REVISITING LIGHTWEIGHT CRYPTOGRAPHY OF IOT ARCHITECTURE

Abstract

A comparison between some of the used encryption algorithms in IoT devices because of their lightweight characteristic like AES-ECC-GCM, AES-ECC-EAX as well as RSA-ECC-EAX

Serene Mathew

B00177451

Shamil Yazeen

B00177246

INDEX

CHAPTER 1

INTRODUCTION

In the world of interconnected devices and increasing reliance on digital communication, sending and receiving sensitive information has become a vital challenge, especially where the current use of IoT (Internet of things) devices has significantly increase. We are heavily reliant on these devices to do our work. From using them in Personal assistant to switching on a living room light. IoT devices are particularly vulnerable due to their limited resources such as processing power, memory and if they are battery powered then energy drainage. As a result, encryption methods must maintain a balance between security and resource efficiency.

Hybrid Encryption is a method to do. This combines the most prominent features of asymmetric and symmetric cryptography to maintain this balance and demands. Asymmetric encryption, such as EEC (Elliptic Curve Cryptography), is a highly secure and suitable method for exchanging keys over unsecure networks. However, this is process and memory intensive for encrypting large data. Symmetric Encryption on the other hand, like AES (Advanced Encryption Standard) is faster and more efficient for encrypting data but requires a secure way to exchange keys. With modes EAX and GCM it becomes harder to choose from as they both have their own strengths. Like GCM is optimized for speed and high output applications but requires careful management. For EAX provides greater robustness against misuse and is better for environments where reliability and integrity are critical over sheer speed.

AES, a symmetric encryption algorithm, is known for its efficiency and speed in encrypting and decrypting data, making it suitable for the constrained computational power and memory of IoT devices. Whereas, ECC, which is asymmetric encryption algorithm, provides strong security with smaller key sizes, which are more resource efficient and easier to manage in IoT devices. This approach to take combination of types which is the hybrid approach which takes into account the strengths of both the encryption types like AES’s ability to process data quickly and ECC’s mechanism for secure key exchange.

As IoT devices have become widely used across various other domains, including home systems such as google nest, industrial automation, and healthcare, this makes It more necessary to have secure data transmission across these devices. Hybrid Encryption using AES and ECC addresses these needs by making

key exchange secure and a quicker way to process data simultaneously. AES, known for its speed and simplicity, is ideal for encrypting data, which is crucial for real-time applications in IoT. ECC, with is smaller key sizes and strong security properties, ensures that even if the data encryption keys are leaked, the overall security is maintained through periodic key rotation and secure key exchange. This dual protection ensures safety particularly in IoT devices, which often is found with limited computational power, storage and battery life. By combining these two methods, it not only secures the data being transmitted but also optimizes performances, ensuring that devices can run smooth without sacrificing security.

Objective:

The main objective of this guide is to explore how the hybrid encryption methods are being used and will continue to do so until a new kind of encryption algorithm is discovered that is memory, process and energy efficient that can match the current hybrid encryption algorithms efficiency. In this guide we will look at how the hybrid encryption is better and efficient that simpler light weight encryption algorithms for IoT devices. We will implement AES ECC GCM encryption and decryption to look at how the resources were used and comparing them with other lightweight ciphers. As well as implementing RSA EEC EAX encryption and AES ECC EAX to compare and find out which is more suitable for IoT devices data exchange.

CHAPTER 2

LITERATURE REVIEW

RSA:

RSA is one of the oldest and most widely used public key cryptographic algorithm or RSA (Rivest, Shamir, Adleman) cryptosystem which was named after its founders. They published their white paper on RSA in 1977. And they have been used still now for most recent applications of Standard RSA are incorporated in key exchanges, digital signatures and more. The Security of RSA relies on factorization of two large prime numbers, hence dubbed as the “factoring problem”. RSA is comparatively slow algorithm to other cryptographic algorithm.[1]

RSA was a patented technology by the same name, RSA has formed the basis of most public encryption in current systems. Its uses can be as public key encryption algorithm and certification process, which is used to exchange data over networks in a protected way, and providing privacy. RSA is difficult to factorize in polynomial time, so conventional computing systems, running in polynomial time, take a large amount of CPU cycles to find factorization solutions. Factorization can be implemented by using parallel processing methods, there is still research to be done in this area. This generally means that this will lead to faster prime number generating algorithms and also finding solutions to factorization questions. Also leading to prime numbers that are less likely to succumb to factorization method [2].

Normally the security provided by RSA can’t be breached easily by normal systems from everyday life. But even knowing that there have been several cases that have reported about breaking RSA ciphers with shorter key size, this makes it more challenging and offers the individuals in the field like cryptographers to come up with a more enhanced version of it [1] . in some encryption methodologies, key management is essential aspect that finalizes how the data will be encrypted. And some of the encryption ratio by which the image loss is calculated is based on this crucial length of key [3].

ECC:

Elliptic curve cryptography or ECC is one of the strongest and most efficient cryptographic methods in modern cryptography, it is an important cryptographic type in public key cryptography, while relies on an elliptic curve to encrypt or decrypt. An elliptic curve is a smooth affine curve with genus 1 in the domain and its expression is written as

$$y^2 = x(x-1)(x-\lambda)\,,\ \lambda \neq 0,1\,,\ \text{or}\ \ y^2 + ay = x^3 + bx^2 + cx + d\,.$$

If domain is not 2 and 3, it can also be written as

$$y^2 = x^3 + ax + b.$$

ECC uses inverse addition in elliptic curve as the key and is able to achieve higher encryption without complex procedures, so efficiency become higher than RSA. ECC is stronger because it has a stronger security level. ECC provides stronger protection than any other encryption algorithms at prevent attacks, this also makes it a better guarantee for mobile internet security. [4] As ECC has a relatively short key of 256 bits, hence it occupies lesser storage. And this is really good for mobile internet technology as a lot of people are using mobile internet to do various things, this makes ECC to provide a better customer experience.

ECDSA or elliptic curve digital signature algorithm refers to a digital signature that means signing certain information with a private key. This signature is verified at the other end of communication using their public key, as the signature can be done only by their private key [4].

AES:

NIST wanted to define an encryption process for non-military information security applications by US government. Other fields also benefited from the work of NIST. AES was a replacement for the Data encryption standard (DES). When compared with DES, AES has a larger block size with key size 128 bits to 256 bits. in the case of mode of operations, we have EAX and GCM

EAX:

It is a block-cipher mode of operation used in AEAD (authenticated-encryption with associated-data). EAX is gotten by initializing a simple composition method, EAX2, and then collapsing the dual keys in to one key. EAX was thought of as an alternative to CCM and is also patent free, making it available for future development and widespread use. [5] has been formally proven to be secure under standard complexity-theoretic assumptions. In conclusion, EAX is a block-cipher mode of operation that was introduced for AEAD that is for protecting the privacy of the message and the authenticity of both the message and the header; and is secure.

GCM:

Advanced encryption standard with Galois Counter Mode provides high validation of authenticity and data confidentiality. The key length of AES-GCM is 256 bit to provide high security system, which also goes through operation of key expansion designed for parallel optimization operation time of AES-GCM. This method was introduced by NIST, it has two main functions that are block cipher encryptions and multiplication over a field. This algorithm encrypts or decrypts with 128,192, or 256-bit cipher key. The transformation rounds executed by AES depends on length of cipher key. [6]

ECC vs RSA:

In general, ECC beats RSA in constrained environments with respect to energy consumption, memory space, and computation time. ECC can give the same level of security that RSA can give but with lesser key length thereby decrease memory space. As ECC uses smaller message size, this results in cost efficiency and faster exchange of data between endpoints that is IoT devices with constrained processing power, memory space. [7]

Hybrid Encryption:

Hybrid encryption is a type of encryption in which it combines the ease of public-key encryption that is asymmetric key encryption and efficiency of a symmetric key cryptosystem.[8] One of the examples are public key cryptography based on simple symmetric algorithm. In this case the hybrid encryption types would be:

1.AES-ECC-GCM
2.AES-ECC-EAX
3.RSA-AES-EAX

In an example like RSA-AES and RSA-DES, the comparison between them shows that RSA-AES is better than the other based on the security it provides. The AES along with RSA for the use of key management provides an efficient technique to make sure that the data transmitted is secure.

As both asymmetric and symmetric have their own advantages and disadvantages, the overall combination makes with help of certain constraint can make lightweight cryptosystems. Hybrid is more powerful in data securing and transmission especially if it is for small systems like IoT devices which are constrained in terms of memory and processing power. It mostly takes the advantages and not use their disadvantages of both cryptosystems. This makes hybrid cryptosystems more powerful and efficient.

Currently, the hybrid version of ECC and AES is being used in most IoT devices. Because of its advantages over traditional encryption system, it is widely accepted in most of devices if the requirements include little to no space and acceptable security, this is the way. In [9], comparison between the cryptosystems, it was found that the computation speed of asymmetric encryption is not as fast as symmetric encryption. As a result hybrid type of encryption was proposed. And in conclusion it was said that among may hybrid encryptions, the AES and ECC is a good example to strengthen data security without compromising the computation problems and memory issues.

As discussed earlier, EAX and GCM provide different levels of security each having their own advantages and disadvantages. EAX modes is a mode of operation of AES block ciphers that are created to provide both encryption and data security making it particularly suitable for resource-constrained IoT devices, such as those used in smart home systems which involve mathematical equations. [10] they also provide resistance to attacks like MITM (man in the middle), side channel and brute force attacks.

Some future trends in this space would be to evade quantum computers but as seeing they are lightweight and need to be lightweight because of the constraints in the IoT device environments. This will be difficult to provide resistance against, the power of Quantum computers is unimaginable, they are able to solve AES encryption with minutes. Being already smaller key size and length they are prone to attacks and can be

easily broken when quantum systems are involved. But quantum system are not found in your everyday house hold, and hence there is no real threat, until quantum computers also become the norm like the computer of today.

Current technology has shifted to machine learning technologies or AI (artificial intelligence), this can be a boon or bane depending on where they are used, there are proposed systems offering adaptive encryption protocols to provide key exchange baes on the type of system it is connected and the size of encryption key it can handle. But there is also a negative aspect to make the machine learn about how much the IoT device's processing power is and memory it can handle as well as the energy it contains, this all together can be used to crack an encryption based on this information alone.

CHAPTER 3

IMPLEMENTATION

Implementation of Hybrid AES-ECC using EAX modes:

1. **Generate ECC key pair:**

   - Using the ECC algorithm using cryptography library in python.
   - Before all of this we need to import all the modules and functions.
   - Here we use cryptography library's asymmetric module and its function **SECP256R1** function to generate public and private key.
   - Getting public key from private key and return both private and public key.

```python
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7
from cryptography.hazmat.primitives.keywrap import aes_key_wrap, aes_key_unwrap
from os import urandom

def generate_ecc_key_pair():
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()
    return private_key, public_key
```

2. **Deriving Shared Key:**

   - Performing ECDH (elliptic Curve Diffie-Hellman) using private key and the other sides public key to generate a shared secret.
   - Using HKDF (HMAC based key derivation function) with SHA256 to derive a 128-bit AES key from the above-mentioned shared secret.
   - Return the derived AES key.

```python
def derive_shared_key(private_key, peer_public_key):
    shared_secret = private_key.exchange(ec.ECDH(), peer_public_key)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data'
    ).derive(shared_secret)
    return derived_key
```

3. **AES Encryption:**

- We will generate a random 128-bit IV.
- Using AES with provided key and IV
- Applying PKCS7 padding to plaintext
- Encrypting the padded test
- Returning the IV and encrypted data.

```python
def aes_encrypt(data, key):
    iv = urandom(16)
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    encryptor = cipher.encryptor()
    padder = PKCS7(128).padder()
    padded_data = padder.update(data) + padder.finalize()
    encrypted_data = encryptor.update(padded_data) + encryptor.finalize()
    return iv, encrypted_data
```

4. **AES Decryption:**

- Using AES is CBC mode with the provided key and IV
- We decrypt the data
- We remove the padding done with PKCS7 from the decrypted data
- Returning the original plain text

```python
def aes_decrypt(encrypted_data, key, iv):
    cipher = Cipher(algorithms.AES(key), modes.CBC(iv))
    decryptor = cipher.decryptor()
    unpadder = PKCS7(128).unpadder()
    padded_data = decryptor.update(encrypted_data) + decryptor.finalize()
    decrypted_data = unpadder.update(padded_data) + unpadder.finalize()
    return decrypted_data
```

5. **Hybrid Encryption:**
    - Generating ECC key pairs for both device 1 and 2
    - Exchanging public keys between devices
    - Deriving a shared AES key for both devices
    - Verifying both devices having same shared key
    - Encrypt a sample plaintext using derived AES key
    - Decrypt the message using same AES key
    - Verify that the decrypted message matches original plaintext

```python
def hybrid_encryption_demo():
    private_key_device1, public_key_device1 = generate_ecc_key_pair()
    private_key_device2, public_key_device2 = generate_ecc_key_pair()

    shared_key_device1 = derive_shared_key(private_key_device1, public_key_device2)
    shared_key_device2 = derive_shared_key(private_key_device2, public_key_device1)

    assert shared_key_device1 == shared_key_device2, "Key derivation mismatch!"

    data = b"Hello, this is a hybrid AES-ECC EAX hybrid encryption demo!"
    iv, encrypted_data = aes_encrypt(data, shared_key_device1)
    print(f"Encrypted Data: {encrypted_data.hex()}")

    decrypted_data = aes_decrypt(encrypted_data, shared_key_device2, iv)
    print(f"Decrypted Data: {decrypted_data.decode()}")
```

6. **Running the demo:**
    - just running the demo getting output as encrypted data in hexadecimal and also the decrypted data which is the original message.

```
if __name__ == "__main__":
    hybrid_encryption_demo()
```

The output:

```
PS C:\Users\seren\OneDrive\Documents\vscode> & C:/Users/seren/AppData/Local/Programs/Python/Python312/python.exe c:/Users/seren/OneDrive/Documents/vscode/Prototype/AESECCEAX.py
Encrypted Data: b800019e7e94b5a4f1beef8aaaca01404a55ed75cabb645e985405a323af96b253e268d1b03388a80d3b4ab897ae132ae2a6f47a137d6d6abf9e013809211391
Decrypted Data: Hello, this is a hybrid AES-ECC EAX hybrid encryption demo!
```

Implementation of Hybrid RSA-ECC using EAX mode:

1. **Generate RSA Key pair:**
    - RSA key pairs are generated of 2048-bit
    - Extracting the private key and public key in PEM format
    - We also import Crypto module to gain access to other cipher functions
    - This outputs private and public key.

```
def hybrid_encryption_demo():
    private_key, public_key = generate_rsa_key_pair()
```

```
from Crypto.Cipher import AES, PKCS1_OAEP
from Crypto.PublicKey import RSA
from Crypto.Random import get_random_bytes

def generate_rsa_key_pair():
    key = RSA.generate(2048)
    private_key = key.export_key()
    public_key = key.publickey().export_key()
    return private_key, public_key
```

2. **Generate a Random AES key**
    - using get_random_bytes (32)
    - this creates a random aes symmetric key

```
aes_key = get_random_bytes(32)
```

3. **Encrypt the AES Key using RSA Public Key**
    - Importing the public key using RSA.import key
    - Then using RSA with OAEP padding which is PKCS1_OAEP function to encrypt the aes key
    - This result in encrypted_aes_key

```
encrypted_aes_key = rsa_encrypt_key(aes_key, public_key)
```

```
def rsa_encrypt_key(aes_key, public_key):
    rsa_key = RSA.import_key(public_key)
    cipher_rsa = PKCS1_OAEP.new(rsa_key)
    encrypted_aes_key = cipher_rsa.encrypt(aes_key)
    return encrypted_aes_key
```

4. **Encrypt the data using AES-EAX**
    - Encrypting the actual message using AES key
    - Creating an AES cipher object in EAX mode using AES.new(aes_key,AES.MODE_EAX)
    - Encrypting the data and generating an authentication tag

```
data = b"Hello, this is a hybrid RSA-AES encryption demo!"

nonce, ciphertext, tag = aes_encrypt(data, aes_key)
print(f"Ciphertext: {ciphertext.hex()}")
```

```python
def aes_encrypt(data, key):
    cipher = AES.new(key, AES.MODE_EAX)
    ciphertext, tag = cipher.encrypt_and_digest(data)
    return cipher.nonce, ciphertext, tag
```

5. **Decrypt the AES key using RSA private key**
    - Retrieving the AES key securely
    - Importing the private key
    - Using RSA with OAEP padding to decrypt the encrypted aes key
    - The results in the original AES key

```python
decrypted_aes_key = rsa_decrypt_key(encrypted_aes_key, private_key)
```

```python
def rsa_decrypt_key(encrypted_aes_key, private_key):
    rsa_key = RSA.import_key(private_key)
    cipher_rsa = PKCS1_OAEP.new(rsa_key)
    aes_key = cipher_rsa.decrypt(encrypted_aes_key)
    return aes_key
```

6. **Decrypt the Data using AES-EAX**
    - using AES.new() function to create AES decryption object
    - then decrypting the cipher text and verify the authentication tag.

```python
decrypted_data = aes_decrypt(nonce, ciphertext, tag, decrypted_aes_key)
print(f"Decrypted Data: {decrypted_data.decode()}")
```

```python
def aes_decrypt(nonce, ciphertext, tag, key):
    cipher = AES.new(key, AES.MODE_EAX, nonce=nonce)
    data = cipher.decrypt_and_verify(ciphertext, tag)
    return data
```

7. **Running the demo**
    - Just running the demo function to get the output with encrypted data and the plaint text it took.

```
if __name__ == "__main__":
    hybrid_encryption_demo()
```

The output:

```
PS C:\Users\seren\OneDrive\Documents\vscode> & C:/Users/seren/AppData/Local/Programs/Python/Python312/python.exe c:/Users/seren/OneDrive/Documents/vscode/Prototype/RSAECCEAX.py
Ciphertext: 409df8ef0feeeb7d70c22503e2a6b3460670899490ff7120d3f03cea21392aa45d2d96e69b9a02935718a5412ee58ac2
Decrypted Data: Hello, this is a hybrid RSA-AES encryption demo!
```

Implementation of Hybrid AES-ECC using EAX mode:

1. **Generate ECC Key pair:**
    - Generating private and public keys using SECP256R1 function curve for each device
    - which results in a private key and a public key
    - importing important modules from cryptography library

```
def hybrid_encryption_demo():
    private_key_ecc_device1, public_key_ecc_device1 = generate_ecc_key_pair()
    private_key_ecc_device2, public_key_ecc_device2 = generate_ecc_key_pair()
```

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.padding import PKCS7
from os import urandom

def generate_ecc_key_pair():
    private_key = ec.generate_private_key(ec.SECP256R1())
    public_key = private_key.public_key()
    return private_key, public_key
```

2. **Deriving a shared key using ECDH:**
   - Using the exchange() function of ECDH to compute a shared secret
   - Deriving a symmetric AES key from shared secret using HKDF which is a key derivation function using hash 256 as the has algorithm.
   - Which results in both devices sharing same shared aes key

```
shared_key_device1 = derive_shared_key(private_key_ecc_device1, public_key_ecc_device2)
shared_key_device2 = derive_shared_key(private_key_ecc_device2, public_key_ecc_device1)
```

```
def derive_shared_key(private_key, peer_public_key):
    shared_secret = private_key.exchange(ec.ECDH(), peer_public_key)
    derived_key = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b'handshake data'
    ).derive(shared_secret)
    return derived_key
```

3. **Encypt the data using AES-GCM**
   - Encrypting the plaint text securely using shared AES key
   - Generating a random 96-bit IV
   - Using AES-GCM for encryption of data and then producing an authentication tag to verify integrity and authenticity
   - Optionally padding the plaintext using PKCS7 for data compatibility

```
def aes_encrypt(data, key):
    iv = urandom(12)
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv))
    encryptor = cipher.encryptor()
    padded_data = PKCS7(128).padder().update(data) + PKCS7(128).padder().finalize()
    ciphertext, tag = encryptor.update(padded_data) + encryptor.finalize(), encryptor.tag
    return iv, ciphertext, tag
```

```
data = b"Hello, this is a hybrid ECC-AES GCM encryption demo!"
iv, ciphertext, tag = aes_encrypt(data, shared_key_device1)
print(f"Ciphertext: {ciphertext.hex()}")
```

4. **Decrypt data using AES-GCM**
   - Decrypting the message and verify its authenticity.
   - Using cipher object with AES-GCM mode, providing the IV and tag from encryption
   - Then decrypting the cipher text
   - Validating the tag to ensure data integrity.
   - the output we get is the decrypted data that is the original plaintext

```
decrypted_data = aes_decrypt(iv, ciphertext, tag, shared_key_device2)
print(f"Decrypted Data: {decrypted_data.decode()}")
```

```
def aes_decrypt(iv, ciphertext, tag, key):
    cipher = Cipher(algorithms.AES(key), modes.GCM(iv, tag))
    decryptor = cipher.decryptor()
    decrypted_data = decryptor.update(ciphertext) + decryptor.finalize()
    return decrypted_data
```

5. **Running the demo**
   - Just running the demo function to get output in terminal as hexadecimal value which is the encrypted data and also the plaintext data which is decrypted data.

```
if __name__ == "__main__":
    hybrid_encryption_demo()
```

The output:

```
PS C:\Users\seren\OneDrive\Documents\vscode> & C:/Users/seren/AppData/Local/Programs/Python/Python312/python.exe
 c:/Users/seren/OneDrive/Documents/vscode/Prototype/ECCAESGCM.py
Ciphertext: 91af10c47781c8b0dcdd5432d999b28bcadcebda18040f8dab2e78e46339aeadfe29ebc8005b93fb50f0aea7749e877bc279
d7d4d03f2ccb5627a98da79f14de7ca99d03
Decrypted Data: Hello, this is a hybrid ECC-AES GCM encryption demo!
```

Implementing CPUProfile and memory usage:

To find out the memory usage as well as the time taken we devised a program to calculate this

- We first import the main demo functions
- Then for each type of hybrid encryption we created a profile
- With each having its own starting time and ending time variable
- And executing the demo of the hybrid encryption
- Then also printing the total time it took
- In the main method this profile would execute result in the following

```
import time
from memory_profiler import profile
import AESECCEAX
import RSAECCEAX
import ECCAESGCM

@profile
def profile_AESECCEAX_hybrid_encryption():
    start_time = time.time()
    AESECCEAX.hybrid_encryption_demo()
    end_time = time.time()
    print(f"AESECCEAX Total Execution Time: {end_time - start_time:.4f} seconds")

@profile
def profile_RSAEECEAX_hybrid_encryption():
    start_time = time.time()
    RSAECCEAX.hybrid_encryption_demo()
    end_time = time.time()
    print(f"RSAEECEAX Total Execution Time: {end_time - start_time:.4f} seconds")

@profile
def profile_ECCAESGCM_hybrid_encryption():
    start_time = time.time()
    ECCAESGCM.hybrid_encryption_demo()
    end_time = time.time()
    print(f"ECCAESGCM Total Execution Time: {end_time - start_time:.4f} seconds")

if __name__ == "__main__":
    profile_AESECCEAX_hybrid_encryption()
    profile_RSAEECEAX_hybrid_encryption()
    profile_ECCAESGCM_hybrid_encryption()
```

Note: the code is available at https://github.com/B00177451/Prototype-LightWeight-Cryptography

CHAPTER 4

EVALUATIONS

As the lightweight cryptographic ciphers should be smaller in memory size as well as be able to run faster with lower energy drain if possible.

The previous code which was created in python we were able to execute all demo functions for the encryption and decryption method we were focusing on and the output for each of them is like this.

Here in all the demos the plaintext are set to be same to get comparison results

1. **AES-ECC-EAX output**

    - From this we get that the memory usage it started with was 30.4 MB
    - And during processing of the demo it was 1.8 MB
    - The total amount of time it took was 0.0000, this result could not be quanitified as it was in microseconds

```
Encrypted Data: 949a42135cb54c2b270bb5c71be67d1242b070e983c11b1aef72c2b2f8ff0663
Decrypted Data: Hello, this is a demo
AESECCEAX Total Execution Time: 0.0000 seconds
Filename: c:\Users\seren\OneDrive\Documents\vscode\Prototype\CPUprofile.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     7     30.4 MiB     30.4 MiB           1   @profile
     8                                         def profile_AESECCEAX_hybrid_encryption():
     9     30.4 MiB      0.0 MiB           1       start_time = time.time()
    10     32.3 MiB      1.8 MiB           1       AESECCEAX.hybrid_encryption_demo()
    11     32.3 MiB      0.0 MiB           1       end_time = time.time()
    12     32.3 MiB      0.0 MiB           1       print(f"AESECCEAX Total Execution Time: {end_time - start_time:.4f} seconds")
```

2. **RSA-ECC-EAX output**

    - from this we get that the memory usage it started with was 32.3 MB
    - and during the processing of demo it was 0.4 MB
    - the total amount of time it took was 1.3117 seconds

```
Ciphertext: c860960514c4acda26e7ae19ad2778ca798a36d68e
Decrypted Data: Hello, this is a demo
RSAEECEAX Total Execution Time: 1.3117 seconds
Filename: c:\Users\seren\OneDrive\Documents\vscode\Prototype\CPUprofile.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    14     32.3 MiB     32.3 MiB           1   @profile
    15                                         def profile_RSAEECEAX_hybrid_encryption():
    16     32.3 MiB      0.0 MiB           1       start_time = time.time()
    17     32.8 MiB      0.4 MiB           1       RSAECCEAX.hybrid_encryption_demo()
    18     32.8 MiB      0.0 MiB           1       end_time = time.time()
    19     32.8 MiB      0.0 MiB           1       print(f"RSAEECEAX Total Execution Time: {end_time - start_time:.4f} seconds")
```

3. ECC-AES-GCM output

- from this we get that the memory usage it started with 32.8 MB
- and during the processing of demo it was 0.1 MB
- the total amount of time it took for the demo was 0.0010 seconds.

```
Ciphertext: 8ace72c1e73581ca02afb0b10fc26776f1fd97c48d5dd62b5bc449579bc8e893a8c50b7204
Decrypted Data: Hello, this is a demo
ECCAESGCM Total Execution Time: 0.0010 seconds
Filename: c:\Users\seren\OneDrive\Documents\vscode\Prototype\CPUprofile.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
    21     32.8 MiB     32.8 MiB           1   @profile
    22                                         def profile_ECCAESGCM_hybrid_encryption():
    23     32.8 MiB      0.0 MiB           1       start_time = time.time()
    24     32.9 MiB      0.1 MiB           1       ECCAESGCM.hybrid_encryption_demo()
    25     32.9 MiB      0.0 MiB           1       end_time = time.time()
    26     32.9 MiB      0.0 MiB           1       print(f"ECCAESGCM Total Execution Time: {end_time - start_time:.4f} seconds")
```

Comparison result:

If we compare all three we realize that the least amount of time taken by any of the three was from AES-ECC-EAX method about 0.0000 seconds

And the least memory used by any of the three was from ECC-AES-GCM about 0.1 MB

Hybrid Encryption type	Memory used	Time taken
AES-ECC-EAX	1.8 MB	0.000 seconds
RSA-ECC-EAX	0.4 MB	1.3117 seconds
AES-ECC-GCM	0.1 MB	0.0010 seconds

From the above table we can figure out that the sweet spot between all three of them is **AES-ECC-GCM** hybrid encryption method where the amount of time and memory used is the comparatively better than others.

CHAPTER 5

CONCLUSION

In conclusion, AES-ECC-GCM is better in efficiency of processing and memory used when compared to AES-RSA-EAX and AES-ECC-EAX, with the basis of this we can figure out why this type of hybrid encryption is used mostly in IoT devices. And is the preferred choice for modern IoT systems as it balances both performance and security with efficient resource utilization.

Because of its parallel processing in GCM mode, which allows for allowing encryption and authentication to run simultaneously, which results in faster speeds and high throughput. This can also be scaled to work in large IoT deployments due to its small key size and GCM's ability to handle simultaneous loads.

This can also be used in real-time IoT applications, this is due to is low latency resulted from GCM's high speed encryption and decryption.

Some examples where this can be used:

- Autonomous vehicles and machines
- Smart surveillance systems
- Industrial IoT systems for real-time monitoring of sensors

This encryption method is also highly compatible with modern processors and accelerators like intel processors with AES-NI extensions and ARM cortex processors which is mostly found in IoT device

CHAPTER 6

REFERENCES

6.1 Literature References

[1] R. Imam, Q. M. Areeb, A. Alturki, and F. Anwer, "Systematic and Critical Review of RSA Based Public Key Cryptographic Schemes: Past and Present Status," *IEEE Access*, vol. 9, pp. 155949–155976, 2021, doi: 10.1109/ACCESS.2021.3129224.

[2] A. Overmars, "Survey of RSA Vulnerabilities," Jun. 2019, doi: 10.5772/intechopen.84852.

[3] S. M. Y. Z. VAHDATI, A. Ghasempour, and M. Salehi, "COMPARISON OF ECC AND RSA ALGORITHMS IN IOT DEVICES," 2019. Accessed: Dec. 21, 2024. [Online]. Available: https://www.semanticscholar.org/paper/COMPARISON-OF-ECC-AND-RSA-ALGORITHMS-IN-IOT-DEVICES-VAHDATI-Ghasempour/a43da6aa57ca8dbeeeb70c144a25b0f288caa8bb

[4] Y. Yan, "The Overview of Elliptic Curve Cryptography (ECC)," *J. Phys.: Conf. Ser.*, vol. 2386, no. 1, p. 012019, Dec. 2022, doi: 10.1088/1742-6596/2386/1/012019.

[5] M. Bellare, P. Rogaway, and D. Wagner, "A Conventional Authenticated-Encryption Mode," 2003. Accessed: Dec. 21, 2024. [Online]. Available: https://www.semanticscholar.org/paper/A-Conventional-Authenticated-Encryption-Mode-Bellare-Rogaway/a72d4dc6f0e84addeab2edb926ec2c7bc99b4442

[6] N. Ahmad, L. M. Wei, and M. H. Jabbar, "Advanced Encryption Standard with Galois Counter Mode using Field Programmable Gate Array.," *J. Phys.: Conf. Ser.*, vol. 1019, no. 1, p. 012008, Jun. 2018, doi: 10.1088/1742-6596/1019/1/012008.

[7] S. M. Y. Z. VAHDATI, A. Ghasempour, and M. Salehi, "COMPARISON OF ECC AND RSA ALGORITHMS IN IOT DEVICES," 2019. Accessed: Dec. 21, 2024. [Online]. Available: https://www.semanticscholar.org/paper/COMPARISON-OF-ECC-AND-RSA-ALGORITHMS-IN-IOT-DEVICES-VAHDATI-Ghasempour/a43da6aa57ca8dbeeeb70c144a25b0f288caa8bb

[8] P. Kuppuswamy and S. Q. Y. A. Khalidi, "Hybrid encryption/decryption technique using new public key and symmetric key algorithm," *IJICS*, vol. 6, no. 4, p. 372, 2014, doi: 10.1504/IJICS.2014.068103.

[9] Q. Zhang, "An Overview and Analysis of Hybrid Encryption: The Combination of Symmetric Encryption and Asymmetric Encryption," *2021 2nd International Conference on Computing and Data Science (CDS)*, pp. 616–622, Jan. 2021, doi: 10.1109/CDS52072.2021.00111.

[10] O. Popoola, M. A. Rodrigues, J. Marchang, A. Shenfield, A. Ikpehai, and J. Popoola, "An optimized hybrid encryption framework for smart home healthcare: Ensuring data confidentiality and security," *Internet of Things*, vol. 27, p. 101314, Oct. 2024, doi: 10.1016/j.iot.2024.101314.

6.2 Code and web references

[1] https://github.com/xbeat/Machine-Learning/blob/main/Cryptography%20with%20Python%20Essentials.md
[2] https://www.programiz.com/python-programming/time
[3] https://pypi.org/project/memory-profiler/
[4] https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution