Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Ching	Student ID:	22013213
Other name(s):	Mao Jin		
Unit name:	Fundamental Concept of Cryptography	Unit ID:	ISEC2000
Lecturer / unit coordinator:	Dr Sivaraman Eswaran	Tutor:	Dr Sivaraman Eswara
Date of submission:	5/1/2025	Which assignment?	2 (Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: CMJ Date of signature: 5/1//2025

(By submitting this form, you indicate that you agree with all the above text.)

Contents

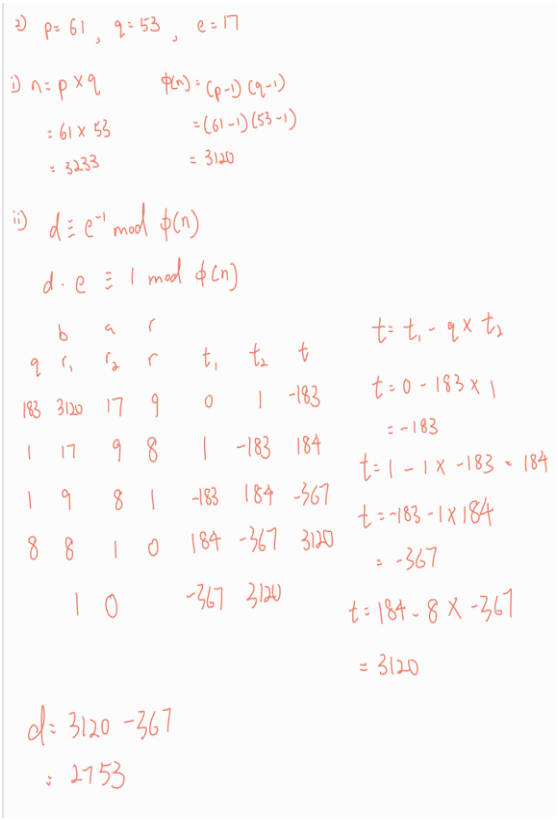
Disclaimer: This table of content follows the page number at the bottom of each page.

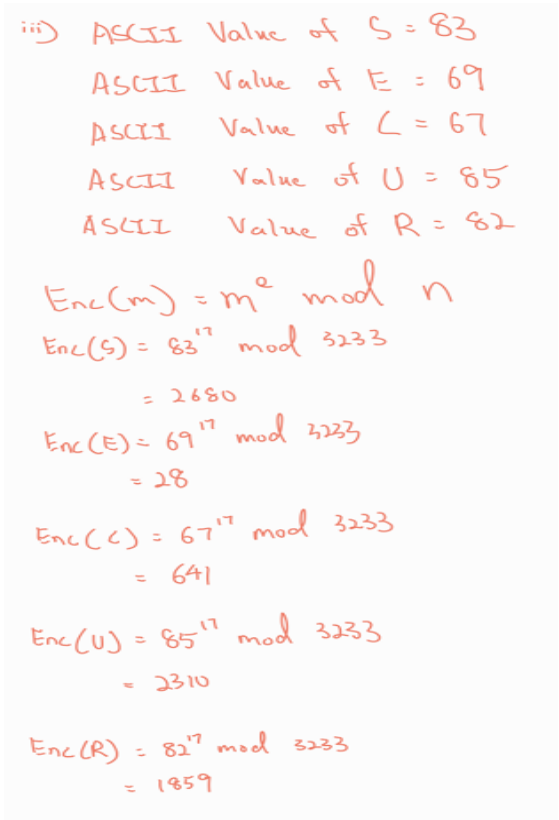
Conceptual Understanding and Question Answering

1) (5 points) Explain the core principle of public key cryptography and how it addresses the issue of secure communication over untrusted channels.

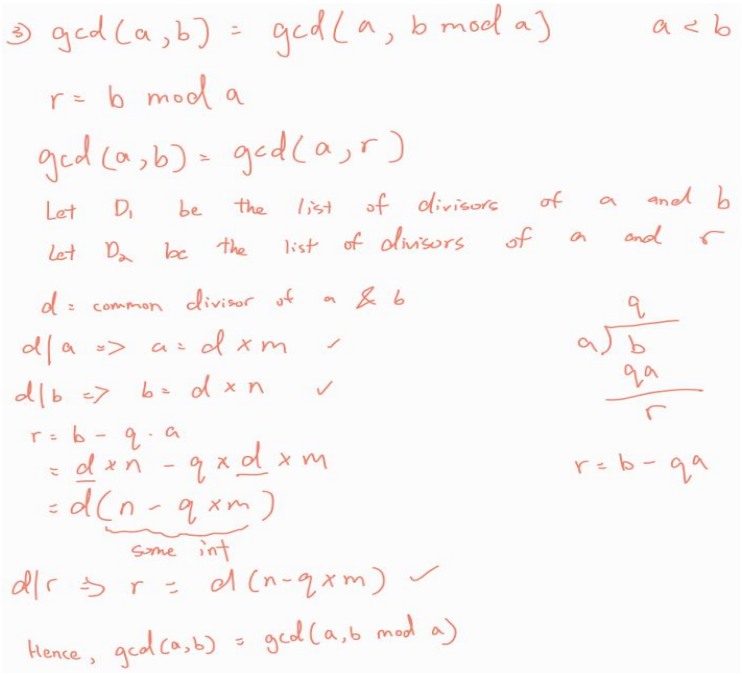
Ans: Public key cryptography is also known as asymmetric cryptography that uses a pair of keys, 1 public and 1 private key. Public key is shared openly while private key is kept secret. When a sender wants to send a message, the sender encrypts the message with the receiver's public key. When the receiver receives the encrypted message, the receiver can decrypt with the receiver's private key. This addresses the issues of sending messages over untrusted channels by ensuring that only the intended recipent can decrypt and read the message which maintains the confidentiality of the message.

2) (15 points) Given two prime numbers p = 61 and q = 53, and public key exponent e = 17. i) Calculate the modulus n, Euler's totient ϕ(n). ii)Compute the private key d. iii)Convert "SECURE" to ASCII and encrypt using RSA. Show the working steps.

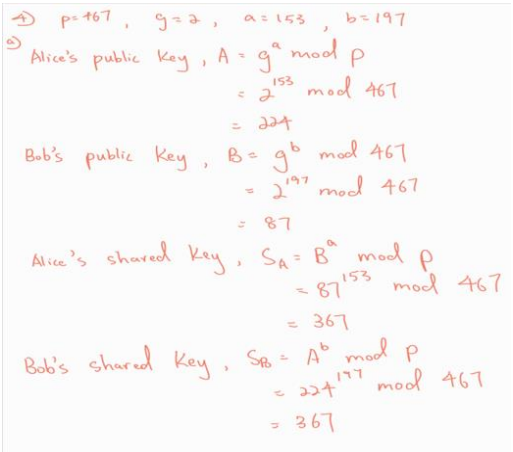


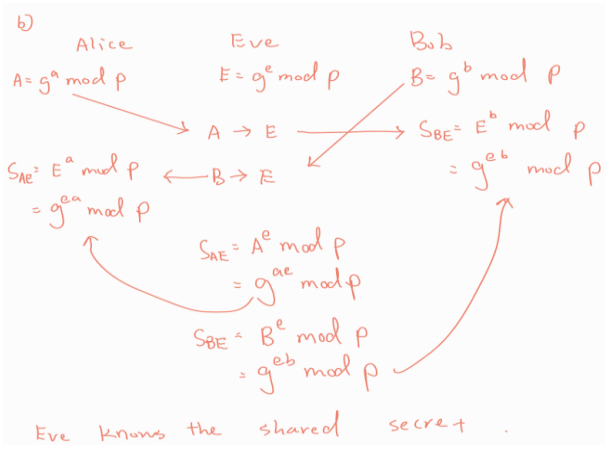


3) Prove gcd(a,b) = gcb(a,b mod a) where a<b



4) (10 points) You are given the following parameters for a simplified Diffie-Hellman key exchange: Prime modulus p = 467, Generator g = 2, Alice's private key a =153, Bob's private key b = 197. A) Please compute both the public keys and the shared key. B) Describe a possible Man-in-the-Middle (MitM) attack sce nario during the key exchange between Alice and Bob. In clude the attacker's steps and how the attack compromises security.





b) Alice computes her public key, A=g^a mod p and Bob computes his public key, B=g^b mod p. When Alice sends her public key,A over to Bob, the eavesdropper intercept Alice's public key to E which is the Eve's public key. When Bob sender his public key, B over to Alice, the eavesdropper intercept Bob's public key to E. Now, the eve knows the shared secret between Alice and Bob which can be used to decrypt messages sending between Alice and Bob which violates the confidentiality of the message.

5) (15 points) Answer the following questions: a. Explain how Elliptic Curve Cryptography (ECC) achieves security with smaller key sizes than RSA. b. Compare RSA-2048 vs ECC-256 in terms of speed and memory. c. If you are designing encryption for IoT devices, would you choose ECC or RSA? Justify with technical reasoning.

Ans: a) RSA's security is based on the trapdoor function of factoring n into two large prime factors. Whereas, ECC relies on the difficulty to compute d when Q=d . G where Q and G are known to the attacker. The only way to obtain the private key, d is through brute-force attack. The private key, d takes any integer from the range of 1 to n-1 where n is the total number of unique points obtained from the curve when adding G to itself repeatedly until it gets back to point G. In real-world curve like secp256r1, n will be a very large integer like 2^256 bits. Private key, d can be any integers close to 2^256 bits will be about 10^77 in decimal. This takes the age of the universe long to find the private key which makes brute force attack infeasible.

b)

Categories	RSA	ECC
Bandwidth	Higher	Lower
Encryption Speed	Faster with small e	Slower
Decryption Speed	Slower	Faster
Memory Usage	High	Low

Bandwidth: RSA uses larger bit key so the encrypted message will also be larger. Hence, RSA has a higher bandwidth

Encryption: RSA uses the formula m^e mod n. This involves arithmetic calculation which is less complex as compared to ECC which uses scalar multiplication on curve. Hence, RSA has a higher encryption speed.

Decryption: RSA uses c^d mod n to decrypt message where d is the modular inverse of e mod phi of n. Since n is a long-bit number, d will have a long bit. This will make RSA to cosume more time for decryption. On the other hand, ECC uses a smaller key size which makes it faster for decryption.

Memory usage: RSA uses more memory. This is because RSA requires more CPU time to compute the key due to larger key size. The key size used in RSA is larger which cosumes more RAM.

c) I would choose ECC because IoT devices tend to have Tiny CPU, less RAM. ECC enables faster computation and able to achieve the same level of security with smaller key size. RSA uses more memory which may be too heavy to loT devices. In overall, ECC serves as a better option as compared to RSA.

Documentation of python's cryptography library

Generate_keys(): I used the python's library function, rsa.generate_private_key(public_exponent=65537, key_size=2048). This function takes two parameters, the public exponent, e and the key_size which is the bit size of n.

What happens internally in the function:

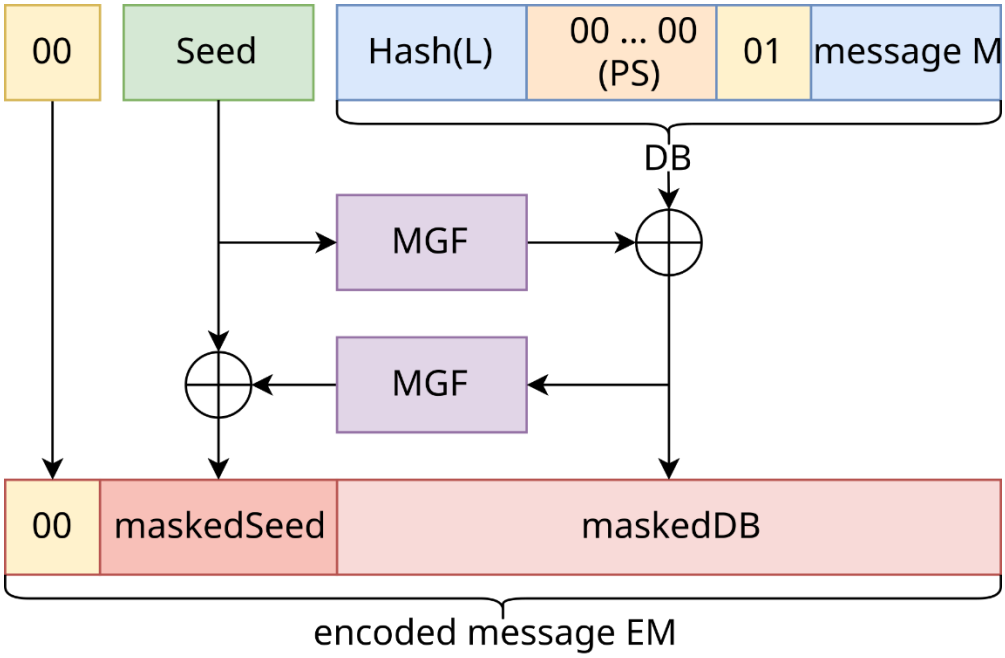
1) Two large prime number p and q will be generated
2) n=p*q and Euler's Totient Func, (p-1)(q-1) will be computed
3) using the given public exponent, e, d will be computed by extended euclidean algorithm
4) all other value that are part of private keys such as p,q,n and d will be stored in private key object

I used private_key.public_key to export e and n computed suring the key generation. After key generation, I need to store public and private key in public.pem and private.pem files respectively.

Storing Keys: I converted the key to byte using private_bytes for private key and public_bytes for public key. This function takes the parameters of encoding, format and encryption algorithm for private key. Encoding requires me to specifies how bytes are converted. "serialization.Encoding.PEM" means follow the encoding form in PEM file which is base64. This will encoded the raw byte to base64(ASCII printable text). The second parameter, format specifies the key structure being used. TraditionalOpenSSL is the PKCS1 format for private key. There's one extra parameter in private key to specify encryption algorithm which is the password used to load the key.

Encryption Document: I applied OAEP padding scheme before encryption. This is to enhance the randomness in the data block which mitigates the risk of attacker manipulating the ciphertext.

How OAEP works internally:



1) A random seed is generated
2) Data block consists of the hashed label, padding 0s, seperator byte 0x01 and the message
3) MGF1 is applied to seed then XOR with the data block to form a maskedDB
4) The seed is then XORed with the mask generated from MGF1 to form masked seed
5) The final encoded message is formed by concatenating 0x00(positive sign), masked seed, masked DB
6) The final encoded message is then involved in encryption
7) This enhances randomness in the data block

```
padding_scheme=padding.OAEP(
    mgf=padding.MGF1(algorithm=hashes.SHA256()), # use SHA256 to generate mask
    algorithm=hashes.SHA256(),  #hash the label using SHA256
    label=None
)
```

This image shows the code snipet from my program

The hash algorithm, SHA256 will be used to generate the mask and the label will be hashed using the same hash algorithm.

```
#load the public key from public.pem file
with open(public_key_file, "rb") as file:
    public_key=serialization.load_pem_public_key(file.read())

#encryp the plaintext
CipherText=public_key.encrypt(PlainText, padding_scheme)
```

My program loaded the public key from public.pem file and uses it for encryption. In RSA Encryption, public key is used for encryption(Enc(M)=M^e mod n) so that only the corresponding private key can decrypt.

Decrypting document: Decryption is the reverse of encryption. My program needs to decrypt first then only unpad which will be done internally by the library function.

```
#unpadding after decryption with this padding scheme
padding_applied=padding.OAEP(
    mgf=padding.MGF1(algorithm=hashes.SHA256()),
    algorithm=hashes.SHA256(),
    label=None
)

#load the private key from private.pem file
with open(private_key_file, "rb") as file:
    private_key=serialization.load_pem_private_key(file.read(), password=None)

PlainText=private_key.decrypt(CipherText, padding_applied)
```

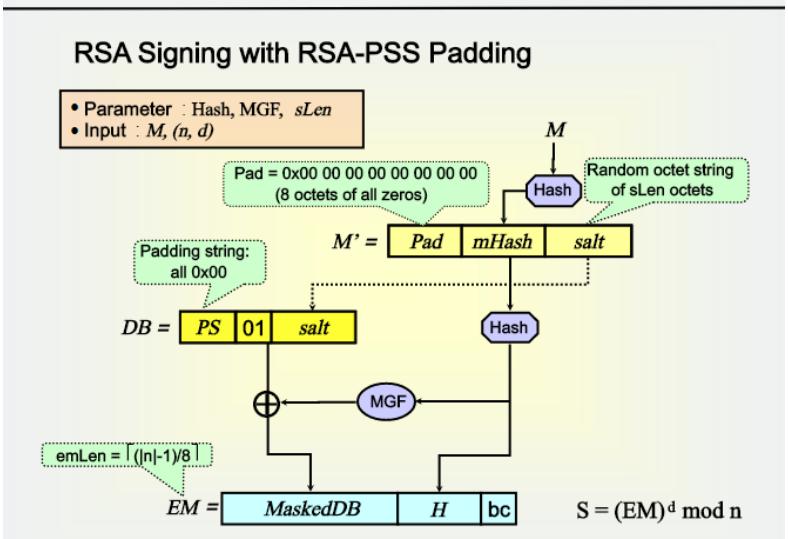
The same padding scheme used in encryption must be the same in decryption. This is to let the decrypt function know how to unpad after decryption.

Signing document: Signature is used for authentication. When the sender sign the message with his signature, he can't deny that he did not send the message if the verification is successful. This enhances non-repudation, autentication and integrity. When the message is modified, the verification will not be successful.

```
#apply PSS padding before encrypting the hashed message
#purpose: prevent attacker forging signature
#How pss works:
#hash the message
#generate a random salt based on the given length
#build M' which consists of Padding, hash(M), and salt
#hash(M')
#create EM which consists of Padding, seperator 0x01, salt and hash(M')
#Mask padding + seperator + salt by repeatedly hashing the block with incrementing counter
#XOR the mask with DB(padding + seperator + salt)
#Final EM=MaskedDB + Hash(M') + bc(0xBC)
#bc: represents the end of the block
padding_scheme=padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),  #use the hash func- sha256 for masking
    salt_length=padding.PSS.MAX_LENGTH  #generate the salt based on the max length
)
```

How I generate signature:

1) I applied PSS padding before encrypting the hashed message to prevent attacker from forging the signature
2) The padding scheme ensures that signature generated each time will be different even the same private key is used.
3) It takes two parameter, one for generating mask and the other for generating salt based on the given length.
4) My comments above explain in details how PSS works.



Flowchart of PSS

Signature function: takes 3 parameter(plaintext, padding, and hashing algorithm)

1) The sign function uses sha256 to hash the message then pads the hashed message
2) The padding determines the structure of final encoded message.
3) The hashing algorithm specifies the algorithm used for hashing the message.
4) Private key is used for generating the signature so that the public key can decrypt the signature for verification

Verifying Signature: Once a signature is generated, it needs to be verified to ensure that the message is not modified and the messsage is coming from the right sender. Public key is used for signature verifation.

How verify works:

1) Decrypt the signature using sender's public key
2) After decrypting, I will get the final encoded message with the padding applied
3) Unpad the encoded message to get the hashed message
4) Hash the plaintext, H(M)
5) Compare the hashed message from the signature and H(M)
6) If matches, verification is successful. Otherwise, my program will throw an exception.

Challenges faced: This assignment requires deep understanding and clear documentation on how the library functions are used, and what's happening internally. Initially, I have difficulities understanding the padding scheme, OAEP and PSS. I did not see the neccessity of using padding before encryption. After hours of research, I understand the purpose of padding which is to enhance the randomness of the data and ensure the ciphertext generated each time will be different. This mitigates the risk of attacker manipulating ciphertext. Morever, understanding the proving logic in question 3 gave me a hard time. I have to constantly think about the steps, how they relates and why it is done this way to eventually prove gcd(a,b) = gcd(a, b mod a). The findings are documented above. In conclusion, this assignment took me a very long time to understand the logic and concepts.

References

ECC. (n.d.). *Elliptic Curve Cryptography (ECC) | Practical Cryptography for Developers*. Practical Cryptography for Developers. Retrieved April 28, 2025, from https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc

ASCII Table. (n.d.). *ASCII table - Table of ASCII codes, characters and symbols*. Retrieved April 28, 2025, from https://www.ascii-code.com/

Python Software Foundation. (n.d.). *RSA — Cryptography 45.0.0.dev1 documentation*. Retrieved April 28, 2025, from https://cryptography.io/en/latest/hazmat/primitives/asymmetric/rsa/

Wikipedia contributors. (n.d.). *Optimal asymmetric encryption padding*. Wikipedia. Retrieved April 28, 2025, from https://en.wikipedia.org/wiki/Optimal_asymmetric_encryption_padding

ZhiHu. (n.d.). RSA signature PSS mode. ZhiHu. Retrieved April 28, 2025, from https://zhuanlan.zhihu.com/p/56678361