Implementation of Cryptographic Algorithms

Documentation

Antonette Robinson
620163576

PART A

Project Overview

The project aims to implement a simplified version of RSA encryption to secure the connection between a client and a server, and then to encrypt and decrypt messages between the client and server using a symmetric encryption implementation. Providing an understanding of how RSA and AES encryption algorithms work hand-in-hand to secure messages sent between a client and server.

How it works

The logic begins when the command python3 crypto_server.py 5001
(or any port number between 1024–49151) is executed in the terminal. The server starts up and prompts the user to enter two prime numbers within the required constraints. After these values are entered, the server uses them to generate the RSA keys, then begins listening for a client connection.

A connection is established when the client is started with the command: python3 crypto_client.py localhost 5001
Once this command is executed, the client connects to the server through the socket.

With the connection established, the server and client enter the communication loop that implements the cryptographic protocol. The client first sends a list of the encryption algorithms it supports. The server responds by selecting the symmetric and asymmetric algorithms it will use(AES for symmetric encryption and RSA for asymmetric encryption), and sends this information back to the client along with its RSA public key and a nonce.

From here, the client uses the RSA algorithm to securely exchange and verify sensitive data such as the session key and the encrypted nonce. Once both sides have successfully shared and confirmed the AES session key, the system switches to using the AES algorithm to encrypt the integers exchanged between the client and server. The server then computes the sum of the integers and returns it to the client, still using AES encryption.

Design Tradeoffs

The implementation uses small prime numbers to generate RSA keys.
This choice simplifies calculations and debugging, but it significantly reduces the security level compared to production RSA systems, which use primes that are hundreds of digits long.
Additonlally, the implementation contains only basic error handling

A more secure system would need extensive handling for malformed messages, replay attacks, unexpected data sizes, and timeouts.

Improvements

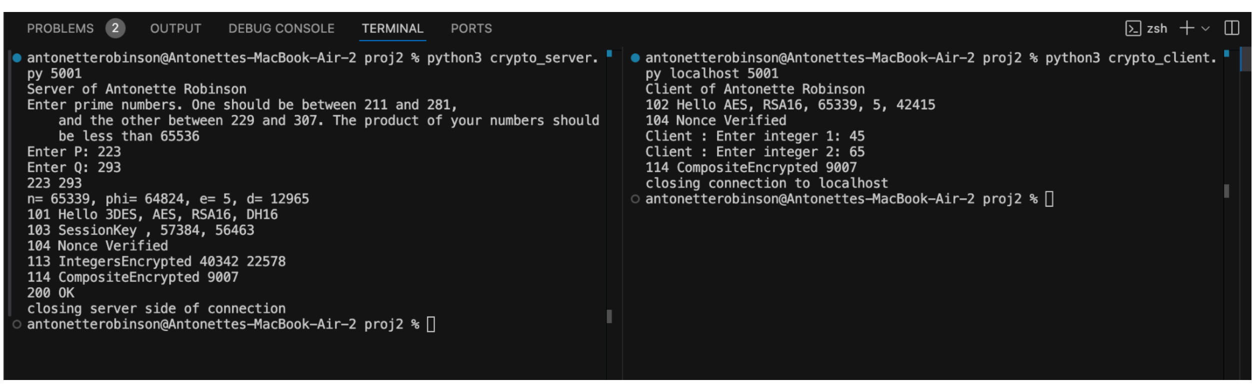
This system could be improved with a more advanced AES and RSA encryption algorithm. The system could also utilize a longer RSA key size for more secure data transmission. Improved error handling, including validation of received data and protection against replay or malformed messages, would further increase the reliability and robustness of the overall communication protocol.

Test Cases

The code was tested using two sets of prime numbers (223,293) and (211, 257). (223,293) ran successfully when two small integers were used. It is noticed, however, that when the integers are too large, the system will return a crash. This happens because a simplified AES implementation cannot handle values too large.
This is expected and normal for simplified AES, as it is designed only for 8-bit or 16-bit values, not large integers.

With the primes (211, 257) the server will not start and prompts the user to try other values because the product of the values must be less than 65536. This is also a successful test case as it shows that the error handling and logic is working properly.

Test case verified:
Sever(left) -Client(right)



Test case verified:

```
Enter P: 211
Enter Q: 257
Number not in range
```

Test case when int is too large:

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS                                   zsh
```

```
antonetterobinson@Antonettes-MacBook-Air-2 proj2 % python3 crypto_server.
py 5001
Server of Antonette Robinson
Enter prime numbers. One should be between 211 and 281,
    and the other between 229 and 307. The product of your numbers should
    be less than 65536
Enter P: 223
Enter Q: 293
223 293
n= 65339, phi= 64824, e= 5, d= 12965
101 Hello 3DES, AES, RSA16, DH16
103 SessionKey , 54202, 52714
104 Nonce Verified

Traceback (most recent call last):
  File "/Users/antonetterobinson/Downloads/proj2/crypto_server.py", line
236, in <module>
    main()
  File "/Users/antonetterobinson/Downloads/proj2/crypto_server.py", line
233, in main
    server.start()
  File "/Users/antonetterobinson/Downloads/proj2/crypto_server.py", line
172, in start
    int1 = self.AESdecrypt(int(parts[2]))
                               ~~~~~^^^

IndexError: list index out of range
antonetterobinson@Antonettes-MacBook-Air-2 proj2 % 
```

```
antonetterobinson@Antonettes-MacBook-Air-2 proj2 % python3 crypto_client.
py localhost 5001
Client of Antonette Robinson
102 Hello AES, RSA16, 65339, 5, 3815
104 Nonce Verified
Client : Enter integer 1: 500
Client : Enter integer 2: 100000
Traceback (most recent call last):
  File "/Users/antonetterobinson/Downloads/proj2/crypto_client.py", line
180, in <module>
    main()
  File "/Users/antonetterobinson/Downloads/proj2/crypto_client.py", line
175, in main
    client.start()
  File "/Users/antonetterobinson/Downloads/proj2/crypto_client.py", line
135, in start
    self.send(f"113 IntegersEncrypted {self.AESencrypt(int1)} {self.AESen
crypt(int2)}")
                                                              ^^^^^^^^^^
^^^^^^^^^^^
  File "/Users/antonetterobinson/Downloads/proj2/crypto_client.py", line
83, in AESencrypt
    ciphertext = simplified_AES.encrypt(plaintext) # Running simplified A
ES.
                 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/antonetterobinson/Downloads/proj2/simplified_AES.py", line
 78, in encrypt
    state = mixCol(shiftRow(sub4NibList(sBox, state)))
                            ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/Users/antonetterobinson/Downloads/proj2/simplified_AES.py", line
 51, in sub4NibList
    return [sbox[e] for e in s]
            ~~~~^^^
```

PART B

Generating and uploading my public key

To begin the process, a PGP keypair was generated on a macOS system using the GnuPG (GPG) tool with the command gpg --full-generate-key. The default key type (RSA and RSA) with a secure 4096-bit key size was selected. The user's full name and UWI email address were entered as the associated User ID, and a strong passphrase was created to protect the private key. After the keypair was generated, its details—including the key ID and fingerprint—were confirmed using gpg --list-keys.

The public key was then exported using the command gpg --armor --export "Antonette Robinson" > my_public_key.asc, producing a plaintext .asc file suitable for sharing. Finally, the public key was uploaded to the Ubuntu keyserver using gpg --keyserver hkp://keyserver.ubuntu.com:80 --send-keys <KEY_ID>.

```
Antonettes-MacBook-Air-2:~ antonetterobinson$ gpg --full-generate-key
gpg (GnuPG/MacGPG2) 2.2.41; Copyright (C) 2022 g10 Code GmbH
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
  (14) Existing key from card
Your selection? 1
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (3072) 4096
Requested keysize is 4096 bits
```

```
GnuPG needs to construct a user ID to identify your key.

Real name: Antonette Robinson
Email address: antonette.robinson@mymona.uwi.edu
Comment: Antonette
You selected this USER-ID:
    "Antonette Robinson (Antonette) <antonette.robinson@mymona.uwi.edu>"
```

```
Antonettes-MacBook-Air-2:~ antonetterobinson$ gpg --keyserver keyserver.ubuntu.com --send-keys 85E38F69046B44C1EC9FB07B76D78F0500D026C4
gpg: sending key 76D78F0500D026C4 to hkp://keyserver.ubuntu.com
```

Obtaining and verifying Classmates Keys

To obtain classmates' keys, their names were searched on the Ubuntu keyserver, and their public key files were downloaded. Verification was performed by comparing the fingerprint displayed on the local system with the fingerprint provided by each classmate, using the command gpg --fingerprint <THEIR_KEY_ID>. Matching fingerprints confirmed the authenticity of each key.

Signing and Re-Uploading the Keys

Once a classmate's key was verified, it was signed to indicate trust and confirm that the identity associated with the key had been validated. The signed key was then uploaded back to the Ubuntu keyserver so that others could see the added certification.

Evidence of signing the key

```
Antonettes-MacBook-Air-2:cryptography antonetterobinson$ gpg --sign-key 47b5712c4c71d120e60699016105b287a5af4dd1

pub  ed25519/6105B287A5AF4DD1
     created: 2025-11-25  expires: 2028-11-25  usage: SC
     trust: unknown       validity: unknown
sub  cv25519/8153CA7B3A619D4E
     created: 2025-11-25  expires: 2028-11-25  usage: E
[ unknown] (1). Christoff Cowan <christoff.cowan@mymona.uwi.edu>


pub  ed25519/6105B287A5AF4DD1
     created: 2025-11-25  expires: 2028-11-25  usage: SC
     trust: unknown       validity: unknown
 Primary key fingerprint: 47B5 712C 4C71 D120 E606  9901 6105 B287 A5AF 4DD1

     Christoff Cowan <christoff.cowan@mymona.uwi.edu>

This key is due to expire on 2028-11-25.
Are you sure that you want to sign this key with your
key "Antonette Robinson (Antonette) <antonette.robinson@mymona.uwi.edu>" (FA59CFDD3C2F178B)
```

Classmate 1: Christoff Cowan

```
pub (4)rsa4096/97b67f174b958c0bf7f6461a4ed9760ab392d885 2025-11-26T03:04:45Z

uid Christoff Cowan (-) <christoff.cowan@mymona.uwi.edu>
sig cert  4ed9760ab392d885 2025-11-26T03:04:45Z ____________________ ____________________ [selfsig]
sig cert  bf9c211494238454 2025-11-26T04:02:55Z ____________________ ____________________ bf9c211494238454
sig cert  fa59cfdd3c2f178b 2025-11-26T04:35:26Z ____________________ ____________________ fa59cfdd3c2f178b

sub (4)rsa4096/afca4fe595db6ccb994734e7230b7623dae7c86f 2025-11-26T03:04:45Z
sig sbind  4ed9760ab392d885 2025-11-26T03:04:45Z ____________________ ____________________ []
```

Classmate 2: Shevar Roulston

```
pub (4)rsa4096/83dada9bc292fc946ac717febf9c211494238454 2025-11-26T03:07:43Z

uid Shevar Roulston (-) <shevar.roulston@mymona.uwi.edu>
sig cert  bf9c211494238454 2025-11-26T03:07:43Z ____________________ ____________________ [selfsig]
sig cert  4ed9760ab392d885 2025-11-26T03:53:45Z ____________________ ____________________ 4ed9760ab392d885
sig cert  fa59cfdd3c2f178b 2025-11-26T04:23:42Z ____________________ ____________________ fa59cfdd3c2f178b

sub (4)rsa4096/ce78701e379ee6ee5da7593589fd4ffa3b5d5e6f 2025-11-26T03:07:43Z
sig sbind  bf9c211494238454 2025-11-26T03:07:43Z ____________________ ____________________ []
```

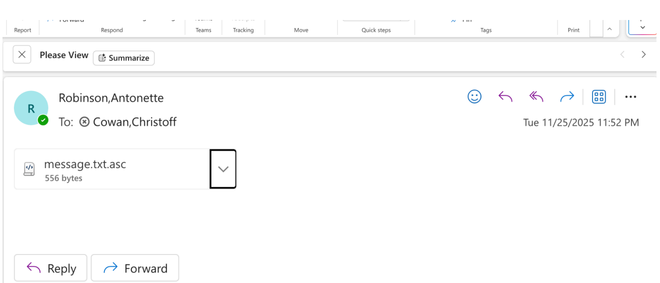
My Key Signed by Christoff Cowan and Shevar Roulston

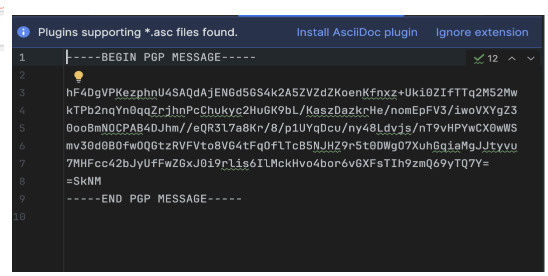
```
pub (4)rsa4096/556abe4fc8717a35e2be1653fa59cfdd3c2f178b 2025-11-26T03:08:47Z

uid Antonette Robinson (Antonette) <antonette.robinson@mymona.uwi.edu>
sig cert  fa59cfdd3c2f178b 2025-11-26T03:08:47Z ____________________ ____________________ [selfsig]
sig cert  4ed9760ab392d885 2025-11-26T04:02:02Z ____________________ ____________________ 4ed9760ab392d885
sig cert  bf9c211494238454 2025-11-26T04:14:12Z ____________________ ____________________ bf9c211494238454

sub (4)rsa4096/c9783add39222183e4ad1c69a7337c26e5471886 2025-11-26T03:08:47Z
sig sbind  fa59cfdd3c2f178b 2025-11-26T03:08:47Z ____________________ ____________________ []
```

Encrypted Email Sent to Christoff Cowan






```
-----BEGIN PGP MESSAGE-----

hF4DgVPKezphnU4SAQdAjENGd5GS4k2A5ZVZdZKoenKfnxz+Uki0ZIfTTq2M52Mw
kTPb2nqYn0qqZrjhnPcChukyc2HuGK9bL/KaszDazkrHe/nomEpFV3/iwoVXYgZ3
0ooBmNOCPAB4DJhm//eQR3l7a8Kr/8/p1UYqDcu/ny48Ldvjs/nT9vHPYwCX0wWS
mv30d0BOfwOQGtzRVFVto8VG4tFqOflTcB5NJHZ9r5t0DWgO7XuhGqiaMgJJtyvu
7MHFcc42bJyUfFwZGxJ0i9rlis6IlMckHvo4bor6vGXFsTIh9zmQ69yTQ7Y=
=SkNM
-----END PGP MESSAGE-----
```