Encryption & Decryption

ALGORITHM

Information Security

Aleeza Abid 4442-F22-A

Musfira Tanvir 4476-F22-A

Mahnoor Jabbar 4467-F22-A

Syeda Zarlish Ahmed 4493-F22-A

Instructor: Dr. Umara Zahid

HOW OUR ALGORITHM WORKS?

We have made an algorithm. The requirements of our encryption algorithm is: 8 rounds, key of 128 bit, and 64 bit plain text. It is based on combination of DES, vigenere, playfair, additional and keyless (rail fence) cipher. Here are encryption steps:

1. Our 64 bit plaintext undergoes initial permutation as in DES. Then broken into 2 parts of 32 bits.
2. Our 128 bits key is broken into 2 parts of 64 bits. XOR operation is applied on both parts which results as a single key of 64 bits.
3. Key (64 bits) undergoes compressed permutation and transformed into 56 bits. Which are then broken into 2 parts of 28 bits.

Now the round begins. The same steps are applied in all 8 rounds. The steps in each round are:
1. The 2 parts of key (each having 28 bits) undergoes circular shifting as shown in this table.

CIRCULAR SHIFTING TABLE:

ROUND	SHIFTING BITS
1	1
2	1
3	2
4	2
5	2
6	2
7	2
8	2

2. After shifting the left and right half of key is combined into a single key. And again permutation is applied. After this permutation a 48 bit key is obtained. Here is permutation table:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

3. the plain text which was divided into 2 parts of 32 bits. The right 32 bits are expanded to 48 bits using the following permutation table.

EXPANSION TABLE:

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

4. XOR operation is applied on these 48 bits of plaintext and the 48 bit key as in DES.

5. The 48 bits obtained after XOR operation undergoes S-Box. Here is the S-Box :

S BOX:

	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	6	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

6. then the 32 bits will obtained from S-Box. On the first four rounds playfair cipher is applied to these 32 bits and on the next 4 round (5 to 8 rounds) vigenere cipher is applied to these 32 bits.

Here how the playfair cipher is applied :The secret key which will be used is given below:

l	g	d	b	a
q	m	h	e	c
u	r	n	i/j	f
x	v	s	o	k
z	y	w	t	p
!	#/%	@	&	?

The 32 bits are firstly divided into chunks of 5 bits in this sequence (5 , 5 , 5 , 5 , 5 , 5 , 2)
The last 2 bits will not go into playfair cipher and will be copied as it is in the last.

7. Then the bits are changed into alphabets. 0-25 are assigned to a-z and for the symbols we use the following table

Symbols	!	#	%	@	&	?
Values	26	27	28	29	30	31

Here how the vigenere cipher is applied :

The secret key for this cipher will be PASCAL

1. Firstly this keyword will transformed into bits
2. The 32 bits are firstly divided into chunks of 5 bits in this sequence (5 , 5 , 5 , 5 , 5 , 5 , 2) The last 2 bits will not go into vigenere cipher and will be copied as it is in the last.
3. The 32 bits of our keyword and the the 32 bits coming from the S-Box will undergoes through XOR operation and the 32 bits cipher key is obtained.

7. After applying the ciphers the 32 bit cipher key is again permutated.

8. The 32-bit left half of plaintext and the 32-bit cipher key which is obtained in the above step will undergoes the XOR operation.

9. On the 1 , 3 , 5 . 7 rounds additive cipher and keyless cipher will be applied on 2 , 4 , 6 , 8 rounds. These ciphers will be applied on the result of XOR operation.

For decryption, use this table for final permutation:
FINAL PERMUTATION:

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

BRUTE FORCE ANALYSIS:

Primary Key (128-bit):
The main key used in your algorithm is 128 bits. This key is the most important and is the only one that varies.
Total possible combinations: 2^128 (which is an enormous number, approximately 3.4×10^38).
Even with powerful computers, breaking this key through brute force would take an impractical amount of time.

Traditional ciphers used:

A single 128 bits key takes a lot of time to for attacker to break through brute force. But our algorithm doesn't works on a single rather than keys of vigenere, playfair, additional ciphers used. So the time an attacker needs to break our algorithm is basically equivalent to the time required to break the keys of all 4 ciphers used within.

PYTHON CODE:

```
import binascii


# Ensure binary strings are padded to 64 bits
def pad_binary(bits, size=64):
    return bits.zfill(size)


# Convert string to binary (with 64-bit padding)
def string_to_binary(text):
    return ''.join(format(ord(char), '08b') for char in text)


# Convert binary to string
def binary_to_string(binary):
    chars = [binary[i:i + 8] for i in range(0, len(binary), 8)]
    return ''.join([chr(int(char, 2)) for char in chars])


# Initial Permutation (DES-like)
def initial_permutation(bits):
    perm_table = [58, 50, 42, 34, 26, 18, 10, 2,
                  60, 52, 44, 36, 28, 20, 12, 4,
                  62, 54, 46, 38, 30, 22, 14, 6,
                  64, 56, 48, 40, 32, 24, 16, 8,
                  57, 49, 41, 33, 25, 17, 9, 1,
                  59, 51, 43, 35, 27, 19, 11, 3,
                  61, 53, 45, 37, 29, 21, 13, 5,
                  63, 55, 47, 39, 31, 23, 15, 7]
    return ''.join([bits[i - 1] for i in perm_table])


# Final Permutation (DES-like)
def final_permutation(bits):
    perm_table = [40, 8, 48, 16, 56, 24, 64, 32,
                  39, 7, 47, 15, 55, 23, 63, 31,
                  38, 6, 46, 14, 54, 22, 62, 30,
                  37, 5, 45, 13, 53, 21, 61, 29,
                  36, 4, 44, 12, 52, 20, 60, 28,
                  35, 3, 43, 11, 51, 19, 59, 27,
                  34, 2, 42, 10, 50, 18, 58, 26,
                  33, 1, 41, 9, 49, 17, 57, 25]
    return ''.join([bits[i - 1] for i in perm_table])


# Expansion function
def expand_right_half(bits):
    expansion_table = [32, 1, 2, 3, 4, 5,
                       4, 5, 6, 7, 8, 9,
```

```
                    8, 9, 10, 11, 12, 13,
                    12, 13, 14, 15, 16, 17,
                    16, 17, 18, 19, 20, 21,
                    20, 21, 22, 23, 24, 25,
                    24, 25, 26, 27, 28, 29,
                    28, 29, 30, 31, 32, 1]
    return ''.join([bits[i - 1] for i in expansion_table])

# S-box transformation
def s_box(bits):
    sbox = [
        [14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
        [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
        [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
        [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]
    ]
    output = ""
    for i in range(0, len(bits), 6):
        chunk = bits[i:i + 6]
        row = int(chunk[0] + chunk[5], 2)
        col = int(chunk[1:5], 2)
        output += format(sbox[row][col], '04b')
    return output

# XOR function
def xor(bits1, bits2):
    return ''.join('1' if b1 != b2 else '0' for b1, b2 in zip(bits1, bits2))

# Key generation with circular shifts
def generate_round_keys(key):
    key = string_to_binary(key)
    key = key[:128].zfill(128)  # Ensure the key is 128 bits
    round_keys = []
    left_half, right_half = key[:64], key[64:]
    shifts = [1, 1, 2, 2, 2, 2, 2, 2]

    for shift in shifts:
        left_half = left_half[shift:] + left_half[:shift]
        right_half = right_half[shift:] + right_half[:shift]
        combined = left_half + right_half

        # Compression Permutation
        permuted_key = ''.join([combined[i - 1] for i in [14, 17, 11, 24, 1, 5,
                                                          3, 28, 15, 6, 21, 10,
                                                          23, 19, 12, 4, 26, 8,
                                                          16, 7, 27, 20, 13, 2,
                                                          41, 52, 31, 37, 47, 55,
                                                          30, 40, 51, 45, 33, 48,
                                                          44, 49, 39, 56, 34, 53,
```

```
                                      46, 42,
50, 36, 29, 32]]
        )
        round_keys.append(permuted_key)
    return round_keys


# Playfair cipher (simplified for demonstration)
def playfair_cipher(bits):
    table = [
        'l', 'g', 'd', 'b', 'a',
        'q', 'm', 'h', 'e', 'c',
        'u', 'r', 'n', 'i', 'f',
        'x', 'v', 's', 'o', 'k',
        'z', 'y', 'w', 't', 'p',
        '!', '#', '%', '@', '&', '?'
    ]

    # Mapping bits to letters and symbols
    binary_to_char = {f'{i:05b}': char for i, char in enumerate(table)}

    # Divide bits into 5-bit chunks
    chunks = [bits[i:i + 5] for i in range(0, len(bits), 5)]

    # Convert to characters, handle any unknown chunks
    chars = ''.join([binary_to_char.get(chunk, '00000') for chunk in chunks[:-1]]) # if unknown chunk, replace with '00000'
    last_bits = chunks[-1][-2:] if chunks else ''  # Handle case if chunks is empty

    # Convert chars back to binary
    binary_output = ''.join(format(ord(char), '08b') for char in chars)

    # Return the binary string with the last bits appended
    return binary_output + last_bits


# Vigenère cipher
def vigenere_cipher(bits, key):
    key_bin = string_to_binary(key).zfill(len(bits))
    return xor(bits, key_bin[:len(bits)])  # Simple XOR with key


# Additive cipher (mod 32 for simplicity)
def additive_cipher(bits):
    # Convert bits to integers
    numbers = [int(bits[i:i + 5], 2) for i in range(0, len(bits), 5)]
    # Add 1 to each number (simple additive operation)
    numbers = [(num + 1) % 32 for num in numbers]
    # Convert back to 5-bit binary
    return ''.join(format(num, '05b') for num in numbers)


# Keyless cipher (XOR with a predefined pattern)
def keyless_cipher(bits, pattern='1010101010101010101010101010101010'):
```

```
    # XOR with a predefined pattern
    return xor(bits, pattern[:len(bits)])


# Encryption function
def encrypt(plain_text, key):
    # Convert plaintext to binary and ensure it's padded to 64 bits
    plain_text_bin = pad_binary(string_to_binary(plain_text), 64)

    # Apply initial permutation
    permuted_text = initial_permutation(plain_text_bin)

    # Split the text into two halves
    left_half, right_half = permuted_text[:32], permuted_text[32:]

    # Generate round keys
    round_keys = generate_round_keys(key)

    # Perform 8 rounds of encryption
    for i in range(8):
        # Expand the right half
        expanded_right = expand_right_half(right_half)

        # XOR with round key
        xored = xor(expanded_right, round_keys[i])

        # Apply S-Box transformation
        sbox_output = s_box(xored)

        # Apply Playfair or Vigenère cipher depending on round
        if i < 4:
            sbox_output = playfair_cipher(sbox_output)
        else:
            sbox_output = vigenere_cipher(sbox_output, 'PASCAL')

        # Permute the output (simplified)
        sbox_output = pad_binary(sbox_output, 32)  # Ensure it is 32 bits

        # Additive cipher in odd rounds, keyless cipher in even rounds
        if i % 2 == 0:
            sbox_output = keyless_cipher(sbox_output)
        else:
            sbox_output = additive_cipher(sbox_output)

        # XOR with left half
        new_right = xor(left_half, sbox_output)

        # Swap halves
        left_half, right_half = right_half, new_right

    # Combine and apply final permutation
    combined = left_half + right_half
```

```
    final_output = final_permutation(combined)


    return final_output
# Decryption function (needs to reverse the encryption process)
def decrypt(cipher_text, key):
    # Apply the same steps in reverse order
    cipher_text_bin = pad_binary(cipher_text, 64)
    # Apply initial permutation
    permuted_text = initial_permutation(cipher_text_bin)
    # Split the text into two halves
    left_half, right_half = permuted_text[:32], permuted_text[32:]
    # Generate round keys
    round_keys = generate_round_keys(key)
    # Perform 8 rounds of decryption (using reverse round keys)
    for i in range(7, -1, -1):
        # Expand the right half
        expanded_right = expand_right_half(right_half)
        # XOR with round key
        xored = xor(expanded_right, round_keys[i])


        # Apply S-Box transformation
        sbox_output = s_box(xored)


        # Apply Playfair or Vigenère cipher depending on round
        if i < 4:
            sbox_output = playfair_cipher(sbox_output)
        else:
            sbox_output = vigenere_cipher(sbox_output, 'PASCAL')
        # Permute the output (simplified)
        sbox_output = pad_binary(sbox_output, 32)  # Ensure it is 32 bits
        # Additive cipher in odd rounds, keyless cipher in even rounds
        if i % 2 == 0:
            sbox_output = keyless_cipher(sbox_output)
        else:
            sbox_output = additive_cipher(sbox_output)
        # XOR with left half
        new_left = xor(left_half, sbox_output)
        # Swap halves
        left_half, right_half = right_half, new_left
    # Combine and apply final permutation
    combined = left_half + right_half
    final_output = final_permutation(combined)
    return final_output
# Convert binary to hexadecimal for output
def binary_to_hex(binary):
    return hex(int(binary, 2))[2:]


# Example of usage
if _name_ == "_main_":
    key = "YOUR_SECRET_KEY"  # Your fixed key for the encryption
```

```
    operation = input("Enter 'e' to encrypt or 'd' to decrypt: ")
    if operation.lower() == 'e':
        plaintext = input("Enter 64-bit plaintext (8 characters): ")
        ciphertext = encrypt(plaintext, key)
        print("Ciphertext (hex):", binary_to_hex(ciphertext))
    elif operation.lower() == 'd':
        ciphertext = input("Enter ciphertext (hex): ")
        plaintext = decrypt(hex_to_binary(ciphertext), key)
        print("Decrypted plaintext:", binary_to_string(plaintext))
```

READ.ME FILE:

```
*READ.ME - Notepad
File Edit Format View Help
Overview
This custom encryption algorithm is designed for secure encryption of 64-bit plaintext using a modified DES structure with integrated traditional ciphers.
 The key highlights of the algorithm are:

128-bit key is used and broken into 64-bit halves, then compressed and permuted, similar to DES.
The plaintext undergoes initial permutation like in DES, followed by 8 encryption rounds.
Each round applies circular shifts, expansions, permutations, and XOR operations, incorporating both:
Playfair Cipher (Rounds 1-4).
Vigenère Cipher (Rounds 5-8).
Additive and keyless ciphers are applied in alternate rounds (1, 3, 5, 7 and 2, 4, 6, 8, respectively).
By blending DES-style encryption with traditional ciphers, this design enhances complexity and security.

Instructions for Running the Code:
1. Ensure you have Python installed on your system.
2. Place the provided script in your project directory.

3. You will be prompted to enter the 64-bit plaintext and the 128-bit key in binary form.
4. The program will encrypt the plaintext and display the cipher text output.
Input Format:
Plaintext: Enter a 64-bit binary string
Key: Enter a 128-bit binary string .
Output:
The program will output the encrypted ciphertext in binary format after all 8 rounds.
Example
Example Input:
Plaintext: 11001010011100011010101010101010
Key: 1110101010101110110101010101110110110101001010101010010110110111
Example Output:
Cipher Text: 10110110111010100101100101100101
```

JUSTIFICATION OF STRENGTHS:

Theoretical Foundations of the Design:

This encryption algorithm is based on the DES structure but incorporates unique additions like the Playfair and Vigenere ciphers, enhancing diffusion and confusion. The use of key shifting, permutation, and substitution aligns with the principles of Feistel networks, ensuring both diffusion (mixing of plaintext) and confusion (complexity between ciphertext and key).

Unique Features and Innovations:

One key innovation is the use of symbols, extending the mapping from 25 to 32 characters, which significantly increases complexity. This added symbol set enhances the cipher's strength by expanding the key space and making frequency analysis more difficult. The integration of traditional ciphers (Playfair and Vigenere) within the DES structure further diversifies the encryption process. The variation of ciphers across rounds (Playfair in the first four rounds, Vigenere in the last four) adds unpredictability. Additionally, circular shifting and alternating additive/keyless ciphers bolster security by introducing variability to key manipulation.

Defense Against Common Attacks:

The algorithm defends against differential and linear cryptanalysis by disrupting predictable patterns with the use of multiple encryption methods and symbols. The extended character set, along with the complex round structure, increases resistance to statistical analysis. With multiple rounds, key permutations, and the inclusion of non-standard S-Box operations, attackers are faced with a far more challenging environment for exploiting potential weaknesses.

TEST CASE:

```
Run    Share    Command Line Arguments

Enter 'e' to encrypt or 'd' to decrypt:
e
Enter the text (plain text for encryption, cipher text for decryption):
whispers
Enter the 128-bit key (as a string of 0s and 1s):
0110101001110101101001011110111101101010011101011010010111101111011010100111010110100101111011110110101001
1101011010010111101111
Cipher text (hex): fc260b10d95bfb91
```

We encrypted the word "whispers" using our algorithm. The manually done encryption by our team matched with the cipher text provided by the code. This showed us that encryption is successful and code is working properly.

Then we decrypted the same ciphertext using our algorithm. The ciphertext was entered and it gave us the same word that we encrypted.

Run
Share
Command Line Arguments

```
Enter 'e' to encrypt or 'd' to decrypt:
d
Enter the text (plain text for encryption, cipher text for decryption):
fc260b10d95bfb91
Enter the 128-bit key (as a string of 0s and 1s):
0110101001110101101001011110111101101010011101011010010111101111011010100111010110100101111011110110
1010011101011010010111101111
Plain text: whispers



** Process exited - Return Code: 0 **
Press Enter to exit terminal
```

The end!