

---

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  error_handler.h: Consult error_handler.cpp
6  */
7  #ifndef ERROR_HANDLER_H
8  #define ERROR_HANDLER_H
9
10 void output_error(char *text, int exit_code);
11 #endif
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  exit_code.h: An enum indicating the type of error that occurred, along with its  ↗
   exit code.
6  */
7  #ifndef EXIT_CODE_H
8  #define EXIT_CODE_H
9
10 // A collection of exit codes, depending on what went wrong
11 enum ExitCode {
12     INVALID_ARG_SYNTAX = 1,
13     INVALID_ENCRYPT_DECRYPT_FLAG = 2,
14     INVALID_PASSWORD = 3,
15     INVALID_MODE = 4,
16     CANNOT_OPEN_FILE = 5,
17
18 };
19 #endif
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  file_processor.h: Consult file_processor.cpp
6  */
7  #ifndef FILE_PROCESSOR_H
8  #define FILE_PROCESSOR_H
9  #include <fstream>
10
11 bool get_next_64_bits(std::fstream &file_stream, uint64_t *destination, int *bytes_read, int file_size);
12 void build_file_size_block(int file_size, uint64_t *block);
13
14 #endif
```

---

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  garbage_producer.h: Consult garbage_producer.cpp
6  */
7  #ifndef GARBAGE_PRODUCER_H
8  #define GARBAGE_PRODUCER_H
9  #include <stdint.h>
10
11 void generate_eight_bytes_of_garbage(uint64_t *destination);
12 #endif
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  keygen.h: Consult keygen.cpp
6  */
7  #ifndef KEYGEN_H
8  #define KEYGEN_H
9  #include <stdint.h>
10
11 // Generates the necessary keys needed and stores them in the key array provided. ↗
12     This also assumes that the provided_key is a valid key.
13 void generate_keys(uint64_t provided_key, uint64_t *keys);
14 #endif
```

---

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  mode.h: An enum to indicate whether or not to encrypt (Mode::ENCRYPTION) or to
        decrypt (Mode::DECRYPTION) the infile.
6  */
7  #ifndef MODE_H
8  #define MODE_H
9
10 enum Mode {
11     ENCRYPTION,
12     DECRYPTION,
13 };
14 #endif
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  permutator.h: Consult permutator.cpp
6  */
7  #ifndef PERMUTATOR_H
8  #define PERMUTATOR_H
9  #include <stdint.h>
10
11 void apply_initial_permutation(uint64_t *source, uint64_t *destination);
12
13 void apply_expansion_permutation(uint32_t *source, uint64_t *destination);
14
15 void apply_final_permutation(uint64_t *source, uint64_t *destination);
16 #endif
```

---

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  reverser.h: Consult reverser.cpp
6  */
7  #ifndef REVERSER_H
8  #define REVERSER_H
9  #include <stdint.h>
10
11 void reverse(uint64_t *source, uint64_t *destination);
12 #endif
```



---

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  rounds.h: Consult rounds.cpp
6  */
7  #ifndef ROUNDS_H
8  #define ROUNDS_H
9  #include <fstream>
10 #include "mode.h"
11
12 void apply_rounds(uint64_t *initial_permutation, uint64_t *output, uint64_t *keys,
13                  Mode mode);
13 #endif
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  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
6          CMPs and jumps)
7  */
8  #ifndef SBOX_H
9  #define SBOX_H
10
11 // The actual S-Boxes for the rounds live here. It is faster to have them declared ↗
        once, rather than having to create them each
12 // time a round is ran.
13 const int S_BOX_ONE[4][16] = {
14     { 14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7 },
15     { 0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8 },
16     { 4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0 },
17     { 15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13 }
18 };
19
20 const int S_BOX_TWO[4][16] = {
21     { 15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10 },
22     { 3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5 },
23     { 0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15 },
24     { 13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9 }
25 };
26
27 const int S_BOX_THREE[4][16] = {
28     { 10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8 },
29     { 13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1 },
30     { 13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7 },
31     { 1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12 }
32 };
33
34 const int S_BOX_FOUR[4][16] = {
35     { 7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15 },
36     { 13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9 },
37     { 10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4 },
38     { 3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14 }
39 };
40
41 const int S_BOX_FIVE[4][16] = {
42     { 2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9 },
43     { 14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6 },
44     { 4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14 },
45     { 11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3 }
46 };
47
48 const int S_BOX_SIX[4][16] = {
49     { 12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11 },
```

```
50     { 10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8 },
51     { 9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6 },
52     { 4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13 }
53 };
54
55 const int S_BOX_SEVEN[4][16] = {
56     { 4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1 },
57     { 13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6 },
58     { 1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2 },
59     { 6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12 }
60 };
61
62 const int S_BOX_EIGHT[4][16] = {
63     { 13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7 },
64     { 1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2 },
65     { 7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8 },
66     { 2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11 }
67 };
68 #endif
```

```
1  /*
2   Mike Koch
3   EECS 4760 - Computer Security
4   DES
5   des.cpp: Main entry point. Handles argument processing and the overall
6   encryption / decryption process.
7  */
8  #include "keygen.h"
9  #include "file_processor.h"
10 #include "permutator.h"
11 #include "rounds.h"
12 #include "reverser.h"
13 #include "garbage_producer.h"
14 #include "error_handler.h"
15 #include "exit_code.h"
16 #include <string.h>
17 #include <time.h>
18 #include <iostream>
19
20 // Constants
21 const int ASCII_ZERO = 48;
22 const int ASCII_NINE = 57;
23 const int ASCII_A = 65;
24 const int ASCII_F = 70;
25 const double NUMBER_OF_MILLISECONDS_IN_SECOND = 1000.0;
26
27 // Function prototypes
28 void process_chunk(uint64_t *next_64_bits, uint64_t *keys, uint64_t *output, Mode mode);
29 void set_ascii_characters_to_key(char most_significant_char, char
30 least_significant_char, int bit_offset, uint64_t *destination);
31
32 int main(int argc, char *argv[])
33 {
34     // 1. Process args
35     // The command-line syntax is: [-[d|D]|-[e|E]] [password] [ECB|ecb] [input
36     // file path] [output file path], where -d is decryption and -e is encryption
37     // If we don't have 5 args (program name + actual args == argc), fail now
38     if (argc != 6) {
39         output_error("", ExitCode::INVALID_ARG_SYNTAX);
40     }
41
42     // Make sure the user provided a valid encrypt/decrypt flag
43     Mode mode;
44     if (std::string(argv[1]) == "-e" || std::string(argv[1]) == "-E") {
45         mode = Mode::ENCRYPTION;
46     }
47     else if (std::string(argv[1]) == "-d" || std::string(argv[1]) == "-D") {
48         mode = Mode::DECRYPTION;
49     }
50     else {
51         output_error("Invalid encryption/decryption flag. Must choose either '-"
```

```
e'/'-E' for encrypt or '-d'/'-D' for decrypt",
ExitCode::INVALID_ENCRYPT_DECRYPT_FLAG);

49     }
50
51     // Key parsing
52     char *raw_password = argv[2];
53     uint64_t password = 0;
54
55     // 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
56     if (raw_password[0] == '\\') {
57         // 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.
58         if (strlen(raw_password) != 10) {
59             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);
60         }
61
62         // The password was entered in as a string. Set each byte to the password
    uint
63         password |= (uint64_t)raw_password[1] << 56;
64         password |= (uint64_t)raw_password[2] << 48;
65         password |= (uint64_t)raw_password[3] << 40;
66         password |= (uint64_t)raw_password[4] << 32;
67         password |= (uint64_t)raw_password[5] << 24;
68         password |= (uint64_t)raw_password[6] << 16;
69         password |= (uint64_t)raw_password[7] << 8;
70         password |= (uint64_t)raw_password[8];
71     }
72     else {
73         // The password was entered as hex. We'll need to look at each character
    and convert it to its hex value
74         // Make sure we have 16 hex characters. If we don't, fail.
75         if (strlen(raw_password) != 16) {
76             output_error("Hex passwords must be *exactly* sixteen hex characters
    between 0-F. Exiting.", ExitCode::INVALID_PASSWORD);
77         }
78
79
80         // The hex characters provided are entered in order from MSB to LSB
81         set_ascii_characters_to_key(raw_password[0], raw_password[1], 56,
    &password);
82         set_ascii_characters_to_key(raw_password[2], raw_password[3], 48,
    &password);
83         set_ascii_characters_to_key(raw_password[4], raw_password[5], 40,
    &password);
84         set_ascii_characters_to_key(raw_password[6], raw_password[7], 32,
    &password);
85         set_ascii_characters_to_key(raw_password[8], raw_password[9], 24,
    &password);
```

```

C:\Sandbox\School Projects\Fall-2016\DES\DES\des.cpp 3
86     set_ascii_characters_to_key(raw_password[10], raw_password[11], 16,  ↗
    &password);
87     set_ascii_characters_to_key(raw_password[12], raw_password[13], 8,  ↗
    &password);
88     set_ascii_characters_to_key(raw_password[14], raw_password[15], 0,  ↗
    &password);
89 }
90
91 // This implementation only supports ECB. If anything else is provided, fail  ↗
    now
92 if ((argv[3][0] != 'e' && argv[3][0] != 'E')
93     || (argv[3][1] != 'c' && argv[3][1] != 'C')
94     || (argv[3][2] != 'b' && argv[3][2] != 'B')) {
95     output_error("Only ECB is supported!", ExitCode::INVALID_MODE);
96 }
97
98 char *input_file_path = argv[4];
99 char *output_file_path = argv[5];
100
101 printf("Starting the %s process. Please wait.\n", mode == Mode::ENCRYPTION ?  ↗
    "encryption" : "decryption");
102
103 // 2. Generate keys
104 // Start a timer here so we can see how long the process takes.
105 clock_t startTime = clock();
106 uint64_t keys[16];
107 generate_keys(password, keys);
108
109 // Open the file for processing
110 std::fstream input_stream;
111 std::fstream output_stream;
112 input_stream.open(input_file_path, std::ios::in | std::ios::binary);
113
114 // Get the length of the file by going to the end of the input stream and  ↗
    checking where the stream index is at. Then go back
115 // to the beginning so we can actually read the data
116 input_stream.seekg(0, input_stream.end);
117 int length_of_file = input_stream.tellg();
118 input_stream.seekg(0, input_stream.beg);
119
120 output_stream.open(output_file_path, std::ios::out | std::ios::binary |  ↗
    std::ios::trunc);
121 if (!input_stream.is_open() || !output_stream.is_open()) {
122     output_error("ERROR: Unable to open file. Make sure that both the input  ↗
        file and output file are accessible!", ExitCode::CANNOT_OPEN_FILE);
123 }
124
125 // If we're encrypting, encrypt the file size.
126 if (mode == Mode::ENCRYPTION) {
127     uint64_t file_size_block;
128     build_file_size_block(length_of_file, &file_size_block);
129

```

```
130     uint64_t output = 0;
131     process_chunk(&file_size_block, keys, &output, mode);
132
133     // Reverse the output so the bytes are saved from MSB to LSB
134     uint64_t reversed_output;
135     reverse(&output, &reversed_output);
136
137     output_stream.write((char*)&reversed_output, sizeof(uint64_t));
138 }
139
140 // Flag to indicate if we've reached the end of the input file.
141 bool keep_going = false;
142 int number_of_bytes_read = 0;
143
144 // These two ints are used only for decryption. We only strip half of the
145 // first chunk during decryption, so we need to know when the first run is. We
146 // also need to know the file size.
147 int first_iteration = true;
148 int decrypted_file_size;
149 do {
150     // 3. Get next 64 bits (8 bytes)
151     uint64_t next_64_bits = 0;
152     int previous_number_of_bytes_read = number_of_bytes_read;
153     keep_going = get_next_64_bits(input_stream, &next_64_bits,
154     &number_of_bytes_read, length_of_file);
155
156     // If we fall into this block, this means our last read was (1) our final
157     // possible read, and (2) we read less than 8 bytes. We need to pad the
158     // rest of the bytes with garbage.
159     // This will ONLY ever happen when encrypting. If we're decrypting and
160     // this if statement is true, very bad things have happened.
161     if (!keep_going && number_of_bytes_read - previous_number_of_bytes_read !=
162     8) {
163         if (mode == Mode::DECRYPTION) {
164             printf("Hold up there! We shouldn't have a block that is not
165             exactly 64 bits long. Stopping.");
166             exit(1);
167         }
168
169         uint64_t garbage;
170         generate_eight_bytes_of_garbage(&garbage);
171         int number_of_bytes_filled = number_of_bytes_read -
172         previous_number_of_bytes_read;
173
174         // Now that we know how many bytes need garbage, push the garbage
175         // data into those bytes by shifting them to the left.
176         // *We haven't reversed yet, so the garbage will be the most
177         // significant byte(s)*
178         next_64_bits |= garbage << (8 * number_of_bytes_filled);
179     }
180
181     // The bytes read in are stored in reverse order. We need to flip them
```

```
        around first.
171     uint64_t reversed_next_64_bits;
172     reverse(&next_64_bits, &reversed_next_64_bits);
173
174     uint64_t output = 0;
175     process_chunk(&reversed_next_64_bits, keys, &output, mode);
176
177     if (first_iteration && mode == Mode::DECRYPTION) {
178         // The first chunk tells us how long the file is. We'll store this in
179         // decrypted_file_size so we know if we need to throw away any
180         // garbage bits.
181         // ANDing the output with the below hex value to only retrieve the
182         // least-significant word; our file length
183         decrypted_file_size = output & 0x00000000FFFFFFFF;
184
185         first_iteration = false;
186     }
187     else {
188         // Output to file
189         uint64_t reversed_output;
190         reverse(&output, &reversed_output);
191
192         // We need to make sure that this isn't the last block during
193         // decryption. If we are on the last block, and there's garbage data,
194         // we must strip it out first.
195         int output_size = sizeof(uint64_t);
196         if (mode == Mode::DECRYPTION && decrypted_file_size <
197             (number_of_bytes_read - 8)) {
198             int number_of_extra_bytes = (number_of_bytes_read - 8) -
199                 decrypted_file_size;
200
201             if (number_of_extra_bytes == 1) reversed_output &= ~
202                 (0xFF00000000000000);
203             if (number_of_extra_bytes == 2) reversed_output &= ~
204                 (0xFFFF000000000000);
205             if (number_of_extra_bytes == 3) reversed_output &= ~
206                 (0xFFFFFFFF00000000);
207             if (number_of_extra_bytes == 4) reversed_output &= ~
208                 (0xFFFFFFFFFF000000);
209             if (number_of_extra_bytes == 5) reversed_output &= ~
210                 (0xFFFFFFFFFFFF0000);
211             if (number_of_extra_bytes == 6) reversed_output &= ~
212                 (0xFFFFFFFFFFFFFF00);
213             if (number_of_extra_bytes == 7) reversed_output &= ~
214                 (0xFFFFFFFFFFFFFFFF);
215
216             output_size -= number_of_extra_bytes;
217         }
218
219         output_stream.write((char*)&reversed_output, output_size);
220     }
221 } while (keep_going);
```



```
208
209     // Everything is done. Get the end time and output how long the process took.
210     clock_t endTime = clock();
211     printf("Process completed in %.4f seconds. Press ENTER to exit.", (endTime - ↗
        startTime)/ NUMBER_OF_MILLISECONDS_IN_SECOND);
212     std::cin.get();
213 }
214
215 void process_chunk(uint64_t *next_64_bits, uint64_t *keys, uint64_t *output, Mode ↗
    mode) {
216     // Initial permutation
217     uint64_t initial_permutation;
218     apply_initial_permutation(next_64_bits, &initial_permutation);
219
220     // Rounds 1 - 16
221     uint64_t round_output;
222     apply_rounds(&initial_permutation, &round_output, keys, mode);
223
224     // Final Permutation
225     apply_final_permutation(&round_output, output);
226 }
227
228 void set_ascii_characters_to_key(char most_significant_char, char ↗
    least_significant_char, int bit_offset, uint64_t *destination) {
229     int left_hand_side = most_significant_char - '0';
230     int right_hand_side = least_significant_char - '0';
231
232     // Make sure the user entered a valid hex digit
233     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)) ||
234         (right_hand_side < (ASCII_ZERO - 48) || (right_hand_side > (ASCII_NINE - ↗
        48) && right_hand_side < (ASCII_A - 48)) || right_hand_side > (ASCII_F - ↗
        48))) {
235         output_error("One or more characters are not valid hex characters", ↗
            ExitCode::INVALID_PASSWORD);
236     }
237
238     *destination |= (uint64_t)((left_hand_side * 16) + right_hand_side) << ↗
        bit_offset;
239 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  error_handler.cpp: Handles outputting errors to the console (displaying the
    provided text) and exiting with the provided exit code
6  */
7  #include "error_handler.h"
8  #include <stdio.h>
9  #include <iostream>
10
11 // Handle outputting an error message, as well as holding the application from
    exiting until input from the keyboard is received.
12 // Then the application exits with the provided exit_code.
13 void output_error(char *text, int exit_code) {
14     printf("ERROR: %s", text);
15     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");
16     printf("\n\nPress ENTER to exit.");
17
18     std::cin.get();
19
20     exit(exit_code);
21 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  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.
6
    Also handles building the file size block to be stored in the
    encrypted file.
7  */
8  #include "file_processor.h"
9  #include "garbage_producer.h"
10
11 bool get_next_64_bits(std::fstream &file_stream, uint64_t *destination, int
    *bytes_read, int file_size) {
12     int number_of_bytes_to_read = 8;
13     if (*bytes_read + number_of_bytes_to_read > file_size) {
14         number_of_bytes_to_read = file_size - *bytes_read;
15     }
16
17     file_stream.read((char *)destination, number_of_bytes_to_read);
18     *bytes_read += number_of_bytes_to_read;
19
20     return *bytes_read != file_size;
21 }
22
23 void build_file_size_block(int file_size, uint64_t *block) {
24     *block = 0;
25
26     // The first 32 bits (left-hand side) are garbage
27     uint64_t garbage;
28     generate_eight_bytes_of_garbage(&garbage);
29
30     // Clear the least significant word so we can put the file size there
31     *block |= garbage << 32;
32
33     // The next 32 bits are the file size
34     *block |= file_size;
35 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  garbage_producer.cpp: Generates 64 random bits to be used by other functions
6  */
7  #include "garbage_producer.h"
8  #include <stdlib.h>
9
10 // Generates eight bytes of garbage data. Why eight bytes?
11 // 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.
12 void generate_eight_bytes_of_garbage(uint64_t *destination) {
13     for (int i = 0; i < 64; i++) {
14         if (rand() % 2) {
15             *destination |= (INT64_C(1) << i);
16         }
17     }
18 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  keygen.cpp: Generates the 16 keys needed for each Feistel round, based on the key ↗
   provided. Handles the compression permutations and rotations
6  */
7  #include "keygen.h"
8  #include <stdio.h>
9
10 const int NUMBER_OF_KEYS_TO_GENERATE = 16;
11
12 // Function prototypes
13 void compress_original_key(uint64_t provided_key, uint64_t *compressed_key);
14 uint32_t rotate_left(uint32_t original, int number_of_bits_to_rotate);
15 void compress_rotated_key(uint64_t *combined_key, uint64_t *compressed_key);
16
17 void generate_keys(uint64_t provided_key, uint64_t *keys)
18 {
19     uint64_t compressed_key = 0;
20     compress_original_key(provided_key, &compressed_key);
21
22     // Split the 56-bit key into two chunks, both 28 bits long.
23     uint32_t original_left_side = (uint32_t)((compressed_key & 0xFFFFFFFF00000000) >> 32);
24     uint32_t original_right_side = (uint32_t)((compressed_key & 0x0000000FFFFFFFFF) >> 4);
25
26     for (int i = 0; i < NUMBER_OF_KEYS_TO_GENERATE; i++) {
27         int rotation_amount = (i == 0 || i == 1 || i == 8 || i == 15)
28             ? 1
29             : 2;
30
31         uint32_t left_side_rotated = rotate_left(original_left_side, rotation_amount);
32         uint32_t right_side_rotated = rotate_left(original_right_side, rotation_amount);
33
34         uint64_t combined_key = 0;
35
36         //-- Taking our two halves, combine them into the combined key
37         if (left_side_rotated & (1 << (32 - 1))) combined_key |= (INT64_C(1) << (64 - 1));
38         if (left_side_rotated & (1 << (32 - 2))) combined_key |= (INT64_C(1) << (64 - 2));
39         if (left_side_rotated & (1 << (32 - 3))) combined_key |= (INT64_C(1) << (64 - 3));
40         if (left_side_rotated & (1 << (32 - 4))) combined_key |= (INT64_C(1) << (64 - 4));
41         if (left_side_rotated & (1 << (32 - 5))) combined_key |= (INT64_C(1) << (64 - 5));
42         if (left_side_rotated & (1 << (32 - 6))) combined_key |= (INT64_C(1) <<
```

```
(64 - 6));  
43     if (left_side_rotated & (1 << (32 - 7))) combined_key |= (INT64_C(1) << 7) & (64 - 7));  
44     if (left_side_rotated & (1 << (32 - 8))) combined_key |= (INT64_C(1) << 8) & (64 - 8));  
45     if (left_side_rotated & (1 << (32 - 9))) combined_key |= (INT64_C(1) << 9) & (64 - 9));  
46     if (left_side_rotated & (1 << (32 - 10))) combined_key |= (INT64_C(1) << 10) & (64 - 10));  
47     if (left_side_rotated & (1 << (32 - 11))) combined_key |= (INT64_C(1) << 11) & (64 - 11));  
48     if (left_side_rotated & (1 << (32 - 12))) combined_key |= (INT64_C(1) << 12) & (64 - 12));  
49     if (left_side_rotated & (1 << (32 - 13))) combined_key |= (INT64_C(1) << 13) & (64 - 13));  
50     if (left_side_rotated & (1 << (32 - 14))) combined_key |= (INT64_C(1) << 14) & (64 - 14));  
51     if (left_side_rotated & (1 << (32 - 15))) combined_key |= (INT64_C(1) << 15) & (64 - 15));  
52     if (left_side_rotated & (1 << (32 - 16))) combined_key |= (INT64_C(1) << 16) & (64 - 16));  
53     if (left_side_rotated & (1 << (32 - 17))) combined_key |= (INT64_C(1) << 17) & (64 - 17));  
54     if (left_side_rotated & (1 << (32 - 18))) combined_key |= (INT64_C(1) << 18) & (64 - 18));  
55     if (left_side_rotated & (1 << (32 - 19))) combined_key |= (INT64_C(1) << 19) & (64 - 19));  
56     if (left_side_rotated & (1 << (32 - 20))) combined_key |= (INT64_C(1) << 20) & (64 - 20));  
57     if (left_side_rotated & (1 << (32 - 21))) combined_key |= (INT64_C(1) << 21) & (64 - 21));  
58     if (left_side_rotated & (1 << (32 - 22))) combined_key |= (INT64_C(1) << 22) & (64 - 22));  
59     if (left_side_rotated & (1 << (32 - 23))) combined_key |= (INT64_C(1) << 23) & (64 - 23));  
60     if (left_side_rotated & (1 << (32 - 24))) combined_key |= (INT64_C(1) << 24) & (64 - 24));  
61     if (left_side_rotated & (1 << (32 - 25))) combined_key |= (INT64_C(1) << 25) & (64 - 25));  
62     if (left_side_rotated & (1 << (32 - 26))) combined_key |= (INT64_C(1) << 26) & (64 - 26));  
63     if (left_side_rotated & (1 << (32 - 27))) combined_key |= (INT64_C(1) << 27) & (64 - 27));  
64     if (left_side_rotated & (1 << (32 - 28))) combined_key |= (INT64_C(1) << 28) & (64 - 28));  
65     if (right_side_rotated & (1 << (32 - 1))) combined_key |= (INT64_C(1) << 1) & (64 - 29));  
66     if (right_side_rotated & (1 << (32 - 2))) combined_key |= (INT64_C(1) << 2) & (64 - 30));  
67     if (right_side_rotated & (1 << (32 - 3))) combined_key |= (INT64_C(1) << 3) & (64 - 31));  
68     if (right_side_rotated & (1 << (32 - 4))) combined_key |= (INT64_C(1) << 4) & (64 - 32);
```

```
(64 - 32));
69     if (right_side_rotated & (1 << (32 - 5))) combined_key |= (INT64_C(1) <<
(64 - 33));
70     if (right_side_rotated & (1 << (32 - 6))) combined_key |= (INT64_C(1) <<
(64 - 34));
71     if (right_side_rotated & (1 << (32 - 7))) combined_key |= (INT64_C(1) <<
(64 - 35));
72     if (right_side_rotated & (1 << (32 - 8))) combined_key |= (INT64_C(1) <<
(64 - 36));
73     if (right_side_rotated & (1 << (32 - 9))) combined_key |= (INT64_C(1) <<
(64 - 37));
74     if (right_side_rotated & (1 << (32 - 10))) combined_key |= (INT64_C(1) <<
(64 - 38));
75     if (right_side_rotated & (1 << (32 - 11))) combined_key |= (INT64_C(1) <<
(64 - 39));
76     if (right_side_rotated & (1 << (32 - 12))) combined_key |= (INT64_C(1) <<
(64 - 40));
77     if (right_side_rotated & (1 << (32 - 13))) combined_key |= (INT64_C(1) <<
(64 - 41));
78     if (right_side_rotated & (1 << (32 - 14))) combined_key |= (INT64_C(1) <<
(64 - 42));
79     if (right_side_rotated & (1 << (32 - 15))) combined_key |= (INT64_C(1) <<
(64 - 43));
80     if (right_side_rotated & (1 << (32 - 16))) combined_key |= (INT64_C(1) <<
(64 - 44));
81     if (right_side_rotated & (1 << (32 - 17))) combined_key |= (INT64_C(1) <<
(64 - 45));
82     if (right_side_rotated & (1 << (32 - 18))) combined_key |= (INT64_C(1) <<
(64 - 46));
83     if (right_side_rotated & (1 << (32 - 19))) combined_key |= (INT64_C(1) <<
(64 - 47));
84     if (right_side_rotated & (1 << (32 - 20))) combined_key |= (INT64_C(1) <<
(64 - 48));
85     if (right_side_rotated & (1 << (32 - 21))) combined_key |= (INT64_C(1) <<
(64 - 49));
86     if (right_side_rotated & (1 << (32 - 22))) combined_key |= (INT64_C(1) <<
(64 - 50));
87     if (right_side_rotated & (1 << (32 - 23))) combined_key |= (INT64_C(1) <<
(64 - 51));
88     if (right_side_rotated & (1 << (32 - 24))) combined_key |= (INT64_C(1) <<
(64 - 52));
89     if (right_side_rotated & (1 << (32 - 25))) combined_key |= (INT64_C(1) <<
(64 - 53));
90     if (right_side_rotated & (1 << (32 - 26))) combined_key |= (INT64_C(1) <<
(64 - 54));
91     if (right_side_rotated & (1 << (32 - 27))) combined_key |= (INT64_C(1) <<
(64 - 55));
92     if (right_side_rotated & (1 << (32 - 28))) combined_key |= (INT64_C(1) <<
(64 - 56));
93
94
95
```

```
96     //-- Pass our combined key into another P-Box to strip this to 48 bits
97     uint64_t permutated_key;
98     compress_rotated_key(&combined_key, &permutated_key);
99
100    //-- store in keys array
101    keys[i] = permutated_key;
102
103    //-- Our "new" 56 bit key is the one we rotated earlier
104    original_left_side = left_side_rotated;
105    original_right_side = right_side_rotated;
106 }
107 }
108
109 void compress_original_key(uint64_t provided_key, uint64_t *compressed_key) {
110     //-- Compression P-Box: 64 -> 56 bits
111     if (provided_key & (INT64_C(1) << (64 - 57))) *compressed_key |= (INT64_C(1) << (64 - 1));
112     if (provided_key & (INT64_C(1) << (64 - 49))) *compressed_key |= (INT64_C(1) << (64 - 2));
113     if (provided_key & (INT64_C(1) << (64 - 41))) *compressed_key |= (INT64_C(1) << (64 - 3));
114     if (provided_key & (INT64_C(1) << (64 - 33))) *compressed_key |= (INT64_C(1) << (64 - 4));
115     if (provided_key & (INT64_C(1) << (64 - 25))) *compressed_key |= (INT64_C(1) << (64 - 5));
116     if (provided_key & (INT64_C(1) << (64 - 17))) *compressed_key |= (INT64_C(1) << (64 - 6));
117     if (provided_key & (INT64_C(1) << (64 - 9))) *compressed_key |= (INT64_C(1) << (64 - 7));
118     if (provided_key & (INT64_C(1) << (64 - 1))) *compressed_key |= (INT64_C(1) << (64 - 8));
119     if (provided_key & (INT64_C(1) << (64 - 58))) *compressed_key |= (INT64_C(1) << (64 - 9));
120     if (provided_key & (INT64_C(1) << (64 - 50))) *compressed_key |= (INT64_C(1) << (64 - 10));
121     if (provided_key & (INT64_C(1) << (64 - 42))) *compressed_key |= (INT64_C(1) << (64 - 11));
122     if (provided_key & (INT64_C(1) << (64 - 34))) *compressed_key |= (INT64_C(1) << (64 - 12));
123     if (provided_key & (INT64_C(1) << (64 - 26))) *compressed_key |= (INT64_C(1) << (64 - 13));
124     if (provided_key & (INT64_C(1) << (64 - 18))) *compressed_key |= (INT64_C(1) << (64 - 14));
125     if (provided_key & (INT64_C(1) << (64 - 10))) *compressed_key |= (INT64_C(1) << (64 - 15));
126     if (provided_key & (INT64_C(1) << (64 - 2))) *compressed_key |= (INT64_C(1) << (64 - 16));
127     if (provided_key & (INT64_C(1) << (64 - 59))) *compressed_key |= (INT64_C(1) << (64 - 17));
128     if (provided_key & (INT64_C(1) << (64 - 51))) *compressed_key |= (INT64_C(1) << (64 - 18));
129     if (provided_key & (INT64_C(1) << (64 - 43))) *compressed_key |= (INT64_C(1) << (64 - 19));
```



```
<< (64 - 19));
130 if (provided_key & (INT64_C(1) << (64 - 35))) *compressed_key |= (INT64_C(1) << (64 - 20));
131 if (provided_key & (INT64_C(1) << (64 - 27))) *compressed_key |= (INT64_C(1) << (64 - 21));
132 if (provided_key & (INT64_C(1) << (64 - 19))) *compressed_key |= (INT64_C(1) << (64 - 22));
133 if (provided_key & (INT64_C(1) << (64 - 11))) *compressed_key |= (INT64_C(1) << (64 - 23));
134 if (provided_key & (INT64_C(1) << (64 - 3))) *compressed_key |= (INT64_C(1) << (64 - 24));
135 if (provided_key & (INT64_C(1) << (64 - 60))) *compressed_key |= (INT64_C(1) << (64 - 25));
136 if (provided_key & (INT64_C(1) << (64 - 52))) *compressed_key |= (INT64_C(1) << (64 - 26));
137 if (provided_key & (INT64_C(1) << (64 - 44))) *compressed_key |= (INT64_C(1) << (64 - 27));
138 if (provided_key & (INT64_C(1) << (64 - 36))) *compressed_key |= (INT64_C(1) << (64 - 28));
139 if (provided_key & (INT64_C(1) << (64 - 63))) *compressed_key |= (INT64_C(1) << (64 - 29));
140 if (provided_key & (INT64_C(1) << (64 - 55))) *compressed_key |= (INT64_C(1) << (64 - 30));
141 if (provided_key & (INT64_C(1) << (64 - 47))) *compressed_key |= (INT64_C(1) << (64 - 31));
142 if (provided_key & (INT64_C(1) << (64 - 39))) *compressed_key |= (INT64_C(1) << (64 - 32));
143 if (provided_key & (INT64_C(1) << (64 - 31))) *compressed_key |= (INT64_C(1) << (64 - 33));
144 if (provided_key & (INT64_C(1) << (64 - 23))) *compressed_key |= (INT64_C(1) << (64 - 34));
145 if (provided_key & (INT64_C(1) << (64 - 15))) *compressed_key |= (INT64_C(1) << (64 - 35));
146 if (provided_key & (INT64_C(1) << (64 - 7))) *compressed_key |= (INT64_C(1) << (64 - 36));
147 if (provided_key & (INT64_C(1) << (64 - 62))) *compressed_key |= (INT64_C(1) << (64 - 37));
148 if (provided_key & (INT64_C(1) << (64 - 54))) *compressed_key |= (INT64_C(1) << (64 - 38));
149 if (provided_key & (INT64_C(1) << (64 - 46))) *compressed_key |= (INT64_C(1) << (64 - 39));
150 if (provided_key & (INT64_C(1) << (64 - 38))) *compressed_key |= (INT64_C(1) << (64 - 40));
151 if (provided_key & (INT64_C(1) << (64 - 30))) *compressed_key |= (INT64_C(1) << (64 - 41));
152 if (provided_key & (INT64_C(1) << (64 - 22))) *compressed_key |= (INT64_C(1) << (64 - 42));
153 if (provided_key & (INT64_C(1) << (64 - 14))) *compressed_key |= (INT64_C(1) << (64 - 43));
154 if (provided_key & (INT64_C(1) << (64 - 6))) *compressed_key |= (INT64_C(1) << (64 - 44));
155 if (provided_key & (INT64_C(1) << (64 - 61))) *compressed_key |= (INT64_C(1) << (64 - 45));
```

```

    << (64 - 45));
156     if (provided_key & (INT64_C(1) << (64 - 53))) *compressed_key |= (INT64_C(1) << (64 - 46));
157     if (provided_key & (INT64_C(1) << (64 - 45))) *compressed_key |= (INT64_C(1) << (64 - 47));
158     if (provided_key & (INT64_C(1) << (64 - 37))) *compressed_key |= (INT64_C(1) << (64 - 48));
159     if (provided_key & (INT64_C(1) << (64 - 29))) *compressed_key |= (INT64_C(1) << (64 - 49));
160     if (provided_key & (INT64_C(1) << (64 - 21))) *compressed_key |= (INT64_C(1) << (64 - 50));
161     if (provided_key & (INT64_C(1) << (64 - 13))) *compressed_key |= (INT64_C(1) << (64 - 51));
162     if (provided_key & (INT64_C(1) << (64 - 5))) *compressed_key |= (INT64_C(1) << (64 - 52));
163     if (provided_key & (INT64_C(1) << (64 - 28))) *compressed_key |= (INT64_C(1) << (64 - 53));
164     if (provided_key & (INT64_C(1) << (64 - 20))) *compressed_key |= (INT64_C(1) << (64 - 54));
165     if (provided_key & (INT64_C(1) << (64 - 12))) *compressed_key |= (INT64_C(1) << (64 - 55));
166     if (provided_key & (INT64_C(1) << (64 - 4))) *compressed_key |= (INT64_C(1) << (64 - 56));
167 }
168
169 /* Helper methods to rotate our 28 bit value left or right:
170
171     The first part of the rotate blindly rotates everything to the left (or
172     right) by the number of bits specified (in our case, 1 or 2)
173     Then, because we lost at least 1 bit during the rotate (if left-rotate, the
174     MS bits; if right-rotate, the LS bits), we rotate right by the number of
175     bits lost to push them to the other end
176 */
177 uint32_t rotate_left(uint32_t original, int number_of_bits_to_rotate) {
178     uint32_t rotated = ((original << number_of_bits_to_rotate) | (original >> (28 -
179         number_of_bits_to_rotate)));
180
181     // Clear bits 29-32 just to be safe
182     rotated &= ~(1 << (32 - 29));
183     rotated &= ~(1 << (32 - 30));
184     rotated &= ~(1 << (32 - 31));
185     rotated &= ~(1 << (32 - 32));
186     return rotated;
187 }
188
189 void compress_rotated_key(uint64_t *combined_key, uint64_t *compressed_key) {
190     *compressed_key = 0;
191
192     if (*combined_key & (INT64_C(1) << (64 - 14))) *compressed_key |= (INT64_C(1) << (64 - 1));
193     if (*combined_key & (INT64_C(1) << (64 - 17))) *compressed_key |= (INT64_C(1) << (64 - 2));

```

```
190     if (*combined_key & (INT64_C(1) << (64 - 11))) *compressed_key |= (INT64_C(1) << (64 - 3));
191     if (*combined_key & (INT64_C(1) << (64 - 24))) *compressed_key |= (INT64_C(1) << (64 - 4));
192     if (*combined_key & (INT64_C(1) << (64 - 1))) *compressed_key |= (INT64_C(1) << (64 - 5));
193     if (*combined_key & (INT64_C(1) << (64 - 5))) *compressed_key |= (INT64_C(1) << (64 - 6));
194     if (*combined_key & (INT64_C(1) << (64 - 3))) *compressed_key |= (INT64_C(1) << (64 - 7));
195     if (*combined_key & (INT64_C(1) << (64 - 28))) *compressed_key |= (INT64_C(1) << (64 - 8));
196     if (*combined_key & (INT64_C(1) << (64 - 15))) *compressed_key |= (INT64_C(1) << (64 - 9));
197     if (*combined_key & (INT64_C(1) << (64 - 6))) *compressed_key |= (INT64_C(1) << (64 - 10));
198     if (*combined_key & (INT64_C(1) << (64 - 21))) *compressed_key |= (INT64_C(1) << (64 - 11));
199     if (*combined_key & (INT64_C(1) << (64 - 10))) *compressed_key |= (INT64_C(1) << (64 - 12));
200     if (*combined_key & (INT64_C(1) << (64 - 23))) *compressed_key |= (INT64_C(1) << (64 - 13));
201     if (*combined_key & (INT64_C(1) << (64 - 19))) *compressed_key |= (INT64_C(1) << (64 - 14));
202     if (*combined_key & (INT64_C(1) << (64 - 12))) *compressed_key |= (INT64_C(1) << (64 - 15));
203     if (*combined_key & (INT64_C(1) << (64 - 4))) *compressed_key |= (INT64_C(1) << (64 - 16));
204     if (*combined_key & (INT64_C(1) << (64 - 26))) *compressed_key |= (INT64_C(1) << (64 - 17));
205     if (*combined_key & (INT64_C(1) << (64 - 8))) *compressed_key |= (INT64_C(1) << (64 - 18));
206     if (*combined_key & (INT64_C(1) << (64 - 16))) *compressed_key |= (INT64_C(1) << (64 - 19));
207     if (*combined_key & (INT64_C(1) << (64 - 7))) *compressed_key |= (INT64_C(1) << (64 - 20));
208     if (*combined_key & (INT64_C(1) << (64 - 27))) *compressed_key |= (INT64_C(1) << (64 - 21));
209     if (*combined_key & (INT64_C(1) << (64 - 20))) *compressed_key |= (INT64_C(1) << (64 - 22));
210     if (*combined_key & (INT64_C(1) << (64 - 13))) *compressed_key |= (INT64_C(1) << (64 - 23));
211     if (*combined_key & (INT64_C(1) << (64 - 2))) *compressed_key |= (INT64_C(1) << (64 - 24));
212     if (*combined_key & (INT64_C(1) << (64 - 41))) *compressed_key |= (INT64_C(1) << (64 - 25));
213     if (*combined_key & (INT64_C(1) << (64 - 52))) *compressed_key |= (INT64_C(1) << (64 - 26));
214     if (*combined_key & (INT64_C(1) << (64 - 31))) *compressed_key |= (INT64_C(1) << (64 - 27));
215     if (*combined_key & (INT64_C(1) << (64 - 37))) *compressed_key |= (INT64_C(1) << (64 - 28));
```

```
216     if (*combined_key & (INT64_C(1) << (64 - 47))) *compressed_key |= (INT64_C(1) << (64 - 29));
217     if (*combined_key & (INT64_C(1) << (64 - 55))) *compressed_key |= (INT64_C(1) << (64 - 30));
218     if (*combined_key & (INT64_C(1) << (64 - 30))) *compressed_key |= (INT64_C(1) << (64 - 31));
219     if (*combined_key & (INT64_C(1) << (64 - 40))) *compressed_key |= (INT64_C(1) << (64 - 32));
220     if (*combined_key & (INT64_C(1) << (64 - 51))) *compressed_key |= (INT64_C(1) << (64 - 33));
221     if (*combined_key & (INT64_C(1) << (64 - 45))) *compressed_key |= (INT64_C(1) << (64 - 34));
222     if (*combined_key & (INT64_C(1) << (64 - 33))) *compressed_key |= (INT64_C(1) << (64 - 35));
223     if (*combined_key & (INT64_C(1) << (64 - 48))) *compressed_key |= (INT64_C(1) << (64 - 36));
224     if (*combined_key & (INT64_C(1) << (64 - 44))) *compressed_key |= (INT64_C(1) << (64 - 37));
225     if (*combined_key & (INT64_C(1) << (64 - 49))) *compressed_key |= (INT64_C(1) << (64 - 38));
226     if (*combined_key & (INT64_C(1) << (64 - 39))) *compressed_key |= (INT64_C(1) << (64 - 39));
227     if (*combined_key & (INT64_C(1) << (64 - 56))) *compressed_key |= (INT64_C(1) << (64 - 40));
228     if (*combined_key & (INT64_C(1) << (64 - 34))) *compressed_key |= (INT64_C(1) << (64 - 41));
229     if (*combined_key & (INT64_C(1) << (64 - 53))) *compressed_key |= (INT64_C(1) << (64 - 42));
230     if (*combined_key & (INT64_C(1) << (64 - 46))) *compressed_key |= (INT64_C(1) << (64 - 43));
231     if (*combined_key & (INT64_C(1) << (64 - 42))) *compressed_key |= (INT64_C(1) << (64 - 44));
232     if (*combined_key & (INT64_C(1) << (64 - 50))) *compressed_key |= (INT64_C(1) << (64 - 45));
233     if (*combined_key & (INT64_C(1) << (64 - 36))) *compressed_key |= (INT64_C(1) << (64 - 46));
234     if (*combined_key & (INT64_C(1) << (64 - 29))) *compressed_key |= (INT64_C(1) << (64 - 47));
235     if (*combined_key & (INT64_C(1) << (64 - 32))) *compressed_key |= (INT64_C(1) << (64 - 48));
236 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  permutator.cpp: Handles the initial, expansion, and final permutations. Stores
   the result in the "destination"
6  */
7  #include "permutator.h"
8  #include <stdio.h>
9
10 void apply_initial_permutation(uint64_t *source, uint64_t *destination) {
11     /*
12     Applies the following permutation based on the following chart:
13
14             58 50 42 34 26 18 10 02
15             60 52 44 36 28 20 12 04
16             62 54 46 38 30 22 14 06
17             64 56 48 40 32 24 16 08
18             57 49 41 33 25 17 09 01
19             59 51 43 35 27 19 11 03
20             61 53 45 37 29 21 13 05
21             63 55 47 39 31 23 15 07
22     */
23     *destination = 0;
24     // 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.
25     if (*source & (INT64_C(1) << (64 - 58))) *destination |= (INT64_C(1) << (64 -
26     1));
27     if (*source & (INT64_C(1) << (64 - 50))) *destination |= (INT64_C(1) << (64 -
28     2));
29     if (*source & (INT64_C(1) << (64 - 42))) *destination |= (INT64_C(1) << (64 -
30     3));
31     if (*source & (INT64_C(1) << (64 - 34))) *destination |= (INT64_C(1) << (64 -
32     4));
33     if (*source & (INT64_C(1) << (64 - 26))) *destination |= (INT64_C(1) << (64 -
34     5));
35     if (*source & (INT64_C(1) << (64 - 18))) *destination |= (INT64_C(1) << (64 -
36     6));
37     if (*source & (INT64_C(1) << (64 - 10))) *destination |= (INT64_C(1) << (64 -
38     7));
39     if (*source & (INT64_C(1) << (64 - 2))) *destination |= (INT64_C(1) << (64 -
40     8));
41     if (*source & (INT64_C(1) << (64 - 60))) *destination |= (INT64_C(1) << (64 -
42     9));
43     if (*source & (INT64_C(1) << (64 - 52))) *destination |= (INT64_C(1) << (64 -
44     10));
45     if (*source & (INT64_C(1) << (64 - 44))) *destination |= (INT64_C(1) << (64 -
46     11));
47     if (*source & (INT64_C(1) << (64 - 36))) *destination |= (INT64_C(1) << (64 -
48     12));
49     if (*source & (INT64_C(1) << (64 - 28))) *destination |= (INT64_C(1) << (64 -
50     13));
```

```
38     if (*source & (INT64_C(1) << (64 - 20))) *destination |= (INT64_C(1) << (64 - 20));
39     if (*source & (INT64_C(1) << (64 - 12))) *destination |= (INT64_C(1) << (64 - 12));
40     if (*source & (INT64_C(1) << (64 - 4))) *destination |= (INT64_C(1) << (64 - 4));
41     if (*source & (INT64_C(1) << (64 - 62))) *destination |= (INT64_C(1) << (64 - 62));
42     if (*source & (INT64_C(1) << (64 - 54))) *destination |= (INT64_C(1) << (64 - 54));
43     if (*source & (INT64_C(1) << (64 - 46))) *destination |= (INT64_C(1) << (64 - 46));
44     if (*source & (INT64_C(1) << (64 - 38))) *destination |= (INT64_C(1) << (64 - 38));
45     if (*source & (INT64_C(1) << (64 - 30))) *destination |= (INT64_C(1) << (64 - 30));
46     if (*source & (INT64_C(1) << (64 - 22))) *destination |= (INT64_C(1) << (64 - 22));
47     if (*source & (INT64_C(1) << (64 - 14))) *destination |= (INT64_C(1) << (64 - 14));
48     if (*source & (INT64_C(1) << (64 - 6))) *destination |= (INT64_C(1) << (64 - 6));
49     if (*source & (INT64_C(1) << (64 - 64))) *destination |= (INT64_C(1) << (64 - 64));
50     if (*source & (INT64_C(1) << (64 - 56))) *destination |= (INT64_C(1) << (64 - 56));
51     if (*source & (INT64_C(1) << (64 - 48))) *destination |= (INT64_C(1) << (64 - 48));
52     if (*source & (INT64_C(1) << (64 - 40))) *destination |= (INT64_C(1) << (64 - 40));
53     if (*source & (INT64_C(1) << (64 - 32))) *destination |= (INT64_C(1) << (64 - 32));
54     if (*source & (INT64_C(1) << (64 - 24))) *destination |= (INT64_C(1) << (64 - 24));
55     if (*source & (INT64_C(1) << (64 - 16))) *destination |= (INT64_C(1) << (64 - 16));
56     if (*source & (INT64_C(1) << (64 - 8))) *destination |= (INT64_C(1) << (64 - 8));
57     if (*source & (INT64_C(1) << (64 - 57))) *destination |= (INT64_C(1) << (64 - 57));
58     if (*source & (INT64_C(1) << (64 - 49))) *destination |= (INT64_C(1) << (64 - 49));
59     if (*source & (INT64_C(1) << (64 - 41))) *destination |= (INT64_C(1) << (64 - 41));
60     if (*source & (INT64_C(1) << (64 - 33))) *destination |= (INT64_C(1) << (64 - 33));
61     if (*source & (INT64_C(1) << (64 - 25))) *destination |= (INT64_C(1) << (64 - 25));
62     if (*source & (INT64_C(1) << (64 - 17))) *destination |= (INT64_C(1) << (64 - 17));
63     if (*source & (INT64_C(1) << (64 - 9))) *destination |= (INT64_C(1) << (64 - 9));
```

```
64     if (*source & (INT64_C(1) << (64 - 1))) *destination |= (INT64_C(1) << (64 - 40));
65     if (*source & (INT64_C(1) << (64 - 59))) *destination |= (INT64_C(1) << (64 - 41));
66     if (*source & (INT64_C(1) << (64 - 51))) *destination |= (INT64_C(1) << (64 - 42));
67     if (*source & (INT64_C(1) << (64 - 43))) *destination |= (INT64_C(1) << (64 - 43));
68     if (*source & (INT64_C(1) << (64 - 35))) *destination |= (INT64_C(1) << (64 - 44));
69     if (*source & (INT64_C(1) << (64 - 27))) *destination |= (INT64_C(1) << (64 - 45));
70     if (*source & (INT64_C(1) << (64 - 19))) *destination |= (INT64_C(1) << (64 - 46));
71     if (*source & (INT64_C(1) << (64 - 11))) *destination |= (INT64_C(1) << (64 - 47));
72     if (*source & (INT64_C(1) << (64 - 3))) *destination |= (INT64_C(1) << (64 - 48));
73     if (*source & (INT64_C(1) << (64 - 61))) *destination |= (INT64_C(1) << (64 - 49));
74     if (*source & (INT64_C(1) << (64 - 53))) *destination |= (INT64_C(1) << (64 - 50));
75     if (*source & (INT64_C(1) << (64 - 45))) *destination |= (INT64_C(1) << (64 - 51));
76     if (*source & (INT64_C(1) << (64 - 37))) *destination |= (INT64_C(1) << (64 - 52));
77     if (*source & (INT64_C(1) << (64 - 29))) *destination |= (INT64_C(1) << (64 - 53));
78     if (*source & (INT64_C(1) << (64 - 21))) *destination |= (INT64_C(1) << (64 - 54));
79     if (*source & (INT64_C(1) << (64 - 13))) *destination |= (INT64_C(1) << (64 - 55));
80     if (*source & (INT64_C(1) << (64 - 5))) *destination |= (INT64_C(1) << (64 - 56));
81     if (*source & (INT64_C(1) << (64 - 63))) *destination |= (INT64_C(1) << (64 - 57));
82     if (*source & (INT64_C(1) << (64 - 55))) *destination |= (INT64_C(1) << (64 - 58));
83     if (*source & (INT64_C(1) << (64 - 47))) *destination |= (INT64_C(1) << (64 - 59));
84     if (*source & (INT64_C(1) << (64 - 39))) *destination |= (INT64_C(1) << (64 - 60));
85     if (*source & (INT64_C(1) << (64 - 31))) *destination |= (INT64_C(1) << (64 - 61));
86     if (*source & (INT64_C(1) << (64 - 23))) *destination |= (INT64_C(1) << (64 - 62));
87     if (*source & (INT64_C(1) << (64 - 15))) *destination |= (INT64_C(1) << (64 - 63));
88     if (*source & (INT64_C(1) << (64 - 7))) *destination |= (INT64_C(1) << (64 - 64));
89
90 }
```

```

91
92 void apply_expansion_permutation(uint32_t *source, uint64_t *destination) {
93     /*
94     Applies the following permutation based on the following chart:
95
96             32 01 02 03 04 05
97             04 05 06 07 08 09
98             08 09 10 11 12 13
99             12 13 14 15 16 17
100            16 17 18 19 20 21
101            20 21 22 23 24 25
102            24 25 26 27 28 29
103            28 29 30 31 32 01
104     */
105     *destination = 0;
106     if (*source & (INT64_C(1) << (32 - 32))) *destination |= (INT64_C(1) << (64 - 1));
107     if (*source & (INT64_C(1) << (32 - 1))) *destination |= (INT64_C(1) << (64 - 2));
108     if (*source & (INT64_C(1) << (32 - 2))) *destination |= (INT64_C(1) << (64 - 3));
109     if (*source & (INT64_C(1) << (32 - 3))) *destination |= (INT64_C(1) << (64 - 4));
110     if (*source & (INT64_C(1) << (32 - 4))) *destination |= (INT64_C(1) << (64 - 5));
111     if (*source & (INT64_C(1) << (32 - 5))) *destination |= (INT64_C(1) << (64 - 6));
112     if (*source & (INT64_C(1) << (32 - 4))) *destination |= (INT64_C(1) << (64 - 7));
113     if (*source & (INT64_C(1) << (32 - 5))) *destination |= (INT64_C(1) << (64 - 8));
114     if (*source & (INT64_C(1) << (32 - 6))) *destination |= (INT64_C(1) << (64 - 9));
115     if (*source & (INT64_C(1) << (32 - 7))) *destination |= (INT64_C(1) << (64 - 10));
116     if (*source & (INT64_C(1) << (32 - 8))) *destination |= (INT64_C(1) << (64 - 11));
117     if (*source & (INT64_C(1) << (32 - 9))) *destination |= (INT64_C(1) << (64 - 12));
118     if (*source & (INT64_C(1) << (32 - 8))) *destination |= (INT64_C(1) << (64 - 13));
119     if (*source & (INT64_C(1) << (32 - 9))) *destination |= (INT64_C(1) << (64 - 14));
120     if (*source & (INT64_C(1) << (32 - 10))) *destination |= (INT64_C(1) << (64 - 15));
121     if (*source & (INT64_C(1) << (32 - 11))) *destination |= (INT64_C(1) << (64 - 16));
122     if (*source & (INT64_C(1) << (32 - 12))) *destination |= (INT64_C(1) << (64 - 17));
123     if (*source & (INT64_C(1) << (32 - 13))) *destination |= (INT64_C(1) << (64 - 18));
124     if (*source & (INT64_C(1) << (32 - 12))) *destination |= (INT64_C(1) << (64 - 19));

```



```
19));  
125     if (*source & (INT64_C(1) << (32 - 13))) *destination |= (INT64_C(1) << (64 - 20));  
126     if (*source & (INT64_C(1) << (32 - 14))) *destination |= (INT64_C(1) << (64 - 21));  
127     if (*source & (INT64_C(1) << (32 - 15))) *destination |= (INT64_C(1) << (64 - 22));  
128     if (*source & (INT64_C(1) << (32 - 16))) *destination |= (INT64_C(1) << (64 - 23));  
129     if (*source & (INT64_C(1) << (32 - 17))) *destination |= (INT64_C(1) << (64 - 24));  
130     if (*source & (INT64_C(1) << (32 - 16))) *destination |= (INT64_C(1) << (64 - 25));  
131     if (*source & (INT64_C(1) << (32 - 17))) *destination |= (INT64_C(1) << (64 - 26));  
132     if (*source & (INT64_C(1) << (32 - 18))) *destination |= (INT64_C(1) << (64 - 27));  
133     if (*source & (INT64_C(1) << (32 - 19))) *destination |= (INT64_C(1) << (64 - 28));  
134     if (*source & (INT64_C(1) << (32 - 20))) *destination |= (INT64_C(1) << (64 - 29));  
135     if (*source & (INT64_C(1) << (32 - 21))) *destination |= (INT64_C(1) << (64 - 30));  
136     if (*source & (INT64_C(1) << (32 - 20))) *destination |= (INT64_C(1) << (64 - 31));  
137     if (*source & (INT64_C(1) << (32 - 21))) *destination |= (INT64_C(1) << (64 - 32));  
138     if (*source & (INT64_C(1) << (32 - 22))) *destination |= (INT64_C(1) << (64 - 33));  
139     if (*source & (INT64_C(1) << (32 - 23))) *destination |= (INT64_C(1) << (64 - 34));  
140     if (*source & (INT64_C(1) << (32 - 24))) *destination |= (INT64_C(1) << (64 - 35));  
141     if (*source & (INT64_C(1) << (32 - 25))) *destination |= (INT64_C(1) << (64 - 36));  
142     if (*source & (INT64_C(1) << (32 - 24))) *destination |= (INT64_C(1) << (64 - 37));  
143     if (*source & (INT64_C(1) << (32 - 25))) *destination |= (INT64_C(1) << (64 - 38));  
144     if (*source & (INT64_C(1) << (32 - 26))) *destination |= (INT64_C(1) << (64 - 39));  
145     if (*source & (INT64_C(1) << (32 - 27))) *destination |= (INT64_C(1) << (64 - 40));  
146     if (*source & (INT64_C(1) << (32 - 28))) *destination |= (INT64_C(1) << (64 - 41));  
147     if (*source & (INT64_C(1) << (32 - 29))) *destination |= (INT64_C(1) << (64 - 42));  
148     if (*source & (INT64_C(1) << (32 - 28))) *destination |= (INT64_C(1) << (64 - 43));  
149     if (*source & (INT64_C(1) << (32 - 29))) *destination |= (INT64_C(1) << (64 - 44));  
150     if (*source & (INT64_C(1) << (32 - 30))) *destination |= (INT64_C(1) << (64 - 45));
```

```

        45));
151     if (*source & (INT64_C(1) << (32 - 31))) *destination |= (INT64_C(1) << (64 - 46));
152     if (*source & (INT64_C(1) << (32 - 32))) *destination |= (INT64_C(1) << (64 - 47));
153     if (*source & (INT64_C(1) << (32 - 1))) *destination |= (INT64_C(1) << (64 - 48));
154 }
155
156 void apply_final_permutation(uint64_t *source, uint64_t *destination) {
157     /*
158     Applies the following permutation based on the following chart:
159             40 08 48 16 56 24 64 32
160             39 07 47 15 55 23 63 31
161             38 06 46 14 54 22 62 30
162             37 05 45 13 53 21 61 29
163             36 04 44 12 52 20 60 28
164             35 03 43 11 51 19 59 27
165             34 02 42 10 50 18 58 26
166             33 01 41 09 49 17 57 25
167     */
168     *destination = 0;
169     if (*source & (INT64_C(1) << (64 - 40))) *destination |= (INT64_C(1) << (64 - 1));
170     if (*source & (INT64_C(1) << (64 - 8))) *destination |= (INT64_C(1) << (64 - 2));
171     if (*source & (INT64_C(1) << (64 - 48))) *destination |= (INT64_C(1) << (64 - 3));
172     if (*source & (INT64_C(1) << (64 - 16))) *destination |= (INT64_C(1) << (64 - 4));
173     if (*source & (INT64_C(1) << (64 - 56))) *destination |= (INT64_C(1) << (64 - 5));
174     if (*source & (INT64_C(1) << (64 - 24))) *destination |= (INT64_C(1) << (64 - 6));
175     if (*source & (INT64_C(1) << (64 - 64))) *destination |= (INT64_C(1) << (64 - 7));
176     if (*source & (INT64_C(1) << (64 - 32))) *destination |= (INT64_C(1) << (64 - 8));
177     if (*source & (INT64_C(1) << (64 - 39))) *destination |= (INT64_C(1) << (64 - 9));
178     if (*source & (INT64_C(1) << (64 - 7))) *destination |= (INT64_C(1) << (64 - 10));
179     if (*source & (INT64_C(1) << (64 - 47))) *destination |= (INT64_C(1) << (64 - 11));
180     if (*source & (INT64_C(1) << (64 - 15))) *destination |= (INT64_C(1) << (64 - 12));
181     if (*source & (INT64_C(1) << (64 - 55))) *destination |= (INT64_C(1) << (64 - 13));
182     if (*source & (INT64_C(1) << (64 - 23))) *destination |= (INT64_C(1) << (64 - 14));
183     if (*source & (INT64_C(1) << (64 - 63))) *destination |= (INT64_C(1) << (64 - 15));

```

```
184     if (*source & (INT64_C(1) << (64 - 31))) *destination |= (INT64_C(1) << (64 - 16));
185     if (*source & (INT64_C(1) << (64 - 38))) *destination |= (INT64_C(1) << (64 - 17));
186     if (*source & (INT64_C(1) << (64 - 6))) *destination |= (INT64_C(1) << (64 - 18));
187     if (*source & (INT64_C(1) << (64 - 46))) *destination |= (INT64_C(1) << (64 - 19));
188     if (*source & (INT64_C(1) << (64 - 14))) *destination |= (INT64_C(1) << (64 - 20));
189     if (*source & (INT64_C(1) << (64 - 54))) *destination |= (INT64_C(1) << (64 - 21));
190     if (*source & (INT64_C(1) << (64 - 22))) *destination |= (INT64_C(1) << (64 - 22));
191     if (*source & (INT64_C(1) << (64 - 62))) *destination |= (INT64_C(1) << (64 - 23));
192     if (*source & (INT64_C(1) << (64 - 30))) *destination |= (INT64_C(1) << (64 - 24));
193     if (*source & (INT64_C(1) << (64 - 37))) *destination |= (INT64_C(1) << (64 - 25));
194     if (*source & (INT64_C(1) << (64 - 5))) *destination |= (INT64_C(1) << (64 - 26));
195     if (*source & (INT64_C(1) << (64 - 45))) *destination |= (INT64_C(1) << (64 - 27));
196     if (*source & (INT64_C(1) << (64 - 13))) *destination |= (INT64_C(1) << (64 - 28));
197     if (*source & (INT64_C(1) << (64 - 53))) *destination |= (INT64_C(1) << (64 - 29));
198     if (*source & (INT64_C(1) << (64 - 21))) *destination |= (INT64_C(1) << (64 - 30));
199     if (*source & (INT64_C(1) << (64 - 61))) *destination |= (INT64_C(1) << (64 - 31));
200     if (*source & (INT64_C(1) << (64 - 29))) *destination |= (INT64_C(1) << (64 - 32));
201     if (*source & (INT64_C(1) << (64 - 36))) *destination |= (INT64_C(1) << (64 - 33));
202     if (*source & (INT64_C(1) << (64 - 4))) *destination |= (INT64_C(1) << (64 - 34));
203     if (*source & (INT64_C(1) << (64 - 44))) *destination |= (INT64_C(1) << (64 - 35));
204     if (*source & (INT64_C(1) << (64 - 12))) *destination |= (INT64_C(1) << (64 - 36));
205     if (*source & (INT64_C(1) << (64 - 52))) *destination |= (INT64_C(1) << (64 - 37));
206     if (*source & (INT64_C(1) << (64 - 20))) *destination |= (INT64_C(1) << (64 - 38));
207     if (*source & (INT64_C(1) << (64 - 60))) *destination |= (INT64_C(1) << (64 - 39));
208     if (*source & (INT64_C(1) << (64 - 28))) *destination |= (INT64_C(1) << (64 - 40));
209     if (*source & (INT64_C(1) << (64 - 35))) *destination |= (INT64_C(1) << (64 - 41));
```

```
210     if (*source & (INT64_C(1) << (64 - 3))) *destination |= (INT64_C(1) << (64 - 3) & *source);
211     if (*source & (INT64_C(1) << (64 - 43))) *destination |= (INT64_C(1) << (64 - 43) & *source);
212     if (*source & (INT64_C(1) << (64 - 11))) *destination |= (INT64_C(1) << (64 - 11) & *source);
213     if (*source & (INT64_C(1) << (64 - 51))) *destination |= (INT64_C(1) << (64 - 51) & *source);
214     if (*source & (INT64_C(1) << (64 - 19))) *destination |= (INT64_C(1) << (64 - 19) & *source);
215     if (*source & (INT64_C(1) << (64 - 59))) *destination |= (INT64_C(1) << (64 - 59) & *source);
216     if (*source & (INT64_C(1) << (64 - 27))) *destination |= (INT64_C(1) << (64 - 27) & *source);
217     if (*source & (INT64_C(1) << (64 - 34))) *destination |= (INT64_C(1) << (64 - 34) & *source);
218     if (*source & (INT64_C(1) << (64 - 2))) *destination |= (INT64_C(1) << (64 - 2) & *source);
219     if (*source & (INT64_C(1) << (64 - 42))) *destination |= (INT64_C(1) << (64 - 42) & *source);
220     if (*source & (INT64_C(1) << (64 - 10))) *destination |= (INT64_C(1) << (64 - 10) & *source);
221     if (*source & (INT64_C(1) << (64 - 50))) *destination |= (INT64_C(1) << (64 - 50) & *source);
222     if (*source & (INT64_C(1) << (64 - 18))) *destination |= (INT64_C(1) << (64 - 18) & *source);
223     if (*source & (INT64_C(1) << (64 - 58))) *destination |= (INT64_C(1) << (64 - 58) & *source);
224     if (*source & (INT64_C(1) << (64 - 26))) *destination |= (INT64_C(1) << (64 - 26) & *source);
225     if (*source & (INT64_C(1) << (64 - 33))) *destination |= (INT64_C(1) << (64 - 33) & *source);
226     if (*source & (INT64_C(1) << (64 - 1))) *destination |= (INT64_C(1) << (64 - 1) & *source);
227     if (*source & (INT64_C(1) << (64 - 41))) *destination |= (INT64_C(1) << (64 - 41) & *source);
228     if (*source & (INT64_C(1) << (64 - 9))) *destination |= (INT64_C(1) << (64 - 9) & *source);
229     if (*source & (INT64_C(1) << (64 - 49))) *destination |= (INT64_C(1) << (64 - 49) & *source);
230     if (*source & (INT64_C(1) << (64 - 17))) *destination |= (INT64_C(1) << (64 - 17) & *source);
231     if (*source & (INT64_C(1) << (64 - 57))) *destination |= (INT64_C(1) << (64 - 57) & *source);
232     if (*source & (INT64_C(1) << (64 - 25))) *destination |= (INT64_C(1) << (64 - 25) & *source);
233 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  reverser.cpp: Reverses a 64-bit value at the byte level (MSByte -> LSByte, etc)  ↗
6  from the source. Stores the result in the "destination" (provided)
7  */
8  #include "reverser.h"
9
10 void reverse(uint64_t *source, uint64_t *destination) {
11     *destination = 0;
12
13     // Everything is going to be reversed at the *byte* level, not the bit level
14     // 01-08 -> 57-64    09-16 -> 49-56    17-24 -> 41-48
15     // 25-32 -> 33-40    33-40 -> 25-32    41-48 -> 17-24
16     // 49-56 -> 09-16    57-64 -> 01-08
17     if (*source & (INT64_C(1) << 0)) *destination |= (INT64_C(1) << 56);
18     if (*source & (INT64_C(1) << 1)) *destination |= (INT64_C(1) << 57);
19     if (*source & (INT64_C(1) << 2)) *destination |= (INT64_C(1) << 58);
20     if (*source & (INT64_C(1) << 3)) *destination |= (INT64_C(1) << 59);
21     if (*source & (INT64_C(1) << 4)) *destination |= (INT64_C(1) << 60);
22     if (*source & (INT64_C(1) << 5)) *destination |= (INT64_C(1) << 61);
23     if (*source & (INT64_C(1) << 6)) *destination |= (INT64_C(1) << 62);
24     if (*source & (INT64_C(1) << 7)) *destination |= (INT64_C(1) << 63);
25     if (*source & (INT64_C(1) << 8)) *destination |= (INT64_C(1) << 48);
26     if (*source & (INT64_C(1) << 9)) *destination |= (INT64_C(1) << 49);
27     if (*source & (INT64_C(1) << 10)) *destination |= (INT64_C(1) << 50);
28     if (*source & (INT64_C(1) << 11)) *destination |= (INT64_C(1) << 51);
29     if (*source & (INT64_C(1) << 12)) *destination |= (INT64_C(1) << 52);
30     if (*source & (INT64_C(1) << 13)) *destination |= (INT64_C(1) << 53);
31     if (*source & (INT64_C(1) << 14)) *destination |= (INT64_C(1) << 54);
32     if (*source & (INT64_C(1) << 15)) *destination |= (INT64_C(1) << 55);
33     if (*source & (INT64_C(1) << 16)) *destination |= (INT64_C(1) << 40);
34     if (*source & (INT64_C(1) << 17)) *destination |= (INT64_C(1) << 41);
35     if (*source & (INT64_C(1) << 18)) *destination |= (INT64_C(1) << 42);
36     if (*source & (INT64_C(1) << 19)) *destination |= (INT64_C(1) << 43);
37     if (*source & (INT64_C(1) << 20)) *destination |= (INT64_C(1) << 44);
38     if (*source & (INT64_C(1) << 21)) *destination |= (INT64_C(1) << 45);
39     if (*source & (INT64_C(1) << 22)) *destination |= (INT64_C(1) << 46);
40     if (*source & (INT64_C(1) << 23)) *destination |= (INT64_C(1) << 47);
41     if (*source & (INT64_C(1) << 24)) *destination |= (INT64_C(1) << 32);
42     if (*source & (INT64_C(1) << 25)) *destination |= (INT64_C(1) << 33);
43     if (*source & (INT64_C(1) << 26)) *destination |= (INT64_C(1) << 34);
44     if (*source & (INT64_C(1) << 27)) *destination |= (INT64_C(1) << 35);
45     if (*source & (INT64_C(1) << 28)) *destination |= (INT64_C(1) << 36);
46     if (*source & (INT64_C(1) << 29)) *destination |= (INT64_C(1) << 37);
47     if (*source & (INT64_C(1) << 30)) *destination |= (INT64_C(1) << 38);
48     if (*source & (INT64_C(1) << 31)) *destination |= (INT64_C(1) << 39);
49     if (*source & (INT64_C(1) << 32)) *destination |= (INT64_C(1) << 24);
50     if (*source & (INT64_C(1) << 33)) *destination |= (INT64_C(1) << 25);
51     if (*source & (INT64_C(1) << 34)) *destination |= (INT64_C(1) << 26);
52     if (*source & (INT64_C(1) << 35)) *destination |= (INT64_C(1) << 27);
```

```
52     if (*source & (INT64_C(1) << 36)) *destination |= (INT64_C(1) << 28);
53     if (*source & (INT64_C(1) << 37)) *destination |= (INT64_C(1) << 29);
54     if (*source & (INT64_C(1) << 38)) *destination |= (INT64_C(1) << 30);
55     if (*source & (INT64_C(1) << 39)) *destination |= (INT64_C(1) << 31);
56     if (*source & (INT64_C(1) << 40)) *destination |= (INT64_C(1) << 16);
57     if (*source & (INT64_C(1) << 41)) *destination |= (INT64_C(1) << 17);
58     if (*source & (INT64_C(1) << 42)) *destination |= (INT64_C(1) << 18);
59     if (*source & (INT64_C(1) << 43)) *destination |= (INT64_C(1) << 19);
60     if (*source & (INT64_C(1) << 44)) *destination |= (INT64_C(1) << 20);
61     if (*source & (INT64_C(1) << 45)) *destination |= (INT64_C(1) << 21);
62     if (*source & (INT64_C(1) << 46)) *destination |= (INT64_C(1) << 22);
63     if (*source & (INT64_C(1) << 47)) *destination |= (INT64_C(1) << 23);
64     if (*source & (INT64_C(1) << 48)) *destination |= (INT64_C(1) << 8);
65     if (*source & (INT64_C(1) << 49)) *destination |= (INT64_C(1) << 9);
66     if (*source & (INT64_C(1) << 50)) *destination |= (INT64_C(1) << 10);
67     if (*source & (INT64_C(1) << 51)) *destination |= (INT64_C(1) << 11);
68     if (*source & (INT64_C(1) << 52)) *destination |= (INT64_C(1) << 12);
69     if (*source & (INT64_C(1) << 53)) *destination |= (INT64_C(1) << 13);
70     if (*source & (INT64_C(1) << 54)) *destination |= (INT64_C(1) << 14);
71     if (*source & (INT64_C(1) << 55)) *destination |= (INT64_C(1) << 15);
72     if (*source & (INT64_C(1) << 56)) *destination |= (INT64_C(1) << 0);
73     if (*source & (INT64_C(1) << 57)) *destination |= (INT64_C(1) << 1);
74     if (*source & (INT64_C(1) << 58)) *destination |= (INT64_C(1) << 2);
75     if (*source & (INT64_C(1) << 59)) *destination |= (INT64_C(1) << 3);
76     if (*source & (INT64_C(1) << 60)) *destination |= (INT64_C(1) << 4);
77     if (*source & (INT64_C(1) << 61)) *destination |= (INT64_C(1) << 5);
78     if (*source & (INT64_C(1) << 62)) *destination |= (INT64_C(1) << 6);
79     if (*source & (INT64_C(1) << 63)) *destination |= (INT64_C(1) << 7);
80 }
```

```
1  /*
2  Mike Koch
3  EECS 4760 - Computer Security
4  DES
5  rounds.cpp: Handles executing the 16 Feistel rounds, given the result of the
    initial permutation and the keys. Stores the result in "output" (provided).
6  */
7  #include "rounds.h"
8  #include "permutator.h"
9  #include "sbox.h"
10
11 void apply_rounds(uint64_t *initial_permutation, uint64_t *output, uint64_t
    *keys, Mode mode) {
12     // Divide the initial permutation into 2 32-bit blocks
13     uint32_t left_half = (*initial_permutation & 0xFFFFFFFF00000000) >> 32;
14     uint32_t right_half = *initial_permutation & 0x00000000FFFFFFFF;
15
16     int startIndex = 0;
17     int endIndex = 16;
18     int incrementer = 1;
19
20     if (mode == Mode::DECRYPTION) {
21         // If we're decrypting, we should start at the last key and loop until we
            get to the first key
22         startIndex = 15;
23         endIndex = -1;
24         incrementer = -1;
25     }
26
27     // Rounds
28     for (int i = startIndex; i != endIndex; i += incrementer) {
29         uint64_t key = keys[i];
30         uint32_t original_right_half = right_half;
31
32         // Apply the initial permutation (the expansion permutation)
33         uint64_t expanded_right_half;
34         apply_expansion_permutation(&right_half, &expanded_right_half);
35
36         // XOR the result of the expansion permutation with the current key
37         uint64_t right_and_key = key ^ expanded_right_half;
38
39         // Splitting the result of the XOR operation into eight sets of 6.
40         uint8_t six_bit_one = (uint8_t)((right_and_key & 0xFC00000000000000) >>
            56);
41         uint8_t six_bit_two = (uint8_t)((right_and_key & 0x03F0000000000000) >>
            50);
42         uint8_t six_bit_three = (uint8_t)((right_and_key & 0x000FC00000000000) >>
            44);
43         uint8_t six_bit_four = (uint8_t)((right_and_key & 0x00003F0000000000) >>
            38);
44         uint8_t six_bit_five = (uint8_t)((right_and_key & 0x000000FC00000000) >>
            32);
```

```

45     uint8_t six_bit_six = (uint8_t)((right_and_key & 0x00000003F0000000) >>
26);
46     uint8_t six_bit_seven = (uint8_t)((right_and_key & 0x00000000FC000000) >>
20);
47     uint8_t six_bit_eight = (uint8_t)((right_and_key & 0x0000000003F00000) >>
14);
48
49     /* The code below is repetitive, and here's what it does:
50     1) Get the current row by checking the value of the first and 6th bit
51     2) Get the value of bits 2 through 5, and add them accordingly to get a
decimal value
52     3) Consult the correct S_BOX 2D array to get the value of the row and
column.
53     */
54     // Six bit one, S-Box 1
55     int row_number = 0;
56     if ((six_bit_one & (1 << 7)) && (six_bit_one & (1 << 2))) row_number = 3;
57     if ((six_bit_one & (1 << 7)) && (six_bit_one & (1 << 2)) == 0) row_number
= 2;
58     if ((six_bit_one & (1 << 7)) == 0 && (six_bit_one & (1 << 2))) row_number
= 1;
59
60     uint8_t s_box_one_value;
61     int bit_two_value = (six_bit_one & (1 << 6)) ? 8 : 0;
62     int bit_three_value = (six_bit_one & (1 << 5)) ? 4 : 0;
63     int bit_four_value = (six_bit_one & (1 << 4)) ? 2 : 0;
64     int bit_five_value = (six_bit_one & (1 << 3)) ? 1 : 0;
65     int sum = bit_two_value + bit_three_value + bit_four_value +
bit_five_value;
66     s_box_one_value = S_BOX_ONE[row_number][sum];
67
68     // Six bit two, S-Box 2
69     row_number = 0;
70     if ((six_bit_two & (1 << 7)) && (six_bit_two & (1 << 2))) row_number = 3;
71     if ((six_bit_two & (1 << 7)) && (six_bit_two & (1 << 2)) == 0) row_number
= 2;
72     if ((six_bit_two & (1 << 7)) == 0 && (six_bit_two & (1 << 2))) row_number
= 1;
73
74     uint8_t s_box_two_value;
75     bit_two_value = (six_bit_two & (1 << 6)) ? 8 : 0;
76     bit_three_value = (six_bit_two & (1 << 5)) ? 4 : 0;
77     bit_four_value = (six_bit_two & (1 << 4)) ? 2 : 0;
78     bit_five_value = (six_bit_two & (1 << 3)) ? 1 : 0;
79     sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
80     s_box_two_value = S_BOX_TWO[row_number][sum];
81
82     // Six bit three, S-Box 3
83     row_number = 0;
84     if ((six_bit_three & (1 << 7)) && (six_bit_three & (1 << 2))) row_number
= 3;
85     if ((six_bit_three & (1 << 7)) && (six_bit_three & (1 << 2)) == 0)

```



```
    row_number = 2;
86     if ((six_bit_three & (1 << 7)) == 0 && (six_bit_three & (1 << 2)))
        row_number = 1;
87
88     uint8_t s_box_three_value;
89     bit_two_value = (six_bit_three & (1 << 6)) ? 8 : 0;
90     bit_three_value = (six_bit_three & (1 << 5)) ? 4 : 0;
91     bit_four_value = (six_bit_three & (1 << 4)) ? 2 : 0;
92     bit_five_value = (six_bit_three & (1 << 3)) ? 1 : 0;
93     sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
94     s_box_three_value = S_BOX_THREE[row_number][sum];
95
96     // Six bit four, S-Box 4
97     row_number = 0;
98     if ((six_bit_four & (1 << 7)) && (six_bit_four & (1 << 2))) row_number = 3;
99     if ((six_bit_four & (1 << 7)) && (six_bit_four & (1 << 2)) == 0)
        row_number = 2;
100    if ((six_bit_four & (1 << 7)) == 0 && (six_bit_four & (1 << 2)))
        row_number = 1;
101
102    uint8_t s_box_four_value;
103    bit_two_value = (six_bit_four & (1 << 6)) ? 8 : 0;
104    bit_three_value = (six_bit_four & (1 << 5)) ? 4 : 0;
105    bit_four_value = (six_bit_four & (1 << 4)) ? 2 : 0;
106    bit_five_value = (six_bit_four & (1 << 3)) ? 1 : 0;
107    sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
108    s_box_four_value = S_BOX_FOUR[row_number][sum];
109
110    // Six bit five, S-Box 5
111    row_number = 0;
112    if ((six_bit_five & (1 << 7)) && (six_bit_five & (1 << 2))) row_number = 3;
113    if ((six_bit_five & (1 << 7)) && (six_bit_five & (1 << 2)) == 0)
        row_number = 2;
114    if ((six_bit_five & (1 << 7)) == 0 && (six_bit_five & (1 << 2)))
        row_number = 1;
115
116    uint8_t s_box_five_value;
117    bit_two_value = (six_bit_five & (1 << 6)) ? 8 : 0;
118    bit_three_value = (six_bit_five & (1 << 5)) ? 4 : 0;
119    bit_four_value = (six_bit_five & (1 << 4)) ? 2 : 0;
120    bit_five_value = (six_bit_five & (1 << 3)) ? 1 : 0;
121    sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
122    s_box_five_value = S_BOX_FIVE[row_number][sum];
123
124    // Six bit six, S-Box 6
125    row_number = 0;
126    if ((six_bit_six & (1 << 7)) && (six_bit_six & (1 << 2))) row_number = 3;
127    if ((six_bit_six & (1 << 7)) && (six_bit_six & (1 << 2)) == 0) row_number
        = 2;
128    if ((six_bit_six & (1 << 7)) == 0 && (six_bit_six & (1 << 2))) row_number
```

```

    = 1;

129
130     uint8_t s_box_six_value;
131     bit_two_value = (six_bit_six & (1 << 6)) ? 8 : 0;
132     bit_three_value = (six_bit_six & (1 << 5)) ? 4 : 0;
133     bit_four_value = (six_bit_six & (1 << 4)) ? 2 : 0;
134     bit_five_value = (six_bit_six & (1 << 3)) ? 1 : 0;
135     sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
136     s_box_six_value = S_BOX_SIX[row_number][sum];
137
138     // Six bit seven, S-Box 7
139     row_number = 0;
140     if ((six_bit_seven & (1 << 7)) && (six_bit_seven & (1 << 2))) row_number = 3;
141     if ((six_bit_seven & (1 << 7)) && (six_bit_seven & (1 << 2)) == 0)
142         row_number = 2;
143     if ((six_bit_seven & (1 << 7)) == 0 && (six_bit_seven & (1 << 2)))
144         row_number = 1;
145
146     uint8_t s_box_seven_value;
147     bit_two_value = (six_bit_seven & (1 << 6)) ? 8 : 0;
148     bit_three_value = (six_bit_seven & (1 << 5)) ? 4 : 0;
149     bit_four_value = (six_bit_seven & (1 << 4)) ? 2 : 0;
150     bit_five_value = (six_bit_seven & (1 << 3)) ? 1 : 0;
151     sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
152     s_box_seven_value = S_BOX_SEVEN[row_number][sum];
153
154     // Six bit eight, S-Box 8
155     row_number = 0;
156     if ((six_bit_eight & (1 << 7)) && (six_bit_eight & (1 << 2))) row_number = 3;
157     if ((six_bit_eight & (1 << 7)) && (six_bit_eight & (1 << 2)) == 0)
158         row_number = 2;
159     if ((six_bit_eight & (1 << 7)) == 0 && (six_bit_eight & (1 << 2)))
160         row_number = 1;
161
162     uint8_t s_box_eight_value;
163     bit_two_value = (six_bit_eight & (1 << 6)) ? 8 : 0;
164     bit_three_value = (six_bit_eight & (1 << 5)) ? 4 : 0;
165     bit_four_value = (six_bit_eight & (1 << 4)) ? 2 : 0;
166     bit_five_value = (six_bit_eight & (1 << 3)) ? 1 : 0;
167     sum = bit_two_value + bit_three_value + bit_four_value + bit_five_value;
168     s_box_eight_value = S_BOX_EIGHT[row_number][sum];
169
170     // Use a straight P box to convert our 32-bit value (8 4-bit S box
    vlaues) to a new 32-bit value
171     // 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.
172     uint32_t straight_p_box_result = 0;
173     if (s_box_four_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32
    - 1));
174     if (s_box_two_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32

```

```
- 2));  
171     if (s_box_five_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32  
- 3));  
172     if (s_box_six_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32  
- 4));  
173     if (s_box_eight_value & (1 << (4 - 1))) straight_p_box_result |= (1 <<  
(32 - 5));  
174     if (s_box_three_value & (1 << (4 - 4))) straight_p_box_result |= (1 <<  
(32 - 6));  
175     if (s_box_seven_value & (1 << (4 - 4))) straight_p_box_result |= (1 <<  
(32 - 7));  
176     if (s_box_five_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32  
- 8));  
177     if (s_box_one_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32  
- 9));  
178     if (s_box_four_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32  
- 10));  
179     if (s_box_six_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32  
- 11));  
180     if (s_box_seven_value & (1 << (4 - 2))) straight_p_box_result |= (1 <<  
(32 - 12));  
181     if (s_box_two_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32  
- 13));  
182     if (s_box_five_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32  
- 14));  
183     if (s_box_eight_value & (1 << (4 - 3))) straight_p_box_result |= (1 <<  
(32 - 15));  
184     if (s_box_three_value & (1 << (4 - 2))) straight_p_box_result |= (1 <<  
(32 - 16));  
185     if (s_box_one_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32  
- 17));  
186     if (s_box_two_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32  
- 18));  
187     if (s_box_six_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32  
- 19));  
188     if (s_box_four_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32  
- 20));  
189     if (s_box_eight_value & (1 << (4 - 4))) straight_p_box_result |= (1 <<  
(32 - 21));  
190     if (s_box_seven_value & (1 << (4 - 3))) straight_p_box_result |= (1 <<  
(32 - 22));  
191     if (s_box_one_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32  
- 23));  
192     if (s_box_three_value & (1 << (4 - 1))) straight_p_box_result |= (1 <<  
(32 - 24));  
193     if (s_box_five_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32  
- 25));  
194     if (s_box_four_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32  
- 26));  
195     if (s_box_eight_value & (1 << (4 - 2))) straight_p_box_result |= (1 <<  
(32 - 27));  
196     if (s_box_two_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32
```

```
- 28));
197     if (s_box_six_value & (1 << (4 - 2))) straight_p_box_result |= (1 << (32 - 29));
198     if (s_box_three_value & (1 << (4 - 3))) straight_p_box_result |= (1 << (32 - 30));
199     if (s_box_one_value & (1 << (4 - 4))) straight_p_box_result |= (1 << (32 - 31));
200     if (s_box_seven_value & (1 << (4 - 1))) straight_p_box_result |= (1 << (32 - 32));
201
202
203     // XOR the straight_p_box_result with the left half
204     right_half = left_half ^ straight_p_box_result;
205
206     left_half = original_right_half;
207 }
208
209 // Since we're done, combine the left and right halves to make our output
210 *output = (uint64_t)right_half << 32 | left_half;
211 }
```