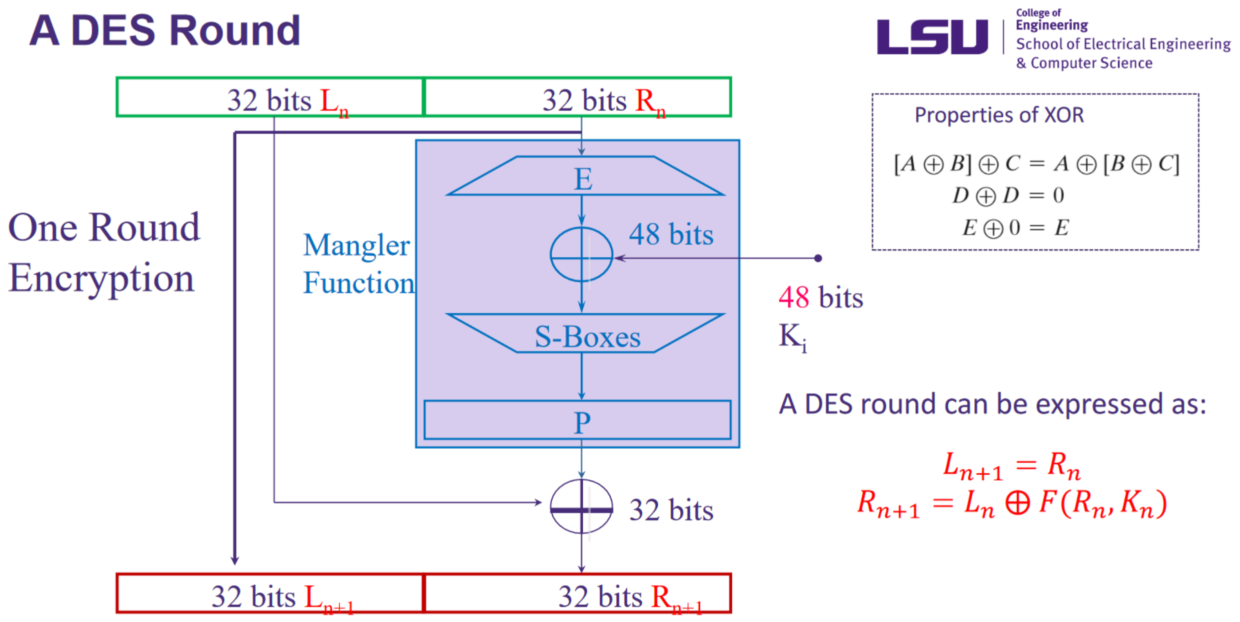
Brandon Lara
4/30/2022
4562 Mobile Security and Cryptography

Project 2 Report

Overview:

The DES encryption implementation presented is broken up into several parts. First, the program splits the plaintext into a left half and a right half. The right half is sent to the mangler function. This mangler function consists of an expansion function to expand the right half from 32 bits to 64 bits (only 48 bits are used here), a custom made 48 bit XOR function with the key as input, and 8 S box functions that shrink the output back down to 32 bits. The mangler function returns the output of the S box functions. This is notable because a fully fledged mangler function would also permute the input bits. The key that is used in the mangler function is static, meaning that new keys are not generated for each round. Generating new keys is a trivial addition, given more time. Other than these two omissions, this mangler function is identical to the one laid out in the slides:



Properties of XOR

$$[A \oplus B] \oplus C = A \oplus [B \oplus C]$$
$$D \oplus D = 0$$
$$E \oplus 0 = E$$

A DES round can be expressed as:

$$L_{n+1} = R_n$$
$$R_{n+1} = L_n \oplus F(R_n, K_n)$$

The decryption function looks nearly identical to the encryption function. The only difference is that LH (Left Half) and RH (Right Half) have been swapped. It is an interesting property of DES that the mangler function need not be reversible, so an inverse mangler function is not required to perform decryption. Instead, the same mangler function can be called with the inputs swapped to produce the correct output. This also means that mangler functions are interchangeable, because the contents of the mangler function does not affect the validity of the encryption or decryption. This is the motivating factor for simplifying the mangler function by omitting some of the functionality.

The DES algorithm runs for 16rounds and uses bitwise masking to interact with the inputs. The ciphertext and plaintext have been represented as 64 bit long long's. The right half and left half have been represented as 32 bit ints.

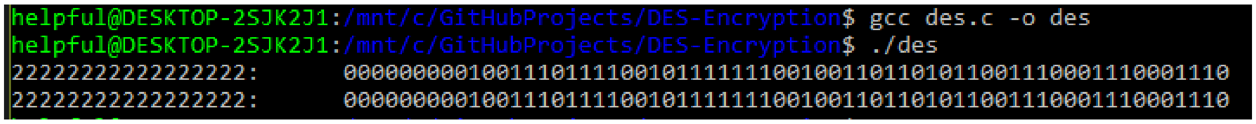
Usage:

Command to run:

```
Gcc des.c -o des
./des
```

The full DES encryption and decryption is self contained inside the des.c file. A global variable is set to define the plaintext. Changing this variable changes what value will be encrypted and decrypted. This is not an incredibly useful DES implementation because actual users would likely desire to encrypt large messages which they would want to represent as strings; however, this implementation is only meant to be a proof of concept. Someone wishing to use this encryption scheme will want to create an intermediary program that breaks their plaintext string into 64 bit long long's and then feeds that output to des.c.

Output:



```
helpful@DESKTOP-2SJK2J1:/mnt/c/GitHubProjects/DES-Encryption$ gcc des.c -o des
helpful@DESKTOP-2SJK2J1:/mnt/c/GitHubProjects/DES-Encryption$ ./des
22222222222222222:      0000000001001110111100101111111001001101101011001110001110001110
22222222222222222:      0000000001001110111100101111111001001101101011001110001110001110
```

Functions:

```c
void _debugPrint32bit(int input)
```

_DebugPrint32bit is a debug function designed to print input as a number and as binary.

```c
void _debugPrint64bit(long long input)
```

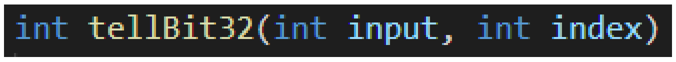
_DebugPrint64bit is a debug function designed to print input as a number and as binary. While this function was originally intended for debugging, it is now being used to demonstrate the correctness of the program. A finished and fully fledged implementation would likely write its output to a file and have a specified format for doing so, but since des.c is only a proof of concept, I felt this way of showing output was fine.

```c
long long setBit64(long long input, int index, int value)
```

setBit64 is a function designed to set the bit at index of input with value. Index indexes the binary number from right to left (little endian). Index can be any number from 0 to 63, value can only be 0 or 1, and input is as long long that is having its bit changed. This function does not check if the bit at index is already equal to value; this is not required because checking the value of the bit would require the same bit masking that is required to change the bit. The function returns the new long long after the modification has been made.

```
int setBit32(int input, int index, int value)
```

setBit32 works exactly the same as setBit64, except that the input is 32 bits. This means that the index can only be from 0 to 31. This is the only difference. The function returns the new int after the modification has been made. Next, manglerFunction(RH) is called to generate a mangled RH value. T



```
int tellBit32(int input, int index)
```

tellBit32 takes in a 32 bit input and an index. The function uses bit masking to identify the value of the bit at the index. The function returns 1 or 0 specifying whether the bit is on or off.

```
int tellBit64(long long input, int index)
```

tellBit64 takes in a 64 bit input and an index. The function uses bit masking to identify the value of the bit at the index. The function returns 1 or 0 specifying whether the bit is on or off.

```
long long DESRounds(long long text)
```

DESRounds takes in a plaintext input and performs 16 DES encryption rounds on that plaintext. The function works by looping 16 times; in each iteration of the loop, getLH(text) and getRH(text) are called to split the input into two 32 bit ints. Next, manglerFunction(RH) is called. Its output is xored with LH. Lastly, reconstructText(newRH, RH) is called to put the two 32 bit inputs back together as 1 64 bit long long.

```
long long DESRoundsDecrypt(long long text)
```

DESRoundsDecrypt takes in a cipher text and performs the same 16 DES encryption rounds on that cipher text. The function works by looping 16 times; in each iteration of the loop getLH(text) is used to get the right half, and getRH(text) is used to get the left half. This might seem confusing at first, but the DES decryption algorithm dictates that the right half and left half are swapped. Next, manglerFunction(RH) is called. Its output is xored with LH. Lastly, reconstructText(newRH, RH) is called to put the two 32 bit inputs back together as 1 64 bit long long.

```
int getRH(long long text)
```

getRH takes in a 64 bit input and returns an int consisting of the bits from the right half of text.

```
int getLH(long long text)
```

getLH takes in a 64 bit input and returns an int consisting of the bits from the left half of text.

```
long long reconstructText(int rightHalf, int leftHalf)
```

reconstructText takes in two 32 bit ints and concatenates them into 1 64 bit long long. The function returns this 64 bit long long.

```
int manglerFunction(int RH)
```

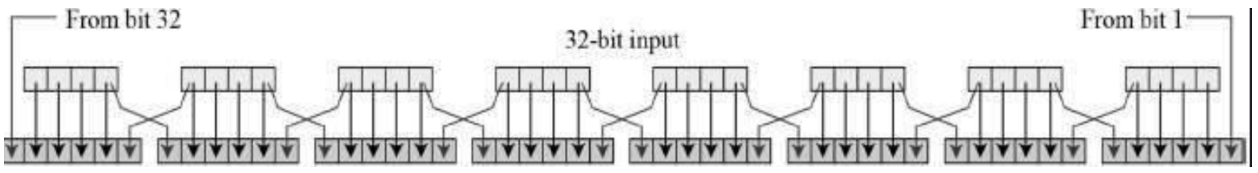
manglerFunction takes in the right half of the DES round. It first calls expansionFunction(RH) to expand the 32 bit input to a 64 bit number (only 48 bits are used). Next the function calls all 8 sbox functions (refer to sbox section). The function then returns the output of the sbox functions.

```
long long XOR48(long long RH, long long key)
```

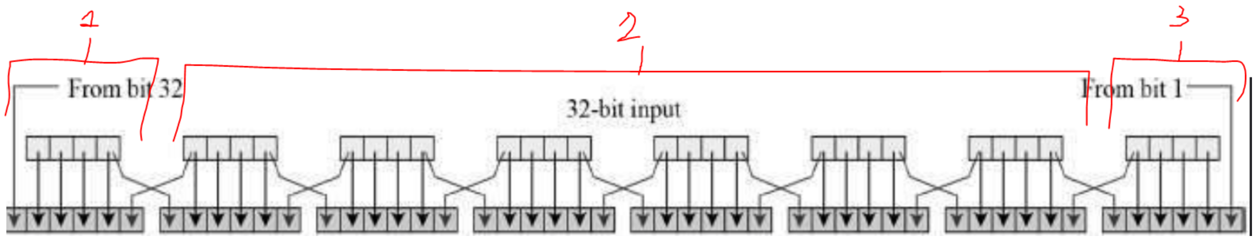
XOR48 takes in two 64 bit long longs and xors only the first 48 bits together. The function returns the result of the xor.

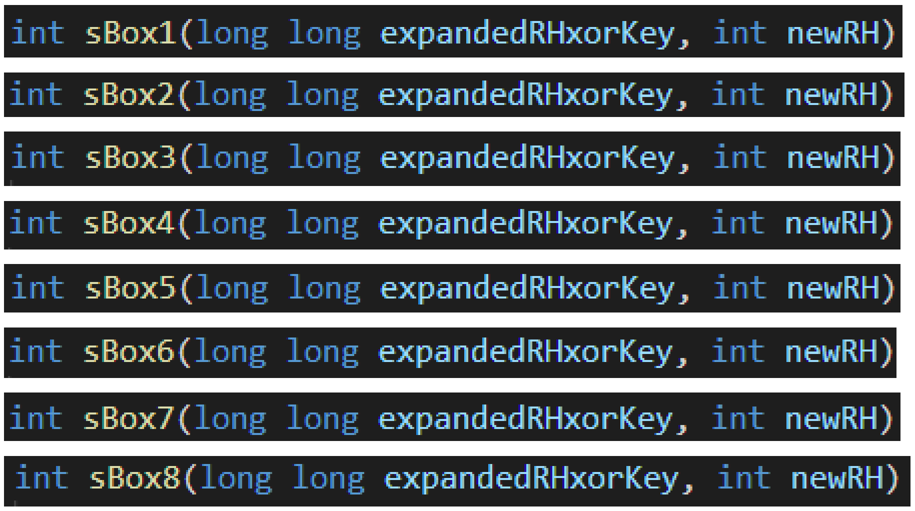
```
long long expansionFunction(int RH)
```

expansionFunction takes in a 32 bit int and expands it to a 64 bit long long (only 48 bits are used). The function is modeled after this diagram from the slides:



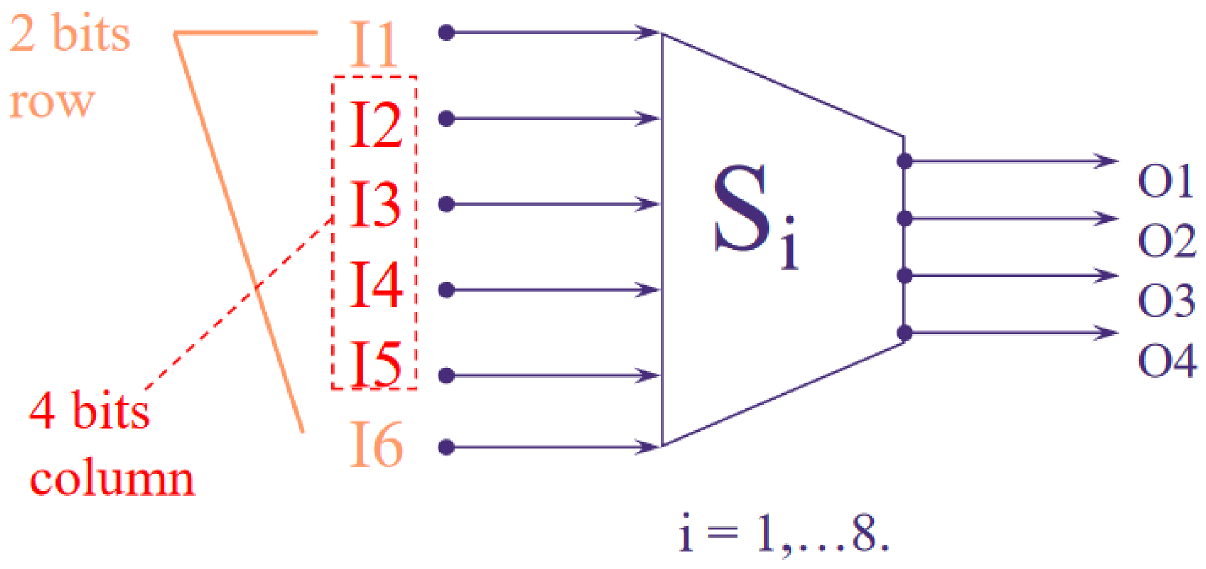
The function works creating a long long variable (this will be the return value) and iterating over the RH variable and assigning the bit value to the appropriate indexes in the 64 bit long long value. The function is best understood as 3 loops. The first loop deals with the first 6 bits in the long long. The second loop iterates over the next 36 bits of the long long. And the final loop handles the last 6 bits. The loops are marked with comments in the source code as “First”, “Second”, and “Third”. Here is an updated diagram that marked the sections associated with each loop:





```
int sBox1(long long expandedRHxorKey, int newRH)
int sBox2(long long expandedRHxorKey, int newRH)
int sBox3(long long expandedRHxorKey, int newRH)
int sBox4(long long expandedRHxorKey, int newRH)
int sBox5(long long expandedRHxorKey, int newRH)
int sBox6(long long expandedRHxorKey, int newRH)
int sBox7(long long expandedRHxorKey, int newRH)
int sBox8(long long expandedRHxorKey, int newRH)
```

The sBox functions are all identical in structure. Each one takes the same long long input and loops over 6 bits of expandedRHxorKey and then takes the first and last bit of the 6 bit value and uses it as the row index, and takes the middle 4 bits and use them as the column index and then looks up the value stored at that location in the associated sbox function. The int newRH is then assigned the bits of that look up value and this is repeated 8 times (once in each function) and 8 * 4 = 32, meaning this process takes in a 48 bit input and shrinks it down to a 32 bit output. A diagram is included as my model for designing the functions:



Follow-Up Work:

This is a fairly complete implementation, but a few notable features are missing. First, the mangler function is missing the permutation function, and the key is not dynamically generated for each round. Follow up work could iterate on this result by adding these two functions to the mangler function. Further, the program is currently difficult to interface with. Additional interaction could be made to provide an API so that the implementation can be flexible for users wishing to use it (This is unlikely since DES is outdated, and no one will want to use my DES encryption scheme over someone else's, but for the sake of completeness these iterations could greatly improve the implementation).