CM2025 Computer Security: Midterm Coursework

SOH YI JIE GABRIEL – (220516695)

UNIVERSITY
OF LONDON

PART B: Cryptography

Question 1

1a)

The program uses a straightforward symmetric encryption algorithm that incorporates modular addition to encrypt a message. The plaintext's letters are all changed to their corresponding numbers (A=0, B=1,..., Z=25). We guarantee that the length of the key will match that of the plaintext. To encrypt a message, the value of each plaintext letter is added to the value of the associated key letter. The result is then taken modulo 26 to wrap around the alphabet. The encrypted text is then created by converting this resultant number back to a letter. For instance, "GABRIEL" becomes "ZHJQWL" when encrypted with the key "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM". By using this technique, the encrypted message will appear random and be challenging to decode without the key. The encryption resists frequency analysis assaults by shifting each letter in the plaintext by a value specified by the matching key letter. By ensuring that the values correctly wrap around within the alphabet, modular addition preserves the integrity of the letter conversion. Each letter is encrypted uniquely when the length of the key matches that of the plaintext, which improves security. This method offers a simple yet efficient way to encrypt text messages.

JavaScript Code:

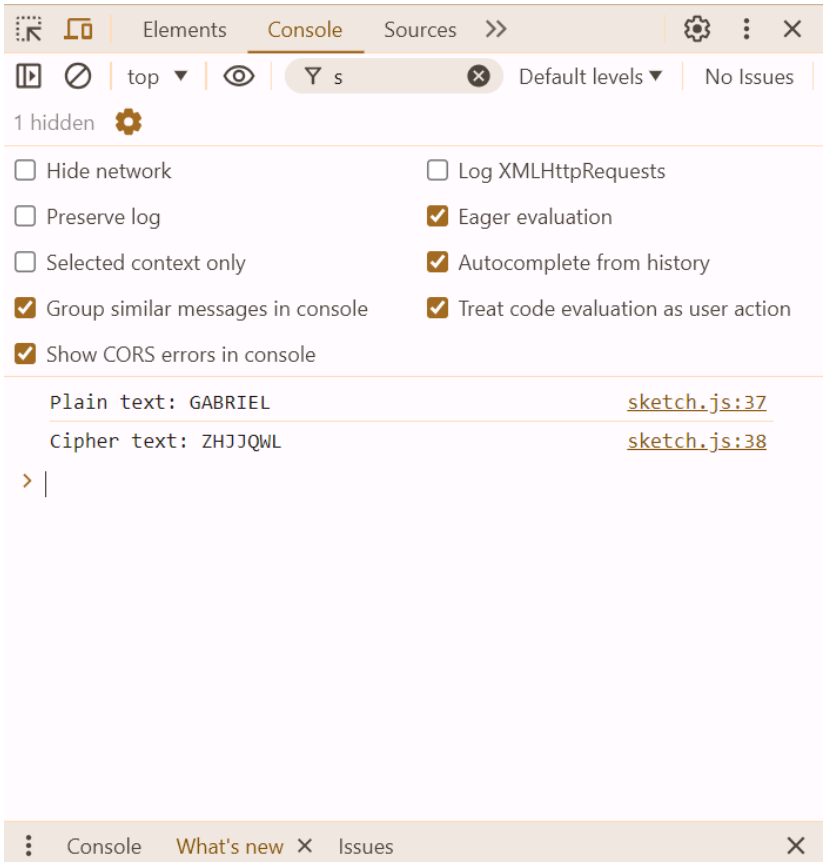
```
function encrypt(plaintext, key) {
  // This function converts letters to numerical values/numbers (A -> 0, B -> 1, ..., Z -> 25)
  function letterToNumber(letter) {
    return letter.charCodeAt(0) - "A".charCodeAt(0);
  }

  // This function Converts numerical values back to letters (0 -> A, 1 -> B, ..., 25 -> Z)
  function numberToLetter(number) {
    return String.fromCharCode((number % 26) + "A".charCodeAt(0));
  }

  // To convert plaintext and key to uppercase (Assumption is they are in uppercase)
  plaintext = plaintext.toUpperCase();
```

```
  key = key.toUpperCase();

  // To ensure the key is at least as long as the plaintext
  if (key.length < plaintext.length) {
    throw new Error("Key must be at least as long as the plaintext.");
  }

  // encrypt the plaintext using the key
  let cipher = "";
  for (let i = 0; i < plaintext.length; i++) {
    const plainLetter = letterToNumber(plaintext[i]);
    const keyLetter = letterToNumber(key[i]);
    const cipherLetter = (plainLetter + keyLetter) % 26;
    cipher += numberToLetter(cipherLetter);
  }

  return cipher;
}

// USAGES
const plaintext = "GABRIEL";
const key = "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM";
const cipher = encrypt(plaintext, key);
console.log("Plain text:", plaintext); // Outputs: GABRIEL
console.log("Cipher text:", cipher); // Outputs: ZHJJQWL
```

Input:

```
// USAGES
const plaintext = "GABRIEL";
const key = "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM";
const cipher = encrypt(plaintext, key);
console.log("Plain text:", plaintext); // Outputs: GABRIEL
console.log("Cipher text:", cipher); // Outputs: ZHJJQWL
```

Output:


Elements
Console
Sources
top
s
Default levels
No Issues
1 hidden
Hide network
Log XMLHttpRequests
Preserve log
Eager evaluation
Selected context only
Autocomplete from history
Group similar messages in console
Treat code evaluation as user action
Show CORS errors in console
Plain text: GABRIEL
sketch.js:37
Cipher text: ZHJJQWL
sketch.js:38
Console
What's new
Issues

1b)

The decrypt function in the program reverses the encryption process to retrieve the original plaintext from the ciphertext. It starts by converting letters to numerical values and vice versa. The ciphertext and key are converted to uppercase to maintain consistency. The function ensures the key's length is at least as long as the ciphertext; otherwise, it raises an error. To decrypt, it iterates through each character of the ciphertext and key, converting them to numerical values. It then performs modular subtraction between the numerical values of the ciphertext and key characters, ensuring a non-negative result by adjusting the value before applying the modulo operation. The resulting number is converted back to a letter and appended to the plaintext string. Finally, the function returns the decrypted plaintext. When tested with the provided ciphertext and key, it successfully returns the original plaintext, demonstrating the decryption process effectively.

JavaScript Code:

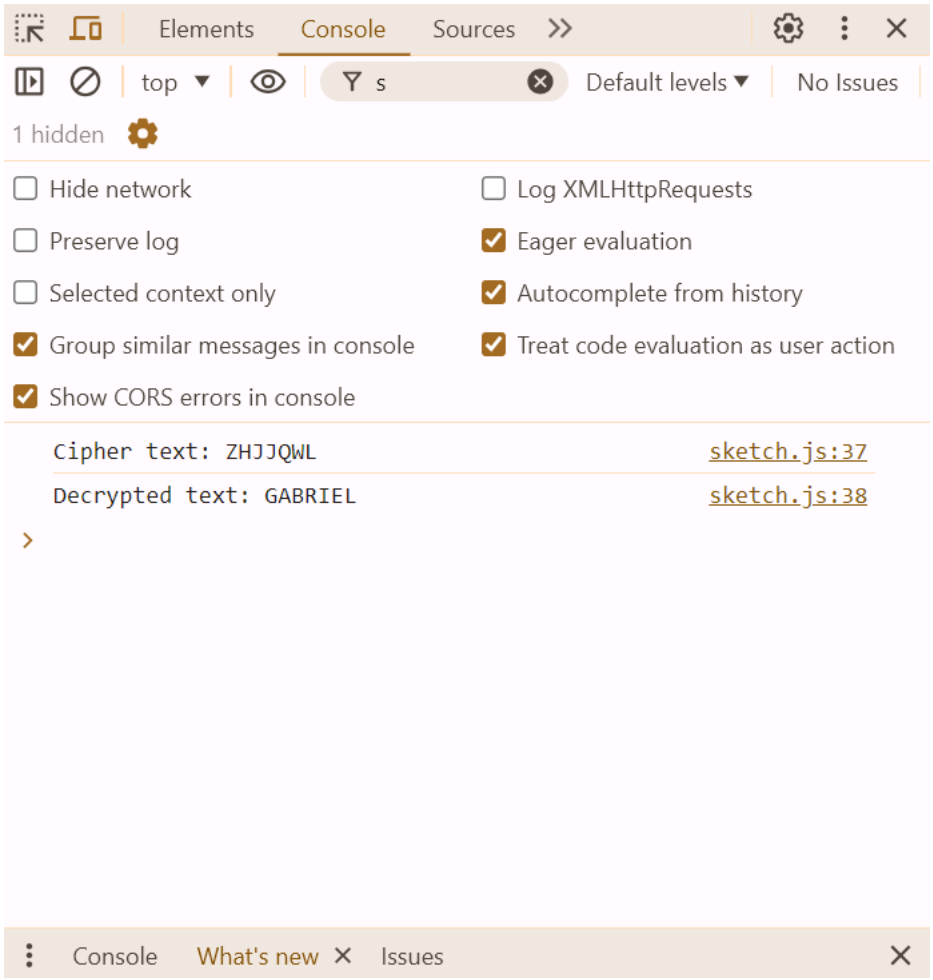
```
function decrypt(ciphertext, key) {
  // This function converts letters to numerical values/numbers (A -> 0, B -> 1, ..., Z -> 25)
  function letterToNumber(letter) {
    return letter.charCodeAt(0) - "A".charCodeAt(0);
  }

  // This function Converts numerical values back to letters (0 -> A, 1 -> B, ..., 25 -> Z)
  function numberToLetter(number) {
    return String.fromCharCode((number % 26) + "A".charCodeAt(0));
  }

  // To convert ciphertext and key to uppercase (Assumption is they are in uppercase)
  ciphertext = ciphertext.toUpperCase();
  key = key.toUpperCase();

  // To ensure the key is at least as long as the ciphertext
  if (key.length < ciphertext.length) {
    throw new Error("Key must be at least as long as the ciphertext.");
```

```
  }

  // Decrypt the ciphertext using the key
  let plaintext = "";
  for (let i = 0; i < ciphertext.length; i++) {
    const cipherLetter = letterToNumber(ciphertext[i]);
    const keyLetter = letterToNumber(key[i]);
    let plainLetter = (cipherLetter - keyLetter + 26) % 26; // ADD 26 TO
ENSURE NON-NEGATIVE RESULT
    plaintext += numberToLetter(plainLetter);
  }

  return plaintext;
}

// USAGES
const ciphertext = "ZHJJQWL";
const key = "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM";
const decryptedText = decrypt(ciphertext, key);
console.log("Cipher text:", ciphertext); // Outputs: ZHJJQWL
console.log("Decrypted text:", decryptedText); // Outputs: GABRIEL
```

Input:

```
// USAGES
const ciphertext = "ZHJJQWL";
const key = "THISISANEXAMPLEKEYINCOMPUTERSECURITYEXAM";
const decryptedText = decrypt(ciphertext, key);
console.log("Cipher text:", ciphertext); // Outputs: ZHJJQWL
console.log("Decrypted text:", decryptedText); // Outputs: GABRIEL
```

Output:


Elements
Console
Sources
top
s
Default levels
No Issues
1 hidden
Hide network
Log XMLHttpRequests
Preserve log
Eager evaluation
Selected context only
Autocomplete from history
Group similar messages in console
Treat code evaluation as user action
Show CORS errors in console
Cipher text: ZHJJQWL
sketch.js:37
Decrypted text: GABRIEL
sketch.js:38
Console
What's new
Issues

1 c)

A flaw in the algorithm is one requirement for cracking a cipher (cryptanalysis). A cipher may be vulnerable to cryptanalysis, the study of deciphering and studying cryptographic algorithms, if it contains inherent flaws or vulnerabilities in its design. Frequency analysis, differential cryptanalysis, and linear cryptanalysis are common methods. A classical encryption method that can be subject to frequency analysis is the Vigenère cipher, for instance, if the key is much shorter than the plaintext [1]. After determining the length of the key by the Kasiski examination or Friedman test, an attacker can utilize frequency analysis to crack the cipher. The ciphertext would be "LXFOPVEFRNHR" if the plaintext was "ATTACKATDAWN" and the key was "LEMON" repeated as "LEMONLEMONLE". Through the examination of letter frequencies and patterns, an intruder may be able to figure out the encryption key and decode the message. Knowing these flaws highlights the significance of strong cryptography practices because it shows that even popular encryption techniques can be broken under specific circumstances.

Bad key management procedures are another way for a cipher to be cracked. They may result in key exposure, which gives hackers access to the key and the ability to decode messages. This involves storing keys in an unsecured way, reusing keys across communications, and creating weak passwords. Uber experienced a notable data breach in 2017 as a result of subpar key management procedures. Uber's engineers used a private GitHub repository that was accessible to attackers; there were plaintext login credentials for Uber's Amazon Web Services (AWS) account. This gave the attackers access to private information that was kept in the cloud and affected 57 million drivers and consumers [2]. One of the main mistakes that caused the breach was that the GitHub account did not have multi-factor authentication (MFA), and credentials were stored in the code. This incident emphasizes how crucial it is to put strong key management processes in place in order to stop vulnerabilities like this and shield private data from unwanted access.

1d)

One critical component that determines an encryption algorithm's strength is key length. Brute-force attacks are much harder when the key length is longer since there are typically more key combinations available. Consider how much more secure a 128-bit key is than a 56-bit key like DES: it has 2^128 potential choices. Even with very powerful computers, attackers cannot computationally try every conceivable combination in a reasonable amount of time due to the expanded key space. Longer keys are an essential component of strong encryption because of this enormous increase in possible key combinations, which significantly increases resistance against brute-force attacks.

In order to ensure that the key is unexpected and does not display patterns that an attacker could exploit, key randomness is another important component. Cryptographically secure pseudorandom number generators (CSPRNGs) or true random number generators (TRNGs) should be used to generate keys. The security of the encryption can be compromised by attackers by guessing or deducing predictable keys or keys generated from weak sources. The probability of a successful key prediction or cryptanalysis is decreased by randomness, which removes patterns and guarantees that every key is distinct from all others. By leveraging strong randomization techniques, the overall robustness of cryptographic systems is greatly enhanced, providing a higher level of security against sophisticated attacks.

Appropriate key management procedures are the final important component since they support the security of encryption keys throughout their whole lifecycle. This covers the creation, distribution, use, and disposal of keys in a secure manner. Key management techniques include implementing multi-factor authentication (MFA), storing keys on hardware security modules (HSMs), rotating keys on a regular basis, and making sure keys are never sent in plaintext. Good key management keeps keys safe and undisclosed, lowers the chance of accidental key disclosure, and stops unwanted access. Insecure key management can make even the most powerful encryption algorithm useless. To protect their cryptographic systems, businesses must thus create and uphold strict key management procedures. Strict adherence to these best practices strengthens the security infrastructure overall and helps shield sensitive data from possible breaches.

Question 2

2a)

Using padding in cryptography methods like DES helps ensure that input data, such as plaintext messages, have the right length for encryption. Data encrypted by the block cipher DES is stored in fixed 64-bit (8-byte) blocks. It follows that padding bytes are appended to the last block, or the gap, if a plaintext message is not divided into these 8-byte segments evenly. All of the data blocks will be of the same size thanks to this method, which also ensures that the data fits neatly into the necessary block size [3]. Padding prevents inaccurate or partial encryption since it allows the encryption algorithm to process data that does not precisely match the block size. The proper operation of the encryption method and the completion of each block are made possible by padding, which results in precise encryption and decryption.

For example, if the plaintext message "HELLO" is only 5 bytes long, it needs 3 more bytes to fill the block and meet the 8-byte requirement. Using a padding scheme such as **PKCS#7,** the padded message would become "HELLO\x03\x03\x03," where each \x03 denotes the number of padding bytes added (the value 3 repeated). After encryption, the padded data can be processed correctly by the DES algorithm [3]. When decrypted, the algorithm recognizes the padding bytes and they are removed to retrieve the original message. This illustrates how important padding is to preserving the correctness and integrity of encrypted data and making sure that the encryption and decryption process functions as a whole. In order for the encryption technique to work properly, padding makes sure that every data block is the appropriate size. Encryption techniques such as DES would not work properly without adequate padding, which could result in data loss or corruption. The decryption procedure may fail because to incorrect or absent padding, leaving incomplete or jumbled data [4]. Padding thereby protects the dependability and security of the encrypted data and is more than just a technical feature of block cipher encryption.

2b)

1st Step (Substituting YYY with first 3 letters of family name (Soh))

The original text: “YYYComputerSecurity”

The replaced text: “SohComputerSecurity”

2nd Step (Determine the block size for DES)

DES operates on 8-byte (64-bit) blocks.

3rd step (Calculate the length of plaintext)

The replaced text: “SohComputerSecurity”

Length: 19 Characters

4th step (Convert the Plaintext to ASCII)

To perform padding, we convert each character of the plaintext to its ASCII equivalent.

Character	ASCII Value	Hexadecimal
S	83	53
o	111	6F
h	104	68
C	67	43
o	111	6F
m	109	6D
p	112	70
u	117	75
t	116	74
e	101	65
r	114	72
S	83	53

e	101	65
c	99	63
u	117	75
r	114	72
i	105	69
t	116	74
y	121	79

Thus, the plaintext “SohComputerSecurity” in hexadecimal is :

536F68636F6D7075746572536563757269747 9

5th step (Apply padding)

In DES, the block size is 64 bits (8 bytes). We must ensure that the plaintext length is a multiple of the block size (8 bytes). Our current plaintext length in bytes (characters) is 19.

To make the length a multiple of 8, we add 5 (which is 24-19) padding bytes (each byte is 00 in hex).

6th step (Calculate padding)

Add five 00 padding bytes to the plaintext

7th step (Combine plaintext with padding)

Combine the hexadecimal values of the original plaintext with the padding:

536F68636F6D70757465725365637572697479 0000000000

Final answer

The final result of padding for the text "SohComputerSecurity" in hexadecimal format is:

536F68636F6D70757465725365637572697479 0000000000.

2c)

To encrypt the plaintext "ComputerSecurity" using RSA, we first need to convert the plaintext into a numerical value.

1st Step

(**Character encoding -** Each character in the plaintext is converted to its ASCII (or UTF-8) value.)

For "ComputerSecurity":

C -> 67

o -> 111

m -> 109

p -> 112

u -> 117

t -> 116

e -> 101

r -> 114

S -> 83

e -> 101

c -> 99

u -> 117

r -> 114

i -> 105

t -> 116

y -> 121

2nd step

(**Concatenate Values -** These ASCII values are concatenated into a single large number. Assuming each value is zero-padded to three digits:

067111109112117116101114083101099117114105116121

3rd step

(In order to safeguard against selected plaintext attacks and make sure the plaintext satisfies the RSA block size criteria, padding is added to the plaintext prior to encryption. As an example, PKCS#1 v1.5 padding entails appending a random byte sequence of a certain format. Assume that our padding scheme adds 0002, then random bytes, and a 00 delimiter..)

Assuming our RSA block size is 256 bytes and the plaintext occupies 45 bytes:

Padding: 0002[Random Bytes]00[Plaintext]

Convert to Number:

0002A1B2C3D4000671111091121171161011140831010991171141051161 21

This hexadecimal string can be directly used as a large integer for the RSA encryption.

If public key (e, N) is provided. Suppose e=65537 and N is a 2048-bit number. The plaintext number
0002A1B2C3D4000671111091121171161011140831010991171141051161 21

is then encrypted using the RSA formula $C=M^e \bmod N$.

By converting the plaintext "ComputerSecurity" to a numerical value through ASCII encoding, concatenation, and appropriate padding, we prepare the message for RSA encryption [5]. This ensures the message is secure and compatible with RSA encryption requirements. This process ensures that each character in the plaintext is transformed into its corresponding ASCII code, which is then concatenated to form a large numerical

representation of the entire message [6]. Appropriate padding is applied to meet the length requirements of the RSA algorithm, which further enhances the security and compatibility of the message for encryption [7]. This method guarantees that the message adheres to the format necessary for RSA encryption, thereby ensuring its secure and effective encryption and subsequent decryption. By following these steps, we maintain the integrity and confidentiality of the message throughout the encryption process.

2d)

A digital signature is a cryptographic method and technique that is used for validation of the integrity and authenticity of software, digital messages, and digital documents [8]. It ensures that the message originates from the stated sender, has not been tampered with during transmission, and that the sender cannot deny sending the message, providing authenticity, integrity, and non-repudiation. Digital signatures verify the sender's identity and the integrity of the content by utilizing the sender's public key for verification and the sender's private key to generate a unique hash value for the message [9]. Because of this, digital signatures are necessary for safe communication, software distribution, money transfers, and the handling of legal documents.

It is possible to create digital signatures with the RSA asymmetric encryption algorithm. During the procedure, a safe cryptographic hash function like SHA-256 is used to hash the original message. A fixed-size hash value that is specific to the original message is produced as a result. The sender's private RSA key is then used to encrypt the hash value that is produced. Together with a few other pieces of data, this encrypted hash value makes up the digital signature. Using the sender's public RSA key, the recipient decrypts the signature to acquire the original hash value. In addition, the recipient hashes the message they received and contrasts the two hash values. The validity of the

signature verifies the integrity and authenticity of the message if they match. If they don't match, there may have been an attempted manipulation or security breach since it suggests that the message has been changed or the signature is invalid. This procedure guarantees the integrity and authenticity of the message because it instantly notifies the recipient of any potential problems if there is a disparity in the hash values.

Since DES uses a symmetric key encryption algorithm—one in which the same key is used for both encryption and decryption—it is not possible to generate digital signatures with it. Symmetric key algorithms, such as DES, are not appropriate for creating digital signatures since they require a pair of keys, one private and one public. A digital signature system uses the private key to sign the message, making it possible for only the owner of the private key to produce a unique signature. The signature is then verified using the associated public key, which is freely distributable. By using a key pair system, non-repudiation and validity are guaranteed, and the signature may be independently confirmed by anybody in possession of the public key. Symmetric key algorithms are unable to facilitate the production of digital signatures since they do not offer this key separation. Asymmetric key algorithms, like RSA, are the foundation of digital signatures. The message is signed with the private key and verified with the public key.

References

1. GeeksforGeeks. (2023). Vigenère Cipher. [online] GeeksforGeeks. Available at: https://www.geeksforgeeks.org/vigenere-cipher/ (Accessed: 3 July 2024).
2. Pascucci, M. (2018). Uber breach: How did a private GitHub repository fail Uber? [online] TechTarget. Available at: https://www.techtarget.com/searchsecurity/answer/Uber-breach-How-did-a-private-GitHub-repository-fail-Uber (Accessed: 3 July 2024).
3. Williamson, B. (2024). DES Padding. [online] A Security Site. Available at: https://asecuritysite.com/symmetric/padding_des (Accessed: 3 July 2024).
4. Wikipedia. (2024). Padding (cryptography). [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Padding_(cryptography) (Accessed: 3 July 2024).
5. Stack Overflow. (2010). How do I treat a plain text message as a numerical value for encryption algorithms? [online] Stack Overflow. Available at: https://stackoverflow.com/questions/3692347/how-do-i-treat-a-plain-text-message-as-a-numerical-value-for-encryption-algorit (Accessed: 3 July 2024).
6. Crypto Stack Exchange. (2021). RSA - The algorithm aside, how are we turning a string into an int and vice versa? [online] Crypto Stack Exchange. Available at: https://crypto.stackexchange.com/questions/97934/rsa-the-algorithm-aside-how-are-we-turning-a-string-into-an-int-and-vice-versa (Accessed: 3 July 2024).
7. Crypto Stack Exchange. (2012). Why is padding used for RSA encryption, given that it is not a block cipher? [online] Crypto Stack Exchange. Available at: https://crypto.stackexchange.com/questions/3608/why-is-padding-used-for-rsa-encryption-given-that-it-is-not-a-block-cipher (Accessed: 3 July 2024).
8. Microsoft. (n.d.). Digital signatures and certificates. [online] Microsoft Support. Available at: https://support.microsoft.com/en-us/office/digital-signatures-and-certificates-8186cd15-e7ac-4a16-8597-22bd163e8e96 (Accessed: 3 July 2024).
9. Wikipedia. (2024). Digital signature. [online] Wikipedia. Available at: https://en.wikipedia.org/wiki/Digital_signature (Accessed: 3 July 2024).