[redacted]

1. The Flow of DES

1.1 The simple description of DES

DES is a kind of ***Symmetric key encryption algorithm***, which means use the same symmetric key to encrypt and decrypt one plaintext.

The encryption process of DES first converts the initial data into binary data, and each 64-bit segment is divided into a group. For the missing parts, '0' is padded or filled using the PKCS7 method. When decrypting, the last padded content needs to be removed. Each group of data first undergoes an IP (Initial Permutation) to get the reset binary data stream, which is then split into two parts, L and R. Next, 16 rounds of data swapping and F-function operations are performed. Data swapping means that the R from the previous round becomes the L for the next round, and the L from the previous round is XORed with the R after the F-function operation to get new data. After 16 rounds of operations, the positions of LR are swapped once more before concatenation. This is done to ensure that the decryption process is correctly reversible. After concatenation, the result undergoes an IP inverse permutation, yielding the encrypted result for the current group.

The F operation in DES involves several steps:

1. **Expansion**: The 32-bit R data is first expanded using the E-box (expansion box) to 48 bits.
2. **XOR with Subkey**: The expanded data is then XORed with a subkey. The original key is processed to generate 16 subkeys, and each round uses the corresponding subkey.
3. **S-boxes**: The XORed result is passed through eight S-boxes to reduce the bit length. The 48-bit data is divided into eight 6-bit segments. The first and last bits of each segment determine the row number of the S-box, while the middle four bits determine the column number. The data at the corresponding position in the S-box, converted to binary, becomes the result for that segment. Each segment is thus reduced from 6 bits to 4 bits, shrinking the entire data from 48 bits back to 32 bits.
4. **P-Permutation**: The output from the S-boxes is then permuted using the P-box (permutation box) to restore the original length.
5. **Final XOR**: The permuted result is XORed with the L data, and this result becomes the R for the next round.

1.2 The processing of Key

In DES, you can theoretically use any data as a key, but the key is standardized to a 64-bit binary format. The process to generate the subkeys involves several steps:

1. **Key Preparation**: The original key is first processed to conform to a 64-bit length.
2. **PC1 Permutation**: The 64-bit key is then permuted using the PC1 box to reduce it to 56 bits.
3. **Key Splitting and Subkey Generation**: The 56-bit key is divided into two parts, C and D. These parts undergo a series of operations to generate 16 subkeys. Initially, C and D are subjected to a series of circular left shifts according to a specific shift schedule, resulting in new C and D values. These are then concatenated and permuted using the PC2 box to produce one subkey. This process is repeated 16 times.
4. **Subkey Correspondence**: The resulting 16 subkeys correspond to the 16 rounds of the F operation in the encryption process.

For decryption, the same subkeys are used in reverse order to decrypt the ciphertext.

2. Code

2.1 the management of conversion of number systems

```
'''
use Base64 to adapt chinese codes
'''
def str2bin(s):
    base64_bytes = base64.b64encode(s.encode('utf-8'))
    return ''.join(format(byte, '08b') for byte in base64_bytes)
def bin2str(b):
    bytes_list = [b[i:i + 8] for i in range(0, len(b), 8)]
```

```
    bytes_obj = bytes([int(byte, 2) for byte in bytes_list])
    try:
        return base64.b64decode(bytes_obj).decode('utf-8')
    except (UnicodeDecodeError, base64.binascii.Error) as e:
        print(f"Failing: {e}")  # this will reflect the wrong key condition
        return
def byte2bin(byte_str):
    return "".join(format(byte, '08b') for byte in byte_str)
```

2.2 the management of key

```
'''
use pbkdf2_hmac function to get the 8 bytes key, then we can get 64 bits key
'''
def process_key(k):
    k_hex = pbkdf2_hmac(
        'sha256',
        k.encode('utf-8'),
        b'salt',
        10000,
        dklen=8
    )
    k_str = byte2bin(k_hex)
    key_bin56 = ""
    for i in PC1_TABLE:
        key_bin56 += k_str[i - 1]
    return key_bin56


# left moving function
def left_turn(my_str, num):
    return my_str[num:] + my_str[:num]


# get all 16 sub-keys
def generate_keys(k):
    keys = []
    key_bin56 = process_key(k)
    c = key_bin56[0:28]
    d = key_bin56[28:]
    for i in MOVE_TABLE:
        c = left_turn(c, i)
        d = left_turn(d, i)
        total_k = c + d
        sub_key = ""
        for j in PC2_TABLE:
            sub_key += total_k[j - 1]
        keys.append(sub_key)
    return keys
```

2.3 the management in DES

```
'''
use '0' padding
'''
def divide(bin_str):
    length = len(bin_str)
    if length % 64 != 0:
        bin_str += "0" * (64 - (length % 64))
    # Divide it into multiple 64-bit segments. The total number of segments is equal to the length of the string, taking 64 bits
each         time.
```

```
    result = [bin_str[i: i + 64] for i in range(0, len(bin_str), 64)]
    return result
'''
the Permutation operation
you can use different Table as 'table'. then you will get the permutated result
'''
def trans(str_bit, table):
    result = ""
    for i in table:
        result += str_bit[i - 1]
    return result
'''
the XORed operation
perform XOR operations and make judgments in sequence.
'''
def xor(str1, str2):
    result = ""
    for c1, c2 in zip(str1, str2):
        result += str(int(c1) ^ int(c2))
    return result
'''
the S operation
'''
# the single apply in one S table. get row and col, then get binary data
def single_s(str_bit, i):
    row = int(str_bit[0] + str_bit[5], 2)
    col = int(str_bit[1:5], 2)
    num = S_BOX[i][row][col]
    return bin(num)[2:].zfill(4)
# whe whole S operations
def s_box(str_bit):
    result = ""
    for i in range(8):
        result += single_s(str_bit[i * 6: i * 6 + 6], i)
    return result
# the whole F function
def F(str_bit, k):
    # E expansion
    str_bit = trans(str_bit, E_TABLE)
    # use sub-key
    str_bit = xor(str_bit, k)
    # all S
    str_bit = s_box(str_bit)
    # P permutation
    str_bit = trans(str_bit, P_TABLE)
    return str_bit
```

2.4 the management of encryption and decryption

```
def encrypt(origin_str, k):
    keys = generate_keys(k)
    bin_str = str2bin(origin_str)
    str_list = divide(bin_str)
    result = ""
    for i in str_list:
        i = trans(i, IP_TABLE)
        L, R = i[0:32], i[32:]
        for j in range(16):
            L, R = R, xor(L, F(R, keys[j]))
        # do not forget the trans between R and L
        i = trans(R + L, IP2_TABLE)
        result += i
    return result
# the same process of encrypt, one important thing is reverse the sub-keys sequence
# another important thing is remove the '0' padding
```

```
def decrypt(origin_str, k):
    keys = generate_keys(k)
    str_list = divide(origin_str)
    result = ""
    for i in str_list:
        i = trans(i, IP_TABLE)
        L, R = i[0:32], i[32:]
        for j in range(16):
            L, R = R, xor(L, F(R, keys[15 - j]))
        i = trans(R + L, IP2_TABLE)
        result += i
    result = result.rstrip('0')
    return bin2str(result)
```

3. Challenge

3.1 about number system

Initially, I considered directly converting the string to binary data, but this approach would truncate Chinese characters, not only failing to obtain the desired data but also causing errors. Then, I tried using hexadecimal as an intermediary for the conversion, but that still presented some unresolved issues. In the end, I decided to use Base64 encoding as the basis for the transcoding, which proved to be very effective. With Base64 encoding, I can ensure the integrity and accuracy of the data, avoiding the problems I encountered before.

3.2 about key

Initially, my approach was simple: convert the key to binary, and if the length was less than 64 bits, pad it with zeros; otherwise, truncate it to the first 64 bits. However, this resulted in only a small part of the key being effective, and if an attacker guessed the initial part correctly, the rest would not affect the key, making it insecure and impractical.

I then explored three solutions, ultimately choosing the third.

The first solution was to append a verification string to the plaintext. If the decrypted result lacked this string, it would signal an incorrect key. However, this method risked exposing the verification string through multiple encryption attempts.

```
VALIDATION_STRING = "VALID_KEY"
origin_str = VALIDATION_STRING + origin_str
if plaintext.startswith(VALIDATION_STRING):
    return plaintext[len(VALIDATION_STRING):]
else:
    return
```

The second solution involved hashing the key data and embedding it within the ciphertext. Upon decryption, the key data would be extracted, and its hash checked to confirm the key's correctness. Yet, this approach also compromised the key's security.

```
key_hash = hashlib.sha256(k.encode()).hexdigest()
###
ciphertext = origin_str[:-64]
key_hash = origin_str[-64:]
hashlib.sha256(k.encode()).hexdigest() != bin2hex(key_hash):
    return
```

In the end, I opted for the `pbkdf2_hmac` method, which includes 10,000 iterations of hashing with a salt. This method not only guarantees a highly secure key but also enables the verification of the key's correctness.