```
/*
Mike Koch
EECS 4760 - Computer Security
DES
error_handler.h: Consult error_handler.cpp
*/
#ifndef ERROR_HANDLER_H
#define ERROR_HANDLER_H

void output_error(char *text, int exit_code);
#endif
```

```c
/*
Mike Koch
EECS 4760 - Computer Security
DES
exit_code.h: An enum indicating the type of error that occurred, along with its exit code.
*/
#ifndef EXIT_CODE_H
#define EXIT_CODE_H

// A collection of exit codes, depending on what went wrong
enum ExitCode {
    INVALID_ARG_SYNTAX = 1,
    INVALID_ENCRYPT_DECRYPT_FLAG = 2,
    INVALID_PASSWORD = 3,
    INVALID_MODE = 4,
    CANNOT_OPEN_FILE = 5,

};
#endif
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
file_processor.h: Consult file_processor.cpp
*/
#ifndef FILE_PROCESSOR_H
#define FILE_PROCESSOR_H
#include <fstream>

bool get_next_64_bits(std::fstream &file_stream, uint64_t *destination, int *bytes_read, int file_size);
void build_file_size_block(int file_size, uint64_t *block);

#endif
```

```
/*
Mike Koch
EECS 4760 - Computer Security
DES
garbage_producer.h: Consult garbage_producer.cpp
*/
#ifndef GARBAGE_PRODUCER_H
#define GARBAGE_PRODUCER_H
#include <stdint.h>

void generate_eight_bytes_of_garbage(uint64_t *destination);
#endif
```

```
/*
Mike Koch
EECS 4760 - Computer Security
DES
keygen.h: Consult keygen.cpp
*/
#ifndef KEYGEN_H
#define KEYGEN_H
#include <stdint.h>

// Generates the necessary keys needed and stores them in the key array provided. This also assumes that the provided_key is a valid key.
void generate_keys(uint64_t provided_key, uint64_t *keys);
#endif
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
mode.h: An enum to indicate whether or not to encrypt (Mode::ENCRYPTION) or to decrypt (Mode::DECRYPTION) the infile.
*/
#ifndef MODE_H
#define MODE_H

enum Mode {
    ENCRYPTION,
    DECRYPTION,
};
#endif
```

```
/*
Mike Koch
EECS 4760 - Computer Security
DES
permutator.h: Consult permutator.cpp
*/
#ifndef PERMUTATOR_H
#define PERMUTATOR_H
#include <stdint.h>

void apply_initial_permutation(uint64_t *source, uint64_t *destination);

void apply_expansion_permutation(uint32_t *source, uint64_t *destination);

void apply_final_permutation(uint64_t *source, uint64_t *destination);
#endif
```

```c
/*
Mike Koch
EECS 4760 - Computer Security
DES
reverser.h: Consult reverser.cpp
*/
#ifndef REVERSER_H
#define REVERSER_H
#include <stdint.h>

void reverse(uint64_t *source, uint64_t *destination);
#endif
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
rounds.h: Consult rounds.cpp
*/
#ifndef ROUNDS_H
#define ROUNDS_H
#include <fstream>
#include "mode.h"

void apply_rounds(uint64_t *initial_permutation, uint64_t *output, uint64_t *keys, Mode mode);
#endif
```

```
/*
Mike Koch
EECS 4760 - Computer Security
DES
s_box.h: The 8 s-boxes used during the Feistel rounds. Using a 2D array is cleaner than a bunch of if-else blocks, and may actually be a little bit faster (two memory accesses instead of several
         CMPs and jumps)
*/
#ifndef SBOX_H
#define SBOX_H

// The actual S-Boxes for the rounds live here. It is faster to have them declared once, rather than having to create them each
// time a round is ran.
const int S_BOX_ONE[4][16] = {
    { 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 }
};

const int S_BOX_TWO[4][16] = {
    { 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 }
};

const int S_BOX_THREE[4][16] = {
    { 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 }
};

const int S_BOX_FOUR[4][16] = {
    { 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 }
};

const int S_BOX_FIVE[4][16] = {
    { 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 }
};

const int S_BOX_SIX[4][16] = {
    { 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 }
};

const int S_BOX_SEVEN[4][16] = {
    { 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 }
};

const int S_BOX_EIGHT[4][16] = {
    { 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 }
};
#endif
```

```cpp
/*
 Mike Koch
 EECS 4760 - Computer Security
 DES
 des.cpp: Main entry point. Handles argument processing and the overall encryption / decryption process.
*/
#include "keygen.h"
#include "file_processor.h"
#include "permutator.h"
#include "rounds.h"
#include "reverser.h"
#include "garbage_producer.h"
#include "error_handler.h"
#include "exit_code.h"
#include <string.h>
#include <time.h>
#include <iostream>

// Constants
const int ASCII_ZERO = 48;
const int ASCII_NINE = 57;
const int ASCII_A = 65;
const int ASCII_F = 70;
const double NUMBER_OF_MILLISECONDS_IN_SECOND = 1000.0;

// Function prototypes
void process_chunk(uint64_t *next_64_bits, uint64_t *keys, uint64_t *output, Mode mode);
void set_ascii_characters_to_key(char most_significant_char, char least_significant_char, int bit_offset, uint64_t *destination);

int main(int argc, char *argv[])
{
    // 1. Process args
    // The command-line syntax is: [-[d|D]|-[e|E]] [password] [ECB|ecb] [input file path] [output file path], where -d is decryption and -e is encryption
    // If we don't have 5 args (program name + actual args == argc), fail now
    if (argc != 6) {
        output_error("", ExitCode::INVALID_ARG_SYNTAX);
    }

    // Make sure the user provided a valid encrypt/decrypt flag
    Mode mode;
    if (std::string(argv[1]) == "-e" || std::string(argv[1]) == "-E") {
        mode = Mode::ENCRYPTION;
    }
    else if (std::string(argv[1]) == "-d" || std::string(argv[1]) == "-D") {
        mode = Mode::DECRYPTION;
    }
    else {
        output_error("Invalid encryption/decryption flag. Must choose either '-
            e'/'-E' for encrypt or '-d'/'-D' for decrypt", ExitCode::INVALID_ENCRYPT_DECRYPT_FLAG);
    }

    // Key parsing
    char *raw_password = argv[2];
    uint64_t password = 0;

    // Parse the password to see if it's a hex value or a string literal. If the key starts with ', it's a string literal
    if (raw_password[0] == '\'') {
        // The password is a string literal. Make sure the password is the proper length (8 chars + 2 single ticks = 10 chars). If it's not, fail.
        if (strlen(raw_password) != 10) {
            output_error("String passwords must be *exactly* eight characters surrounded by single tick marks ('). If your key includes spaces, wrap the entire string in double-quotes (\"). Exiting.", ExitCode::INVALID_PASSWORD);
        }

        // The password was entered in as a string. Set each byte to the password uint
        password |= (uint64_t)raw_password[1] << 56;
        password |= (uint64_t)raw_password[2] << 48;
        password |= (uint64_t)raw_password[3] << 40;
        password |= (uint64_t)raw_password[4] << 32;
        password |= (uint64_t)raw_password[5] << 24;
        password |= (uint64_t)raw_password[6] << 16;
        password |= (uint64_t)raw_password[7] << 8;
        password |= (uint64_t)raw_password[8];
    }
    else {
        // The password was entered as hex. We'll need to look at each character and convert it to its hex value
        // Make sure we have 16 hex characters. If we don't, fail.
        if (strlen(raw_password) != 16) {
            output_error("Hex passwords must be *exactly* sixteen hex characters between 0-F. Exiting.", ExitCode::INVALID_PASSWORD);
        }


        // The hex characters provided are entered in order from MSB to LSB
        set_ascii_characters_to_key(raw_password[0], raw_password[1], 56, &password);
        set_ascii_characters_to_key(raw_password[2], raw_password[3], 48, &password);
        set_ascii_characters_to_key(raw_password[4], raw_password[5], 40, &password);
        set_ascii_characters_to_key(raw_password[6], raw_password[7], 32, &password);
        set_ascii_characters_to_key(raw_password[8], raw_password[9], 24, &password);
```

```cpp
        set_ascii_characters_to_key(raw_password[10], raw_password[11], 16, &password);
        set_ascii_characters_to_key(raw_password[12], raw_password[13], 8, &password);
        set_ascii_characters_to_key(raw_password[14], raw_password[15], 0, &password);
    }

    // This implementation only supports ECB. If anything else is provided, fail now
    if ((argv[3][0] != 'e' && argv[3][0] != 'E')
        || (argv[3][1] != 'c' && argv[3][1] != 'C')
        || (argv[3][2] != 'b' && argv[3][2] != 'B')) {
        output_error("Only ECB is supported!", ExitCode::INVALID_MODE);
    }

    char *input_file_path = argv[4];
    char *output_file_path = argv[5];

    printf("Starting the %s process. Please wait.\r", mode == Mode::ENCRYPTION ? "encryption" : "decryption");

    // 2. Generate keys
    // Start a timer here so we can see how long the process takes.
    clock_t startTime = clock();
    uint64_t keys[16];
    generate_keys(password, keys);

    // Open the file for processing
    std::fstream input_stream;
    std::fstream output_stream;
    input_stream.open(input_file_path, std::ios::in | std::ios::binary);

    // Get the length of the file by going to the end ofthe input stream and checking where the stream index is at. Then go back
    // to the beginning so we can actually read the data
    input_stream.seekg(0, input_stream.end);
    int length_of_file = input_stream.tellg();
    input_stream.seekg(0, input_stream.beg);

    output_stream.open(output_file_path, std::ios::out | std::ios::binary | std::ios::trunc);
    if (!input_stream.is_open() || !output_stream.is_open()) {
        output_error("ERROR: Unable to open file. Make sure that both the input file and output file are accessible!", ExitCode::CANNOT_OPEN_FILE);
    }

    // If we're encrypting, encrypt the file size.
    if (mode == Mode::ENCRYPTION) {
        uint64_t file_size_block;
        build_file_size_block(length_of_file, &file_size_block);

        uint64_t output = 0;
        process_chunk(&file_size_block, keys, &output, mode);

        // Reverse the output so the bytes are saved from MSB to LSB
        uint64_t reversed_output;
        reverse(&output, &reversed_output);

        output_stream.write((char*)&reversed_output, sizeof(uint64_t));
    }

    // Flag to indicate if we've reached the end of the input file.
    bool keep_going = false;
    int number_of_bytes_read = 0;

    // These two ints are used only for decryption. We only strip half of the first chunk during decryption, so we need to know when the first run is. We also need to know the file size.
    int first_iteration = true;
    int decrypted_file_size;
    do {
        // 3. Get next 64 bits (8 bytes)
        uint64_t next_64_bits = 0;
        int previous_number_of_bytes_read = number_of_bytes_read;
        keep_going = get_next_64_bits(input_stream, &next_64_bits, &number_of_bytes_read, length_of_file);

        // If we fall into this block, this means our last read was (1) our final possible read, and (2) we read less than 8 bytes. We need to pad the rest of the bytes with garbage.
        // This will ONLY ever happen when encrypting. If we're decrypting and this if statement is true, very bad things have happened.
        if (!keep_going && number_of_bytes_read - previous_number_of_bytes_read != 8) {
            if (mode == Mode::DECRYPTION) {
                printf("Hold up there! We shouldn't have a block that is not exactly 64 bits long. Stopping.");
                exit(1);
            }

            uint64_t garbage;
            generate_eight_bytes_of_garbage(&garbage);
            int number_of_bytes_filled = number_of_bytes_read - previous_number_of_bytes_read;

            // Now that we know how many bytes need garbage, push the garbage data into those bytes by shifting them to the left.
            //    *We haven't reversed yet, so the garbage will be the most significant byte(s)*
            next_64_bits |= garbage << (8 * number_of_bytes_filled);
        }

        // The bytes read in are stored in reverse order. We need to flip them
```

```cpp
          around first.
        uint64_t reversed_next_64_bits;
        reverse(&next_64_bits, &reversed_next_64_bits);

        uint64_t output = 0;
        process_chunk(&reversed_next_64_bits, keys, &output, mode);

        if (first_iteration && mode == Mode::DECRYPTION) {
            // The first chunk tells us how long the file is. We'll store this in decrypted_file_size so we know if we need to throw away any garbage bits.
            // ANDing the output with the below hex value to only retrieve the least-significant word; our file length
            decrypted_file_size = output & 0x00000000FFFFFFFF;

            first_iteration = false;
        }
        else {
            // Output to file
            uint64_t reversed_output;
            reverse(&output, &reversed_output);

            // We need to make sure that this isn't the last block during decryption. If we are on the last block, and there's garbage data, we must strip it out first.
            int output_size = sizeof(uint64_t);
            if (mode == Mode::DECRYPTION && decrypted_file_size < (number_of_bytes_read - 8)) {
                int number_of_extra_bytes = (number_of_bytes_read - 8) - decrypted_file_size;

                if (number_of_extra_bytes == 1) reversed_output &= ~(0xFF00000000000000);
                if (number_of_extra_bytes == 2) reversed_output &= ~(0xFFFF000000000000);
                if (number_of_extra_bytes == 3) reversed_output &= ~(0xFFFFFF0000000000);
                if (number_of_extra_bytes == 4) reversed_output &= ~(0xFFFFFFFF00000000);
                if (number_of_extra_bytes == 5) reversed_output &= ~(0xFFFFFFFFFF000000);
                if (number_of_extra_bytes == 6) reversed_output &= ~(0xFFFFFFFFFFFF0000);
                if (number_of_extra_bytes == 7) reversed_output &= ~(0xFFFFFFFFFFFFFF00);

                output_size -= number_of_extra_bytes;
            }

            output_stream.write((char*)&reversed_output, output_size);
        }
    } while (keep_going);

    // Everything is done. Get the end time and output how long the process took.
    clock_t endTime = clock();
    printf("Process completed in %.4f seconds. Press ENTER to exit.", (endTime - startTime)/ NUMBER_OF_MILLISECONDS_IN_SECOND);
    std::cin.get();
}

void process_chunk(uint64_t *next_64_bits, uint64_t *keys, uint64_t *output, Mode mode) {
    // Initial permutation
    uint64_t initial_permutation;
    apply_initial_permutation(next_64_bits, &initial_permutation);

    // Rounds 1 - 16
    uint64_t round_output;
    apply_rounds(&initial_permutation, &round_output, keys, mode);

    // Final Permutation
    apply_final_permutation(&round_output, output);
}

void set_ascii_characters_to_key(char most_significant_char, char least_significant_char, int bit_offset, uint64_t *destination) {
    int left_hand_side = most_significant_char - '0';
    int right_hand_side = least_significant_char - '0';

    // Make sure the user entered a valid hex digit
    if ((left_hand_side < (ASCII_ZERO - 48) || (left_hand_side >(ASCII_NINE - 48) && left_hand_side < (ASCII_A - 48)) || left_hand_side >(ASCII_F - 48)) ||
        (right_hand_side < (ASCII_ZERO - 48) || (right_hand_side >(ASCII_NINE - 48) && right_hand_side < (ASCII_A - 48)) || right_hand_side >(ASCII_F - 48))) {
        output_error("One or more characters are not valid hex characters", ExitCode::INVALID_PASSWORD);
    }

    *destination |= (uint64_t)((left_hand_side * 16) + right_hand_side) << bit_offset;
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
error_handler.cpp: Handles outputting errors to the console (displaying the provided text) and exiting with the provided exit code
*/
#include "error_handler.h"
#include <stdio.h>
#include <iostream>

// Handle outputting an error message, as well as holding the application from exiting until input from the keyboard is received.
// Then the application exits with the provided exit_code.
void output_error(char *text, int exit_code) {
    printf("ERROR: %s", text);
    printf("\nUsage: ./des [-[d|D]|-[e|E]] [password] [ECB|ecb] [input file path] [output file path], where -d or -D is decryption and -e or -E is encryption");
    printf("\n\nPress ENTER to exit.");

    std::cin.get();

    exit(exit_code);
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
file_processor.cpp: Handles safely retrieving the next 64 bits (or less if less than 64 bits exist) of a file using the provided file stream, number of bytes read, and the size of the file.
                    Also handles building the file size block to be stored in the encrypted file.
*/
#include "file_processor.h"
#include "garbage_producer.h"

bool get_next_64_bits(std::fstream &file_stream, uint64_t *destination, int *bytes_read, int file_size) {
    int number_of_bytes_to_read = 8;
    if (*bytes_read + number_of_bytes_to_read > file_size) {
        number_of_bytes_to_read = file_size - *bytes_read;
    }

    file_stream.read((char *)destination, number_of_bytes_to_read);
    *bytes_read += number_of_bytes_to_read;

    return *bytes_read != file_size;
}

void build_file_size_block(int file_size, uint64_t *block) {
    *block = 0;

    // The first 32 bits (left-hand side) are garbage
    uint64_t garbage;
    generate_eight_bytes_of_garbage(&garbage);

    // Clear the least significant word so we can put the file size there
    *block |= garbage << 32;

    // The next 32 bits are the file size
    *block |= file_size;
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
garbage_producer.cpp: Generates 64 random bits to be used by other functions
*/
#include "garbage_producer.h"
#include <stdlib.h>

// Generates eight bytes of garbage data. Why eight bytes?
// The most garbage we'll ever need is 7 bytes (for padding plaintext), and only generating 7 bytes seemed strange to me. So I went with 8 bytes.
void generate_eight_bytes_of_garbage(uint64_t *destination) {
    for (int i = 0; i < 64; i++) {
        if (rand() % 2) {
            *destination |= (INT64_C(1) << i);
        }
    }
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
keygen.cpp: Generates the 16 keys needed for each Feistel round, based on the key provided. Handles the compression permutations and rotations
*/
#include "keygen.h"
#include <stdio.h>

const int NUMBER_OF_KEYS_TO_GENERATE = 16;

// Function prototypes
void compress_original_key(uint64_t provided_key, uint64_t *compressed_key);
uint32_t rotate_left(uint32_t original, int number_of_bits_to_rotate);
void compress_rotated_key(uint64_t *combined_key, uint64_t *compressed_key);

void generate_keys(uint64_t provided_key, uint64_t *keys)
{
    uint64_t compressed_key = 0;
    compress_original_key(provided_key, &compressed_key);

    // Split the 56-bit key into two chunks, both 28 bits long.
    uint32_t original_left_side = (uint32_t)((compressed_key & 0xFFFFFFF000000000) >> 32);
    uint32_t original_right_side = (uint32_t)((compressed_key & 0x0000000FFFFFFF00) >> 4);

    for (int i = 0; i < NUMBER_OF_KEYS_TO_GENERATE; i++) {
        int rotation_amount = (i == 0 || i == 1 || i == 8 || i == 15)
            ? 1
            : 2;

        uint32_t left_side_rotated = rotate_left(original_left_side, rotation_amount);
        uint32_t right_side_rotated = rotate_left(original_right_side, rotation_amount);

        uint64_t combined_key = 0;

        //-- Taking our two halfs, combine them into the combined key
        if (left_side_rotated & (1 << (32 - 1))) combined_key |= (INT64_C(1) << (64 - 1));
        if (left_side_rotated & (1 << (32 - 2))) combined_key |= (INT64_C(1) << (64 - 2));
        if (left_side_rotated & (1 << (32 - 3))) combined_key |= (INT64_C(1) << (64 - 3));
        if (left_side_rotated & (1 << (32 - 4))) combined_key |= (INT64_C(1) << (64 - 4));
        if (left_side_rotated & (1 << (32 - 5))) combined_key |= (INT64_C(1) << (64 - 5));
        if (left_side_rotated & (1 << (32 - 6))) combined_key |= (INT64_C(1) <<
(64 - 6));
        if (left_side_rotated & (1 << (32 - 7))) combined_key |= (INT64_C(1) << (64 - 7));
        if (left_side_rotated & (1 << (32 - 8))) combined_key |= (INT64_C(1) << (64 - 8));
        if (left_side_rotated & (1 << (32 - 9))) combined_key |= (INT64_C(1) << (64 - 9));
        if (left_side_rotated & (1 << (32 - 10))) combined_key |= (INT64_C(1) << (64 - 10));
        if (left_side_rotated & (1 << (32 - 11))) combined_key |= (INT64_C(1) << (64 - 11));
        if (left_side_rotated & (1 << (32 - 12))) combined_key |= (INT64_C(1) << (64 - 12));
        if (left_side_rotated & (1 << (32 - 13))) combined_key |= (INT64_C(1) << (64 - 13));
        if (left_side_rotated & (1 << (32 - 14))) combined_key |= (INT64_C(1) << (64 - 14));
        if (left_side_rotated & (1 << (32 - 15))) combined_key |= (INT64_C(1) << (64 - 15));
        if (left_side_rotated & (1 << (32 - 16))) combined_key |= (INT64_C(1) << (64 - 16));
        if (left_side_rotated & (1 << (32 - 17))) combined_key |= (INT64_C(1) << (64 - 17));
        if (left_side_rotated & (1 << (32 - 18))) combined_key |= (INT64_C(1) << (64 - 18));
        if (left_side_rotated & (1 << (32 - 19))) combined_key |= (INT64_C(1) << (64 - 19));
        if (left_side_rotated & (1 << (32 - 20))) combined_key |= (INT64_C(1) << (64 - 20));
        if (left_side_rotated & (1 << (32 - 21))) combined_key |= (INT64_C(1) << (64 - 21));
        if (left_side_rotated & (1 << (32 - 22))) combined_key |= (INT64_C(1) << (64 - 22));
        if (left_side_rotated & (1 << (32 - 23))) combined_key |= (INT64_C(1) << (64 - 23));
        if (left_side_rotated & (1 << (32 - 24))) combined_key |= (INT64_C(1) << (64 - 24));
        if (left_side_rotated & (1 << (32 - 25))) combined_key |= (INT64_C(1) << (64 - 25));
        if (left_side_rotated & (1 << (32 - 26))) combined_key |= (INT64_C(1) << (64 - 26));
        if (left_side_rotated & (1 << (32 - 27))) combined_key |= (INT64_C(1) << (64 - 27));
        if (left_side_rotated & (1 << (32 - 28))) combined_key |= (INT64_C(1) << (64 - 28));
        if (right_side_rotated & (1 << (32 - 1))) combined_key |= (INT64_C(1) << (64 - 29));
        if (right_side_rotated & (1 << (32 - 2))) combined_key |= (INT64_C(1) << (64 - 30));
        if (right_side_rotated & (1 << (32 - 3))) combined_key |= (INT64_C(1) << (64 - 31));
        if (right_side_rotated & (1 << (32 - 4))) combined_key |= (INT64_C(1) <<
          (64 - 32));
        if (right_side_rotated & (1 << (32 - 5))) combined_key |= (INT64_C(1) << (64 - 33));
        if (right_side_rotated & (1 << (32 - 6))) combined_key |= (INT64_C(1) << (64 - 34));
        if (right_side_rotated & (1 << (32 - 7))) combined_key |= (INT64_C(1) << (64 - 35));
        if (right_side_rotated & (1 << (32 - 8))) combined_key |= (INT64_C(1) << (64 - 36));
        if (right_side_rotated & (1 << (32 - 9))) combined_key |= (INT64_C(1) << (64 - 37));
        if (right_side_rotated & (1 << (32 - 10))) combined_key |= (INT64_C(1) << (64 - 38));
        if (right_side_rotated & (1 << (32 - 11))) combined_key |= (INT64_C(1) << (64 - 39));
        if (right_side_rotated & (1 << (32 - 12))) combined_key |= (INT64_C(1) << (64 - 40));
        if (right_side_rotated & (1 << (32 - 13))) combined_key |= (INT64_C(1) << (64 - 41));
        if (right_side_rotated & (1 << (32 - 14))) combined_key |= (INT64_C(1) << (64 - 42));
        if (right_side_rotated & (1 << (32 - 15))) combined_key |= (INT64_C(1) << (64 - 43));
        if (right_side_rotated & (1 << (32 - 16))) combined_key |= (INT64_C(1) << (64 - 44));
        if (right_side_rotated & (1 << (32 - 17))) combined_key |= (INT64_C(1) << (64 - 45));
        if (right_side_rotated & (1 << (32 - 18))) combined_key |= (INT64_C(1) << (64 - 46));
        if (right_side_rotated & (1 << (32 - 19))) combined_key |= (INT64_C(1) << (64 - 47));
        if (right_side_rotated & (1 << (32 - 20))) combined_key |= (INT64_C(1) << (64 - 48));
        if (right_side_rotated & (1 << (32 - 21))) combined_key |= (INT64_C(1) << (64 - 49));
        if (right_side_rotated & (1 << (32 - 22))) combined_key |= (INT64_C(1) << (64 - 50));
        if (right_side_rotated & (1 << (32 - 23))) combined_key |= (INT64_C(1) << (64 - 51));
        if (right_side_rotated & (1 << (32 - 24))) combined_key |= (INT64_C(1) << (64 - 52));
        if (right_side_rotated & (1 << (32 - 25))) combined_key |= (INT64_C(1) << (64 - 53));
        if (right_side_rotated & (1 << (32 - 26))) combined_key |= (INT64_C(1) << (64 - 54));
        if (right_side_rotated & (1 << (32 - 27))) combined_key |= (INT64_C(1) << (64 - 55));
        if (right_side_rotated & (1 << (32 - 28))) combined_key |= (INT64_C(1) << (64 - 56));



        //-- Pass our combined key into another P-Box to strip this to 48 bits
        uint64_t permutated_key;
        compress_rotated_key(&combined_key, &permutated_key);

        //-- store in keys array
        keys[i] = permutated_key;

        //-- Our "new" 56 bit key is the one we rotated earlier
        original_left_side = left_side_rotated;
        original_right_side = right_side_rotated;
    }
}

void compress_original_key(uint64_t provided_key, uint64_t *compressed_key) {
    // Compression P-Box: 64 -> 56 bits
    if (provided_key & (INT64_C(1) << (64 - 57))) *compressed_key |= (INT64_C(1) << (64 - 1));
    if (provided_key & (INT64_C(1) << (64 - 49))) *compressed_key |= (INT64_C(1) << (64 - 2));
    if (provided_key & (INT64_C(1) << (64 - 41))) *compressed_key |= (INT64_C(1) << (64 - 3));
    if (provided_key & (INT64_C(1) << (64 - 33))) *compressed_key |= (INT64_C(1) << (64 - 4));
    if (provided_key & (INT64_C(1) << (64 - 25))) *compressed_key |= (INT64_C(1) << (64 - 5));
    if (provided_key & (INT64_C(1) << (64 - 17))) *compressed_key |= (INT64_C(1) << (64 - 6));
    if (provided_key & (INT64_C(1) << (64 - 9))) *compressed_key |= (INT64_C(1) << (64 - 7));
    if (provided_key & (INT64_C(1) << (64 - 1))) *compressed_key |= (INT64_C(1) << (64 - 8));
    if (provided_key & (INT64_C(1) << (64 - 58))) *compressed_key |= (INT64_C(1) << (64 - 9));
    if (provided_key & (INT64_C(1) << (64 - 50))) *compressed_key |= (INT64_C(1) << (64 - 10));
    if (provided_key & (INT64_C(1) << (64 - 42))) *compressed_key |= (INT64_C(1) << (64 - 11));
    if (provided_key & (INT64_C(1) << (64 - 34))) *compressed_key |= (INT64_C(1) << (64 - 12));
    if (provided_key & (INT64_C(1) << (64 - 26))) *compressed_key |= (INT64_C(1) << (64 - 13));
    if (provided_key & (INT64_C(1) << (64 - 18))) *compressed_key |= (INT64_C(1) << (64 - 14));
    if (provided_key & (INT64_C(1) << (64 - 10))) *compressed_key |= (INT64_C(1) << (64 - 15));
    if (provided_key & (INT64_C(1) << (64 - 2))) *compressed_key |= (INT64_C(1) << (64 - 16));
    if (provided_key & (INT64_C(1) << (64 - 59))) *compressed_key |= (INT64_C(1) << (64 - 17));
    if (provided_key & (INT64_C(1) << (64 - 51))) *compressed_key |= (INT64_C(1) << (64 - 18));
    if (provided_key & (INT64_C(1) << (64 - 43))) *compressed_key |= (INT64_C(1)
```

```cpp
        << (64 - 19));
    if (provided_key & (INT64_C(1) << (64 - 35))) *compressed_key |= (INT64_C(1) << (64 - 20));
    if (provided_key & (INT64_C(1) << (64 - 27))) *compressed_key |= (INT64_C(1) << (64 - 21));
    if (provided_key & (INT64_C(1) << (64 - 19))) *compressed_key |= (INT64_C(1) << (64 - 22));
    if (provided_key & (INT64_C(1) << (64 - 11))) *compressed_key |= (INT64_C(1) << (64 - 23));
    if (provided_key & (INT64_C(1) << (64 - 3))) *compressed_key |= (INT64_C(1) << (64 - 24));
    if (provided_key & (INT64_C(1) << (64 - 60))) *compressed_key |= (INT64_C(1) << (64 - 25));
    if (provided_key & (INT64_C(1) << (64 - 52))) *compressed_key |= (INT64_C(1) << (64 - 26));
    if (provided_key & (INT64_C(1) << (64 - 44))) *compressed_key |= (INT64_C(1) << (64 - 27));
    if (provided_key & (INT64_C(1) << (64 - 36))) *compressed_key |= (INT64_C(1) << (64 - 28));
    if (provided_key & (INT64_C(1) << (64 - 63))) *compressed_key |= (INT64_C(1) << (64 - 29));
    if (provided_key & (INT64_C(1) << (64 - 55))) *compressed_key |= (INT64_C(1) << (64 - 30));
    if (provided_key & (INT64_C(1) << (64 - 47))) *compressed_key |= (INT64_C(1) << (64 - 31));
    if (provided_key & (INT64_C(1) << (64 - 39))) *compressed_key |= (INT64_C(1) << (64 - 32));
    if (provided_key & (INT64_C(1) << (64 - 31))) *compressed_key |= (INT64_C(1) << (64 - 33));
    if (provided_key & (INT64_C(1) << (64 - 23))) *compressed_key |= (INT64_C(1) << (64 - 34));
    if (provided_key & (INT64_C(1) << (64 - 15))) *compressed_key |= (INT64_C(1) << (64 - 35));
    if (provided_key & (INT64_C(1) << (64 - 7))) *compressed_key |= (INT64_C(1) << (64 - 36));
    if (provided_key & (INT64_C(1) << (64 - 62))) *compressed_key |= (INT64_C(1) << (64 - 37));
    if (provided_key & (INT64_C(1) << (64 - 54))) *compressed_key |= (INT64_C(1) << (64 - 38));
    if (provided_key & (INT64_C(1) << (64 - 46))) *compressed_key |= (INT64_C(1) << (64 - 39));
    if (provided_key & (INT64_C(1) << (64 - 38))) *compressed_key |= (INT64_C(1) << (64 - 40));
    if (provided_key & (INT64_C(1) << (64 - 30))) *compressed_key |= (INT64_C(1) << (64 - 41));
    if (provided_key & (INT64_C(1) << (64 - 22))) *compressed_key |= (INT64_C(1) << (64 - 42));
    if (provided_key & (INT64_C(1) << (64 - 14))) *compressed_key |= (INT64_C(1) << (64 - 43));
    if (provided_key & (INT64_C(1) << (64 - 6))) *compressed_key |= (INT64_C(1) << (64 - 44));
    if (provided_key & (INT64_C(1) << (64 - 61))) *compressed_key |= (INT64_C(1)
<< (64 - 45));
    if (provided_key & (INT64_C(1) << (64 - 53))) *compressed_key |= (INT64_C(1) << (64 - 46));
    if (provided_key & (INT64_C(1) << (64 - 45))) *compressed_key |= (INT64_C(1) << (64 - 47));
    if (provided_key & (INT64_C(1) << (64 - 37))) *compressed_key |= (INT64_C(1) << (64 - 48));
    if (provided_key & (INT64_C(1) << (64 - 29))) *compressed_key |= (INT64_C(1) << (64 - 49));
    if (provided_key & (INT64_C(1) << (64 - 21))) *compressed_key |= (INT64_C(1) << (64 - 50));
    if (provided_key & (INT64_C(1) << (64 - 13))) *compressed_key |= (INT64_C(1) << (64 - 51));
    if (provided_key & (INT64_C(1) << (64 - 5))) *compressed_key |= (INT64_C(1) << (64 - 52));
    if (provided_key & (INT64_C(1) << (64 - 28))) *compressed_key |= (INT64_C(1) << (64 - 53));
    if (provided_key & (INT64_C(1) << (64 - 20))) *compressed_key |= (INT64_C(1) << (64 - 54));
    if (provided_key & (INT64_C(1) << (64 - 12))) *compressed_key |= (INT64_C(1) << (64 - 55));
    if (provided_key & (INT64_C(1) << (64 - 4))) *compressed_key |= (INT64_C(1) << (64 - 56));
}

/* Helper methods to rotate our 28 bit value left or right:

    The first part of the rotate blindly rotates everything to the left (or right) by the number of bits specified (in our case, 1 or 2)
    Then, because we lost at least 1 bit during the rotate (if left-rotate, the MS bits; if right-rotate, the LS bits), we rotate right by the number of bits lost to push them to the other end
*/
uint32_t rotate_left(uint32_t original, int number_of_bits_to_rotate) {
    uint32_t rotated = ((original << number_of_bits_to_rotate) | (original >> (28 - number_of_bits_to_rotate)));

    // Clear bits 29-32 just to be safe
    rotated &= ~(1 << (32 - 29));
    rotated &= ~(1 << (32 - 30));
    rotated &= ~(1 << (32 - 31));
    rotated &= ~(1 << (32 - 32));
    return rotated;
}

void compress_rotated_key(uint64_t *combined_key, uint64_t *compressed_key) {
    *compressed_key = 0;

    if (*combined_key & (INT64_C(1) << (64 - 14))) *compressed_key |= (INT64_C(1) << (64 - 1));
    if (*combined_key & (INT64_C(1) << (64 - 17))) *compressed_key |= (INT64_C(1) << (64 - 2));
```

```cpp
    if (*combined_key & (INT64_C(1) << (64 - 11))) *compressed_key |= (INT64_C(1) << (64 - 3));
    if (*combined_key & (INT64_C(1) << (64 - 24))) *compressed_key |= (INT64_C(1) << (64 - 4));
    if (*combined_key & (INT64_C(1) << (64 - 1))) *compressed_key |= (INT64_C(1) << (64 - 5));
    if (*combined_key & (INT64_C(1) << (64 - 5))) *compressed_key |= (INT64_C(1) << (64 - 6));
    if (*combined_key & (INT64_C(1) << (64 - 3))) *compressed_key |= (INT64_C(1) << (64 - 7));
    if (*combined_key & (INT64_C(1) << (64 - 28))) *compressed_key |= (INT64_C(1) << (64 - 8));
    if (*combined_key & (INT64_C(1) << (64 - 15))) *compressed_key |= (INT64_C(1) << (64 - 9));
    if (*combined_key & (INT64_C(1) << (64 - 6))) *compressed_key |= (INT64_C(1) << (64 - 10));
    if (*combined_key & (INT64_C(1) << (64 - 21))) *compressed_key |= (INT64_C(1) << (64 - 11));
    if (*combined_key & (INT64_C(1) << (64 - 10))) *compressed_key |= (INT64_C(1) << (64 - 12));
    if (*combined_key & (INT64_C(1) << (64 - 23))) *compressed_key |= (INT64_C(1) << (64 - 13));
    if (*combined_key & (INT64_C(1) << (64 - 19))) *compressed_key |= (INT64_C(1) << (64 - 14));
    if (*combined_key & (INT64_C(1) << (64 - 12))) *compressed_key |= (INT64_C(1) << (64 - 15));
    if (*combined_key & (INT64_C(1) << (64 - 4))) *compressed_key |= (INT64_C(1) << (64 - 16));
    if (*combined_key & (INT64_C(1) << (64 - 26))) *compressed_key |= (INT64_C(1) << (64 - 17));
    if (*combined_key & (INT64_C(1) << (64 - 8))) *compressed_key |= (INT64_C(1) << (64 - 18));
    if (*combined_key & (INT64_C(1) << (64 - 16))) *compressed_key |= (INT64_C(1) << (64 - 19));
    if (*combined_key & (INT64_C(1) << (64 - 7))) *compressed_key |= (INT64_C(1) << (64 - 20));
    if (*combined_key & (INT64_C(1) << (64 - 27))) *compressed_key |= (INT64_C(1) << (64 - 21));
    if (*combined_key & (INT64_C(1) << (64 - 20))) *compressed_key |= (INT64_C(1) << (64 - 22));
    if (*combined_key & (INT64_C(1) << (64 - 13))) *compressed_key |= (INT64_C(1) << (64 - 23));
    if (*combined_key & (INT64_C(1) << (64 - 2))) *compressed_key |= (INT64_C(1) << (64 - 24));
    if (*combined_key & (INT64_C(1) << (64 - 41))) *compressed_key |= (INT64_C(1) << (64 - 25));
    if (*combined_key & (INT64_C(1) << (64 - 52))) *compressed_key |= (INT64_C(1) << (64 - 26));
    if (*combined_key & (INT64_C(1) << (64 - 31))) *compressed_key |= (INT64_C(1) << (64 - 27));
    if (*combined_key & (INT64_C(1) << (64 - 37))) *compressed_key |= (INT64_C(1) << (64 - 28));
    if (*combined_key & (INT64_C(1) << (64 - 47))) *compressed_key |= (INT64_C(1) << (64 - 29));
    if (*combined_key & (INT64_C(1) << (64 - 55))) *compressed_key |= (INT64_C(1) << (64 - 30));
    if (*combined_key & (INT64_C(1) << (64 - 30))) *compressed_key |= (INT64_C(1) << (64 - 31));
    if (*combined_key & (INT64_C(1) << (64 - 40))) *compressed_key |= (INT64_C(1) << (64 - 32));
    if (*combined_key & (INT64_C(1) << (64 - 51))) *compressed_key |= (INT64_C(1) << (64 - 33));
    if (*combined_key & (INT64_C(1) << (64 - 45))) *compressed_key |= (INT64_C(1) << (64 - 34));
    if (*combined_key & (INT64_C(1) << (64 - 33))) *compressed_key |= (INT64_C(1) << (64 - 35));
    if (*combined_key & (INT64_C(1) << (64 - 48))) *compressed_key |= (INT64_C(1) << (64 - 36));
    if (*combined_key & (INT64_C(1) << (64 - 44))) *compressed_key |= (INT64_C(1) << (64 - 37));
    if (*combined_key & (INT64_C(1) << (64 - 49))) *compressed_key |= (INT64_C(1) << (64 - 38));
    if (*combined_key & (INT64_C(1) << (64 - 39))) *compressed_key |= (INT64_C(1) << (64 - 39));
    if (*combined_key & (INT64_C(1) << (64 - 56))) *compressed_key |= (INT64_C(1) << (64 - 40));
    if (*combined_key & (INT64_C(1) << (64 - 34))) *compressed_key |= (INT64_C(1) << (64 - 41));
    if (*combined_key & (INT64_C(1) << (64 - 53))) *compressed_key |= (INT64_C(1) << (64 - 42));
    if (*combined_key & (INT64_C(1) << (64 - 46))) *compressed_key |= (INT64_C(1) << (64 - 43));
    if (*combined_key & (INT64_C(1) << (64 - 42))) *compressed_key |= (INT64_C(1) << (64 - 44));
    if (*combined_key & (INT64_C(1) << (64 - 50))) *compressed_key |= (INT64_C(1) << (64 - 45));
    if (*combined_key & (INT64_C(1) << (64 - 36))) *compressed_key |= (INT64_C(1) << (64 - 46));
    if (*combined_key & (INT64_C(1) << (64 - 29))) *compressed_key |= (INT64_C(1) << (64 - 47));
    if (*combined_key & (INT64_C(1) << (64 - 32))) *compressed_key |= (INT64_C(1) << (64 - 48));
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
permutator.cpp: Handles the initial, expansion, and final permutations. Stores the result in the "destination"
*/
#include "permutator.h"
#include <stdio.h>

void apply_initial_permutation(uint64_t *source, uint64_t *destination) {
    /*
    Applies the following permutation based on the following chart:

                    58 50 42 34 26 18 10 02
                    60 52 44 36 28 20 12 04
                    62 54 46 38 30 22 14 06
                    64 56 48 40 32 24 16 08
                    57 49 41 33 25 17 09 01
                    59 51 43 35 27 19 11 03
                    61 53 45 37 29 21 13 05
                    63 55 47 39 31 23 15 07
    */
    *destination = 0;
    // Using these quick and dirty if statements to only set the bits that are "on" from the source. Faster than using a loop and a look-up table.
    if (*source & (INT64_C(1) << (64 - 58))) *destination |= (INT64_C(1) << (64 - 1));
    if (*source & (INT64_C(1) << (64 - 50))) *destination |= (INT64_C(1) << (64 - 2));
    if (*source & (INT64_C(1) << (64 - 42))) *destination |= (INT64_C(1) << (64 - 3));
    if (*source & (INT64_C(1) << (64 - 34))) *destination |= (INT64_C(1) << (64 - 4));
    if (*source & (INT64_C(1) << (64 - 26))) *destination |= (INT64_C(1) << (64 - 5));
    if (*source & (INT64_C(1) << (64 - 18))) *destination |= (INT64_C(1) << (64 - 6));
    if (*source & (INT64_C(1) << (64 - 10))) *destination |= (INT64_C(1) << (64 - 7));
    if (*source & (INT64_C(1) << (64 - 2))) *destination |= (INT64_C(1) << (64 - 8));
    if (*source & (INT64_C(1) << (64 - 60))) *destination |= (INT64_C(1) << (64 - 9));
    if (*source & (INT64_C(1) << (64 - 52))) *destination |= (INT64_C(1) << (64 - 10));
    if (*source & (INT64_C(1) << (64 - 44))) *destination |= (INT64_C(1) << (64 - 11));
    if (*source & (INT64_C(1) << (64 - 36))) *destination |= (INT64_C(1) << (64 - 12));
    if (*source & (INT64_C(1) << (64 - 28))) *destination |= (INT64_C(1) << (64 - 13));
    if (*source & (INT64_C(1) << (64 - 20))) *destination |= (INT64_C(1) << (64 - 14));
    if (*source & (INT64_C(1) << (64 - 12))) *destination |= (INT64_C(1) << (64 - 15));
    if (*source & (INT64_C(1) << (64 - 4))) *destination |= (INT64_C(1) << (64 - 16));
    if (*source & (INT64_C(1) << (64 - 62))) *destination |= (INT64_C(1) << (64 - 17));
    if (*source & (INT64_C(1) << (64 - 54))) *destination |= (INT64_C(1) << (64 - 18));
    if (*source & (INT64_C(1) << (64 - 46))) *destination |= (INT64_C(1) << (64 - 19));
    if (*source & (INT64_C(1) << (64 - 38))) *destination |= (INT64_C(1) << (64 - 20));
    if (*source & (INT64_C(1) << (64 - 30))) *destination |= (INT64_C(1) << (64 - 21));
    if (*source & (INT64_C(1) << (64 - 22))) *destination |= (INT64_C(1) << (64 - 22));
    if (*source & (INT64_C(1) << (64 - 14))) *destination |= (INT64_C(1) << (64 - 23));
    if (*source & (INT64_C(1) << (64 - 6))) *destination |= (INT64_C(1) << (64 - 24));
    if (*source & (INT64_C(1) << (64 - 64))) *destination |= (INT64_C(1) << (64 - 25));
    if (*source & (INT64_C(1) << (64 - 56))) *destination |= (INT64_C(1) << (64 - 26));
    if (*source & (INT64_C(1) << (64 - 48))) *destination |= (INT64_C(1) << (64 - 27));
    if (*source & (INT64_C(1) << (64 - 40))) *destination |= (INT64_C(1) << (64 - 28));
    if (*source & (INT64_C(1) << (64 - 32))) *destination |= (INT64_C(1) << (64 - 29));
    if (*source & (INT64_C(1) << (64 - 24))) *destination |= (INT64_C(1) << (64 - 30));
    if (*source & (INT64_C(1) << (64 - 16))) *destination |= (INT64_C(1) << (64 - 31));
    if (*source & (INT64_C(1) << (64 - 8))) *destination |= (INT64_C(1) << (64 - 32));
    if (*source & (INT64_C(1) << (64 - 57))) *destination |= (INT64_C(1) << (64 - 33));
    if (*source & (INT64_C(1) << (64 - 49))) *destination |= (INT64_C(1) << (64 - 34));
    if (*source & (INT64_C(1) << (64 - 41))) *destination |= (INT64_C(1) << (64 - 35));
    if (*source & (INT64_C(1) << (64 - 33))) *destination |= (INT64_C(1) << (64 - 36));
    if (*source & (INT64_C(1) << (64 - 25))) *destination |= (INT64_C(1) << (64 - 37));
    if (*source & (INT64_C(1) << (64 - 17))) *destination |= (INT64_C(1) << (64 - 38));
    if (*source & (INT64_C(1) << (64 - 9))) *destination |= (INT64_C(1) << (64 - 39));
    if (*source & (INT64_C(1) << (64 - 1))) *destination |= (INT64_C(1) << (64 - 40));
    if (*source & (INT64_C(1) << (64 - 59))) *destination |= (INT64_C(1) << (64 - 41));
    if (*source & (INT64_C(1) << (64 - 51))) *destination |= (INT64_C(1) << (64 - 42));
    if (*source & (INT64_C(1) << (64 - 43))) *destination |= (INT64_C(1) << (64 - 43));
    if (*source & (INT64_C(1) << (64 - 35))) *destination |= (INT64_C(1) << (64 - 44));
    if (*source & (INT64_C(1) << (64 - 27))) *destination |= (INT64_C(1) << (64 - 45));
    if (*source & (INT64_C(1) << (64 - 19))) *destination |= (INT64_C(1) << (64 - 46));
    if (*source & (INT64_C(1) << (64 - 11))) *destination |= (INT64_C(1) << (64 - 47));
    if (*source & (INT64_C(1) << (64 - 3))) *destination |= (INT64_C(1) << (64 - 48));
    if (*source & (INT64_C(1) << (64 - 61))) *destination |= (INT64_C(1) << (64 - 49));
    if (*source & (INT64_C(1) << (64 - 53))) *destination |= (INT64_C(1) << (64 - 50));
    if (*source & (INT64_C(1) << (64 - 45))) *destination |= (INT64_C(1) << (64 - 51));
    if (*source & (INT64_C(1) << (64 - 37))) *destination |= (INT64_C(1) << (64 - 52));
    if (*source & (INT64_C(1) << (64 - 29))) *destination |= (INT64_C(1) << (64 - 53));
    if (*source & (INT64_C(1) << (64 - 21))) *destination |= (INT64_C(1) << (64 - 54));
    if (*source & (INT64_C(1) << (64 - 13))) *destination |= (INT64_C(1) << (64 - 55));
    if (*source & (INT64_C(1) << (64 - 5))) *destination |= (INT64_C(1) << (64 - 56));
    if (*source & (INT64_C(1) << (64 - 63))) *destination |= (INT64_C(1) << (64 - 57));
    if (*source & (INT64_C(1) << (64 - 55))) *destination |= (INT64_C(1) << (64 - 58));
    if (*source & (INT64_C(1) << (64 - 47))) *destination |= (INT64_C(1) << (64 - 59));
    if (*source & (INT64_C(1) << (64 - 39))) *destination |= (INT64_C(1) << (64 - 60));
    if (*source & (INT64_C(1) << (64 - 31))) *destination |= (INT64_C(1) << (64 - 61));
    if (*source & (INT64_C(1) << (64 - 23))) *destination |= (INT64_C(1) << (64 - 62));
    if (*source & (INT64_C(1) << (64 - 15))) *destination |= (INT64_C(1) << (64 - 63));
    if (*source & (INT64_C(1) << (64 - 7))) *destination |= (INT64_C(1) << (64 - 64));

}
```

```

void apply_expansion_permutation(uint32_t *source, uint64_t *destination) {
    /*
    Applies the following permutation based on the following chart:

                   32 01 02 03 04 05
                   04 05 06 07 08 09
                   08 09 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 01
    */
    *destination = 0;
    if (*source & (INT64_C(1) << (32 - 32))) *destination |= (INT64_C(1) << (64 - 1));
    if (*source & (INT64_C(1) << (32 - 1))) *destination |= (INT64_C(1) << (64 - 2));
    if (*source & (INT64_C(1) << (32 - 2))) *destination |= (INT64_C(1) << (64 - 3));
    if (*source & (INT64_C(1) << (32 - 3))) *destination |= (INT64_C(1) << (64 - 4));
    if (*source & (INT64_C(1) << (32 - 4))) *destination |= (INT64_C(1) << (64 - 5));
    if (*source & (INT64_C(1) << (32 - 5))) *destination |= (INT64_C(1) << (64 - 6));
    if (*source & (INT64_C(1) << (32 - 4))) *destination |= (INT64_C(1) << (64 - 7));
    if (*source & (INT64_C(1) << (32 - 5))) *destination |= (INT64_C(1) << (64 - 8));
    if (*source & (INT64_C(1) << (32 - 6))) *destination |= (INT64_C(1) << (64 - 9));
    if (*source & (INT64_C(1) << (32 - 7))) *destination |= (INT64_C(1) << (64 - 10));
    if (*source & (INT64_C(1) << (32 - 8))) *destination |= (INT64_C(1) << (64 - 11));
    if (*source & (INT64_C(1) << (32 - 9))) *destination |= (INT64_C(1) << (64 - 12));
    if (*source & (INT64_C(1) << (32 - 8))) *destination |= (INT64_C(1) << (64 - 13));
    if (*source & (INT64_C(1) << (32 - 9))) *destination |= (INT64_C(1) << (64 - 14));
    if (*source & (INT64_C(1) << (32 - 10))) *destination |= (INT64_C(1) << (64 - 15));
    if (*source & (INT64_C(1) << (32 - 11))) *destination |= (INT64_C(1) << (64 - 16));
    if (*source & (INT64_C(1) << (32 - 12))) *destination |= (INT64_C(1) << (64 - 17));
    if (*source & (INT64_C(1) << (32 - 13))) *destination |= (INT64_C(1) << (64 - 18));
    if (*source & (INT64_C(1) << (32 - 12))) *destination |= (INT64_C(1) << (64 -
      19));
    if (*source & (INT64_C(1) << (32 - 13))) *destination |= (INT64_C(1) << (64 - 20));
    if (*source & (INT64_C(1) << (32 - 14))) *destination |= (INT64_C(1) << (64 - 21));
    if (*source & (INT64_C(1) << (32 - 15))) *destination |= (INT64_C(1) << (64 - 22));
    if (*source & (INT64_C(1) << (32 - 16))) *destination |= (INT64_C(1) << (64 - 23));
    if (*source & (INT64_C(1) << (32 - 17))) *destination |= (INT64_C(1) << (64 - 24));
    if (*source & (INT64_C(1) << (32 - 16))) *destination |= (INT64_C(1) << (64 - 25));
    if (*source & (INT64_C(1) << (32 - 17))) *destination |= (INT64_C(1) << (64 - 26));
    if (*source & (INT64_C(1) << (32 - 18))) *destination |= (INT64_C(1) << (64 - 27));
    if (*source & (INT64_C(1) << (32 - 19))) *destination |= (INT64_C(1) << (64 - 28));
    if (*source & (INT64_C(1) << (32 - 20))) *destination |= (INT64_C(1) << (64 - 29));
    if (*source & (INT64_C(1) << (32 - 21))) *destination |= (INT64_C(1) << (64 - 30));
    if (*source & (INT64_C(1) << (32 - 20))) *destination |= (INT64_C(1) << (64 - 31));
    if (*source & (INT64_C(1) << (32 - 21))) *destination |= (INT64_C(1) << (64 - 32));
    if (*source & (INT64_C(1) << (32 - 22))) *destination |= (INT64_C(1) << (64 - 33));
    if (*source & (INT64_C(1) << (32 - 23))) *destination |= (INT64_C(1) << (64 - 34));
    if (*source & (INT64_C(1) << (32 - 24))) *destination |= (INT64_C(1) << (64 - 35));
    if (*source & (INT64_C(1) << (32 - 25))) *destination |= (INT64_C(1) << (64 - 36));
    if (*source & (INT64_C(1) << (32 - 24))) *destination |= (INT64_C(1) << (64 - 37));
    if (*source & (INT64_C(1) << (32 - 25))) *destination |= (INT64_C(1) << (64 - 38));
    if (*source & (INT64_C(1) << (32 - 26))) *destination |= (INT64_C(1) << (64 - 39));
    if (*source & (INT64_C(1) << (32 - 27))) *destination |= (INT64_C(1) << (64 - 40));
    if (*source & (INT64_C(1) << (32 - 28))) *destination |= (INT64_C(1) << (64 - 41));
    if (*source & (INT64_C(1) << (32 - 29))) *destination |= (INT64_C(1) << (64 - 42));
    if (*source & (INT64_C(1) << (32 - 28))) *destination |= (INT64_C(1) << (64 - 43));
    if (*source & (INT64_C(1) << (32 - 29))) *destination |= (INT64_C(1) << (64 - 44));
    if (*source & (INT64_C(1) << (32 - 30))) *destination |= (INT64_C(1) << (64 -
```

```cpp
45));
    if (*source & (INT64_C(1) << (32 - 31))) *destination |= (INT64_C(1) << (64 - 46));
    if (*source & (INT64_C(1) << (32 - 32))) *destination |= (INT64_C(1) << (64 - 47));
    if (*source & (INT64_C(1) << (32 - 1))) *destination |= (INT64_C(1) << (64 - 48));
}

void apply_final_permutation(uint64_t *source, uint64_t *destination) {
    /*
    Applies the following permutation based on the following chart:
                        40 08 48 16 56 24 64 32
                        39 07 47 15 55 23 63 31
                        38 06 46 14 54 22 62 30
                        37 05 45 13 53 21 61 29
                        36 04 44 12 52 20 60 28
                        35 03 43 11 51 19 59 27
                        34 02 42 10 50 18 58 26
                        33 01 41 09 49 17 57 25
    */
    *destination = 0;
    if (*source & (INT64_C(1) << (64 - 40))) *destination |= (INT64_C(1) << (64 - 1));
    if (*source & (INT64_C(1) << (64 - 8))) *destination |= (INT64_C(1) << (64 - 2));
    if (*source & (INT64_C(1) << (64 - 48))) *destination |= (INT64_C(1) << (64 - 3));
    if (*source & (INT64_C(1) << (64 - 16))) *destination |= (INT64_C(1) << (64 - 4));
    if (*source & (INT64_C(1) << (64 - 56))) *destination |= (INT64_C(1) << (64 - 5));
    if (*source & (INT64_C(1) << (64 - 24))) *destination |= (INT64_C(1) << (64 - 6));
    if (*source & (INT64_C(1) << (64 - 64))) *destination |= (INT64_C(1) << (64 - 7));
    if (*source & (INT64_C(1) << (64 - 32))) *destination |= (INT64_C(1) << (64 - 8));
    if (*source & (INT64_C(1) << (64 - 39))) *destination |= (INT64_C(1) << (64 - 9));
    if (*source & (INT64_C(1) << (64 - 7))) *destination |= (INT64_C(1) << (64 - 10));
    if (*source & (INT64_C(1) << (64 - 47))) *destination |= (INT64_C(1) << (64 - 11));
    if (*source & (INT64_C(1) << (64 - 15))) *destination |= (INT64_C(1) << (64 - 12));
    if (*source & (INT64_C(1) << (64 - 55))) *destination |= (INT64_C(1) << (64 - 13));
    if (*source & (INT64_C(1) << (64 - 23))) *destination |= (INT64_C(1) << (64 - 14));
    if (*source & (INT64_C(1) << (64 - 63))) *destination |= (INT64_C(1) << (64 - 15));
    if (*source & (INT64_C(1) << (64 - 31))) *destination |= (INT64_C(1) << (64 - 16));
    if (*source & (INT64_C(1) << (64 - 38))) *destination |= (INT64_C(1) << (64 - 17));
    if (*source & (INT64_C(1) << (64 - 6))) *destination |= (INT64_C(1) << (64 - 18));
    if (*source & (INT64_C(1) << (64 - 46))) *destination |= (INT64_C(1) << (64 - 19));
    if (*source & (INT64_C(1) << (64 - 14))) *destination |= (INT64_C(1) << (64 - 20));
    if (*source & (INT64_C(1) << (64 - 54))) *destination |= (INT64_C(1) << (64 - 21));
    if (*source & (INT64_C(1) << (64 - 22))) *destination |= (INT64_C(1) << (64 - 22));
    if (*source & (INT64_C(1) << (64 - 62))) *destination |= (INT64_C(1) << (64 - 23));
    if (*source & (INT64_C(1) << (64 - 30))) *destination |= (INT64_C(1) << (64 - 24));
    if (*source & (INT64_C(1) << (64 - 37))) *destination |= (INT64_C(1) << (64 - 25));
    if (*source & (INT64_C(1) << (64 - 5))) *destination |= (INT64_C(1) << (64 - 26));
    if (*source & (INT64_C(1) << (64 - 45))) *destination |= (INT64_C(1) << (64 - 27));
    if (*source & (INT64_C(1) << (64 - 13))) *destination |= (INT64_C(1) << (64 - 28));
    if (*source & (INT64_C(1) << (64 - 53))) *destination |= (INT64_C(1) << (64 - 29));
    if (*source & (INT64_C(1) << (64 - 21))) *destination |= (INT64_C(1) << (64 - 30));
    if (*source & (INT64_C(1) << (64 - 61))) *destination |= (INT64_C(1) << (64 - 31));
    if (*source & (INT64_C(1) << (64 - 29))) *destination |= (INT64_C(1) << (64 - 32));
    if (*source & (INT64_C(1) << (64 - 36))) *destination |= (INT64_C(1) << (64 - 33));
    if (*source & (INT64_C(1) << (64 - 4))) *destination |= (INT64_C(1) << (64 - 34));
    if (*source & (INT64_C(1) << (64 - 44))) *destination |= (INT64_C(1) << (64 - 35));
    if (*source & (INT64_C(1) << (64 - 12))) *destination |= (INT64_C(1) << (64 - 36));
    if (*source & (INT64_C(1) << (64 - 52))) *destination |= (INT64_C(1) << (64 - 37));
    if (*source & (INT64_C(1) << (64 - 20))) *destination |= (INT64_C(1) << (64 - 38));
    if (*source & (INT64_C(1) << (64 - 60))) *destination |= (INT64_C(1) << (64 - 39));
    if (*source & (INT64_C(1) << (64 - 28))) *destination |= (INT64_C(1) << (64 - 40));
    if (*source & (INT64_C(1) << (64 - 35))) *destination |= (INT64_C(1) << (64 - 41));
```

```cpp
    if (*source & (INT64_C(1) << (64 - 3))) *destination |= (INT64_C(1) << (64 - 42));
    if (*source & (INT64_C(1) << (64 - 43))) *destination |= (INT64_C(1) << (64 - 43));
    if (*source & (INT64_C(1) << (64 - 11))) *destination |= (INT64_C(1) << (64 - 44));
    if (*source & (INT64_C(1) << (64 - 51))) *destination |= (INT64_C(1) << (64 - 45));
    if (*source & (INT64_C(1) << (64 - 19))) *destination |= (INT64_C(1) << (64 - 46));
    if (*source & (INT64_C(1) << (64 - 59))) *destination |= (INT64_C(1) << (64 - 47));
    if (*source & (INT64_C(1) << (64 - 27))) *destination |= (INT64_C(1) << (64 - 48));
    if (*source & (INT64_C(1) << (64 - 34))) *destination |= (INT64_C(1) << (64 - 49));
    if (*source & (INT64_C(1) << (64 - 2))) *destination |= (INT64_C(1) << (64 - 50));
    if (*source & (INT64_C(1) << (64 - 42))) *destination |= (INT64_C(1) << (64 - 51));
    if (*source & (INT64_C(1) << (64 - 10))) *destination |= (INT64_C(1) << (64 - 52));
    if (*source & (INT64_C(1) << (64 - 50))) *destination |= (INT64_C(1) << (64 - 53));
    if (*source & (INT64_C(1) << (64 - 18))) *destination |= (INT64_C(1) << (64 - 54));
    if (*source & (INT64_C(1) << (64 - 58))) *destination |= (INT64_C(1) << (64 - 55));
    if (*source & (INT64_C(1) << (64 - 26))) *destination |= (INT64_C(1) << (64 - 56));
    if (*source & (INT64_C(1) << (64 - 33))) *destination |= (INT64_C(1) << (64 - 57));
    if (*source & (INT64_C(1) << (64 - 1))) *destination |= (INT64_C(1) << (64 - 58));
    if (*source & (INT64_C(1) << (64 - 41))) *destination |= (INT64_C(1) << (64 - 59));
    if (*source & (INT64_C(1) << (64 - 9))) *destination |= (INT64_C(1) << (64 - 60));
    if (*source & (INT64_C(1) << (64 - 49))) *destination |= (INT64_C(1) << (64 - 61));
    if (*source & (INT64_C(1) << (64 - 17))) *destination |= (INT64_C(1) << (64 - 62));
    if (*source & (INT64_C(1) << (64 - 57))) *destination |= (INT64_C(1) << (64 - 63));
    if (*source & (INT64_C(1) << (64 - 25))) *destination |= (INT64_C(1) << (64 - 64));
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
reverser.cpp: Reverses a 64-bit value at the byte level (MSByte -> LSByte, etc) from the source. Stores the result in the "destination" (provided)
*/
#include "reverser.h"

void reverse(uint64_t *source, uint64_t *destination) {
    *destination = 0;

    // Everything is going to be reversed at the *byte* level, not the bit level
    // 01-08 -> 57-64     09-16 -> 49-56     17-24 -> 41-48
    // 25-32 -> 33-40     33-40 -> 25-32     41-48 -> 17-24
    // 49-56 -> 09-16     57-64 -> 01-08
    if (*source & (INT64_C(1) << 0)) *destination |= (INT64_C(1) << 56);
    if (*source & (INT64_C(1) << 1)) *destination |= (INT64_C(1) << 57);
    if (*source & (INT64_C(1) << 2)) *destination |= (INT64_C(1) << 58);
    if (*source & (INT64_C(1) << 3)) *destination |= (INT64_C(1) << 59);
    if (*source & (INT64_C(1) << 4)) *destination |= (INT64_C(1) << 60);
    if (*source & (INT64_C(1) << 5)) *destination |= (INT64_C(1) << 61);
    if (*source & (INT64_C(1) << 6)) *destination |= (INT64_C(1) << 62);
    if (*source & (INT64_C(1) << 7)) *destination |= (INT64_C(1) << 63);
    if (*source & (INT64_C(1) << 8)) *destination |= (INT64_C(1) << 48);
    if (*source & (INT64_C(1) << 9)) *destination |= (INT64_C(1) << 49);
    if (*source & (INT64_C(1) << 10)) *destination |= (INT64_C(1) << 50);
    if (*source & (INT64_C(1) << 11)) *destination |= (INT64_C(1) << 51);
    if (*source & (INT64_C(1) << 12)) *destination |= (INT64_C(1) << 52);
    if (*source & (INT64_C(1) << 13)) *destination |= (INT64_C(1) << 53);
    if (*source & (INT64_C(1) << 14)) *destination |= (INT64_C(1) << 54);
    if (*source & (INT64_C(1) << 15)) *destination |= (INT64_C(1) << 55);
    if (*source & (INT64_C(1) << 16)) *destination |= (INT64_C(1) << 40);
    if (*source & (INT64_C(1) << 17)) *destination |= (INT64_C(1) << 41);
    if (*source & (INT64_C(1) << 18)) *destination |= (INT64_C(1) << 42);
    if (*source & (INT64_C(1) << 19)) *destination |= (INT64_C(1) << 43);
    if (*source & (INT64_C(1) << 20)) *destination |= (INT64_C(1) << 44);
    if (*source & (INT64_C(1) << 21)) *destination |= (INT64_C(1) << 45);
    if (*source & (INT64_C(1) << 22)) *destination |= (INT64_C(1) << 46);
    if (*source & (INT64_C(1) << 23)) *destination |= (INT64_C(1) << 47);
    if (*source & (INT64_C(1) << 24)) *destination |= (INT64_C(1) << 32);
    if (*source & (INT64_C(1) << 25)) *destination |= (INT64_C(1) << 33);
    if (*source & (INT64_C(1) << 26)) *destination |= (INT64_C(1) << 34);
    if (*source & (INT64_C(1) << 27)) *destination |= (INT64_C(1) << 35);
    if (*source & (INT64_C(1) << 28)) *destination |= (INT64_C(1) << 36);
    if (*source & (INT64_C(1) << 29)) *destination |= (INT64_C(1) << 37);
    if (*source & (INT64_C(1) << 30)) *destination |= (INT64_C(1) << 38);
    if (*source & (INT64_C(1) << 31)) *destination |= (INT64_C(1) << 39);
    if (*source & (INT64_C(1) << 32)) *destination |= (INT64_C(1) << 24);
    if (*source & (INT64_C(1) << 33)) *destination |= (INT64_C(1) << 25);
    if (*source & (INT64_C(1) << 34)) *destination |= (INT64_C(1) << 26);
    if (*source & (INT64_C(1) << 35)) *destination |= (INT64_C(1) << 27);
    if (*source & (INT64_C(1) << 36)) *destination |= (INT64_C(1) << 28);
    if (*source & (INT64_C(1) << 37)) *destination |= (INT64_C(1) << 29);
    if (*source & (INT64_C(1) << 38)) *destination |= (INT64_C(1) << 30);
    if (*source & (INT64_C(1) << 39)) *destination |= (INT64_C(1) << 31);
    if (*source & (INT64_C(1) << 40)) *destination |= (INT64_C(1) << 16);
    if (*source & (INT64_C(1) << 41)) *destination |= (INT64_C(1) << 17);
    if (*source & (INT64_C(1) << 42)) *destination |= (INT64_C(1) << 18);
    if (*source & (INT64_C(1) << 43)) *destination |= (INT64_C(1) << 19);
    if (*source & (INT64_C(1) << 44)) *destination |= (INT64_C(1) << 20);
    if (*source & (INT64_C(1) << 45)) *destination |= (INT64_C(1) << 21);
    if (*source & (INT64_C(1) << 46)) *destination |= (INT64_C(1) << 22);
    if (*source & (INT64_C(1) << 47)) *destination |= (INT64_C(1) << 23);
    if (*source & (INT64_C(1) << 48)) *destination |= (INT64_C(1) << 8);
    if (*source & (INT64_C(1) << 49)) *destination |= (INT64_C(1) << 9);
    if (*source & (INT64_C(1) << 50)) *destination |= (INT64_C(1) << 10);
    if (*source & (INT64_C(1) << 51)) *destination |= (INT64_C(1) << 11);
    if (*source & (INT64_C(1) << 52)) *destination |= (INT64_C(1) << 12);
    if (*source & (INT64_C(1) << 53)) *destination |= (INT64_C(1) << 13);
    if (*source & (INT64_C(1) << 54)) *destination |= (INT64_C(1) << 14);
    if (*source & (INT64_C(1) << 55)) *destination |= (INT64_C(1) << 15);
    if (*source & (INT64_C(1) << 56)) *destination |= (INT64_C(1) << 0);
    if (*source & (INT64_C(1) << 57)) *destination |= (INT64_C(1) << 1);
    if (*source & (INT64_C(1) << 58)) *destination |= (INT64_C(1) << 2);
    if (*source & (INT64_C(1) << 59)) *destination |= (INT64_C(1) << 3);
    if (*source & (INT64_C(1) << 60)) *destination |= (INT64_C(1) << 4);
    if (*source & (INT64_C(1) << 61)) *destination |= (INT64_C(1) << 5);
    if (*source & (INT64_C(1) << 62)) *destination |= (INT64_C(1) << 6);
    if (*source & (INT64_C(1) << 63)) *destination |= (INT64_C(1) << 7);
}
```

```cpp
/*
Mike Koch
EECS 4760 - Computer Security
DES
rounds.cpp: Handles executing the 16 Feistel rounds, given the result of the initial permutation and the keys. Stores the result in "output" (provided).
*/
#include "rounds.h"
#include "permutator.h"
#include "sbox.h"

void apply_rounds(uint64_t *initial_permutation, uint64_t *output, uint64_t *keys, Mode mode) {
    // Divide the initial permutation into 2 32-bit blocks
    uint32_t left_half = (*initial_permutation & 0xFFFFFFFF00000000) >> 32;
    uint32_t right_half = *initial_permutation & 0x00000000FFFFFFFF;

    int startIndex = 0;
    int endIndex = 16;
    int incrementer = 1;

    if (mode == Mode::DECRYPTION) {
        // If we're decrypting, we should start at the last key and loop until we get to the first key
        startIndex = 15;
        endIndex = -1;
        incrementer = -1;
    }

    // Rounds
    for (int i = startIndex; i != endIndex; i += incrementer) {
        uint64_t key = keys[i];
        uint32_t original_right_half = right_half;

        // Apply the initial permutation (the expansion permutation)
        uint64_t expanded_right_half;
        apply_expansion_permutation(&right_half, &expanded_right_half);

        // XOR the result of the expansion permutation with the current key
        uint64_t right_and_key = key ^ expanded_right_half;

        // Splitting the result of the XOR operation into eight sets of 6.
        uint8_t six_bit_one = (uint8_t)((right_and_key & 0xFC00000000000000) >> 56);
        uint8_t six_bit_two = (uint8_t)((right_and_key & 0x03F0000000000000) >> 50);
        uint8_t six_bit_three = (uint8_t)((right_and_key & 0x000FC00000000000) >> 44);
        uint8_t six_bit_four = (uint8_t)((right_and_key & 0x00003F0000000000) >> 38);
        uint8_t six_bit_five = (uint8_t)((right_and_key & 0x000000FC00000000) >> 32);
        uint8_t six_bit_six = (uint8_t)((right_and_key & 0x00000003F0000000) >> 26);
        uint8_t six_bit_seven = (uint8_t)((right_and_key & 0x000000000FC00000) >> 20);
        uint8_t six_bit_eight = (uint8_t)((right_and_key & 0x00000000003F0000) >> 14);

        /* The code below is repetitive, and here's what it does:
        1) Get the current row by checking the value of the first and 6th bit
        2) Get the value of bits 2 through 5, and add them accordingly to get a decimal value
        3) Consult the correct S_BOX 2D array to get the value of the row and column.
        */
        // Six bit one, S-Box 1
        int row_number = 0;
        if ((six_bit_one & (1 << 7)) && (six_bit_one & (1 << 2))) row_number = 3;
        if ((six_bit_one & (1 << 7)) && (six_bit_one & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_one & (1 << 7)) == 0 && (six_bit_one & (1 << 2))) row_number = 1;

        uint8_t s_box_one_value;
        int bit_two_value = (six_bit_one & (1 << 6)) ? 8 : 0;
        int bit_three_value = (six_bit_one & (1 << 5)) ? 4 : 0;
        int bit_four_value = (six_bit_one & (1 << 4)) ? 2 : 0;
        int bit_five_value = (six_bit_one & (1 << 3)) ? 1 : 0;
        int sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_one_value = S_BOX_ONE[row_number][sum];

        // Six bit two, S-Box 2
        row_number = 0;
        if ((six_bit_two & (1 << 7)) && (six_bit_two & (1 << 2))) row_number = 3;
        if ((six_bit_two & (1 << 7)) && (six_bit_two & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_two & (1 << 7)) == 0 && (six_bit_two & (1 << 2))) row_number = 1;

        uint8_t s_box_two_value;
        bit_two_value = (six_bit_two & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_two & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_two & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_two & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_two_value = S_BOX_TWO[row_number][sum];

        // Six bit three, S-Box 3
        row_number = 0;
        if ((six_bit_three & (1 << 7)) && (six_bit_three & (1 << 2))) row_number = 3;
        if ((six_bit_three & (1 << 7)) && (six_bit_three & (1 << 2)) == 0)
```

```
          row_number = 2;
        if ((six_bit_three & (1 << 7)) == 0 && (six_bit_three & (1 << 2))) row_number = 1;

        uint8_t s_box_three_value;
        bit_two_value = (six_bit_three & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_three & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_three & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_three & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_three_value = S_BOX_THREE[row_number][sum];

        // Six bit four, S-Box 4
        row_number = 0;
        if ((six_bit_four & (1 << 7)) && (six_bit_four & (1 << 2))) row_number = 3;
        if ((six_bit_four & (1 << 7)) && (six_bit_four & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_four & (1 << 7)) == 0 && (six_bit_four & (1 << 2))) row_number = 1;

        uint8_t s_box_four_value;
        bit_two_value = (six_bit_four & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_four & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_four & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_four & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_four_value = S_BOX_FOUR[row_number][sum];

        // Six bit five, S-Box 5
        row_number = 0;
        if ((six_bit_five & (1 << 7)) && (six_bit_five & (1 << 2))) row_number = 3;
        if ((six_bit_five & (1 << 7)) && (six_bit_five & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_five & (1 << 7)) == 0 && (six_bit_five & (1 << 2))) row_number = 1;

        uint8_t s_box_five_value;
        bit_two_value = (six_bit_five & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_five & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_five & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_five & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_five_value = S_BOX_FIVE[row_number][sum];

        // Six bit six, S-Box 6
        row_number = 0;
        if ((six_bit_six & (1 << 7)) && (six_bit_six & (1 << 2))) row_number = 3;
        if ((six_bit_six & (1 << 7)) && (six_bit_six & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_six & (1 << 7)) == 0 && (six_bit_six & (1 << 2))) row_number
```

```cpp
 = 1;

        uint8_t s_box_six_value;
        bit_two_value = (six_bit_six & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_six & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_six & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_six & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_six_value = S_BOX_SIX[row_number][sum];

        // Six bit seven, S-Box 7
        row_number = 0;
        if ((six_bit_seven & (1 << 7)) && (six_bit_seven & (1 << 2))) row_number = 3;
        if ((six_bit_seven & (1 << 7)) && (six_bit_seven & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_seven & (1 << 7)) == 0 && (six_bit_seven & (1 << 2))) row_number = 1;

        uint8_t s_box_seven_value;
        bit_two_value = (six_bit_seven & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_seven & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_seven & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_seven & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_seven_value = S_BOX_SEVEN[row_number][sum];

        // Six bit eight, S-Box 8
        row_number = 0;
        if ((six_bit_eight & (1 << 7)) && (six_bit_eight & (1 << 2))) row_number = 3;
        if ((six_bit_eight & (1 << 7)) && (six_bit_eight & (1 << 2)) == 0) row_number = 2;
        if ((six_bit_eight & (1 << 7)) == 0 && (six_bit_eight & (1 << 2))) row_number = 1;

        uint8_t s_box_eight_value;
        bit_two_value = (six_bit_eight & (1 << 6)) ? 8 : 0;
        bit_three_value = (six_bit_eight & (1 << 5)) ? 4 : 0;
        bit_four_value = (six_bit_eight & (1 << 4)) ? 2 : 0;
        bit_five_value = (six_bit_eight & (1 << 3)) ? 1 : 0;
        sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
        s_box_eight_value = S_BOX_EIGHT[row_number][sum];

        // Use a straight P box to convert our 32-bit value (8 4-bit S box vlaues) to a new 32-bit value
        // I'm too lazy to convert the S box values into one 32-bit value, so I'm going to treat bit "5" as sbox 2, bit 1, etc.
        uint32_t straight_p_box_result = 0;
        if (s_box_four_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 1));
        if (s_box_two_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32
 - 2));
        if (s_box_five_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 3));
        if (s_box_six_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 4));
        if (s_box_eight_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 5));
        if (s_box_three_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 6));
        if (s_box_seven_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 7));
        if (s_box_five_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 8));
        if (s_box_one_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 9));
        if (s_box_four_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 10));
        if (s_box_six_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 11));
        if (s_box_seven_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 12));
        if (s_box_two_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 13));
        if (s_box_five_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 14));
        if (s_box_eight_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 15));
        if (s_box_three_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 16));
        if (s_box_one_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 17));
        if (s_box_two_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 18));
        if (s_box_six_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 19));
        if (s_box_four_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 20));
        if (s_box_eight_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 21));
        if (s_box_seven_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 22));
        if (s_box_one_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 23));
        if (s_box_three_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 24));
        if (s_box_five_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 25));
        if (s_box_four_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 26));
        if (s_box_eight_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 27));
        if (s_box_two_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32
          - 28));
        if (s_box_six_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 29));
        if (s_box_three_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 30));
        if (s_box_one_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 31));
        if (s_box_seven_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 32));


        // XOR the straight_p_box_result with the left half
        right_half = left_half ^ straight_p_box_result;

        left_half = original_right_half;
    }

    // Since we're done, combine the left and right halves to make our output
    *output = (uint64_t)right_half << 32 | left_half;
}
```