RSA

**Selecting p and q**

We need to ensure that the product of p and q should be larger than k we had to follow the following steps

For "CAKE":

* ASCII values:
  + 'C' = 67
  + 'A' = 65
  + 'K' = 75
  + 'E' = 69

the numeric representation will be 67×256^3+65×256^2+75×256+69×256 which is **1743657061** in decimal

so I chose 256 bit values:

* + p=70679880538383312502071352291856443545915428455819811944250026371670838470319
  + q=115103074580370081613695071598840503525162509721316244662542820653816484667741

to check this:

n=p×q=n>k

\*n should be the product of pxq

Compute phi n (ϕ(n))

Euler’s Totient function ϕ(n) is calculated as: ϕ(n)=(p−1)×(q−1)

derivation of e

e is the public exponent which should be a value between 2 and **ϕ(n)** -1 so that gcd (e, phi of **ϕ(n)**) =1 which in this case which is 17

having 17 will allow faster computation processing while having more security than 3

derivation of d

d is the the private key which is calculated by e inverse phi of n used for decryption

in case of k being larger than modulos n

if k is large than mod n then the encryption would be invalid a way to counter this would be to split the key into smaller bits than mod n and encrypting each segment using RSA or we can look into AES transmit the symmetric key

how alice would encrypt and bob would decrypt

**Bob (Receiver)**:

* Generates an RSA key pair (public n,e =8135471560941182295798785049605725658124249048110445169465175210728243291904485605961718855752914676000706084521873432921705300424198233021764289405279379, 17) and private d= 1914228602574395834305596482260170743088058599555398863403570637818410186330423487766258847613835037547485973547012318819653685733550927099938541666386193).
* Shares the public key (n,e) with Alice.

**Alice (Sender)**:

* Converts the message "CAKE" to its numeric representation.
* Encrypts the message using Bob’s public key (n,e)
* Sends the encrypted message (ciphertext c) to Bob.

**Bob (Receiver)**:

* Receives the encrypted message c.
* Uses his private key d to decrypt it.
* Converts the decrypted numeric representation back to the original message (which was "CAKE").

more efficient exponentiation

The Square-and-Multiply algorithm is used here for efficient modular exponentiation in RSA. It reduces the computational complexity by utilizing the binary representation of exponents, enabling quick calculations of powers mod n. In your script, this algorithm allows for efficient encryption and decryption processes, ensuring performance even with large integers

HASH FUNCTION

I have created a simple hash function that allows inputs of any length and converts any given message to a fixed sized digest of 160 bits and any produced hash is impossible to guess the input message modifications in the message, Changing the input would change the final hash meaning it is sensitive to all input bits

First we convert the input to its numerical ASCII value with each character in the message is represented by an 8-bit ASCII value. Function used in the hashing script

def string\_to\_ascii(message):

return [ord(char) for char in message]

padding the input message to ensure the input is processed in fixed block sizes. The message is padded to make its length a multiple of 16 bytes (128 bits). If the message length is already a multiple of 16, no padding is added. Padding is done with 0s

def pad\_message(ascii\_values):

length = len(ascii\_values)

padding\_length = (16 - (length % 16)) % 16

return ascii\_values + [0] \* padding\_length

divide the message into 16 byte blocks

* Splitting the padded messages into 16 byte and if the message is shorter than 16 byte block then we add one block
* This ensures that the hash function can handle messages of arbitrary length by breaking them into manageable chunks.

def split\_into\_blocks(padded\_message):

return [padded\_message[i:i + 16] for i in range(0, len(padded\_message), 16)]

initialize the hash value

* With a 128 bit hash value composed of 4 32 bit values with the following constants

H0 = 0x12345678

H1 = 0x9ABCDEF0

H2 = 0xFEDCBA98

H3 = 0x76543210

processing each block

* For each 16byte block we perform the following
  + Split the block into 4 32 bit words w1,w2,w3,w4
  + Update the hash values

W0 = (block[0] + (block[1] << 8) + (block[2] << 16) + (block[3] << 24))

W1 = (block[4] + (block[5] << 8) + (block[6] << 16) + (block[7] << 24))

W2 = (block[8] + (block[9] << 8) + (block[10] << 16) + (block[11] << 24))

W3 = (block[12] + (block[13] << 8) + (block[14] << 16) + (block[15] << 24))

H0 = (H0 + W\_0) XOR (W\_1 >> 3) = The value of H0​ is updated by adding W0​ to the current value of H0 Then, the result is XORed with W1​ after a right bit shift by 3 positions. The right bit shift (>>3) divides W1​ by 8 (since 3 bits shift means two 2^3=82^3 ).

H1 = (H1 + W\_1) XOR (W\_2 << 5) =H1​ is updated by adding W1​ to it. The result is then XORed with W2​, which has been left bit shifted by 5 positions. Left bit shift (<<5) is equivalent to multiplying W2​ by 2^5=32, shifting its bits to the left.

And so fourth

Updating the hash values

The hash values are updated using simple addition and bitwise XOR operations. Each hash value is modified by one of the words (W0, W1, W2, W3), and the words themselves are shifted before being XORed into the hash values.  
  
 H0 = (H0 + W\_0) XOR (W\_1 >> 3)

H1 = (H1 + W\_1) XOR (W\_2 << 5)

H2 = (H2 + W\_2) XOR (W\_3 >> 7)

H3 = (H3 + W\_3) XOR (W\_0 << 11)

hash\_values[0] = (hash\_values[0] + W0) ^ (W1 >> 3)

hash\_values[1] = (hash\_values[1] + W1) ^ (W2 << 5)

hash\_values[2] = (hash\_values[2] + W2) ^ (W3 >> 7)

hash\_values[3] = (hash\_values[3] + W3) ^ (W0 << 11)

**H0 = (H0 + W\_0) XOR (W\_1 >> 3) =** First, the value of W0​ is added to H0 updating H0 Then, W1​ is right-shifted by 3 bits. The right bit shift (>>3) effectively divides W1​ by 8, moving its bits to the right. Finally, the result of H0+W0 ​ is XORed with the shifted value of W1 XORing ensures non-linear mixing, meaning even small changes in W1​ or H0​ will result in a drastically different output.

hash output

After processing all blocks, the hash values are concatenated and converted to hexadecimal format to produce the final message digest. The result is a 128-bit hash (32 hex digits).

1. Output for the message - "Exams are on red USB drive in JO 18.103. Password is CaKe314."   
= 15F0ACAB181A391CD427DAD2266395A456043F

2. Output for the message – “Exams are on red USB drive in JO 18.103. saddw is CaKe314.”

= 142668E4DCE1E8741F23A4D4235304FB07BC02

3. Output for the message - "Exams are on red USB drive in JO 18.103. password is CaKe314."

= 15F0ACAB181A3D1CD427DACC266395A456043F

While the hashing is very simple it does cover the some security standards for a hashing algorithm to a shallow extend while the first and second hashes are very different from each other the 3rd hash digest has only character that changed to the small change in the capital letter of the message while the hash was successful in changing the hash digest it is not very secure and complex suggesting a weak resistance to second pre imaging but it also worthy to note that the hashing algorithm has low chance of spitting out the same hash for different input.

Digital signature

I have implemented a digital signature into the RSA algorithm with the same process and formulas used from task 4 and included the custom hashing algorithm and so where The digital signature is created by applying the square-and-multiply algorithm for modular exponentiation, which calculates:

signature=hash\_message^d mod n

Here, hash\_message is the numeric representation of the hashed message, and d is the private key.

The verification process involves checking the signature using the public key

decrypted\_hash=signature^e mod n

The decrypted hash is compared to the original hash of the message to confirm authenticity:

is\_valid=(decrypted\_hash==original\_hash)

**Signature Creation**:

* Once Alice has the hash of the message, she creates the signature by encrypting this hash using her private key (d\_a) and the modulus (n\_a) from her RSA key pair.
* The signature is generated using the rsa\_sign() function, which takes the message hash, Alice's private key, and modulus. It uses the **square-and-multiply** method to perform the modular exponentiation required for RSA signing.

signature = rsa\_sign(message\_hash, d\_a, n\_a)

**RSA Signing Process**:

* The rsa\_sign() function converts the message hash from its hexadecimal string representation into an integer.
* The signature is then computed using the following mathematical operation:

signature=hash^d mod n

* In this case, d is Alice's private exponent, and n is her RSA modulus

def rsa\_sign(message\_hash, d, n):

message\_hash\_numeric = int(message\_hash, 16)

signature = square\_and\_multiply(message\_hash\_numeric, d, n)

return signature

Decrypting by bob:

Bob decrypts the received ciphertext using rsa\_decrypt, which applies modular exponentiation using Bob's private key (d\_b) and modulus (n\_b).

decrypted\_message = rsa\_decrypt(ciphertext, d\_b, n\_b)  
  
def rsa\_decrypt(ciphertext, d, n):

decrypted\_numeric = square\_and\_multiply(ciphertext, d, n)

decrypted\_message\_hex = hex(decrypted\_numeric)[2:] # Convert to hex and strip '0x'

return bytes.fromhex(decrypted\_message\_hex).decode() # Convert hex back to string

The ciphertext is decrypted by performing modular exponentiation ciphertext^d\_b % n\_b using Bob’s private key.

The result is then converted from numeric form back to a hexadecimal string and finally back to the original message string.