密码分析学

第 4 次大作业

第六组

2021 年 11 月 19 日

目录

1 Task 1 **3**
1.1 线性分析原理 . . . 3
1.1.1 一轮加密 . . . 3
1.1.2 二轮加密 . . . 4
1.2 自动搜索高概率线性壳 . . . 4
1.2.1 获得线性分布表 . . . 4
1.2.2 一轮线性逼近式 . . . 5
1.2.3 多轮线性逼近式 . . . 5
1.3 通过高概率线性壳筛选秘钥 . . . 6
1.3.1 算法原理 . . . 6
1.3.2 算法测试 . . . 7
1.3.3 成功率 . . . 8
1.4 代码实现 . . . 9

2 Task2 **10**
2.1 零相关线性分析原理 . . . 10
2.2 自动搜索最长轮数的零相关线性壳 . . . 11
2.3 代码实现 . . . 14

摘要

组员分工

王文凯 201900401024：task1 的 python 代码实现，task1 密钥恢复报告书写

初婧雯 201900460011：task1 原理报告书写、task2 原理报告书写

周睿泽 201900460023：task1 的 python 代码实现，task1 自动化搜索报告书写，报告整理

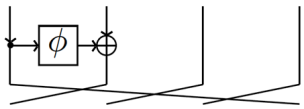
林若妍 201900460027：task1 的 python 代码实现，报告整理、latex 书写

张自平 201900460035：task1、task2 的代码实现，task2 自动化搜索报告书写、报告整体校对

作业要求

1）根据课上介绍的方法，对 5 轮 CipherFour（注意，课上是 CIPHERD 算法）算法进行线性分析。（明密文对应关系同作业二的 txt 文件，该文件依次列出了明文从 0 开始对应的密文取值）。提交资料：线性分析过程说明文档，包含 S 盒线性近似表，每轮线性特征，偏差，攻击采用的线性壳轮数及概率 p，密钥恢复攻击过程及数据（分别测试已知 2|p-1/2|-2、8|p-1/2|-2 个明密文时可恢复的密钥集合）；代码。

2）找出如下图所示的非平衡 Feistel 结构的最长轮数的零相关线性壳，其中函数 ϕ 可逆。提交资料：结果及理由说明文档。



1 Task 1

1.1 线性分析原理

线性分析是一种已知明文攻击。异或加密：已知 (m,c)，利用 $k = m \oplus c$ 可以解出密钥 k。因此可以利用不随机现象即明文和密文的异或为固定值来求解密钥。

1.1.1 一轮加密

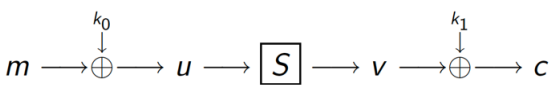
- 明文 m，密文 c，密钥 $k = (k_0, k_1)$，k_0, k_1 均为 n-bit，$S : \{0,1\}^n \to \{0,1\}^n$ 的置换
- 加密：$c = S(m \oplus k_0) \oplus k_1$
- 根据下图可以得到关系式：$u = m \oplus k_0, v = S(u), c = v \oplus k_1$

  假设 m，c，k_0，k_1 之间存在线性关系：$\alpha \cdot u = \beta \cdot v \; Pr(\alpha \cdot u = \beta \cdot v) = p$

  将上述关系式转换为：

$$\alpha \cdot m \oplus \beta \cdot c = \alpha \cdot k_0 \oplus \beta \cdot k_1, \;\; Pr(\alpha \cdot m \oplus \beta \cdot c = \alpha \cdot k_0 \oplus \beta \cdot k_1) = p$$

- 有效性：$|p - \frac{1}{2}|$

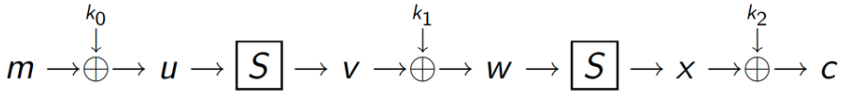


1.1.2 二轮加密

- 明文 m，密文 c，密钥 $k=(k_0,k_1,k_2)$，k_0，k_1，k_2 均为 n-bit，$S:\{0,1\}^n\to\{0,1\}^n$ 的置换
- 加密：$c=S(S(m\oplus k_0)\oplus k_1)\oplus k_2$
- 根据下图可以得到关系式：$\alpha\cdot m=\alpha\cdot k_0\oplus\alpha\cdot u,\beta\cdot v=\beta\cdot k_1\oplus\beta\cdot w,\gamma\cdot x=\gamma\cdot k_2\oplus\gamma\cdot c$
  假设存在线性关系：$\alpha\cdot u=\beta\cdot v$ 且 $Pr(\alpha\cdot u=\beta\cdot v)=p_1$ $\beta\cdot v=\gamma\cdot x$ 且 $Pr(\beta\cdot v=\gamma\cdot x)=p_2$
  那么可以将上述关系式转化为：

$$\alpha\cdot m\oplus\gamma\cdot c=\alpha\cdot k_0\oplus\beta\cdot k_1\oplus\gamma\cdot k_2$$

$$Pr(\alpha\cdot m\oplus\gamma\cdot c=\alpha\cdot k_0\oplus\beta\cdot k_1\oplus\gamma\cdot k_2)=\frac{1}{2}+2(p_1-\frac{1}{2})(p_2-\frac{1}{2})$$



1.2 自动搜索高概率线性壳

线性分析秘钥恢复算法，关键在于找到高概率的线性壳，但是线性壳的寻找是困难的，因为一对线性壳可能对应多条路径，每条路径各不相同，路径的和才是线性壳真正的概率。我们知道影响线性路径概率的只有非线性的 S 盒，P 置换只起到比特扩散的作用，那么我们首先分析一下 S 盒的线性分布表：

1.2.1 获得线性分布表

构造线性近似表 LAT，遍历所有 (α,β), 计算满足 $\alpha\cdot x=\beta\cdot S(x)$ 的 x 个数 $N_s(\alpha,\beta)-2^3$。表格中数字的绝对值越大，对应线性逼近式越有效。获得线性分布表代码如下:

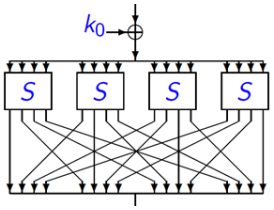
```
'''计算S盒的线性分布表和概率分布表'''
def get_S_linear_table(Abs=0):
    for alpha in range(16):
        for beta in range(16):
            for x in range(16):
                if JS(alpha&x) == JS(beta&S(x)):
                    S_table[alpha][beta]+=1
    for i in range(16):
        for j in range(16):
            if Abs:
                S_table_p[i][j] = abs(S_table[i][j])/16. # 得到加权有向图的邻接矩阵
            else:
                S_table_p[i][j] = S_table[i][j]/16.
```

由此得到如下线性分布表：

S盒线性分布表

α\β	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.125	0.125	0.0	0.25	-0.125	0.125	0.0	0.125	0.0	-0.25	-0.125	0.125	0.0	0.0	0.125
2	0.0	0.125	0.0	0.125	0.0	0.125	0.25	-0.125	0.125	0.0	0.125	0.0	-0.125	-0.25	0.125	0.0
3	0.0	0.0	0.125	-0.125	0.0	0.0	0.125	0.375	0.0	0.0	0.125	-0.125	0.0	0.0	0.125	-0.125
4	0.0	-0.125	0.125	0.0	-0.25	-0.125	-0.125	0.0	0.125	0.0	0.0	-0.125	0.125	-0.25	0.0	0.125
5	0.0	0.0	-0.25	0.0	0.0	-0.25	0.0	0.0	0.0	-0.25	0.0	0.0	0.0	0.0	0.25	0.0
6	0.0	0.0	-0.125	-0.125	0.0	-0.25	0.125	-0.125	0.0	0.25	0.125	-0.125	0.0	0.0	-0.125	-0.125
7	0.0	-0.125	0.0	-0.375	0.0	0.125	0.0	-0.125	0.125	0.0	-0.125	0.0	-0.125	0.0	0.125	0.0
8	0.0	0.25	0.0	0.0	-0.25	0.0	0.0	0.0	0.25	0.0	0.0	0.0	0.0	0.25	0.0	0.0
9	0.0	0.125	-0.125	0.0	0.0	0.125	-0.125	0.0	-0.125	0.25	0.0	-0.125	0.125	0.0	0.25	0.125
a	0.0	-0.125	0.0	0.125	0.0	-0.125	0.0	0.125	0.125	0.25	-0.125	0.25	-0.125	0.0	0.125	0.0
b	0.0	0.0	-0.125	-0.125	0.0	0.0	0.125	0.125	0.0	0.0	0.125	0.125	0.0	0.0	-0.125	0.375
c	0.0	0.125	0.125	0.0	0.0	-0.125	-0.125	0.0	-0.125	0.0	0.0	-0.125	-0.375	0.0	0.0	0.125
d	0.0	0.0	0.0	0.0	-0.25	0.0	0.25	0.0	-0.25	0.0	-0.25	0.0	0.0	0.0	0.0	0.0
e	0.0	0.25	-0.125	-0.125	0.0	0.0	-0.125	0.125	0.0	0.0	-0.125	0.125	0.0	-0.25	-0.125	-0.125
f	0.0	-0.125	-0.25	0.125	0.0	0.125	0.0	0.125	0.125	0.0	-0.125	-0.25	-0.125	0.0	-0.125	0.0

1.2.2 一轮线性逼近式



S 盒中存在线性逼近式及偏差如下:$(\alpha_1,\alpha_2,\alpha_3,\alpha_4) \xrightarrow{S} (\beta_1,\beta_2,\beta_3,\beta_4)$，$\epsilon(\alpha_i \xrightarrow{S_i} \beta_i) = \epsilon_i$

P 置换中存在线性逼近式如下：$(\beta_1,\beta_2,\beta_3,\beta_4) \xrightarrow{P} (\gamma_1,\gamma_2,\gamma_3,\gamma_4)$

因此一轮加密及偏差：$(\alpha_1,\alpha_2,\alpha_3,\alpha_4) \xrightarrow{1-r} (\gamma_1,\gamma_2,\gamma_3,\gamma_4)$

$\epsilon((\alpha_1,\alpha_2,\alpha_3,\alpha_4) \xrightarrow{1-r} (\gamma_1,\gamma_2,\gamma_3,\gamma_4)) = \epsilon((\alpha_1,\alpha_2,\alpha_3,\alpha_4) \xrightarrow{S} (\beta_1,\beta_2,\beta_3,\beta_4))$

完全取决于 S 盒的线性逼近式的偏差。查找线性分布表，较高的偏差绝对值对应的线性逼近式有如 $c \to c$ 等，偏差绝对值 ϵ 是 0.375，即 $\epsilon(c \xrightarrow{1-round} c) = 0.375$。

1.2.3 多轮线性逼近式

和差分路径一样，一轮最优并不能保证多轮最优，尤其在 P 置换的影响下，某些该概率的一轮线性掩码，会因为非零比特扩散到其他 S 盒上，而导致后续路径的概率大幅下降。多轮线性逼近式的概率要使用堆积引理计算：

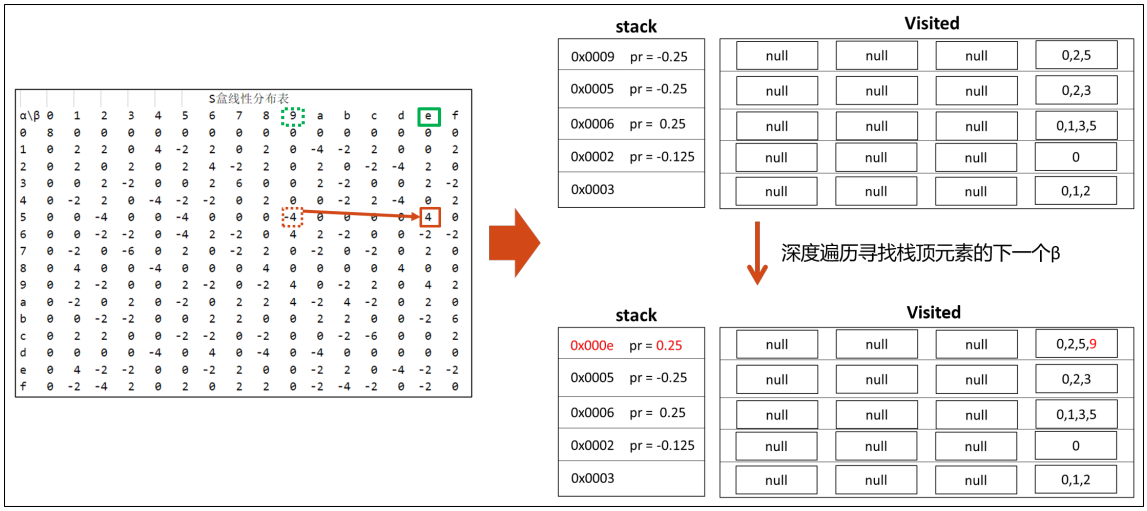
$$P(X_1 \oplus X_2 \oplus ... \oplus X_n) = \frac{1}{2} + 2^{n-1} \prod_{i=1}^{n} (p_i - \frac{1}{2})$$

要得到**多轮的高概率线性壳**，最简单的方法就是利用穷举所有的输入掩码，和其后续的所有路径找到最优线性壳。类似最优差分路径的搜索，我们将线性分布表看做一个**加权有向图**。而寻找一定轮数的线性路径的过程，就是对线性分布表进行深度优先遍历的过程。

但是整个过程耗时过长，必须考虑剪枝以减少时间消耗。最简单的剪枝就是控制输入掩码 α，在差分路径的搜索过程中，我们探究过 P 置换对路径概率的影响，所以首要的就是控制输入掩码非零 S 盒的个数。

其次，在尝试运行后发现，线性分布表上很多低概率的路径，其实完全可以舍弃掉，比如 ± 0.125 的路径，舍弃这些路径对最终的筛选结果并不会有明显影响。

对线性分布表做 DFS 深度优先遍历的流程示意图如下：



1.3 通过高概率线性壳筛选秘钥

1.3.1 算法原理

1. 找到偏差比较大的线性壳，线性壳的选取主要依赖上文的自动搜索算法，根据自动搜索算法，可以找到下表中的线性壳。

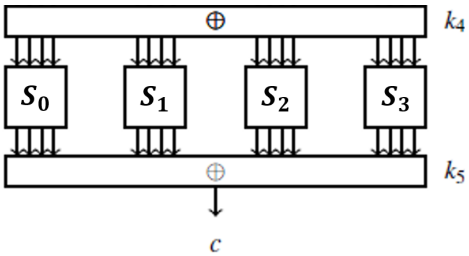
2. 根据线性壳中的**相关 S 盒**，我们可分别求得 $K_{0,0}, K_{0,1}, K_{0,2}, K_{0,3}$，以下表中第二行数据为例，相关 S 盒只有 S_0，可用于恢复密钥 $K_{0,0}$。

3. 对于每一个可能的密钥 $K_0^i \in \{0, 1, 2...15\}$，初始化一个计数器 $T_0^i = 0$

4. 对每个 K_0^i，对所选的所有明密文对进行以下操作（其中 $f(c, K_0^i) = S^{-1}(c_0 \oplus K_0^i)$）：

   计算 $y^i = f(c, K_0^i)$，如果 $\alpha \cdot m \oplus \beta \cdot y^i = 0$，那么 $T_0^i := T_0^i + 1$。

5. 计算 $|T_0^i - \frac{N}{2}|$，取其中的最大值作为恢复出的密钥。



相关 S 盒	α	β	ϵ
S_0	0xd000	0x4000	0.046875
S_1	0x8000	0x0400	0.046875
S_0,S_2	0x1000	0x4040	0.0390625
S_0,S_2	0x8000	0x1010	0.0390625
S_3	0xd000	0x0004	0.0625
S_0,S_3	0x0880	0x4004	0.048828125
S_1,S_3	0x0008	0x0404	-0.04736328125
S_1,S_3	0x8000	0x0404	-0.0936279296875

以上线性壳的选取是经过多次尝试，选取的效果较好的线性壳，有些线性壳虽然具有很高概率，但是在筛选秘钥时，发现并不能很好的起到区分器的作用，经过多次尝试多个高概率的线性壳，选择如上几个作为秘钥恢复算法的线性壳。

伪代码示意如下：

Require

1. N chosen plaintext-ciphertext pairs obtained with an unknown user-supplied key K
2. A **LINEAR HULL** with the $InputMask = \alpha$，$OutputMask = \beta$，and the $Deviation = \epsilon$

Algorithm

1. For every possible key k^i，Initialize a counter $T_0^i = 0$
2. For every guessed k^i，do：

   For each pair (m, c)，do：

   $y^i \leftarrow f(c, k^i)$

   If $\alpha \cdot m \oplus \beta \cdot y^i = 0$，Then

   $T_0^i := T_0^i + 1$
3. Take the maximum $|T_0^i - \frac{N}{2}|$ and its corresponding k^i

Data complexity

N chosen plaintext-ciphertext pairs

1.3.2 算法测试

主要从 2 个方面来观察密钥恢复算法的好坏：

1）计数器最大值对应的即为正确密钥的次数 T_1

2）正确密钥在最大的前五个计数器对应的密钥集合中的次数 T_2

其中 N 为明文数量

重复 100 次密钥恢复算法，得到以下表格：

密钥$K_{5,0}$ 的恢复

N	$2*\epsilon^{-2}$	$8*\epsilon^{-2}$
T_1	76	100
T_2	96	100

密钥$K_{5,1}$ 的恢复

重复以上步骤，得到如下表格：

N	$2*\epsilon^{-2}$	$8*\epsilon^{-2}$
T_1	98	100
T_2	100	100

密钥$K_{5,2}$ 的恢复

N	$2*\epsilon^{-2}$	$8*\epsilon^{-2}$
T_1	18	23
T_2	65	90

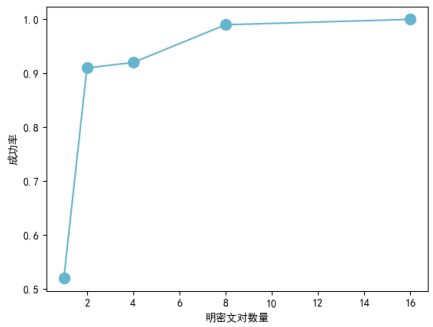
密钥$K_{5,3}$ 的恢复

N	$2*\epsilon^{-2}$	$8*\epsilon^{-2}$
T_1	20	27
T_2	49	90

可以看到密钥 $K_{5,0}$ 与 $K_{5,1}$ 的恢复效果比较好，而密钥 $K_{5,2}$ 与 $K_{5,3}$ 难以得到区分。

1.3.3 成功率

随着明文数量的增加，恢复效果越来越好，以 $K_{5,0}$ 为例，可做出下图（其中横坐标单位为 ϵ^{-2}）：



1.4 代码实现

```
'''寻找比较好的线性特征路径'''
def getgoodpath(S_num):
    GLP = []     # good linear path
    clear_visited(-1)
    #print_S_linear_table(S_table_p)
    '''预设概率，如果当前栈中的路径堆积概率小于这个值，那就舍弃当前路径，
       主要作用在剪枝，次作用是筛选,因为高概率线性路径本身就是高概率的而不是靠小概率的路径积累出来的'''
    Pr = Duiji([0.1]*4)
    for alpha in range(1,65536):
        alpha_tmp = "{:0>4}".format(hex(alpha)[2:])
        if alpha_tmp.count('0')<2:              # 控制输入线性掩码的非零位置
            continue
        stack = []
        stack.append(alpha)
        flag = 1
        pr_stack = [1]
        while flag:                             # 开始寻找 线性壳头为alpha的所有高概率线性路径
            pathpr = 0                          # 储存当前栈上的路径的概率乘积，需要使用堆积引理
            '''核心函数，stack[-1]是栈顶元素，函数返回下一个alpha过S盒后的beta,以及概率p'''
            beta,pr = S_linear(stack[-1],len(stack))

            if beta:  # 如果有beta输出
                pr_stack.append(pr)                 # 将概率压栈
                pathpr = Duiji(pr_stack[1:])        # 计算当前路径的概率乘积，可能是一个负数
                if abs(pathpr) > Pr:                # 如果当前路径的绝对概率已经小于一个预设值Pr，那么就抛弃当前路径，也是剪枝的思想
                    stack.append(P(beta))           # 如果概率比预设Pr大，就压入栈
                else:
                    pr_stack.pop()                  # 弹出栈顶元素的概率
                    tmp = P_inv(stack.pop()) & 0x000f  #加入到栈顶的是P置换之后的元素，需要经过逆置换再加入Visited
                    pr_stack.pop()                  # stack.pop()之后一定要跟一个pr.pop

                    '''更新已访问集合，把弹出的栈顶元素的最后一个位置加入已访问，在S_diff函数中就可以避免重复访问或漏掉元素'''
                    Visited[len(stack)][0].append(tmp)
                    clear_visited(len(stack))       # 清空栈高位的已访问集合，因为栈底层的元素变动了，所以栈顶方向的元素需要重新遍历
            else:
                '''如果当前栈顶元素alpha的所有beta都已经遍历过了，返回值就是False，进入else，栈弹出一个元素，相当于回退'''
                tmp = P_inv(stack.pop())&0x000f
                pr_stack.pop()
                Visited[len(stack)][0].append(tmp)
                clear_visited(len(stack))           # 同上述作用

            if len(stack)==R+1:
                '''栈的长度已经达到R轮加密的要求，可以加入GLP，同时将当前栈顶元素的最后4bit加入相应的已访问队列'''
                str_beta = "{:0>4}".format(hex(P(beta))[2:])
                if str_beta.count('0') > 1 and str_beta[S_num] != '0':
```

```
                    GLP.append((stack[:],pathpr))    # 这里还要筛选一下beta，去除beta中第S_num个S
盒为0的输出差分

                tmp = beta & 0x000f
                Visited[-1][0].append(tmp)  # 更新已访问集合，
                stack.pop()
                pr_stack.pop()

            elif len(stack)==0:             # 如果当前alpha的所有路径都已经访问过了，栈就会回退为
空，然后我们结束while
                flag=0

    # 统计线性壳特征
    linear_pr = dict()
    for path in GLP:
        alpha,beta,pr_ = path[0][0],path[0][-1],path[-1]
        tmp = linear_pr.get((alpha,beta))
        if tmp:
            linear_pr[(alpha,beta)] += pr_
        else:
            linear_pr[(alpha,beta)] = pr_
    path_order = sorted(linear_pr.items(), key=lambda x: abs(x[1]),reverse=True)

    k = 10
    print("\n--------------从左向右，第{}个S盒高概率线性路径共{}条，最优的{}个线性壳如下
--------------".format(S_num+1,len(GLP),k))
    i=0
    for item in path_order:
        print(" ={}，  ={}，Pr={}".format(hex(item[0][0]),hex(item[0][1]),item[-1]))
        i += 1
        if i>k:
            break
    return list(path_order[0])
```

2 Task2

2.1 零相关线性分析原理

线性分析是找到线性关系，让偏差 $|\epsilon(\alpha,\beta)| = |p(\alpha,\beta)-\frac{1}{2}|$ 尽可能大，从而来获得正确密钥。而零相关线性是通过找相关度为零即偏差 $\epsilon(\alpha,\beta) = 0$ 来获得正确密钥。因此零线性相关就是利用与密钥取值无关的、相关度一定为 0 的线性壳找正确密钥。

我们利用如下方式找到相关度为 0 的线性壳，即从头部和尾部分别利用两条概率为 1 的线性逼近式向中间推，直到遇到一个矛盾点，这样就找到了一个零相关线性的模型。至于矛盾点，通常是利用概率为 1 的线性掩码的传播规则来寻找的，即：

概率为 1 的线性掩码的传播规则：

- 异或运算（XOR）：输入掩码分别为 α_1、α_2，输出掩码为 β，则 $\alpha_1 = \alpha_2 = \beta$

- 分支运算（Branching）：输入掩码为 α，输出掩码分别为 β_1 β_2，则 $\beta_1 \oplus \beta_2 = \alpha$
- 线性映射（Linear Map）：输入掩码为 α，输出掩码为 β，则 $\alpha = M^T\beta$
- 非线性映射（Linear Map）：输入掩码为 $\alpha \neq 0$，则输出掩码为 $\beta \neq 0$

得到各部件的矛盾点：

- 异或运算（XOR）：输入掩码分别为 α_1、α_2，输出掩码为 β，但 $\alpha_1 = \alpha_2 = \beta$ 至少一个等式不成立
- 分支运算（Branching）：输入掩码为 α，输出掩码分别为 β_1、β_2，则 $\beta_1 \oplus \beta_2$ α
- 线性映射（Linear Map）：输入掩码为 α，输出掩码为 β，则 $\alpha \neq M^T\beta$
- 非线性映射（Linear Map）：输入掩码为 α 和输出掩码为 β ，有且仅有一个为零

因此利用 2 条概率为 1 的线性逼近式和矛盾点，找到对所有明密文计算出的相关度始终为 0 的密钥，这个密钥就是正确密钥。

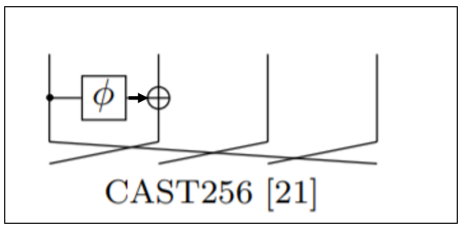
2.2 自动搜索最长轮数的零相关线性壳

零相关线性分析的关键是找到正向和逆向的概率为 1 的线性掩码路径，然后找到路径的矛盾点。第一步就是寻找传播概率为 1 的线性掩码传播规则。

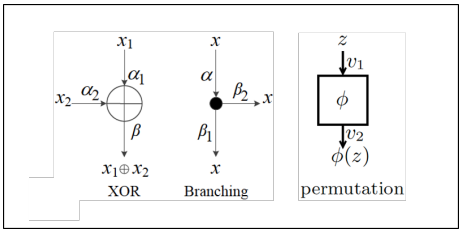
我们参考不可能差分中的分析方式，先将掩码空间进行分类，分为 $K = \{0, 1, 1^*, 2^*, ?\}$ 的五个集合，意义如下：

集合名	代表意义
0	0 掩码
1	非零且取值不固定的掩码
1*	非零且取值固定的掩码
2*	$2^* = 1 \oplus 1^*$
?	不确定的掩码

线性传播中，主要部件有分支、异或、线性盒和非线性盒四种变换，在 task2 的非平衡 Festial 结构中，出现了异或和非线性盒，我们就针对考虑这三种变化下，上述 5 种分类的集合间的转换：



CAST256 [21]



异或运算

- 不妨设 $\alpha_1 = 0$，$\alpha_2 = k$ where $k \in K = \{0, 1, 1^*, 2^*, ?\}$，那么 $\alpha_1 \cdot x_1 \oplus \alpha_2 \cdot x_2 = 0 \oplus k \cdot x_2 = k \cdot x_2$，我们只考虑线性掩码的变换，就有 $0 \oplus k = k$。
- 不妨设 $\alpha_1 =?$，$\alpha_2 = k$ where $k \in K = \{0, 1, 1^*, 2^*, ?\}$，那么 $\alpha_1 \cdot x_1 \oplus \alpha_2 \cdot x_2 =? \oplus k \cdot x_2$，我们知道不确定的掩码和其他任何掩码的运算都应该是不确定的。我们只考虑线性掩码的变换，就有 $? \oplus k =?$。
- 如果 $\alpha_1 = 1, \alpha_2 = 1$，那么 $\alpha_1 \cdot x_1 \oplus \alpha_2 \cdot x_2 = 1 \cdot x_1 \oplus 1 \cdot x_2$，因为 1 集合是取值不确定的非零掩码，所以两个掩码点乘之后的异或不能确定是否非零，也不能确定所以是不确定差分。即 $1 \oplus 1 =?$。
- 如果 $\alpha_1 = 1, \alpha_2 = 1^*$，那么 $\alpha_1 \cdot x_1 \oplus \alpha_2 \cdot x_2 = 1 \cdot x_1 \oplus 1^* \cdot x_2$，因为 1 集合是取值不确定的非零掩码，1是取值确定的掩码，所以 $1 \oplus 1^* \neq 1^*$，所以结果不是?，根据集合定义，结果为 $1 \oplus 1^* = 2^*$。
- 如果 $\alpha_1 = 1, \alpha_2 = 2^*$，那么 $\alpha_1 \cdot x_1 \oplus \alpha_2 \cdot x_2 = 1 \cdot x_1 \oplus 2^* \cdot x_2 = 1 \cdot x_1 \oplus 1 \cdot x_2 \oplus 1^* \cdot x_2 = ? \oplus 1^* =?$，就有 $1 \oplus 2^* =?$。
- 同理还有 $2^* \oplus 2^* =?$，$1^* \oplus 2^* =?$，$1^* \oplus 2^* =?$

非线性 F 函数

类比不可能差分中的分析，我们定义运算 "$\circ$"，有 $F(k) = k \circ 1_F$，$0 = k \circ 0$，$k = k \circ 1$。对于分类集合 $K = \{0, 1, 1^*, 2^*, ?\}$，考虑"$k \circ 1_F$" 的取值：

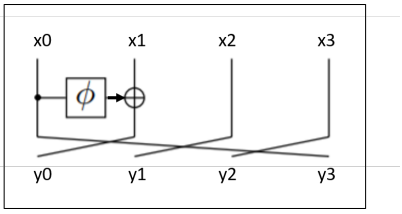
- 如果 $k = 0 \to F(0) = 0 \to 0 \circ 1_F = 0$
- 如果 $k = 1 \to F(1)$，非零且取值不固定的掩码 α 经过 S 盒之后，根据 S 盒的线性分布表，我们可以知道，会变成非零且取值不固定的掩码 β，所以 $1 \circ 1_F = 1$。
- 如果 $k = 1^* \to F(1^*)$，非零且取值固定的掩码 α 经过 S 盒之后，根据 S 盒的线性分布表，我们可以知道，会变成非零且取值不固定的掩码 β，所以 $1^* \circ 1_F = 1$。
- 如果 $k = 2^* \to F(2^*) = F(1 \oplus 1^*)$，$1 \oplus 1^*$ 是有可能为 0 的，根据 S 盒的线性分布表，我们可以知道，不确定的掩码，所以 $2^* \circ 1_F =?$。
- 如果 $k =? \to F(?) =?$，$?^* \circ 1_F =?$。

两种运算的规则统计如下：

” $\oplus$ ” 运算	” $\circ$ ” 运算
$0 \oplus k = k$	$k \circ 0 = 0$
$1 \oplus 1 = ?\ 1 \oplus 1^* = 2^*\ 1 \oplus 2^* = ?$	$k \circ 1 = k$
$1^* \oplus 1^* = ?, 1^* \oplus 2^* = ?$	$0 \circ 1_F = 0, 1 \circ 1_F = 1$
$2^* \oplus 2^* = ?$	$1^* \circ 1_F = 1, 2^* \circ 1_F = ?$
$? \oplus k = k$	$? \circ 1_F = ?$

寻找最长的线性路径：

一轮加密过程如下：



有如下关系式：

$$(x_0, x_1, x_2, x_3) \xrightarrow{R} (y_0, y_1, y_2, y_3) \Longrightarrow \begin{cases} y_0 = F(x_0) \oplus x_1 = x_0 \cdot 1_F \oplus x_1 \cdot 1 \oplus x_2 \cdot 0 \oplus x_3 \cdot 0 \\ y_1 = x_2 = x_0 \cdot 0 \oplus x_1 \cdot 0 \oplus x_2 \cdot 1 \oplus x_3 \cdot 0 \\ y_2 = x_3 = x_0 \cdot 0 \oplus x_1 \cdot 0 \oplus x_2 \cdot 0 \oplus x_3 \cdot 1 \\ y_3 = x_0 = x_0 \cdot 1 \oplus x_1 \cdot 0 \oplus x_2 \cdot 0 \oplus x_3 \cdot 0 \end{cases}$$

$$(y_0, y_1, y_2, y_3) = (x_0, x_1, x_2, x_3) \begin{pmatrix} 1_F & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

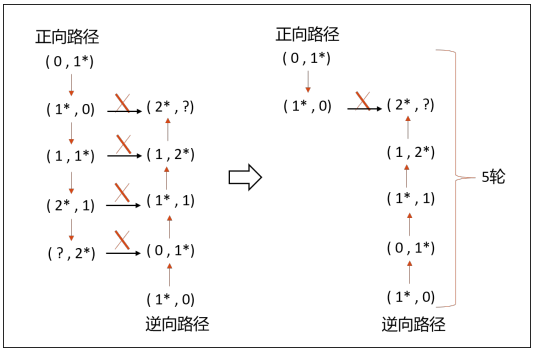
$$(x_0, x_1, x_2, x_3) = (y_0, y_1, y_2, y_3) \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1_F & 0 & 0 \end{pmatrix}$$

确定好正向和逆向的变换矩阵之后，我们就可以分别遍历 (x_0, x_1, x_2, x_3) 和 (y_0, y_1, y_2, y_3) 其中 $x_i, y_i \in K = \{0, 1, 1^*, 2^*, ?\}$ 来寻找最长的正向的概率为 1 的线性传播路径和逆向路径。

寻找集合间的矛盾：

寻找路径之后，就要找到一条正向一条逆向路径之间的矛盾然后级联成长轮数的零相关线性路径。

示意图如下：



关于集合间的矛盾，从集合的定义中就可以看出

- 0 零掩码的矛盾集合就是非零的掩码，$\overline{0} = \{1, 1^*\}$
- 1 非零且取值不确定的掩码的矛盾集合只有 0 掩码，其他掩码都有可能是非零且取值不定的，$\overline{1} = \{0\}$
- 1非零且取值确定的掩码，因此与 0 掩码矛盾，因为 $2^* \neq 1^*$ 与 2^* 也矛盾，$\overline{1^*} = \{0, , 1 \oplus 1^* = 2^*\}$
- $2^* = 1 \oplus 1^*$，因为 1 是非零的，所以 $2^* \neq 1^*$，其他的都是可能的，$\overline{2^*} = \{1^*\}$
- ? 不确定掩码和任何掩码都不矛盾

寻找矛盾路线的方法比较简单，就是对于每条正向路径的每个位置，都遍历一遍所有的逆向路径的每个位置，遍历过程中记录下路径长度，大于等于当前路径长度时再去检验矛盾，如果是矛盾的，就说明当前路径是目前最长的零相关路径，然后记录下来。

2.3 代码实现

寻找概率为 1 的最长正向和逆向路径：

```
def task2():
    E_matrix = [[0xf,  0,  0,  1],
                [  1,  0,  0,  0],
                [  0,  1,  0,  0],
                [  0,  0,  1,  0]]
    D_matrix = [[  0,  1,  0,  0],
                [  0,  0,  1,  0],
                [  0,  0,  0,  1],
                [  1,0xf,  0,  0]]
    linear_set = [0,1,2,3,4] # 对应 0,1,1*,2*,?
    forward_path = []
    back_path = []
    for x0 in linear_set:  # 寻找正向路径
        for x1 in linear_set:
            for x2 in linear_set:
                for x3 in linear_set:
                    tmp = [(x0,x1,x2,x3)]
                    if tmp[0].count(0)+tmp[0].count(4) == 4 :
                        continue
```

```
                while 1:
                    out = dot(tmp[-1],E_matrix)          # 正向路径，用的E矩阵
                    if out.count(0)+out.count(4) == 4:  # 结束条件 如果四个值只有0和?，比如
(0,?,0,?)
                        if len(tmp)>1:
                            forward_path.append(tmp)
                        break
                    tmp.append(out)
for y0 in linear_set: # 寻找反向路径
    for y1 in linear_set:
        for y2 in linear_set:
            for y3 in linear_set:
                tmp = [(y0,y1,y2,y3)]
                if tmp[0].count(0)+tmp[0].count(4) == 4 :
                    continue
                while 1:
                    out = dot(tmp[-1],D_matrix)   # 反向路径，用的D矩阵
                    if out.count(0)+out.count(4) == 4: # 结束条件 如果四个值只有0和?，比如
(?,?,0,?)
                        if  len(tmp)>1:
                            back_path.append(tmp)
                        break
                    tmp.append(out)
get_Longest_path(forward_path,back_path) # 寻找最长矛盾路径
```

寻找最长的矛盾路径：

```
def get_Longest_path(forward_path,back_path):
    '''遍历，找出所有的矛盾级联'''
    Longest_path = []
    longest = 0
    for f_path in forward_path:   # 遍历正向路径
        tmp_f = []
        for f in range(len(f_path)):# 在正向路径的f位置，遍历逆向路径的所有位置
            tmp_f.append(trans(f_path[f]))
            for b_path in back_path:
                tmp_b = []
                '''遍历逆向路径主要就是找最长轮数和判断冲突'''
                for b in range(len(b_path)):
                    tmp_b.append(trans(b_path[b]))
                    long = f+b
                    if long >= longest:
                        '''如果当前长度超过最长的，再判断是否矛盾'''
                        if Conflict(f_path[f],b_path[b]):
                            if long > longest:
                                longest = long
                                '''如果发现了更长的矛盾路径，那么就重置最长路径'''
                                Longest_path.clear()
                            if [tmp_f,tmp_b] not in Longest_path:
                                '''一样长的矛盾路径，记录下来'''
                                Longest_path.append([tmp_f[:],tmp_b[:]])
                            continue
    print("最长的轮数为{},零相关线性路径共有{}条".format(longest,len(Longest_path)))
    for i in range(len(Longest_path)):
```

```
        print("\n第{}条:".format(i+1))
        print("正向路径:{}".format(Longest_path[i][0]).replace("'",'').replace("), ",")->").replace
    (", ",",","))
        print("逆向路径:{}".format(Longest_path[i][1]).replace("'",'').replace("), ",")->").replace
    (", ",",","))
```

运行结果:

```
----------------------------------------Task2----------------------------------------
最长的轮数为19,零相关线性路径共有8条

第1条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)->(1*,?,?,1)->(1,2*,?,?)
    ->(?,?,2*,?)->(?,?,?,2*)->(2*,?,?,?)]

第2条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)->(1*,?,?,1)->(1,2*,?,?)
    ->(?,?,2*,?)->(?,?,?,2*)]

第3条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)->(1*,?,?,1)->(1,2*,?,?)
    ->(?,?,2*,?)]

第4条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)->(1,1*,1,1)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)->(1*,?,?,1)->(1,2*,?,?)]

第5条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)->(1,1*,1,1)
    ->(2*,1,1,1)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)->(1*,?,?,1)]

第6条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)->(1,1*,1,1)
    ->(2*,1,1,1)->(?,1,1,2*)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)->(?,?,1,1*)]

第7条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)->(1,1*,1,1)
    ->(2*,1,1,1)->(?,1,1,2*)->(?,1,2*,?)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)->(1,1,1*,?)]

第8条:
正向路径:[(0,0,0,1*)->(0,0,1*,0)->(0,1*,0,0)->(1*,0,0,0)->(1,0,0,1*)->(1,0,1*,1)->(1,1*,1,1)
```

```
    ->(2*,1,1,1)->(?,1,1,2*)->(?,1,2*,?)->(?,2*,?,?)]
逆向路径:[(1*,0,0,0)->(0,1*,0,0)->(0,0,1*,0)->(0,0,0,1*)->(1*,1,0,0)->(0,1*,1,0)->(0,0,1*,1)
    ->(1,1,0,1*)->(1*,?,1,0)->(0,1*,?,1)]
task2 costs 9.24612021446228s
```