MysteryTwister

THE CRYPTO CHALLENGE CONTEST

Twisting the Rubik's Cube

Katia Rezek, Maximilian Rademacher

December 2025

Foreword — Encryption Using Rubik’s Cubes

In this challenge, you will attack a symmetric cipher built on top of the **Feistel** network structure and a digital **Rubik’s Cube**, given only one plaintext/ciphertext pair. Sounds fun? Let’s go!

Preliminaries — Overview

Here are some details about the cipher:

- It uses a 16-byte secret **key** and can encrypt arbitrarily long messages.
- In this scenario, the cipher runs in ECB mode.
- The cipher uses the Feistel structure with **2** rounds, a custom **round function**, and a custom **key schedule**.
- The custom **round function** is based on a 3×3 Rubik’s Cube.
- The **plaintext** is padded to a multiple of 108 bytes and ingested in 108-byte blocks, meaning the left and right halves are 54 bytes each.
- You will be given the complete source code of the cipher.

Preliminaries — The Rubik’s Cube

Since the Feistel structure is sufficiently explained by online resources, we only need to explain the round function and the key schedule. As stated before, it is based on a 3×3 Rubik’s Cube. To use it for encryption, we first have to formalize its structure. We define a 3×3 Rubik’s Cube as having 6 distinct faces: **up, down, front, back, left, right**. See FIGURE 1 on page 3.

Now we can define a set of **fundamental moves** that can be performed on the cube. Each face can be rotated 90, 180, or 270 degrees clockwise. We use the following notation for a move: *U1* turns the up face 90 degrees clockwise, *L2* turns the left face 180 degrees clockwise, and *B3* turns the back face 270 degrees clockwise, for example.

For example, the series of moves *U1,U3* will not change the cube.

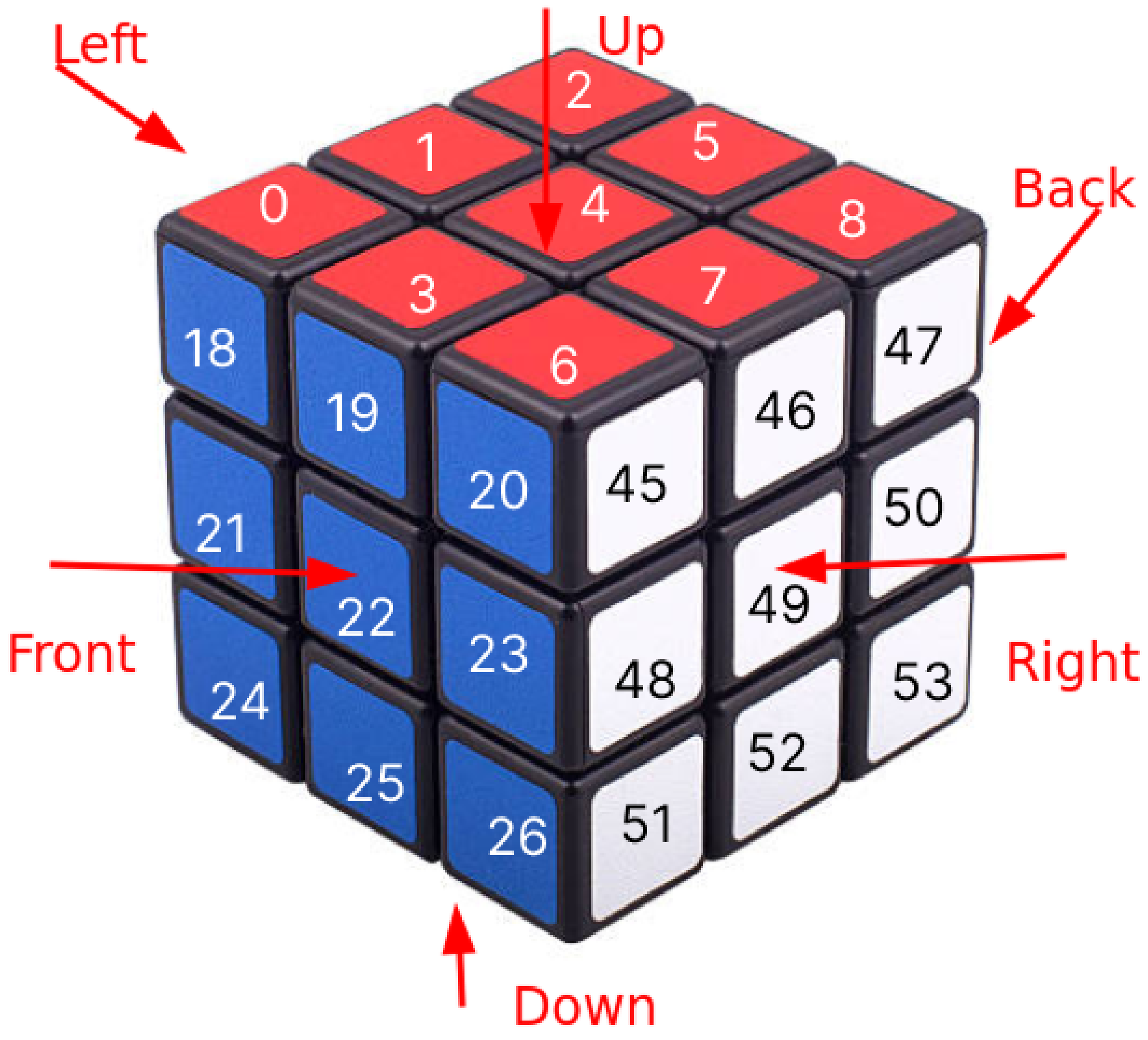

FIGURE 1: Faces of a Rubik's cube

We define the **state of the cube** as a one-dimensional list representing where each field of the cube is positioned. Here, we agree on a specific order for counting the fields of the cube. For each face, we count from top to bottom, left to right. Additionally, we agree that the cube is "filled" with values and read out (when getting its state as a 1D list) in the following order: **up (U), down (D), front (F), back (B), left (L), right (R)**.

The following numerical values are assigned to the positions on the 6 areas:

- U: 0 - 8
- D: 9 - 17
- F: 18 - 26

- B: 27 - 35
- L: 36 - 44
- R: 45 - 53

See LISTING 1 with the numbered fields.

For example, if we fill the Rubik's Cube with 0 to 53 in the order we agreed on, and perform the fundamental move *U1*, the top face will contain the following values: *6, 3, 0, 7, 4, 1, 8, 5, 2*

LISTING 1: Numbered fields (0–53) per face

```
up:      0  1  2
         3  4  5
         6  7  8

down:    9  10 11
         12 13 14
         15 16 17

front:   18 19 20
         21 22 23
         24 25 26

back:    27 28 29
         30 31 32
         33 34 35

left:    36 37 38
         39 40 41
         42 43 44

right:   45 46 47
         48 49 50
         51 52 53
```

The Round Function

LISTING 2 and FIGURE 2 show pseudocode for the round function used in the Feistel structure. The round function ingests half of the Feistel input (54 bytes) and the corresponding round key. Based on the round key, which is always 16 bytes, for example `k = 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff`, a list of rotations is built. This is done by indexing into a list of all possible fundamental moves with each byte of the key, modulo the length of the list. Then, the cube is populated with the 54-byte input in the order defined above. Next, the cube is rotated according to the previously built rotation list. Lastly, the state of the cube is read back into a 1D list, and each element is XORed with the corresponding key byte. Since the state is longer than the key itself, the algorithm wraps around the key when needed.

Example: a state `s`, e.g., starting with `[0x06, 0x03, 0x00, 0x07, …]`, has 54 bytes; with key `k = 00 11 22 … ff`, the XOR wraps to the start of k after 16 bytes. The implementation in Python looks as follows:

LISTING 2: Round function in Python

```python
def round_func(key, _input):

    # build the rotation key
    rot_key = [POSSIBLE_MOVES[b % len(POSSIBLE_MOVES)] for b in key]

    # populate the cube
    cube = Cube(list(_input))

    # do the rotations
    [cube.do_rotation(m) for m in rot_key]

    # xor the state with the key (wrap around)
    return [x ^ key[i % len(key)] for i, x in enumerate(cube.get_state
        ())]
```

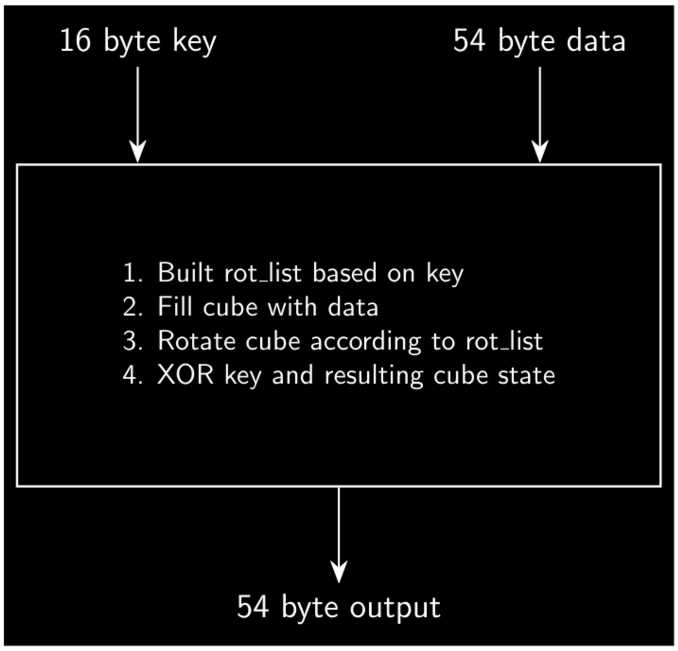
16 byte key
54 byte data
1. Built rot_list based on key
2. Fill cube with data
3. Rotate cube according to rot_list
4. XOR key and resulting cube state
54 byte output

FIGURE 2: Pseudocode for the round function

The Key Schedule

The key schedule is simple. Let k be the original 16-byte key. Then, $k_0 = k$ and $k_i = \mathrm{ror}(k_{i-1})$ for all $i \geq 1$, where ror denotes the byte-wise right rotation. In contrast to a right shift, the last byte is retained by prepending it to the left side of the rotated input.

The Challenge

You will be given exactly one plaintext/ciphertext pair. The key is unknown to you. Your task is to recover at least 12 out of 16 key bytes (our attack was able to recover that many—maybe you can do more?). The verification script running on the server will output 1 if you have at least 12 out of 16 key bytes correct.

Please submit solutions in a hex-encoded form like this:

```
ffeed????baa9988??665544332211??
```

where `??` represents an unknown key byte.

Additional Files

- `cube.py` — Rubik's Cube structure
- `roundfunction.py` — Round function implementation
- `algo.py` — Feistel-structure ECB implementation
- `challenge.py` — Plaintext/ciphertext pair