
├── aes_256_CFB.key          # AES 256-bit CFB key
├── rsa_private.pem          # RSA private key
├── rsa_public.pem           # RSA public key
├── aes_128_ecb.bin          # AES encrypted file
├── rsa_encrypted.bin        # RSA encrypted file
├── sign.sig                 # RSA signature file
└── timing_results.png       # Timing graph
```

References

1. **Python Cryptography Library**: https://cryptography.io/en/latest/
    - Primary reference for cryptographic primitives implementation
    - Used for AES, RSA, and hashing functions
2. **Cryptography Library Documentation**: https://cryptography.io/en/latest/hazmat/primitives/
    - Reference for hazmat primitives API
    - Used for understanding cipher modes and padding schemes
3. **Python Official Documentation**: https://docs.python.org/3/
    - Reference for Python standard library functions
    - Used for file I/O, base64 encoding, and time measurement
4. **Matplotlib Documentation**: https://matplotlib.org/stable/contents.html
    - Reference for graph generation
    - Used for timing visualization
5. **Claude AI**: https://claude.ai/
    - Used for improving Command Line Interface (CLI)
    - Improve code quality and readability with comments and functions
    - Fasten debug process