1. 第九轮注入故障

正确密文：

```
the_cipher = [0x66, 0xe9, 0x4b, 0xd4,
                    0xef, 0x8a, 0x2c, 0x3b,
                    0x88, 0x4c, 0xfa, 0x59,
                    0xca, 0x34, 0x2b, 0x2e]
```

错误密文：

```
faulty_cipher_1 = [0xf8, 0xe9, 0x4b, 0xd4,
                    0xef, 0x8a, 0x2c, 0x35,
                    0x88, 0x4c, 0x91, 0x59,
                    0xca, 0x50, 0x2b, 0x2e]
faulty_cipher_2 = [0x26, 0xe9, 0x4b, 0xd4,
                    0xef, 0x8a, 0x2c, 0x56,
                    0x88, 0x4c, 0x15, 0x59,
                    0xca, 0xb7, 0x2b, 0x2e]
```

要求恢复 K[0],K[13],K[10],K[7]。

部分正确密钥为：K[0] = 0xb4, K[13] = 0x8f

B4 11 52 8F

2. 第八轮注入故障

第一对密文对：

正确密文：

```
The_cipher = [0x66, 0xe9, 0x4b, 0xd4,
                    0xef, 0x8a, 0x2c, 0x3b,
                    0x88, 0x4c, 0xfa, 0x59,
                    0xca, 0x34, 0x2b, 0x2e]
```

错误密文：

```
faulty_cipher_1 = [0xfd, 0x77, 0x6e, 0x56,
                      0x79, 0xbd, 0xe2, 0xf5,
                      0x1d, 0xc6, 0x79, 0x25,
                      0xe3, 0xfd, 0x61, 0xc8]
faulty_cipher_2 = [0x6d, 0x6b, 0x8f, 0x2a,
                      0xd4, 0xba, 0x24, 0x3d,
                      0xeb, 0x3b, 0x4b, 0xe2,
                      0x4f, 0xe2, 0x02, 0xcf]
```

要求恢复第十轮全部十六个字节的密钥：

前四个字节的密钥分别为 K[0] = 0xb4,K[1] = 0xef, K[2] = 0x5b, K[3] = 0xcb

B4EF5BCB3E92E21123E951CF6F8F188E

第二对密文对：

正确密文：

```
the_cipher = [0x69, 0xc4, 0xe0, 0xd8,
                    0x6a, 0x7b, 0x04, 0x30,
```

```
                    0xd8, 0xcd, 0xb7, 0x80,
                    0x70, 0xb4, 0xc5, 0x5a]
```

错误密文：

```
faulty_cipher_1 = [0x0a, 0xfd, 0xd3, 0x71,
                       0xff, 0x34, 0xd1, 0xaf,
                       0x12, 0x96, 0x3c, 0x02,
                       0xb4, 0x06, 0xe8, 0x52]
faulty_cipher_2 = [0x52, 0x30, 0x76, 0x7e,
                   0x08, 0xcf, 0x4f, 0x19,
                   0x27, 0x2c, 0xc1, 0x5f,
                   0x87, 0x53, 0x82, 0xd5]
```

要求恢复第十轮全部十六个字节的密钥：

前四个字节的密钥分别为 K[0] = 0x13,K[1] = 0x11, K[2] = 0x1d, K[3] = 0x7f

13111D7FE3944A17F37A78B4D2B30C5

DFA 实现：

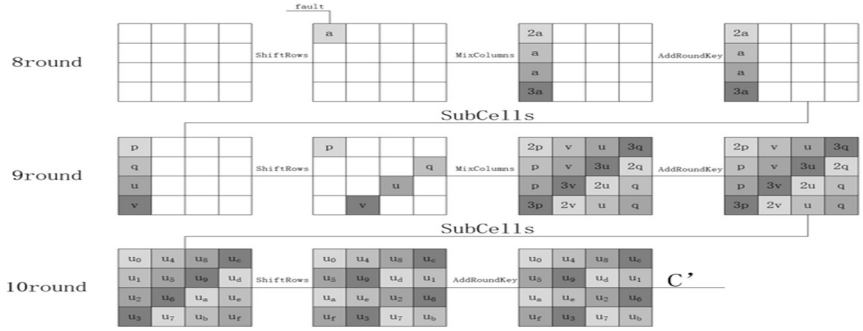
根据 https://eprint.iacr.org/2003/010.pdf 的描述，以及对 https://github.com/skyking94/DFA_AES/blob/master/main.py 和 https://github.com/Daeinar/dfa-aes 的参考

首先概括出 DFA 原理为：利用已知是第八轮或者第九轮注入位置未知，数量未知的故障，并且得到了第十轮的正常密文和两次错误密文，我们可以根据下图两个方程得到

$$S(X_0+2\varepsilon)=S(X_0)+\varepsilon_0 \qquad S(X_0)\oplus K_{10}(0)=C(0)$$
$$S(X_1+\varepsilon)=S(X_1)+\varepsilon_1 \qquad S(X_1)\oplus K_{10}(13)=C(13)$$
$$S(X_2+\varepsilon)=S(X_2)+\varepsilon_2 \qquad S(X_2)\oplus K_{10}(10)=C(10)$$
$$S(X_3+3\varepsilon)=S(X_3)+\varepsilon_3 \qquad S(X_3)\oplus K_{10}(7)=C(7)$$

（该图描述了在第九轮注入故障时，对第十轮 Key 进行部分恢复的公式）

如果我们要实现一次完整的针对 AES128 第十轮的 DFA 攻击，我们需要知道八个第九轮的错误密文，或者两个第八轮的错误密文，这个数量可以根据这张图理解：



如果故障注入在第九轮，一个故障，会影响到最后输出的 4 个字节，而在第八轮发生的话，会影响最后 16 个，过程为：1）第八轮影响四个字节，随后输出到第九轮 Sbox 部分；2）第九轮查表，进行字节替换，随后进行第九轮行变换等操作，影响范围扩大至 16 个字节；3）第十轮 Sbox 查表操作，最后进行行变换和轮密钥加法，得到错误密文 C‘

针对 Round9 攻击的计算：

```
const auto group0 = groupIntersections[0].candidates[0];
key[DFA::FaultIndex[0][0]] ^= DFA::forward_box[group0.values[0].values[0]];
key[DFA::FaultIndex[0][1]] ^= DFA::forward_box[group0.values[1].values[0]];
key[DFA::FaultIndex[0][2]] ^= DFA::forward_box[group0.values[2].values[0]];
key[DFA::FaultIndex[0][3]] ^= DFA::forward_box[group0.values[3].values[0]];

const auto group1 = groupIntersections[1].candidates[0];
key[DFA::FaultIndex[1][0]] ^= DFA::forward_box[group1.values[0].values[0]];
key[DFA::FaultIndex[1][1]] ^= DFA::forward_box[group1.values[1].values[0]];
key[DFA::FaultIndex[1][2]] ^= DFA::forward_box[group1.values[2].values[0]];
key[DFA::FaultIndex[1][3]] ^= DFA::forward_box[group1.values[3].values[0]];

const auto group2 = groupIntersections[2].candidates[0];
key[DFA::FaultIndex[2][0]] ^= DFA::forward_box[group2.values[0].values[0]];
key[DFA::FaultIndex[2][1]] ^= DFA::forward_box[group2.values[1].values[0]];
key[DFA::FaultIndex[2][2]] ^= DFA::forward_box[group2.values[2].values[0]];
key[DFA::FaultIndex[2][3]] ^= DFA::forward_box[group2.values[3].values[0]];

const auto group3 = groupIntersections[3].candidates[0];
key[DFA::FaultIndex[3][0]] ^= DFA::forward_box[group3.values[0].values[0]];
key[DFA::FaultIndex[3][1]] ^= DFA::forward_box[group3.values[1].values[0]];
key[DFA::FaultIndex[3][2]] ^= DFA::forward_box[group3.values[2].values[0]];
key[DFA::FaultIndex[3][3]] ^= DFA::forward_box[group3.values[3].values[0]];
```

利用 Round9 的检测递归，实现 round8 的攻击：

```
static inline auto convert_r8_fault(u128 r8_fault, u128 r9_reference)
{
    const auto masks = DFA::blend_mask();
    static_assert(masks.size() == 4);

    std::array<u128, 4> r9_faults;

    for (auto i = 0u; i < sizeof(u128); ++i)
    {
        r9_faults[0][i] = masks[0][i] ? (r8_fault[i]) : (r9_reference[i]);
        r9_faults[1][i] = masks[1][i] ? (r8_fault[i]) : (r9_reference[i]);
        r9_faults[2][i] = masks[2][i] ? (r8_fault[i]) : (r9_reference[i]);
        r9_faults[3][i] = masks[3][i] ? (r8_fault[i]) : (r9_reference[i]);
    }

    return r9_faults;
}
```

对 Round9 的攻击进行预先判定，即对计算出的 X0 到 X3 进行计算

```
static inline std::array<FaultCandidateList, 4> solve_r9_candidates(const T& r9_faults,
                                                                    const u128 ref) noexcept
{
    std::array<FaultCandidateList, 4> keyCandidates;

    for (const auto& r9_fault : r9_faults)
    {
        DifferentialFault<DFA> fault{ r9_fault, ref };

        if (fault.is_group_affected( group: 0) && !keyCandidates[0].solved())
        {
            const auto newCandidates = fault.candidates_for_group( group: 0);
            intersect_candidates( &: keyCandidates[0],  new_candidates: newCandidates);
        }

        else if (fault.is_group_affected( group: 1) && !keyCandidates[1].solved())
        {
            const auto newCandidates = fault.candidates_for_group( group: 1);
            intersect_candidates( &: keyCandidates[1],  new_candidates: newCandidates);
        }

        else if (fault.is_group_affected( group: 2) && !keyCandidates[2].solved())
        {
            const auto newCandidates = fault.candidates_for_group( group: 2);
            intersect_candidates( &: keyCandidates[2],  new_candidates: newCandidates);
        }

        else if (fault.is_group_affected( group: 3) && !keyCandidates[3].solved())
        {
            const auto newCandidates = fault.candidates_for_group( group: 3);
            intersect_candidates( &: keyCandidates[3],  new_candidates: newCandidates);
        }
    }

    return keyCandidates;
}
```