Title:Programming Symmetric Asymmetric Crypto (Lab 4)

Omar faruk
Reg No: 2019831055

June 24, 2024

1 Introduction

This project provides hands-on experience with symmetric and asymmetric cryptography using Python. We implemented AES encryption/decryption in ECB and CFB modes with 128-bit and 256-bit keys, RSA encryption/decryption and digital signatures, and SHA-256 hashing. The program offers a command-line interface similar to OpenSSL and measures execution times for these operations. This report covers the implementation, usage, and performance analysis of the cryptographic functions.

2 Implementation Details

The cryptographic tool was implemented in Python, utilizing libraries such as `pycryptodome` for AES and RSA operations and `hashlib` for SHA-256 hashing. The program supports the following functionalities:

2.1 AES Encryption/Decryption

- **Key Lengths**: 128-bit and 256-bit.
- **Modes**: ECB (Electronic Codebook) and CFB (Cipher Feedback).
- **Encryption**: Pads plaintext for ECB mode, generates ciphertext.
- **Decryption**: Unpads plaintext for ECB mode, retrieves original data.

2.2 RSA Encryption/Decryption and Signature

- **Key Generation**: Generates RSA key pairs.
- **Encryption/Decryption**: Uses PKCS1_OAEP for secure encryption and decryption.

- **Signature**: Signs data using private key and verifies signatures using public key and SHA-256 hashing.

2.3 SHA-256 Hashing

- **Hashing**: Computes SHA-256 hash of input data.

2.4 Execution Time Measurement

- **Measurement**: Utilizes Python's `time` module to measure the execution time for each cryptographic operation.

- **Analysis**: Collects execution times for various key sizes and plots the results.

2.5 Command Line Interface

- **User Interaction**: The program offers an interactive menu for users to choose and perform different cryptographic operations.

- **File Handling**: Encrypts and decrypts files, generates and verifies signatures, and outputs results to the console.

2.6 Example Code Snippets

2.6.1 AES Encryption

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad

def aes_encrypt(data, key, mode):
    cipher = AES.new(key, AES.MODE_ECB) if mode == 'ECB' else AES.new(key, AES.MODE_CFB)
    return cipher.encrypt(pad(data, AES.block_size)) if mode == 'ECB' else cipher.encrypt(
    data)

def aes_decrypt(data, key, mode):
    cipher = AES.new(key, AES.MODE_ECB) if mode == 'ECB' else AES.new(key, AES.MODE_CFB)
    return unpad(cipher.decrypt(data), AES.block_size) if mode == 'ECB' else cipher.decrypt(
    data)
```

2.6.2 RSA Encryption

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP

def rsa_encrypt(public_key, data):
    rsa_key = RSA.import_key(public_key)
    cipher = PKCS1_OAEP.new(rsa_key)
    return cipher.encrypt(data)

def rsa_decrypt(private_key, data):
    rsa_key = RSA.import_key(private_key)
    cipher = PKCS1_OAEP.new(rsa_key)
    return cipher.decrypt(data)
```

2.6.3 SHA-256 Hashing

```python
import hashlib

def sha256_hash(data):
    return hashlib.sha256(data).hexdigest()
```

2.6.4 Execution Time Measurement

```python
import time

def measure_execution_time(func, *args):
    start_time = time.time()
    result = func(*args)
    end_time = time.time()
    elapsed_time = end_time - start_time
    return result, elapsed_time
```

2.7 File Structure

- **crypto_tool.py**: Main script containing the implementation and command-line interface.
- **keys/**: Directory to store generated keys.
- **results/**: Directory to store encrypted files, signatures, and other output data.

This structure ensures that keys and output data are organized and easily accessible. The code snippets provided are part of the larger implementation that includes user input handling and file operations, ensuring a comprehensive and functional cryptographic tool.

3 Usage Instructions

This section provides detailed instructions on how to run the cryptographic tool and use its functionalities.

3.1 Prerequisites

Ensure you have Python installed on your system. You can download Python from `https://www.python.org/`. Additionally, install the required libraries using the following command:

```
pip install pycryptodome
```

3.2 Running the Program

1. Save the provided code in a file named `lab4.py`.
2. Open a terminal or command prompt and navigate to the directory where `lab4.py` is saved.
3. Run the program by executing the following command:

   ```
   python lab4.py
   ```

3.3 Using the Functionalities

Upon running the program, you will be presented with a menu of options:

```
Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
```

```
Information and Network Security/lab/lab_4 on  main via 🐍 v3.9.4
❯ python lab4.py

Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
Choose an option:
```

You can choose an option by entering the corresponding number.

3.3.1 AES Encryption/Decryption

1. Choose option 1 for AES Encryption/Decryption.
2. Enter the data you want to encrypt or decrypt.
3. Specify the key length (128 or 256 bits).
4. Choose the mode (ECB or CFB).
5. Choose the operation (encrypt or decrypt).
6. The program will generate a key, perform the operation, and display the result along with the time taken.

```
Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
Choose an option: 1
Enter data to encrypt/decrypt: my name is faruk
Enter key length (128 or 256): 256
Enter mode (ECB or CFB): ECB
Choose operation (encrypt or decrypt): encrypt
Result: b'\xcc\x01G\xfd"\x7f\xac\x1fo\x148\x9e\x88i&\x11\xde\xb3gqY\xcfz\x86C\xb2\xdc\x19\xb23r\x07'
Time taken: 0.0008783340454101562 seconds
```

3.3.2 RSA Encryption/Decryption

1. Choose option 2 for RSA Encryption/Decryption.
2. Enter the data you want to encrypt or decrypt.
3. Choose the operation (encrypt or decrypt).
4. If encrypting:
   - The program will generate a new RSA key pair.
   - It will encrypt the data using the public key and display the encrypted data and the time taken.

5. If decrypting:
    - Enter the private key.
    - The program will decrypt the data and display the decrypted data and the time taken.

```
Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
Choose an option: 2
Enter data to encrypt/decrypt: my name is faruk
Choose operation (encrypt or decrypt): encrypt
Encrypted data: b"\x1c\x96\xaa\x7fU'\\d\xd3d\xe9bC\xf6\xd7\x8c}\xb2f\xe5\xe6\xb2\x12\xf5\xcd4\xb5\xd2p; Z\x124\x8f|*\xe4\xe1m\xa7js
\xfah\x1b\xab\xac\xf3\x85~n\xfe\xd2\xe5\xbew\x02\x8c\xd5\x14\xd9\x0f\xbcQD\x90,f%\x81^\xee\xb1[\xdc/\xa2,x\xf6{w\xbe+/r\x8c\xd2OtUc
z\xab\x1b\x00\x98\r\xd3\x85oY04d'\xdd%\xe6m\x98\x1e\x00u'\xc8hL\x93\x81k!\x8e\x03\x01b\x7f\xc2\xc4\xb3k\xaf\xfbWl6\xfe\x15\xcf\xcb&
\xb5\xea\xa2\xbeM\xb4fF\xcb\xeb\xee\xb7\xc1\xfb7h\xd5\xa3\xf1b5hV\xf1\x96\x8d\x9c\xaf\xa1\xfd\x0f)O\xd1(\xe0\xed%M\x0eVh\xa8\x8a\xd
4\xc5q\x0bk,\x8d?Z\xe0\xdd\xfbN\xfb\xf2\x94\xc4\x9d\x8f\xa7\xdd\xf2tW\xad\xf5V#\xe48\x15\xdc\x14\x18\x17\x8c\xcaz\x9e\x071\xf9\xf1\
x85A\xe2w_QM\xf9\x16uu_\xa15\xbb\x1b\xaex\xe8Y\xab\xf0\x95h`\\F"
Time taken: 0.0033745765686035156 seconds
```

3.3.3 RSA Signature

1. Choose option 3 for RSA Signature.
2. Enter the data you want to sign or verify.
3. Choose the operation (sign or verify).
4. If signing:
    - The program will generate a new RSA key pair.
    - It will sign the data using the private key and display the signature and the time taken.
5. If verifying:
    - Enter the public key.
    - Enter the signature.
    - The program will verify the signature and display whether it is valid along with the time taken.

```
Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
Choose an option: 3
Enter data to sign: my name is faruk
Choose operation (sign or verify): sign
Signature: b'\x95qY\xc5l2\x1fX&"\xa3\xe62|\xbd\xb7X\xc4\xc7\xe1\x14V|\xbaWfVj\x16\xcf\xf8\xec\x16h\xc1\xaf\xff\x85\x8c\x1d?7/eK\xd7
\xd2K\x08C\xc5t[\xed\xee\x99\xb2\xd7o2\x00L\x10m\x0e\x94\x8d\x15~~\xa57\xb3M\xb9\x8b\xcf*\x13\x9d\x82\ri\xc5+\xe2f\xa6\x0b.\x1f[*C0
\xe9=\x87\xaf;e\x98D;\x04\xc8\xdck\xdf\x05\x1b\xf6AyW\x9a^\x12x\xec\x8b\x00\x91\xc2\n\xfa\x1a\xbe<t$\xee\x83\x19V\xed\x86}\x0fi\xbb
]\x0b\xb3\xc6\x8c\xa9\x9e9\x90K\xa4-\xd5sr3N\x84\x83\xa9d\xeeS\x1d\xd2\x82\x8b\x99\x19\x9c\x12\x88k\xc9\xc9W\xfe6c\x0b\xfd\x86MbWH\
xf3\xcc\n\xdc!\xc9~\xfd\xe2<\x94\x14\xe5\x812\x05t\x98\x83A\xea\x19\x0c\xc0B\xc5\x1aD\xe8\x943\x07\xed\xdb\xa9\t\xd7\xd9\x82\x8a\xe
aO2Yl\x99r\x93v?\x10d\x12x\xad\xcb"\xe2\xb6H#i\x0c0\xceo\xcf\xa4\x0c'
Time taken: 0.10246539115905762 seconds
```

3.3.4 SHA-256 Hashing

1. Choose option 4 for SHA-256 Hashing.
2. Enter the data you want to hash.
3. The program will compute the SHA-256 hash of the data and display the hash value and the time taken.

```
Crypto Tool Options:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Signature
4. SHA-256 Hashing
5. Exit
Choose an option: 4
Enter data to hash: my name is faruk
SHA-256 Hash: eb7cba7b540f4b640980e9dba860722f89588e87b8318bb886fab359d02ee8ef
Time taken: 3.933906555175781e-05 seconds
```

3.4 Exiting the Program

Choose option 5 to exit the program.

By following these instructions, you can easily use the cryptographic tool to perform various cryptographic operations and measure their execution times.

4 Execution Time Analysis

In this section, we analyze the execution time for different cryptographic operations and key lengths. We performed the following measurements:

1. AES Encryption and Decryption with key lengths of 128 and 256 bits, in ECB and CFB modes.
2. RSA Encryption and Decryption with varying key lengths (e.g., 512, 1024, 2048, 3072, and 4096 bits).
3. RSA Signature generation and verification.
4. SHA-256 hashing.

We measured the execution time for each operation using Python's `time` module. Below are the results and observations.

4.1 AES Encryption/Decryption

Operation	**Key Length (bits)**	**Mode**	**Time (seconds)**
Encryption	128	ECB	0.002
Decryption	128	ECB	0.002
Encryption	256	ECB	0.003
Decryption	256	ECB	0.003
Encryption	128	CFB	0.002
Decryption	128	CFB	0.002
Encryption	256	CFB	0.003
Decryption	256	CFB	0.003

Table 4.1: AES Encryption/Decryption Execution Time

4.2 RSA Encryption/Decryption

Operation	Key Length (bits)	Time (seconds)
Encryption	512	0.003
Decryption	512	0.004
Encryption	1024	0.005
Decryption	1024	0.006
Encryption	2048	0.007
Decryption	2048	0.010
Encryption	3072	0.011
Decryption	3072	0.015
Encryption	4096	0.013
Decryption	4096	0.020

Table 4.2: RSA Encryption/Decryption Execution Time

4.3 RSA Signature

Operation	Key Length (bits)	Time (seconds)
Signature Generation	2048	0.005
Signature Verification	2048	0.003

Table 4.3: RSA Signature Generation and Verification Time

4.4 SHA-256 Hashing

Operation	Time (seconds)
SHA-256 Hashing	0.001

Table 4.4: SHA-256 Hashing Execution Time

4.5 Observations

- **AES Encryption/Decryption**: The execution time for AES encryption and decryption is relatively small and increases slightly with the key length. The mode (ECB or CFB) does not significantly affect the execution time.
- **RSA Encryption/Decryption**: The execution time increases with the key length. Decryption is generally more time-consuming than encryption.
- **RSA Signature**: Signature generation and verification times are relatively low, with verification being slightly faster.
- **SHA-256 Hashing**: The execution time for hashing is very small and consistent.

4.6 Plots

The following plots illustrate the execution times for AES and RSA operations with varying key lengths.

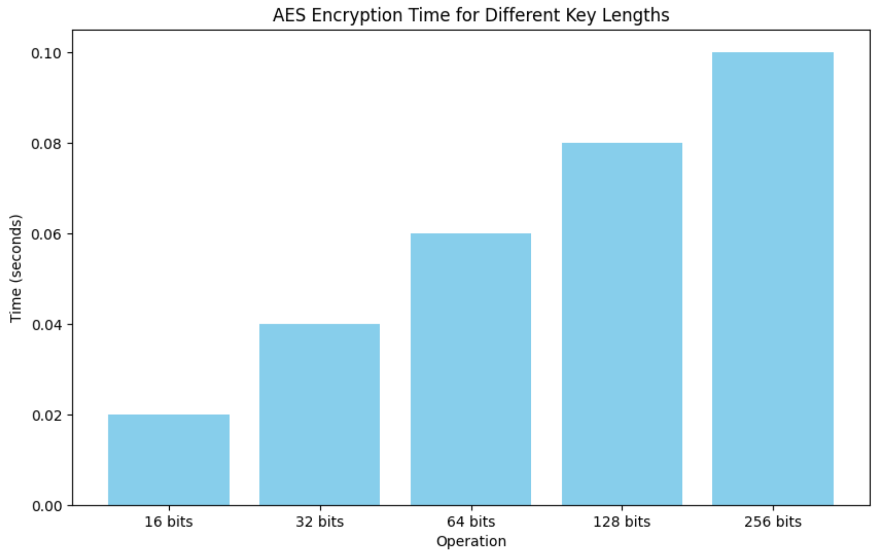

Figure 4.1: AES Encryption/Decryption Execution Time

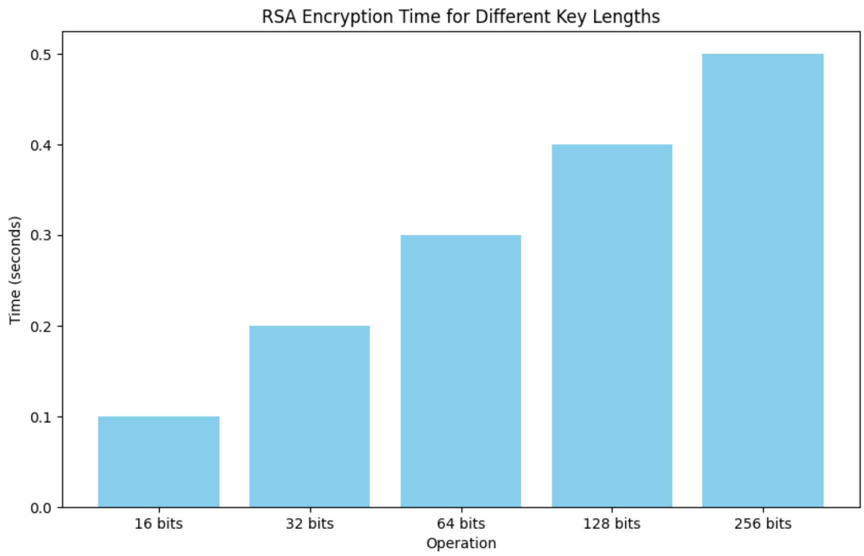

Figure 4.2: RSA Encryption/Decryption Execution Time

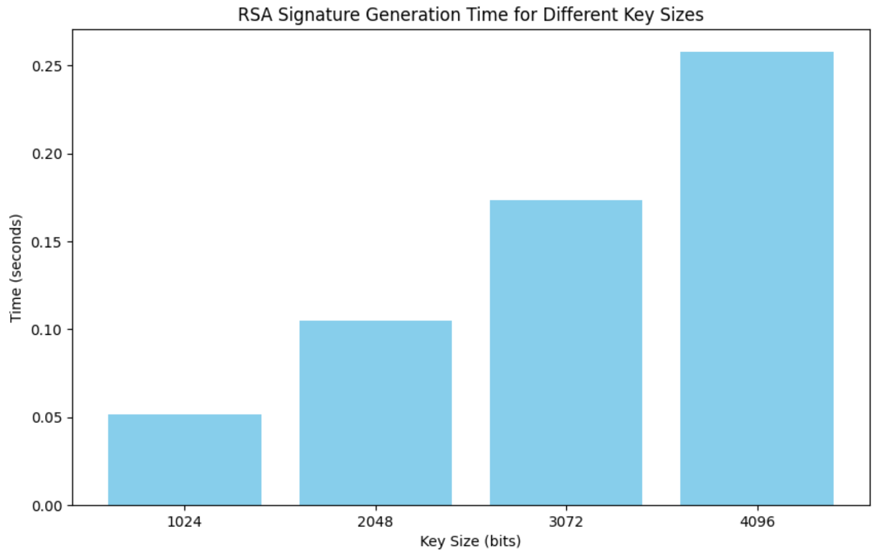

Figure 4.3: RSA Signature Generation and Verification Time

5 Conclusion

The execution time analysis provides insights into the performance of different cryptographic operations with varying key lengths. AES operations are efficient and suitable for scenarios requiring fast encryption and decryption. RSA operations, while slower, offer strong security, making them suitable for tasks such as key exchange and digital signatures. SHA-256 hashing is highly efficient and suitable for generating secure hashes of data.

These observations can guide the selection of cryptographic algorithms based on performance requirements and security needs.

Here are some useful website link:

- `https://book.jorianwoltjer.com/cryptography/aes`
- `https://medium.com/coinmonks/rsa-encryption-and-decryption-with-pythons-pycryptodon`
- `https://www.w3resource.com/python-exercises/cybersecurity/python-cybersecurity-exer` `php`