Hybrid Block Encryption Project

MADR-CS 281 Intro. to Cryptography and Computer Security (SP 2025)

Midterm Project by Hanna Chang

This document describes and displays code for a hybrid block encryption project. The document consists of explaining to:

- Generate a 256-bit key from a user's email address via SHA-256 and split it into eight 32-bit segments.
- Apply an IP to a 256-bit plaintext block using a permutation table generated from that user's email.
- Process each 32-bit block by expanding it to 48 bits (using a DES expansion table), substituting bits with S-box operations, and XORing with the corresponding key segment.
- Simulate decryption by "reversing" these steps with theoretical placeholder functions (because in real DES decryption, the round keys would be applied in reverse order and the operations would be inherently reversible).

> **Note:** The reverse functions for the expansion and S-box steps in this demo are placeholders due to time constraints. Because the DES expansion and S-box functions are many-to-one, their inversion is not exact. This project is meant for learning and demonstration purposes.

Project Files

1. key_generation.py

This file generates a 256-bit key by computing the SHA-256 hash of the user's email address and splits the resulting 64-character hexadecimal string into 8 segments of 8 hex digits each (each representing 32 bits).

```
# key_generation.py
import hashlib

def generate_key(email: str):
    hash_obj = hashlib.sha256(email.encode())
    hex_key = hash_obj.hexdigest()  # 64 hex digits = 256 bits
    key_segments = [hex_key[i:i+8] for i in range(0, 64, 8)]
    return key_segments

if __name__ == "__main__":
    email = input("Enter your email for key generation: ")
    key_segments = generate_key(email)
    print("Generated Key Segments:", key_segments)
```

2. permutation.py

This part creates a unique permutation table for a 256-bit block using the user's email address as a seed. The table is used for the IP and can be inverted to recover the original order during decryption.

```
# permutation.py
import random

def generate_user_permutation(email: str, size=256):
    seed = sum(ord(c) for c in email)
    random.seed(seed)
    table = list(range(size))
    random.shuffle(table)
    return table

def apply_permutation(bit_string: str, table: list) -> str:
    if len(bit_string) != len(table):
        raise ValueError("Bit string length must equal permutation table
size.")
    return ''.join(bit_string[i] for i in table)

def inverse_permutation_table(table: list) -> list:
    inv = [0] * len(table)
    for i, pos in enumerate(table):
        inv[pos] = i
    return inv
```

3. tables.py

This code defines the DES expansion table and eight S-boxes. The expansion function changes the 32-bit binary string into a 48-bit string. The S-box substitution then reduces that 48-bit string back to 32 bits by processing eight 6-bit blocks. Additionally, placeholder reverse functions are included for demonstration purposes.

```
# tables.py

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
]

S_BOXES = [
    [   # 1
        [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]
        ],
        [   # 2
            [15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
            [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
            [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
            [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]
        ],
        [   # 3
            [10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
            [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
            [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
            [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]
        ],
        [   # 4
            [7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
            [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
            [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
            [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]
        ],
        [   # 5
            [2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
            [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
            [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
            [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]
        ],
        [   # 6
            [12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
            [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
            [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
            [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]
        ],
        [   # 7
            [4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
            [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
            [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
            [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]
        ],
        [   # 8
            [13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
            [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
            [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
            [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]
        ]
    ]

    def expansion(input_32bit: str) -> str:
        result = ""
        for pos in expansion_table:
            result += input_32bit[pos - 1]  # table is 1-indexed
        return result

    def s_box_substitution(expanded: str) -> str:
        output = ""
```

```
    for i in range(8):
        block = expanded[i*6:(i+1)*6]
        row = int(block[0] + block[5], 2)
        col = int(block[1:5], 2)
        s_val = S_BOXES[i][row][col]
        output += format(s_val, '04b')
    return output

# The following reverse functions are placeholders to show structure.
# In proper DES decryption, the same S-boxes would be used with reversed
round keys.
# DES S-boxes are many-to-one; in proper DES decryption, the process is
different.
def reverse_s_box_substitution(substituted: str) -> str:
    return "000000" * 8

# Placeholder reverse for the expansion function.
# Since expansion loses information, this function simply returns the
first 32 bits.
def reverse_expansion(expanded: str) -> str:
    return expanded[:32]
```

4. encryption.py

This module contains the encryption process. It:

- Converts a 64-digit hex plaintext (256 bits) into a binary string.
- Applies an IP using the user-generated permutation table.
- Splits the permuted block into eight 32-bit segments.
- For each segment:
  - Expands it to 48 bits.
  - Applies the DES-style S-box substitution (reducing to 32 bits).
  - XORs with the corresponding key segment.
- Reassembles the encrypted segments into a 256-bit encrypted block.

```
# encryption.py
from tables import expansion, s_box_substitution
from permutation import apply_permutation, generate_user_permutation

def hex_to_bin(hex_str: str, bits: int) -> str:
    return format(int(hex_str, 16), '0{}b'.format(bits))

def xor_bin(bin_str1: str, bin_str2: str) -> str:
    return ''.join('1' if a != b else '0' for a, b in zip(bin_str1,
bin_str2))

def encrypt_message(plaintext_hex: str, key_segments: list,
permutation_table: list) -> str:
    if len(plaintext_hex) != 64:
        raise ValueError("Plaintext must be 64 hex digits (256 bits).")
```

```
    plaintext_bin = hex_to_bin(plaintext_hex, 256)

    # Apply IP
    permuted_plaintext = apply_permutation(plaintext_bin,
permutation_table)

    encrypted_block = ""
    for i in range(8):
        segment = permuted_plaintext[i*32:(i+1)*32]
        expanded = expansion(segment)                    # 32 -> 48 bits
        substituted = s_box_substitution(expanded)       # 48 -> 32 bits
        key_bin = hex_to_bin(key_segments[i], 32)        # Convert key
segment to 32-bit binary
        encrypted_segment = xor_bin(substituted, key_bin) # XOR with key
segment
        encrypted_block += encrypted_segment

    return encrypted_block

if __name__ == "__main__":
    plaintext_hex = input("Enter 64 hex digits (256-bit) plaintext:
").strip()
    key_segments = []
    for i in range(8):
        seg = input(f"Enter key segment {i+1} (8 hex digits): ").strip()
        if len(seg) != 8:
            raise ValueError("Key segment must be 8 hex digits.")
        key_segments.append(seg)
    email = input("Enter your email (for permutation generation):
").strip()
    perm_table = generate_user_permutation(email, size=256)
    encrypted_bin = encrypt_message(plaintext_hex, key_segments,
perm_table)
    print("Encrypted block (binary):", encrypted_bin)
    encrypted_hex = format(int(encrypted_bin, 2), '064x')
    print("Encrypted block (hex):", encrypted_hex)
```

5. decryption.py

This part is for the decryption process. It splits a 256-bit encrypted block into eight segments, XORs each segment with the corresponding key segment, applies the placeholder reverse S-box and expansion functions, and finally applies the inverse of the initial permutation to recover the plaintext. Because the reverse functions are placeholders, the output is not the original plaintext.

```
# decryption.py
from tables import reverse_s_box_substitution, reverse_expansion
from permutation import apply_permutation, generate_user_permutation,
inverse_permutation_table

def xor_bin(bin_str1: str, bin_str2: str) -> str:
    return ''.join('1' if a != b else '0' for a, b in zip(bin_str1,
```

```
bin_str2))

def hex_to_bin(hex_str: str, bits: int) -> str:
    return format(int(hex_str, 16), '0{}b'.format(bits))

def decrypt_message(encrypted_hex: str, key_segments: list,
permutation_table: list) -> str:
    if len(encrypted_hex) != 64:
        raise ValueError("Encrypted block must be 64 hex digits (256
bits).")
    encrypted_bin = hex_to_bin(encrypted_hex, 256)

    decrypted_block = ""
    for i in range(8):
        encrypted_segment = encrypted_bin[i*32:(i+1)*32]
        key_bin = hex_to_bin(key_segments[i], 32)
        substituted = xor_bin(encrypted_segment, key_bin)
        expanded = reverse_s_box_substitution(substituted)  # Placeholder
reverse substitution
        original_segment = reverse_expansion(expanded)      # Placeholder
reverse expansion
        decrypted_block += original_segment

    # Apply inverse IP
    inv_perm_table = inverse_permutation_table(permutation_table)
    recovered_plaintext = apply_permutation(decrypted_block,
inv_perm_table)
    return recovered_plaintext

if __name__ == "__main__":
    encrypted_hex = input("Enter encrypted block (64 hex digits):
").strip()
    key_segments = []
    for i in range(8):
        seg = input(f"Enter key segment {i+1} (8 hex digits): ").strip()
        if len(seg) != 8:
            raise ValueError("Key segment must be 8 hex digits.")
        key_segments.append(seg)
    email = input("Enter your email (for permutation generation):
").strip()
    perm_table = generate_user_permutation(email, size=256)
    decrypted_bin = decrypt_message(encrypted_hex, key_segments,
perm_table)
    print("Decrypted block (binary):", decrypted_bin)
    decrypted_hex = format(int(decrypted_bin, 2), '064x')
    print("Decrypted block (hex):", decrypted_hex)
```

6. main.py

This is the main script with everything together. It prompts the user to select encryption or decryption mode, generates the key segments and permutation table based on the user's email, and calls the appropriate functions.

```
# main.py
import key_generation
import encryption
import decryption
from permutation import generate_user_permutation

def main():
    mode = input("Select mode: E for encryption, D for decryption:
").strip().upper()
    email = input("Enter your email (for key generation and permutation):
").strip()
    key_segments = key_generation.generate_key(email)
    print("Generated Key Segments:", key_segments)

    # Generate a unique permutation table based on the email
    perm_table = generate_user_permutation(email, size=256)

    if mode == 'E':
        plaintext_hex = input("Enter plaintext as 64 hex digits (256
bits): ").strip()
        encrypted_bin = encryption.encrypt_message(plaintext_hex,
key_segments, perm_table)
        encrypted_hex = format(int(encrypted_bin, 2), '064x')
        print("Encrypted block (hex):", encrypted_hex)
    elif mode == 'D':
        encrypted_hex = input("Enter encrypted block as 64 hex digits:
").strip()
        decrypted_bin = decryption.decrypt_message(encrypted_hex,
key_segments, perm_table)
        decrypted_hex = format(int(decrypted_bin, 2), '064x')
        print("Decrypted block (hex):", decrypted_hex)
    else:
        print("Invalid mode selected.")

if __name__ == "__main__":
    main()
```

Usage Instructions

1. Key Generation:

- Run `key_generation.py` to generate key segments based on your email address. The key is a 256-bit value split into eight 32-bit segments.

> **Example Terminal Output for `key_generation.py`**
>
> Enter your email for key generation:
> ***hchang@macalester.edu***
> Generated Key Segments:
> ['7cf39410', '455894aa', '66763e4a', '0a70ab14', '0d248291', 'fbc1e590', '83fa7097', 'a80d2d43']`

2. Encryption:

- Run `main.py` and choose mode **E** for encryption.
- Enter your email (used both for key generation and for creating a unique permutation table).
- Provide a 256-bit plaintext as 64 hex digits (each hex digit is 4 bits). For example: 0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
- The program outputs the encrypted block in hexadecimal.

> **Example Terminal Output for `main.py`**
>
> Select mode: E for encryption, D for decryption:
> ***E***
> Enter your email (for key generation and permutation):
> ***hchang@macalester.edu***
> Generated Key Segments:
> ['7cf39410', '455894aa', '66763e4a', '0a70ab14', '0d248291', 'fbc1e590', '83fa7097', 'a80d2d43']
> Enter plaintext as 64 hex digits (256 bits):
> ***0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef***
> Encrypted block (hex):
> 89f018e10716891cf711f312439ea4652ee98d5884891f137a3cc292c9a0bc35

3. Decryption:

- Run `main.py` and choose mode **D** for decryption. -Enter the same email used during encryption. -Provide the encrypted block (64 hex digits) to attempt decryption. -Because the reverse functions are placeholders, the decryption will not recover the original plaintext exactly.

> **Example Terminal Output for `main.py`**
>
> Select mode: E for encryption, D for decryption:
> ***D***
> Enter your email (for key generation and permutation):
> ***hchang@macalester.edu***
> Generated Key Segments:
> ['7cf39410', '455894aa', '66763e4a', '0a70ab14', '0d248291', 'fbc1e590', '83fa7097', 'a80d2d43']
> Enter encrypted block as 64 hex digits:
> ***89f018e10716891cf711f312439ea4652ee98d5884891f137a3cc292c9a0bc35***
> Decrypted block (hex):
> 0000000000000000000000000000000000000000000000000000000000000000
