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 two 3-round characteristics v.i.z., we considered plaintext pairs with XOR values `0x00200008 00000400` and `0x40080000 04000000` (this is the XOR value that would go as input to the $1st$ round). Before applying the initial permutation, the XOR values would be `0x00000801 00100000` and `0x00008010 00004000` respectively. We generated $10,000$ pairs of plaintexts with these XOR values.

The logic: The XOR value at the end of the $3rd$ round is `0x04000000 40080000`. S-boxes S2, S5, S6, S7, S8 will have zero input in the $4th$ round and hence, their output will also be 0 with a probability of $\frac{1}{16}$. These outputs will be permuted and XORed with zeros as the left half of output of $3rd$ round is 0x04000000. These 0 values will be copied in the left half of 5th round, which will XOR with the right half of the ciphertext and then reverse permuted to get the output of Sboxes of $6th$ round. Hence, we will know the output of Sbox S2, S5, S6, S7 and S8 with a probability of 0.0625. Similarly, using the other differential in which the plaintext XOR value after the initial permutation is `0x00200008 00000400`, we can find the output XOR of Sboxes S1, S2, S4, S5, S6 with probability $\frac{1}{16}$.

After obtaining the cipher text from the server, 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 the XOR of the left half of the output of the $5th$ round. But because the corresponding bits in the left block was 0 for the particular S-boxes we were interested in, it didn't matter. To this we applied the reverse of the permutation that DES uses in the Feistel network. This would be the XOR of the output of the S-box for each pair (stored in `sbox_out_xor.txt`).

We got the right half of $5th$ round output from the right half of ciphertext(after inverse final permutation)

In the $6th$ round, knowing the XOR of the input and output to some of the S-boxes, we enumerated the 16 possible pairs of output values of those S-boxes 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 of `extended_block` to get possible key values.

From this, we got 30 bits of the key. The round-6 key we obtained from this characteristic is: `XXXXXX 010101 XXXXXX XXXXXX 110010 000101 010011 000`. Similarly we got 30 bits (corresponding to S-boxes S1, S2, S4, S5, and S6) of the key from the $2nd$ characteristic: `010101 010001 XXXXXX 010011 110010 000101 XXXXXX XXXXXX`. Here `X's` denote bits we didn't yet know the value of. We took it as confirmation of the correctness of our partial key that the bits corresponding to s-boxes S2, S5, and S6 matched.

From this combined partial $6th$ round key, we got the partial 56-bit key by applying the reverse of the key generation process (in `key_schedule.ipynb`) as `X00XX1X X01010X 101XX10 X10001X 1110011 X000000 01X10X0 010X010` (ignoring the parity bits of the key).

The remaining 14 bits are bruteforced to find the rest of the 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 values `0x00000801 00100000` and `0x00008010 00004000`.
- `generate_ciphertexts.sh`: Takes as input `inputtexts.txt` and generates the corresponding cipher texts from the server.
- `cipher_processing.ipynb`: Takes as input `ciphertexts.txt` and generates `binciphertexts.txt` (the binary representation of the ciphertexts using the assumed mapping).
- `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.
- `brute_force.ipynb`: Takes `candidate_keys.ipynb` as input and brute forces each of the candidate keys.

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 character range (with the exception of 0), we converted it to ASCII.

- (56-bit) Key: `0bX00XX1X X01010X 101XX10 X10001X 1110011 X000000 01X10X0 010X010`
- Password: `mkwhhbfigf000000`