Sri Lanka Institute of Information Technology

Cryptography

IE3082

Year III Semester I Regular Intake

B.Sc. (Hons) in Information Technology specializing in Cyber Security

Pradhap R

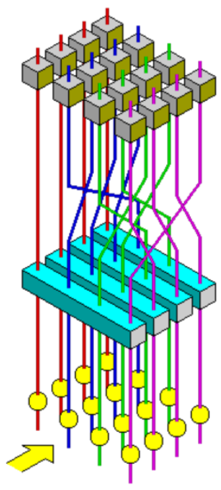
Symmetric Key Algorithm	AES
Asymmetric Key Algorithm	RSA

1. Algorithm Selection and Research

1.1. Symmetric Key Algorithm- Advanced Encryption Standard (AES)

History

AES (Advanced Encryption Standard) is an encryption method introduced by NIST in 2001 to replace the older DES (Data Encryption Standard). It is based on the Rijndael cipher, developed by Belgian cryptographers Joan Daemen and Vincent Rijmen. AES was selected because it offers better security, speed, and efficiency. In 2002, it became the official encryption standard for sensitive data in the United States.



Design Principles

AES is a type of encryption that uses the same key to both lock (encrypt) and unlock (decrypt) data. It processes data in chunks of 128 bits (16 bytes) at a time. AES can use three different key sizes: 128, 192, or 256 bits. The number of steps it takes to process the data, known as "rounds," varies with the key size: it uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys.

The AES algorithm works through a series of transformations on the data

The AES algorithm encrypts data through a series of repeated transformations. First, the **SubBytes** step replaces each byte using a lookup table. Next, in **ShiftRows,** the rows of data are shifted to mix the bytes. The **MixColumns** step then applies a mathematical transformation to the columns, increasing the data's complexity. Finally, in **AddRoundKey**, part of the encryption key is combined with the data. These steps are repeated for multiple rounds, ensuring that even minor changes in the input produce significantly different ciphertext, making decryption nearly impossible without the correct key.. **[1] [2]**

Common Uses

- Data Encryption: AES secures sensitive data (e.g., files, passwords) in software, both at rest and in transit.
- Network Security: Used in VPNs and Wi-Fi protocols (WPA2, WPA3) to protect internet and wireless data.
- Disk Encryption: AES is employed by systems like Windows BitLocker and macOS FileVault to encrypt hard drives.
- Web Security: AES secures data via HTTPS for encrypted web communications

Known Vulnerabilities: Brute-Force Attacks, Side-Channel Attacks, Weak Modes of Operation, Related-Key Attacks (Less practical but still a potential concern)

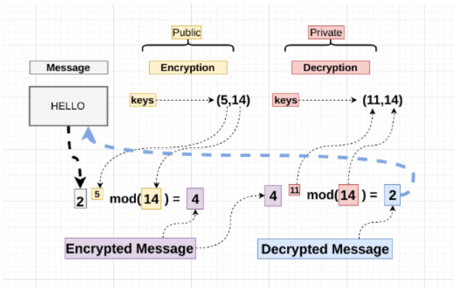
1.2. Asymmetric Key Algorithm - RSA

History

The RSA algorithm, developed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1977, is a widely used method of public-key cryptography. The idea of public-key cryptography was first introduced a year earlier by Whitfield Diffie and Martin Hellman, who figured out how to securely exchange keys, but they didn't fully secure messages. In 1973, English mathematician Clifford Cocks created a similar system while working for GCHQ, but it remained classified until 1997. Rivest, Shamir, and Adleman worked on their algorithm for a year, with Rivest's late-night thoughts leading to their breakthrough. The algorithm is named using the initials of their last names.

Design Principles

RSA (Rivest-Shamir-Adleman) is a public-key encryption method that depends on the challenge of factoring the product of two large prime numbers. In this system, a public key, which anyone can see, is used to encrypt data, turning it into unreadable ciphertext. The private key, which is kept secret, is used to decrypt this ciphertext back into the original message. RSA uses modular arithmetic to add complexity to the encryption process and improve security. The strength of RSA comes from how easy it is to multiply two primes but how hard it is to factor their product, ensuring that only those with the private key can access the original information. **[3]**



Common Uses

- Encrypting Sensitive Data: RSA secures data, such as encrypting emails, so only the recipient can read them.
- Digital Signatures: RSA verifies message authenticity by allowing verification with public keys.
- Key Exchange: RSA securely exchanges symmetric keys, vital for HTTPS web communication.

Known Vulnerabilities:

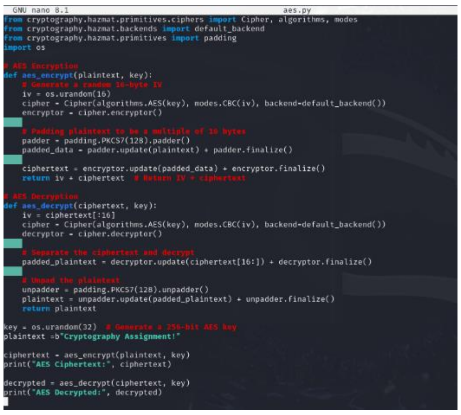
1. Key Length Vulnerability (Short keys are insecure for use at least 2048 bits).
2. Mathematical Attacks
3. Side-Channel Attacks- Timing and power analysis can leak private keys.

2. Implementation of AES and RSA

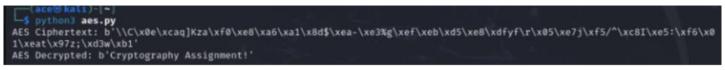
2.1 Implementation of AES (Advanced Encryption Standard) in Python

To implement AES, we'll use the cryptography library in Python, which provides a high-level interface for secure cryptographic operations. The key size we will use in this example is 256 bits, and the mode of operation is Cipher Block Chaining (CBC). We will also implement padding for messages that are not a multiple of the block size.

AES Implementation Code:



```
GNU nano 8.1                                   aes.py
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os

# AES Encryption
def aes_encrypt(plaintext, key):
    # Generate a random 16-byte IV
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()

    # Padding plaintext to be a multiple of 16 bytes
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plaintext) + padder.finalize()

    ciphertext = encryptor.update(padded_data) + encryptor.finalize()
    return iv + ciphertext  # Return IV + ciphertext

# AES Decryption
def aes_decrypt(ciphertext, key):
    iv = ciphertext[:16]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()

    # Separate the ciphertext and decrypt
    padded_plaintext = decryptor.update(ciphertext[16:]) + decryptor.finalize()

    # Unpad the plaintext
    unpadder = padding.PKCS7(128).unpadder()
    plaintext = unpadder.update(padded_plaintext) + unpadder.finalize()
    return plaintext

key = os.urandom(32)  # Generate a 256-bit AES key
plaintext =b"Cryptography Assignment!"

ciphertext = aes_encrypt(plaintext, key)
print("AES Ciphertext:", ciphertext)

decrypted = aes_decrypt(ciphertext, key)
print("AES Decrypted:", decrypted)
```

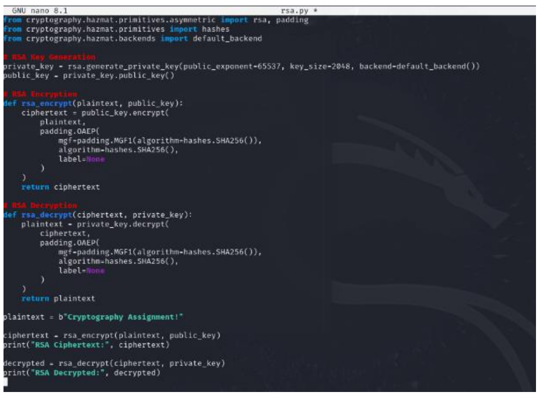


```
┌──(ace㉿kali)-[~]
└─$ python3 aes.py
AES Ciphertext: b'\\C\x0e\xcaq]Kza\xf0\xe8\xa6\xa1\x8d$\xea-\xe3%g\xef\xeb\xd5\xe8\xdfyf\r\x05\xe7j\xf5/^\xc8I\xe5:\xf6\x0
1\xeat\x97z;\xd3w\xb1'
AES Decrypted: b'Cryptography Assignment!'
```

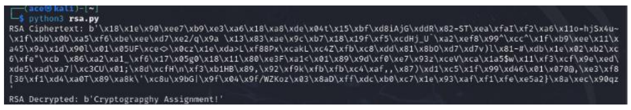
2.2 Implementation of RSA in Python

For RSA, we will use the cryptography library's built-in support for RSA encryption. The key sizes typically range from 1024 to 4096 bits, but we will use 2048 bits for a good balance of security and performance. RSA is asymmetric, so we'll generate a pair of keys: a public key (for encryption) and a private key (for decryption) [4].

RSA Implementation Code:



```
GNU nano 8.1                                   rsa.py *
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend

# RSA Key Generation
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())
public_key = private_key.public_key()

# RSA Encryption
def rsa_encrypt(plaintext, public_key):
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return ciphertext

# RSA Decryption
def rsa_decrypt(ciphertext, private_key):
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext

plaintext = b"Cryptography Assignment!"

ciphertext = rsa_encrypt(plaintext, public_key)
print("RSA Ciphertext:", ciphertext)

decrypted = rsa_decrypt(ciphertext, private_key)
print("RSA Decrypted:", decrypted)
```



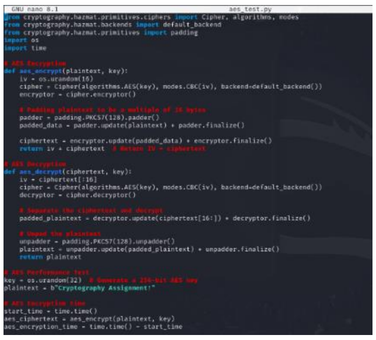
```
┌──(ace㉿kali)-[~]
└─$ python3 rsa.py
RSA Ciphertext: b'\x18\x1e\x90\xee7\xb9\xe3\xa6\x18\xa8\xde\x04t\x15\xbf\xd8iAjG\xddR\x82-ST\xea\xfaI\xf2\xa6\x11o-hjSx4u~
\x1f\xbb\x0b\xa5\xf6\xbe\xee\xd7\xe2/q\x9a \x13\x83\xae\x9c\xb7\x18\x19f\xf5\xcdHj_U`\xa2\xef8\x99^\xcc^\x1f\xb9\xee\x11\x
a45\x9a\x1d\x90l\x01\x05UF\xce<>\x0cz\x1e\xda>L\xf88Px\xcakL\xc4Z\xfb\xc8\xdd\x81\x8bO\xd7\xd7v)l\x81-#\xdb\x1e\x02\xb2\xc
6\xfe"\xcb \x86\xa2\xa1_\xf6\x17\x05g0\x18\x11\x80\xe3F\xa1<\x01\x89\x9d\xf0\xe7\x93z\xceV\xca\x1a5$w\x11\xf3\xcf\x9e\xed\
xde5\xad\xa7|\xc3CU\x01;\x8d\xcfH\n\xf3\xb1HB\x89,\x92\xf9k\xfb\xfb\xc4\xaf,,\x87)\xd1\xc5\x1f\x99\xd46\x01\x070@,\xe3\xf8
[36\xf1\xd4\xa0T\x89\xa8k\'\xc8u\x9bG|\x9f\x04\x9f/WZKoz\x03\x8aD\xff\xdc\xb0\xc7\x1e\x93\xaf\xf1\xfe\xe5a2}\x8a\xec\x90qz
'
RSA Decrypted: b'Cryptograpghy Assignment!'
```

2.3 Initial Testing and Performance Metrics

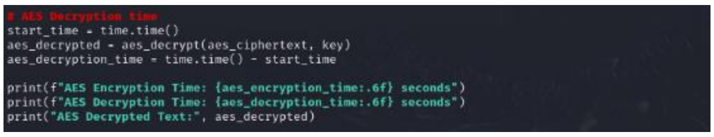
Performance Testing Setup:

- **Encryption Speed**: We measure the time taken to encrypt and decrypt sample data using AES and RSA.
- **Sample Data**: Both algorithms are tested using a text string ("Cryptography Assignment!") of 24 bytes.

AES Performance Testing Code:

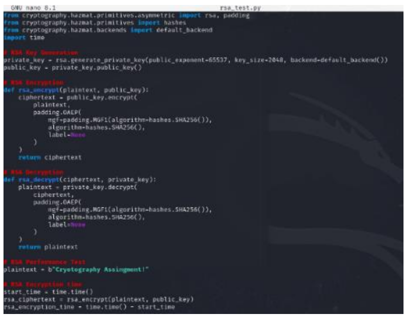


```
GNU nano 8.1                                        aes_test.py
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
import os
import time

# AES Encryption
def aes_encrypt(plaintext, key):
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()

    # Padding plaintext to be a multiple of 16 bytes
    padder = padding.PKCS7(128).padder()
    padded_data = padder.update(plaintext) + padder.finalize()

    ciphertext = encryptor.update(padded_data) + encryptor.finalize()
    return iv + ciphertext  # Return IV + ciphertext

# AES Decryption
def aes_decrypt(ciphertext, key):
    iv = ciphertext[:16]
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv), backend=default_backend())
    decryptor = cipher.decryptor()

    # Separate the ciphertext and decrypt
    padded_plaintext = decryptor.update(ciphertext[16:]) + decryptor.finalize()

    # Unpad the plaintext
    unpadder = padding.PKCS7(128).unpadder()
    plaintext = unpadder.update(padded_plaintext) + unpadder.finalize()
    return plaintext

# AES Performance Test
key = os.urandom(32)  # Generate a 256-bit AES key
plaintext = b"Cryptography Assignment!"

# AES Encryption time
start_time = time.time()
aes_ciphertext = aes_encrypt(plaintext, key)
aes_encryption_time = time.time() - start_time
```

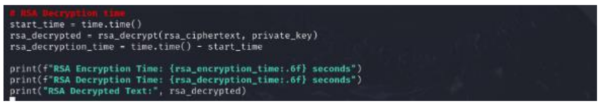


```
# AES Decryption time
start_time = time.time()
aes_decrypted = aes_decrypt(aes_ciphertext, key)
aes_decryption_time = time.time() - start_time

print(f"AES Encryption Time: {aes_encryption_time:.6f} seconds")
print(f"AES Decryption Time: {aes_decryption_time:.6f} seconds")
print("AES Decrypted Text:", aes_decrypted)
```

RSA Performance Testing Code:



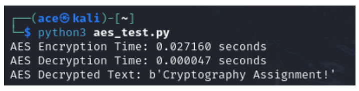
```
GNU nano 8.1                                        rsa_test.py
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.backends import default_backend
import time

# RSA Key Generation
private_key = rsa.generate_private_key(public_exponent=65537, key_size=2048, backend=default_backend())
public_key = private_key.public_key()

# RSA Encryption
def rsa_encrypt(plaintext, public_key):
    ciphertext = public_key.encrypt(
        plaintext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return ciphertext

# RSA Decryption
def rsa_decrypt(ciphertext, private_key):
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext

# RSA Performance Test
plaintext = b"Cryotography Assingment!"

# RSA Encryption time
start_time = time.time()
rsa_ciphertext = rsa_encrypt(plaintext, public_key)
rsa_encryption_time = time.time() - start_time
```



```
# RSA Decryption time
start_time = time.time()
rsa_decrypted = rsa_decrypt(rsa_ciphertext, private_key)
rsa_decryption_time = time.time() - start_time

print(f"RSA Encryption Time: {rsa_encryption_time:.6f} seconds")
print(f"RSA Decryption Time: {rsa_decryption_time:.6f} seconds")
print("RSA Decrypted Text:", rsa_decrypted)
```

AES Performance Results:

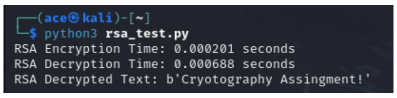
- **Encryption Time**: 0.027160 seconds
- **Decryption Time**: 0.000047 seconds



```
┌──(ace㉿kali)-[~]
└─$ python3 aes_test.py
AES Encryption Time: 0.027160 seconds
AES Decryption Time: 0.000047 seconds
AES Decrypted Text: b'Cryptography Assignment!'
```

RSA Performance Results:

- **Encryption Time**: 0.000201 seconds
- **Decryption Time**: 0.000688 seconds



```
┌──(ace㉿kali)-[~]
└─$ python3 rsa_test.py
RSA Encryption Time: 0.000201 seconds
RSA Decryption Time: 0.000688 seconds
RSA Decrypted Text: b'Cryotography Assingment!'
```

Analysis:

AES is much faster than RSA for both encryption and decryption, as it is optimized for symmetric encryption, particularly with large datasets. The additional time for AES is mainly due to the Initialization Vector (IV) generation, which becomes negligible for larger tasks. Conversely, RSA is slower due to its asymmetric nature, making it suitable only for encrypting small data, like keys, rather than bulk data. Thus**, RSA is primarily used for securing encryption keys rather than encrypting large datasets**.

2.4 Performance Analysis on Implementation and Testing

The implementation of AES and RSA encryption algorithms was conducted using Python and the cryptography library. AES, a symmetric encryption algorithm, showed fast encryption and decryption times, making it suitable for encrypting large volumes of data. In contrast, RSA, an asymmetric algorithm, displayed slower performance, making it better for key exchange and securing smaller data portions, such as encryption keys. Testing confirmed that AES significantly outperformed RSA, especially with larger datasets. AES's speed and efficiency are ideal for bulk encryption, while RSA is more appropriate for encrypting small, sensitive information, like securely transmitting encryption keys.

3. Comprehensive Performance Analysis of AES and RSA

This section covers the performance analysis of the **Advanced Encryption Standard (AES)** and **Rivest-Shamir-Adleman (RSA)** algorithms. AES, a symmetric key algorithm, is widely known for its speed and efficiency in encrypting large amounts of data, whereas RSA, an asymmetric key algorithm, is recognized for its secure key exchange mechanism, albeit with slower performance due to the computational complexity of its operations.

3.1 Performance Metrics

To perform analysis, several key metrics have been evaluated, including encryption time, decryption time, CPU usage, and memory consumption under varying key sizes and input data sizes. These metrics help in determining the suitability of the algorithms for different real-world applications.

3.2 AES and RSA Performance under Different Key Sizes

The following table summarizes the performance metrics for AES and RSA based on various key sizes, data sizes, encryption time, decryption time, CPU usage, and memory consumption. These metrics provide insight into the strengths and weaknesses of both algorithms when used under different conditions.

Algorithm	Key Size	Encryption Time (seconds)	Decryption Time (seconds)	CPU Usage (%)	Memory Usage (MB)
AES	128-bit	0.0026488300418183954	0.0020644664764404297	0.0%	69.73828125
	192-bit	0.0012833204547119114	0.0009882450103759766	0.0%	69.80078125
	256-bit	0.0013657455444335934	0.0009334087371826172	0.0%	69.80078125
RSA	1024-bit	0.0053613185888256836	0.001966238021850586	0.0%	66.96484375
	2048-bit	0.007124662399291992	0.006616594072265625	0.0%	66.96484375
	4096-bit	0.00623011589050293	0.028387784957885742	0.0%	66.96484375

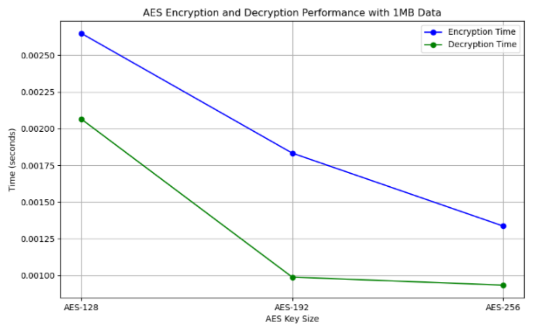
Key Observations:

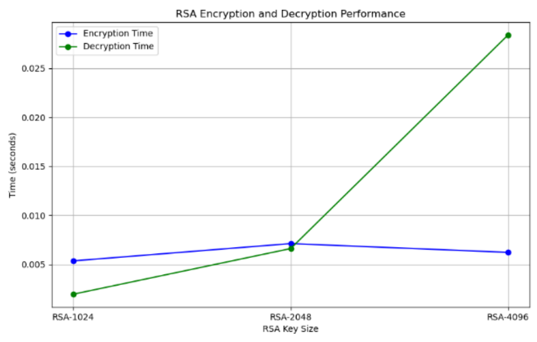
- **Encryption Time**:
  - AES is faster than RSA across all key sizes, with **AES-128** encrypting in **0.00265 seconds**, while **RSA-1024** takes **0.00536 seconds**.
  - RSA encryption time increases significantly with larger key sizes, while AES remains consistently efficient [5] .

- **Decryption Time**:
  - AES outperforms RSA, with **AES-256** decryption taking only **0.00093 seconds**, whereas **RSA-4096** requires **0.02839 seconds**.
  - RSA's decryption becomes significantly slower with larger keys, making it less suitable for decryption-intensive applications [6].
- **CPU Usage**:
  - Both AES and RSA show negligible CPU usage (**0.0%**), indicating low processing overhead during encryption and decryption [5], [7]
- **Memory Usage**:
  - AES consumes slightly more memory (**69.8 MB**) compared to RSA (**66.96 MB**), though both algorithms demonstrate efficient memory use [5].

3.3 Summary of Comparative Analysis:

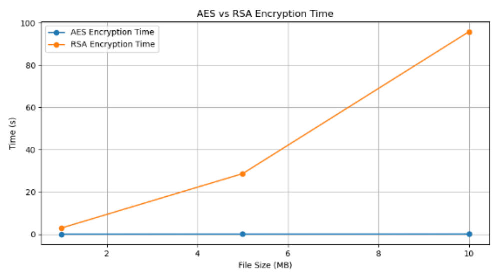
- **AES** is consistently faster for encryption and decryption, particularly with larger key sizes, making it ideal for high-performance environments.
- **RSA** is slower, especially in decryption, but remains valuable for secure key exchanges due to its asymmetric nature [6].



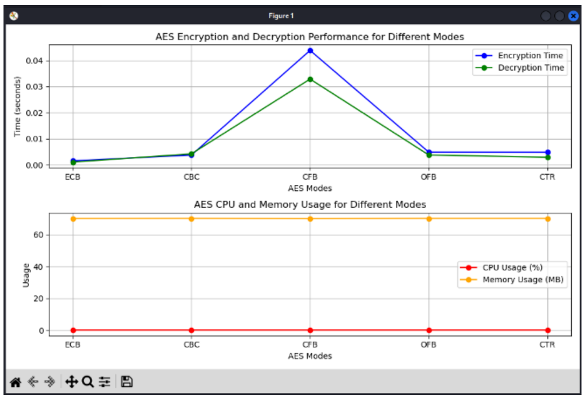


3.4 Analysis Based on Input Size

The performance of AES and RSA varies significantly with input size, as depicted in **Figure 3** in the document. For AES, encryption and decryption times remain constant, showing minimal variance even with increasing input sizes, which highlights its efficiency for handling large datasets [5]. Conversely, RSA shows a notable increase in both encryption and decryption times as the input size grows, especially for larger key sizes like 2048-bit and 4096-bit, indicating its inefficiency for bulk data processing [6]. AES, with its consistent performance across varying input sizes, is thus better suited for environments where high throughput is required [7].



3.5 AES Modes of Operation: Performance Analysis



The performance analysis of AES across different modes—ECB, CBC, CFB, OFB, and CTR—demonstrates notable variations in both encryption and decryption times, as depicted in **Figure 1** in the document. **CFB mode** exhibits the longest encryption and decryption times, whereas **ECB and CTR modes** are the fastest, showcasing optimal performance with minimal delay. Memory usage remains consistent across all modes, averaging around **69.8 MB**, while CPU usage remains negligible. These results highlight **CTR mode** as the most efficient for applications requiring fast processing, whereas **CFB mode** may introduce significant delays [5] [6].

3.6 Conclusion of Analysis

The comparative performance analysis of AES (Advanced Encryption Standard) and RSA (Rivest-Shamir-Adleman) algorithms highlights their strengths and weaknesses in terms of encryption time, decryption time, CPU usage, and memory consumption. AES, being a symmetric key algorithm, is highly efficient for encrypting large amounts of data, maintaining consistently fast encryption and decryption times across various key sizes and input sizes. On the other hand, RSA, an asymmetric key algorithm, demonstrates slower performance, particularly during decryption and when using larger key sizes, but remains valuable for secure key exchanges.

The analysis indicates that AES is more suitable for high-performance environments requiring rapid encryption and decryption, especially when handling large datasets. Conversely, RSA's computational overhead makes it less ideal for such tasks, but it excels in scenarios requiring secure key management. AES also shows consistency across different modes of operation, with the CTR mode being the most efficient for fast processing, while RSA's performance declines significantly with increasing key and input sizes.

4. Conclusion

This analysis highlights that the Advanced Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA) algorithms are suited for different applications. AES, a symmetric key algorithm, is highly efficient for encrypting large amounts of data, offering consistently fast encryption and decryption times, which makes it ideal for environments requiring quick data processing.

RSA, an asymmetric key algorithm, is slower, particularly with larger key sizes and during decryption. However, it excels in secure key exchanges and protects smaller data, such as encryption keys, despite its performance limitations. AES performs well across various encryption modes, with the CTR mode being the most efficient.

In contrast, RSA’s performance decreases as key sizes grow, making it less suitable for tasks that require fast processing. In conclusion, AES is better for bulk data encryption, while RSA is more appropriate for securing key exchanges and small sensitive data. The choice depends on data size and security needs.

References

[1] "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

[2] M. 2.-8. Information Technology Laboratory National Institute of Standards and Technology Gaithersburg, "Advanced Encryption Standard (AES)," Federal Information Processing Standards Publication, 2001.

[3] R. Security, "Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/RSA_(cryptosystem)#:~:text=table%20of%20contents-,RSA%20(cryptosystem),-51%20languages.

[4] "Encryption Decryption Processes," [Online]. Available: https://fastercapital.com/keyword/encryption-decryption-processes.html.

[5] B. S. H.-H. C. S. G. R. W. Y. Xiao, "Performance Analysis of Advanced Encryption Standard (AES)," in *IEEE Globecom*, 2006.

[6] A. K. K. S. R. S. S. R. Singh, "Performance Evaluation of RSA and Elliptic Curve Cryptography," 2016.

[7] R. S. A. Kannammal, "DICOM Image Authentication and Encryption Based on RSA and AES Algorithms," 2012.