CS641: Chapter 4
Team name: team7 (chaos)

Komal Kalra (18111032) Riya James (18111054) Sristi Jaiswal (18111074)

March 5, 2019

Getting to the Cipher

We *enter*ed from the first chamber through an opening to the side of a lake. We *dive*d into the lake, came back up because of the cold, *dive*d in again and came across a wand-like object underwater. We came *back* up for air, and then dived in again to get a closer look at the object. We *pull*ed at it. Then we *enter*ed the first chamber again. Remembering the spirit from Chapter 3, we decided to go back to free it with our newly acquired wand. We went *back* to Chapter 3, and from the first chamber entered the next one, where the spirit was. We *wave*d our wand and freed the spirit. We then went on to chapter 4 and tried to *read* from the glass panel in the first chamber. The spirit translated for us.

Mapping the Code

The spirit said the mapping of the cipher text was in such a way that two characters were represented with a single byte. That is, each character was represented with 4 bits. Generating a few arbitrary encryptions, it was obvious that the coded text only used letters between f and u. In the absence of any other information, we decided the mapping had to be simple: f mapped to `0000`, g to `0001` and so on.

Chosen Plaintext Attack

We started with 6 round DES because 4 round DES seemed simple. We used the 4 round characteristic discussed in class i.e., we considered plaintext pairs with XOR value `0x405C0000 04000000`. Before applying the initial permutation, the XOR value would be `0x00009010 10005000`. We generated 90,000 pairs of plaintexts with this XOR value. After 4 rounds of encryption, the XOR of the texts would be `0x00540000 04000000` with a probability of 0.00038. At this point, we knew the output XOR of the fifth round with the same probability and the right half with probability 1.

(After applying the inverse final permutation on the output and *swapping* the left and right blocks:) We know that the left half of the output of the $6th$ round is the right half of the input to the $6th$ round. From the known values of the right half of the input to the $6th$ round, we computed both the output of the expansion box (`expand_out_1.txt`) and the XOR of the input to the S-box (`sbox_inxor.txt`).

Knowing the cipher text, we reverse mapped it to binary using the same assumed mapping of 4 bits per letter. We first applied the reverse final permutation, and then XORed the resulting right blocks of the pair with each other and with `0x04000000` (the XOR of left block of output of the $5th$ round). To this we applied the reverse of the round permutation function. This is the XOR of the output of the S-box for each pair (`sbox_outxor.txt`).

Knowing the XOR of the input and output to the S-box, we enumerated the 16 possible pairs of output values of the S-box that would satisfy the condition. For each of the 16 pairs we found which of the 4 possible input S-box values that could've generated the output also satisfied the XOR constraint. Each of these we XORed with the corresponding bits in `expand_out_1.txt` to get the possible key values. We counted the

possible keys for each S-box. In this way, we got the value of the key corresponding to S-boxes S1, S2, S5, S6, S7, S8.

We plugged the $6th$ round key to get the bit positions (in the 56-bit key) of the values we know. The remaining 20 bits (for two S-boxes we did not get a significant difference for any key from the other candidate keys) were bruteforced to find the complete key.

Programs

The sequence of programs that can be run to regenerate our entire process:

- `generate_input.ipynb`: Generates random input text pairs (`inputtexts.txt`) with the fixed XOR `0x00009010 10005000`.
- `generate_ciphertexts.sh`: Takes as input `inputtexts.txt` and generates the corresponding cipher texts (`ciphertexts.txt`).
- `cipher_processing.ipynb`: Takes as input `ciphertexts.txt` and generates `binciphertexts.txt`.
- `differential_propagation.ipynb`: Takes as input `binciphertexts.txt` and generates as output `sbox_inxor.txt`, `sbox_outxor.txt`, and `expand_out_1.txt`. (These contain the data explained above.)
- `reverse_sbox.ipynb`: Takes as input the three `.txt` files generated above and generates `keyfreq.txt`. This contains the frequencies of the possible keys for each of 8 S-boxes. From here we manually computed the most likely key for the $6th$ round and hardcoded the bits we obtained into the program `key_schedule.ipynb`.
- `key_schedule.ipynb`: Takes the $6th$ round key and generates the original 56 bit key, filling in 'X' for bits we don't know the value of.
- `candidate_keys.ipynb`: The key we got above we feed into `candidate_keys.ipynb` which generates `key_out.txt`. This contains a list of all keys that need to be brute forced.
- We used an implementation of DES we found online for bruteforcing and decrypting the password.

The Key and Getting the Password

The encrypted password: `ngffffqsrupmtfkjuhhkkmtptkspigfn` On decrypting we got: 109, 107, 119, 104, 104, 98, 104, 105, 103, 102, 48, 48, 48, 48, 48, 48. Because all the byte values were within the ASCII alphabet range (with the exception of '0'), we converted it to ASCII.

- (56-bit) Key: `0b0001010 1010101 1011010 0100010 1110011 1000000 0101000 0100010`
- Password: `mkwhhbfigf000000`