实验四：SM4 的密钥恢复攻击

组员：刘舒畅，李昕，林宗茂

2023 年 11 月 5 日

成员信息及完成部分：

刘舒畅：202122460175 负责不可能差分分析

李昕：202100460065 负责差分分析

林宗茂：202100460128 负责线性分析

摘　要

在本次实验中，我们对 SM4 分别进行了差分分析，不可能差分分析和线性分析。

在差分分析中，我们参考 [2] 的思路，利用 SM4 的拉线结构，找到了一条五轮差分路线，并在此基础上构造了 18 轮差分路线，描述了一个复杂度 $2^{47.5}$ 的 5 轮密钥恢复攻击和一个复杂度 $2^{126.6}$ 的 21 轮密钥恢复攻击。

在不可能差分中，我们首先利用 u 方法找到两条 6 轮不可能差分，然后利用差分分析中找到的 5 轮路线，构造 7 轮和 12 轮路线，利用 12 轮不可能差分对 16 轮 SM4 的进行复杂度 2^{107} 的密钥恢复攻击，

在线性分析中，我们首先构造短轮数可迭代的线性路线，将其进行多次迭代得到长轮数的线性路线，找到 SM4 的 20 轮线性近似路线并进行 24 轮线性攻击，复杂度约为 2^{120} 次加密操作。

关键词：SM4　差分分析　不可能差分　线性分析

目录

1 **问题重述** 1

2 **实验准备** 1

2.1 针对 SM4 的差分分析 1

2.1.1 针对 SM4 的 5-轮差分分析 1

2.1.2 5 轮密钥恢复攻击 3

2.1.3 针对 SM4 的 $5n + m$ 轮差分分析和 21 轮差分攻击 3

2.2 针对 SM4 的不可能差分分析 5

2.2.1 6 轮不可能差分 5

2.2.2 7 轮不可能差分 6

2.2.3 12 轮不可能差分 6

2.2.4 对 16 轮 SM4 的密钥恢复攻击：利用 12 轮不可能差分 7

2.3 针对 SM4 的线性分析 8

2.3.1 SM4 的一轮可迭代线性近似路线 8

2.3.2 SM4 的两轮可迭代线性近似路线 10

2.3.3 SM4 的三轮可迭代线性近似路线 11

2.3.4 对 SM4 的 20 轮线性近似路线与 24 轮线性攻击 12

2.3.5 24 轮线性攻击复杂度分析 13

3 **实验过程** 14

3.1 搜索 SM4 的 5-轮差分路线 14

3.2 利用 U 方法搜索 SM4 的不可能差分路线 17

3.3 寻找 SM4 的线性路线 19

A **附录** 24

A.1 寻找线性近似式 24

A.2 寻找差分近似式 29

A.3 u 方法寻找不可能差 34

1 问题重述

问题一 对 SM4 进行密钥恢复攻击（攻击方法和攻击轮数不限，要有详细分析和推导过程，不要照搬网上他人结果，如果基于他人工作请务必标明引用），分别给出一个复杂度在 2^{30} 的攻击和一个复杂度较高但优于穷举攻击的分析。

2 实验准备

2.1 针对 SM4 的差分分析

2.1.1 针对 SM4 的 5-轮差分分析

本部分思路主要来自 [2] 中对 SM4 广义 Feistel 结构特点的讨论，观察 SM4 的拉线结果，可以看到其每次进入 T 函数前，会进行三线异或（本部分差分不涉及轮密钥异或，故默认不讨论），如下图所示：

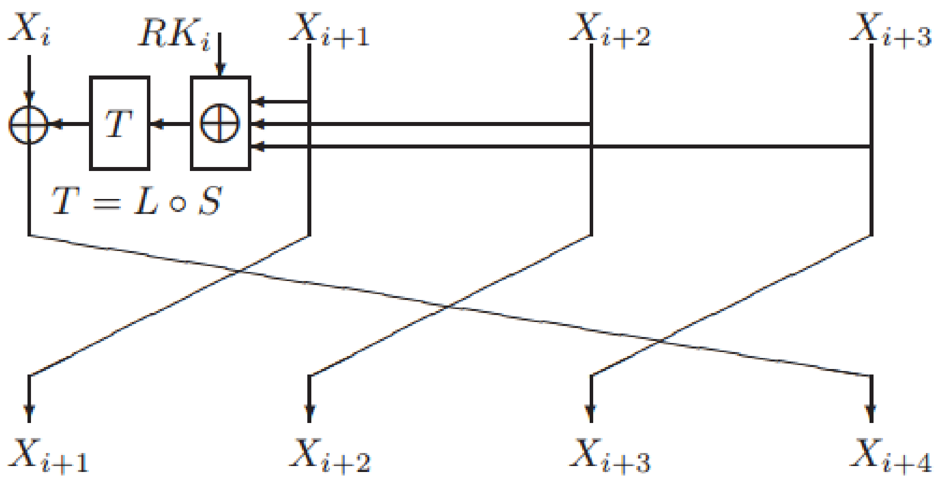

Figure 1: $X_{i+1}, X_{i+2}, X_{i+3}$ 异或

可以看到当 $X_{i+1}, X_{i+2}, X_{i+3}$ 三者中两者相等时，会使得第三者”独自”进入 S 盒，那么为了降低在多轮差分路线中活跃 S 盒的数量，如果能够找到 $\{X_i, X_{i+1}, X_{i+2}, X_{i+3}\}$ 的构造方法, 使得在多轮差分中，使得某些轮 S 盒的输入差分为 0，就可以以 1 的概率传播，提高整体路线的概率。

参考 [2] 中思路，定义 32 比特的非零差分 $\alpha \in Z_2^{32} \setminus \{0\}$，考虑如下构造方法：

第一轮：选择构造第一轮输入差分为 $(\alpha, \alpha, \alpha, \alpha)$, 则由上述讨论，T 函数的输入差

分为 α, 将输出差值为 α 的概率记为 $Prob_T(\alpha \to \alpha)$，易得到输出差为 $(\alpha, \alpha, \alpha, 0)$，概率为 $Prob_T(\alpha \to \alpha)$。

第二轮：第一轮的输出差分进入第二轮，为 $(\alpha, \alpha, \alpha, 0)$, 由于三线异或进入 T 函数，故可得到 T 函数的输入差分为 0, 易得到输出差为 $(\alpha, \alpha, 0, \alpha)$，概率为 1。

第三轮：第二轮的输出差分进入第三轮，为 $(\alpha, \alpha, 0\ \alpha)$, 和第二轮相似的，T 函数的输入差分为 0, 得到输出差为 $(\alpha, 0, \alpha, \alpha)$，概率为 1。

第四轮：第三轮的输出差分进入第四轮，为 $(\alpha, 0, \alpha, \alpha)$, 依然相似的，T 函数的输入差分依然为 0, 得到输出差为 $(0, \alpha, \alpha, \alpha)$，概率为 1。

第五轮：最后一轮与第一轮类似，输入差分为 $(0, \alpha, \alpha, \alpha)$, 三线异或得到 α，故可得到 T 函数的输入差分为 α, 易得到输出差为 $(\alpha, \alpha, \alpha, \alpha)$，概率为 $Prob_T(\alpha \to \alpha)$。

可以发现这是一个五轮迭代差分路线,$(\alpha, \alpha, \alpha, \alpha) \to (\alpha, \alpha, \alpha, \alpha)$,概率为 $(Prob_T(\alpha \to \alpha))^2$，为了使 $Prob_T(\alpha \to \alpha)$ 足够高，我们需要选择合适的 α，我们下面引用 [2] 中的一个定义和一条定理：

定义 1. (分支数) 定义 $W(\cdot)$ 表示字节权重函数，即非零字节数, 线性变换 $L: Z_2^{32} \to Z_2^{32}$ 的分支数为:

$$\min_{a \neq 0, a \in Z_2^{32}} (W(a) + W(L(a)))$$

定理 1. 在 SM4 的轮函数中，线性变换 L 分支数为 5。

为了提高差分传播概率，我们需要使第一轮和最后一轮的活跃动 s 盒的数量尽可能少。考虑到在上述差分中，T 的输入和输出差均为 α，根据定理 1，L 的分支数为 5，可以知道 α 与 $L(\alpha)$ 的最小非零字节数为 5, 即 α 最小有 3 个非零字节，即可以设 α 形如 $(0, x_1, x_2, x_3)$，其中 $x_i \in Z_2^8 \setminus \{0\}$. 同时，由于 T 是 $\alpha \to \alpha$ 的差分，S 盒的输出差分 $L^{-1}(\alpha)$ 也必须形如 $(0, y_1, y_2, y_3)$，其中 $y_i \in Z_2^8 \setminus \{0\}$。

与 α 形如 $(0, x_1, x_2, x_3)$ 同理，将 00 定位在四个位置中的任意一个都可以得到我们想要的范围，根据 SM4S 盒差分分布表显然可知，对于任何非零输入差分，只有 127 个可能的输出差分。因此，$Prob_T(\alpha \to \alpha)$ 不等于 0 的情况对于每个形式都有 $2^{16} * (1/2)^3 = 2^{13} \approx 7905$ 个, 四个形式共有 2^{15} 个。特别的，由于 SM4 使用了四个相同的 S 盒，所有事实上这四种形式得到的差分路线是等价的，只有循环移位上的区别，我们将在 Part3 中手动搜索并验证这个结论。

在确定 α 范围后，可以得到以下概率等式：

$$Prob_T(\alpha \to \alpha) = \sum_{i=1}^{3} Prob_{Sbox}(x_i \to y_i)$$

$$Prob_{5round}\{(\alpha,\alpha,\alpha,\alpha)\rightarrow(\alpha,\alpha,\alpha,\alpha)\}=(\sum_{i=1}^{3}Prob_{Sbox}(x_i\rightarrow y_i))^2$$

我们将在 Part3 利用代码搜索所有可能的该结构的差分路线，在此处我们根据 SM4S 盒的差分分布表给出该路线的理论概率上界和下界，即 2^{-42} 到 2^{-36}，但是按照我们搜索的结果，该结构差分路线实际最优概率仅为 2^{-38}。

2.1.2 5 轮密钥恢复攻击

我们首先尝试给出一个六轮的短轮数密钥恢复攻击，我们在找到的五轮区分器后面增加一轮，针对第六轮涉及的 32bit 密钥信息进行恢复，但经过分析，由于我们找到的五轮区分器概率太低，正确密钥计数器的期望值为 $2^{-38}m$，甚至低于错误密钥的期望值 $2^{-32}m$，故该攻击无效。

于是我们尝试基于五轮中的后四轮，给出一个 5 轮的短轮数密钥恢复攻击，我们截取五轮区分器的后四轮，并在后面增加一轮，针对新的第五轮涉及的 32bit 密钥信息进行恢复，对于每个候选密钥，设置对应的 2^{32} 个计数器，初始化为 0.

选择 m 个互不相同的明文,将明文与最优差分异或得到另一半明文 (即与 $(\alpha,\alpha,\alpha,0)$ 异或, $\alpha=002cf5cd$), 经过此操作得到 m 对输入差分为 002cf5cd 的明文对.

用每一个猜测的轮密钥 RK5 解密第 5 轮，$X_5=X_9\oplus T(X_8\oplus X_7\oplus X_8\oplus RK5)$，判断是否与区分器的尾差分吻合，若相等则对应计数器 +1，处理完所有密钥后，正确密钥计数器的期望值为 $2^{-19}m$，概率远大于错误密钥的期望值 $2^{-32}m$，从而恢复出正确密钥。

复杂度分析：首先我们计算该攻击的信噪比 $S/N=\frac{2^{k}*p}{\alpha*\beta}$, 其中 k 是猜测的密钥比特数 32bit，p 是差分特征的概率 2^{-19}，α 是恢复密钥阶段每个方程平均求得的解数 1，β 是过滤比例 1，计算得到信噪比为 2^{13}. 根据 Biham 和 Shamir 在 [3] 中的建议，当 $S/N=2$ 时，成功进行差分攻击大约需要 20 到 40 对正确的明密文对，而当信噪比较大 ($S/N>=1000$) 时，只需要 3、4 对正确对即可，因此我们选择 $m=2^{20}$，使得正确对个数为 $2^{-19}*2^{20}=2$, 数据复杂度为 2^{20}。对于时间复杂度，共需要 $2*2^{32}*2^{20}/5\approx 2^{47.5}$ 次单轮加密。

2.1.3 针对 SM4 的 $5n+m$ 轮差分分析和 21 轮差分攻击

利用找到的 5 轮可迭代差分路线，可以很容易的构造 $5n+m$ 轮的差分路线，其中 $n\ m\in Z_+$ 且 $0<m<5$. 唯一需要注意的是迭代差分路线的概率需要低于随机置换

2^{-128}，根据这个概率下界，我们可以构造出最长 5*3+3=18 轮的差分路线，即 3 个完整的五轮迭代差分路线加上第 2 到 4 轮概率为 1 的差分路线，其差分概率平均为 2^{-126}。

为了实现 21 轮密钥恢复攻击，我们需要稍稍调整五轮差分路线的起点，得到以下 18 轮差分路线：

$$(\alpha,\alpha,\alpha,0) \xrightarrow{5-round} (0,\alpha,\alpha,\alpha) \xrightarrow{5-round} (0,\alpha,\alpha,\alpha) \xrightarrow{5-round} (\alpha,\alpha,\alpha,0) \xrightarrow{3-round} (0,\alpha,\alpha,\alpha)$$

我们将 18 轮差设为 0 到 17 轮，并选择明文对的差分为 $(\alpha,\alpha,\alpha,0) \in Diff$，那么第 17 轮的正确对的输出差值为 $(0,\alpha,\alpha,\alpha)$，接下来三轮的输出差值假定为 $(\alpha,\alpha,\alpha,*)$，$(\alpha,\alpha,*,*)$ 和 $(\alpha,*,*,*)$，其中 $*$ 表示未知，下面利用这条 18 轮差分路线构造 21 轮攻击：

1. 选择 m 个结构，每个结构 2^{72} 个明文，其中每个结构中 56 位字节 0、4、8、12、13、14、15 是固定的，其他 72 位取所有可能的值。然后每个结构产生大约 $(2^{72})^2/2 = 2^{143}$ 对不同的明文对, 差分为 ((0,*,*,*)，(0,*,*,*)，(0,*,*,*)，(0,0,0,0))，m 个结构总共可以产生大约 m $\cdot$ 2^{143} 对明文对。

2. 对于每个明文对,检查明文对的差分是否属于 Diff 集合 (Diff = ((0,u,v,w),(0,u,v,w),(0,u,v,w),(0, 0, 0, 0)).),如果不属于,就丢弃这对,测试后,预计将保留约 $m \cdot 2^{143} \cdot (2^{13}/2^{72}) = m \cdot 2^{84}$ 对明文。

3. 对于剩下的每一对 (P_i, P_j),计算明文差分并表示为 ((0,u,v,w),(0,u,v,w),(0,u,v,w),(0, 0, 0, 0))。然后计算对应的密文对 (C_i, C_j) 的差分,检查密文差值的第一个字是否等于 (0,u,v,w)。如果不是,就丢弃这对。经过这个测试,剩下 $m \cdot 2^{84} \cdot 2^{-32} = m \cdot 2^{52}$ 对。

4. 对于第 20 轮中 32 位子密钥 RK20 的每次猜测，执行以下操作：

(1) 对筛选剩下的明密文对, 解密第 20 轮的一部分, 即 $X_{20} = X_{24} \oplus T(X_{21} \oplus X_{22} \oplus X_{23} \oplus RK20)$, 检查第 19 轮输出差分的第一个字是否等于 (0,u,v,w)，丢弃掉不相等的对，大概可以保留 $m \cdot 2^{52} \cdot 2^{-32} = m \cdot 2^{20}$ 对明密文。

(2) 对于每个 32 位子密钥 RK19 的猜测，解密第 19 轮的一部分:$X_{19} = X_{23} \oplus T(X_{20} \oplus X_{21} \oplus X_{22} \oplus RK19)$。检查第 18 轮输出差分的第一个字是否等于 (0,u,v,w)，如果不是，则丢弃该对。经过这个测试，对于 RK20 和 RK19 的每一次猜测，仍然有大约 $m \cdot 2^{20} \cdot 2^{-32} = m \cdot 2^{-12}$ 对明密文.

(3) 尝试子密钥 RK18 的所有 2^{32} 个可能值，并解密第 18 轮的一部分:$X_{18} = X_{22} \oplus T(X_{19} \oplus X_{20} \oplus X_{21} \oplus RK18)$。检查第 17 轮输出差的第一个字是否等于 0。如果不是这样，就丢弃这对。经过这次检验，RK20、RK19 和 RK18 的每一次猜测，都保留约 $m \cdot 2^{-12} \cdot 2^{-32} = m \cdot 2^{-44}$ 对。

5. 在 step4 后每组 RK20, RK19 和 RK18 都会得到一个计数器值，值为经过 (1)(2)(3) 后剩余的明密文对，计数器值最大的一组我们可以认为是正确密钥。原因是由于 18 轮差分路线的概率为 2^{-126}，对于正确的 21，20，19 轮密钥，预计仍有约 $m \cdot 2^{84} \cdot 2^{-126} = m \cdot 2^{-42}$ 对明密文保留, 大于错误的子密钥猜测下，在步骤 4-(3）之后的剩余对的期望数量 $(m \cdot 2^{-12} \cdot 2^{-32} = m \cdot 2^{-44})$.

21 **轮差分攻击的复杂度分析**：同样，首先我们计算该攻击的信噪比 $S/N = \frac{2^k * p}{\alpha * \beta}$，其中 k 是猜测的密钥比特数 96bit，p 是差分特征的概率 2^{-114}，α 是明密文对可以恢复的密钥的平均比特数 1，β 是 step3 中的过滤比例 2^{-32}。计算得到信噪比约为 4，根据 Biham 和 Shamir 在 [3] 中的建议，当 $S/N = 2$ 时，成功进行差分攻击大约需要 20 到 40 对正确的明密文对，而当 $S/N >= 2$ 时，需要更少的对数，因此，在此攻击中，我们可以选择 $m = 2^{46}$，期望的正确配对数约为 $2^{46} \cdot 2^{-42} = 16$，因此，攻击总共需要 $2^{46} \cdot 2^{72} = 2^{118}$ 对选择的明文，数据复杂度为 2^{119}。对于时间复杂度，其需要的加密次数为 $2 * 2^{52} * 2^{46} * 2^{32}/21 \approx 2^{126.6}$ 次 21 轮加密.

2.2 针对 SM4 的不可能差分分析

目前较好的攻击方法是 16 轮的不可能差分攻击。本部分从不同方式出发，给出了不同轮数的不可能差分。

2.2.1 6 轮不可能差分

通过 u 方法, 我们可以找到两条 6 轮的不可能差分, 分别是 $(0, \alpha, 0, 0) \to (0, 0, \alpha, 0)$ 和 $(\alpha, 0, 0, 0) \to (0, 0, 0, \alpha)$。实验三已经对其原理进行了分析，故具体分析过程此处不再赘述，我们将在 part3 中给出分析代码。需要注意的是，u 方法作为通用分析方法有其局限性：在处理多个已知明文异或的情况下，无法有效的扩展分析（该项会被归约到 t)，因此，我们手动尝试推导了其他不可能差分。

2.2.2 7 轮不可能差分

本部分的主要结构基础为前一部分差分分析中提出的 $(\alpha,\alpha,\alpha,0) \xrightarrow{3-round} (0,\alpha,\alpha,\alpha)$ 这一概率为 1 的差分路线。显然有一条 6 轮的不可能差分路线 $(\alpha,\alpha,\alpha,0) \xrightarrow{6-round} (0,\alpha,\alpha,\alpha)$，具体分析如下：

$Round(i)\downarrow$	Δ_0	Δ_1	Δ_2	Δ_3	$Round(i)\uparrow$	Δ_0	Δ_1	Δ_2	Δ_3
0	α	α	α	0	3	α	α	α	0
1	α	α	0	α	4	α	α	0	α
2	α	0	α	α	5	α	0	α	α
output	0	α	α	α	output	0	α	α	α

此时，我们可以通过对第三轮的拓展得到更长轮数的不可能差分。具体而言，我们将差分为 α 的两中间值经过 T 变换后得到的值得异或记作 $T'(\alpha)$，记 $x = T'(\alpha)$，则有分析如下：

$Round(i)\downarrow$	Δ_0	Δ_1	Δ_2	Δ_3	$Round(i)\uparrow$	Δ_0	Δ_1	Δ_2	Δ_3
0	α	α	α	0	4	α	α	α	0
1	α	α	0	α	5	α	α	0	α
2	α	0	α	α	6	α	0	α	α
3	0	α	α	α					
output	α	α	α	x	output	0	α	α	α

由于 $x \neq 0$，因此出现冲突，于是我们找到了一个 7 轮的不可能差分路线。

2.2.3 12 轮不可能差分

根据论文 [4] 所述，上文提到的 6 轮的差分可以从开头和结尾各延长 3 轮，从而扩展出一个 12 轮的不可能差分。记 $y = T'(x)$，$z = T'(x \oplus y \oplus \alpha)$，则有

$Round(i)\downarrow$	Δ_0	Δ_1	Δ_2	Δ_3	$Round(i)\uparrow$	Δ_0	Δ_1	Δ_2	Δ_3
0	α	α	α	0	4	z	y	x	α
1	α	α	0	α	5	y	x	α	α
2	α	0	α	α	6	x	α	α	α
3	0	α	α	α	9	α	α	α	0
4	α	α	α	x	10	α	α	0	α
5	α	α	x	y	11	α	0	α	α
output	α	x	y	z	output	0	α	α	α

论文在 α 确定的情况下，对 x、y、z 所处的集合做了遍历，并声明“不存在这样的一个六元组 $(x_1, y_1, z_1, x_2, y_2, z_2)$，使得两轮数据完全相同”。其中，$z_1 = \alpha$ 可以等价表示为 $x_1 \oplus y_1 \oplus \alpha = 0$。此处由于存储功能限制，没有对查找过程进行复现（y 所在的集合已有约 2^{48} 个元素，远超家用计算机所能调度的内存极限）。此外，需要注意的是，此处出于降低复杂度的考量，α 的取值范围为 $\{0, 1, ..., 2^{16} - 1\}$。

2.2.4 对 16 轮 SM4 的密钥恢复攻击：利用 12 轮不可能差分

根据论文所言，我们可以利用 12 轮的不可能差分进行 16 轮的密钥恢复攻击。具体而言，我们将不可能差分前后各扩展两轮。以向前扩展为例，扩展一轮时，由于 α 仅有两个活跃 S 盒，因此可能的一轮扩展差分种类数为 127^2（此处 127 是因为非零差分过 S 盒不可能出现 0 差分）[5]；再通过 L 变换后，新的差分会扩展到所有位置，因此可能的二轮扩展差分种类数为 127^6。类似的，可能的二轮向后扩展差分种类数也为 127^6。记初始轮为第 0 轮，不可能差分置于 2 13 轮，最终轮为 15 轮；可能的输入差分集合记为 Σ_1，输出差分集合记为 Σ_2。

1. 选择 2^9 个有 2^{96} 条明密文对的结构体，为保持 $(?, ?, \alpha, \alpha)$ 的结构，128bits 明文右的两个字的前 16bits 为一定值，其余 96bits 取遍所有可能。共有 $2^9(2^{96}/2)^2 = 2^{199}$ 个符合差分的明密文对。再用 $P_i \oplus P_j = \Sigma_1$ 与 $C_i \oplus C_j = \Sigma_2$ 筛选出可能符合不可能差分头尾的明密文对。

2. 对于剩余的密文对 (C_i, C_j)，提前计算对应的不可能尾部差分所能生成的在 L 变换之前的 15 轮中间值，记作 $(\Delta^{15}_{i,j,0}, \Delta^{15}_{i,j,1}, \Delta^{15}_{i,j,2}, \Delta^{15}_{i,j,3})$。之后，分开遍历 8bits 密钥以解密 4 个字节至 L 变换之前，记作 $T_{i,l}, T_{j,l}$，仅保留对于 $l = 0...3$，均有 $T_{i,l} \oplus T_{j,l} = \Delta^{15}_{i,j,l}$ 的 (C_i, C_j)。

之后再往上一轮，计算对应的不可能尾部差分所能生成的在 L 变换之前的 15 轮后 32bits 中间值，$(\Delta^{14}_{i,j,2}, \Delta^{14}_{i,j,3})$，之后，分开遍历 8bits 密钥以解密后 2 字节至 L 变换之前，记作 $R_{i,l}, R_{j,l}$，仅保留对于 $l = 2, 3$，均有 $R_{i,l} \oplus R_{j,l} = \Delta^{14}_{i,j,l}$ 的 (C_i, C_j)。

3. 对剩余对的明文，我们也做类似的筛选：如果最后存在一组中间值 $S_{i,l} \oplus S_{j,l} = \Delta^{1}_{i,j,l}$，则产生了一对不可能差分，证明我们在这一组明密文对中所使用的 96bits 轮密钥组是错误的，丢弃；反之则记录下来。遍历完所有对。

4. 对于我们筛选出的 96bits 轮密钥组，至多有 2^96 个 128bits 的主密钥。之后我们用已知明密文对进行筛选：如果有符合的，则输出 128bits 的主密钥；反之，回到 Step 2.。

正确性分析： 通过 Σ 的降噪提供了 $\frac{127^6}{2^{64}} \times \frac{127^6}{2^{128}} \approx 2^{-108.12}$ 的过滤。对于任意错误对，每一字节的测试成功率均为 $frac1127$。因而通过倒数第二级测试的对数约为 $2^{13.99}$，在进行最后一轮测试之后，只有 $2^{96} \times (1-\frac{1}{127})^{2^{13.99}} \approx 2^{-88.32}$ 个可能的密钥组。此外，期望通过第四轮测试的错误密钥有 $2^{-128} \times 2^{96} = 2^{-32}$，因此可以判定输出即为正确密钥。

复杂度分析： 攻击总计需要 2^{105} 量级的选择明文。2-4 的时间开销为 $\Sigma^{11}_{l=1}(2 \times^{90.88} \times 2^{8l} \times \frac{1}{127^l - 1} \times \frac{1}{16}) + 2 \times 2^{96} \times [1 + (1 - \frac{1}{127} + ... + (1 - \frac{1}{127})^{2^{13.99}}] \times \frac{1}{16} \approx 2^{107}$ 次 16 轮加密。

2.3 针对 SM4 的线性分析

要对 SM4 算法进行线性攻击，我们首先想到的方法是首先构造短轮数可迭代的线性路线，将其进行多次迭代得到长轮数的线性路线。首先寻找是否存在 SM4 的可迭代一轮线性近似路线。下面是基本操作的一些性质。

性质 1 异或操作 对于函数 $f(x_0, x_1) = x_0 \oplus x_1$，其输入掩码分别为 $\Gamma = (\Gamma_0, \Gamma_1)$ 和 Λ，那么 $Pr(\Lambda \cdot f(x_0, x_1) = \Gamma_0 \cdot x_0 \oplus \Gamma_1 \cdot x_1) \neq \frac{1}{2}$ 当且仅当 $\Gamma_0 = \Gamma_1 = \Lambda$

性质 2 三分支操作 函数 $f(x) = (x, x)$，输入和输出掩码向量分别为 Γ 和 $\Lambda = (\Lambda_0, \Lambda_1)$，那么 $Pr(\Gamma \cdot x = \Lambda \cdot f(x)) \neq \frac{1}{2}$ 当且仅当 $\Gamma = \Lambda_0 \oplus \Lambda_1$

性质 3 线性映射

2.3.1 SM4 的一轮可迭代线性近似路线

SM4 的轮函数结构图如下

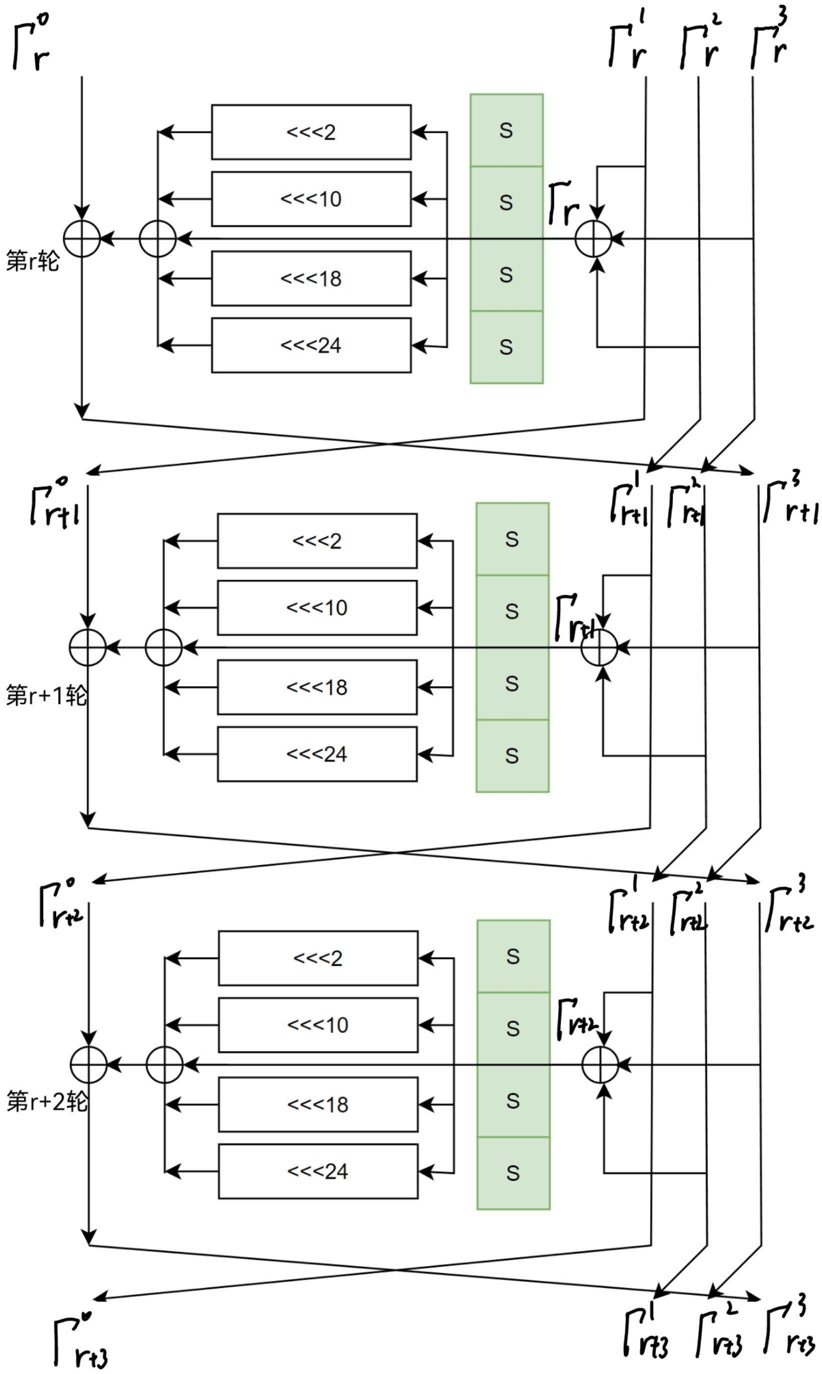

Figure 2: SM4

这里将一轮中的 4 个 S 盒称为 1 组 S 盒，第 r 轮的一组 S 盒的线性掩码为 Γ_r。如果存在一轮可迭代线性近似，则以下关系成立：

$$\Gamma_r^0 = \Gamma_{r+1}^0$$

$$\Gamma_r^1 = \Gamma_{r+1}^1$$

$$\Gamma_r^2 = \Gamma_{r+1}^2$$

$$\Gamma_r^3 = \Gamma_r^0$$

利用三分支操作的性质，有

$$\Gamma_r^1 = \Gamma_{r+1}^0 \oplus \Gamma_r$$

$$\Gamma_r^2 = \Gamma_{r+1}^1 \oplus \Gamma_r$$

$$\Gamma_r^3 = \Gamma_{r+1}^2 \oplus \Gamma_r$$

由以上等式我们可以得到

$$\Gamma_{r+1}^1 = \Gamma_r^0 \oplus \Gamma_r$$

$$\Gamma_r^0 \oplus \Gamma_r \oplus \Gamma_r = \Gamma_{r+1}^2$$

$$\Gamma_r^3 = \Gamma_r^0 \oplus \Gamma_r \oplus \Gamma_r \oplus \Gamma_r = \Gamma_r^0 \oplus \Gamma_r = \Gamma_r^0$$

这些式子表明 $\Gamma_r = 0$，即 S 盒线性掩码全为 0，这使得该轮没有活跃 S 盒，因此 SM4 不存在一轮可迭代线性近似。

2.3.2 SM4 **的两轮可迭代线性近似路线**

如上图所示，若存在两轮可迭代线性近似，则有

$$\Gamma_r^0 = \Gamma_{r+2}^0$$

$$\Gamma_r^1 = \Gamma_{r+2}^1$$

$$\Gamma_r^2 = \Gamma_{r+2}^2$$

$$\Gamma_r^3 = \Gamma_{r+1}^0$$

由三分支操作性质，有

$$\Gamma_r^1 = \Gamma_{r+1}^0 \oplus \Gamma_r$$

$$\Gamma_r^2 = \Gamma_{r+2}^0 \oplus \Gamma_r \oplus \Gamma_{r+1}$$

$$\Gamma_r^3 = \Gamma_{r+2}^1 \oplus \Gamma_r \oplus \Gamma_{r+1}$$

$$\Gamma_{r+2}^2 = \Gamma_r^0 \oplus \Gamma_{r+1}$$

由以上八个式子可以推出

$$\Gamma_r^2 = \Gamma_r^0 \oplus \Gamma_r \oplus \Gamma_{r+1}$$

$$\Gamma_r^3 = \Gamma_r^1 \oplus \Gamma_r \oplus \Gamma_{r+1}$$

$$\Gamma_r^2 = \Gamma_r^0 \oplus \Gamma_{r+1}$$

所以有 $\Gamma_{r+1} = 0$，同理，没有活跃 S 盒，因此也不存在两轮可迭代线性近似。

2.3.3 SM4 的三轮可迭代线性近似路线

由于 $\Gamma_r^0=\Gamma_{r+3}^0$，结合性质 1 与 2 有

$$\Gamma_r^0=\Gamma_r^3\oplus\Gamma_r\oplus\Gamma_{r+1}\oplus\Gamma_{r+2}$$

由于 $\Gamma_r^1=\Gamma_{r+3}^1$，结合性质 1 与 2 有

$$\Gamma_r^1=\Gamma_r^0\oplus\Gamma_{r+1}\oplus\Gamma_{r+2}$$

由于 $\Gamma_r^2=\Gamma_{r+3}^2$，结合性质 1 与 2 有

$$\Gamma_r^2=\Gamma_r^1\oplus\Gamma_r\oplus\Gamma_{r+2}$$

由于 $\Gamma_r^3=\Gamma_{r+3}^3$，结合性质 1 与 2 有

$$\Gamma_r^2=\Gamma_{r+2}^0\oplus\Gamma_r\oplus\Gamma_{r+1}$$

由以上等式可推导出下列式子

$$\Gamma_{r+1}=\Gamma_r\oplus\Gamma_{r+2}$$

$$\Gamma_r^0=\Gamma_r^3$$

为了使构造出的线性近似具有更大的偏差值，我们要令线性路线中的活跃 S 盒数量更少。若只有 1 个活跃 S 盒，可推导出所有 S 盒均不是活跃 S 盒，与条件矛盾，不可能只有一个活跃 S 盒。若有 2 个活跃 S 盒，则 Γ_r，Γ_{r+1} 或 Γ_{r+2} 中有一个为 0，这里假设 $\Gamma_r=0$，推算过程如下，其中 Λ_r 是 Γ_r 经过 T 函数的输出结果。

$$\because \Gamma_r=0\ \therefore \Lambda_r=0\ \Gamma_r^0=0$$

$$\therefore \Gamma_r^3=\Gamma_{r+3}^0=\Gamma_{r+3}^3=\Gamma_r^0=0$$

$$\because \Gamma_{r+2}^0=\Gamma_{r+3}^3$$

$$\therefore \Gamma_{r+2}^0=\Lambda_{r+2}=\Gamma_{r+2}=0$$

$$\therefore \Gamma_{r+1}=0$$

同理可得，任意一组 S 盒线性掩码为 0 时，在满足性质的条件下另外两组 S 盒掩码也为 0，即三组 S 盒都必须为活跃 S 盒。

使活跃 S 盒数量尽可能少，现在搜索每组 4 个 S 盒中只有一个活跃 S 盒的 3 轮可迭代线性近似路线。由上面的式子 $\Gamma_{r+1}=\Gamma_r\oplus\Gamma_{r+2}$ 可知，三组 S 盒的线性掩码相同，所以 3 组 s 盒中活跃 S 盒的位置是相同的。基于以上结论遍历查找 3 轮线性路线代码思路如下：

2.3.4 对 SM4 的 20 轮线性近似路线与 24 轮线性攻击

三轮线性近似路线搜索算法在实际运行中的复杂度 2 利用算法搜索后能得到若干条偏差为 2^{-10} 的三轮可迭代线性近似路线，利用其中任意一个三轮线性近似我们都可以构造出偏差为 2^{-58} 的 19 轮，或偏差为 2^{-61} 的 20 轮线性近似路线

以下为一条 20 轮线性近似路线，表格中 * 代表不确定的值

轮数	i	Γ_i^0	Λ_i	Γ_i	Bias	Γ_i^1	Γ_i^2	Γ_i^3
1	0	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
2	1	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
3	2	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
4	3	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
5	4	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
6	5	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
7	6	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
8	7	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
9	8	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
10	9	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
11	10	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
12	11	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
13	12	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
14	13	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
15	14	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
16	15	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
17	16	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
18	17	$8808A228$	00008200	$0000AA00$	2^{-4}	88080828	88086828	$8808A228$
19	18	$8808A228$	00008200	00006000	2^{-4}	$8808C228$	88080828	$8808A228$
20	19	$8808A228$	00008200	$0000CA00$	2^{-4}	88086828	$8808C228$	$8808A228$
21	20	$8808A228$	00008200	$*$	$*$	88080828	88086828	$8808A228$

通过在该线性路线的头尾各添加两轮，分别定义为第 0 23 轮。我们将第 2 轮以及第 22 轮的输入掩码表示为 $\Gamma_2^0 = \Gamma_2^3 = \Gamma_{22}^0 = \Gamma_{22}^3 = 0x8808A228, \Gamma_2^1 = 0x8808C228, \Gamma_2^2 = \Gamma_{22}^1 = 0x88080828, \Gamma_{22}^2 = 0x88086828, \Lambda_1 = \Lambda_{22} = 0x00008200$ ，将 S 盒输出表示为 $\Lambda_{1,2} = \Lambda_{22,2} = 0x82$

从这条线性路线中我们有

$$\Gamma_2^0 \cdot X_2^0 \oplus \Gamma_2^1 \cdot X_2^1 \oplus \Gamma_2^2 \cdot X_2^2 \oplus \Gamma_2^3 \cdot X_2^3 \oplus \Gamma_{22}^0 \cdot X_{22}^0 \oplus \Gamma_{22}^1 \cdot X_{22}^1 \oplus \Gamma_{22}^2 \cdot X_{22}^2 \oplus \Gamma_{22}^3 \cdot X_{22}^3 = \kappa$$

上式的左边又可以表示为 $\Gamma_2^0 \cdot P_2 \oplus \Gamma_2^1 \cdot P_3 \oplus \Gamma_2^2 \cdot P_0 \oplus \Gamma_2^0 \cdot P_1 \oplus \Gamma_2^0 \cdot C_2 \oplus \Gamma_2^2 \cdot C_3 \oplus \Gamma_{22}^2 \cdot C_0 \oplus \Gamma_2^0 \cdot C_1 \oplus \Gamma_2^2 \cdot XX_0 \oplus \Gamma_2^0 \cdot XX_1 \oplus \Gamma_2^0 \cdot XX_{22} \oplus \Gamma_2^2 \cdot XX_{23}$

其中 XX_r 是第 R 轮 T 变换后的结果，XS_r 是第 R 轮 S 盒之后的结果。

攻击步骤如下：

第 1 步随机选择 N 对明密文，

第 2 步准备 2^{81} 个计数器并初始化为 0

第 3 步对每一对明密文，计算 $\omega = \mathcal{O} \| \left(P_0 \oplus P_2 \oplus P_3\right)[16{-}23] \| (C_3 \oplus C_0 \oplus C_1)[16-23]$ $\| C_0 \oplus C_1 \oplus C_2 \| P_1 \oplus P_2 \oplus P_3$，然后计数器 $V_0[w]$ 加一

第 4 步 猜测 32bit 的 k_{23} ，令 2^{49} 个计数器 $V_1[0], \ldots, V_1\left[2^{49}-1\right]$ 置零

第 5 步 对每个 $0 \leq w \leq 2^{81}-1$， $\mathrm{O} \Leftarrow \mathcal{O} \oplus\ \Gamma_2^2 \cdot T\left(C_0 \oplus\ C_1 \oplus C_2 \oplus k_{23}\right), XX_{23}$ 与 x=O$\|\left(P_0 \oplus P_2 \oplus P_3\right)[16-23] \| \left(C_3 \oplus C_0 \oplus C_1\right)[16-23] \oplus XX_{23}[16-23] \| P_1 \oplus P_2 \oplus P_3$, $V_1[x] += V_0[w]$

第 6 步 猜测 8-bit $\mathrm{k}_{22}[16-23]$. 2^{41} $V_2[0], \ldots, V_2\left[2^{41}-1\right]$.

第 7 步 对每个 $0 \leq x \leq 2^{49}-1$, 计算 $\mathcal{O} \Leftarrow \mathcal{O} \oplus \Lambda_{1,2} \cdot S\left(\left(C_3 \oplus C_0 \oplus C_1\right)[16-23] \oplus\right.$ $XX_{23}[16-23] \oplus k_{22}[16-23]$ 与 $y = \mathcal{O} \| \left(P_0 \oplus P_2 \oplus P_3\right)[16-23] \| P_1 \oplus P_2 \oplus P_3$ ，最后 \$ $\mathrm{V}_2[y] += V_1[x]$。

第 8 步 猜测 32bit k_0 。令 2^9 个计数器 $V_3[0], \ldots, V_3\left[2^9-1\right]$ 置零。

第 9 步 对每个 $0 \leq y \leq 2^{41}-1$, 计算 $\mathcal{O} \Leftarrow O \oplus \Gamma_2^2 \cdot T\left(P_1 \oplus P_2 \oplus P_3 \oplus k_0\right)$，$XX_0$ 以及 $z = \mathcal{O} \| \left(P_0 \oplus P_2 \oplus P_3\right)[16-23] \oplus XX_0[16-23]$。最后 $V_3[z] += V_2[y]$。

第 10 步 猜测 8-bit $k_1[16-23]$ 。初始化 2^{80} 个计数器 $V_{\text{key}}[0], \ldots, V_{\text{key}}\left[2^{80}-1\right]$ 为 0.

第 11 步 对每个 $0 \leq z \leq 2^9-1$, 计算 $\mathcal{O} \Leftarrow \mathcal{O} \oplus \Lambda_{1,2}$ ，$S\left(\left(P_0 \oplus P_2 \oplus P_3\right)[16-23] \oplus\right.$ $XX_0[16{-}23] \oplus k_1[16{-}23]$ 。如果 $\mathcal{O} = 0$, 则令计数器 $V_{\text{key}}\left[k_0 \| k_1[16-23] \| k_{22}[16-23] \| k_{23}\right]$ 增加 $V_3[z]$; 否则减少 $V_3[z]$。

第 12 步 我们保留 V_{key} 中绝对值最大的 2^{33} 个值，对每个已经得到的子密钥, 我们猜测剩下的 88 位 $k_1[0-15,24-31] \| k_2 \| k_3$ ，再利用密钥生成算法还原出主密钥。

2.3.5 24 轮线性攻击复杂度分析

时间复杂度 在第三步中，时间复杂度约为 $2^{126.6}$ ，约为 $2^{126.6}/24 = 2^{120}$ 次 24 轮加

密操作，第 5 步与第 9 步需要 2^{113} 次一轮加密与解密，第 7 步与第 11 步需要 2^{89} 次一轮加密与解密，第 12 步需要 2^{121} 次 24 轮加密，总时间复杂度约为 $2^{122.6}$ 次加密

空间复杂度 第 2 步中所需空间为 $2^{81} \times 8 = 2^{84}$ 字节，V_{key} 需要 $2^{80} \times 16 = 2^{84}$ 字节，共需 2^{85} 字节

3 实验过程

3.1 搜索 SM4 的 5-轮差分路线

为了搜索差分路径，我们首先要完成 SM4 的基本构造 S 盒和 L 函数，对于 S 盒，建立查找表即可，对于 L 函数，构造移位异或，代码如下：

```
def s_(x):#s盒 8输入8输出
    row = x >> 4
    nuw = x & 0xf
    return sbox[row][nuw]
def xun(X,i):#循环左移
 return ((X<<i)& 0xffffffff)|(X>>(32-i))
def L_(x):#线性L变换 32输入32输出
    return x^xun(x,2)^xun(x,10)^xun(x,18)^xun(x,24)
```

然后我们构造 S 盒的差分分布表，SM4S 盒是 8 输入 8 输出的 S 盒，其 S 盒差分分布表为 256*256 的表，代码如下：

```
diff_table = np.zeros((256, 256), dtype=np.uint8)
oresult = []#初始化大于2差分概率的 列表
iresult = []#初始化大于2差分概率的 列表
result = []
# 打印差分分布表
def find_differential_distribution_table(table):
    print(f"{'Input Diff':^10s} | {'Output Diff':^10s} | {'Count':^10s}")
    print("-" * 34)
    for input_diff in range(256):
        for output_diff in range(256):
            count = table[input_diff, output_diff]
            if count >= 2:
```

```
            #if input_diff==0x02 and output_diff==0x81:
              #print(hex(input_diff),hex(output_diff),count)
            #print(hex(input_diff),hex(output_diff),count)
            result.append([input_diff,output_diff])
            oresult.append(output_diff)
            iresult.append(input_diff)
if __name__ == '__main__':
  for i in range(256):
    for j in range(256):
      #print(hex(i), hex(j))
      din=i^j
      dout=s_(i)^s_(j)
      diff_table[din][dout]=diff_table[din][dout]+ 1
  find_differential_distribution_table(diff_table)
```

下面我们需要寻找所有符合条件的 α, 即形如 $(0, x_1, x_2, x_3)$，其中 $x_i \in Z_2^8 \setminus \{0\}$. 同时，由于 T 是 $\alpha \to \alpha$ 的差分，S 盒的输出差分 $L^{-1}(\alpha)$ 也必须形如 $(0, y_1, y_2, y_3)$，其中 $y_i \in Z_2^8 \setminus \{0\}$，为了代码的兼容性，我们没有构造 $L^{-1}(\alpha)$ 函数，而是遍历 $L^{-1}(\alpha)$ 并计算 $L(L^{-1}(\alpha))$, 这样做还有一个好处，即将搜索复杂度从 2^{32} 降低到 2^{24}, 代码如下:

```
for k in range(0x00ffffff):
  if L_(k)<0x00ffffff:
      if [(L_(k) >> 16) & 0xFF,(k >> 16) &0xFF] in result :
        if [(L_(k) >> 8) & 0xFF,(k >> 8) &0xFF] in result :
          if [(L_(k)) & 0xFF,(k) &0xFF] in result :
            #print("\n 输入差分",hex(L_(k)),"输出差分",hex(k))
            pr=P(k)
            print(str(num),"\n  输入差分",str(hex(L_(k))),"输出差分",str(hex(k)),str(pr),file=f)
            num=num+1
```

运行结果如下，找到了 7905 条符合条件的差分路线:

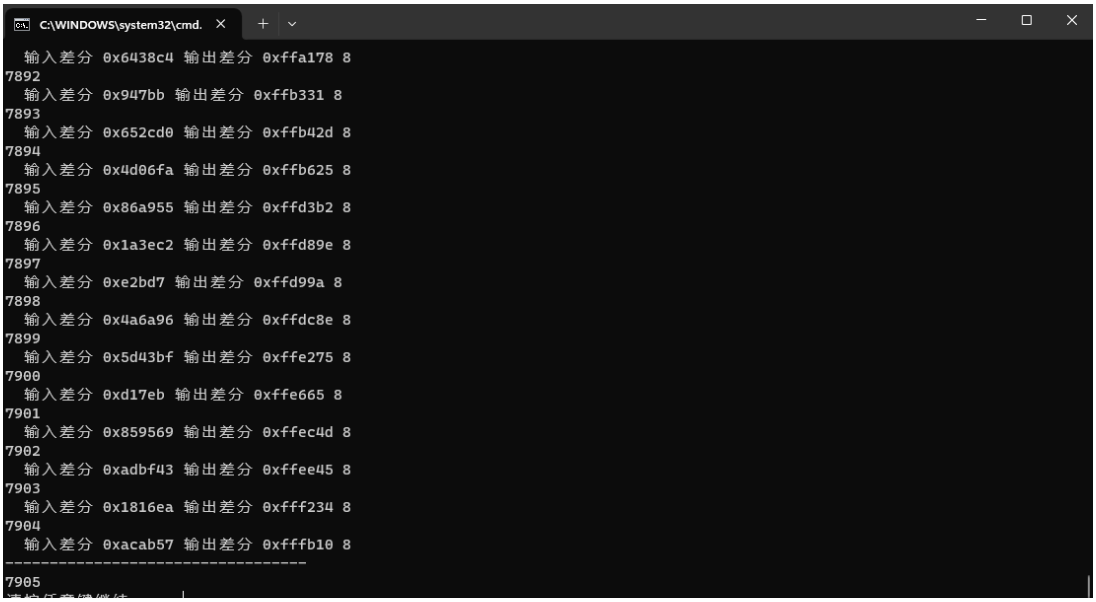

Figure 3: 搜索到 7905 条符合条件的差分路线

其中只有三条概率最大为 2^{-19}，即三个活跃 S 盒的传播概率分别为 $2^{-7}/2^{-6}/2^{-6}$，S 盒的输入和输出差分分别是:

$$002cf5cd \to 00383904$$

$$00c30290 \to 00908145$$

$$00d2c822 \to 00e9bf58$$

修改搜索范围，搜索 α 形如 $(x_1, x_2, x_3, 0)$ 的形式，得到以下结果:

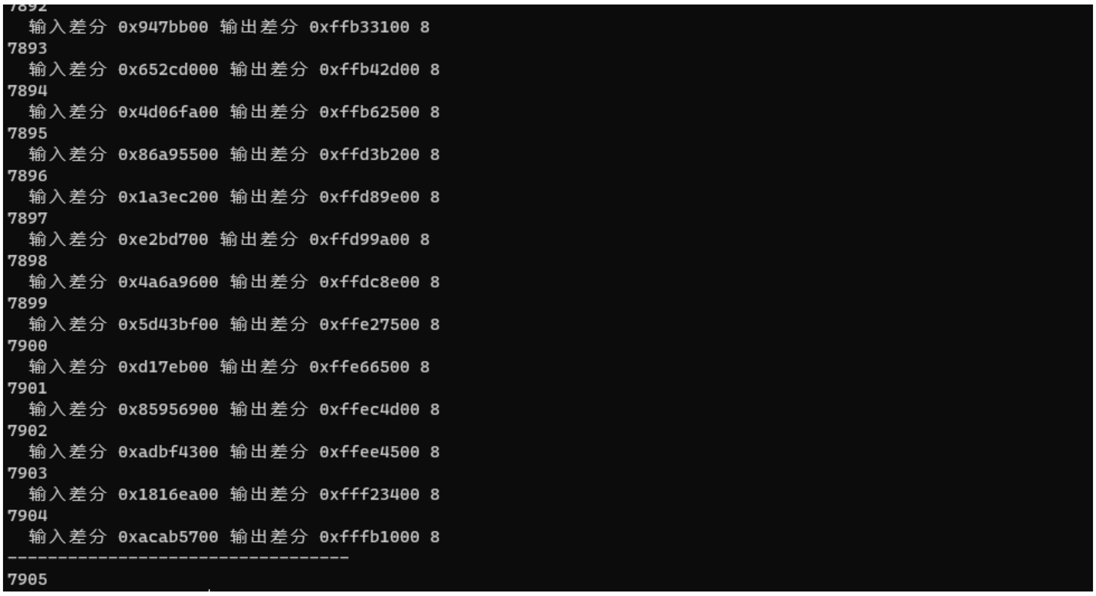

Figure 4: 同样搜索到 7905 条符合条件的差分路线

同样,其中只有三条概率最大为 2^{-19},即三个活跃 S 盒的传播概率分别为 $2^{-7}/2^{-6}/2^{-6}$, S 盒的输入和输出差分分别是:

$$2cf5cd00 \to 38390400$$

$$c3029000 \to 90814500$$

$$d2c82200 \to e9bf5800$$

可以看到这三条结果与之前找到的三条只有比特的循环移位区别，从而验证了在 Part2 实验准备中的讨论，即四种 α 形式事实上是等价的，具有相同的差分路径数和循环移位相同的差分路线。

3.2 利用 U 方法搜索 SM4 的不可能差分路线

我们利用实验 3 中的 u 方法，对 SM4 进行不可能差分路线的自动化搜索，根据 SM4 的算法结构，将算法用矩阵刻画，分别遍历输入输出差分以 1 概率传播的所有可能，部分代码如下:

```
E = [['0','1','0','0'],
     ['0','0','1','0'],
     ['0','0','0','1'],
     ['1','f','f','f']]
```

```
D = [['f','f','f','1'],
     ['1','0','0','0'],
     ['0','1','0','0'],
     ['0','0','1','0']]

     pc = [['0' if j == '0' else '1*' for j in bin(i)[2:].zfill(4)] for i in range(0,16)]

pi = []
ci = []
for t in pc[1:]:
    pi.append(CLF.enc(t,10))
    ci.append(CLF.dec(t,10))

def check(x,y):
    for i in range(4):
        if(x[i] == '0'):
            if (y[i] == '1')|(y[i] == '1*'):
                return 0
        elif x[i] == '1':
            if y[i] == '0':
                return 0
        elif x[i] == '1*':
            if (y[i] != '1')&(y[i] != 't'):
                return 0
        elif x[i] == '2*':
            if y[i] == '1*':
                return 0
        else:
            return 1

def reshape(l):
    return [l[1],l[0],l[3],l[2]]

for i in range(len(pi)):
    for j in range(len(ci)):
        p = pi[i]
        c = ci[j]
```

```python
    maxn = 0
    pl0,cl0 = 0,0
    for pl in range(10):
        for cl in range(10):
            if(check(p[pl],c[cl])==0) & (maxn < pl+cl+3):
                pl0 = pl
                cl0 = cl
                maxn = pl+cl+2
    if(maxn >= 6):
        print(maxn,pc[i+1],reshape(pc[j+1]),pl0,cl0,p[pl0],c[cl0])
```

得到两条路径如下运行结果所示：

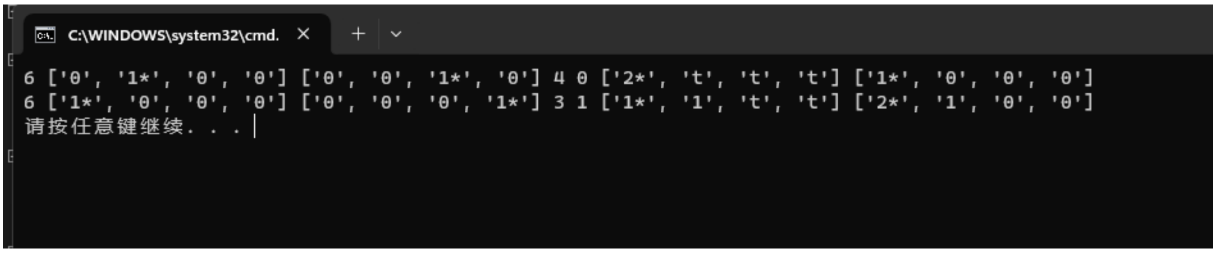

Figure 5: 利用 U 方法自动化搜索

找到两条 6 轮的不可能差分，分别是 $(0,\alpha,0,0) \to (0,0,\alpha,0)$ 和 $(\alpha,0,0,0) \to (0,0,0,\alpha)$

3.3 寻找 SM4 的线性路线

首先求 SM4 的 S 盒线性近似表。SM4 使用 8Bit→8bit 的 S 盒，共 $2^8 = 256$ 对输入输出，构建线性近似表时需要遍历 256*256 对输入、输出掩码，对每一对掩码遍历 256 对 S 盒输入输出，统计表达式为 0 的个数，减去 128 并将结果储存在 256*256 的线性近似表中。具体代码如下：

```python
import time
import numpy as np
lenoflat=64
sbox = [[0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14, 0xc2, 0x28, 0xfb, 0x2c, 0x05,],
```

```
[0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,],
[0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b, 0x43, 0xed, 0xcf, 0xac, 0x62,],
[0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94, 0xfa, 0x75, 0x8f, 0x3f, 0xa6,],
[0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c, 0x19, 0xe6, 0x85, 0x4f, 0xa8,],
[0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f, 0x4b, 0x70, 0x56, 0x9d, 0x35,],
[0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c, 0x3b, 0x01, 0x21, 0x78, 0x87,],
[0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02, 0xe7, 0xa0, 0xc4, 0xc8, 0x9e,],
[0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2, 0xce, 0xf9, 0x61, 0x15, 0xa1,],
[0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32, 0x30, 0xf5, 0x8c, 0xb1, 0xe3,],
[0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23, 0xab, 0x0d, 0x53, 0x4e, 0x6f,],
[0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a, 0x72, 0x6d, 0x6c, 0x5b, 0x51,],
[0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c, 0x41, 0x1f, 0x10, 0x5a, 0xd8,],
[0x0a, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0, 0x12, 0xb8, 0xe5, 0xb4, 0xb0,],
[0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1, 0x09, 0xc5, 0x6e, 0xc6, 0x84,],
[0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f, 0x3e, 0xd7, 0xcb, 0x39, 0x48]]

def S(x):                    #s盒8位输入，查表字节代换8位输出
    row = (x>>4) & 0xf       #前4位行索引
    col = x & 0xf            #后4位列索引
    return sbox[row][col]

LAT_T=[]
t=[]#存储每一行偏差的最大值
```

```
mask_1 = 0x1

for t_out in range(256):#遍历输出掩码
    temp=[]
    for t_in in range(lenoflat):#遍历输入掩码
        ans=-128
        for s_in in range(256): #遍历s盒的256对输入输出
            s_out = S(s_in) #计算s盒输出
            res = 0 #线性近似表达式的值
            #计算线性近似表达式
            for x in range(8):
                if((t_in>>x) & mask_1==1):
                    res ^= ((s_in>>x) & mask_1)
            for y in range(8):
                if((t_out>>y) & mask_1==1):
                    res ^= ((s_out>>y) & mask_1)
            if(res==1): ans+=1
        temp.append(abs(ans))
        t.append(max(temp))
    LAT_T.append(temp)

#构造线性运算矩阵的转置Lt
lt=[[0]*32]*32


Lt=np.array([lt])

#将整数转换为32位向量
def int_to_vector(int_val):
    binary_str = np.binary_repr(int_val, width=32)
    vector = np.array(list(binary_str), dtype=int)
    return vector


result=[]
Gamma=[[0]*4,[0]*4,[0]*4,[0]*4]#每一轮的输入
for l in range(2,17):
    for j in range(4):#一组4个S盒
```

```
#for遍历j位置的所有不为0的s盒输出掩码, 同时其他S盒输出掩码为0:
Lam=[[0]*4,[0]*4,[0]*4]#三轮S盒输出掩码,非j位置的所有s盒输出为0
Gam=[[0]*4,[0]*4,[0]*4]#三轮S盒输入掩码

for Lam0j in range(256):
    for Lam1j in range(256):
        for Lam2j in range(256):
            Lam[0][j]=Lam0j
            Lam[1][j]=Lam1j
            Lam[2][j]=Lam2j
            R0=[]
            R1=[]
            R2=[]
            for i0 in range(lenoflat):#找到偏差最大的
                if (LAT_T[Lam0j][i0]==t[Lam0j]): R0.append(i0)
            for i1 in range(lenoflat):#找到偏差最大的
                if (LAT_T[Lam1j][i1]==t[Lam1j]): R1.append(i1)
            for i2 in range(lenoflat):#找到偏差最大的
                if (LAT_T[Lam2j][i1]==t[Lam2j]): R2.append(i2)
            #
            gamma=[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
            for k0 in range(len(R0)):
                Gam[0][j]=R0[k0]
                for k1 in range(len(R1)):
                    Gam[1][j]=R1[k1]
                    for k2 in range(len(R2)):
                        Gam[2][j]=R2[k2]
                        gam=[0,0,0]
                        for i in range(3):
                            tt=((Lam[i][0]&0xff)<<24) + ((Lam[i][1]&0xff)<<16) + ((Lam[i][2]&0xff)<<8) + ((Lam[i][3]&0xff))
                            lam=int_to_vector(tt)
                            Gamma[i][0]= np.dot(Lt,lam)
                        for i in range(3):
                            gam[i]=((Gam[i][0]&0xff)<<24)+((Gam[i][1]&0xff)<<16)+((Gam[i][2]&0xff)<<8)+((Gam[i][3]&0xff))
                            Gamma[3][0]=Gamma[0][0]
                            Gamma[3][1]=Gamma[0][0]^gam[1]^gam[2]
```

```
                        Gamma[3][2]=Gamma[1][0]^gam[2]
                        Gamma[0][3]=Gamma[0][0]^gam[0]^gam[1]^gam[2]
                        Gamma[0][1]=Gamma[1][0]^gam[0]
                        Gamma[0][2]=Gamma[2][0]^gam[0]^gam[1]
                        if ((Gamma[0][3]==Gamma[2][0])and(Gamma[0][1]==Gamma
[3][1])and(Gamma[0][2]==Gamma[3][2])):
                            result= [Gam[0][0],Gam[1][0],Gam[2][0],Gam[3][0],gam[0],gam
[1],gam[2]]
                    else:
                        continue
l=l-2
```

参考文献

[1] 王美琴等. 密码分析学 [M]. 2023.

[2] Zhang L, Zhang W, Wu A W. Cryptanalysis of Reduced-Round SMS4 Block Cipher[J]. information security and privacy. 2008.

[3] Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer, Heidelberg .1993.

[4] J. Lu, “Attacking Reduced-Round Versions of the SMS4 Block Cipher in the Chinese WAPI Standard” , Proc. ICICS 2007, LNCS 4861, pp. 306-318, 2007.

[5] Lu, J., Kim, J., Keller, N., Dunkelman, O.: Improving the efficiency of impossible differential cryptanalysis of reduced Camellia and MISTY1, Archive available at: http://jiqiang.googlepages.com

[6] Liu Y ,Liang H ,Wang W , et al. New Linear Cryptanalysis of Chinese Commercial Block Cipher Standard SM4[J]. Security and Communication Networks,2017,2017.

A 附录

A.1 寻找线性近似式

```
import time
import numpy as np
lenoflat=64
sbox = [[0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14, 0xc2, 0x28, 0xfb, 0x2c, 0x05,],
    [0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,],
    [0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b, 0x43, 0xed, 0xcf, 0xac, 0x62,],
    [0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94, 0xfa, 0x75, 0x8f, 0x3f, 0xa6,],
    [0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c, 0x19, 0xe6, 0x85, 0x4f, 0xa8,],
    [0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f, 0x4b, 0x70, 0x56, 0x9d, 0x35,],
    [0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c, 0x3b, 0x01, 0x21, 0x78, 0x87,],
    [0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02, 0xe7, 0xa0, 0xc4, 0xc8, 0x9e,],
    [0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2, 0xce, 0xf9, 0x61, 0x15, 0xa1,],
    [0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32, 0x30, 0xf5, 0x8c, 0xb1, 0xe3,],
    [0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23, 0xab, 0x0d, 0x53, 0x4e, 0x6f,],
    [0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a, 0x72, 0x6d, 0x6c, 0x5b, 0x51,],
    [0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c, 0x41, 0x1f, 0x10, 0x5a, 0xd8,],
    [0x0a, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0, 0x12, 0xb8, 0xe5, 0xb4, 0xb0,],
    [0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1, 0x09, 0xc5, 0x6e, 0xc6, 0x84,],
    [0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f, 0x3e, 0xd7, 0xcb, 0x39, 0
```

```
    x48]]

def S(x):                    #s盒8位输入，查表字节代换8位输出
    row = (x>>4) & 0xf          #前4位行索引
    col = x & 0xf              #后4位列索引
    return sbox[row][col]
#将二进制数组转化为整数
def bin_array_to_int(bin_array):
    binary_str = "".join(str(x) for x in bin_array)
    print(binary_str)
    return int(binary_str, 2)


LAT_T=[]
t=[]#存储每一行偏差的最大值
mask_1 = 0x1

for t_out in range(256):#遍历输出掩码
    temp=[]
    for t_in in range(lenoflat):#遍历输入掩码
        ans=-128
        for s_in in range(256):  #遍历s盒的256对输入输出
            s_out = S(s_in)  #计算s盒输出
            res = 0  #线性近似表达式的值
            #计算线性近似表达式
            for x in range(8):
                if((t_in>>x) & mask_1==1):
                    res ^= ((s_in>>x) & mask_1)
            for y in range(8):
                if((t_out>>y) & mask_1==1):
                    res ^= ((s_out>>y) & mask_1)
            if(res==1): ans+=1
        temp.append(abs(ans))
        t.append(max(temp))
    LAT_T.append(temp)

#构造线性运算矩阵的转置Lt
lt=[[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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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],
    [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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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, 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],
    [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, 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, 0, 0, 0, 0]]
shift=[0,2,10,18,24]

for i in range(32):
    for j in range(5):
        lt[i][shift[j]]=1
        shift[j]= (shift[j]+1)%32

```

```
Lt=np.array(lt)

#将整数转换为32位向量
def int_to_vector(int_val):
    binary_str = np.binary_repr(int_val, width=32)
    vector = np.array(list(binary_str), dtype=int)
    return vector


result=[]
Gamma=[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]#每一轮的输入
for l in range(2,17):

    for j in range(4):#一组4个S盒
        #for遍历j位置的所有不为0的s盒输出掩码，同时其他S盒输出掩码为0:
        Lam=[[0,0,0,0],[0,0,0,0],[0,0,0,0]]#三轮S盒输出掩码,非j位置的所有s盒输出为0
        Gam=[[0,0,0,0],[0,0,0,0],[0,0,0,0]]#三轮S盒输入掩码
        for Lam0j in range(256):
            for Lam1j in range(256):
                for Lam2j in range(256):
                    Lam[0][j]=Lam0j & 0xff
                    #print(Lam[0][j])
                    Lam[1][j]=Lam1j & 0xff
                    Lam[2][j]=Lam2j & 0xff
                    R0=[]
                    R1=[]
                    R2=[]
                    for i0 in range(lenoflat):#找到偏差最大的
                        if (LAT_T[Lam0j][i0]!=0): R0.append(i0)
                    for i1 in range(lenoflat):#找到偏差最大的
                        if (LAT_T[Lam1j][i1]!=0): R1.append(i1)
                    for i2 in range(lenoflat):#找到偏差最大的？
                        #if (LAT_T[Lam2j][i1]==t[Lam2j]): R2.append(i2)
                        if (LAT_T[Lam2j][i1]!=0): R2.append(i2)
                    #
                    #gamma=[[0,0,0,0],[0,0,0,0],[0,0,0,0],[0,0,0,0]]
                    #print(hex(len(R0)))
                    for k0 in range(len(R0)):
```

```
                    Gam[0][j]=R0[k0]
                    for k1 in range(len(R1)):
                        Gam[1][j]=R1[k1]
                        for k2 in range(len(R2)):
                            Gam[2][j]=R2[k2]
                            gam=[0,0,0]

                            for i in range(3):
                                tt=((Lam[i][0]&0xff)<<24) + ((Lam[i][1]&0xff)<<16) + ((Lam[i][2]&0xff)<<8) + ((Lam[i][3]&0xff))
                                lam=int_to_vector(tt)
                                #print(Lt)
                                #print('*********')
                                #print(lam)
                                #print('*********')
                                tempp = np.dot(lam,Lt)%2
                                #print(tempp)
                                #print('*********')
                                Gamma[i][0]= bin_array_to_int(tempp)
                                #print(bin_array_to_int(tempp))
                                #print('************************')
                            for i in range(3):
                                gam[i]=((Gam[i][0]&0xff)<<24)+((Gam[i][1]&0xff)<<16)+((Gam[i][2]&0xff)<<8)+((Gam[i][3]&0xff))
                            Gamma[3][0]=Gamma[0][0]
                            Gamma[3][1]=Gamma[0][0]^gam[1]^gam[2]
                            Gamma[3][2]=Gamma[1][0]^gam[2]
                            Gamma[0][3]=Gamma[0][0]^gam[0]^gam[1]^gam[2]
                            Gamma[0][1]=Gamma[1][0]^gam[0]
                            Gamma[0][2]=Gamma[2][0]^gam[0]^gam[1]
                            if ((Gamma[0][3]==Gamma[2][0])and(Gamma[0][1]==Gamma[3][1])and(Gamma[0][2]==Gamma[3][2])):
                                result= [Gam[0][0],Gam[1][0],Gam[2][0],Gam[3][0],gam[0],gam[1],gam[2]]
                            else:
                                continue
l=l-2
```

A.2 寻找差分近似式

```python
import numpy as np
from tqdm import tqdm

sbox = [[0xd6, 0x90, 0xe9, 0xfe, 0xcc, 0xe1, 0x3d, 0xb7, 0x16, 0xb6, 0x14, 0xc2, 0x28, 0xfb, 0x2c, 0x05,],
    [0x2b, 0x67, 0x9a, 0x76, 0x2a, 0xbe, 0x04, 0xc3, 0xaa, 0x44, 0x13, 0x26, 0x49, 0x86, 0x06, 0x99,],
    [0x9c, 0x42, 0x50, 0xf4, 0x91, 0xef, 0x98, 0x7a, 0x33, 0x54, 0x0b, 0x43, 0xed, 0xcf, 0xac, 0x62,],
    [0xe4, 0xb3, 0x1c, 0xa9, 0xc9, 0x08, 0xe8, 0x95, 0x80, 0xdf, 0x94, 0xfa, 0x75, 0x8f, 0x3f, 0xa6,],
    [0x47, 0x07, 0xa7, 0xfc, 0xf3, 0x73, 0x17, 0xba, 0x83, 0x59, 0x3c, 0x19, 0xe6, 0x85, 0x4f, 0xa8,],
    [0x68, 0x6b, 0x81, 0xb2, 0x71, 0x64, 0xda, 0x8b, 0xf8, 0xeb, 0x0f, 0x4b, 0x70, 0x56, 0x9d, 0x35,],
    [0x1e, 0x24, 0x0e, 0x5e, 0x63, 0x58, 0xd1, 0xa2, 0x25, 0x22, 0x7c, 0x3b, 0x01, 0x21, 0x78, 0x87,],
    [0xd4, 0x00, 0x46, 0x57, 0x9f, 0xd3, 0x27, 0x52, 0x4c, 0x36, 0x02, 0xe7, 0xa0, 0xc4, 0xc8, 0x9e,],
    [0xea, 0xbf, 0x8a, 0xd2, 0x40, 0xc7, 0x38, 0xb5, 0xa3, 0xf7, 0xf2, 0xce, 0xf9, 0x61, 0x15, 0xa1,],
    [0xe0, 0xae, 0x5d, 0xa4, 0x9b, 0x34, 0x1a, 0x55, 0xad, 0x93, 0x32, 0x30, 0xf5, 0x8c, 0xb1, 0xe3,],
    [0x1d, 0xf6, 0xe2, 0x2e, 0x82, 0x66, 0xca, 0x60, 0xc0, 0x29, 0x23, 0xab, 0x0d, 0x53, 0x4e, 0x6f,],
    [0xd5, 0xdb, 0x37, 0x45, 0xde, 0xfd, 0x8e, 0x2f, 0x03, 0xff, 0x6a, 0x72, 0x6d, 0x6c, 0x5b, 0x51,],
    [0x8d, 0x1b, 0xaf, 0x92, 0xbb, 0xdd, 0xbc, 0x7f, 0x11, 0xd9, 0x5c, 0x41, 0x1f, 0x10, 0x5a, 0xd8,],
    [0x0a, 0xc1, 0x31, 0x88, 0xa5, 0xcd, 0x7b, 0xbd, 0x2d, 0x74, 0xd0, 0x12, 0xb8, 0xe5, 0xb4, 0xb0,],
    [0x89, 0x69, 0x97, 0x4a, 0x0c, 0x96, 0x77, 0x7e, 0x65, 0xb9, 0xf1, 0x09, 0xc5, 0x6e, 0xc6, 0x84,],
    [0x18, 0xf0, 0x7d, 0xec, 0x3a, 0xdc, 0x4d, 0x20, 0x79, 0xee, 0x5f, 0x3e, 0xd7, 0xcb, 0x39, 0x48]]

```

```
FK0=0XA3B1BAC6
FK1=0X56AA3350
FK2=0X677D9197
FK3=0XB27022DC

CK = [0x00070e15, 0x1c232a31, 0x383f464d, 0x545b6269, 0x70777e85, 0x8c939aa1, 0xa8afb6bd, 0xc4cbd2d9,
      0xe0e7eef5, 0xfc030a11, 0x181f262d, 0x343b4249, 0x50575e65, 0x6c737a81, 0x888f969d, 0xa4abb2b9,
      0xc0c7ced5, 0xdce3eaf1, 0xf8ff060d, 0x141b2229, 0x30373e45, 0x4c535a61, 0x686f767d, 0x848b9299,
      0xa0a7aeb5, 0xbcc3cad1, 0xd8dfe6ed, 0xf4fb0209, 0x10171e25, 0x2c333a41, 0x484f565d, 0x646b7279]

rk = [0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
      0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
  #初始轮密钥全部置零

def s_(x):#s盒 8输入8输出
    row = x >> 4
    nuw = x & 0xf
    return sbox[row][nuw]

def T_(x):#字变换组合 32输入32输出
    a0 = (x >> 24) & 0xff
    a1 = (x >> 16) & 0xff
    a2 = (x >> 8) & 0xff
    a3 = (x >> 0) & 0xff
    B = (s_(a0)<<24)^(s_(a1)<<16)^(s_(a2)<<8)^(s_(a3)<<0)#先s盒变换
    return L_(B)#再L变换

def T(x):#密钥扩展字变换
    a0 = (x >> 24) & 0xff
    a1 = (x >> 16) & 0xff
    a2 = (x >> 8) & 0xff
    a3 = (x >> 0) & 0xff
    B = (s_(a0) << 24) ^ (s_(a1) << 16) ^ (s_(a2) << 8) ^ (s_(a3) << 0)  # 先s盒变换
```

```
    return B ^ (xun(B,13)) ^ (xun(B,23))

def K_(MK0,MK1,MK2,MK3):#SM4密钥扩展算法
    K = [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, 0, 0, 0, 0, 0, 0]
    K[0] = MK0 ^ FK0
    K[1] = MK1 ^ FK1
    K[2] = MK2 ^ FK2
    K[3] = MK3 ^ FK3
    for i in range (32):
        rk[i]=K[i+4]=K[i]^T(K[i+1]^K[i+2]^K[i+3]^CK[i])

def SMJ(x):#SM4加密算法
    X = [0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
         0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0]
    X[0] = (x >> 96) & 0xffffffff
    X[1] = (x >> 64) & 0xffffffff
    X[2] = (x >> 32) & 0xffffffff
    X[3] = (x >> 0) & 0xffffffff
    for i in range(33):
        X[i+4] = X[i]^T_(X[i+1]^X[i+2]^X[i+3]^rk[i])#轮函数
    return (X[35]<<96)^(X[34]<<64)^(X[33]<<32)^X[32]

diff_table = np.zeros((256, 256), dtype=np.uint8)
oresult = []#初始化大于2差分概率的 列表
iresult = []#初始化大于2差分概率的 列表
result = []
# 打印差分分布表
def find_differential_distribution_table(table):
    print(f"{'Input Diff':^10s} | {'Output Diff':^10s} | {'Count':^10s}")
    print("-" * 34)
    for input_diff in range(256):
        for output_diff in range(256):
            count = table[input_diff, output_diff]
            if count >= 2:
                #if input_diff==0x02 and output_diff==0x81:
```

```
                #print(hex(input_diff),hex(output_diff),count)
               #print(hex(input_diff),hex(output_diff),count)
               result.append([input_diff,output_diff])
               oresult.append(output_diff)
               iresult.append(input_diff)


def xun(X,i):#循环左移
 return ((X<<i)& 0xffffffff)|(X>>(32-i))


def L_(x):#线性L变换 32输入32输出
   return x^xun(x,2)^xun(x,10)^xun(x,18)^xun(x,24)

def L_inverse(x):#线性L的逆变换 32输入32输出
   M2 = x^xun(x, 24)
   M1 = M2^xun(M2, 2)^xun(M2, 10)^xun(M2, 18)
   return M1^xun(M1, 24)

def xor(a, b):
   return a^b

def P(k):
   x1=(L_(k) >> 16) & 0xFF
   x2=(k >> 16) &0xFF  #k是输出差分
   y1=(L_(k) >> 8) & 0xFF
   y2=(k >> 8) &0xFF  #k是输出差分
  # z1=(L_(k) ) & 0xFF
  # z2=(k ) &0xFF  #k是输出差分
   z1=(L_(k)>>24 ) & 0xFF
   z2=(k>>24 ) &0xFF
   p=diff_table[x1][x2]*diff_table[y1][y2]*diff_table[z1][z2]
   return p

if __name__ == '__main__':

  for i in range(256):
    for j in range(256):
```

```
        #print(hex(i), hex(j))
        din=i^j
        dout=s_(i)^s_(j)
        diff_table[din][dout]=diff_table[din][dout]+ 1
find_differential_distribution_table(diff_table)
#print("Input diffs with count >= 2:",result)
x_list = []  #遍历逆L
print("例子1: ",hex(L_(0x00908145)))  #输出差分经过L变换成为输入差分0xc30290
print("例子1: ",hex(L_(0x00010c34)))  #输入差分0x00e5edec
print("-" * 34)
num=0  #论文提到7905
f = open('log.txt','w')
#for k in range(0x00ffffff):
for k in range(0x00000000, 0xffffff00,0x100):
  #if L_(k)<0x00ffffff:
  if(L_(k)) & 0xFF==0x00:
        '''if [(L_(k) >> 16) & 0xFF,(k >> 16) &0xFF] in result :
            if [(L_(k) >> 8) & 0xFF,(k >> 8) &0xFF] in result :
                if [(L_(k)) & 0xFF,(k) &0xFF] in result :
                    #print("\n 输入差分",hex(L_(k)),"输出差分",hex(k))
                    pr=P(k)
                    #print(str(num),"\n  输入差分",str(hex(L_(k))),"输出差分",str(hex(k)),str(
   pr),file=f)
                    print(str(num),"\n  输入差分",str(hex(L_(k))),"输出差分",str(hex(k)),str(
   pr))
                    num=num+1'''
        if [(L_(k) >> 24) & 0xFF,(k >> 24) &0xFF] in result :
            if [(L_(k) >> 16) & 0xFF,(k >> 16) &0xFF] in result :
                if [(L_(k)>>8) & 0xFF,(k>>8) &0xFF] in result :
                    #print("\n 输入差分",hex(L_(k)),"输出差分",hex(k))
                    pr=P(k)
                    print(str(num),"\n  输入差分",str(hex(L_(k))),"输出差分",str(hex(k)),str(
   pr),file=f)
                    #print(str(num),"\n  输入差分",str(hex(L_(k))),"输出差分",str(hex(k)),str(
   pr))
                    num=num+1
f.close()
'''
```

```
for input_diff_1,_,_ in tqdm(result4):
    for input_diff_2,_,_ in result4:
        for input_diff_3,_,_ in result4:
            x = (input_diff_1 << 16) | (input_diff_2 << 8) | input_diff_3
            x_list.append(x)
            #num=num+1

print("Generated x list:")
'''
for x in x_list:
  print(x)

print("-" * 34)
print(num)
#print(hex(T_(L_(0x00010c34))))
#print(hex(L_(0x00010c34)))

#都是0x802b059b
```

A.3 u 方法寻找不可能差

```
class U_way:
    def __init__(self,E,D) -> None:
        self.E = E
        self.D = D

    def mul(self,x,y):
        # print(x,y)
        if y == '1':
            return x
        elif y == '0':
            return '0'
        elif x == '1*':
            return '1'
        elif x == '2*':
            return 't'
```

```
    else:
        return x

def add(self,x,y):
    # print(x,y)
    if x == '0':
        return y
    elif y == '0':
        return x
    # elif (x == '1') & (y == '1'):
    #     return 't'
    elif ((x == '1') & (y == '1*')) | ((x == '1*') & (y == '1')):
        return '2*'
    # elif ((x == '1') & (y == '2*')) | ((x == '2*') & (y == '1')):
    #     return 't'
    else:
        return 't'

def mat_mul(self,X,M):
    res = ['0']*len(M)
    for i in range(len(M)):
        for j in range(len(M[i])):
            tmp = self.mul(X[j],M[i][j])
            # print(tmp)
            res[i] = self.add(res[i],tmp)
    return res

def enc(self,P,r) -> list:
    l = []
    for _ in range(r):
        P = self.mat_mul(P,self.E)
        l.append(P)
        # print(_+1,P)
    return l

def dec(self,C,r):
    l = []
    for _ in range(r):
```

```
            C = self.mat_mul(C,self.D)
            l.append(C)
            # print(_+1,C)
        return l

E = [['0','1','0','0'],
     ['0','0','1','0'],
     ['0','0','0','1'],
     ['1','f','f','f']]

D = [['f','f','f','1'],
     ['1','0','0','0'],
     ['0','1','0','0'],
     ['0','0','1','0']]


# E = [['f','1'],
#      ['1','0']]

CLF = U_way(E,D)
# tmp.enc(['1','0','0','0'],5)
# CLF.enc(['0','0','0','1*'],10)
# tmp.dec(['1*','0','0','0'],10)
# tmp.enc(['1*','0','0','0'],10)

pc = [['0' if j == '0' else '1*' for j in bin(i)[2:].zfill(4)] for i in range(0,16)]

pi = []
ci = []
for t in pc[1:]:
    pi.append(CLF.enc(t,10))
    ci.append(CLF.dec(t,10))

def check(x,y):
    for i in range(4):
        if(x[i] == '0'):
            if (y[i] == '1')|(y[i] == '1*'):
                return 0
```

```
        elif x[i] == '1':
            if y[i] == '0':
                return 0
        elif x[i] == '1*':
            if (y[i] != '1')&(y[i] != 't'):
                return 0
        elif x[i] == '2*':
            if y[i] == '1*':
                return 0
        else:
            return 1

def reshape(l):
    return [l[1],l[0],l[3],l[2]]

for i in range(len(pi)):
    for j in range(len(ci)):
        p = pi[i]
        c = ci[j]
        maxn = 0
        pl0,cl0 = 0,0
        for pl in range(10):
            for cl in range(10):
                if(check(p[pl],c[cl])==0) & (maxn < pl+cl+3):
                    pl0 = pl
                    cl0 = cl
                    maxn = pl+cl+2
        if(maxn >= 6):
            print(maxn,pc[i+1],reshape(pc[j+1]),pl0,cl0,p[pl0],c[cl0])
```