Report computer security

|  |  |
| --- | --- |
| Student name | Position |
| NG JUN KAI | Group leader |
| LOH LOG TIAN | Member |
| NUR ARIFSHAH BIN ROHIZAM | Member |
| LUCAS LEE KWONG YEK | Member |

Contents

[RSA encryption and decryption 3](#_Toc184029670)

[RSA introduction 3](#_Toc184029671)

[RSA example 4](#_Toc184029672)

[RSA vulnerability 5](#_Toc184029673)

[SHA introduction 5](#_Toc184029674)

[SHA example 6](#_Toc184029675)

[RSA limitation 8](#_Toc184029676)

[AES introduction 8](#_Toc184029677)

[AES example 9](#_Toc184029678)

[Code 14](#_Toc184029679)

[testing,py 14](#_Toc184029680)

[RSA.py 27](#_Toc184029681)

# RSA encryption and decryption

## RSA introduction

## RSA example

# RSA vulnerability

## SHA introduction

SHA256withRSA is a hybrid cryptographic algorithm that leverages the SHA-256 hashing algorithm and the RSA digital signature scheme. It utilizes SHA-256 to generate a hash value for the data and then signs the hash using RSA with a private key. The result signature can be verified with a corresponding public key, ensuring the integrity and authenticity of the data.

SHA-256 is a member of the SHA2 family of secure hash functions, and there are not currently any cryptographic weaknesses publicly known for SHA2. It might be less secure than SHA-512, but 256 bits is already completely impractical to brute force. The only viable attacks would require finding a weakness in the hash algorithm itself, and it's not necessarily the case that SHA-512 would be more resistant to such an attack than SHA-256. There is a new SHA3 standard, but it's not yet widely implemented, so browsers probably wouldn't be able to verify the certificate's signature at all if they used SHA3 in the signing algorithm.

RSA is a current standard for public-key cryptography, and a properly generated 2048-bit RSA key is strong enough to resist factoring for decades. We can use a 4096-bit key if we want to (it'll take a lot longer to generate and slightly longer to use, but once the certificate's signature is verified, that doesn't matter anymore), and that would take even longer to break. However, neither certificate is valid for more than two years anyhow.

## SHA example

**Step 1: Convert Message to Binary**

Input: "hello world"

ASCII: 104 101 108 108 111 32 119 111 114 108 100

Binary: 01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111 01110010 01101100 01100100

**Step 2: Padding**

SHA-256 requires the input length to be a multiple of 512 bits:

Append a 1 bit:  
...01110010 01101100 01100100 1

Add 0 bits to make the total length 448 bits (mod 512):  
...01110010 01101100 01100100 10000000 00000000 ...

Append the original message length (in bits) as a 64-bit binary:  
Length = 88 bits (11 characters × 8).  
Append: 00000000 00000000 00000000 00000000 00000000 00000000 01011000.

Final padded message (512 bits):

01101000 01100101 01101100 01101100 01101111 00100000 01110111 01101111

01110010 01101100 01100100 10000000 00000000 ... 01011000

**Step 3: Initialize Constants**

SHA-256 uses 8 initial hash values (H0 to H7) and 64 constants (K0 to K63). These values are derived from fractional parts of square roots of prime numbers.

Initial hash values (H):

H0 = 6a09e667, H1 = bb67ae85, H2 = 3c6ef372, ...

Round constants (K):

K0 = 428a2f98, K1 = 71374491, ...

**Step 4: Process Each Chunk**

The padded message is divided into 512-bit chunks. For each chunk:

1. **Prepare Message Schedule**:

Expand the 512-bit chunk into 64 words (W0 to W63) of 32 bits each.

For the first 16 words: Use the chunk's original 512 bits.

For W16 to W63:

Wt = σ1(Wt-2) + Wt-7 + σ0(Wt-15) + Wt-16

where:

σ0(x) = ROTR7(x) XOR ROTR18(x) XOR SHR3(x)

σ1(x) = ROTR17(x) XOR ROTR19(x) XOR SHR10(x)

1. **Initialize Working Variables**:

a = H0, b = H1, c = H2, ..., h = H7

1. **Compression Loop** (64 rounds): For each round t (0 to 63):

T1 = h + Σ1(e) + Ch(e, f, g) + Kt + Wt

T2 = Σ0(a) + Maj(a, b, c)

h = g, g = f, f = e, e = d + T1

d = c, c = b, b = a, a = T1 + T2

where:

Σ0(x) = ROTR2(x) XOR ROTR13(x) XOR ROTR22(x)

Σ1(x) = ROTR6(x) XOR ROTR11(x) XOR ROTR25(x)

Ch(x, y, z) = (x AND y) XOR ((NOT x) AND z)

Maj(x, y, z) = (x AND y) XOR (x AND z) XOR (y AND z)

1. **Update Hash Values**:

H0 = H0 + a, H1 = H1 + b, ..., H7 = H7 + h

**Step 5: Output Final Hash**

After processing all chunks, concatenate H0 to H7 to produce the final 256-bit hash:

b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9

# RSA limitation

## AES introduction

Encrypting big files directly with techniques like RSA is wasteful and virtually impossible since RSA is meant to encrypt little quantities of data, such as cryptographic keys or brief communications. RSA uses difficult mathematical operations (modular exponentiation) over big integers, making it computationally costly and limiting the amount of data that can be encrypted to a fraction of the RSA key size (for example, a 2048-bit RSA key can only encrypt data up to around 256 bytes).

AES is a symmetric method that employs the same 128, 192, or 256-bit key for encryption and decryption (the security of an AES system grows exponentially with key length). With a 128-bit key, cracking AES by testing each of the 2128 possible key values (a "brute force" assault) is so computationally demanding that even the fastest supercomputer would take more than 100 trillion years to complete. In reality, AES has never been broken and, according to current technical trends, is anticipated to stay safe for many years to come.

The AES algorithm performs a sequence of mathematical modifications on each 128-bit data block. Because to its minimal processing needs, AES may be utilized with consumer computing devices like laptops and smartphones, as well as to swiftly encrypt massive volumes of data. For example, the IBM z14 mainframe series employs AES to allow ubiquitous encryption, which encrypts all data in the system, whether at rest or in transit.

Large files are encrypted using AES (Advanced Encryption Standard). AES is ideal for encrypting huge datasets since it processes data in blocks and is computationally lighter than RSA. To combine the strengths of RSA and AES, a hybrid encryption method is utilized.

## AES example

**Encrypt “hello world”**

**First step padding input into 16 byte and generate random key in 16 byte**

1. Convert "hello world" to bytes:

ASCII: hello world → [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

Pad to 16 bytes: [104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100, 5, 5, 5, 5, 5]

1. Represent as a 4x4 matrix (AES state):

[104, 111, 114, 5]

[101, 32, 108, 5]

[108, 119, 100, 5]

[108, 111, 5, 5]

1. Key (example, 16 bytes):

[0x2b, 0x7e, 0x15, 0x16]

[0x28, 0xae, 0xd2, 0xa6]

[0xab, 0xf7, 0x32, 0xfe]

[0xf9, 0x19, 0x59, 0xc7]

**AES Steps**

**Step 1: Add Round Key**

Each byte in the state is XORed with the corresponding byte of the key:

Example XOR of 104 (0x68) with 0x2b: 0x68 ^ 0x2b = 0x43.

Resulting state:

[0x43, 0x1d, 0x61, 0x13]

[0x55, 0x9e, 0xba, 0xa3]

[0x87, 0x86, 0x6a, 0xfb]

[0xb1, 0x18, 0x5c, 0xc2]

**Step 2: SubBytes**

Each byte is substituted using the AES S-Box. For example:

0x43 → S\_BOX[0x43] = 0xa5.

After substitution:

[0xa5, 0x69, 0x85, 0x7c]

[0xfe, 0x91, 0x3a, 0x6b]

[0xc6, 0xca, 0x9d, 0xb2]

[0x47, 0x3f, 0x72, 0xde]

**Step 3: Shift Rows**

The rows are shifted left by 0, 1, 2, and 3 positions:

[0xa5, 0x69, 0x85, 0x7c]

[0x91, 0x3a, 0x6b, 0xfe]

[0x9d, 0xb2, 0xc6, 0xca]

[0xde, 0x47, 0x3f, 0x72]

**Step 4: Mix Columns**

Each column is multiplied by a fixed matrix in GF(2^8). Example for the first column:

[a5, 91, 9d, de] × [2, 3, 1, 1]

Result (simplified):

[0x57, 0x4c, 0x19, 0x73]

After Mix Columns:

[0x57, 0xb1, 0x34, 0x9f]

[0x4c, 0x6d, 0xab, 0x3a]

[0x19, 0x82, 0xd7, 0x1e]

[0x73, 0x5e, 0xf8, 0xa9]

**Steps 5-10: Repeat for 9 More Rounds**

The above steps are repeated with additional key scheduling for each round. In the final round, **Mix Columns** is skipped.

**Result**

The encrypted output is:

[0x47, 0x95, 0x2c, 0x7e]

[0x6d, 0x8b, 0x91, 0x4f]

[0xa8, 0xf3, 0x4e, 0x76]

[0x1d, 0xe9, 0x32, 0xab]

**Decrypt Ciphertext**

**AES Decryption Steps**

**Step 1: Add Round Key (Last Round Key)**

XOR the ciphertext with the final round key (computed during key schedule). Example for the first byte:

0x47 ^ 0x2b = 0x6c.

Resulting state:

[0x6c, 0xe1, 0x39, 0x68]

[0x45, 0x25, 0x43, 0xe9]

[0x03, 0x04, 0x7c, 0x88]

[0x24, 0xf0, 0x6b, 0x6c]

**Step 2: Reverse Shift Rows**

Reverse the row shifts:

* Row 0: No shift.
* Row 1: Shift right by 1.
* Row 2: Shift right by 2.
* Row 3: Shift right by 3.

Reversed:

[0x6c, 0xe1, 0x39, 0x68]

[0xe9, 0x45, 0x25, 0x43]

[0x7c, 0x88, 0x03, 0x04]

[0x6c, 0x24, 0xf0, 0x6b]

**Step 3: Reverse SubBytes**

Use the **inverse S-Box** to substitute bytes back to their original values. For example:

0x6c → INV\_S\_BOX[0x6c].

Resulting state:

[0xa5, 0x69, 0x85, 0x7c]

[0xfe, 0x91, 0x3a, 0x6b]

[0xc6, 0xca, 0x9d, 0xb2]

[0x47, 0x3f, 0x72, 0xde]

**Step 4: Reverse Mix Columns**

Apply the **inverse Mix Columns** transformation. For example:

[0xa5, 0xfe, 0xc6, 0x47] × Inverse Mix Matrix.

Resulting state:

[0x43, 0x1d, 0x61, 0x13]

[0x55, 0x9e, 0xba, 0xa3]

[0x87, 0x86, 0x6a, 0xfb]

[0xb1, 0x18, 0x5c, 0xc2]

**Steps 5-10: Repeat for Previous Rounds**

Repeat steps **Add Round Key**, **Shift Rows**, **SubBytes**, and **Mix Columns** for all 10 rounds in reverse order, applying the corresponding round keys.

**Final Step: Remove Padding**

After decrypting all rounds, the state returns to:

[104, 101, 108, 108]

[111, 32, 119, 111]

[114, 108, 100, 5]

[ 5, 5, 5, 5]

Convert back to bytes and remove padding (0x05):

[104, 101, 108, 108, 111, 32, 119, 111, 114, 108, 100]

Convert to ASCII: "hello world".

## Code

### testing,py

import base64

import json

import queue

import socket

import threading

import time

import tkinter as tk

from tkinter import filedialog, messagebox

import random

import math

import uuid

import RSA

import os

#use p=7,q=3, should get from Certificate autority

CA\_e = 7

CA\_d = 103

CA\_n= 143

SHA256 = RSA.SHA256()

mac\_addr = ':'.join(['{:02x}'.format((uuid.getnode() >> ele) & 0xff) for ele in range(0, 8 \* 6, 8)][::-1])

if(mac\_addr != "f8:5e:a0:bd:cc:8d"):

    mac\_addr = "f8:5e:a0:bd:cc:8d"

    client = socket.socket(socket.AF\_BLUETOOTH, socket.SOCK\_STREAM, socket.BTPROTO\_RFCOMM)

    print("Waiting for connection...")

    client.connect(("f8:5e:a0:bd:cc:8d", 7))

    print(f"Connected to: {mac\_addr}")

    def receive():

        while True:

            try:

                message = client.recv(1024).decode('utf-8')  # Receive and decode the message

                decoded\_message = decode\_message(message)  # Process the received message

                if decoded\_message:  # If decoding was successful, update the listbox

                    message\_listbox.insert(tk.END, decoded\_message)

            except Exception as e:

                # Log the error and close the client socket

                print(f"An error occurred: {e}")

                client.close()

                break

    def write(message):

        json\_message = json.dumps(message)  # Convert the dictionary to a JSON string

        client.send(json\_message.encode('utf-8'))  # Send the JSON string

else:

    server = socket.socket(socket.AF\_BLUETOOTH, socket.SOCK\_STREAM, socket.BTPROTO\_RFCOMM)

    server.bind((mac\_addr, 7))

    server.listen(1)

    print("Waiting for connection...")

    while True:

        try:

            client, client\_address = server.accept()

            break

        except:

            print("connection fail")

    print(f"Connected to: {client\_address}")

    def receive():

        while True:

            try:

                message = client.recv(1024).decode('utf-8')  # Receive and decode the message

                decoded\_message = decode\_message(message)  # Process the received message

                if decoded\_message:  # If decoding was successful, update the listbox

                    message\_listbox.insert(tk.END, decoded\_message)

            except Exception as e:

                # Log the error and close the client socket

                print(f"An error occurred: {e}")

                client.close()

                server.close()

                break

    def write(message):

        json\_message = json.dumps(message)

        client.send(json\_message.encode('utf-8'))

def decode\_message(message):

    try:

        message = json.loads(message)  # Convert JSON string to a dictionary

        if (message["type"] == "key"):

            #for public key

            global r\_public\_key

            r\_public\_key = message["public key"]

            global r\_n

            r\_n = message["n"]

            recieve\_key\_label.config(text=str(r\_public\_key))

        elif (message["type"] == "message"):

            #for message

            MD1 = RSA.RSAdecryption(message["Digital signature"], CA\_d, CA\_n)

            MD2 = SHA256.compute(message["content"])

            print("MD1: ",MD1)

            print("DS: ",message["Digital signature"])

            print("MD1: ", MD2)

            print("encrypted message: ",message["content"])

            if (MD1 == MD2):

                message = RSA.RSAdecryption(message["content"], key["decryption key"], key["n"])

                return message

            else:

                print("message not match")

            tk.messagebox.showwarning("Warning", "decode error")

        elif(message["type"] == "file"):

            #for file

            MD1 = RSA.RSAdecryption(message["Digital signature"], CA\_d, CA\_n)

            MD2 = SHA256.compute(message["key"])

            print("DS: ",message["Digital signature"])

            print("MD1: ",MD1)

            print("MD2: ", MD2)

            print("encrypted key: ",message["key"])

            if (MD1 == MD2):

                #get aes key

                AES\_key = RSA.RSAdecryption(message["key"],key["decryption key"], key["n"])

                print("AES key",AES\_key)

                #get file data

                file\_data = RSA.aes\_decrypt(message["content"], AES\_key)

                print(file\_data)

                #get file name

                file\_name = message["file name"]

                try:

                    recieve\_file = open(file\_name,"xb")

                except IOError:

                    recieve\_file = open(file\_name,"wb")

                recieve\_file.write(file\_data)

                recieve\_file.close()

                return file\_name

            else:

                print("key not match")

        else:

            tk.messagebox.showwarning("Warning", "Unknown message type")

    except json.JSONDecodeError:

        tk.messagebox.showwarning("Warning", "Invalid message format")

    return True

def is\_prime(n):

    if n <= 1:

        return False

    for i in range(2, int(math.sqrt(n)) + 1):

        if n % i == 0:

            return False

    return True

def auto\_generate\_pq():

    p\_entry.delete(0, tk.END)

    q\_entry.delete(0, tk.END)

    while True:

            p = random.choice(range(10,50))

            q = random.choice(range(10,50))

            if (p==q):

                continue

            if (is\_prime(p) and is\_prime(q)):

                break

    p\_entry.insert(0, p)

    q\_entry.insert(0, q)

def generate\_key():

    p = eval(p\_entry.get())

    q = eval(q\_entry.get())

    if (p\_entry.get() == "") or (q\_entry.get() == ""):

        tk.messagebox.showwarning("Warning", "Please enter p & q")

    else:

        if (p==q):

            tk.messagebox.showwarning("Warning", "q and q is same")

        elif (is\_prime(p) and is\_prime(q)):

            n = p \* q

            fi = (p - 1) \* (q - 1)

            e = 2

            while True:

                if math.gcd(e, fi) == 1:

                    break

                e += 1

            for d in range(1, fi):

                if (e \* d) % fi == 1:

                    break

            global key

            key ={

                "encryption key":e,

                "decryption key":d,

                "n":n

            }

            private\_label.config(text = d)

            public\_label.config(text = e)

        else:

            tk.messagebox.showwarning("Warning", "q and q is not prime number")

def send\_public\_key():

    message={

        "type":"key",

        "public key":key["encryption key"],

        "n":key["n"]

    }

    print("send\_public\_key : success")

    write(message)

def send\_message():

    message = m\_entry.get()

    encrypted\_message = RSA.RSAencryption(message,r\_public\_key,r\_n)

    MD1 = SHA256.compute(encrypted\_message)

    DS = RSA.RSAencryption(MD1, CA\_e, CA\_n)

    print("MD1: ",MD1)

    print("DS: ",DS)

    message ={

        "type":"message",

        "content": encrypted\_message,

        "Digital signature":DS

    }

    write(message)

    tk.messagebox.showinfo("Success")

def browseFiles():

    filename = filedialog.askopenfilename(initialdir = "/",

                                          title = "Select a File",

                                          filetypes = (("Text files",

                                                        "\*.txt\*"),

                                                       ("all files",

                                                        "\*.\*")))

    # Change label contents

    f\_entry.insert(0,filename)

def send\_file():

    #get file content

    file\_path = f\_entry.get()

    file\_name = file\_path.split('/')[-1]

    with open(file\_path, "rb") as file:

        plaintext = file.read()

        print("plaintext : ",plaintext)

    AES\_key = os.urandom(16)

    print("AES\_key",AES\_key)

    ciphertext= RSA.aes\_encrypt(plaintext, AES\_key)

    #RSA encryption

    AES\_key = base64.b64encode(AES\_key).decode('utf-8')

    encrypted\_key = RSA.RSAencryption(AES\_key,r\_public\_key,r\_n)

    MD1 = SHA256.compute(encrypted\_key)

    DS = RSA.RSAencryption(MD1, CA\_e, CA\_n)

    #make into dictionary

    print("AES\_MD1: ",MD1)

    print("AES\_DS: ",DS)

    print("ciphertext",ciphertext)

    message ={

        "type":"file",

        "key": encrypted\_key,

        "Digital signature":DS,

        "file name":file\_name,

        "content":ciphertext,

    }

    write(message)

    tk.messagebox.showinfo("Success")

receive\_thread = threading.Thread(target=receive)

receive\_thread.daemon = True  # Ensure the thread exits when the main program does

receive\_thread.start()

# Main Application Window

root = tk.Tk()

root.title("RSA Encryption")

root.geometry("600x700")

root.resizable(False, False)

# p Entry

tk.Label(root, text="Enter p:").grid(row=0, column=0, padx=10, pady=10, sticky="w")

p\_entry = tk.Entry(root, width=20)

p\_entry.grid(row=0, column=1, padx=10, pady=10)

# q Entry

tk.Label(root, text="Enter q:").grid(row=0, column=2, padx=10, pady=10, sticky="w")

q\_entry = tk.Entry(root, width=20)

q\_entry.grid(row=0, column=3, padx=10, pady=10)

# Auto-generate p & q Button

auto\_button = tk.Button(root, text="Auto Generate p & q", command=auto\_generate\_pq)

auto\_button.grid(row=0, column=4, pady=10)

# Send Encryption Key Button

send\_key\_button = tk.Button(root, text="gennerate Encryption and decryption Key", command=generate\_key)

send\_key\_button.grid(row=1, column=0, columnspan=2,pady=10, padx=10)

send\_key\_button = tk.Button(root, text="send Encryption Key(public key)", command=send\_public\_key)

send\_key\_button.grid(row=1, column=2, columnspan=2,pady=10, padx=10)

tk.Label(root, text="the encryption key:").grid(row=2, column=0, columnspan=2, padx=10, pady=10, sticky="w")

private\_label = tk.Label(root, text="")

private\_label.grid(row=2, column=2, columnspan=1,padx=10, pady=10, sticky="w")

tk.Label(root, text="the decryption key:").grid(row=2, column=3, columnspan=2, padx=10, pady=10, sticky="w")

public\_label = tk.Label(root, text="")

public\_label.grid(row=2, column=5, columnspan=1, padx=10, pady=10, sticky="w")

tk.Label(root, text="encryption key recieve").grid(row=3, column=0, columnspan=2, padx=10, pady=10, sticky="w")

recieve\_key\_label = tk.Label(root, text="")

recieve\_key\_label.grid(row=3, column=2, columnspan=1, padx=10, pady=10, sticky="w")

# Message Listbox

tk.Label(root, text="message").grid(row=4, column=0, padx=10, pady=10,columnspan=2, sticky="w")

message\_listbox = tk.Listbox(root, height=10, width=80)

message\_listbox.grid(row=5, column=0, columnspan=5, padx=10, pady=10)

m\_entry = tk.Entry(root, width=80)

m\_entry.grid(row=6, column=0, columnspan=5, padx=10, pady=10)

# send message button

message\_button = tk.Button(root, text="send message",command=send\_message)

message\_button.grid(row=8, column=0, columnspan=6,pady=10)

# send file button

f\_entry = tk.Entry(root, width=80)

f\_entry.grid(row=9, column=0, columnspan=5, padx=10, pady=10)

browse\_button = tk.Button(root, text="browse",command=browseFiles)

browse\_button.grid(row=10, column=0, columnspan=3,pady=10)

file\_button = tk.Button(root, text="send file",command=send\_file)

file\_button.grid(row=10, column=1, columnspan=3,pady=10)

# Start the application

root.mainloop()

### RSA.py

import hashlib

import math

from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

from Crypto.Random import get\_random\_bytes

from base64 import b64encode, b64decode

def RSAencryption(message,public\_key,n):

    if not isinstance(message, str):

        message=str(message)

    encrypted\_message = [pow(ord(char), public\_key, n) for char in message]

    return encrypted\_message

def RSAdecryption(encrypted\_message,private\_key,n):

    decrypted\_message = ''.join(chr(pow(char, private\_key, n)) for char in encrypted\_message)

    return decrypted\_message

class SHA256:

    def \_\_init\_\_(self):

        # Initial hash values (first 32 bits of the fractional parts of the square roots of the first 8 primes)

        self.h = [

            0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,

            0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19

        ]

        # Round constants (first 32 bits of the fractional parts of the cube roots of the first 64 primes)

        self.k = [

            0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,

            0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01,

            0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,

            0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,

            0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152,

            0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,

            0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc,

            0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,

            0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,

            0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08,

            0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,

            0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,

            0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

        ]

    def \_right\_rotate(self, value, shift):

        """Right rotate a 32-bit value by a given shift."""

        return ((value >> shift) | (value << (32 - shift))) & 0xffffffff

    def \_pad(self, message):

        """Pad the message to make its length a multiple of 512 bits."""

        message = bytearray(message, 'utf-8')

        message\_len = len(message) \* 8

        message.append(0x80)

        while (len(message) \* 8) % 512 != 448:

            message.append(0x00)

        message += message\_len.to\_bytes(8, 'big')

        return message

    def \_chunks(self, message, chunk\_size):

        """Divide the message into chunks of fixed size."""

        for i in range(0, len(message), chunk\_size):

            yield message[i:i + chunk\_size]

    def \_process\_chunk(self, chunk):

        """Process a single 512-bit chunk."""

        w = [0] \* 64

        for i in range(16):

            w[i] = int.from\_bytes(chunk[i \* 4:(i + 1) \* 4], 'big')

        for i in range(16, 64):

            s0 = self.\_right\_rotate(w[i - 15], 7) ^ self.\_right\_rotate(w[i - 15], 18) ^ (w[i - 15] >> 3)

            s1 = self.\_right\_rotate(w[i - 2], 17) ^ self.\_right\_rotate(w[i - 2], 19) ^ (w[i - 2] >> 10)

            w[i] = (w[i - 16] + s0 + w[i - 7] + s1) & 0xffffffff

        a, b, c, d, e, f, g, h = self.h

        for i in range(64):

            s1 = self.\_right\_rotate(e, 6) ^ self.\_right\_rotate(e, 11) ^ self.\_right\_rotate(e, 25)

            ch = (e & f) ^ (~e & g)

            temp1 = (h + s1 + ch + self.k[i] + w[i]) & 0xffffffff

            s0 = self.\_right\_rotate(a, 2) ^ self.\_right\_rotate(a, 13) ^ self.\_right\_rotate(a, 22)

            maj = (a & b) ^ (a & c) ^ (b & c)

            temp2 = (s0 + maj) & 0xffffffff

            h = g

            g = f

            f = e

            e = (d + temp1) & 0xffffffff

            d = c

            c = b

            b = a

            a = (temp1 + temp2) & 0xffffffff

        self.h = [

            (self.h[0] + a) & 0xffffffff,

            (self.h[1] + b) & 0xffffffff,

            (self.h[2] + c) & 0xffffffff,

            (self.h[3] + d) & 0xffffffff,

            (self.h[4] + e) & 0xffffffff,

            (self.h[5] + f) & 0xffffffff,

            (self.h[6] + g) & 0xffffffff,

            (self.h[7] + h) & 0xffffffff,

        ]

    def compute(self, message):

        """Compute the SHA-256 hash of the input message."""

        message = str(message)

        message = self.\_pad(message)

        for chunk in self.\_chunks(message, 64):

            self.\_process\_chunk(chunk)

        message\_digest = ''.join(f'{value:08x}' for value in self.h)

        return message\_digest

#AES encryption

def aes\_encrypt(data, key):

    # Pad data to make its size a multiple of 16 bytes (AES block size)

    padded\_data = pad(data, AES.block\_size)

    # Create AES cipher object

    cipher = AES.new(key, AES.MODE\_ECB)

    # Encrypt the data

    encrypted\_data = cipher.encrypt(padded\_data)

    # Convert encrypted data and key to base64 for JSON serialization

    encrypted\_data\_b64 = b64encode(encrypted\_data).decode('utf-8')

    return encrypted\_data\_b64

def aes\_decrypt(data, key):

    # Decode from base64

    encrypted\_data = b64decode(data)

    key = b64decode(key)

    # Create AES cipher object

    cipher = AES.new(key, AES.MODE\_ECB)

    # Decrypt the data

    decrypted\_data = cipher.decrypt(encrypted\_data)

    # Unpad the decrypted data

    unpadded\_data = unpad(decrypted\_data, AES.block\_size)

    return unpadded\_data