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:	Imran	Student ID:	20696366
Other name(s):	Nuha		
Unit name:	Fundamentals of Cryptography	Unit ID:	ISEC2000
Lecturer / unit coordinator:	Dr Mohan Geetha	Tutor:	-
Date of submission:	25/06/23	Which assignment?	First (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: Nuha Imran Date of signature: 25/06/23

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

6/23/2023

Cryptography Assignment

Nuha Imran
ID:20696366

Glossary:

1)Letter Frequency Analysis Attack

-code explanation step by step

-The substitution table

2)Affine Cipher

-code explanation step by step

-the key I found

3)DES (Data encryption Standard)

- explanation of code step by step

-What issues I ran into and If I was able to decrypt

- What will happen if the key is initialized as all 0-bits

Letter Frequency Analysis Attack:

What is being done in the code step-by-step?

Step 1: First, the file is read from the command line using system arguments and f.read(), and an exception is thrown in case of the file is not found in the same directory

Reasoning: This is being done so that the file that is to be decrypted is read

```
# Step 1: Reading the file that has to be decrypted
try:
    with open(sys.argv[1]) as f:
        message = f.read()
except FileNotFoundError:
    print("File not found")
    exit(1)
```

Step 2: The next step is simply taking all the content in the file and making it lowercase so that it doesn't intervene with the decryption process.

Reasoning: If all letters don't have a constant format they might intervene with the process of decryption

```
# Step 2: Removing all characters and formatting that would int
message = message.lower()
```

Step 3: Next step is calculating the number of letters in the text file to evaluate the frequencies. Once counted the repetition of each letter is being counted. Once that's done the frequency is calculated which is the count of letters over the total amount of letters all this is stored in a list

Reasoning: This is simply to calculate the frequencies of the letters with the formula count/total letters in file and storing in a dictionary as letter and its count.

```
# Step 3: Checking all occurrences of letters in the text to evaluate the repeated frequencies
message_length = len(message)

letter_counts = {letter: message.lower().count(letter) for letter in string.ascii_lowercase}

letter_frequencies = {letter: (count / message_length) for letter, count in letter_counts.items()}
```

Step 4: Next, I got a table for English frequencies from SAN and put them in a list.

Reference: https://blogs.sas.com/content/iml/2014/09/19/frequency-of-letters.html

```
# Step 4: Putting in the dictionary of English letter analysis because the encrypted text is in English
eng = {
    'e': 12.49, 't': 9.28, 'a': 8.04, 'o': 7.64, 'i': 7.57, 'n': 7.23,
    's': 6.51, 'r': 6.28, 'h': 5.05, 'l': 4.07, 'd': 3.82, 'c': 3.34,
    'u': 2.73, 'm': 2.51, 'f': 2.40, 'p': 2.14, 'g': 1.87, 'w': 1.68,
    'y': 1.66, 'b': 1.48, 'v': 1.05, 'k': 0.54, 'x': 0.23, 'j': 0.16,
    'q': 0.12, 'z': 0.09
}
```

Step 5: After getting both lists I mapped them onto each other, first sorted them in descending order as in highest to lowest hence the reverse:True this was to prioritize the most frequent elements. After sorting was done, I made a loop and mapped each sorted element next to each other in a dictionary so hypothetically I made a table of mapped frequencies from the text and the English letter frequencies.

```
# Step 5: Creating a mapping dictionary based on the frequencies in a logical way
mapping_dict = {}
sorted_text_frequencies = sorted(letter_frequencies, key=lambda x: letter_frequencies[x], reverse=True)
sorted_eng_frequencies = sorted(eng, key=lambda x: eng[x], reverse=True)

for i in range(len(sorted_text_frequencies)):
    mapping_dict[sorted_text_frequencies[i]] = sorted_eng_frequencies[i]
```

Step 6: Next step I just used that dictionary and replaced every text letter in the file with its mapped letter from the English frequency.

```
# Step 6: Decrypting the text based on the mapping dictionary
decrypted_text = ""
for letter in message:
    decrypted_letter = mapping_dict.get(letter, letter)
    decrypted_text += decrypted_letter
```

Step 7: After this was done and I have the decrypted text I printed it to a file.

```
# Step 7: Writing the decrypted text to an output file
with open('decrypted_output.txt', 'w') as f:
    f.write(decrypted_text)
```

The substitution table that was formed.

```
v → e
o → t
j → a
z → o
i → i
l → n
h → s
w → r
e → h
q → l
s → d
r → c
p → u
y → m
b → f
t → p
c → g
d → w
m → y
x → b
u → v
n → k
g → x
f → j
k → q
a → z
```

Reasoning as to why the text wasn't decrypted:

The letter frequency analysis attack was not able to decrypt the text due to the random assignment of substitutions, which did not follow the typical letter frequency distribution of the English language. As a result, the disrupted patterns and frequencies hindered the effectiveness of the letter frequency analysis technique. In such cases, where the substitutions do not reflect the actual letter frequencies of the target language, an alternative approach such as a substitution cipher would be more suitable for decrypting the text. A substitution cipher involves replacing each letter with a different predetermined letter, allowing for a consistent and predictable pattern that can be deciphered with the appropriate key or knowledge of the substitution scheme. In case I changed the substitution table it changed the entire outcome however still did not decrypt showing the efficiency of the algorithm

Affine Cipher Brute Force Attack:

What have you done in your code step-by-step?

Step 1: Is a function that reads the file entered in the command line removes the trailing white spaces and returns it.

```
# Step 1: Read the encrypted text from a file(Strip is to remove trailing white spaces for decryption process)
def read_file(file_path):
    try:
        with open(file_path, 'r') as f:
            encrypted_text = f.read().strip()
        return encrypted_text
    except FileNotFoundError:
        print("File not found.")
        exit(1)
```

Step 2: These functions are required for the decryption process. They include calculating the greatest common divisor (GCD) of two numbers, finding the modular inverse of a number, and checking if a word is present in the decrypted text. The concept of checking word is a small dictionary with words that appear in almost every text.

```python
# Step 2: Defining all necessary functions needed for the decryption
# Step 2a:Function to calculate the greatest common divisor (GCD) of two numbers
# Reasoning:Needed because if a and m are not coprimes the key a,b is not possible
def gcd(a, b):
    if b == 0:
        return a
    return gcd(b, a % b)

# Step 2b: Function to find the modular inverse of a number
# Reasoning: If a modular inverse does not exist then the a,b keys cannot be used for decryption
def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return None
```

```python
# Step 2c: Function to check if a word is present in the decrypted text
# Reasoning: Simply an extra function to ensure the right decrypted key is being chosen
def check_word_in_text(word, decrypted_text):
    decrypted_words = decrypted_text.lower().split()
    if word.lower() in decrypted_words:
        return True
    return False
```

Step 3: Here I just declared all my variables that could be encrypted and made a small dictionary with words that are present in almost every sentence, I added this so that latter when the keys are being brute forced through the code can itself choose the correct decrypted version.

```python
# Step 3: Setting up all necessary variables and lists
# Step 3a: Define the alphabet (characters that can be encrypted)
alphabet = 'abcdefghijklmnopqrstuvwxyz'

#Step 3b: Basic English word list for checking words(small dictionary for checking words that exist in most sentences
english_words = ['the', 'you', 'are', 'who', 'of', 'are', 'a']
```

Step 4: In this part, I'm error checking and reading the file given in the command line through the method I made.

```python
# Step 4: Performing the brute-force attack
# Step 4a: Read the file path from command line argument & error checking
if len(sys.argv) < 2:
    print("Please provide the file path as a command line argument.")
    exit(1)
file_path = sys.argv[1]

# Step 4b: Read the encrypted text from the file
encrypted_text = read_file(file_path)
```

Step 5: This is where I'm doing the brute force with a dictionary of all possible key values the possible keys is to store the possible key pairs then I am iterating for a and then checking if a and 26 are coprime as in their greatest common divisor is one then iterating for the value of b and then iterating for values of b

```
# Step 5: Decrypting the text using the brute-force approach
possible_keys = []

for a in range(1, 26):
    if gcd(a, 26) == 1:
        for b in range(26):
            decrypted_text = ''
```

Step 6: Each character in the encrypted text is iterated over by this block of code. If the character is alphabetical, it uses the formula (a_inverse * (char_idx - b)) to determine the decrypted character index. %26 where 'a_inverse' is the modular inverse of 'a', 'char_idx' is the index of the character in the alphabet, 'b' is the additive offset, and '26' denotes the total number of characters in the alphabet. A character that has been decrypted is then added to the decrypted_text string. A space or other non-alphabetical character, for example, is immediately added to the decrypted_text string.

```
# Step 6: Decrypt the text using the current key pair
for char in encrypted_text:
    if char.isalpha():
        char_idx = alphabet.index(char.lower())
        a_inverse = mod_inverse(a, 26)
        decrypted_char_idx = (a_inverse * (char_idx - b)) % 26
        decrypted_char = alphabet[decrypted_char_idx]
        decrypted_text += decrypted_char
    else:
        decrypted_text += char
```

Step 7:In step 7 I am iterating over each word in the English_words list and checking the word exists in the decrypted text if it's true I return true if a=false I return false the all() function calls for each word return true and if all words found the condition evaluates to true. If this condition is satisfied the code executes and a key pair is appended to the list indicating a potential solution. The message is printed telling the key pair and giving the decrypted text.

```
# Step 7: Check if the decrypted text contains basic English words
if all(check_word_in_text(word, decrypted_text) for word in english_words):
    possible_keys.append((a, b))
    print("Possible key found: a={}, b={}".format(a, b))
    print("Decrypted text: {}".format(decrypted_text))
    print()
```

Step 8: first I check if the possible key list is not empty if its not iterate over all keys, the key pair is then received and decrypted_text is initialised as an empty string, the code then iterates over each character in the encrypted text and checks if its an alphabetic character using the is alpha if it is alphabetic the code obtains the index of the character in the alphabet string considering it is in lowercase, then it calculates the modular inverse of a using mod inverse function. Then it calculates the decrypted char index using the key pair. Then retrieves the decrypted character from the alphabet string based on decrypted characters index it appends this decrypted character to the decrypted text string in case of the character not being alphabetical e.g whitespace or punctuation it simply appends the character to decrypted text string. After decrypting the entire thing using the current key pair the code prints the decrypted text along with the keys a and b and if no possible keys were retrieved it just prints that message.

```
# Step 8: Decrypt the text using the correct key
if len(possible_keys) > 0:
    for key in possible_keys:
        a, b = key
        decrypted_text = ''

        for char in encrypted_text:
            if char.isalpha():
                char_idx = alphabet.index(char.lower())
                a_inverse = mod_inverse(a, 26)
                decrypted_char_idx = (a_inverse * (char_idx - b)) % 26
                decrypted_char = alphabet[decrypted_char_idx]
                decrypted_text += decrypted_char
            else:
                decrypted_text += char

        print("Decrypted text with key (a={}, b={}):".format(a, b))
        print(decrypted_text)
else:
    print("No possible keys found.")
```

The key found by brute-force attack?

The key found in the brute force attack is (3,9) where a is 3 and b is 9

3.DES (Data encryption standard):

Initial Permutation (IP) table: The IP table rearranges the input plaintext bits to a specific permutation before the main encryption rounds begin. It helps to distribute the input data across multiple rounds and ensures that each bit influences the subsequent encryption steps.

Expansion table (E): The expansion table expands the right half of the data from 32 bits to 48 bits. It replicates certain bits and rearranges them to introduce diffusion and provide more mixing of data during the encryption process.

Permutation (P) table: The P-box permutation is applied after the Feistel function in each encryption round. It shuffles the bits to increase the complexity and ensure that the output of each round affects the subsequent rounds in a non-linear manner.

S-boxes: The S-boxes are substitution tables used for the main confusion step in each encryption round. They take 6 bits as input and output 4 bits. The S-boxes introduce a high degree of non-linearity, making the encryption resistant to linear and differential cryptanalysis attacks.

Final Permutation (FP) table: The FP table is the inverse of the initial permutation. It rearranges the bits of the ciphertext generated after the encryption rounds to provide the final output in the desired format.

PC-1 and PC-2 tables: The PC-1 and PC-2 tables are used in the key generation process. PC-1 permutes the initial key bits to eliminate the parity bits and select the 56 most significant bits. PC-2 selects the 48 bits from the shifted and compressed key to generate the subkeys for each encryption round.

These tables and permutations collectively contribute to the security properties of DES, including diffusion (spreading the influence of each plaintext bit to multiple ciphertext bits) and confusion (making the relationship between the key and the ciphertext complex and difficult to analyze). They also provide resistance against known attacks like differential and linear cryptanalysis.

The necessary functions:

1. Key Scheduler:

```python
# Step 8: Key Scheduler
def key_scheduler(initial_key):

    initial_key = hex_to_binary(initial_key)
    initial_key = adjust_key_length(initial_key)

    # Step 8.1: Perform PC-1 permutation
    key_permuted = permute(initial_key, PC_1_TABLE)

    # Step 8.2: Remove 8 parity bits
    key_no_parity = key_permuted[0:56]

    # Step 8.3: Break the 56 bits into two halves
    c = key_no_parity[0:28]
    d = key_no_parity[28:56]

    # Step 8.4: Perform left circular shift on each half
    round_keys = []
    for i in range(1, 17):
        c, d = lshift(c, d, i)
        round_key = permute(c + d, PC_2_TABLE)  # Perform PC-2 permutation
        round_keys.append(round_key)

    return round_keys
```

This key scheduler method takes the initial key and converts it from hexadecimal to binary using a function made above in the code named ‘hex_to_binary’ and the length of the key is modified to 64 bits using another method named ‘adjust_key_length’ once its converted and made to 64bits, PC-1 permutation is applied to rearrange the bits of the key to generate 56-bit key with parity bits. Further, the 8 parity bits are removed from the permuted key in order to obtain a 56bit key without parity. The key is then divided into two halves named ‘c’ and ‘d’ based on the round number using the lshift function made above. The number of positions being shifted is defined by LSHIFT_MAP. After the circular shift on c and d the halves are concatenated and undergo the PC-2 permutation using the ‘permute’ function and the PC_2_table. This permutation produces a 48bit round key. The round key is appended to a list of round keys. Finally, the list of round keys is returned.

2. Permutation:

```python
#Step 5: Making a permute function
def permute(bits, permutation_table):
    permuted_bits = [bits[index - 1] for index in permutation_table]
    return ''.join(permuted_bits)
```

This function performs permutation it basically takes as input a string and a permutation table and performs permutation on the input bits according to the specified permutation table. In-depth for each index in the permutation table, the function retrieves the bit from the corresponding position in the input bit string. The retrieved bits are stored in a list, ‘permuted_bits’, in order that is specified by the permutation table. In the end the function joins the bits in ‘permuted_bits’ together into a string and returns the resulting permuted bit string.

3. S_box:

```
# Step 10: S-box substitution
def s_box_substitution(bits):
    sbox_output = ''
    for i in range(8):
        row = binary_to_decimal(int(bits[i * 6] + bits[i * 6 + 5]))
        column = binary_to_decimal(int(bits[i * 6 + 1] + bits[i * 6 + 2] + bits[i * 6 + 3] + bits[i * 6 + 4]))
        value = S_BOXES[i][row][column]
        sbox_output = sbox_output + decimal_to_binary(value)

    return sbox_output
```

S-boxes are used in this function to accomplish substitution.S-boxes are tables that enable nonlinear substitution operations on data blocks in symmetric key cryptgraphic algorithms like DES.The function first creates a string variable named "sbox_output" that is empty to hold the results of the s_box subsitition.Then, each of the 8-S boxes is iterated over by the functions. The row index is then calculated by adding the first and last bits of the 6-bit input block (bits[i * 6] + bits[i * 6 + 5]) using the binary-to-decimal function. This method is carried out for every S-box. The column index is then calculated by concatenating the middle 4 bits of the 6 bit input blck (bits[i * 6 + 1] through bits[i * 6 + 2], bits[i * 6 + 3], and bits[i * 6 + 4]) using the binary_to_decimal function. It retrieves the value from the S-box table using the calculated row and column indices (S_BOXES[i]).The obtained value is then converted into a binary representation using the decimal_to_binary function.The binary value obtained from the S-box substitution is concatenated to produce the sbox_output string. The function returns the final sbox_output string, which represents the S-box substitution outcome, after processing all 8 S-boxes.In summary, the s_box_substitution function performs S-box substitution on a given input block of bits. It uses the bits as indices to access values from predefined S-box tables, converts the retrieved values to binary representation, and concatenates them to form the output of the S-box substitution. S-box substitution adds a nonlinear component to cryptographic algorithms and contributes to the strength of the encryption process.

4. F function

```
def f(left, right, subkey, iteration):
    # Step 1: Expansion permutation
    expanded_right = expansion_permutation(right)

    # Step 2: First XOR
    xored_right1 = xor(expanded_right, subkey)

    # Step 3: S-box substitution
    sbox_output = s_box_substitution(xored_right1)

    # Step 4: Permutation
    permuted_output = permute(sbox_output, p_box_permutation)

    # Step 5: Second XOR
    xor_right2 = xor(left, permuted_output)

    left = xor_right2

    if iteration != 15:
        left, right = right, left


    return left, right
```

This function is the key component of DES algorithms, representing the round function that operates on the left and right halves of the data block . The f function takes four parameters: left, which represents the left half of the data block, right, which represents the right half of the data block, subkey, which is the round key for the current iteration, and iteration, which indicates the current iteration number.

The first step here is expansion permutation the expanded_right variable is assigned the result of the expansion_permutation function, which expands the right half from 32 bits to 48 bits using a predefined expansion table.The second step is XOR function. The xored_right1 variable is assigned the result of the xor function, which performs a bitwise XOR operation between the expanded_right and subkey.

This XOR operation combines the expanded right half with the round key.

The third step is the s-box substitution. The sbox_output variable is assigned the result of the s_box_substitution function, which performs S-box substitution on the xored_right1 value.

S-box substitution replaces the 48-bit input with 32-bit output using a series of eight S-box tables.

The fourth step is where the permutation is performed. The permuted_output variable is assigned the result of the permute function, which performs a permutation operation on the sbox_output using a predefined permutation table called p_box_permutation. The permutation table rearranges the bits in sbox_output to produce the final output.

Fifth step here is the second XOR.The xor_right2 variable is assigned the result of the xor function, which performs a bitwise XOR operation between the left half and permuted_output.

This XOR operation combines the output of the permutation step with the original left half.

The left variable is updated with the value of xor_right2. This step is necessary because the left and right variables will be swapped in the next step, except for the last iteration.

If the current iteration is not the 15th (iterations are zero-based):

The values of left and right are swapped using the tuple assignment left, right = right, left.

This swap ensures that the new left value becomes the right value for the next iteration.

Finally, the updated left and right values are returned.

4.Encryption:

```
def des_encrypt(plaintext, round_keys):
    # Step 1: Convert the plaintext and key to binary strings
    plaintext_binary = hex_to_binary(plaintext)

    if len(plaintext_binary) % 64 != 0:
        plaintext_binary = pad_binary(plaintext_binary)

    # Step 4: Initial permutation
    list_block= []
    no_of_blocks = int(len(plaintext_binary) / 64)
    for i in range(no_of_blocks):
        block = plaintext_binary[i * 64:(i+1) * 64]
        block = permute(block, initial_permutation)
        left = block[0:32]
        right = block[32:64]
        for j in range(16):
            left, right = f(left, right, round_keys[j], j)
        combined = left + right
        ciphertext = permute(combined, final_permutation)
        list_block.append(ciphertext)

    ciphertext_hex = ''.join(list_block)

    return ciphertext_hex
```

The plaintext and round keys are input into the des_encrypt function, which uses the DES (Data Encryption Standard) algorithm to accomplish the encryption.

The plaintext is transformed into a binary string as the first step in the encryption process.Using the hex_to_binary function, the plaintext is transformed from a hexadecimal string to a binary string.The length of the plaintext binary string is next tested to see if it is a multiple of 64 bits: The modulus operator% is used to determine whether the plaintext_binary's length is evenly divisible by 64. Adding padding to make it a multiple of 64 bits if necessary. Padding ensures that the plaintext can be divided into equal-sized blocks for encryption. The plaintext_binary is divided into blocks of 64 bits each.

For each block, the initial permutation is applied using the permute function and the initial_permutation table.

The initial permutation rearranges the bits in each block based on a predefined permutation table.

Loop over the blocks and perform the encryption process: The no_of_blocks variable is calculated based on the number of 64-bit blocks in the plaintext_binary. A loop is executed for each block. Within each block, the left and right halves are extracted from the block using slicing operations. A nested loop is executed 16 times (for each of the 16 rounds of DES encryption). Inside the nested loop, the f function is called with the left, right, and the corresponding round_key as arguments. The iteration value is also passed as the loop index j.

The f function updates the left and right values based on the DES encryption steps.

Combine the updated left and right values and perform the final permutation:

The left and right halves are combined into a single string called combined.

The final permutation is applied to combined using the permute function and the final_permutation table.

The final permutation rearranges the bits in combined based on a predefined permutation table.

The final block of ciphertext is added to a list of other blocks.

The ciphertext block list should be transformed into a hexadecimal string.

The binary ciphertext is converted into a single string representation by joining the list of blocks together using the function ".join(list_block). After that, a hexadecimal string is created from the binary ciphertext string. The ciphertext's hexadecimal form is given back. Overall, the des_encrypt function divides the plaintext into blocks, applies the initial permutation, performs 16 rounds of encryption using the f function and the round keys, and applies the final permutation to generate the ciphertext.

5.Decryption:

```
def des_decrypt(ciphertext, key):
    keys_reversed = key[::-1]
    plainText = des_encrypt(ciphertext, keys_reversed)
    return plainText
```

The des_decrypt function performs the decryption process using the DES algorithm.

Reverse the order of the key: The key is reversed using the slicing operation key[::-1]. Reversing the key is necessary because the round keys used for encryption and decryption in DES are applied in the reverse order. Perform decryption using des_encrypt with reversed keys: The ciphertext and reversed keys_reversed are passed as arguments to the des_encrypt function. The des_encrypt function is called to perform the encryption process in reverse, which effectively decrypts the ciphertext using the reversed round keys. The result of the decryption is assigned to plainText.

Return the decrypted plaintext: The decrypted plaintext is returned as the output of the des_decrypt function.

By reversing the key and passing it to the des_encrypt function, the des_decrypt function effectively reverses the encryption process. It applies the reversed round keys in reverse order to undo the encryption and obtain the original plaintext.

(a) Have you successfully recovered the plaintext? What are the lessons you learned, and difficulties you met, in the process of implementing DES?

Yes, I was able to successfully recover the plaintext. However, I encountered several challenges and gained valuable insights during the implementation of the DES algorithm.

First and foremost, understanding the intricacies of the algorithm itself proved to be a significant task. With multiple rounds and complex steps involved, I had to handle both the plaintext and the key and combine them effectively. Grasping the algorithm at a deep level to implement it was certainly not a straightforward process.

The key scheduler component posed numerous challenges initially. I faced issues with proper conversion of the key bits, which resulted in errors. Through diligent debugging and careful examination, I discovered that the incorrect key bits were being generated due to my oversight in converting them to binary representation. Rectifying this flaw was essential to ensure the algorithm's correctness.

Furthermore, the extensive conversions throughout the code, including hex to binary, string manipulation, and ASCII representation, sometimes caused slight confusion and required meticulous attention to detail. Accuracy in these conversion steps was crucial for the overall functioning of the algorithm.

During the execution of the code, I encountered potential errors that prevented successful decryption. One such issue was the absence of key rounding in the decryption function. Through multiple sessions of debugging and rigorous testing, I was able to identify and rectify these errors, which significantly improved my understanding of the code.

In the process, I realized that while DES is an intelligently designed algorithm, it may not be as secure as originally perceived. With a 56-bit key length, it has become susceptible to brute-force attacks using modern computing technology. Obtaining the decrypted text with the correct key is no longer an arduous task. Consequently, I personally believe that opting for other algorithms, such as AES, would provide enhanced security and peace of mind.

Overall, implementing DES has been an enlightening journey that deepened my understanding of encryption algorithms and their challenges. It highlighted the importance of robust implementation, thorough testing, and staying abreast of advancements in cryptographic security.

(b) What will happen if the key is initialized as all 0-bits? Explain your reasoning.

The security of the encryption will be seriously jeopardised if the DES algorithm initialises the key as only 0-bits. In DES, the key is essential for creating the subkeys that are used for both encryption and decryption. When a key has only 0 bits, all subkeys that are derived from it will also have 0 bits. As a result, each round's key will always be the same, and the encryption process will basically consist of a series of XOR operations with a fixed value.

Due to the repetitive usage of the same key, which leaves the encryption vulnerable to numerous types of attacks such brute-force attacks, differential cryptanalysis, and linear cryptanalysis, this scenario significantly reduces the security of DES. An attacker can learn more about the encryption process and possibly recover the plaintext with a great deal less effort by taking advantage of the recurrence of the key.

The use of a robust and unpredictable key is essential to ensuring DES security. A key with only 0 bits weakens the encryption's security and should be avoided.

References:

https://www.geeksforgeeks.org/data-encryption-standard-des-set-1/