7/21/2023

Cryptography

Name: Nuha Imran
ID:20696366

Glossary:

Question 1:

The Euclidean algorithm is based on the following assertion. Given two integers a, b, (a > b), gcd(a, b) = gcd(b, a mod b). (1) Prove the assertion (1) mathematically. (Note that proof by example is NOT appropriate here)

Euclidean Algorithm baredon assertion given $a, b, (a > b)$

where a is greator than b

$gcd(a, b) = gcd(b, a \bmod b)$

Base case: If $b = 0$, then $gcd(a, b) = a = gcd(a, 0)$

Let a & b be two integers such that $a > b > n$. By the Euclidean Algorithm. we can find integers q and r such that $a = qb + r$ and $0 \leq r < b$

since $r < b$, we know that r is a common divisor of a and b

Also, since r is the remainder when a is divided by b, we know that r is equal to a mod b

Therefore, $gcd(a, b) = gcd(b, r) = gcd(b, a \bmod b)$

conclusion: By the principle of mathematical induction, the assertion is true for all pairs of integers a and b such that $a > b$

To show, $gcd(a, b) = gcd(b, a \bmod b)$

let $a \bmod b = a - nb$ for some $n \in \mathbb{Z}$

let $gcd(a, b) = d$

i.e $d | a$ & $d | b$

$\iff d | a - nb |$

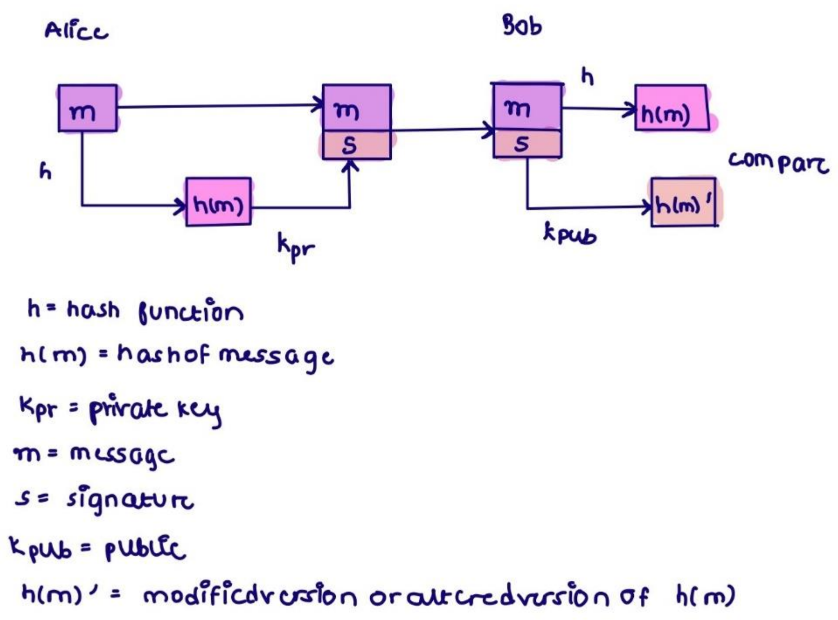
Thus a, b & $a - nb$ have same common divisors

Therefore some greatest common divisor, Hence $gcd(a, b) = gcd(b, a \bmod b)$

To begin proving this I'll suppose that a and b are two positive integers where a>b. I'll start by assuming a=b*q+r where 0<= r < b where a=dividend b=divisor q=quotient r=remainder, my second assumption will be that d=gcd(a,b) this basically denotes the fact that d is the largest possible integer that divides a and b without leaving any remainder. To prove this statement, I'll prove that d divides with both b and a mod b and any other common divisors of these numbers. Since a = b*q+r and d is the greatest common divisor of both a and b it denotes that d is divisible by both a and b. Henceforth d divides with a-b*q which is the same as a mod b since r= a-b*q. Further proving that d = gcd(a,b) divides both d and a mod b and any other common divisor of b and a mod b divides d henceforth d=gcd(a,b) is the greatest common divisor of b and a mod b proving the assertion(1) gcd(a,b) = gcd(b,a mod b) proven for all positive integers a and b where a>b

Question 2:

Assuming that Alice signed a document m using the RSA signature scheme. (You should describe the RSA signature structure first with a diagram and explain the authentication principle). The signature is sent to Bob. Accidentally Bob found one message m′ (m≠ m′) such that H(m) = H(m′), where H() is the hash function used in the signature scheme. Describe clearly how Bob can forge a signature of Alice with such m′ .



e = public key

d = private key

H(m) = hash value generated by using a hash function on m(message)

H(m’) = hash function value generated but not from the original message (message has been altered)

s = signature

s’ = not signature

<u>RSA Signature Scheme Overview:</u>

The RSA signature algorithm is a common method used to verify the authenticity and integrity of a message. It involves two main processes: key generation and signing.

Key Generation:

Alice generates a pair of keys: a private key (d, n) and a public key (e, n). 'n' is the product of two large prime numbers, and it is used as the modulus for both keys. 'e' is a small, fixed, and public exponent, while 'd' is a large, secret, and private exponent.

Signing:

To sign a message 'm,' Alice performs the following steps:

She first computes the hash value of the message using the hash function, H(m). The hash function condenses the message into a fixed-size string. Next, Alice applies a mathematical process called "padding" to the hash value to make it the same length as the RSA modulus 'n.' She then raises the padded hash value to the power of her private exponent 'd' modulo 'n,' resulting in the signature 's.'

Authentication process:

The authentication principle of the RSA signature relies on the fact that only Alice possesses the private key (d). The original message's hash is signed using this private key, yielding a distinctive signature that cannot be replicated without the private key. The procedures Bob takes to confirm the signature are as follows after he receives the signed message (m, s) from Alice:

He computes the hash value of the received message, H(m).

Bob raises the signature 's' to the power of the public exponent 'e' modulo 'n' to get 's'.

If the computed 's' matches the hash value H(m), Bob knows the signature is valid, and the message has not been tampered with.

Forging a Signature:

Now, This is an observation of how Bob can forge a signature for a different message, m′, given that H(m) = H(m′). Observe the Signature Structure:

Bob receives a valid signed message (m, s) from Alice, where s = H(m)^d (mod n).

Find a Different Message with the Same Hash:

Bob wants to forge a signature for a different message, m′, but with the same hash value, H(m) = H(m′).

Forgery Process:

Since s = H(m)^d (mod n), to forge a signature for m′, Bob needs to find a value s′ such that s′ = H(m′)^d (mod n).

However, Bob already knows that H(m) = H(m′).

This allows him to set s′ = H(m)^d (mod n) since H(m) = H(m′).

So, s′ = s, and Bob now has a valid signature (m′, s).

Verification:

When someone, say Annie, receives the signed message (m′, s) from Bob and wants to verify the signature:

Annie computes the hash value of m′, H(m′).

He raises the signature 's' to the power of the public exponent 'e' modulo 'n' to get s^e (mod n).

Since s = H(m)^d (mod n) and H(m) = H(m′), s^e (mod n) will be equal to H(m)^d (mod n), which is the value Bob calculated for s′.

Thus, Annie will believe the signature to be valid, even though it was forged.

Summary:

Bob can forge a signature for a different message with the same hash value because the RSA signature scheme only relies on the hash value of the message, not the message itself. By finding a different message with the same hash, he can generate a valid signature for that message and deceive others into thinking it's authentic. This is why hash functions play a crucial role in the RSA signature scheme to ensure the security and authenticity of the signatures.

Question 3a:

What are the lessons you learned, and difficulties you met, in the process of implementing RSA?

Lessons Learned:

I had to start by learning the RSA algorithm itself which delved me into various mathematical concepts starting with modular arithmetic then Euler’s totient, multiplicative inverse, Euler's theorem, Fermat’s theorem, and so on once all these concepts were learned then only, I proceeded with the coding. I further learned about how public key and private keys are generated and how they function. I also learned how breaking the code into smaller bits will help it come all together in the end. Python random library was also a new concept to me provided I have never used it before. I learned how the algorithm works on its own and what the logic behind finding p and q prime numbers is rather than just taking any number as n . Then I learned how the encryption and decryption functions inside RSA work and how we used the private key to decrypt and the public key to encrypt below is a diagram attached of the internal work that I had to learn to implement RSA.

RSA Algorithm : (Rivest, Shahmir, Adlemann)

publickey = reciever, sender both known
privatekey =

only sender knows
only reciever knows
- (e, n) public key
- (d, n) private key

1) choose two prime p & q $\phi(n)$ = phi

$p = 61 \quad q = 53$

2) compute : $n = p \times q = 61 \times 53 = 3233$

$\boxed{n = 3233}$

3) $\phi(n) = \phi(p \times q) = \theta(p) \times \phi(q)$ Apply Eulers Toient function

$= (p - 1) \times (q - 1) = 60 \times 52$

$\phi n = 3120$ (we found n)

4) choose 'e' ; $1 \leq e < \phi(n)$, coprime to $\phi(n)$

$e = 17$ $gcd(17, 3120) = 1$

$\therefore (e, n)$ = publickey $\underbrace{(17, 3233)}_{\text{coprimes}}$ when you apply g,c,d it should be 1

5) Determine 'd' as $ed = 1 \bmod \phi(n)$

$d = e^{-1} \bmod \phi(n)$ (d is MI of e)

$\Rightarrow 17 \times d = 1 \bmod 3120$

$\boxed{d = 2753}$ $\therefore (d, n)$ = privatekey (2753, 3233)

$\underbrace{\text{finding 'd'}}$ coprime(gcd = 1)

$ed = 1 \bmod \phi(n)$

$$\boxed{d = \frac{(\phi(n) \times i) + 1}{e}}$$

$d = \frac{(3120 \times 1) + 1}{17} = 183.58$

$d = \frac{(3120 \times 2) + 1}{17} = 367.11$

$d = \frac{(3120 \times 3) + 1}{17} = 550.647$

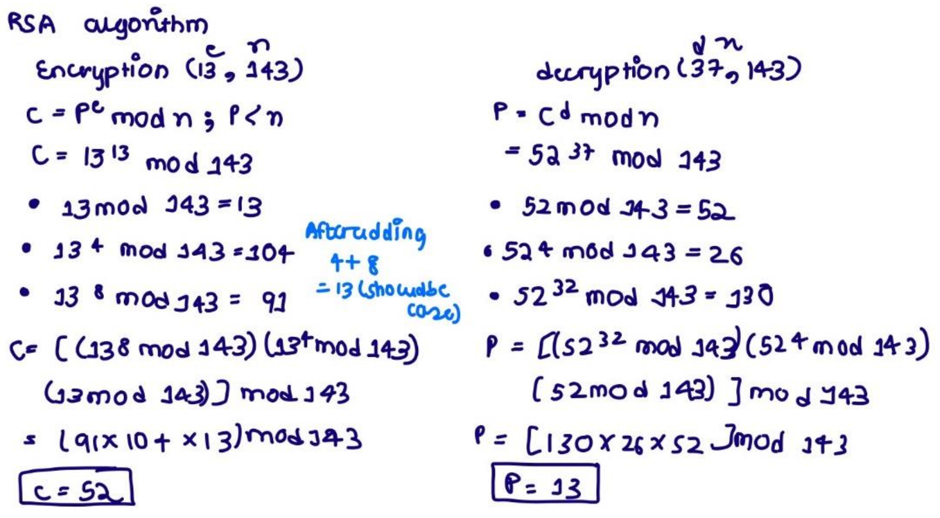
EXPLAINATION

$d = \frac{(3120 \times 4) + 1}{17} = 734.17$

$d = \frac{(3120 \times 12) + 1}{17} = 2202.411$

$d = \frac{(3120 \times 15) + 1}{17} = 2753$

Keep doing till you don't get an integer



RSA algorithm

Encryption $(\overset{e}{13}, \overset{n}{143})$

$C = P^e \bmod n \,;\, P < n$

$C = 13^{13} \bmod 143$

- $13 \bmod 143 = 13$
- $13^4 \bmod 143 = 104$
- $13^8 \bmod 143 = 91$

After adding 4+8 = 13 (shouldbe case)

$C = [(13^8 \bmod 143)(13^4 \bmod 143)$
$(13 \bmod 143)] \bmod 143$

$= (91 \times 104 \times 13) \bmod 143$

$\boxed{C = 52}$

decryption $(\overset{d}{37}, \overset{n}{143})$

$P = C^d \bmod n$

$= 52^{37} \bmod 143$

- $52 \bmod 143 = 52$
- $52^4 \bmod 143 = 26$
- $52^{32} \bmod 143 = 130$

$P = [(52^{32} \bmod 143)(52^4 \bmod 143)$
$(52 \bmod 143)] \bmod 143$

$P = [130 \times 26 \times 52] \bmod 143$

$\boxed{P = 13}$

Difficulties Encountered:

I did not encounter any major difficulties while coding however some of the math functions did cause problems while I was coding for example, the modular inverse function that I used pow() in Python for generated overflow errors when I initially started coding as the keys generated were getting too large for it. However, after tweaking my code a bit and changing a few libraries the function did start working. Initially, I used sympy to generate large prime numbers but after running my code on my Ubuntu I realized that you need extra installations to run that library which was not in the scope of the requirements of the assignment so I introduced random library instead to produce a number greater than 2^64 and checked it using a prime test to ensure that the numbers being generated are prime. Then I had to satisfy the m < n condition by encrypting bit by bit this was a bit of a task for me. The last error I ran into was source encoding and decoding my file repeatedly kept printing weird characters soon I realized that I was converting to and from binary which was not at all required. The code was treating my binary values as decimal numbers so I made changes accordingly in my encrypt and decrypt to remove the unnecessary steps after which the code started functioning properly.

Question 3b:

Describe what you have done for source coding and decoding.

Functions and how they fulfill the requirement(all this is explanation of code source encoding and decoding in the end):

•Prime Test: To determine whether a given integer is prime, I implemented the is_prime(num) function. If the number is divisible by anything other than 1 and itself, it is determined using a straightforward trial division method. If the number has any factors other than 1 and itself, or if it is less than or equal to 1, the code returns False. If not, True is returned, denoting that the integer is a prime.

•Random Large Prime Generation: Using the is_probable_prime(n) function, the generate_random_large_prime() method creates random numbers between 26^4 and 26^5 and checks each one for primality. The Miller-Rabin primality test, an effective probabilistic primality test, is used by the is_probable_prime(n) function. To boost confidence in the primality of the generated number, the function runs the Miller-Rabin test k times. The generated integer is regarded as a random huge prime if it passes the primality test.

•Extended Euclidean Algorithm: The gcd_extended(a, b) function applies the extended Euclidean algorithm to determine the greatest common divisor (GCD) of two numbers, a and b, and it also calculates the coefficients x and y such that ax + by = GCD(a, b). Finding the modular inverse, which is required to compute the private key in RSA, is dependent on this function.

•Euler's Totient Function: For two supplied prime numbers, p and q, the euler_totient(p, q) function computes Euler's totient function, also referred to as the phi function. To calculate the public key in RSA, it is necessary to know the number of positive integers less than n that are relatively prime to n. This information is obtained using Euler's totient function.

•Modular Exponentiation: The built-in pow() function in Python is used to calculate modular exponentiation using the mod_exp(base, exponent, modulus) function. It effectively calculates base-exponent mod modulus, which is necessary for RSA's encryption and decryption processes.

•Key Generation: The public and private keys for RSA encryption and decryption are generated by the generate_keys() method. It generates the huge prime integers p and q at random, determines the modulus n by multiplying p by q, and determines the Euler totient function phi = (p - 1) * (q - 1). When gcd_extended(e, phi)[0] == 1, it then chooses a random value e between 2 and phi - 1 such that. Finally, it applies the extended Euclidean algorithm to determine the modular inverse d of e modulo phi.

•Modular exponentiation is used to encrypt the integer representation of each character throughout the encryption process in the encrypt_character() method. Calculating ciphertext = mod_exp(ord(character), e, n) is a stage in the encryption process. In this case, ord(character) returns the character's integer representation. The modulus procedure guarantees that the resultant ciphertext will always be smaller than the modulus n because e and n are both positive integers.

All standard keyboard characters immediately satisfy the criterion m < n during encryption since the modulus n is greater than any character's integer representation.

RSA Encryption:

To generate RSA keys, I started by picking up two large prime numbers that I generated using the random library of python, let's call them 'p' and 'q'. I then calculated their product, which gave me the modulus 'n'. Next, I computed Euler's totient function by subtracting 1 from both 'p' and 'q', and then multiplying them together to get 'phi'.

I then proceeded to select a random number 'e' between 2 and 'phi - 1', making sure that 'e' and 'phi' are coprime, meaning they have no common factors other than 1.

Once I got my public key (e, n), I started encrypting the plaintext. For each character in the text, I converted it into a number using its ASCII value. Then, I raised this number to the power of 'e' and take the remainder when divided by 'n'. The resulting number is the encrypted value of that character.

To make the encrypted ciphertext more readable, I converted the list of integers into a hexadecimal string.

RSA Decryption:

For decryption, I need the private key (d, n). I read the private key from the file, where 'd' is the secret exponent and 'n' is still the same modulus used during encryption.

I started decrypting the ciphertext by converting the hexadecimal representation back to its decimal form. Then, I raised this number to the power of 'd' and took the remainder when divided by 'n'. The resulting number is the decrypted value of that character.

Finally, I concatenated all the decrypted characters to get the original plaintext.

The main function handles the whole process by reading the input file, performing the requested operation (either encryption or decryption), and writing the result to the output file.

This source coding and decoding strategy ensures that our RSA encryption and decryption program can securely handle standard keyboard characters, fulfilling the specified criteria.

<u>Source Encoding:</u>

The input text (plaintext) is supplied character by character to the encrypt_character() function in my encrypt_text() method, which then uses RSA encryption to encrypt each individual character. Using modular exponentiation, my encrypt_character() function transforms a character's ASCII value into its equivalent ciphertext. The outcome is an integer that represents the character's encrypted value.

My code uses the hex() function to turn each of these integers into a hexadecimal string so that they can be stored in a human-readable format. The list of integers (ciphertexts) is passed to the convert_to_hex() function, which converts each integer to its hexadecimal equivalent. After joining the resulting hexadecimal values with spaces, a single string of those values is returned. The encrypted text is represented by this string of hexadecimal numbers.

Source Decoding:

The input ciphertext (in hexadecimal format) is separated into individual hexadecimal values using hex_ciphertext.split(" "), which is then used in my decrypt_text() method. The decrypt_character() method is then called while the list of hexadecimal values is passed character by character.

Using the int(ciphertext, 16) method, my decrypt_character() function transforms each hexadecimal value back into its decimal integer form. The original plaintext character is then obtained by decrypting the decimal ciphertext using modular exponentiation and the private key (d, n).

The original plaintext is then created by joining the decrypted integers for each character. This plaintext is the result of the decrypt_text() method.