Jake Levy
Shruthi Kotha
Phil Shafer
Network Security
Due 4/08/14
DES REPORT AND CRACK

DES CRACKED KEY: The key found by our implementation of the cracker was **UKRAINE** , it was found in approximately 6.3 billion tries (about 30ish hours) starting from AAAAAAA.

The DES implementation we used was that provided to us by Dr. Hnatyshin. Our implementation of the main file for the cracker can be found on the attached pages. Our report on the DES implementation can be found below.

DATA STRUCTURES

Prior to the actual DES algorithm code, there are several data structures that have been initialized to allow for the operations that are a part of how the DES algorithm obfuscates the data.

table_DES_IP - this array simulates the Initial Permutation table. It is used in the ComputeIP function to determine the initial permutation of the data. It describes the "new" ordering of the bits. The index indicates the initial position of the input whereas the value of the array element indicates the bit's position in the output.

table_DES_FP - this array simulates the Final Permutation table. It is used in the ComputeFP function to perform the final permutation of the data. It is similar in design to the IP table.

table_DES_PC1 - this array represents the table used to perform the initial permutation of the key, to create the first round key. It is similar in design to the previously described permutation tables.

table_DES_PC2 - this array represents the order that specifies the 48 bit subkey after the circular shift has been performed on 56 bit roundkey.

table_DES_E - this array represents the table used to expand the right half of the data (that which is manipulated in the algorithm) to 48 bits to be XORed with the current sub-key. It is used in the mangler function (in this algorithm ComputeF).

table_DES_P - this array represents the table used to permute the output of the 8 S-boxes.

table_DES_S - this is a 2D array that is used to represent the S-boxes. Each subarray represents an individual S-box. These are used to perform substitutions on the 48 bit result of XORing the current sub-key and the PC2 table to get a 32-bit result. These are used in the Mangler function (ComputeF)

1. **void EncryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose);**

   This is the function called to perform encryption. This function takes a 56-bit encryption key, an output block of 64-bits for return data, an input block of 64-bits for input data, and a boolean flag if you want to run the function in verbose mode.

   This function starts by getting round key using from the provided key. Next an initial permutation on **inBlk** splits this input into two 32-bit blocks for **L** (left) and **R** (right). The function then loops though 16 rounds of substitution and permutation.

During each round, the roundKey is shifted left at least once. All rounds except 1,2,8, and 16 will perform an additional shift left on the roundKey. Next, the mangler function (**ComputeF**) is run on the right half (**R**) of the input using the roundKey. Then each bit in the result of the mangler function is XOR'd with it's corresponding bit in the left half (**L**) of the input block. The next step the round swaps the bits on the left (**L**) with the bits on the right (**R**).

After the 16 rounds are complete the left half is swapped with the right half one more time, and a final permutation is performed that combines the left half (**L**) with the right half (**R**). The result of the final permutation is stored in outBlk for return.

2. void DecryptDES(bool key[56], bool outBlk[64], bool inBlk[64], int verbose)

This is the function called to perform decryption. This function takes a 56-bit decryption key (Same as the encryption key), an output block of 64-bits for return data (in this case the original message, i.e. plaintext), an input block of 64-bits for input data (in this case the case the ciphertext), and a boolean flag if you want to run the function in verbose mode

This function, like EncryptDES, begins by first computing a round key from the provided key. Next the function splits the given input block into the Left (**L**) and Right (**R**) halves of 32 bits each. Then it also runs through 16 rounds of permutation and substitution. The difference between this function and the EncryptDES function is what occurs during each round.

During decryption, the function needs to perform the reverse of what occurs in encryption. The elegance of DES is that rather than perform the mangler function in reverse, it reverses the process by changing the order in which the substitutions and permutations occur (as well as performing a circular shift on the key in the opposite direction). First, the mangler function is run on the current right half of the input (**R**) using the current roundKey. The result is then XORed with the left half of the input block (**L**) bit by bit. The left and right halves of the block are then swapped. Finally a right circular shift (opposite of that performed in Encrypt DES) is performed on the key, once during rounds 1,2, 8, and 16 and twice during all other rounds.

After all 16 rounds have been completed, there is one final exchange between the **L** & **R** and a final permutation is performed which combines the two halves. The final result is then stored in the outBlk to be returned. The final result is the plaintext message (if the ciphertext supplied was the result of the EncryptDES function).

3. static void ComputeRoundKey (bool roundKey[56], bool key[56])

This function generates a roundkey. This operation makes use of table PC1 or PC2 which are called permutation Choice 1 and Permutation choice 2. 8 Parity bits are discarded by PC1 (table_DES_PC1) while preprocessing the key.

4. **Static void RotateRoundKeyLeft(bool roundKey[56]);**

   Here 56 bit Roundkey considered as 2 halves and rotated 1 bit to left. This function uses 2 temp bits to store 27^{th} and 55^{th} bit in roundkey and later assigned to 0^{th} and 28^{th} position after remaining 54 bits are shifted to left by a bit.

5. **Static void RotateRoundKeyRight(bool roundKey[56]);**

   Similar to above function here also 56 bit Roundkey considered as 2 halves and rotated 1 bit to right. This function uses 2 temp bits to store 0^{th} and 28^{th} bit in roundkey and later assigned to 27^{th} and 55^{th} position after remaining 54 bits are shifted to right by a bit.

6. **static void ComputeIP(bool L[32], bool R[32], bool inBlk[64]);**

   This function is used to permute the input 64 bits and later splits to two 32 bit halves. Here we use table_DES_IP to permute whose output bit (table_DES_IP[i]) equals input bit i. Later this 64 bit permuted bits split to [L] and [R] blocks. 0-31 bits goes to R block where as remaining 32-63 bits goes to L block.

7. **static void ComputeFP(bool outBlk[64], bool L[32], bool R[32]);**

   This function is similar to above function except it does in reverse. It combines the [L] and [R] blocks created above and does the final permutation using table_DES_FP. The final output would be a 64 bit assigned to outBlk.

8. **static void ComputeF(bool fout[32], bool R[32], bool roundKey[56]);**

   This is the mangler function as described in the slides. It expands the 32-bit right half **R** to 48-bits using **ComputeExpansionE**. Once the right half is expanded a **subkey** is obtained from the provided **roundKey**. This subkey is XOR'd with the expanded block of 48-bits obtained earlier. The block is then divided into 8 6-bit chunks and processed through the 8 previously defined S tables. After the S table processing/lookup the 48-bit block is been converted to 32-bits. The last step is to compute the permutation of these 32-bits using **ComputeP**. The result of the mangler function is made available in **fout**.

9. **static void ComputeP(bool output[32], bool input[32]);**

   This function computes the permutation of the S-Table results. This maps the 32-bits from **input** to the resulting 32-bit **output** table. The value of the bit at position ***i*** in the input is set to the position in output for the corresponding value defined in **table_DES_P[*i*].**

10. **static void ComputeS_Lookup(int k, bool output[4], bool input[6]);**

    Using the 8 S-tables in **table_DES_S** this function will map the 6-bits from **input** to an integer value. This integer value is used to find a corresponding value in the k^{th} table in **table_DES_S**. Once we have an integer from **table_DES_S** this value is converted to a 4-bit binary representation and returned using output.

11. **static void ComputePC2(bool subkey[48], bool roundKey[56]);**

    The function maps the 56 bit **roundKey** to a 48-bit **subkey** using the table defined in **table_DES_PC2**. It takes the first 48 bits from **roundKey** and assigns the corresponding value from **table_DES_PC2** to **subkey**. The result is returned in **subkey**.

12. **static void ComputeExpansionE(bool expandedBlock[48], bool R[32]);**

    This function expands the 32-bit block **R** to 48-bits. Each bit location (***i***) in **expandedBlock** is assigned a value in **R** at the location defined by **table_DES_E[*i*].**

13. **static void Exchange_L_and_R(bool L[32], bool R[32]);**

    This function exchanges the right and left halves ([L] and [R] blocks) of the data between rounds. This is performed because the algorithm exclusively operates on one half (the right) of the data at one time. Once the half-block passes through the algorithm, it is exchanged with the other half which is then passed through the algorithm. It essentially allows for the algorithm to operate on both halves of the data.

14. **static void DumpBin(char *str, bool *b, int bits)**This function is **non-essential** to the DES algorithm. It basically allows for verbose mode to be enabled allowing the user to see the results of each round during the encryption/decryption phases.