Department of Electronic & Telecommunication Engineering
University of Moratuwa

EN4720 - Security in Cyber-Physical Systems

Project - Milestone II
Cryptographic API Implementation

23rd March 2025

Amarasekara A.T.P.	200023C
Bandara D.M.D.V	200061N
Samarasekera A.M.P.S.	200558U
Wijetunga W.L.N.K.	200733D

1 Introduction

In Milestone II, the objective was to develop APIs for both symmetric and asymmetric encryption and decryption, as well as APIs for hashing and verifying hashed tokens (digests). For the development, we used FastAPI, and the Cryptography library, which provides standard cryptographic functions. Testing was initially conducted using Postman API while system was hosted locally, and it is then deployed on,

https://pase.pythonanywhere.com

This API will remain available until June 23rd, 2025, under the free three-month hosting period.

2 Design of the System

2.1 Generate Hash

The Hashing API generates a secure digest of input data using a cryptographic hash function. SHA-512 was chosen over SHA-256 because it offers a longer hash (512 bits) and better resistance to brute-force attacks. However user can choose and check both algorithms.

```
    url_genhash = f"{BASE_URL}/generate-hash"

    data = "You are a wizard, Harry!"
    hashing_algorithm = "SHA-256"

    response = requests.post(url_genhash, json= {"data": data,
       "algorithm": hashing_algorithm})
    print(response.json())
```

Listing 1: sample hashing request

2.2 Verify Hash

This API verifies whether a given plaintext corresponds to a previously generated hash by comparing the hash of the input data with the stored hash. The output will be a boolean indicating validity.

```
    url_verify_hash = f"{BASE_URL}/verify-hash"

    data = "You are a muggle, Dudley!"
    hash_value = "hwF6QB5z7ddmj5/0qaBx0iuo8riY57mAZc1z0MTZ4c8="
    hashing_algorithm = "SHA-256"

    response = requests.post(url_verify_hash, json= {"data": data,
       "hash_value": hash_value, "algorithm": hashing_algorithm})
    print(response.json())
```

Listing 2: sample hashing verification

2.3 Key Generation

- **AES Key Generation**: A single symmetric key is generated for AES encryption. This key is used for both encryption and decryption, and the key size can be selected as 128, 192, or 256 bits.
- **RSA Key Generation**: RSA generate two keys: a public key for encryption and a private key for decryption. RSA key sizes are 2048, 3072, and 4096 bits. RSA key sizes of 512 and 1024 bits are considered insecure due to advances in computational power, making them vulnerable to factorization attacks. Hence key sizes 512 and 1024 are excluded.

Only these two types of KEY generation algorithms and KEY sizes are allowed.

```
    url_keygen = f"{BASE_URL}/generate-key"

    key_type = "RSA"
    key_size = 4096

    response = requests.post(url_keygen, json = {"key_type":
       key_type, "key_size": key_size})
    print(response.json())
```

Listing 3: sample key generation request

You will be able to receive *key_id* and *key*. AES will provide one private key, while RSA will provide two keys: one public and one private.

2.4 Encryption and Decryption

Two encryption methods were implemented which are described below.

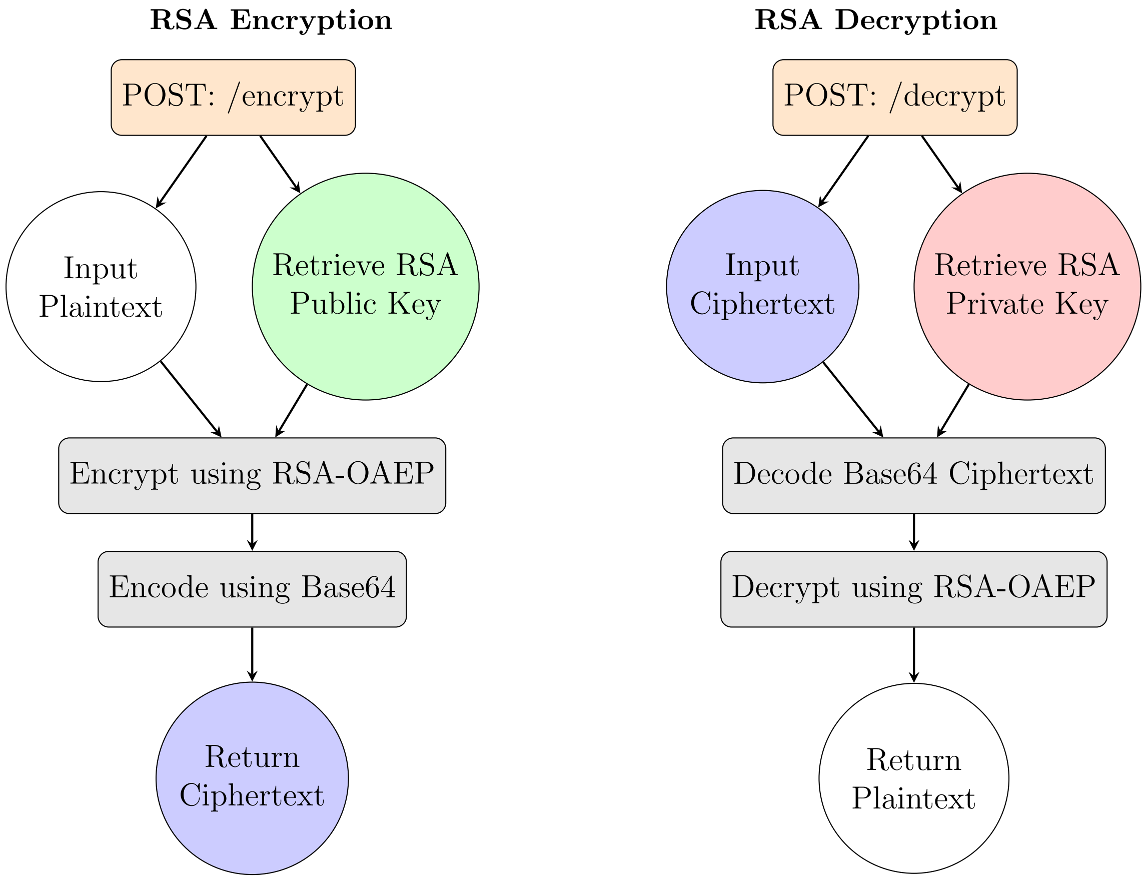
- **AES (Advanced Encryption Standard):** Uses a symmetric key for fast encryption and decryption. AES-CBC was used instead of ECB due to its ability to provide better diffusion properties.
- **RSA (Rivest-Shamir-Adleman):** Uses a public key for encryption and a private key for decryption. $N = 655537 = 17 \times 38561$ which are both prime, was chosen. *(Note that this is a commonly used N value and we can always change it to make the encryption more secure.)* The decision to use RSA-OAEP padding was made to prevent chosen-ciphertext attacks and add randomness, making encryption more secure than traditional PKCS#1 v1.5 padding [1, 2].

  1. **PEM(Privacy-Enhanced Mail) for Encoding Keys**:
     - **PEM** is an encoding format which uses Base64 encoding to represent binary data in a text format and it is used for encoding keys in this implementation. The reasons to use it are as follows.
       * PEM is supported by many cryptographic libraries and tools such as OpenSSL and Python's cryptography library which is used in this implementation.
       * Since PEM is a text-based format, it makes easy for humans to read, share, and debug. It uses delimiters like (`-----BEGIN PRIVATE KEY-----`) which are human-readable.

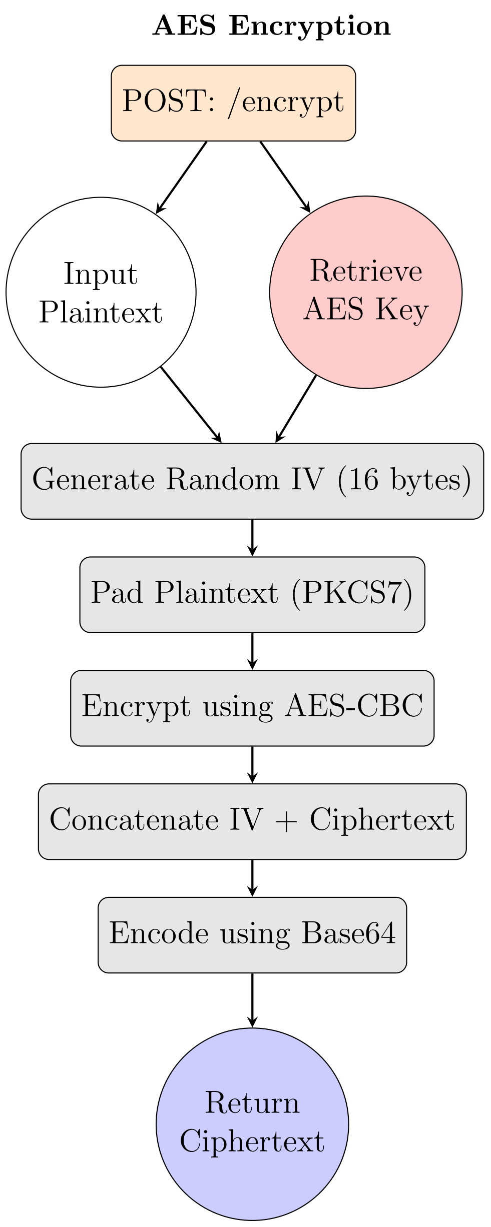
        * PEM-encoded keys can be easily embedded in configuration files and environment variables, and can be easily transmitted over text-based protocols like HTTP.

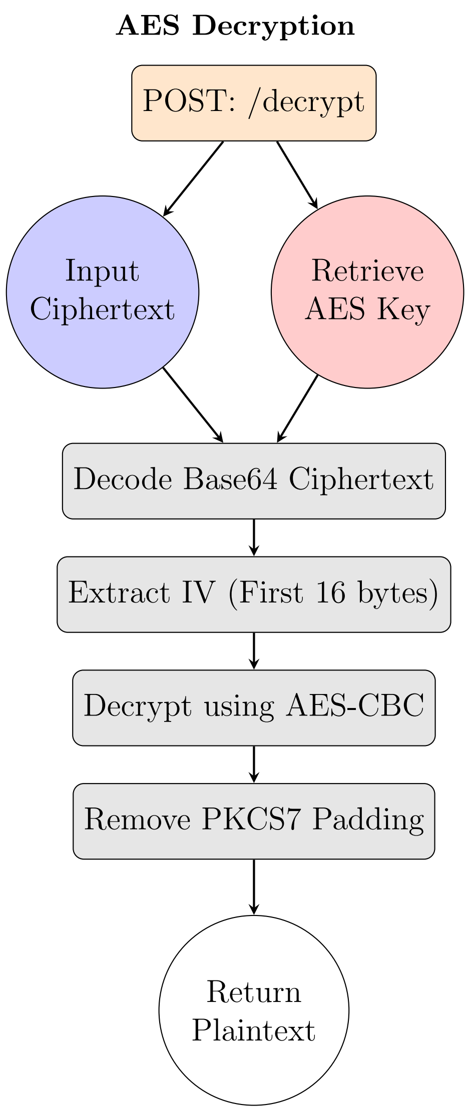
2. **Public-Key Cryptography Standards #8 for Private Key Storage**
    - **PKCS#8** is used for storing private keys in this implementation. The reasons are as follows.
        * PKCS#8 is a widely used standard for storing private key information which is supported by many cryptographic libraries and tools.
        * PKCS#8 supports both encrypted and unencrypted private key storage which makes it suitable for different security requirement.
        * Since PKCS#8 has a clear structure for storing private key data, including metadata, valuable metadata such as the algorithm used and any attributes can also be stored in an organized manner.

RSA Encryption & Decryption Flow



AES Encryption & Decryption Flow

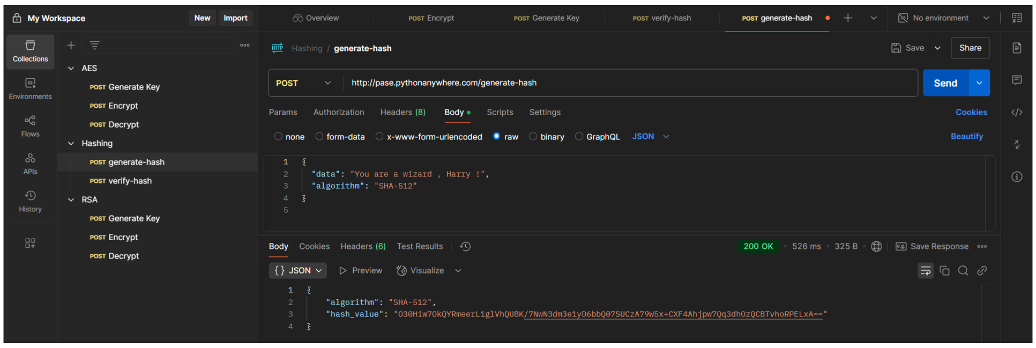




3 Testing of the System

The system was tested both locally and globally using Postman. Further testing was also conducted using a Python notebook developed for it, which is available here.

3.1 Hashing and Verifying



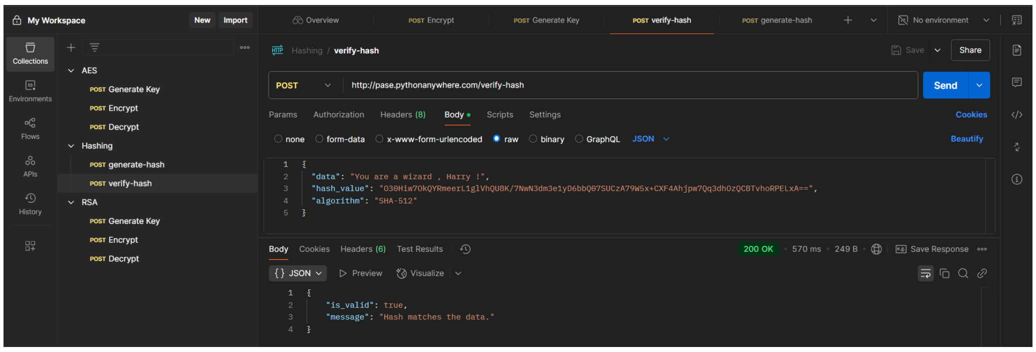
POST http://pase.pythonanywhere.com/generate-hash

```
{
  "data": "You are a wizard , Harry !",
  "algorithm": "SHA-512"
}
```

200 OK · 526 ms · 325 B

```
{
    "algorithm": "SHA-512",
    "hash_value": "O30Hiw7OkQYRmeerL1glVhQU8K/7NwN3dm3e1yD6bbQ07SUCzA79W5x+CXF4Ahjpw7Qq3dhOzQCBTvhoRPELxA=="
}
```

(a) Hashing (digesting) the plaintext (algorithm = "SHA-512")



POST http://pase.pythonanywhere.com/verify-hash

```
{
  "data": "You are a wizard , Harry !",
  "hash_value": "O30Hiw7OkQYRmeerL1glVhQU8K/7NwN3dm3e1yD6bbQ07SUCzA79W5x+CXF4Ahjpw7Qq3dhOzQCBTvhoRPELxA==",
  "algorithm": "SHA-512"
}
```

200 OK · 570 ms · 249 B

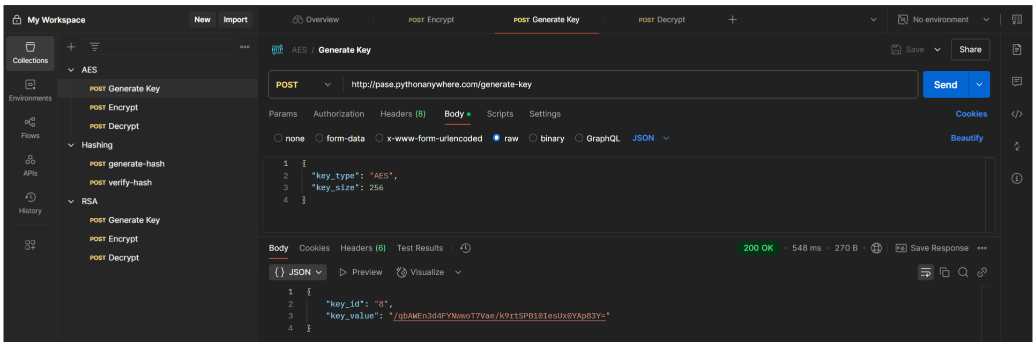
```
{
    "is_valid": true,
    "message": "Hash matches the data."
}
```

(b) Verifying the digest with the plaintext (algorithm = "SHA-512")

Figure 1: Hashing and Verifying

3.2 Key Generation, Encryption and Decryption

3.2.1 "AES" key type



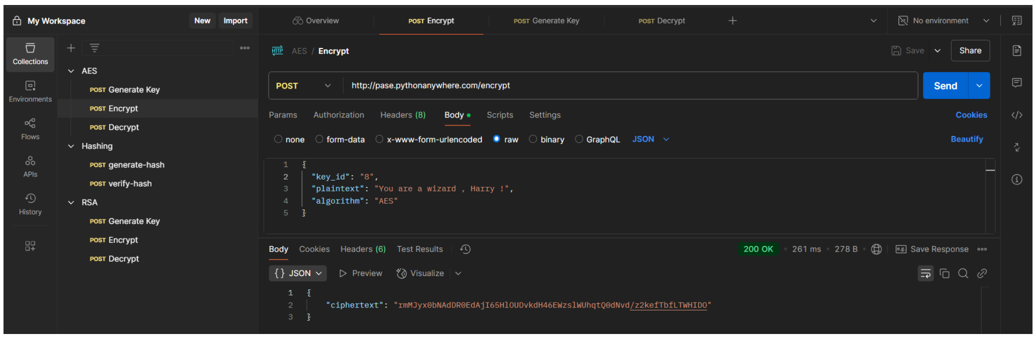
POST http://pase.pythonanywhere.com/generate-key

```
{
  "key_type": "AES",
  "key_size": 256
}
```

200 OK · 548 ms · 270 B

```
{
    "key_id": "8",
    "key_value": "/qbAWEn3d4FYNwwoT7Vae/k9rtSPB10IesUx0YAp83Y="
}
```

Figure 2: AES key generation (eg: key id = 8)

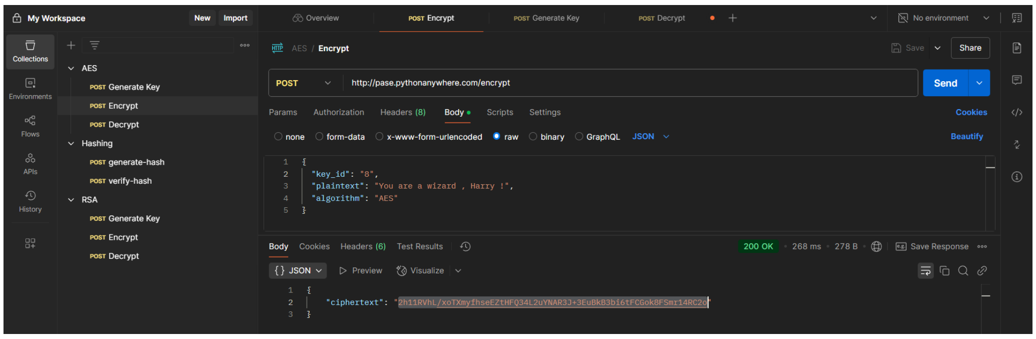

POST http://pase.pythonanywhere.com/encrypt

```
{
  "key_id": "8",
  "plaintext": "You are a wizard , Harry !",
  "algorithm": "AES"
}
```

200 OK · 261 ms · 278 B

```
{
    "ciphertext": "rmMJyx0bNAdDR0EdAjI65HlOUDvkdH46EWzslWUhqtQ0dNvd/z2kefTbfLTWHIDO"
}
```

(a) First ciphertext (key-id = 8 and algorithm = "AES")


POST http://pase.pythonanywhere.com/encrypt

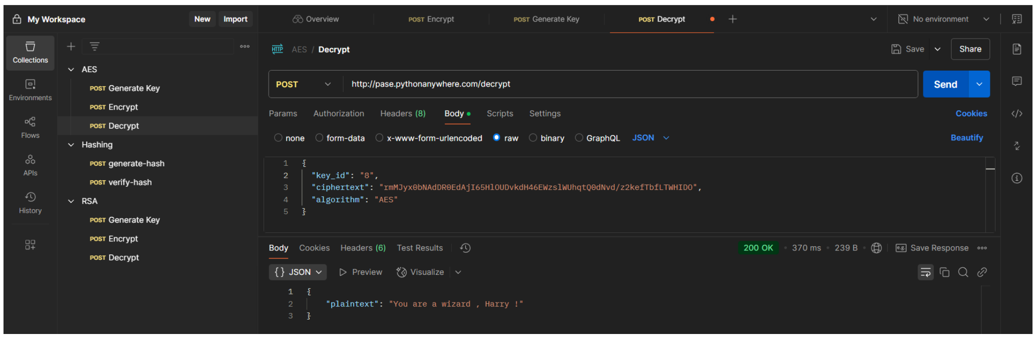
```
{
  "key_id": "8",
  "plaintext": "You are a wizard , Harry !",
  "algorithm": "AES"
}
```

200 OK · 268 ms · 278 B

```
{
    "ciphertext": "2h11RVhL/xoTXmyfhseEZtHFQ34L2uYNAR3J+3EuBkB3bi6tFCGok8FSmr14RC2o"
}
```

(b) Second ciphertext (key-id = 8 and algorithm = "AES")

Figure 3: AES Encryption with the same key and plaintext to obtain two ciphertexts

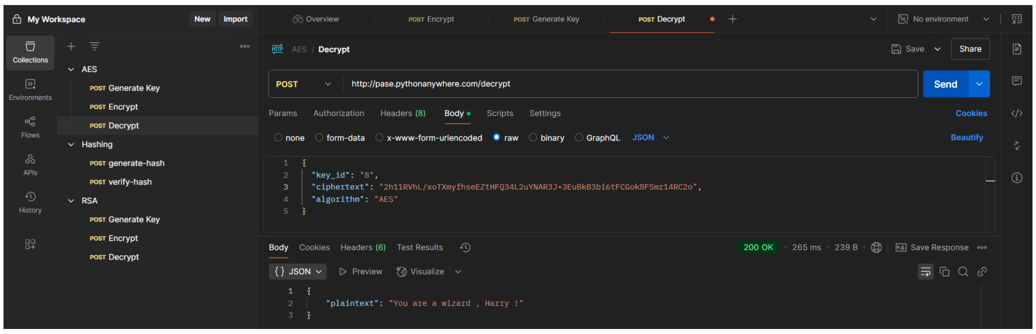

POST http://pase.pythonanywhere.com/decrypt

```
{
  "key_id": "8",
  "ciphertext": "rmMJyx0bNAdDR0EdAjI65HlOUDvkdH46EWzslWUhqtQ0dNvd/z2kefTbfLTWHIDO",
  "algorithm": "AES"
}
```

200 OK · 370 ms · 239 B

```
{
    "plaintext": "You are a wizard , Harry !"
}
```

(a) First ciphertext decryption (key-id = 8 and algorithm = "AES")


POST http://pase.pythonanywhere.com/decrypt

```
{
  "key_id": "8",
  "ciphertext": "2h11RVhL/xoTXmyfhseEZtHFQ34L2uYNAR3J+3EuBkB3bi6tFCGok8FSmr14RC2o",
  "algorithm": "AES"
}
```

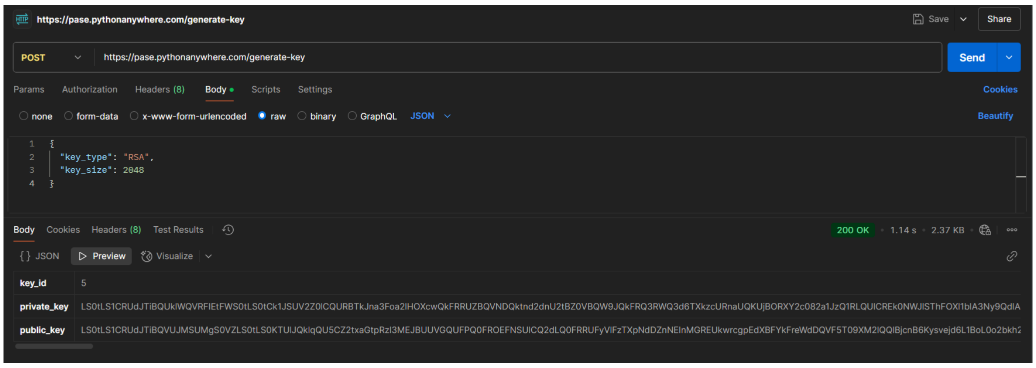
200 OK · 265 ms · 239 B

```
{
    "plaintext": "You are a wizard , Harry !"
}
```

(b) Second ciphertext decryption (key-id = 8 and algorithm = "AES")

Figure 4: Both ciphertexts providing same plaintext using the same key

3.2.2 “RSA” key type



POST https://pase.pythonanywhere.com/generate-key

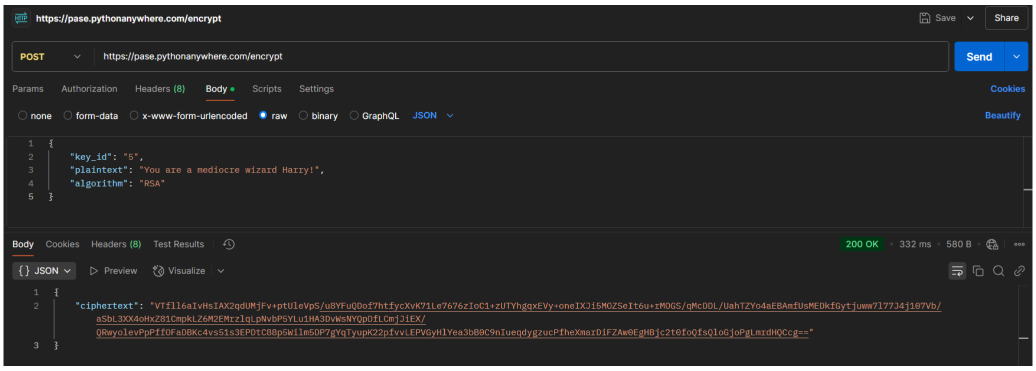
raw JSON

```
{
  "key_type": "RSA",
  "key_size": 2048
}
```

200 OK · 1.14 s · 2.37 KB

key_id	5
private_key	LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1JSUV2Z0lCQURBTkJna3Foa2lHOXcwQkFRRUZBQVNDQktnd2dnU2tBZ0VBQW9JQkFRQ3RWQ3d6TXkzcURnaUQKUjBORXY2c082a1JzQ1RLQUlCREk0NWJlSThFOXl1blA3Ny9QdlA
public_key	LS0tLS1CRUdJTiBQVUJMSUMgS0VZLS0tLS0KTUlJQklqQU5CZ2txaGtpRzl3MEJBUUVGQUFPQ0FROEFNSUlCQ2dLQ0FRRUFyVlFzTXpNdDZnNElnMGREUkwrcgpEdXBFYkFreWdDQVF5T09XM2lQQlBjcnB6Kysvejd6L1BoL0o2bkh2

Figure 5: RSA key generation (eg: key id = 5)



POST https://pase.pythonanywhere.com/encrypt

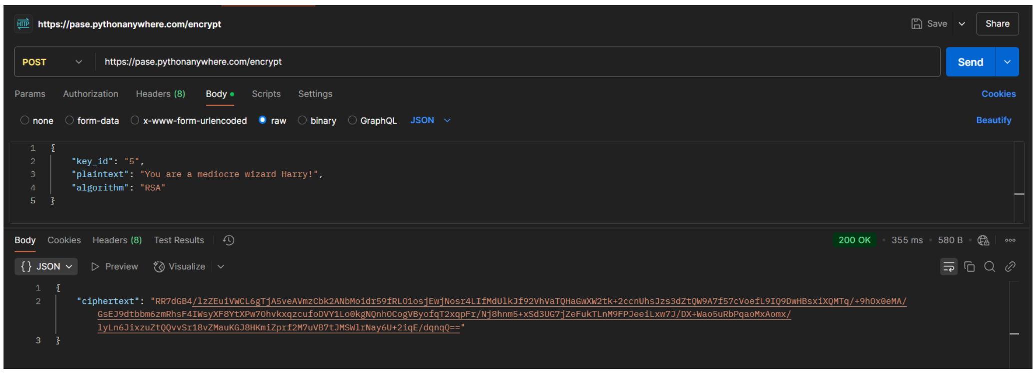
raw JSON

```
{
    "key_id": "5",
    "plaintext": "You are a mediocre wizard Harry!",
    "algorithm": "RSA"
}
```

200 OK · 332 ms · 580 B

```
{
    "ciphertext": "VTfll6aIvHsIAX2qdUMjFv+ptUleVpS/u8YFuQDof7htfycXvK71Le7676zIoC1+zUTYhgqxEVy+oneIXJi5MOZSeIt6u+rMOGS/qMcDDL/UahTZYo4aEBAmfUsMEDkfGytjuww7l77J4j107Vb/
        aSbL3XX4oHxZ81CmpkLZ6M2EMrzlqLpNvbP5YLu1HA3DvWsNYQpDfLCmjJiEX/
        QRwyolevPpPffOFaDBKc4vs51s3EPDtC88p5Wilm5DP7gYqTyupK22pfvvLEPVGyHlYea3bB0C9nIueqdygzucPfheXmarDiFZAw0EgHBjc2t0foQfsQloGjoPgLmrdHQCcg=="
}
```

(a) First ciphertext (key-id = 5 and algorithm = “RSA”)



POST https://pase.pythonanywhere.com/encrypt

raw JSON

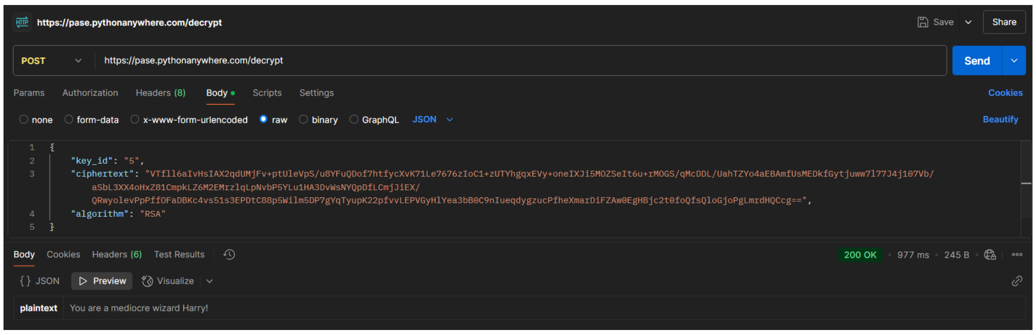
```
{
    "key_id": "5",
    "plaintext": "You are a mediocre wizard Harry!",
    "algorithm": "RSA"
}
```

200 OK · 355 ms · 580 B

```
{
    "ciphertext": "RR7dGB4/lzZEuiVWCL6gTjA5veAVmzCbk2ANbMoidr59fRLO1osjEwjNosr4LIfMdUlkJf92VhVaTQHaGwXW2tk+2ccnUhsJzs3dZtQW9A7f57cVoefL9IQ9DwHBsxiXQMTq/+9hOx0eMA/
        GsEJ9dtbbm6zmRhsF4IWsyXF8YtXPw7OhvkxqzcufoDVY1Lo0kgNQnhOCogVByofqT2xqpFr/Nj8hnm5+xSd3UG7jZeFukTLnM9FPJeeiLxw7J/DX+Wao5uRbPqaoMxAomx/
        lyLn6JixzuZtQQvvSr18vZMauKGJ8HKmiZprf2M7uVB7tJMSWlrNay6U+2iqE/dqnqQ=="
}
```

(b) Second ciphertext (key-id = 5 and algorithm = “RSA”)

Figure 6: RSA Encryption with the same key and plaintext to obtain two ciphertext



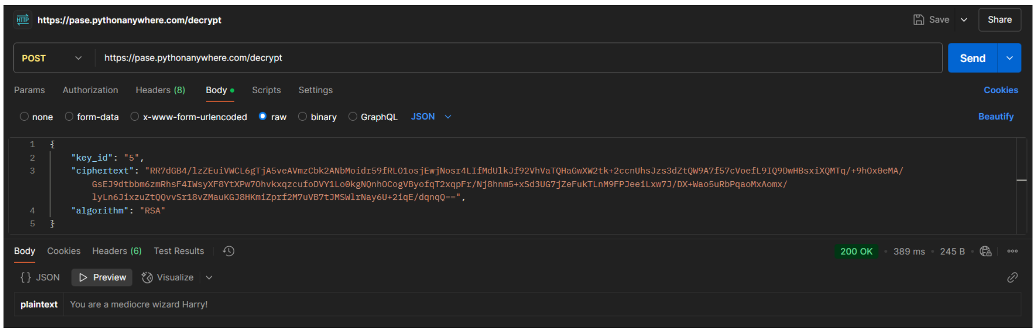
POST https://pase.pythonanywhere.com/decrypt

```
{
    "key_id": "5",
    "ciphertext": "VTfll6aIvHsIAX2qdUMjFv+ptUleVpS/u8YFuQDof7htfycXvK71Le7676zIoC1+zUTYhgqxEVy+oneIXJi5MOZSeIt6u+rMOGS/qMcDDL/UahTZYo4aEBAmfUsMEDkfGytjuww7l77J4j107Vb/
        aSbL3XX4oHxZ81CmpkLZ6M2EMrzlqLpNvbP5YLu1HA3DvWsNYQpDfLCmjJiEX/
        QRwyolevPpPffOFaDBKc4vs51s3EPDtC88p5Wilm5DP7gYqTyupK22pfvvLEPVGyHlYea3bB0C9nIueqdygzucPfheXmarDiFZAw0EgHBjc2t0foQfsQloGjoPgLmrdHQCcg==",
    "algorithm": "RSA"
}
```

200 OK · 977 ms · 245 B

plaintext: You are a mediocre wizard Harry!

(a) First ciphertext decryption (key-id = 5 and algorithm = "RSA")



POST https://pase.pythonanywhere.com/decrypt

```
{
    "key_id": "5",
    "ciphertext": "RR7dGB4/lzZEuiVWCL6gTjA5veAVmzCbk2ANbMoidr59fRLO1osjEwjNosr4LIfMdUlkJf92VhVaTQHaGwXW2tk+2ccnUhsJzs3dZtQW9A7f57cVoefL9IQ9DwHBsxiXQMTq/+9hOx0eMA/
        GsEJ9dtbbm6zmRhsF4IWsyXF8YtXPw7OhvkxqzcufoDVY1Lo0kgNQnhOCogVByofqT2xqpFr/Nj8hnm5+xSd3UG7jZeFukTLnM9FPJeeiLxw7J/DX+Wao5uRbPqaoMxAomx/
        lyLn6JixzuZtQQvvSr18vZMauKGJ8HKmiZprf2M7uVB7tJMSWlrNay6U+2iqE/dqnqQ==",
    "algorithm": "RSA"
}
```

200 OK · 389 ms · 245 B

plaintext: You are a mediocre wizard Harry!

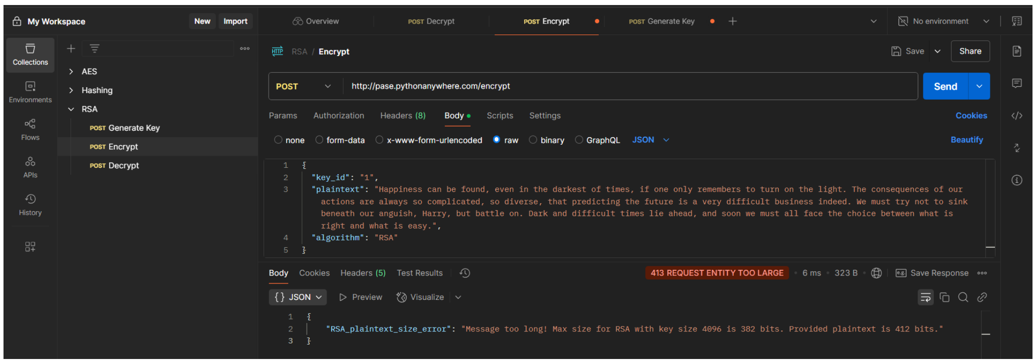
(b) Second ciphertext decryption (key-id = 5 and algorithm = "RSA")

Figure 7: Both ciphertexts providing same plaintext using the same key

3.3 Error Handling

Upon testing the system, several errors were identified, and the following error messages were generated accordingly:

1. Status code: 400 ("Bad Request")
    - Unsupported key type provided in key generation.
    - Invalid key size specified for AES or RSA key generation.
    - Unsupported encryption algorithm specified.
    - Unsupported decryption algorithm specified.
    - Unsupported hashing algorithm specified.
2. Status code: 404 ("Not Found")
    - Key ID not found in the key storage.
3. Status code: 413 ("Content Too Large")
    - Request payload exceeds the allowed size limit. RSA encryption have a limit on the size of plaintext that can be encrypted depending on the key size. Hence this error was specially handled.



POST http://pase.pythonanywhere.com/encrypt

none form-data x-www-form-urlencoded [✓ raw] binary GraphQL JSON

```
{
  "key_id": "1",
  "plaintext": "Happiness can be found, even in the darkest of times, if one only remembers to turn on the light. The consequences of our
    actions are always so complicated, so diverse, that predicting the future is a very difficult business indeed. We must try not to sink
    beneath our anguish, Harry, but battle on. Dark and difficult times lie ahead, and soon we must all face the choice between what is
    right and what is easy.",
  "algorithm": "RSA"
}
```

413 REQUEST ENTITY TOO LARGE · 6 ms · 323 B

```
{
  "RSA_plaintext_size_error": "Message too long! Max size for RSA with key size 4096 is 382 bits. Provided plaintext is 412 bits."
}
```

Figure 8: Error: Plaintext is too large for RSA algorithm and key size.

4. Status code: 500 (“Internal Server Error”)
   - Key generation failed due to an internal unexpected error.
   - Encryption process encountered an unexpected error.
   - Decryption process encountered an unexpected error.
   - Hash generation process encountered an unexpected error.
   - Hash verification process encountered an unexpected error.

4 Web Hosting

The original application was built with **FastAPI** [3], but was changed to **Flask** [4] to easily deploy on the server. For hosting we looked into Render and pythonanywhere, and chose pythonanywhere due to its free availability.

Once the app was programmed using Flask, the Flask app files, were uploaded to the pythonanywhere file system. Then a virtual environment was created to install the required dependencies. Finally, using the pythonanywhere web app configuration page, it was set up with the required WSGI configuration to connect the app to the web server. Then testing was done using the URL on both Postman and a python notebook generated for testing purposes.

4.1 Accessing the API

The codes used for the Milestone II are available in the following **Github Repository**.

`https://github.com/D-Vinod/Cryptographic-API-Implementation`

The API can be accessed using the base URL `https://pase.pythonanywhere.com` with the following for to POST requests.

- Key generation: *https://pase.pythonanywhere.com/generate-key*
- Encryption: *https://pase.pythonanywhere.com/encrypt*
- Decryption:*https://pase.pythonanywhere.com/decrypt*

- Hashing: *https://pase.pythonanywhere.com/generate-hash*
- Verify hash: *https://pase.pythonanywhere.com/verify-hash*

This Jupyter Notebook can be used for testing as well.

4.1.1 Generate a Hash

POST /generate-hash: Creates a hash digest using SHA-256 or SHA-512.

Request Body

```
{
  "data": "Hello, world!",
  "algorithm": "SHA-256" # OR "SHA-512"
}
```

Listing 4: Sample hash generation request

4.1.2 Verify a Hash

POST /verify-hash: Verifies if the given plaintext matches a previously generated hash.

Request Body

```
{
  "data": "Hello, world!",
  "hash_value": "...previously generated hash...", # use correct
     digest with the data
  "algorithm": "SHA-256" # OR "SHA-512"
}
```

Listing 5: Sample hash verification request

4.1.3 Generate a Key

POST /generate-key: Generates an AES or RSA key.

Request Body

```
{
  "key_type": "RSA", # OR AES
  "key_size": 2048 # Any valid size mentioned above
}
```

Listing 6: Sample key generation request

4.1.4 Encrypt Data

POST /encrypt: Encrypts a plaintext using AES or RSA.

Request Body (AES Example)

```
{
  "key_id": "1", # Change according to generated key ID
  "plaintext": "Hello, world!",
  "algorithm": "AES" # OR RSA
}
```

Listing 7: Sample encryption request

4.1.5 Decrypt Data

POST /decrypt: Decrypts a ciphertext using AES or RSA.

Request Body (AES Example)

```
{
  "key_id": "1", # Change according to generated key ID
  "ciphertext": "...base64 encoded ciphertext...", # Use generated
      ciphertext
  "algorithm": "AES" # OR RSA
}
```

Listing 8: Sample decryption request

5 References

References

[1] "SSL Certificate Formats - PEM, PFX, KEY, DER, CSR, P7B etc." SSLmentor, 2020. [Online]. Available: https://www.sslmentor.com/help/ssl-certificate-formats

[2] "A SSL Certificate File Extension Explanation: PEM, PKCS7, DER, and PKCS12," Comodo SSL Resources, 12 2019. [Online]. Available: https://comodosslstore.com/resources/a-ssl-certificate-file-extension-explanation-pem-pkcs7-der-and-pkcs12/

[3] "Tutorial - user guide - fastapi," fastapi.tiangolo.com. [Online]. Available: https://fastapi.tiangolo.com/tutorial/

[4] Flask, "Welcome to flask — flask documentation (3.0.x)," Palletsprojects.com. [Online]. Available: https://flask.palletsprojects.com/en/stable/