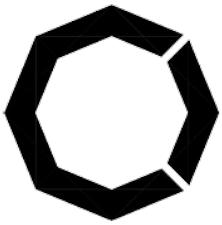

CalPolyPomona

College of Science

Computer Science

Encryption Project

Symmetric and Asymmetric Cryptography

Noah King
(ncking@cpp.edu)
https://github.com/nck2e3/CS-4600-Final-Project

Tingting Chen
CS 4600
Summer 2024

Department of Computer Science

July 2, 2024

Abstract

This paper details a cryptographic scheme that combines asymmetric and symmetric cryptography. The asymmetric cryptography utilized is Rivest-Shamir-Adleman (RSA), while the symmetric cryptography employed is a variant of the Rijndael block cipher as detailed in NIST's Advanced Encryption Standard (AES).

Contents

1 Packet Format

1.1 Packet Structure (Header + Payload)

The components of the message are serialized into a single byte-string (packet) before transmission. Figure 1 below depicts the structure of the data transmitted. It consists of an RSA encrypted AES key that is 32 Bytes in length (KEY), a GCM tag that is 16 Bytes in length (TAG), a 12 Byte nonce (NONCE), and a variable length message (MP).

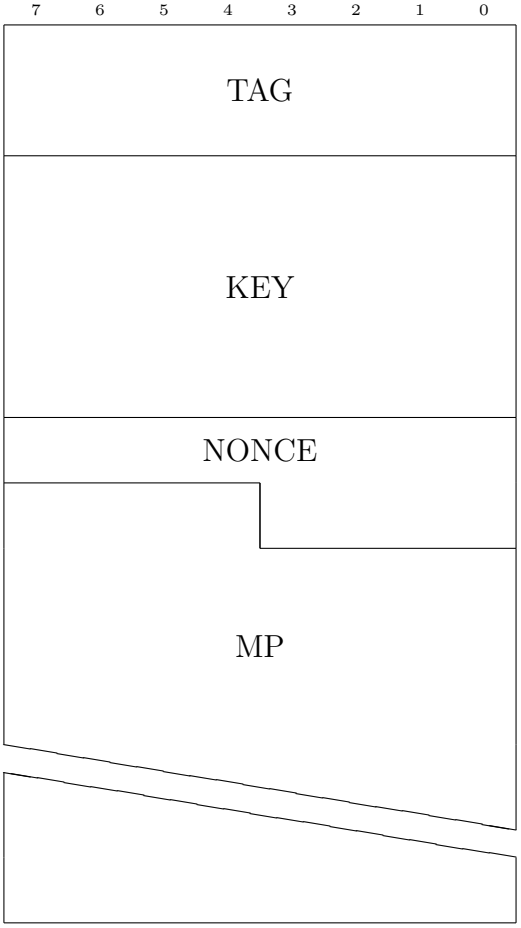

Figure 1: Packet Structure Bytefield

The initial element of the Packet is the Galois Counter Mode (GCM) tag of the AES encrypted message used for checking the authenticity and integrity of the AES encrypted message (MP). A traditional MAC or HMAC is not included in the message because Galois Counter Mode (GCM) includes authentication by default as a feature of the block-cipher mode, design wise it would be redundant to include a HMAC or MAC in addition to a GCM tag. Following the GCM tag, the second element of the message is the RSA encrypted AES key of the encrypted message (MP), the recipient of the message M would decrypt the AES key using their private key, then decrypt (MP) using the AES key. The next element of the overall message is the nonce of the AES encrypted message which ensures the uniqueness of the encrypted message (see symmetric encryption section for more detail). The final element of the packet is the actual AES encrypted message, which is of variable length.

Full Name of Packet Term	Abbreviation	Length in Bytes
GCM Tag	TAG	16
AES Key	KEY	32
AES Nonce	NONCE	12
Message Payload	MP	Variable

Table 1: Packet Terms

2 Encryption Scheme

2.1 Hybrid Cryptosystem (RSA + AES)

The encryption scheme used in this paper is a hybrid scheme combining symmetric and assymetric cryptography. We first encrypt a message payload (MP) using symmetric AES encryption, then encrypt the AES key used to decrypt the encrypted message (KEY) using RSA prior to transmission (it is worth noting the other data necessary in AES-GCM decryption, such as a nonce or tag are not encrypted using RSA but are appended to the message unencrypted as detailed in Section 1.1). Below is a mode-agnostic diagram of a message being sent from sender to recipient (Alice to Bob).

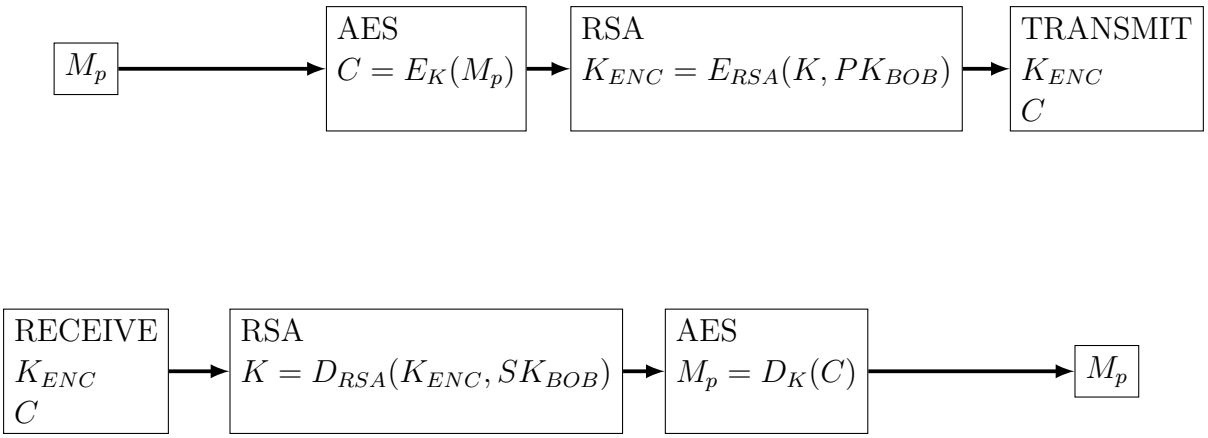

Figure 2: Data Processing Stages from Alice to Bob

A message M_p is encrypted using an AES key K, yielding ciphertext C. The AES key K is then encrypted using the RSA public key of the recipient PK_{BOB} which yields the encrypted AES key K_{ENC}. Alice then transmits the encrypted AES key K_{ENC} alongside the ciphertext C (as well as the mode specific elements necessary for AES decryption omitted in this diagram). The recipient Bob then recieves this data, then decrypts the AES key K_{ENC} using his private key SK_{BOB} to yield the decrypted AES key K. He then uses K to decrypt the ciphertext C to recover the original message M_p.

2.2 Key Length Considerations (AES & RSA)

The implementation of the encryption scheme discussed in section 2.1 initially used a 2048-bit RSA key length, following NIST's recommendation for minimum security. An attempt was made to increase this to the highest key length recommended by NIST, 15,360 bits. However, it became apparent that the computational resources available to me were insufficient to handle such a large key. As a result, a 2048-bit key length was selected because it is the minimum length recommended by NIST that balances security with the practicality of generating keys efficiently on available hardware. Additionally, the key length for the symmetric encryption is 256 bits as it is the highest level of security recommended by NIST.

2.3 Block Cipher Mode & Authentication Considerations (GCM)

Galois Counter Mode was chosen as the block-cipher mode for our chosen symmetric encryption algorithm AES. This mode was chosen in contrast to the other modes due to it's built-in authenticity and integrity checking features. These features making the inclusion of a traditional MAC or HMAC unnecessary as the encryption process automatically generates a GCM tag which serves the same purpose. GCM is more computationally efficient then other modes where the Encryption and Authentication mechanisms are separate such as CBC+MAC as it combines authentication and encryption in single passes. Furthermore, GCM is more parallelizable than CBC, this is clear in the two diagrams below, which shows more operations happening concurrently in comparison to CBC, which has operations occur sequentially in a serial nature. Furthermore, one can see the generation of the GCM tag in Figure 3, whereas there is no built in authentication in the CBC case (or OFB, ECB, CFB, and CTR mode for that matter). Additionally, GCM has minimal error propogation in the ciphertext, leaving errors isolated to the block in which they occur, which is more ideal than CBC and other block-cipher modes.

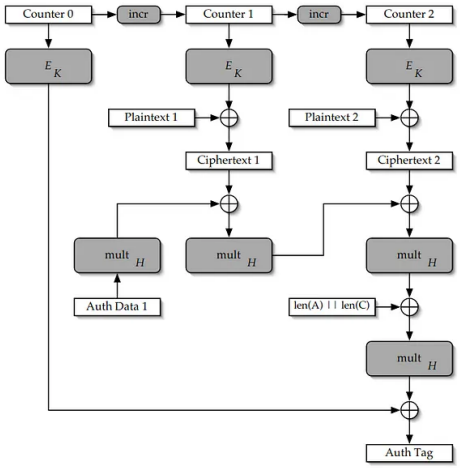

Figure 3: AES in Galois Counter Mode

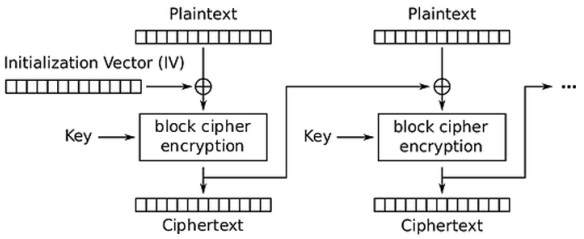

Figure 4: AES in CBC Mode

In short, GCM was chosen because it made implementation easier and more performant in comparison to the other available options. It provides built-in authentication and integrity verification, which made the addition of a MAC or HMAC to the overall encryption scheme unecessary.

2.4 Authentication and Integrity Verification (MAC + HMAC)

As mentioned previously, a MAC and HMAC was not implemented as Galois Counter Mode (GCM) innately performs the functions of a MAC or HMAC. See section 2.3 for more details.

3 Implementation Details

3.1 Language and Library Selection

Python and the Pycryptodome library was chosen due to the plethora of documentation and learning resources available. Additionally I am already familiar with python as a language, and it is faster for me to prototype in Python.

3.2 Abstractions

The below "RSA_User" class simplifies the generation of RSA keypairs. The constructor of RSA_User objects generates an RSA keypair that is associated to a username defined in the single operand of the constructor "name". The "writeToFile()" function writes the RSA public key and private key to files with filenames that are dependent on the "name" member variable of RSA_User.

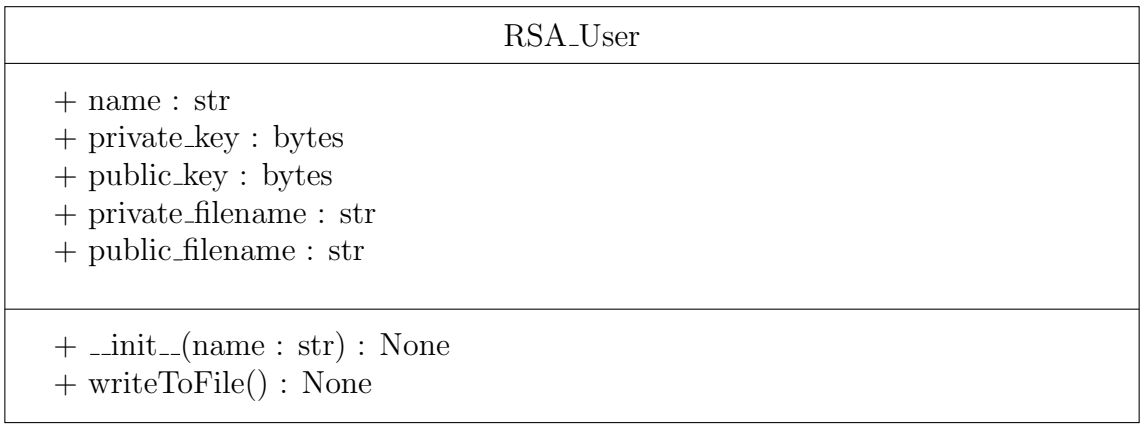

Figure 5: RSA_User UML Diagram (Python)

The Packet class depicted on the next page encapsulates functionality related to the encryption and decryption of messages, using both symmetric (AES-256) and asymmetric (RSA) cryptography. It provides two static methods: encrypt and decrypt. The encrypt method takes a recipient's public key file and a message as inputs. It generates a random AES-256 key and a nonce for Galois Counter Mode (GCM) encryption, encrypts the message using AES-256 in GCM mode to produce ciphertext and an authentication tag (GCM tag), reads the recipient's RSA public key from a file and uses it to encrypt the AES key, and then serializes the GCM tag, encrypted AES key, nonce, and ciphertext into a single byte object for transmission. The decrypt method takes the recipient's private key file and a serialized packet as inputs. It unpacks the serialized packet to retrieve the GCM tag, encrypted AES key, nonce, and ciphertext, reads the recipient's RSA private key from a file and uses it to decrypt the AES key, decrypts the ciphertext using the decrypted AES key and the nonce in GCM mode, and verifies the authenticity and integrity of the message using the GCM tag. If verification fails, it returns an error message; otherwise, it returns the decrypted message as a string.

Packet
+ encrypt(public_key_recipient_file : str, message : bytes) : bytes + decrypt(private_key_recipient_file : str, serialized_packet : bytes) : str

Figure 6: Packet UML Diagram (Python)

4 Demonstration

4.1 How to Use the Demo Scripts (Simple Demo)

To run the simple demo, simply execute the script inside the "Minimal Demo" folder. The code for the main function for the "Minimal Demo" is shown below. Additionally a sample console output is shown as well. Observe changes in the console output and the ".txt" file containing encrypted data.

```
PS C:\Users\nk2e3\Desktop\CS-4600 Final Project> & "C:/Program Files/Python310/python.exe" "c:/Users/nk2e3/Desktop/CS-4600 Final Project/Minimal Demo/main.py"

UNENCRYPTED MESSAGE (Alice's End before Transmission): THIS IS A MESSAGE

[BEGIN ENCRYPTED MESAGE IN TRANSIT]
b'\x80\x04\x95@\x01\x00\x00\x00\x00\x00(C\x10\x89"\xb8#\x03\xa2#.4\xfdJ\xcc\xb2\xae\xc2%\x94B\x00\x01\x00\x00\x90\xab\x01\x1a5\xdd.\xebxw\xcf\x12=\x8fk\xf3\
xb0T\xb2`\t\xf2\x04\xf0\x99\xc9\x01\x99q\xe5E\xf5`\xb0\x9d\xf3\r\xe9\xc3\xdfBf~\x97\xf6<\xad\xe5\xd6\xc4\xe8_\x10V\xbd__q~s5\x08O8\xa34\xcb\x9d\xd1\x0cp\xfc\xea
\xcf\x15\x1c\x1e\xddR?\xcfq\xc1}<AV\xd4\x9b\xaf,\x8d|\xa3\xda)\x19\xbaRRw\x05-\xcb\xf4B*>2\t5\x00\xf9\xba\x18g\x81\x00u\x90\xcak\x7f\x0e\xe7\xbaBv\xa2zv\xee\xc8
\x9ae\x7fjp6\xc4\x02dL\xc3afh\x06\xa1\x98\x864\xce\xe1\x110\xfc;\xef\x80\xdfPa{<\xa5\x93x\x13\x8a\xae\x15\xddC@T6\xf7x\xbdM\x1c\xf9<8\x8f\xbb\xa6\xe30J\x8a\xce\
xfd/M\xf4\xd4\xa3\xc6*\xe2\xa6\xe5\x93\x9f\xcf\x8d\x88p\xe0\xf8\x8b\xd8\x1cK;\x0f\xb2\xff\xac\x90\x97\x00;\xb6\xe8\xaa:t\xfe|\x84\xbf\xb2\xdaV5\x936\xffX\x0b~UT
\xee\xa1\xc2\x9d\xd0 \xbb\xa9\xa8\x92\x94C\x0cl\xd0\x0b\x16\xcaE\xf7w\xda\xe4\xd6X\x94C\x11\xd4\xc2\xaa\x8e\x15\x93\x82y\x94\x17\xfeL\xc4\xf0\xf0|\xc2\x94t\x94.
'
[END ENCRYPTED MESAGE IN TRANSIT]

Decrypted Message (Bob's End): THIS IS A MESSAGE

PS C:\Users\nk2e3\Desktop\CS-4600 Final Project>
```

Figure 7: Simple Demo, Console Output

```python
def main():
    #Generating RSA keypairs...
    RSA_Alice = RSA_User("Alice")
    RSA_Bob = RSA_User("Bob")
    #Writing these keypairs to files...
    RSA_Alice.writeToFile()
    RSA_Bob.writeToFile()
    #SIMPLE DEMO - Alice sends a message to Bob
    message = "THIS IS A MESSAGE"
    #Unencrypted message prior to transmission, see encrypt() function in Packet class for more details...
    print(f"\nUNENCRYPTED MESSAGE (Alice's End before Transmission): {message}")
    #Message in Transit to Recipient, encrypt message before transmission...
    encrypted_message = Packet.encrypt(RSA_Bob.public_filename, message.encode('utf-8'))
    print(f"\n[BEGIN ENCRYPTED MESAGE IN TRANSIT]\n{encrypted_message}\n[END ENCRYPTED MESAGE IN TRANSIT]\
    n")
    #Writing encrypted message to a file...
    with open(f"TRANSMITTED MESSAGE ({RSA_Alice.name} to {RSA_Bob.name}).txt", 'wb') as file:
        file.write(encrypted_message)
    #Message upon delivery to recipient, see decrypt() function in Packet class for more details...
    decrypted_message = Packet.decrypt(RSA_Bob.private_filename, encrypted_message)
    print(f"Decrypted Message ({RSA_Bob.name}'s End): {decrypted_message}\n")
```

Figure 8: Simple Demo, main()

4.2 How to Use the Demo Scripts (Complex Demo)

To use the "Complex Demo", simply run the script and follow the user input prompts. The complex demo differs from the simple demo in that it demonstrates communication both ways. In this demo, the user chooses the sender, recipient, and message sent.

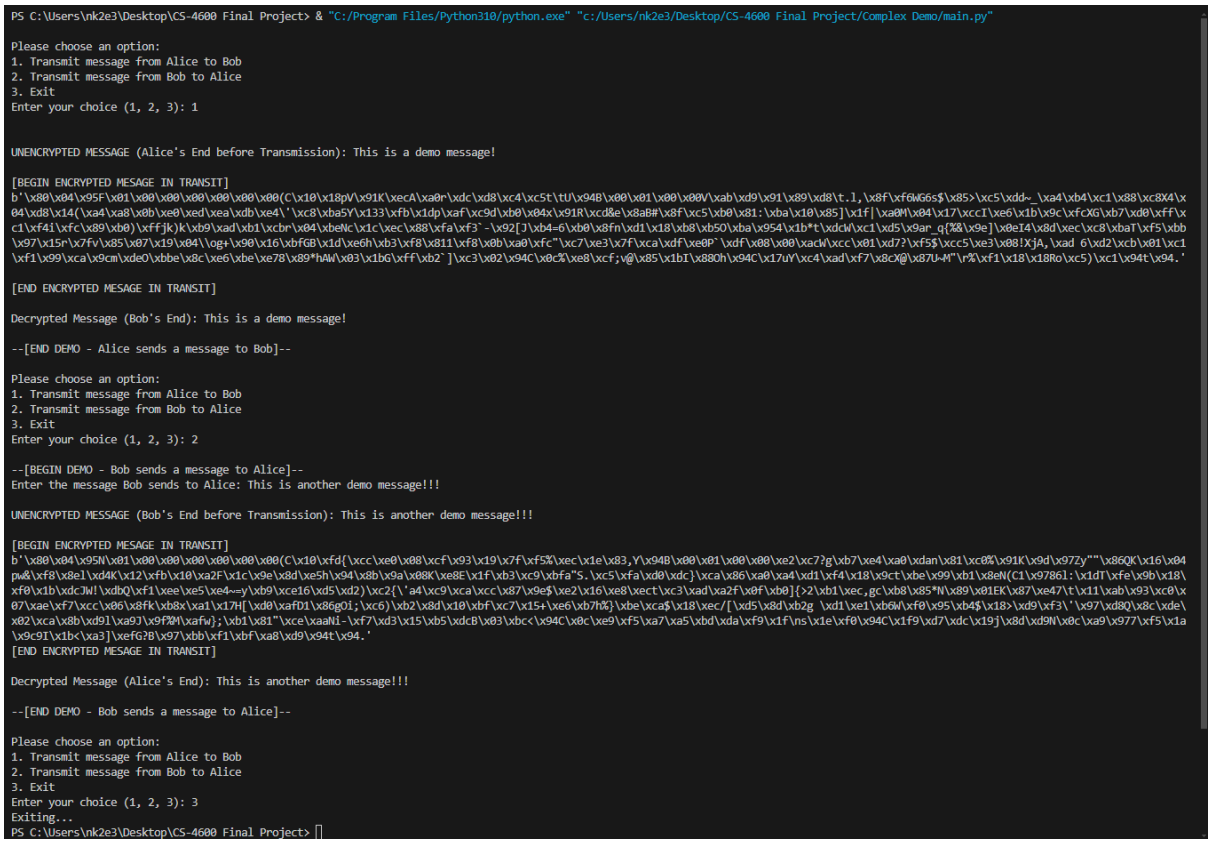

Figure 9: Complex Demo, Console Input/Output

```python
def main():
    #Generating RSA keypairs...
    RSA_Alice = RSA_User("Alice")
    RSA_Bob = RSA_User("Bob")
    #Writing these keypairs to files...
    RSA_Alice.writeToFile()
    RSA_Bob.writeToFile()
    while True:
            print("\nPlease choose an option:\n1. Transmit message from Alice to Bob\n2. Transmit message from Bob to Alice\n3. Exit")
            choice = input("Enter your choice (1, 2, 3): ")
            match choice:
                case '1':
                    #Alice sends a message to Bob...
                    transmit_message_demo(RSA_Alice, RSA_Bob)
                case '2':
                    #Bob sends a message to Alice...
                    transmit_message_demo(RSA_Bob, RSA_Alice)
                case '3':
                    print("Exiting...")
                    break
                case _:
                    print("Invalid choice. Please try again.")
```

Figure 10: Complex Demo, main()

References

NIST Special Publication 800-57 Part 1, Revision 5

FIPS PUB 197: Advanced Encryption Standard (AES)

https://pypi.org/project/pycryptodome/