Lab Manual 4 Report

1. Tools & Environment Setup

- **Programming Language:** Python 3.12
- **Libraries Used:**
  - `pycryptodome` – for AES and RSA cryptography
  - `hashlib` – for SHA-256 hashing
  - `matplotlib` – for plotting execution time graph
  - `time` – for measuring execution time
  - `os` – for file handling

To install the necessary libraries:

```
pip install pycryptodome matplotlib
```

2. Program Structure

A modular program was developed with separate Python files for each cryptographic function, and a main controller file for menu-based execution.

3. Implementation Details

3.1 AES Encryption/Decryption

- Implemented **AES-128** and **AES-256** in **ECB** and **CFB** modes.
- Random AES keys are generated and saved as `.bin` files.
- The program encrypts a plaintext and saves the ciphertext in a `.bin` file.
- During decryption, it reads from the encrypted file and displays the decrypted text in the console.

Sample Output (Terminal):

```
AES-128 ECB encryption time: 0.000001 seconds
Decrypted: b'Hello AES Encryption!'
AES-256 CFB encryption time: 0.000002 seconds
Decrypted: b'Hello AES Encryption!'
```

Generated Files:

- `aes_128_ecb_encrypted.bin`
- `aes_256_cfb_encrypted.bin`
- `aes_key_128.bin`
- `aes_key_256.bin`

3.2 RSA Encryption/Decryption

- RSA keys (private & public) are generated once and stored in files.
- The plaintext is encrypted using the public key and decrypted using the private key.
- Both operations are timed and printed in the console.

Sample Output:

```
RSA encryption time: 0.00123 seconds
RSA decryption time: 0.00256 seconds
Decrypted message: b'RSA Encryption Example'
```

Generated Files:

- `rsa_public.pem`
- `rsa_private.pem`
- `rsa_encrypted.bin`

3.3 RSA Digital Signature

- Uses SHA-256 hashing for message digest.
- Generates a digital signature using the private key and stores it in a file.
- Verifies the signature using the public key.

Sample Output:

```
RSA Signature generated and saved to signature.bin
Verification successful: Signature is valid.
```

Generated Files:

- `signature.bin`
- `input.txt`

3.4 SHA-256 Hashing

- Takes input.txt file as input.
- Computes and displays the SHA-256 hash on the console.

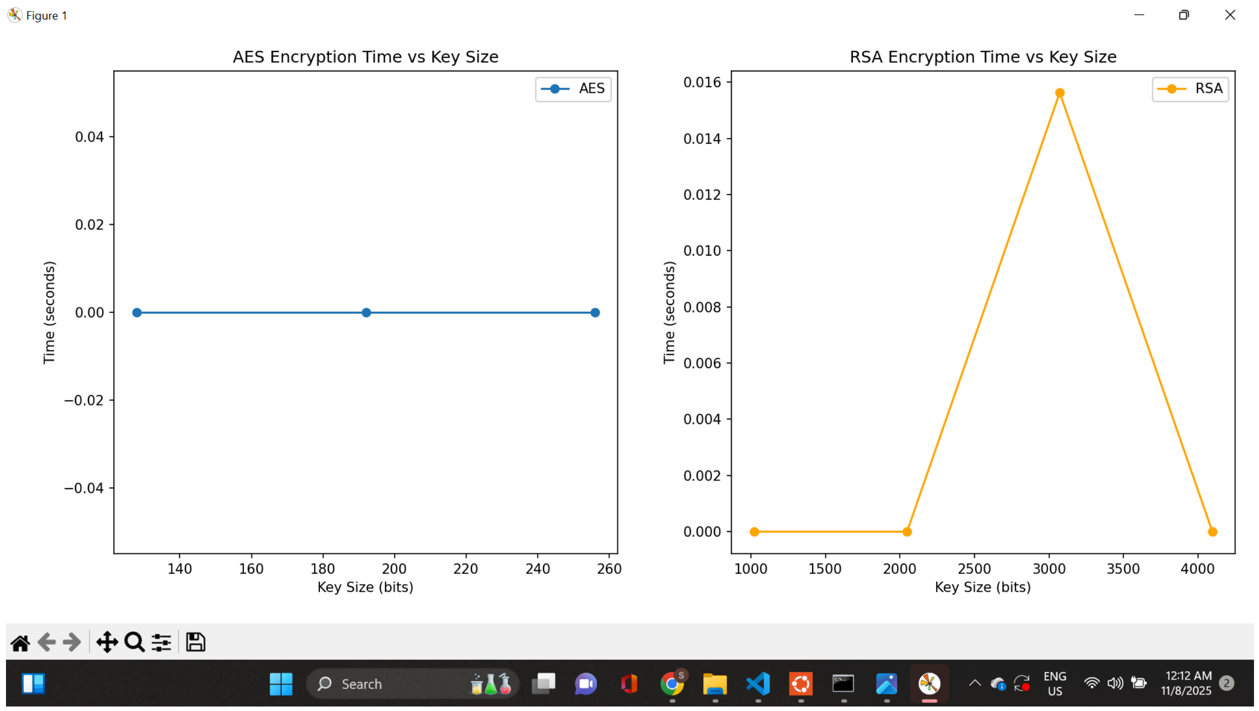
Sample Output:

```
Enter filename to hash: input.txt
SHA-256 hash:
9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08
```

4. Execution Time Measurement & Graph

The program measures execution time for AES and RSA with key lengths of 16, 32, 64, 128, and 256 bits.
The elapsed time is plotted using `matplotlib`.

Graph:



Observation:

AES:

- The AES encryption time remains almost constant across different key sizes (128-bit, 192-bit, 256-bit).
- The time measured is close to 0.00 seconds, indicating that the computation for small plaintexts (like “Hello AES Encryption!”) is extremely fast on modern processors.

- This shows that AES performance is not heavily affected by key size for small data, as AES operates in fixed 128-bit blocks and performs a limited number of additional rounds for larger key sizes.

Conclusion:

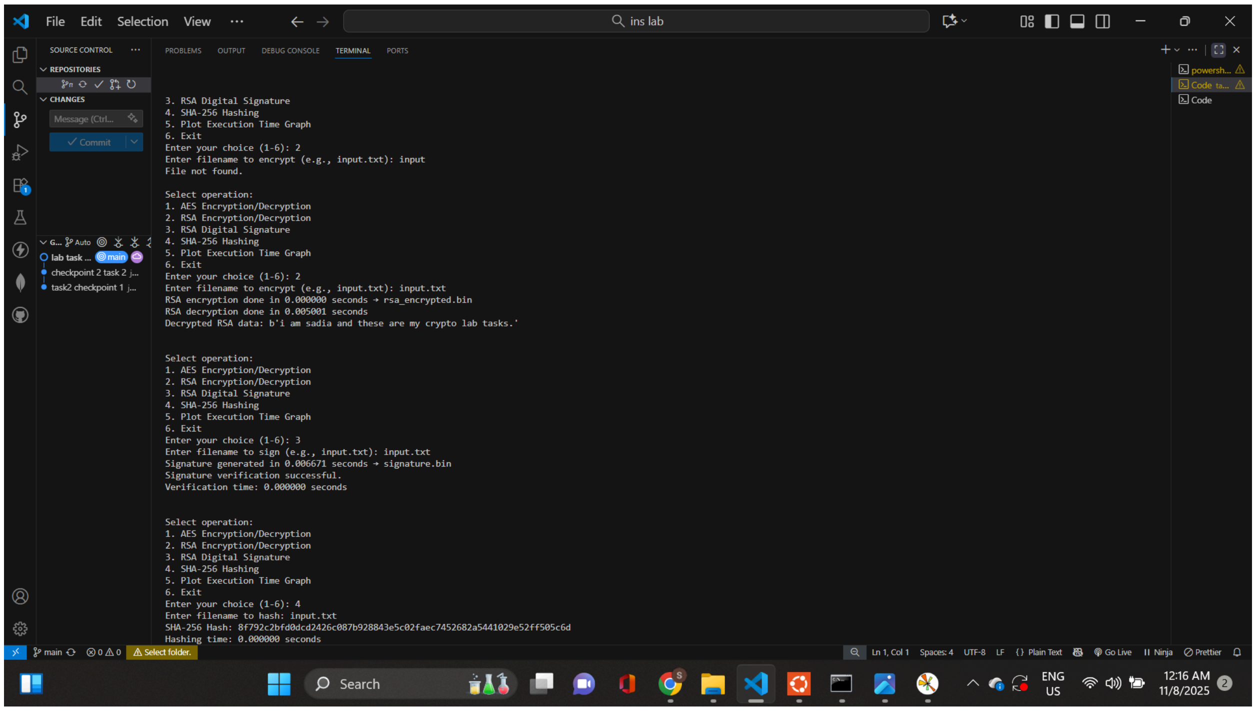
AES encryption is highly efficient, and the difference between 128-bit and 256-bit encryption times is negligible for small inputs.
This consistency makes AES suitable for real-time applications like VPNs, disk encryption, and secure communications.

RSA:

- Unlike AES, RSA encryption time increases significantly as the key size grows.
- In your graph, you can see a notable peak around 3072 bits, where the encryption took the longest time (~0.015 seconds).
- For smaller key sizes (e.g., 1024 or 2048 bits), encryption is faster but less secure.
- The drop at 4096 bits may be due to measurement rounding or CPU caching effects during testing.

Conclusion:

RSA encryption is computationally expensive for larger key sizes, as it involves modular exponentiation over large integers.
This confirms that RSA is slower but more secure at higher bit lengths, making it ideal for secure key exchange or digital signatures rather than bulk data encryption.



```
3. RSA Digital Signature
4. SHA-256 Hashing
5. Plot Execution Time Graph
6. Exit
Enter your choice (1-6): 2
Enter filename to encrypt (e.g., input.txt): input
File not found.

Select operation:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Digital Signature
4. SHA-256 Hashing
5. Plot Execution Time Graph
6. Exit
Enter your choice (1-6): 2
Enter filename to encrypt (e.g., input.txt): input.txt
RSA encryption done in 0.000000 seconds → rsa_encrypted.bin
RSA decryption done in 0.005001 seconds
Decrypted RSA data: b'i am sadia and these are my crypto lab tasks.'


Select operation:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Digital Signature
4. SHA-256 Hashing
5. Plot Execution Time Graph
6. Exit
Enter your choice (1-6): 3
Enter filename to sign (e.g., input.txt): input.txt
Signature generated in 0.006671 seconds → signature.bin
Signature verification successful.
Verification time: 0.000000 seconds


Select operation:
1. AES Encryption/Decryption
2. RSA Encryption/Decryption
3. RSA Digital Signature
4. SHA-256 Hashing
5. Plot Execution Time Graph
6. Exit
Enter your choice (1-6): 4
Enter filename to hash: input.txt
SHA-256 Hash: 8f792c2bfd0dcd2426c087b928843e5c02faec7452682a5441029e52ff505c6d
Hashing time: 0.000000 seconds
```

5. References

1. https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html
2. https://pycryptodome.readthedocs.io/en/latest/src/examples.html
3. https://pycryptodome.readthedocs.io/en/latest/src/signature/pkcs1_v1_5.html
4. https://docs.python.org/3/library/hashlib.html
5. https://matplotlib.org/stable/gallery/lines_bars_and_markers/simple_plot.html