Analysis of Hybrid RSA-AES Encryption for Secure Messaging

Angelica Kierra Ninta Gurning - 13522048[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522048@std.itb.ac.id

***Abstract*—Over the years, communication platforms has transformed, with instant messaging services like Whatsapp dominating, hosting 2.44 billion users and facilitating over 100 billion daily messages by April 2022. However, the growing reliance on social media introduces potential vulnerabilities in data handling. To address this, popular messaging platforms such as Whatsapp, Telegram, and iMessage adopt end-to-end encryption. Whatsapp, employing the Signal Protocol, employs key pairs and "ratcheting" to safeguard messages, ensuring a continually evolving and secure system to prevent data leaks and protect user privacy.**

***Keywords*—cryptography,encryption,security**

I. Introduction

Over the course of decades, forms of communication have varied, With the rapid growth if technology and the internet. Society's form of communication has shifted into the instant messaging services. As social media becomes more prevalent in our day-to-day routine, text messaging applications have become society's main form of communication. Delivering messages through instant messaging applications has become an irreplaceable part in social interactions.

Taking data from Whatsapp alone, by April 2022, there are 2.44 billion active users globally. Over 100 billion messages are sent through Whatsapp each day. Let alone from other messaging applications, resulting in a surge amount of text messages exchanged daily.

However, with the increasing amount of dependency in social media, there is also an increasing amount of vulnerability. These vulnerabilities stem from the potential mishandling of a client's data. Consequently, secure protocols are crucial to build a protected chatting system. While messaging, users can sometimes include private data or information. If not handled correctly, it could pose a serious risk for the client, resulting in data leaks.

Chat security and privacy are essential to have a safe and secure data transaction within clients. In response to a possible security breach, chat messaging applications are always constantly being developed to follow the trend. Popular text-messaging services such as Whatsapp, Telegram, and iMessage have implemented a form of security protection to mitigate possible data leaks,

One of the most common practices is using end-to-end encryption as implemented by Whatsapp, Telegram, and iMessage. To simplify, end-to-end encryption will intercept a client text messages if it detects an outside party by scrambling it to be unreadable. Whatsapp in particular, uses Signal Protocol which provides a set of cryptographic specifications for end-to-end encryption. As an overview, Signal Protocol uses a technique called 'racheting' which sets a combination of temporary and permanent pairs of public and private keys to both clients and updates those keys with every message sent.

II. Cryptography Theory

Formally, cryptography is the art of maintaining the security of messages by encoding them into another unreadable and meaningless form. Its purpose is to provide confidential messages so that unauthorized parties cannot read them. Messages are received in the form of *plaintext* (readable and understandable) and delivered in the form of *ciphertext* (unreadable and meaningless). *Encryption* is the process of encoding plaintext into ciphertext. Whereas *decryption* is the process of returning the ciphertext to its plaintext.

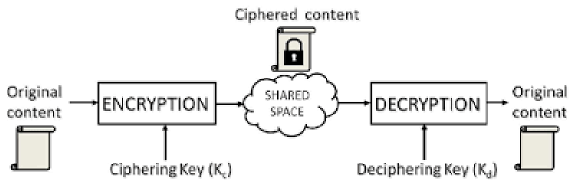


Figure 1. Encrypting and decrypting scheme (https://www.researchgate.net/figure/Main-stages-of-a-common-encryption-chain-The-content-is-encrypted-using-a-ciphering_fig2_282333370)

Historically, the study of cryptography is divided into three stages. Early cryptography goes back to the time of Julius Caesar. Early encryption took the form of substitution cyphers where the alphabets are shifted to create a ciphertext.

The second stage of cryptography revolves around the World War II period and based on cryptographic engines such as the German Enigma Machine. These machines are used to encrypt and transmit coded messages which the German used during the war. The final stage is today's modern cryptography, which highly relies on computers and mathematical calculations.

Cryptography itself has protocols and are required to be made public. Those protocols are embedded in the form of keys. Keys ar data, not algorithms. A key is a sequence of characters which contains an encryption of data and is made to a seemingly random appearance. Its function is similar to a physical key, allowing those with the correct key to decrypt (unlock) the encryption.

Generally, cryptography is classified into 2 main methods, Symmetric Encryptions, and Asymmetric Encryptions.

A. Symmetric Encryptions

Symmetric key encryption involves transforming a message using a key which then can be used for decryption. It is referred to as a secret key, shared through a private channel between the sender and receiver. The original plaintext will have a larger size than the ciphertext. This form of encryption is typically used for transferring a massive amount of data. Symmetric key encryption utilizes keys in 128 or 256 bits. Compared to asymmetric encryptions, it has lower security due to only using one key for both encryption and decryption, though the symmetric encryption process is very fast.

Symmetric key algorithms are categorized into 2 types based on the input data. Block cipher converts plaintext by taking one plaintext's block at a time (64/128/256 bits). Whereas stream cipher converts plaintext by taking 1 byte of a plain text at a time.

Block ciphers require an initialization vector (IV), a random sequence of characters to encrypt the first block of plaintext. It acts as the initialization vector for subsequent blocks. There are different modes used to generate *IV,* Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cyber Feedback (CFB), Counter (CTR), Output Feedback (OFB), Galois (GCM).

In stream ciphers, initially a key will be generated by a random bit generator which then produces a random 8-bit output that will be treated as a *keystream.* Keystream is a combination of plaintext with random characters to produce ciphertext. Characters vary from bits, or even conventional alphabets. Keys are usually made longer to prevent brute force attacks. As an example, plaintext will undergo XOR operation to generate ciphertext.

Encryption

```
Plaintext      : 1101010
Keystream      : 0100110
-------------------------------- XOR
Ciphertext     : 1001100
```

Decryption

```
Ciphertext     : 1001100
Keystream      : 0100110
-------------------------------- XOR
Plaintext      : 1101010
```

Some examples of block cipher symmetric systems are DES, IDEA, Blowfish, and others. For stream cipher symmetric systems are RC4, TKIP, and others.

B. Asymmetric Encryptions

In contrast to symmetric encryption, asymmetric encryptions require two keys. One key is made publicly available and the other private, together used to encrypt and the other to decrypt. Both keys are acquired from mathematical calculations.

In asymmetric encryption, the size of ciphertext is same as the original plaintext. Asymmetric encryption is mainly used for small amounts of data, as its process is slower than symmetric encryption. Asymmetric encryptions need a higher rate if CPU utilization. Compared to symmetric encryption, asymmetric encryption is more secure as it needs two separate keys from encryption and decryption. Asymmetric systems provide enhanced security in the form of digital signatures.

Asymmetric encryptions are mainly used for authentication and confidentiality. With the use of public key, anyone can encrypt a message, allowing confidentiality. Even so, only the private key owner can decrypt the message.

Another principal foundation of cryptography is none other than 'Hash Function'. It is an algorithm which takes an input of arbitrary length then converts it into a binary string with a fixed size of bits. In other words, hash functions use mathematical calculations to transform a numerical input into a more compressed value. When dealing with hash functions it is important to avoid collisions. Some widely used hash functions are :

1. Secure Hashing Algorithm (SHA)
2. RACE Integrity Primitives Evaluation Message Digest
3. Message Digest Algorithm
4. Whirlpool

III. Discussion

The process of encrypting for secure messaging will be put into different parts, such as :

A. End to End Encryption

End-to-end encryption is a system for communication. It allows only the sender and receiver to be able to view the message. End-to-end encryption is used in most instant messaging services nowadays.

Data is encrypted on the sender's device and only the receiver is allowed to decrypt it. It prevents third parties from obtaining the data while it's being transferred. By using E2E (End-to-end encryption), it halts data tampering. In E2E, decryption keys are not transmitted, as the receiver will already have it in their possession, this is the reason why E2E is able to halt tampering. Tampering is a malicious threat, if a message is tampered with or altered during transmission, the receiver will not be able to decrypt the contents of the message.

E2E utilizes cryptographic keys for encrypting and decrypting messages, the keys will be stored on endpoints. Once the public key is shared, other parties will be able to encrypt a message and send it to the owner of the public key. The messages will only be able to be read using the private key.

In secure messaging, end-to-end encryption is achieved using two methods, RSA method for asymmetric encryption and AES method for symmetric encryptions. As an overview there are two steps of how both asymmetric and symmetric encryption are carried through:

1. Key Exchange (RSA)

The RSA asymmetric algorithm will generate each pair of keys, private keys and public keys for the sender and receiver. The public key will be openly distributed, whereas the private key will be kept strictly confidential.

When the two parties want to start a conversation, they will exchange their public keys. As said earlier, the public keys are used to allow other parties to convey messages to the owner. Secure messaging implements the DHKE (Diffie-Hellman key exchange) algorithm for secure public key exchange between the sender and receiver. Once both parties receive the exchanged keys, they will generate a shared key. Only the owner with the corresponding private key can decrypt the message and retrieve the shared keys.

2. Data Encryption and Encryption (AES)

Once the symmetric keys are obtained securely using RSA, both parties will use AES to encrypt and decrypt the actual message or data. AES is a symmetric algorithm that make use of the shared keys for decrypting the encrypted message.

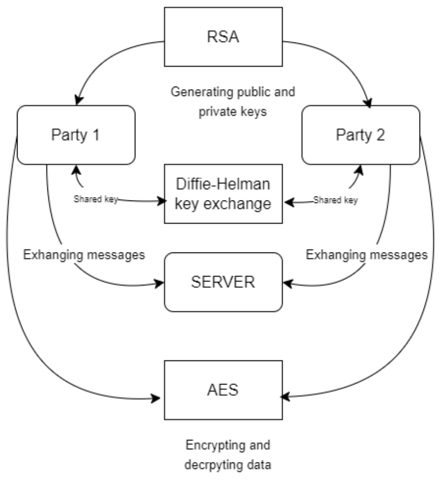

RSA
Generating public and private keys
Party 1
Party 2
Diffie-Helman key exchange
Shared key
Shared key
Exhanging messages
Exhanging messages
SERVER
AES
Encrypting and decrpyting data

Figure 2. End-to-End Encryption Scheme (source: writer's archive)

B. RSA Method

The RSA method is developed by 3 MIT researchers, Ronald Rivest, Adi Shamir, and Leonard Adleman in 1976, hence the name RSA (Risvest-Shamir-Adleman). As it is an asymmetric encryption, two keys will be generated, a private key for decrypting and a public key for encrypting.

A brief glossary for commonly used terminologies for generating public keys and private keys using RSA method:

1. Public key pairs **(n,e)**
2. Private key pairs **(n,d)**
3. Two distinct prime numbers **p,q**
4. Product of two prime numbers **n**
5. Totient function **φ(n)**
6. Public exponent **e**
7. Private exponent **d**
8. Plaintext **m**
9. Ciphertext **c**

The first step to generate a RSA key is to choose two distinct prime numbers *p* and *q*. Generally, a large number is commonly chosen to keep the security of the keys. Using a large number complicates outside parties since there will be a plethora of combinations for factorizations. Therefore, security is still preserved, even if *n* is made public. Determine *n* (not to be kept private)

$$n = p \times q \qquad (1)$$

Determine the totient function *φ(n)*. The totient function is used to generate the public exponent by. Pick a number such that it is an integer and not a factor of *n* and is relative prime to *φ(n)*

$$\varphi(n) = (p-1)(q-1) \qquad (2)$$

$$\gcd\big(e, \varphi(n)\big) = 1 \qquad (3)$$

The public key is generated as *(n,e)*. The public key is used to create *d* (private exponent). Determine *d* by checking congruence, such that

$$ed \equiv 1\ (mod\ \varphi(n)) \qquad (4)$$

The private key is generated as *(n,d)*. With the existence of both the private key and public key, messages can be encrypted and decrypted. To encrypt a plaintext into ciphertext, divide it into blocks to optimize computing, then represent the message into an integer between 0 and n-1. Encrypt by raising m to the power of *e*, to decrypt raise to the power of *d*

$$c \equiv\ m^e\ mod\ n \qquad (5)$$

$$m \equiv\ c^d\ mod\ n \qquad (6)$$

The RSA algorithm is secure; however, it has its limitations. Especially when computing large amounts of data. Therefore, in messaging end-to-end encryptions, AES is used to process the data, complimenting RSA's flaw.

C. AES Method

AES method is a symmetric encryption algorithm used for encrypting and decrypting the data in E2E messaging services. It is first introduced in 1998. It increased the previously 64 bits block-size into 128 bits. It takes a plaintext in the size of 128 bits or 16 bytes and generates a key. The key length varies from 128,192, or 256 bits. The standard AES algorithm specifies three blockciphers: AES128, AES192, AES265 (corresponding to the key length). In general, the AES functions as

```
Function AES(M)
    (K0,..,K10) ← expand(K)
    S ← M ⊕ K0
    For r = 1 to 10 do
        s ← S(s)
        s ← shift-rows(s)
        if r ≤ 9 then s ← mix-cols(s)
        s ← s ⊕ Kr
    Endfor
    Return s
```

From the function above AES takes M, the message, in the size of 128 bits. The *s* is called the *state*, M will be the initialisation state and C (ciphertext) will be the final state. The keys created are represented as *K*. Each iteration is called a *round*, the number of rounds varies according to the length of key that will be produced. In the function above each round is represented as *r*, the number of 1-10 states it is an AES128 operation.

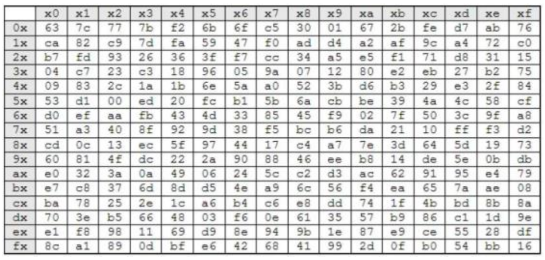
Key Length	Rounds
128 bits (16 bytes)	10 Rounds
192 bits (24 bytes)	12 Rounds
256 bits (32 bytes)	14 Rounds

Table 1. AES Round numbers

In each *round*, identical operations are performed with different subkeys (*Kr*). There are four procedures, that are utilized in each round:

1. The Byte Sub Transformation (BS)

This process is represented as *S*. Byte Sub Transformation uses an S-Box for its operation. An S-Box is a 16 x 16 matrix of byte values. It contains a precalculated permutation of all possible 256 8-bit value.



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3. S-box (https://www.researchgate.net/figure/AES-S-Box-Rijndael-S-Box-16_fig4_318906543)

Each byte of *M* (message) will be substituted with the corresponding byte from the S-box, depending on the position of the rows and columns. AES will find the compatible byte and selects it as a unique 8-bit value for substitution.

As an example *(will be made in a smaller scale than original for explaining purposes)*,

Plaintext(M)	Ciphertext(C)
$\begin{bmatrix} 00 & 01 \\ a0 & a1 \end{bmatrix}$	$\begin{bmatrix} 63 & 7c \\ e0 & 32 \end{bmatrix}$

Table 2. AES Byte Sub Transformation Example

The input 'a0' is divided into two parts, 'a' is the row in S-box, '0' is the column in S-box. The output ciphertext will be the corresponding element based on the row and column.

2. The Shift Row Transformation (SR)

This process is represented as *shift-rows*. In this process, the state rows are shifted left circularly over multiple offsets. Row 0 is not shifted, shifting starts from row 1. The row will be moved *(n)* bytes, with *n* being its row index.

As an example,

Before	After
$00 \quad 01 \quad 02$	$00 \quad 01 \quad 02$
$a3 \quad a5 \quad a6$	$a5 \quad a6 \quad a3$
$ef \quad e2 \quad e1$	$e1 \quad ef \quad e2$

Table 3. AES Shift Row Example

As seen from the table, row 0 is not shifted, row 1's element is shifted one byte circularly to the, and row 2's element is shifted 2 bytes circularly to the left.

3. The Mix Column Transformation (MC)

This process is represented as *mix*-cols. It is not performed in the last round. The shift row transformations produce a *state matrix*. In the Mix Column Transformation, the state matrix is treated as a 4x4 matrix. A multiplication operation is then performed/ The multiplication involves multiplying each element in the state matrix with the corresponding element in a pre-defined *mix column* matrix, usually from the Galois Field *(for the purpose of the essay, a simplified matrix will be used)*.

The pre-defined matrix is represented as,

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Say the state matrix is represented as,

$$\begin{bmatrix} 63 & ca & eb & af \\ 25 & 12 & 34 & 98 \\ ab & 09 & dc & ba \\ 13 & 4f & 56 & 8c \end{bmatrix}$$

The operation involves multiplying the pre-defined matrix row elements with the state matrix column elements and operating a XOR (⊕) operation on each individual element. The formula for each column in the state matrix as follows,

$$\begin{aligned} (P_{00} \times S_{0a}) \oplus (P_{01} \times S_{1a}) \oplus (P_{02} \times S_{2a}) \oplus (P_{03} \times S_{3a}) &= A1 \\ (P_{10} \times S_{0a}) \oplus (P_{11} \times S_{1a}) \oplus (P_{12} \times S_{2a}) \oplus (P_{13} \times S_{3a}) &= A2 \\ (P_{20} \times S_{0a}) \oplus (P_{21} \times S_{1a}) \oplus (P_{22} \times S_{2a}) \oplus (P_{23} \times S_{3a}) &= A3 \\ (P_{30} \times S_{0a}) \oplus (P_{31} \times S_{1a}) \oplus (P_{32} \times S_{2a}) \oplus (P_{33} \times S_{3a}) &= A4 \end{aligned} \quad (7)$$

P_{ij} refers to the pre-defined matrix's elements, S_{ia} refers to the state matrix's elements where a is the column's index. AX refers to the new state matrix elements (per column). For instance, here is an example of the first row's calculation,

$$\begin{aligned} (02 \times 63) \oplus (03 \times 25) \oplus (01 \times ab) \oplus (01 \times 13) &= 11 \\ (01 \times 63) \oplus (02 \times 25) \oplus (03 \times ab) \oplus (01 \times 13) &= 3b \\ (01 \times 63) \oplus (01 \times 25) \oplus (02 \times ab) \oplus (03 \times 13) &= 29 \\ (01 \times 63) \oplus (01 \times 25) \oplus (01 \times ab) \oplus (02 \times 13) &= 81 \end{aligned}$$

Once the mix colum operation has been carried through the new state matrix should be,

$$\begin{bmatrix} 11 & e4 & c0 & a0 \\ 3b & ba & 41 & 3d \\ 29 & 27 & 65 & e7 \\ 81 & db & 85 & 37 \end{bmatrix}$$

4. Add Round Key (ARK)

In each round, a set of round keys will be obtained from the other three operations. An XOR operation will be performed between the previous round keys with the new state matrix. The result will be the final key generated.

For example, there is a set of round keys matrix, and the new state matrix,

$$\begin{bmatrix} 3e & a7 & 35 & 68 \\ 1c & 92 & 6b & fe \\ a1 & 47 & f0 & 4d \\ c3 & 10 & b7 & a1 \end{bmatrix} \begin{bmatrix} 11 & e4 & c0 & a0 \\ 3b & ba & 41 & 3d \\ 29 & 27 & 65 & e7 \\ 81 & db & 85 & 37 \end{bmatrix}$$

Round keys State matrix

The Add Round Key operation formula is stated as follows,

$$S_{ij} \oplus R_{ij} = K_{ij} \quad (8)$$

S_{ij} represent the state matrix elements, whereas R_{ij} represents the round key matrix elements. The new key is represented with K_{ij}. As an example, the first row will be calculated int the form of,

$$\begin{aligned} 11 \oplus 3e &= 2f \\ e4 \oplus a7 &= 43 \\ c0 \oplus 35 &= f5 \\ a0 \oplus 68 &= c8 \end{aligned}$$

The final state matrix result for each round iteration is,

$$\begin{bmatrix} 2f & 43 & f5 & c8 \\ 27 & 28 & 2a & c3 \\ 88 & 60 & 95 & aa \\ 42 & cb & 32 & 96 \end{bmatrix}$$

The four operations will be executed with the round numbers accordingly.

D. Diffie-Helman Key Exchange

In plain sight, the Diffie-Helman Key Exchange is similar with RSA, though both hold significant differences. Diffie-Hellman is used for secure key exchange, whereas RSA is used for encryption and decryption. Diffie-Hellman key generation is based on modular arithmetic, contrasting with RSA which is heavily based on the complexity of number factorization. T

The Diffie-Helman procedure requires two parties, both parties will select two prime numbers g and p. The parties then will choose a private key individually a or b. This is used to generate a key using the equation,

$$x = g^a \bmod p$$
$$y = g^b \bmod p$$

After that, they both will exchange public keys. Alongside the private key, they will generate a shared secret key with the formula,

$$K1 = y^a \bmod p$$
$$K2 = y^b \bmod p$$

Mathematically, it can be shown that K1 = K2 and both parties now have a shared secret key to encrypt.

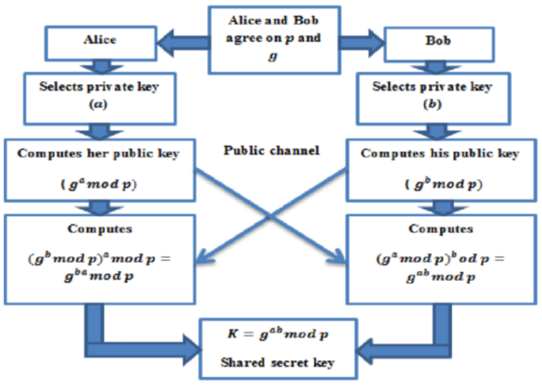

Alice and Bob agree on p and g
Alice
Bob
Selects private key (a)
Selects private key (b)
Computes her public key ($g^a mod\ p$)
Public channel
Computes his public key ($g^b mod\ p$)
Computes $(g^b mod\ p)^a mod\ p = g^{ba} mod\ p$
Computes $(g^a mod\ p)^b od\ p = g^{ab} mod\ p$
$K = g^{ab} mod\ p$
Shared secret key

Figure 4. Diffie – Hellman Procedure (https://www.researchgate.net/figure/Block-diagram-of-the-Diffie-Hellman-algorithm_fig1_349609600)

As an example.

Let ,

p = 17
g = 3

Party I private key = 5
Party II private key = 7

$x = 3^5$ *mod* 17 = *5*
$y = 3^7$ *mod* 17 = *11*

Party I shared private key:
11^5 *mod* 17 = 10

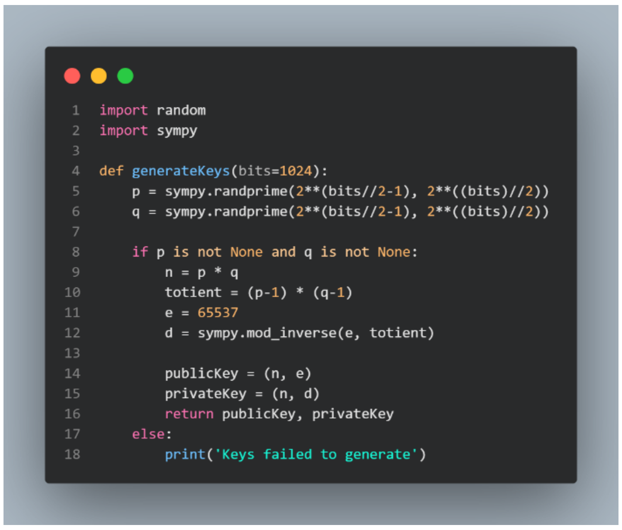
Party II shared private key:
5^7 *mod* 17 = 10

The shared private key between the two parties is 10.

IV. EXPERIMENT

A. Implementing RSA

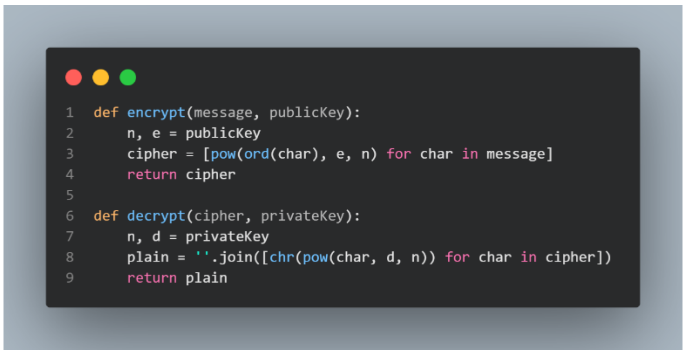
Though many cryptographic code libraries provide a generated RSA algorithm, for testing purposes below is an excerpt of how RSA algorithms work generally. The *random* and *sympy* python module will be used.



```python
import random
import sympy

def generateKeys(bits=1024):
    p = sympy.randprime(2**(bits//2-1), 2**((bits)//2))
    q = sympy.randprime(2**(bits//2-1), 2**((bits)//2))

    if p is not None and q is not None:
        n = p * q
        totient = (p-1) * (q-1)
        e = 65537
        d = sympy.mod_inverse(e, totient)

        publicKey = (n, e)
        privateKey = (n, d)
        return publicKey, privateKey
    else:
        print('Keys failed to generate')
```

Figure 5. Generating Public Key and Private Key (source: writer's archive)

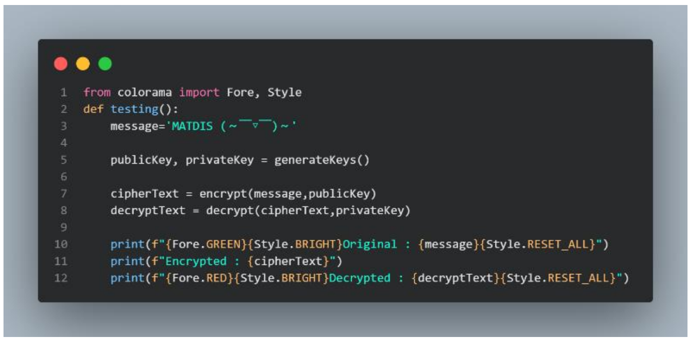
Above is a function to generate RSA public and private keys. In this function specifically, the keys sizes will be 1024 bits. Numbers *p* and q are generated by 'sympy.randprime' with the parameter **(2**(bits//2-1), 2**((bits)//2))** to ensure it generates the correct bit length. *e* is chosen arbitrarily, *d* is calculated using 'sympy.mod_inverse' function.



```python
def encrypt(message, publicKey):
    n, e = publicKey
    cipher = [pow(ord(char), e, n) for char in message]
    return cipher

def decrypt(cipher, privateKey):
    n, d = privateKey
    plain = ''.join([chr(pow(char, d, n)) for char in cipher])
    return plain
```

Figure 6. Encrypting and Decrypting RSA Function (source: writer's archive)

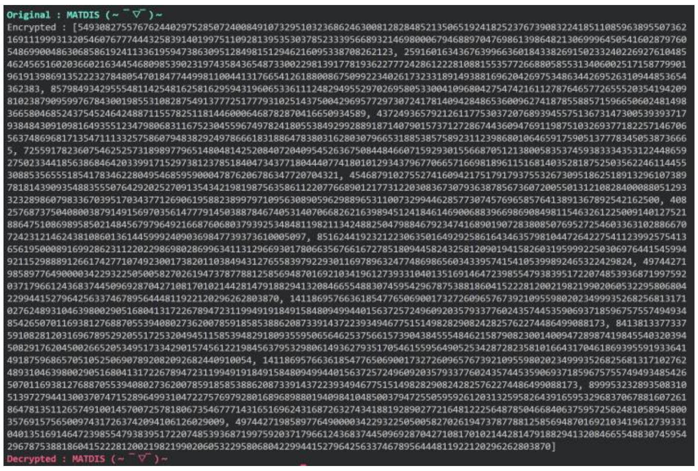
The *encrypt* function turns the characters in message into ASCII and encrypt them using the RSA algorithm, whereas *decrypt* function turns the ASCII back into characters and appends each character to decrypt the message.



```python
from colorama import Fore, Style
def testing():
    message='MATDIS (~￣▽￣)~'

    publicKey, privateKey = generateKeys()

    cipherText = encrypt(message,publicKey)
    decryptText = decrypt(cipherText,privateKey)

    print(f"{Fore.GREEN}{Style.BRIGHT}Original : {message}{Style.RESET_ALL}")
    print(f"Encrypted : {cipherText}")
    print(f"{Fore.RED}{Style.BRIGHT}Decrypted : {decryptText}{Style.RESET_ALL}")
```

Figure 7. RSA Testing Scheme (source: writer's archive)

The message '**MATDIS (~￣▽￣)~**' will be encrypted and decrypted using the program. The *coloroma* module is only implemented for testing displaying and formatting purposes. The result is projected as follows:

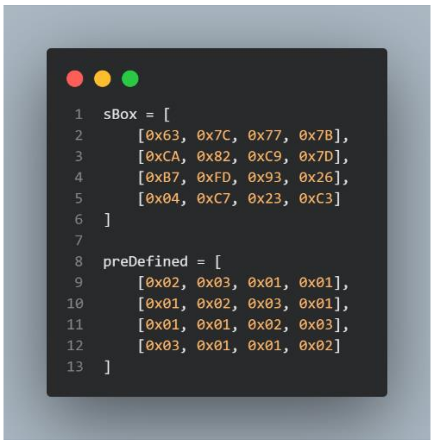


```
Original : MATDIS (~￣▽￣)~
Encrypted : [549308275576762440297528507240084910732951032368624630008[illegible]
169111999313205460767774443258391401997511092813953530378523339566893[illegible]
548699004863068586192411336195947386309512849815129462160953387082621[illegible]
462456516020366021634454680985390231974358436548733002298139177819362[illegible]
961913986913522232784805470184774499811004413176654126188008675099223[illegible]
362383, 857984934295554811425481625816295943196065336111248294955297[illegible]
810238790959976784300198553108287549137772517779310251437500429695772[illegible]
366580468524375452464248871155782511814460006468782870416650934589, 4[illegible]
938484309109816493551234798068311675230455967497824180553849299288918[illegible]
563748696817135471113325758607948382924978666183188647838031628030796[illegible]
5, 7255917823607546252573189897796514804814252084072040954526367508844[illegible]
275023344185638684642033991715297381237851840473437718044407741801012[illegible]
308853565551854178346228049546859590004787620678634772070432 1, 454687[illegible]
781814390935488355507642920252709135434219819875635861122077668901217[illegible]
323289860798336703951703437712690619588238997971095630890596298896531[illegible]
[illegible]
Decrypted : MATDIS (~￣▽￣)~
```

Figure 8. RSA Testing Result (source: writer's archive)

B. Implementing AES

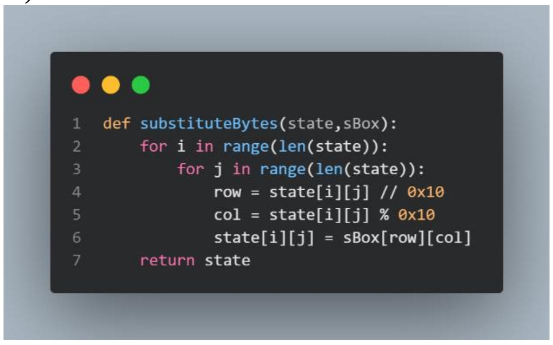
To implement AES algorithm from scratch needs a bigger complexity, therefore for implementation purposes, below is a simplified function of how the AES algorithm works.



```python
sBox = [
    [0x63, 0x7C, 0x77, 0x7B],
    [0xCA, 0x82, 0xC9, 0x7D],
    [0xB7, 0xFD, 0x93, 0x26],
    [0x04, 0xC7, 0x23, 0xC3]
]

preDefined = [
    [0x02, 0x03, 0x01, 0x01],
    [0x01, 0x02, 0x03, 0x01],
    [0x01, 0x01, 0x02, 0x03],
    [0x03, 0x01, 0x01, 0x02]
]
```

Figure 9. AES S-Box and Pre-Defined Matrix (source: writer's archive)

First, the S-box matrix and preDefined (used for Mix Column operation) are defined.

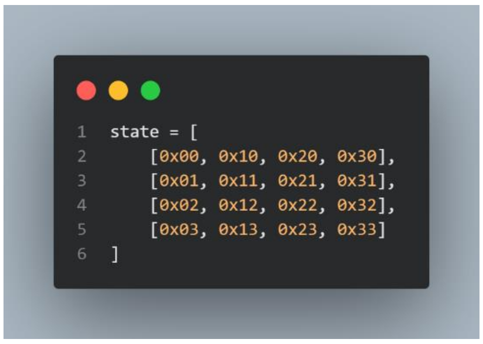


```python
def substituteBytes(state,sBox):
    for i in range(len(state)):
        for j in range(len(state)):
            row = state[i][j] // 0x10
            col = state[i][j] % 0x10
            state[i][j] = sBox[row][col]
    return state
```

Figure 10. AES Byte Substitution Function (source: writer's archive)

The subtituteByte function matches the input matrix element and switches it with the corresponding element in the **S-box** (*Fig.9*). '//0x10' and '% 0x10' are used to disassemble the element, so that the first digit refers to the S-box's row index and the second digit refers to the S-box's column index. As an example:

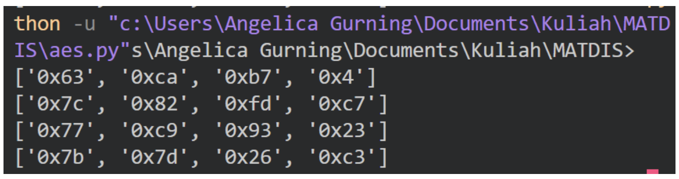
Say you have a state matrix,



```
state = [
    [0x00, 0x10, 0x20, 0x30],
    [0x01, 0x11, 0x21, 0x31],
    [0x02, 0x12, 0x22, 0x32],
    [0x03, 0x13, 0x23, 0x33]
]
```

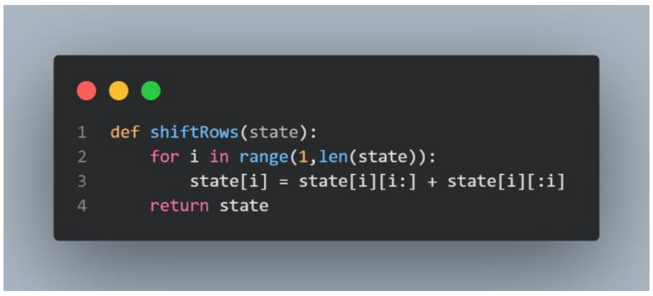
Figure 11. State Matrix
(source: writer's archive)

The output from the subsituteBytefunction is,



```
thon -u "c:\Users\Angelica Gurning\Documents\Kuliah\MATD
IS\aes.py"s\Angelica Gurning\Documents\Kuliah\MATDIS>
['0x63', '0xca', '0xb7', '0x4']
['0x7c', '0x82', '0xfd', '0xc7']
['0x77', '0xc9', '0x93', '0x23']
['0x7b', '0x7d', '0x26', '0xc3']
```

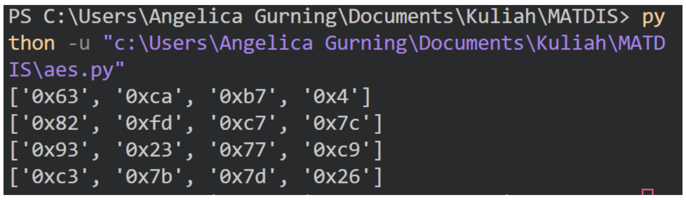
Figure 12. Byte Substitution Result
(source: writer's archive)



```python
def shiftRows(state):
    for i in range(1,len(state)):
        state[i] = state[i][i:] + state[i][:i]
    return state
```

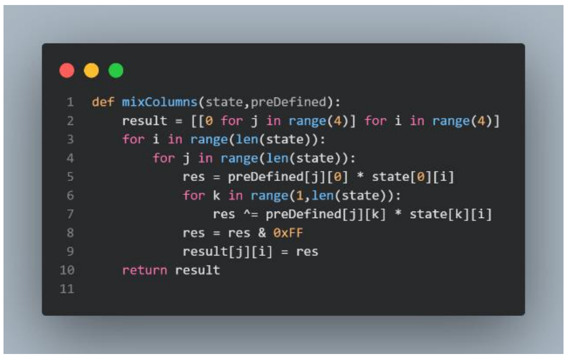
Figure 13. AES Shift Row Function
(source: writer's archive)

The shiftRows function shifts all the elements in each row (except row 0) circularly left. Below is the result of the shifted matrx using the matrix in Fig.12.



```
PS C:\Users\Angelica Gurning\Documents\Kuliah\MATDIS> py
thon -u "c:\Users\Angelica Gurning\Documents\Kuliah\MATD
IS\aes.py"
['0x63', '0xca', '0xb7', '0x4']
['0x82', '0xfd', '0xc7', '0x7c']
['0x93', '0x23', '0x77', '0xc9']
['0xc3', '0x7b', '0x7d', '0x26']
```

Figure 14. Shift Row Result
(source: writer's archive)

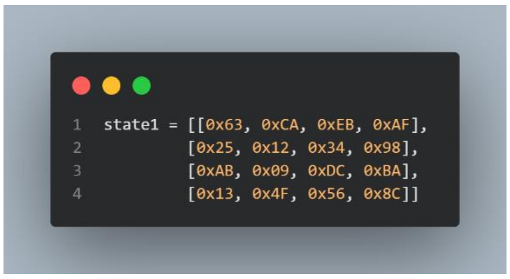


```python
def mixColumns(state,preDefined):
    result = [[0 for j in range(4)] for i in range(4)]
    for i in range(len(state)):
        for j in range(len(state)):
            res = preDefined[j][0] * state[0][i]
            for k in range(1,len(state)):
                res ^= preDefined[j][k] * state[k][i]
            res = res & 0xFF
            result[j][i] = res
    return result
```

Figure 15. AES Mix Column Function
(source: writer's archive)

The mix column will process state matrix by each column and perform a multiplication and XOR operation between state matrix and preDefined matrix *(Fig.9)*. For an in-depth explanation refer to Eq. (7). As an example:

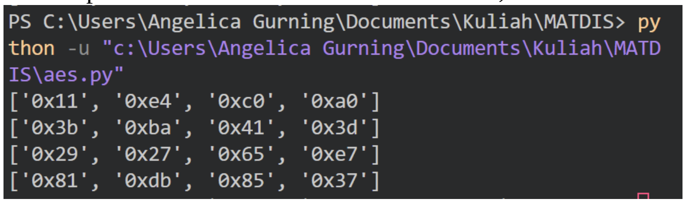
Say you have a state matrix,



```
state1 = [[0x63, 0xCA, 0xEB, 0xAF],
          [0x25, 0x12, 0x34, 0x98],
          [0xAB, 0x09, 0xDC, 0xBA],
          [0x13, 0x4F, 0x56, 0x8C]]
```

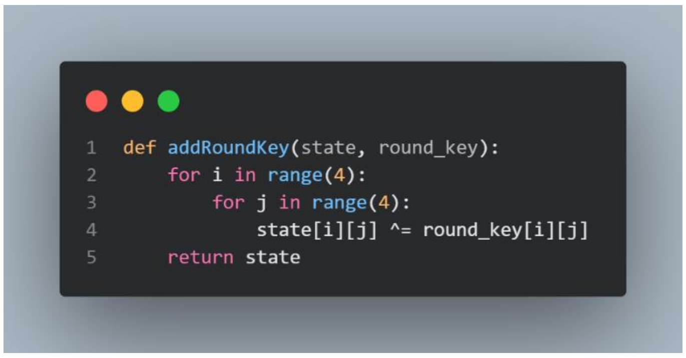
Figure 16. State Matrix
(source: writer's archive)

The output from mixColumns function will be,



```
PS C:\Users\Angelica Gurning\Documents\Kuliah\MATDIS> py
thon -u "c:\Users\Angelica Gurning\Documents\Kuliah\MATD
IS\aes.py"
['0x11', '0xe4', '0xc0', '0xa0']
['0x3b', '0xba', '0x41', '0x3d']
['0x29', '0x27', '0x65', '0xe7']
['0x81', '0xdb', '0x85', '0x37']
```

Figure 17. Mix Column Result
(source: writer's archive)

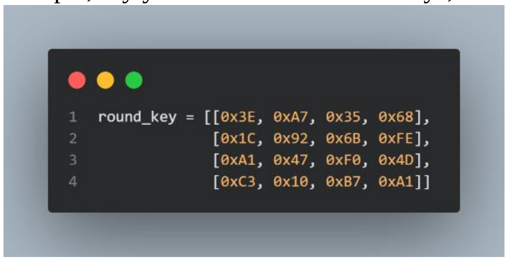


```python
def addRoundKey(state, round_key):
    for i in range(4):
        for j in range(4):
            state[i][j] ^= round_key[i][j]
    return state
```

Figure 18. AES Add Round Key Function
(source: writer's archive)

The last process is addRoundKey where it will perform a XOR operation between the current state matrix and the round_key from previous iteration.

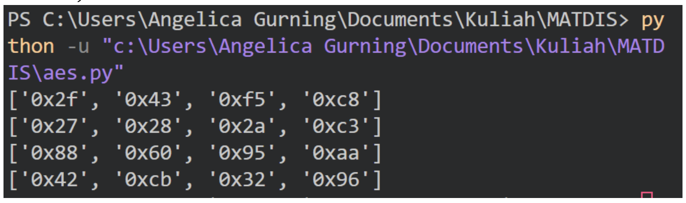
As an example, say you have a set of round keys,



```
round_key = [[0x3E, 0xA7, 0x35, 0x68],
             [0x1C, 0x92, 0x6B, 0xFE],
             [0xA1, 0x47, 0xF0, 0x4D],
             [0xC3, 0x10, 0xB7, 0xA1]]
```

Figure 19. Round Key Matrix
(source: writer's archive)

Once calculated using the state matrix from Fig. 17 the output will be,



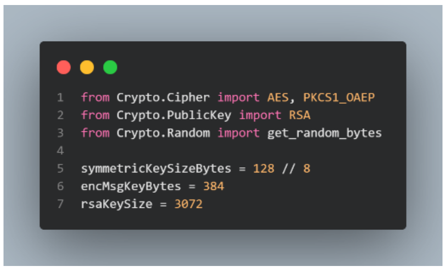
```
PS C:\Users\Angelica Gurning\Documents\Kuliah\MATDIS> py
thon -u "c:\Users\Angelica Gurning\Documents\Kuliah\MATD
IS\aes.py"
['0x2f', '0x43', '0xf5', '0xc8']
['0x27', '0x28', '0x2a', '0xc3']
['0x88', '0x60', '0x95', '0xaa']
['0x42', '0xcb', '0x32', '0x96']
```

Figure 20. Add Round Key Result
(source: writer's archive)

Conventionally, the 4 steps will be iterated according to the number of rounds in the calculation.

C. *Simulation*

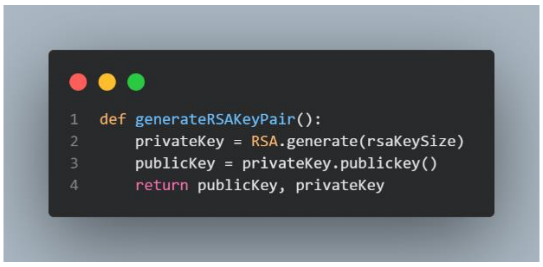
The Hybrid RSA-AES Simulation will be executed using python and the *pycryptodome module*.



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

symmetricKeySizeBytes = 128 // 8
encMsgKeyBytes = 384
rsaKeySize = 3072
```

Figure 21. Modules and Constant Declarations
(source: writer's archive)

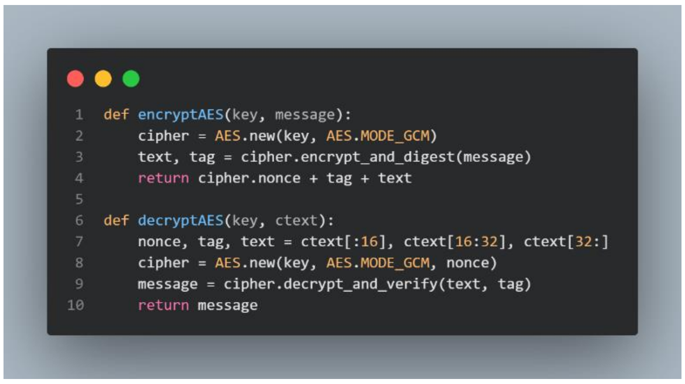
The modules used are imported to the environment along with declaring constants. *symmetricKeySizeBytes* refers to the AES key size (16 bytes), *encMsgKeyBytes* refers to the message bit size, and *rsaKeySize* refers to the RSA key bit size.



```
def generateRSAKeyPair():
    privateKey = RSA.generate(rsaKeySize)
    publicKey = privateKey.publickey()
    return publicKey, privateKey
```

Figure 22. Generating RSA Key Pair
(source: writer's archive)

The function will return a tuple of publicKey and privateKey generated with RSA algorithm.

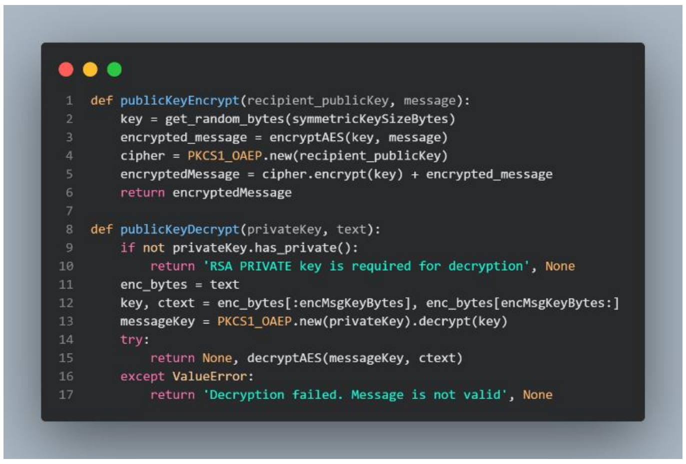


```
def encryptAES(key, message):
    cipher = AES.new(key, AES.MODE_GCM)
    text, tag = cipher.encrypt_and_digest(message)
    return cipher.nonce + tag + text

def decryptAES(key, ctext):
    nonce, tag, text = ctext[:16], ctext[16:32], ctext[32:]
    cipher = AES.new(key, AES.MODE_GCM, nonce)
    message = cipher.decrypt_and_verify(text, tag)
    return message
```

Figure 23. Encrypting and Decrypting Messages (AES)
(source: writer's archive)

The *encryptAES* function creates a cipher object using GCM (Galois Counter Mode), the message is then encrypted. The function returns a *nonce (number used once)*, security tag and encrypted message. Nonce and the tag are generated to enhance security.

The *decryptAES* function breaks down the encrypted message and uses the key to generate a cipher object. The message is then decrypted using the cipher object.

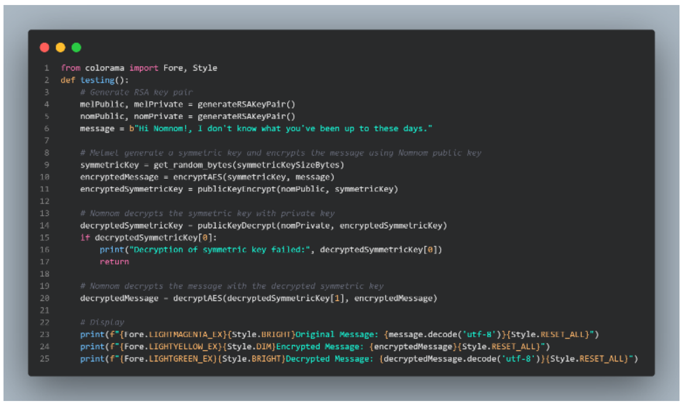


```
def publicKeyEncrypt(recipient_publicKey, message):
    key = get_random_bytes(symmetricKeySizeBytes)
    encrypted_message = encryptAES(key, message)
    cipher = PKCS1_OAEP.new(recipient_publicKey)
    encryptedMessage = cipher.encrypt(key) + encrypted_message
    return encryptedMessage

def publicKeyDecrypt(privateKey, text):
    if not privateKey.has_private():
        return 'RSA PRIVATE key is required for decryption', None
    enc_bytes = text
    key, ctext = enc_bytes[:encMsgKeyBytes], enc_bytes[encMsgKeyBytes:]
    messageKey = PKCS1_OAEP.new(privateKey).decrypt(key)
    try:
        return None, decryptAES(messageKey, ctext)
    except ValueError:
        return 'Decryption failed. Message is not valid', None
```

Figure 24. Encrypting and Decrypting Key
(source: writer's archive)

The *publicKeyEncrypt* takes in the receiver's public key which will be used to generate a new cipher object. A random symmetric key is generated and the *encryptAES* function is called to create the encrypted message. The function will then finalize the encryption by concatenating the symmetric key and the message.

The *publicKeyDecrypt* will check the existence of the user's private key. If valid, the function will take in the encrypted text and split it into the key and text itself. The encrypted key is decrypted using the receiver's private key, resulting in the final key. Once the final key is obtained, the function will call *decryptAES* to decrypt the message. 'PKCS1_0AEP' is a padding scheme (for security) commonly used in RSA.



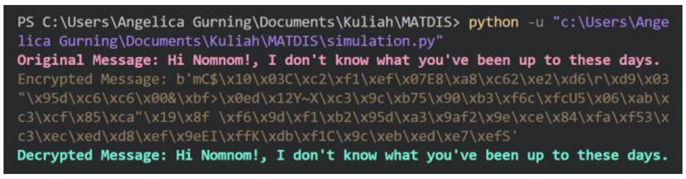
```python
from colorama import Fore, Style
def testing():
    # Generate RSA key pair
    melPublic, melPrivate = generateRSAKeyPair()
    nomPublic, nomPrivate = generateRSAKeyPair()
    message = b"Hi Nomnom!, I don't know what you've been up to these days."

    # Melmel generate a symmetric key and encrypts the message using Nomnom public key
    symmetricKey = get_random_bytes(symmetricKeySizeBytes)
    encryptedMessage = encryptAES(symmetricKey, message)
    encryptedSymmetricKey = publicKeyEncrypt(nomPublic, symmetricKey)

    # Nomnom decrypts the symmetric key with private key
    decryptedSymmetricKey = publicKeyDecrypt(nomPrivate, encryptedSymmetricKey)
    if decryptedSymmetricKey[0]:
        print("Decryption of symmetric key failed:", decryptedSymmetricKey[0])
        return

    # Nomnom decrypts the message with the decrypted symmetric key
    decryptedMessage = decryptAES(decryptedSymmetricKey[1], encryptedMessage)

    # Display
    print(f"{Fore.LIGHTMAGENTA_EX}{Style.BRIGHT}Original Message: {message.decode('utf-8')}{Style.RESET_ALL}")
    print(f"{Fore.LIGHTYELLOW_EX}{Style.DIM}Encrypted Message: {encryptedMessage}{Style.RESET_ALL}")
    print(f"{Fore.LIGHTGREEN_EX}{Style.BRIGHT}Decrypted Message: {decryptedMessage.decode('utf-8')}{Style.RESET_ALL}")
```

Figure 25. Hybrid RSA-AES Testing Scheme
(source: writer's archive)

There will be two user personas, one will be referred to as 'Melmel', the other will be referred to as 'Nomnom'. Melmel will attempt to send a message to Nomnom.

The program will first generate a set of public key and private key individually. *Melmel* will write a message, 'b' is added to the string to mark it as a byte object to create an ASCII representation (will be encoded again once displayed). *Melmel* will encrypt the message using *Nomnom's* public key as well as generating a symmetric key. *Nomnom* will decrypt the key by using their private key and finally, decrypting the message.

The *colorama* module is only for displaying purposes and has no effect on the algorithm.



```
PS C:\Users\Angelica Gurning\Documents\Kuliah\MATDIS> python -u "c:\Users\Ange
lica Gurning\Documents\Kuliah\MATDIS\simulation.py"
Original Message: Hi Nomnom!, I don't know what you've been up to these days.
Encrypted Message: b'mC$\x10\x03C\xc2\xf1\xef\x07E8\xa8\xc62\xe2\xd6\r\xd9\x03
"\x95d\xc6\xc6\x00&\xbf>\x0ed\x12Y~X\xc3\x9c\xb75\x90\xb3\xf6c\xfcU5\x06\xab\x
c3\xcf\x85\xca"\x19\x8f \xf6\x9d\xf1\xb2\x95d\xa3\x9af2\x9e\xce\x84\xfa\xf53\x
c3\xec\xed\xd8\xef\x9eEI\xffK\xdb\xf1C\x9c\xeb\xed\xe7\xefS'
Decrypted Message: Hi Nomnom!, I don't know what you've been up to these days.
```

Figure 26. Final Result
(source: writer's archive)

V. CONCLUSION

Data security has become a compulsory mechanism to ensure safe and secure data transfers. Especially, with the rapid growth of social media and messaging services, users must receive protection for their data. Hybrid RSA-AES encryption has developed into the standard security protocol in messaging applications. Utilizing both symmetry and asymmetry encryption which guarantees the safety of data transmission, whilst mitigating from malicious and hostile interception or threats.

Nevertheless, room for improvement is always required. With technology getting more sophisticated, threats are also developing. With that, security protocols also need to be updated and revolutionized.

VII. ACKNOWLEDGMENT

Before everything, the researcher would like to thank the Lord for His Grace and kindness. The researcher would also like to express the biggest gratitude for Dr. Nur Ulfa Maulidevi, S.T, M.Sc., for acting as the researcher's Discrete Mathematics Lecturer and for sharing the knowledge and guidance which led into helping the researcher to write this essay.

Not to mention, to other Discrete Mathematics lecturers who also have contributed to sharing their knowledge.

Finally, the researcher would give thanks to her fellow peers for giving support and confidence in finishing this essay.

REFERENCES

[1] M. B. KILIÇ, "Encryption Methods and Comparison of Popular Chat Applications," p. 53, 2021.

[2] K. K. S. Agoorukasetty Adithya, "Applications of RSA and AES256 in End-to-End encryption using DiffieHellman Key Exchange," pp. 1219,1220, 2022.

[3] P. V. K. Kalyani Ganesh Kadam, "AN IMPLEMENTATION OF HYBRID ENCRYPTION-DECRYPTION (RSA WITH AES AND SHA256) FOR USE IN DATA EXCHANGE BETWEEN CLIENT APPLICATIONS AND WEB SERVICES," pp. 52-54, 2015.

[4] A. M. S. Ammar Hammad Ali, "Design of Secure Chatting Application with End to End Encryption for Android Platform," p. 23, 2017.

[5] P. R. Mihir Bellare, Introduction to Modern Cryptography, California: 16, 50-51, 204, 2005.

[6] H. A. a. N. Q. M. Mohammed N. Alenezi, "Symmetric Encryption Algorithms: Review and Evaluation study," pp. 256-257, 2020.

[7] J. O'Mora, "Demystifying the Signal Protocol for End-to-End Encryption (E2EE)," Medium, 18 8 2017. [Online]. Available: https://medium.com/@justinomora/demystifying-the-signal-protocol-for-end-to-end-encryption-e2ee-ad6a567e6cb4. [Accessed 8 12 2023].

[8] Cooby, "WhatsApp Statistics," TrueList, [Online]. Available: https://truelist.co/blog/texting-statistics/#:~:text=How%20many%20texts%20are%20sent,to%20270%2C000%20texts%20every%20second.. [Accessed 10 November 2023].

[9] GeeksforGeeks, "Difference between Block Cipher and Stream Cipher," [Online]. Available: https://www.geeksforgeeks.org/difference-between-block-cipher-and-stream-cipher/. [Accessed 30 November 2023].

[10] PyPI, "hybrid-rsa-aes," [Online]. Available: https://pypi.org/project/hybrid-rsa-aes/. [Accessed 9 December 2023].

[11] GeeksforGeeks, "Implementation of Diffie-Hellman Algorithm," [Online]. Available: https://www.geeksforgeeks.org/implementation-diffie-hellman-algorithm/. [Accessed 9 December 2023].

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 9 Desember 2023

[illegible]

Angelica Kierra Ninta Gurning/13522048