基于 NFSR 的分组密码算法 SPRING

田甜　　戚文峰　　叶晨东　　谢晓锋

摘　要

本文介绍分组密码算法 SPRING-n-m的设计参数以及相关的分析评估结果，其中n表示分组长度，m表示密钥长度。SPRING-n-m是 SP 结构的分组密码，主要面向硬件实现设计，采用基于非线性反馈移位寄存器（NFSR）的32-比特 S-盒和基于非线性反馈移位寄存器的密钥扩展算法。由于没有 S-盒的存储，SPRING 算法的硬件实现面积比较小。根据不同的应用需求，可以选择不同的实现方式。基于轮的实现，硬件面积最小，在 TSMC 16nm 工艺库下，SPRING-128-128 算法的硬件实现面积只有约 1046 平方微米；全部轮函数的展开实现，加/解密速率最大， SPRING-128-128 算法的加密速率可以达到 2084 $Mbps$，此时硬件实现面积约8079平方微米。

SPRING 的含义为 **SP** 结构分组密码和环状串联非线性反馈移位寄存器（A **Ring**-like cascade connection of NFSRs）。

目 录

1. 算法描述

本节详细介绍分组密码算法 SPRING 的设计参数。

1.1 概述

SPRING 是 SP 结构的迭代分组密码，支持的分组长度/密钥长度参数为128/128、128/256和256/256。SPRING-n-m表示分组长度为n且密钥长度为m的分组密码算法。迭代轮数N_r由分组长度和密钥长度决定，参见表 1。加密算法和解密算法的结构如图 1。

分组长度 \ 密钥长度	128	256
128	10	14
256	–	18

表 1　加密轮数与分组长度/密钥长度的关系

ENCRYPTION($Plaintext$, Key)
{
　$State \leftarrow Plaintext$;
　$(k_1, k_2, \ldots, k_{N_r+1}) =$ KeyExpansion(Key);
　For $i = 1$ to $N_r - 1$ do
　　Round($State$, k_i);
　End For
　FinalRound($State$, k_{N_r}, k_{N_r+1});
}

DECRYPTION($Ciphertext$, Key)

{

$State \leftarrow Ciphertext$;

$(k_1^{inv}, k_2^{inv}, \ldots, k_{N_r+1}^{inv}) =$ InvKeyExpansion(Key);

For $i = 1$ to $N_r - 1$ do

InvRound($State$, k_1^{inv});

End For

FinalRound($State$, $k_{N_r}^{inv}, k_{N_r+1}^{inv}$);

图 1　ENCRYPTION 和 DECRYPTION 函数的描述

1.2 SPRING-128-128的设计

1.2.1 SPRING-128-128的整体结构

SPRING-128-128的内部状态以字节为单位，看作$\mathbb{F}_{2^8}$上的16维向量或者$\mathbb{F}_{2^8}$上4×4的矩阵，即

$$State = s_0 \| s_1 \| s_2 \| s_3 \| s_4 \| s_5 \| s_6 \| s_7 \| s_8 \| s_9 \| s_{10} \| s_{11} \| s_{12} \| s_{13} \| s_{14} \| s_{15}$$

$$= (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15})$$

$$= \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{pmatrix},$$

其中s_i为一个字节，即8比特，$0 \le i \le 15$。设128比特的明文分组为$p_0 \| p_1 \| \cdots \| p_{15}$，密文分组为$c_0 \| c_1 \| \cdots \| c_{15}$。

SPRING-128-128 加密算法的整体结构见图 2，而解密算法的结构与加密算法结构是完全一致的，即当图 2 输入为密文、NFSR-SR 替换为 NFSR-SR-Inv、轮子密钥为解密密钥时，输出为明文。

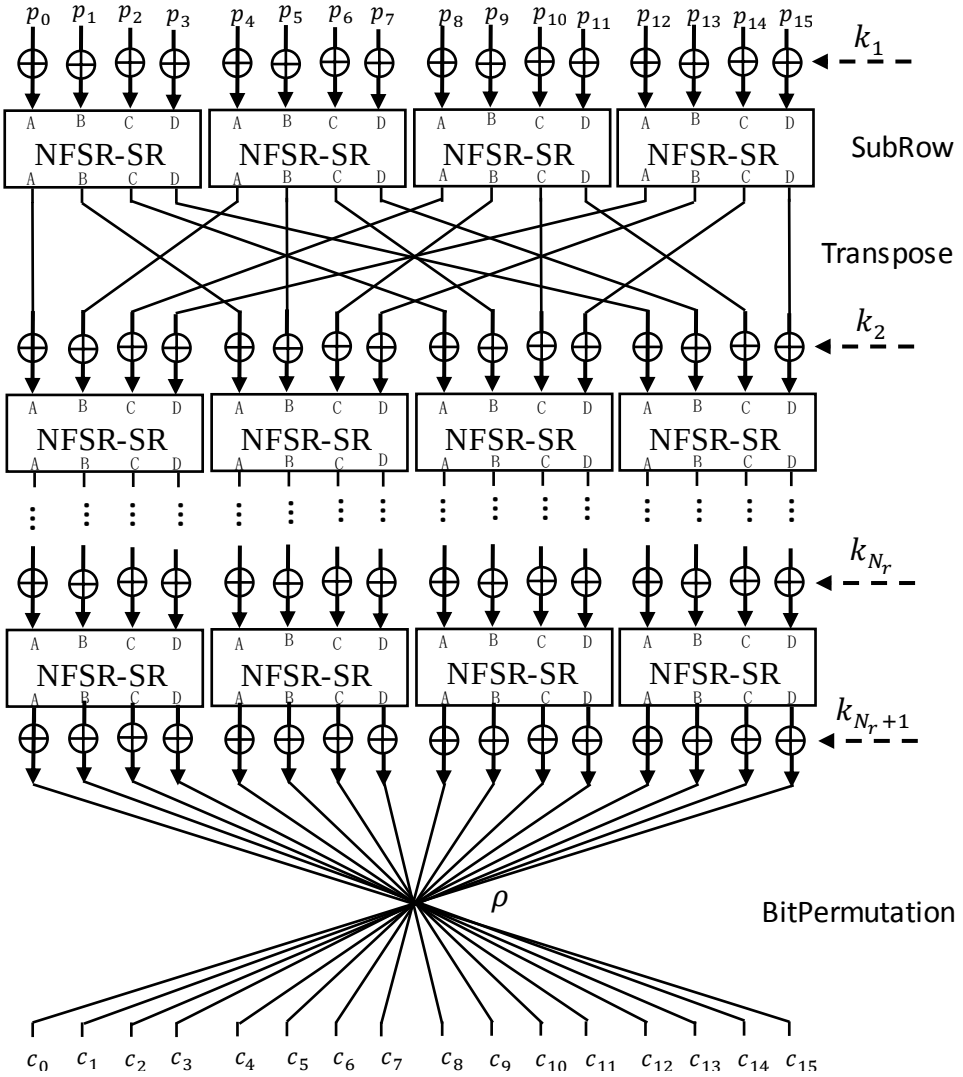


图 2　SPRING-128-128 加密算法整体示意图

1.2.2 SPRING-128-128的轮函数设计

SPRING-128-128轮函数 Round 包括三个子函数：AddRoundKey，SubRow 和 Transpose。在每一轮轮变换中，首先加轮子密钥，然后对内部状态矩阵的每一行分别做 SubRow 运算，最后对内部状态进行一次转置。Round 函数的伪代码描述见图 3。

SPRING-128-128加密算法的整体结构见图 2。

```
Round(State, k_i)
{
    AddRoundKey(State, k_i);
    For i = 1 to 4 do
        SubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    Transpose(State);
}
```

图 3　SPRING-128-128 中 Round 函数的描述

```
FinalRound(tate, k_Nr, k_Nr+1)
{
    AddRoundKey(State, k_Nr);
    For i = 1 to 4 do
        SubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    AddRoundKey(State, k_Nr+1);
    BitPermutation(State);
}
```

图 4　SPRING-128-128 中 FinalRound 函数的描述

1.2.2.1 AddRoundKey 定义

AddRoundKey 完成对内部状态与轮子密钥异或加的功能。设 AddRoundKey 的输入内部状态为

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{pmatrix},$$

输入轮子密钥为

$$\begin{pmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \end{pmatrix},$$

那么 AddRoundKey 的计算规则为

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{pmatrix} \oplus \begin{pmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \end{pmatrix} = \begin{pmatrix} s'_{11} & s'_{12} & s'_{13} & s'_{14} \\ s'_{21} & s'_{22} & s'_{23} & s'_{24} \\ s'_{31} & s'_{32} & s'_{33} & s'_{34} \\ s'_{41} & s'_{42} & s'_{43} & s'_{44} \end{pmatrix}$$

其中

$$s'_{ij} = s_{ij} \oplus k_{ij}, 1 \le i,j \le 4.$$

1.2.2.2 SubRow定义

SubRow 是$\mathbb{F}_2^{32}$到$\mathbb{F}_2^{32}$的映射，对内部状态的一行做非线性变换，参见图 5，是算法中唯一的非线性运算。注意到存储一个$\mathbb{F}_2^{32}$到$\mathbb{F}_2^{32}$的 S-盒需要较大的内存空间，SubRow 实际是由一个32比特的 Galois NFSR 来实现，称为 NFSR-SR。NFSR-SR 是由四个8-bit 寄存器 A、B、C、D 互相反馈的环状串联结构，具体设计参数请参见第 1.2.2.3 节。

设 SubRow 运算的输入为$s_{i1}, s_{i2}, s_{i3}, s_{i4}$，即内部状态的第$i$行，$1 \le i \le 4$，则用$s_{i1}, s_{i2}, s_{i3}, s_{i4}$分别填充寄存器 A、B、C、D，然后 NFSR-SR 更新32拍，输出内部状态，参见图 6。

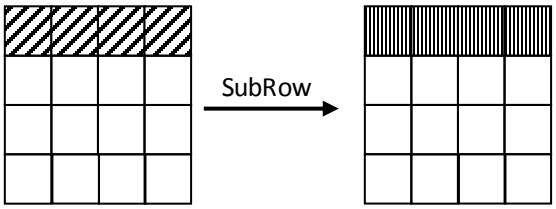


图 5 SubRow 示意图

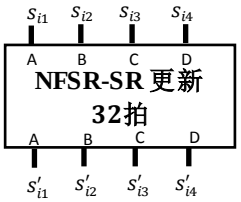


图 6 SubRow($s_{i1}, s_{i2}, s_{i3}, s_{i4}$) 示意图

1.2.2.3 NFSR-SR 设计参数

NFSR-SR 是四个8-bit 寄存器 A、B、C、D 互相反馈的环状串联结构，如图 7 所示，其中四个寄存器 A、B、C、D 的反馈函数分别为

$$f_A = x_6 \oplus x_4x_3 \oplus x_0,$$

$$f_B = x_5 \oplus x_4x_3 \oplus x_0,$$

$$f_C = x_4x_5 \oplus x_3 \oplus x_0,$$

$$f_D = x_4x_5 \oplus x_2 \oplus x_0.$$

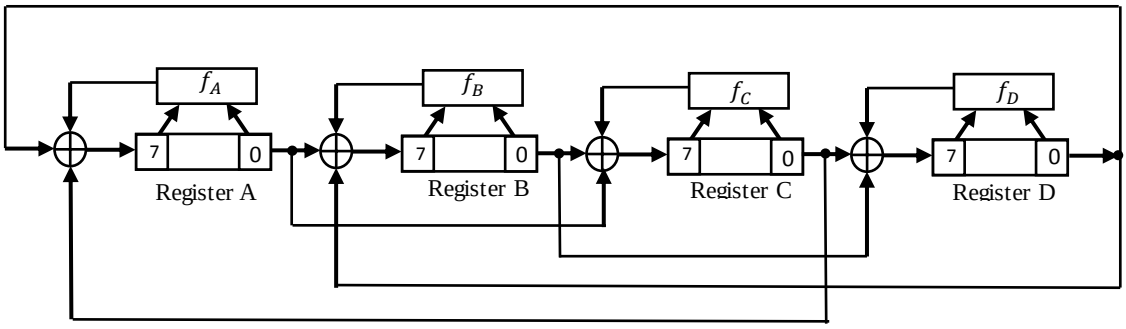


图 7　SubRow 的 NFSR 结构

为了描述的清晰性，下面给出32比特 NFSR-SR 的状态转移函数。设t时刻 NFSR-SR 的内部状态为

$$S(t) = \big(A(t), B(t), C(t), D(t)\big)$$

$$= \big(a_7(t), \dots, a_0(t); b_7(t), \dots, b_0(t); c_7(t), \dots, c_0(t); d_7(t), \dots, d_0(t)\big),$$

则第$t+1$时刻的状态为

$$a_7(t+1) = a_6(t) \oplus a_4(t)a_3(t) \oplus a_0(t) \oplus c_0(t) \oplus d_0(t)$$

$$a_i(t+1) = a_{i+1}(t), i = 6,5, \dots,0$$

$$b_7(t+1) = b_5(t) \oplus b_4(t)b_3(t) \oplus b_0(t) \oplus a_0(t) \oplus d_0(t)$$

$$b_i(t+1) = b_{i+1}(t), i = 6,5, \dots,0$$

$$c_7(t+1) = c_4(t)c_5(t) \oplus c_3(t) \oplus c_0(t) \oplus b_0(t) \oplus a_0(t)$$

$$c_i(t+1) = c_{i+1}(t), i = 6,5, \dots,0$$

$$d_7(t+1) = d_4(t)d_5(t) \oplus d_2(t) \oplus d_0(t) \oplus c_0(t) \oplus b_0(t)$$

$$d_i(t+1) = d_{i+1}(t), i = 6,5, \dots,0$$

可以证明 NFSR-SR 的内部状态更新是可逆的。

性质 1 非线性反馈移位寄存器 NFSR-SR 的内部状态更新函数是可逆的。

证明 这里我们直接给出状态更新函数的逆映射。设已知第$t+1$时刻内部状态$S(t+1)=\big(A(t+1),B(t+1),C(t+1),D(t+1)\big)$，并记

$$\alpha = a_7(t+1) \oplus a_5(t+1) \oplus a_3(t+1)a_2(t+1),$$
$$\beta = b_7(t+1) \oplus b_4(t+1) \oplus b_3(t+1)b_2(t+1),$$
$$\gamma = c_7(t+1) \oplus c_3(t+1)c_4(t+1) \oplus c_2(t+1),$$
$$\theta = d_7(t+1) \oplus d_3(t+1)d_4(t+1) \oplus d_1(t+1).$$

下面给出第t时刻内部状态$S(t)$的计算公式：

$$a_0(t) = \alpha \oplus \beta \oplus \gamma$$
$$a_i(t) = a_{i-1}(t+1), i = 1,2,\dots,7$$
$$b_0(t) = \beta \oplus \gamma \oplus \theta$$
$$b_i(t) = b_{i-1}(t+1), i = 1,2,\dots,7$$
$$c_0(t) = \alpha \oplus \gamma \oplus \theta$$
$$c_i(t) = c_{i-1}(t+1), i = 1,2,\dots,7$$
$$d_0(t) = \alpha \oplus \beta \oplus \theta$$
$$d_i(t) = d_{i-1}(t+1), i = 1,2,\dots,7.$$

从而结论成立。

由 NFSR-SR 更新函数的可逆性，易见 SubRow 运算是可逆的。

性质 2 SubRow 是$\mathbb{F}_2^{32}$到$\mathbb{F}_2^{32}$的一一映射。

1.2.2.4 Transpose 定义

Transpose 对内部状态矩阵进行一次转置，参见图 8。

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{pmatrix} \xrightarrow{\text{Transpose}} \begin{pmatrix} s_{11} & s_{21} & s_{31} & s_{41} \\ s_{12} & s_{22} & s_{32} & s_{42} \\ s_{13} & s_{23} & s_{33} & s_{43} \\ s_{14} & s_{24} & s_{34} & s_{44} \end{pmatrix}$$

图 8 Transpose 示意图

1.2.2.5 BitPermutation 定义

最后一轮中的 BitPermutation 对内部状态的比特位置进行置换。设$(a_7a_6a_5a_4a_3a_2a_1a_0)$是一个字节的8比特，其中$a_7$是高位比特，记

$$\rho(a_7a_6a_5a_4a_3a_2a_1a_0) = (a_0a_1a_2a_3a_4a_5a_6a_7),$$

即ρ表示对字节内部比特的一次旋转。BitPermutation 对内部状态的作用参见图 9。

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{pmatrix} \xrightarrow{\text{Bit-Permutation}} \begin{pmatrix} \rho(s_{44}) & \rho(s_{43}) & \rho(s_{42}) & \rho(s_{41}) \\ \rho(s_{34}) & \rho(s_{33}) & \rho(s_{32}) & \rho(s_{31}) \\ \rho(s_{24}) & \rho(s_{23}) & \rho(s_{22}) & \rho(s_{21}) \\ \rho(s_{14}) & \rho(s_{13}) & \rho(s_{12}) & \rho(s_{11}) \end{pmatrix}$$

图 9 Bit-Permutation 示意图

1.2.3 SPRING-128-128加密算法中 KeyExpansion 的设计

本节介绍 SPRING-128-128加密算法中的密钥扩展算法。设私钥为$Key = (K_1, K_2, \dots, K_{16})$，其中$K_i$是一个8比特字节。

轮子密钥由一个128比特的 Galois NFSR 生成，称为 NFSR-KeyGeneration。该 NFSR-KeyGeneration 可以看成是8个16-bit 寄存器的环状串联结构，如图 10 所示，其中8个寄存器的反馈函数分别为

$$f_1 = x_1 \oplus x_3 \oplus x_8x_9$$

$$f_2 = x_1 \oplus x_4 \oplus x_8x_9$$

$$f_3 = x_1 \oplus x_5 \oplus x_8x_9$$

$$f_4 = x_1 \oplus x_6 \oplus x_8x_9$$

$$f_5 = x_{15} \oplus x_7x_8 \oplus x_{10}$$

$$f_6 = x_{15} \oplus x_7x_8 \oplus x_{11}$$

$$f_7 = x_{15} \oplus x_7x_8 \oplus x_{12}$$

$$f_8 = x_{15} \oplus x_7x_8 \oplus x_{13}.$$

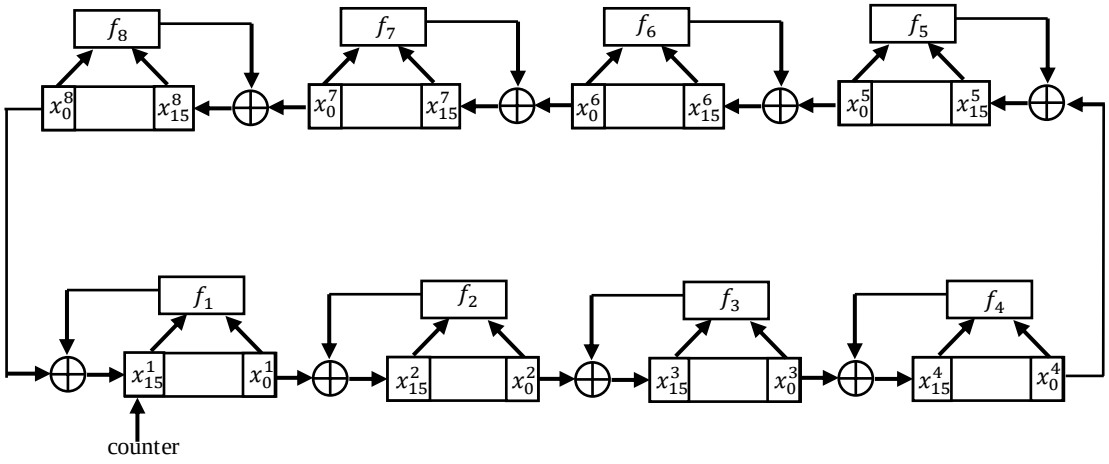


图 10　SPRING-128-128 的密钥扩展算法中的 NFSR-KeyGeneration 示意图

设图 10 中 NFSR 的内部状态为$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8)$，其中$(IS_1^i, IS_2^i)$是寄存器$i$的内部状态，并且$(IS_1^i, IS_2^i) = (x_{15}^i, x_{14}^i, \ldots, x_0^i)$。具体的轮子密钥生成算法参见图 11，其中 counter 取值参见表 2。常值 counter 可以由一个8比特的 LFSR 生成，特征多项式为$f(y) = y^8 + y^4 + y^3 + y^2 + 1$，每隔8拍取内部状态为 counter 值。

KEYEXPANSION(Key)
{
寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8) = (K_1, K_2, \ldots, K_{16})$
$k_1 = IS$
for i from 2 to $N_r + 1$ do
1. $IS_2^1 \leftarrow IS_2^1 \oplus \text{counter}[i-1]$　$\left(\text{注：counter}[i-1] = (y_7, y_6, \ldots, y_0), \text{IS}_1^1 \oplus \text{counter}[i] = (x_{15}^1 \oplus y_7, x_{14}^1 \oplus y_6, \ldots, x_8^1 \oplus y_0)\right)$
2. NFSR-KeyGeneration 运行16拍
3. $k_i = IS$

End for
Return $(k_1, k_2, \ldots, k_{N_r+1})$
}

图 11　SPRING-128-128 的密钥扩展算法

counter[1]	(1,1,0,0,0,0,0,0)
counter[2]	(0,0,1,0,0,1,0,0)
counter[3]	(0,0,1,1,1,0,1,1)

counter[4]	(0,1,0,0,0,0,0,1)
counter[5]	(0,1,1,0,1,1,0,1)
counter[6]	(0,1,0,0,1,1,0,1)
counter[7]	(1,1,0,0,0,0,1,1)
counter[8]	(1,0,1,1,0,1,1,1)
counter[9]	(1,1,0,1,0,1,1,1)
counter[10]	(0,1,0,0,0,1,0,1)

表 2　SPRING-128-128中 counter 取值列表

1.2.4 SPRING-128-128解密算法的设计

1.2.4.1 解密算法中轮函数的设计

SPRING-128-128解密算法的轮函数 InvRound 和 InvFinalRound 的结构与加密算法轮函数是完全一致的，伪代码描述参见图 12。

```
InvRound(State, k_i^inv)
{
    AddRoundKey(State, k_i^inv);
    For i = 1 to 4 do
        InvSubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    Transpose(State);
}
```

```
InvFinalRound(State, k_Nr^inv, k_Nr+1^inv)
{
    AddRoundKey(State, k_Nr^inv);
    For i = 1 to 4 do
        InvSubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    AddRoundKey(State, k_Nr+1^inv);
    Bit-Permutation(State);
}
```

图 12　InvRound 函数的描述

1.2.4.2 解密算法中 InvSubRow 的非线性反馈移位寄存器结构

由性质 2 我们知道 SubRow 运算是可逆的，并且我们给出了 NFSR-SR 状态更新函数的逆函数。记实现 InvSubRow 运算的 Galois 非线性反馈为 NFSR-SR-Inv，参见图 13。NFSR-SR-Inv 仍然是四个8比特寄存器 A、B、C、D 的环状串联，四个寄存器的反馈函数与 SubRow 中是一样的。事实上 NFSR-SR-Inv 与

NFSR-SR 几乎可以用同一个电路来实现，仅需额外增加一个选择器。我们在图 14 中同时给出了 NFSR-SR 与 NFSR-SR-Inv 的工作示意图，其中 NFSR-SR 和 NFSR-SR-Inv 分别选择红色或绿色拉线，其余线路均是共用的。

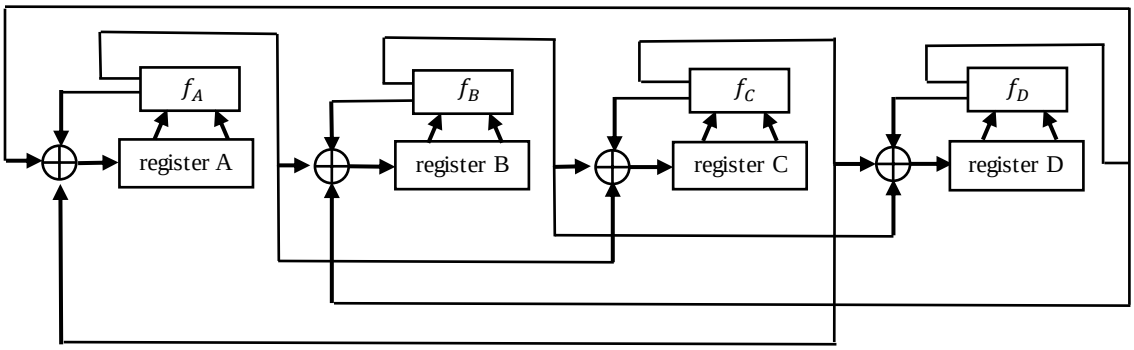
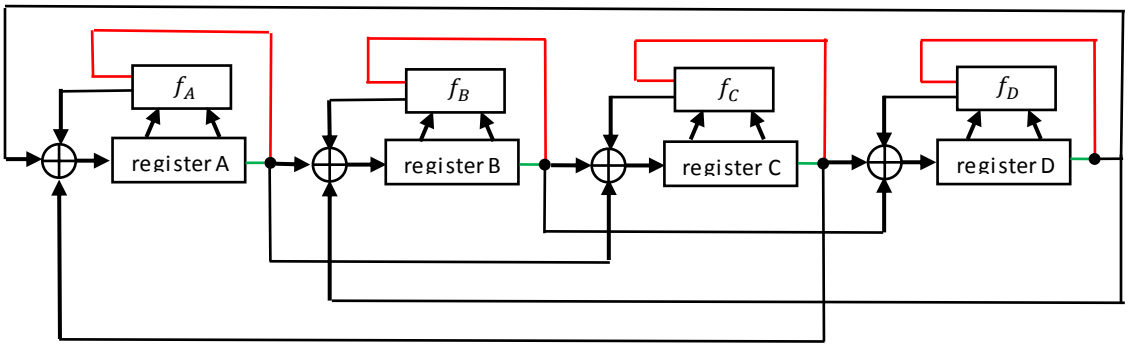


图 13　NFSR-SR-Inv 的结构图



注: NFSR-SR 选择绿色拉线，NFSR-SR-Inv 选择红色拉线。

图 14　NFSR-SR 与 NFSR-SR-Inv 统一结构图

InvSubRow 的设计具有有以下性质。

性质 3　设算法总轮数为 r，若加密算法第i轮SubRow运算后的内部状态为 S，则对应密文经过解密算法第 $n-i$ 轮 InvSubRow 运算后的内部状态为 BitPermutation(S)。

1.2.4.3 解密算法中的 InvKeyExpansion 的设计

若在加密时，存储了所有的轮子密钥$(k_1, k_2, \ldots, k_{N_r+1})$，则在解密时对每个子密钥做一个简单的线性变换即可得到解密轮子密钥，参见图 15。关于 InvKeyExpansion 的正确性，有以下两点说明：

（1）根据性质 3，在 InvKeyExpansion 中我们对所有轮子密钥进行了

BitPermutation运算。

（2）为了保持解密算法与加密算法结构的一致性，我们在解密算法中实际上交换了轮密钥加和 Transpose 的运算顺序，所以在 InvKeyExpansion 中我们对部分轮子密钥进行了 Transpose 运算。

InvKeyExpansion$(k_1, k_2, \dots, k_{N_r+1})$
{
    $k_1^{inv} = \text{BitPermutation}(k_{N_r+1})$;
    for $i = 2$ **to** N_r **do**
        $k_i^{inv} = \text{Transpose}(\text{BitPermutation}(k_{N_r-i+2}))$;
    end for
    $k_{N_r+1}^{inv} = \text{BitPermutation}(k_1)$;
    return $(k_1^{inv}, k_2^{inv}, \dots, k_{N_r+1}^{inv})$
}

图 15　已知加密轮子密钥条件下的 InvKeyExpansion 算法

若不存储所有轮子密钥，也可以通过最后一轮轮子密钥重新生成解密轮子密钥。设最后一轮轮子密钥为$FinalKey = (FK_1, FK_2, \dots, FK_{16})$，其中$FK_i$是一个8比特的字节。

解密轮子密钥也是由一个128比特的 Galois NFSR 生成，称为 NFSR-KeyGeneration-Inv，如图 16 所示。比较图 10 与图 16 易见，NFSR-KeyGeneration-Inv 与 NFSR-KeyGeneration 仅有以下两点不同：

（1） counter 值加入的位置不同；

（2） 反馈函数输出的拉线方向不同。

除此之外的其它运算都是一样的，包括8个寄存器的反馈函数$f_1, f_2, \dots, f_8$。因此，在硬件设计时，同时实现 NFSR-KeyGeneration-Inv 与 NFSR-KeyGeneration 的功能几乎不需要额外的资源消耗。

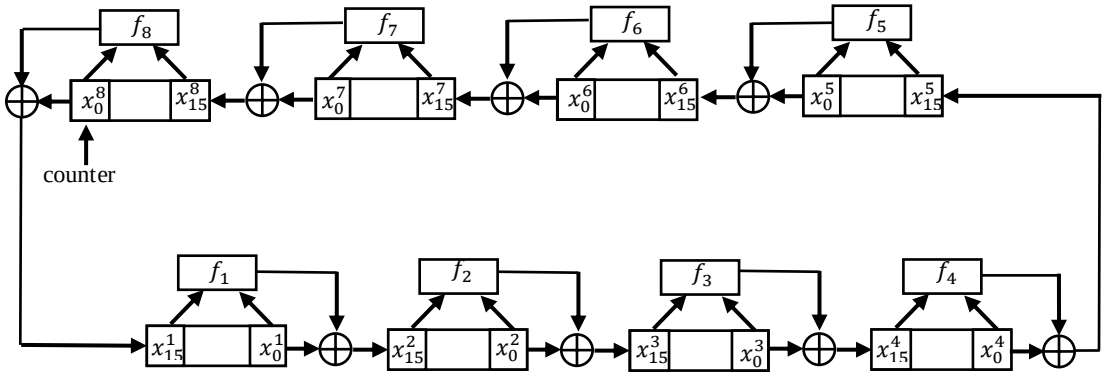


图 16　NFSR-KeyGeneration-Inv 示意图

设 NFSR-KeyGeneration-Inv 的内部状态为$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \dots, IS_1^8, IS_2^8)$，其中$(IS_1^i, IS_2^i)$是寄存器$i$的内部状态，并且$(IS_1^i, IS_2^i) = (x_{15}^i, x_{14}^i, \dots, x_0^i)$。解密轮子密钥生成算法参见图 17。

INVKEYEXPANSION($FinalKey$)

{

寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \dots, IS_1^8, IS_2^8) = (\rho(FK_{16}), \rho(FK_{15}), \dots, \rho(FK_1))$

$k_1^{inv} = IS$

for i **from** 2 **to** N_r **do**

1. NFSR-KeyGeneration-Inv 运行16拍
2. $IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[i])$
3. $k_i^{inv} = \text{Transpose}(IS)$

end for

NFSR-KeyGeneration-Inv 运行16拍

$IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[1])$

$k_{N_r+1}^{inv} = IS$

return $(k_1^{inv}, k_2^{inv}, \dots, k_{N_r+1}^{inv})$

}

图 17　基于 NFSR-KeyGeneration-Inv 的解密密钥扩展算法

1.3 SPRING-128-256的设计

SPRING-128-256与 SPRING-128-128的轮函数是一样的，不同之处仅在加密轮数和密钥扩展算法，加密轮数请参见表 1。以下我们对 SPRING-128-256的密钥扩展算法进行详细描述。

1.3.1 SPRING-128-256加密算法中 KeyExpansion 的设计

设私钥为$Key = (K_1, K_2, \dots, K_8)$，其中$K_i$是32比特字，$1 \le i \le 8$。SPRING-128-256密钥扩展算法的设计与 SPRING-128-128类似。轮子密钥由一个256比特的 Galois NFSR 生成，称为 NFSR-KeyGeneration-256。该 NFSR-KeyGeneration-256可以看成是8个32-bit 寄存器的环状串联结构，如图 18 所示，其中8个寄存器的反馈函数分别为

$$g_1 = x_1 \oplus x_5 \oplus x_{15}x_{16}$$
$$g_2 = x_1 \oplus x_6 \oplus x_{15}x_{16}$$
$$g_3 = x_1 \oplus x_7 \oplus x_{15}x_{16}$$
$$g_4 = x_1 \oplus x_8 \oplus x_{15}x_{16}$$
$$g_5 = x_{31} \oplus x_{16}x_{17} \oplus x_{24}$$
$$g_6 = x_{31} \oplus x_{16}x_{17} \oplus x_{25}$$
$$g_7 = x_{31} \oplus x_{16}x_{17} \oplus x_{26}$$
$$g_8 = x_{31} \oplus x_{16}x_{17} \oplus x_{27}.$$

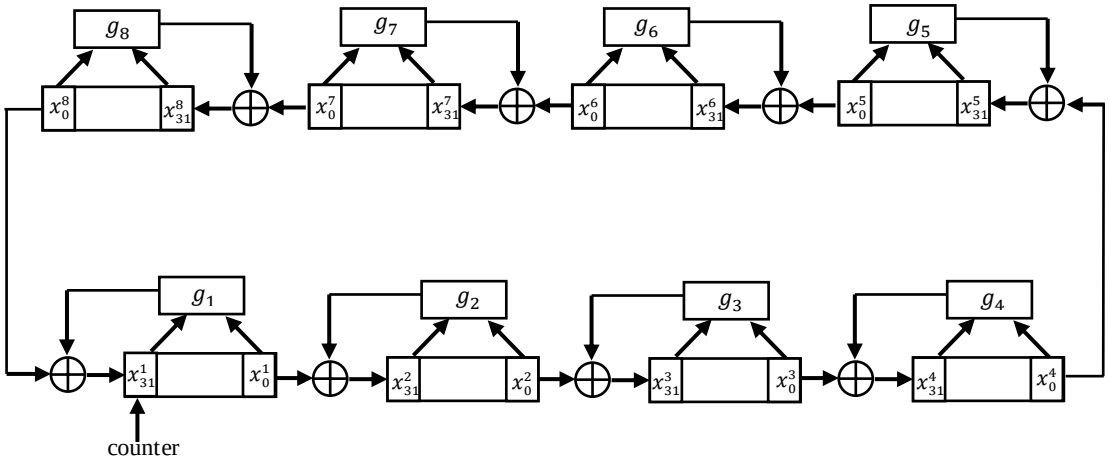


图 18　NFSR-KeyGeneration-256 示意图

设图 18 中 NFSR 的内部状态为$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \dots, IS_1^8, IS_2^8)$，其中$(IS_1^i, IS_2^i)$是寄存器$i$的内部状态，并且$(IS_1^i, IS_2^i) = (x_{15}^i, x_{14}^i, \dots, x_0^i)$。具体的轮子密钥生成算法参见图 19。常值 counter 可以由一个8比特的 LFSR 生成，特征多项式为$f(y) = y^8 + y^4 + y^3 + y^2 + 1$，每隔8拍取内部状态为 counter 值，具体取值

参见表 3。

KEYEXPANSION(Key, n)
{
寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8) = (K_1, K_2, \ldots, K_8)$
$k_1 = (IS_1^1, IS_2^1, IS_1^3, IS_2^3, IS_1^5, IS_2^5, IS_1^7, IS_2^7)$
for i **from** 2 **to** $N_r + 1$ **do**
4. $IS_1^1 \leftarrow IS_1^1 \oplus \text{counter}[i-1]$
5. NFSR-KeyGeneration-256 运行32拍
6. $k_i = (IS_1^1, IS_2^1, IS_1^3, IS_2^3, IS_1^5, IS_2^5, IS_1^7, IS_2^7)$
end for
return $(k_1, k_2, \ldots, k_{N_r+1})$
}

图 19　SPRING-128-256 的密钥扩展算法

表 3　counter 取值列表

counter[1]	(1,1,0,0,0,0,0,0)
counter[2]	(0,0,1,0,0,1,0,0)
counter[3]	(0,0,1,1,1,0,1,1)
counter[4]	(0,1,0,0,0,0,0,1)
counter[5]	(0,1,1,0,1,1,0,1)
counter[6]	(0,1,0,0,1,1,0,1)
counter[7]	(1,1,0,0,0,0,1,1)
counter[8]	(1,0,1,1,0,1,1,1)
counter[9]	(1,1,0,1,0,1,1,1)
counter[10]	(0,1,0,0,0,1,0,1)
counter[11]	(1,1,0,1,1,0,0,0)
counter[12]	(0,1,1,1,1,0,0,0)
counter[13]	(1,1,0,0,1,1,1,0)
counter[14]	(0,1,1,0,1,0,0,0)
counter[15]	(1,0,0,0,1,0,0,1)
counter[16]	(0,1,0,1,0,0,1,0)
counter[17]	(1,0,1,1,1,0,0,1)

counter[18]	(1,0,0,1,1,0,1,1)

1.3.2 SPRING-128-256解密算法中 InvKeyExpansion 的设计

若在加密时，存储了所有的轮子密钥$(k_1, k_2, \ldots, k_{N_r+1})$，则 SPRING-128-256 算法的解密密钥仅需对每个子密钥做一个简单的线性变换即可得到，该算法与 SPRING-128-128中的解密密钥生成算法相同，具体参见图 15。其正确性可以参见第 1.2.4.3 节。

若不存储所有轮子密钥，也可以通过最后一轮轮子密钥重新生成解密轮子密钥。设最后一轮轮子密钥为$FinalKey = (FK_1, FK_2, \ldots, FK_8)$，其中$FK_i$是一个32比特的字。

解密轮子密钥也是由一个256比特的 Galois NFSR 生成，称为 NFSR-KeyGeneration-Inv-256，如图 20 所示，其中8个寄存器的反馈函数$g_1, g_2, \ldots, g_8$与 NFSR-KeyGeneration-256中是一样的。易见，NFSR-KeyGeneration-Inv-256与 NFSR-KeyGeneration-256的电路几乎是一样的。

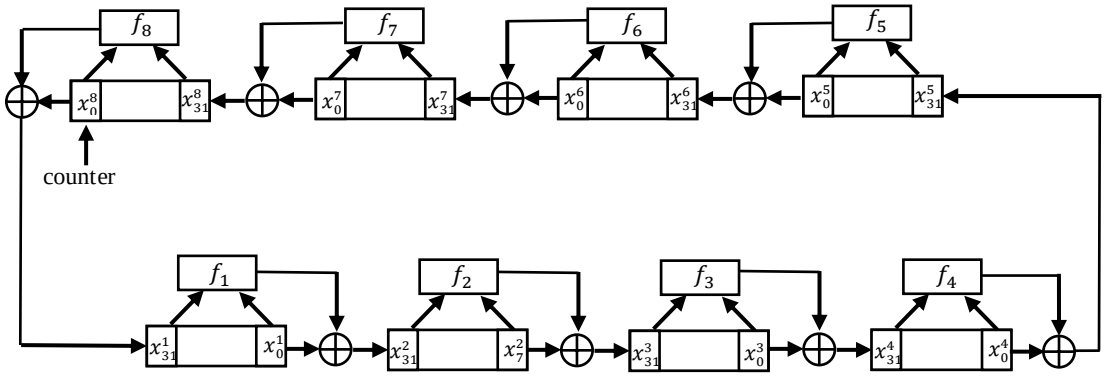


图 20　NFSR-KeyGeneration- Inv-256 示意图

设 NFSR-KeyGeneration-Inv-256 的内部状态为$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8)$，其中$(IS_1^i, IS_2^i)$是寄存器$i$的内部状态，并且$(IS_1^i, IS_2^i) = (x_{31}^i, x_{30}^i, \ldots, x_0^i)$。解密轮子密钥生成算法参见图 21。

INVKEYEXPANSION($FinalKey, n$)

{

寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8) = (\rho(FK_8), \rho(FK_7), \ldots, \rho(FK_1))$

$k_1^{inv} = (IS_1^2, IS_2^2, IS_1^4, IS_2^4, IS_1^6, IS_2^6, IS_1^8, IS_2^8)$

for i **from** 2 **to** N_r **do**

4. NFSR-KeyGeneration-Inv-32 运行32拍
5. $IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[i])$
6. $k_i^{inv} = \text{Transpose}((IS_1^2, IS_2^2, IS_1^4, IS_2^4, IS_1^6, IS_2^6, IS_1^8, IS_2^8))$

end for

NFSR-KeyGeneration-Inv-32 运行32拍

$IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[1])$

$k_{N_r+1}^{inv} = (IS_1^2, IS_2^2, IS_1^4, IS_2^4, IS_1^6, IS_2^6, IS_1^8, IS_2^8)$

return $(k_1^{inv}, k_2^{inv}, \ldots, k_{N_r+1}^{inv})$

}

图 21 分组长度为128时基于 NFSR-KeyGeneration-Inv-256 的解密密钥扩展算法

1.4 SPRING-256-256的设计

本节介绍 SPRING-256-256的设计参数。

1.4.1 SPRING-256-256的整体结构

SPRING-256-256的内部状态以字节为单位，看作$\mathbb{F}_{2^8}$上的32维向量或者$\mathbb{F}_{2^8}$上8×4的矩阵，即

$$\begin{aligned} State &= s_0 \| s_1 \| s_2 \| s_3 \| \cdots \| s_{30} \| s_{31} \\ &= (s_0, s_1, s_2, s_3, \ldots, s_{30}, s_{31}) \\ &= \begin{pmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \\ s_{16} & s_{17} & s_{18} & s_{19} \\ s_{20} & s_{21} & s_{22} & s_{23} \\ s_{24} & s_{25} & s_{26} & s_{27} \\ s_{28} & s_{29} & s_{30} & s_{31} \end{pmatrix}, \end{aligned}$$

其中s_{ij}为一个字节长度，即8比特。设256比特的明文分组为$p_0 \| p_1 \| \cdots \| p_{31}$，

密文分组为$c_0 \| c_1 \| \cdots \| c_{31}$。

SPRING-256-256加密算法的整体结构见图 22，而解密算法的结构与加密算法是完全一致的，即图 22 中的输入为密文和解密密钥时，输出为明文。

1.4.2 SPRING-256-256轮函数设计

SPRING-256-256轮函数 Round 包括三个子函数：AddRoundKey，SubRow 和 ShiftColumns。在每一轮轮变换中，首先加轮子密钥，然后对内部状态每一行分别做 SubRow 运算，最后对内部状态矩阵的列做循环移位。Round 函数的伪代码描述见图 23。

SPRING-256-256 最后一轮轮函数 FinalRound 包括两个子函数：AddRoundKey，Bit-Permutation。首先加轮子密钥，然后对内部状态的比特进行位置的置换。FinalRound 函数的伪代码描述见图 24。

SPRING-256-256算法中针对32比特状态所做的变换 SubRow 和 SPRING-128中的相同。与 SPRING-128不同的是，SPRING-256-256的 Transpose 变换用的是针对256比特状态设计的 Transpose-256，该置换同样保证了算法加解密的相似性。

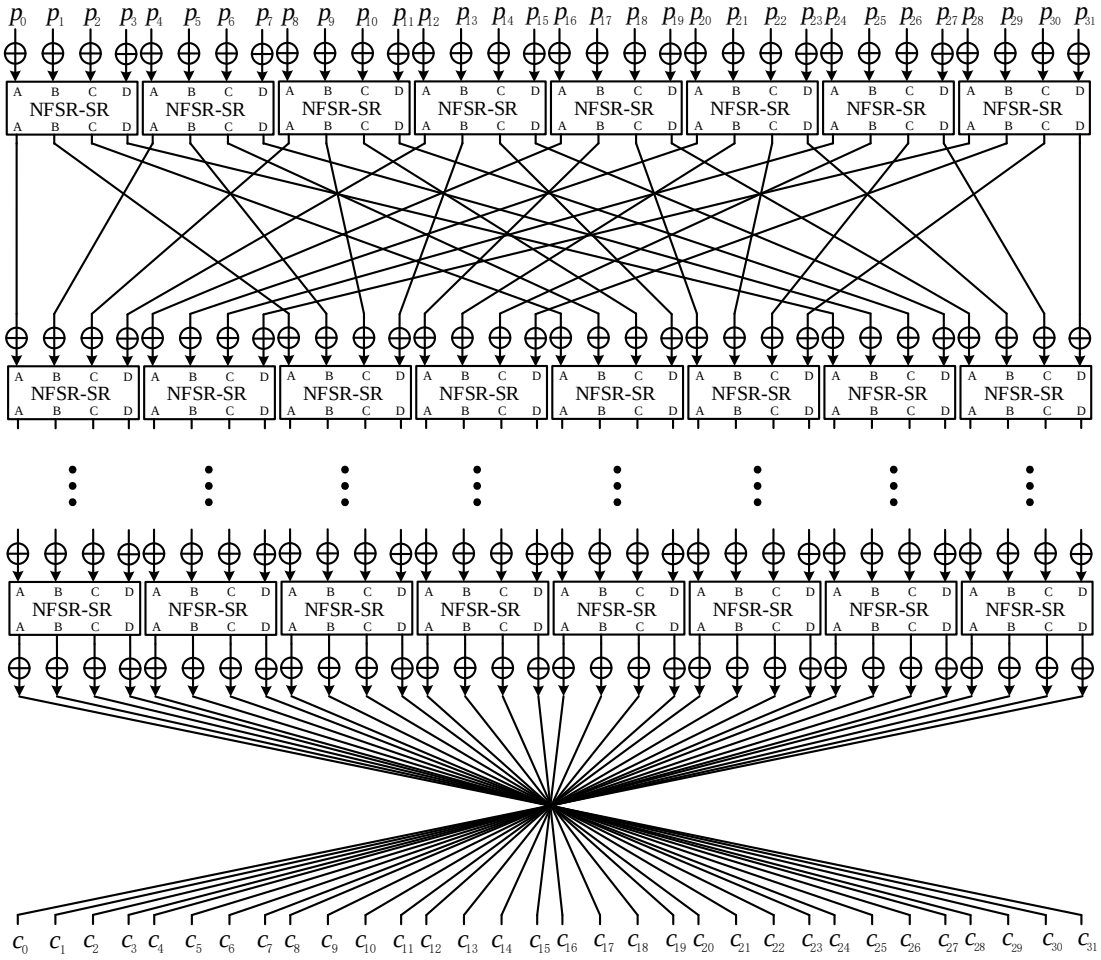


图 22　SPRING-256-256加密算法整体示意图

```
Round(State, k_i)
{
   AddRoundKey(State, k_i);
   for i from 1 to 8 do
      SubRow(s_i1, s_i2, s_i3, s_i4);
   end for
   Transpose-256 (State);
}
```

图 23　SPRING-256-256中 Round 函数的描述

```
FinalRound(State, k_{N_r}, k_{N_r+1})
{
    AddRoundKey(State, k_{N_r});
    for i from 1 to 8 do
        SubRow(s_{i1}, s_{i2}, s_{i3}, s_{i4});
    end for
    AddRoundKey(State, k_{N_r+1});
    BitPermutation(State);
}
```

图 24　SPRING-256-256中 FinalRound 函数的描述

1.4.2.1 AddRoundKey 定义

设 AddRoundKey 的输入内部状态为

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \\ s_{51} & s_{52} & s_{53} & s_{54} \\ s_{61} & s_{62} & s_{63} & s_{64} \\ s_{71} & s_{72} & s_{73} & s_{74} \\ s_{81} & s_{82} & s_{83} & s_{84} \end{pmatrix},$$

输入轮子密钥为

$$\begin{pmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \\ k_{51} & k_{52} & k_{53} & k_{54} \\ k_{61} & k_{62} & k_{63} & k_{64} \\ k_{71} & k_{72} & k_{73} & k_{74} \\ k_{81} & k_{82} & k_{83} & k_{84} \end{pmatrix},$$

那么 AddRoundKey 的计算规则为

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \\ s_{51} & s_{52} & s_{53} & s_{54} \\ s_{61} & s_{62} & s_{63} & s_{64} \\ s_{71} & s_{72} & s_{73} & s_{74} \\ s_{81} & s_{82} & s_{83} & s_{84} \end{pmatrix} \oplus \begin{pmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \\ k_{51} & k_{52} & k_{53} & k_{54} \\ k_{61} & k_{62} & k_{63} & k_{64} \\ k_{71} & k_{72} & k_{73} & k_{74} \\ k_{81} & k_{82} & k_{83} & k_{84} \end{pmatrix} = \begin{pmatrix} s'_{11} & s'_{12} & s'_{13} & s'_{14} \\ s'_{21} & s'_{22} & s'_{23} & s'_{24} \\ s'_{31} & s'_{32} & s'_{33} & s'_{34} \\ s'_{41} & s'_{42} & s'_{43} & s'_{44} \\ s'_{51} & s'_{52} & s'_{53} & s'_{54} \\ s'_{61} & s'_{62} & s'_{63} & s'_{64} \\ s'_{71} & s'_{72} & s'_{73} & s'_{74} \\ s'_{81} & s'_{82} & s'_{83} & s'_{84} \end{pmatrix}$$

其中

$$s'_{ij} = s_{ij} \oplus k_{ij}, 1 \le j \le 4, 1 \le j \le 8.$$

1.4.2.2 Transpose-256 定义

Transpose-256对内部状态矩阵进行如下置换：

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \\ s_{51} & s_{52} & s_{53} & s_{54} \\ s_{61} & s_{62} & s_{63} & s_{64} \\ s_{71} & s_{72} & s_{73} & s_{74} \\ s_{81} & s_{82} & s_{83} & s_{84} \end{pmatrix} \xrightarrow{\text{Transpose}} \begin{pmatrix} s_{11} & s_{21} & s_{31} & s_{41} \\ s_{51} & s_{61} & s_{71} & s_{81} \\ s_{12} & s_{22} & s_{32} & s_{42} \\ s_{52} & s_{62} & s_{72} & s_{82} \\ s_{13} & s_{23} & s_{33} & s_{43} \\ s_{53} & s_{63} & s_{73} & s_{83} \\ s_{14} & s_{24} & s_{34} & s_{44} \\ s_{54} & s_{64} & s_{74} & s_{84} \end{pmatrix}.$$

图 25　Transpose-256 示意图

Transpose-256的逆变换 Transpose-256-Inv 定义如下：

$$\begin{pmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \\ s_{51} & s_{52} & s_{53} & s_{54} \\ s_{61} & s_{62} & s_{63} & s_{64} \\ s_{71} & s_{72} & s_{73} & s_{74} \\ s_{81} & s_{82} & s_{83} & s_{84} \end{pmatrix} \xrightarrow{\text{Transpose}-256-\text{Inv}} \begin{pmatrix} s_{11} & s_{31} & s_{51} & s_{71} \\ s_{12} & s_{32} & s_{52} & s_{72} \\ s_{13} & s_{33} & s_{53} & s_{73} \\ s_{14} & s_{34} & s_{54} & s_{74} \\ s_{21} & s_{41} & s_{61} & s_{81} \\ s_{22} & s_{42} & s_{62} & s_{82} \\ s_{23} & s_{43} & s_{63} & s_{83} \\ s_{24} & s_{44} & s_{64} & s_{84} \end{pmatrix}$$

图 26　Transpose-256 示意图

1.4.2.3 BitPermutation-256定义

最后一轮中的 BitPermutation 对内部状态的比特位置进行置换，参见图　27。

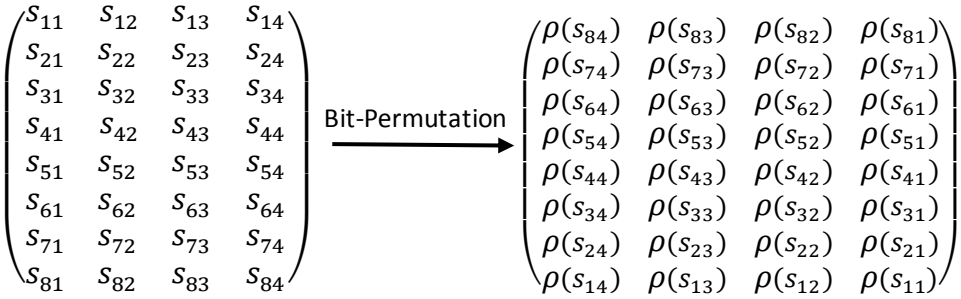


图 27　Bit-Permutation 示意图

1.4.3 SPRING-256-256加密算法中 KeyExpansion 的设计

SPRING-256-256的密钥扩展算法 KeyExpansion 与 SPRING-128-256的基本相同，具体参见图 28。需要注意的是， SPRING-256-256取的是寄存器的全状态作为轮子密钥。

KEYEXPANSION(Key, n)
{
　寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8) = (K_1, K_2, \ldots, K_8)$
　$k_1 = IS$
　for i **from** 2 **to** $N_r + 1$ **do**
　7. $IS_1^1 \leftarrow IS_1^1 \oplus \text{counter}[i-1]$
　8. NFSR-KeyGeneration-256 运行32拍
　9. $k_i = IS$
　end for
　return $(k_1, k_2, \ldots, k_{N_r+1})$
}

图 28　SPRING-256-256的密钥扩展算法

1.4.4 SPRING-256-256解密算法的设计

1.4.3.1　SPRING-256-256解密算法轮函数的设计

SPRING-256-256解密算法的轮函数 InvRound 和 InvFinalRound 的结构与加密算法轮函数结构同样是完全一致的，非线性变换使用的是 InvSubRow，线性变换用的是 Transpose-Inv-256其伪代码描述参见图 29。

```
InvRound(State, k_i^inv)
{
    AddRoundKey(State, k_i^inv);
    For i = 1 to 8 do
        InvSubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    Transpose-Inv-256(State);
}
```

```
InvFinalRound(State, k_Nr^inv, k_Nr+1^inv)
{
    AddRoundKey(State, k_Nr^inv);
    For i = 1 to 8 do
        InvSubRow(s_i1, s_i2, s_i3, s_i4);
    End for
    AddRoundKey(State, k_Nr+1^inv);
    Bit-Permutation(State);
}
```

图 29　SPRING-256-256中 InvRound 函数的描述

1.4.3.2　SPRING-256-256解密算法中 InvKeyExpansion 的设计

SPRING-256-256解密算法的密钥扩展算法InvKeyExpansion与SPRING-128-256的基本相同，具体参见图 30。其中，$(k_1, k_2, \dots, k_{N_r+1})$为加密过程中的轮子密钥。需要注意的是， 图 30 中涉及的Transpose函数为 Transpose-256。

```
InvKeyExpansion(k_1, k_2, ..., k_Nr+1)
{
    k_1^inv = BitPermutation(k_Nr+1);
    for i = 2 to N_r do
        k_i^inv = Transpose − 256(BitPermutation(k_Nr−i+2));
    end for
    k_Nr+1^inv = BitPermutation(k_1);
    return (k_1^inv, k_2^inv, ..., k_Nr+1^inv)
}
```

图 30　SPRING-256-256算法已知加密轮子密钥条件下的 InvKeyExpansion 算法

若不存储所有轮子密钥，也可以通过最后一轮轮子密钥重新生成解密轮子密钥。设最后一轮轮子密钥为$FinalKey = (FK_1, FK_2, \dots, FK_8)$，其中$FK_i$是一个32比特的字。则解密轮子密钥可由 NFSR-KeyGeneration-Inv-32生成，具体参见图 31。

INVKEYEXPANSION($FinalKey$)

{

寄存器初态$IS = (IS_1^1, IS_2^1, IS_1^2, IS_2^2 \ldots, IS_1^8, IS_2^8) = (\rho(FK_{16}), \rho(FK_{15}), \ldots, \rho(FK_1))$

$k_1^{inv} = IS$

for i **from** 2 **to** N_r **do**

7. NFSR-KeyGeneration-Inv-32 运行32拍
8. $IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[i])$
9. $k_i^{inv} = \text{Transpose} - 256(IS)$

end for

NFSR-KeyGeneration-Inv-32 运行32拍

$IS_1^8 \leftarrow IS_1^8 \oplus \rho(\text{counter}[1])$

$k_{N_r+1}^{inv} = IS$

return $(k_1^{inv}, k_2^{inv}, \ldots, k_{N_r+1}^{inv})$

}

图 31　分组长度为256时基于 NFSR-KeyGeneration-Inv 的解密密钥扩展算法

2. 安全性分析

2.1 环状串联 NFSR 的性质

SPRING 算法中实现 S-盒功能的 NFSR-SR 和实现密钥扩张功能的 NFSR-KeyGeneration 均是环状串联 NFSR。NFSR-KeyGeneration 是典型的环状串联结构，完全符合文[1]的研究模型。NFSR-SR 是一个扩展的环状串联结构，为了扩散更快，在原文[1]的研究模型基础上，增加了四根对称的拉线，依然符合一个看似环状串联的结构。

对于环状串联结构的 NFSR，我们认为经过充分多的内部状态更新后，每个比特寄存器具有非常等价的地位，从而输出比特关于输入比特具有相似的密码学性质，并且整体性强，输入输出不容易分割成独立的小单元。文[1]中对圈结构的研究结果，可以看成是这一想法的例证。在文[1]中，作者证明了环状串联 NFSR 中每个寄存器的输出序列具有相同的周期性质。

2.2 S-盒的基本性质

在本小节中，我们对 S-盒的一些基本性质进行了分析和测试。SPRING 算法中 S-盒的功能是由 NFSR-SR 实现的。

2.2.1 代数性质

首先，我们对 S-盒的代数性质进行了测试。通过对 S-盒每一个分位函数的真值表进行 Meobius 变换，我们计算出了 S-盒每一个分位函数的代数正规型。表 4 给出了每一个分位函数的代数次数和项数。观察表 4 可以发现，S-盒各个比特分位函数的代数次数比较高并且项数稠密，可以认为 S-盒的每个输出比特具有较为均衡和理想的代数性质。

表 4 S-盒代数各分位函数的代数次数和项数

分位函数	项数	代数次数	分位函数	项数	代数次数
D-8 分位	2085284096	27	B-8 分位	876767360	23
D-7 分位	2130904448	28	B-7 分位	1378792960	25
D-6 分位	2145079168	29	B-6 分位	1876528384	26
D-5 分位	2147259008	30	B-5 分位	2095093632	27
D-4 分位	2147258240	31	B-4 分位	2140070784	28
D-3 分位	2147948672	31	B-3 分位	2147493376	30
D-2 分位	2147233280	31	B-2 分位	2147715328	30
D-1 分位	2147719936	31	B-1 分位	2147126272	31
C-8 分位	1395194880	25	A-8 分位	858840448	23
C-7 分位	1816200960	26	A-7 分位	1419995520	24
C-6 分位	2050073088	27	A-6 分位	1917255424	26
C-5 分位	2129575424	28	A-5 分位	2111323904	28
C-4 分位	2144957696	29	A-4 分位	2144641152	29
C-3 分位	2147493376	31	A-3 分位	2147665024	30

C-2 分位	2148022400	31	A-2 分位	2148039680	31
C-1 分位	2146931328	31	A-1 分位	2147504384	31

2.2.2 扩散性质

我们对 S-盒的扩散性质进行了测试。实验结果显示，经过25轮迭代后，S-盒的每一个分位函数与 S-盒的每一个输入比特相关。因此，可以认为 S-盒(经过32轮迭代)的每一个输入比特能够较为充分地扩散到 S-盒的每一个输出比特。

2.2.3 差分性质

本文对 S-盒的差分性质进行了相关测试。因为算法中采用了32-比特 S-盒，所以计算整个 S-盒的差分分布表的复杂度是2^{63}，远远超出了普通计算机的计算能力。

一般来说，相对于多重的差分，单重的差分扩散更为缓慢。于是，我们首先计算了所有的单重输入差分对应的输出差分分布表。表 5 给出了单重输入差分到输出差分的最大差分概率。

表 5 单重输入差分的输出差分的最大概率

输入差分	输出差分的最大概率	输入差分	输出差分的最大概率	输入差分	输出差分的最大概率	输入差分	输出差分的最大概率
A-8 分位	$13/2^{31}$	B-8 分位	$10/2^{31}$	C-8 分位	$13/2^{31}$	D-8 分位	$10/2^{31}$
A-7 分位	$9/2^{31}$	B-7 分位	$9/2^{31}$	C-7 分位	$12/2^{31}$	D-7 分位	$10/2^{31}$
A-6 分位	$10/2^{31}$	B-6 分位	$9/2^{31}$	C-6 分位	$12/2^{31}$	D-6 分位	$10/2^{31}$
A-5 分位	$9/2^{31}$	B-5 分位	$10/2^{31}$	C-5 分位	$20/2^{31}$	D-5 分位	$9/2^{31}$
A-4 分位	$10/2^{31}$	B-4 分位	$9/2^{31}$	C-4 分位	$11/2^{31}$	D-4 分位	$10/2^{31}$
A-3 分位	$10/2^{31}$	B-3 分位	$9/2^{31}$	C-3 分位	$10/2^{31}$	D-3 分位	$9/2^{31}$
A-2 分位	$10/2^{31}$	B-2 分位	$9/2^{31}$	C-2 分位	$9/2^{31}$	D-2 分位	$10/2^{31}$
A-1 分位	$11/2^{31}$	B-1 分位	$10/2^{31}$	C-1 分位	$9/2^{31}$	D-1 分位	$9/2^{31}$

其次，因为 SPRING 算法的线性扩散层是字节拉线，所以我们对单字节到单字节的差分概率进行了检测，并且记录了各输入字节到各输出字节的最大差分概率，见表 6。

表 6 各输入字节到各输出字节最大差分概率

输入差分 / 输入差分	第 1 字节	第 2 字节	第 3 字节	第 4 字节
第 1 字节	$\frac{6}{2^{31}}$	$\frac{5}{2^{31}}$	$\frac{5}{2^{31}}$	$\frac{6}{2^{31}}$
第 2 字节	$\frac{6}{2^{31}}$	$\frac{6}{2^{31}}$	$\frac{5}{2^{31}}$	$\frac{6}{2^{31}}$
第 3 字节	$\frac{7}{2^{31}}$	$\frac{6}{2^{31}}$	$\frac{5}{2^{31}}$	$\frac{6}{2^{31}}$
第 4 字节	$\frac{5}{2^{31}}$	$\frac{6}{2^{31}}$	$\frac{7}{2^{31}}$	$\frac{6}{2^{31}}$

最后，我们随机选取了1000组重量为16的输入差分进行测试，其中最大的差分概率是$12/2^{31}$，其具体分布如表 7。

表 7 最大差分概率分布情况

最大差分概率	数量
$9/2^{31}$	477
$10/2^{31}$	490
$11/2^{31}$	32
$12/2^{31}$	1

从上述的测试结果可以看出，目前找到的最大差分概率为$20/2^{31}$。值得注意的是，对于8比特 S-盒其最大差分概率至少为2^{-6}。而$\frac{20}{2^{31}} < 2^{-26} < (2^{-6})^4$，也即，基于目前找到的最优差分概率，我们的 S-盒相较于4个并置的8比特 S-盒仍有一定的优势。

2.2.4 线性性质

类似地，对于32-比特的 S-盒，我们难以找出最优线性逼近。因此，我们首先随机选择了1000个输入输出掩码对并计算了其线性逼近优势。实验结果显示，其

最优线性逼近概率为$\left(\frac{119}{2^{22}}\right)^2 \approx 2^{-30.21}$。

2.3 差分分析

我们通过差分特征的活动 S 盒个数和目前已知的 S 盒最优差分概率，对算法抵抗差分分析的能力进行评估。

因为 SPRING 三个版本的算法扩散层只有基于字节的位置置换，所以最差可能情况是，一条r轮差分特征的活动S盒个数等于r，即每一轮只有一个活动S盒。那么，SPRING-128-128 经过5轮后至少有5个活动 S-盒。按照目前找到的 S-盒最优差分概率，我们有$\left(\frac{20}{2^{31}}\right)^5 < (2^{-26})^5 = 2^{-130}$，也即，**5-轮 SPRING-128-128不存在有效的差分特征**。类似地，**5轮 SPRING-128-256 不存在有效的差分特征，10轮 SPRING-256-256不存在有效的差分特征**。本文给出了 SPRING-128-128/256 的4轮差分特征以及 SPRING-256-256 的8轮差分特征，见图 32 和图 33。其中，SPRING-128-128/256 的4轮差分特征成立概率为$2^{-114.45}$，SPRING-256-256 的8轮差分特征成立的概率为$2^{-229.64}$。

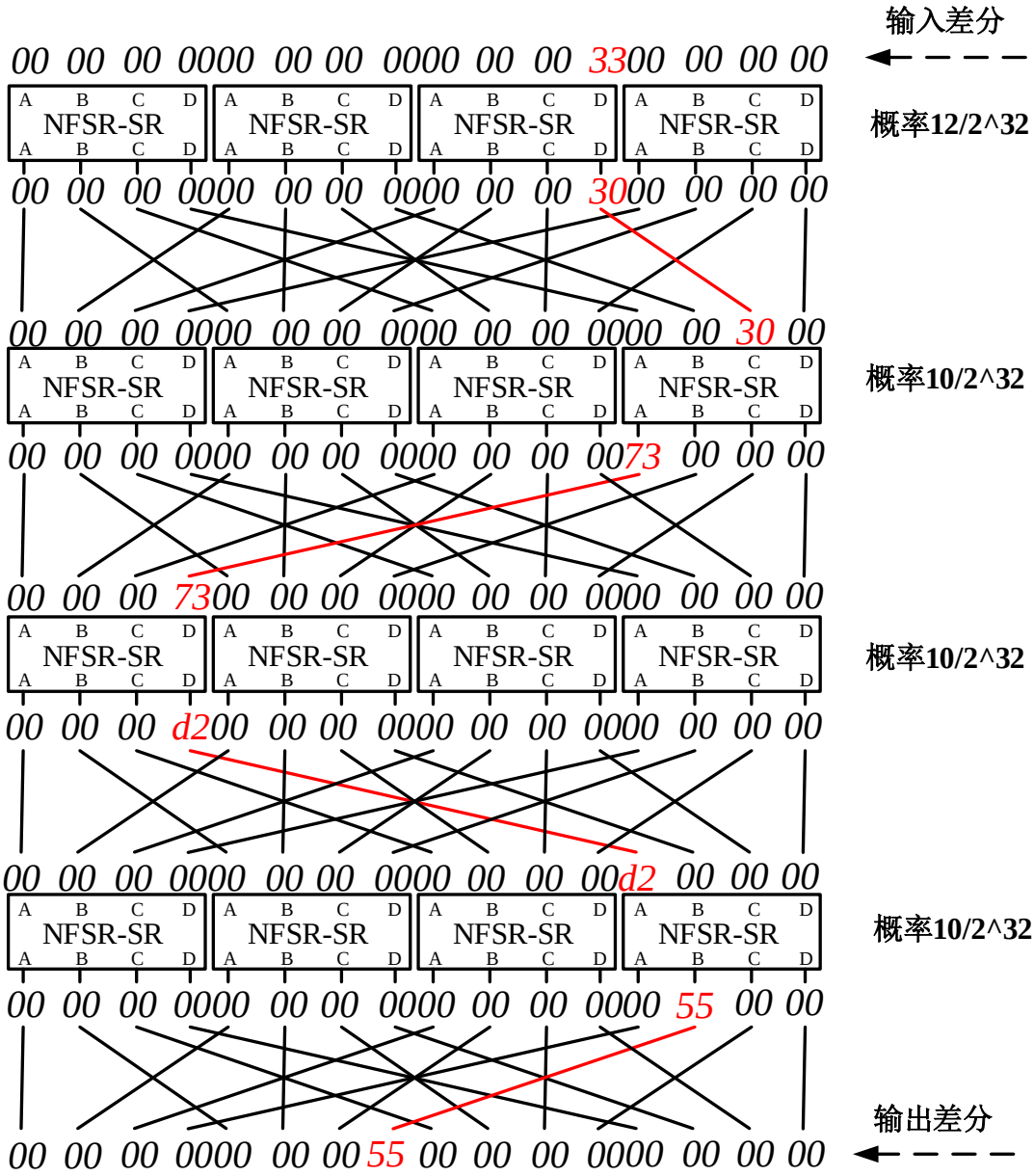


图 32 SPRING-128-128/256 的4轮差分特征

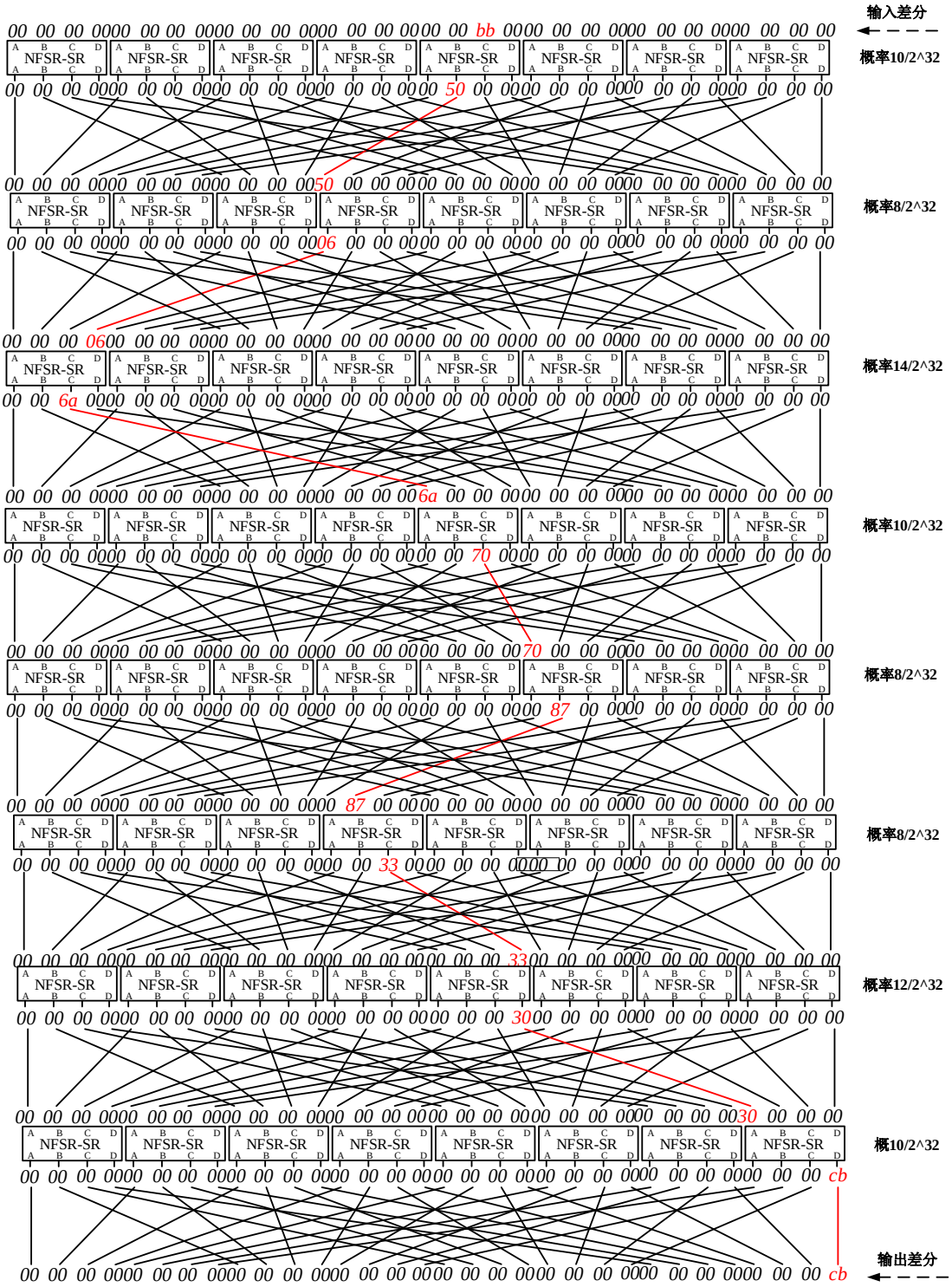


图 33 SPRING-256-256 的8轮差分特征

2.4 线性分析

类似差分分析活动 S 盒的估计，r轮 SPRING 算法活动 S 盒个数下界为r，

SPRING-128经过5轮后至少有5个活动 S-盒，SPRING-256 经过10轮后至少有10个活动 S 盒。因此，按照目前找到的最优线性逼近概率，我们有$(2^{-30.21})^5 < 2^{-151}$，也即，5轮 SPRING-128不存在有效的线性特征，9轮 SPRING-256不存在有效的线性特征。

2.5 不可能差分分析

在这一节中，我们对 SPRING 系列算法抵抗不可能差分分析的能力进行了评估。

一般来说，选取单块的固定输入输出差分能够搜索到更长的不可能差分路径。因此，我们针对单块的固定输入输出差分，利用中间相遇攻击的思想，搜索了 SPRING 系列算法的不可能差分，以 SPRING-128-128 为例，我们简要说明一下我们的搜索方法。

(1) 我们不妨选取第一字节为非零固定输入差分，其余字节均为零差分，记为$(\alpha 000,0000,0000,0000)$，经过一轮向后传播，得到$(\alpha_1 000,\alpha_2 000,\alpha_3 000,\alpha_4 000)$，其中$\alpha_1,\alpha_2,\alpha_3,\alpha_4$均为任意差分。

(2) 选取第一字节为非零固定输入差分，其余字节均为零差分，记为$(\beta 000,0000,0000,0000)$ 经过一轮向前传播，得到$(\beta_1\beta_2\beta_3\beta_4,0000,0000,0000)$，其中$\beta_1,\beta_2,\beta_3,\beta_4$均为任意差分。

(3) 若此时$(\alpha 000,0000,0000,0000) \to (\beta 000,0000,0000,0000)$为不可能差分，则必须满足$\alpha_1 \neq \beta_1$且$\alpha_i = \beta_i = 0(i = 2,3,4)$，我们穷搜了$\alpha$和$\beta$，并未发现存在某对$\alpha$和$\beta$使得上述条件成立。

相似地，按照上述方法我们对输入差分和输出差分的位置也进行了穷搜，仍旧无法搜索到 SPRING-128-128 的2轮不可能差分路径。同理我们也无法搜索到 SPRING-256-256 的3轮不可能差分路径。

我们给出 SPRING 系列算法我们能找到的不可能差分轮数和路径，具体结果见表 8。对于不可能差分分析, SPRING 系列算法有比较充足的安全余量。

表 8 SPRING 系列算法不可能差分区分其轮数

模式	不可能	不可能差分路径

	差分攻击轮数	
SPRING-128-128	1	$(\alpha000,0000,0000,0000) \rightarrow (0000,\beta000,0000,0000)$
SPRING-128-256	1	$(\alpha000,0000,0000,0000) \rightarrow (0000,\beta000,0000,0000)$
SPRING-256-256	2	$(\alpha000,0000,0000,0000,0000,0000,0000,0000) \rightarrow (0000,\beta000,0000,0000,0000,0000,0000,0000)$

2.6 积分分析

本节对 SPRING 针对积分分析的安全性进行分析。

由于 SPRING 的整体结构与 Present 算法相似，因此对于 SPRING 的积分分析，我们参考 Present 算法已有的分析过程。针对 Present 算法的积分分析，目前最有效的方法是利用基于比特的可分性质进行积分器搜索。下面我们将该方法应用到 SPRING 算法上。

文献**错误!未找到引用源。**中描述了如何将基于比特的可分性的传播规律转化为混合整数线性规划问题的方法，利用该方法，我们对 SPRING 算法进行建模并放入 gurobi 求解器中求解，从而进行积分区分器的搜索。为了找到轮数尽可能长的积分区分器， 我们遍历 SPRING-128-128 的所有127维的输入空间，实验结果表明，利用基于比特的可分性无法得到 SPRING-128-128的3轮积分器。

对于2轮 SPRING-128-128/256 算法，我们给出32维积分区分器如下：

$$(A^{32}, C^{96}) \xrightarrow{2\text{轮}} B^{128}$$

即当输入明文子空间跑遍第一个 S 盒的所有可能取值时，第二轮的所有输出比特都是平衡的。

对于 4 轮 SPRING-256-256算法，我们给出128维的积分区分器如下：

$$(A^{32}, C^{32}, A^{32}, C^{32}, A^{32}, C^{32}, A^{32}, C^{32}) \xrightarrow{4\text{轮}} B^{256}$$

对于255维的输入集合(A^{255}, C^{1})，我们利用基于比特的可分性，判断出其经过5

轮 SPRING-256-256算法后得到的输出比特是不平衡的。

2.7 中间相遇分析

我们运用 Demirci-Selçuk 中间相遇攻击技术[7]对 SPRING 算法不同分组长度和密钥长度的三个设计版本进行了分析，分别得到了最长轮数的中间相遇区分器和最大轮数的中间相遇攻击，具体结果参见表 9。

表 9 SPRING算法中间相遇攻击参数

算法	区分器轮数	攻击轮数	选择明文量	存储复杂度	时间复杂度
SPRING-128-128	2	5	2^{32}	2^{56}	2^{64}
SPRING-128-256	3	7	2^{32}	2^{184}	2^{192}
SPRING-256-256	3	7	2^{32}	2^{184}	2^{192}

2.7.1 SPRING-128-128 的2轮中间相遇区分器和4轮中间相遇攻击

在 AES 中间相遇区分器的建立中，从一个δ-集出发，考虑δ-集差分经过轮函数加密后到某个输出字节差分的映射关系全体，其中一个 δ-集包含256个明文分组，这些明文分组满足在一个字节位置恰好遍历所有256种可能值，其它字节差分为0。在 SPRING 算法的中间相遇分析中，我们也考虑有且仅有一个活动字节的δ-集。另外，由于 SPRING 算法中轮子密钥加在 S 盒运算前面，所以我们直接考虑一个活动字节到一个输出字节的映射关系，考虑它们的差分关系不会降低中间相遇攻击的复杂度。

性质 4 经过 2 轮的 SPRING-128-128 加密后，第 0 字节活跃的$\delta-$集$(P_0,P_1,\dots,P_{255})$满足如下图所示的差分传播路径，则 $(P_0,P_i)(i=0,1,\dots,255)$在$X_3[0]$处的差分形成的多重集至多有$2^{48}$种可能。

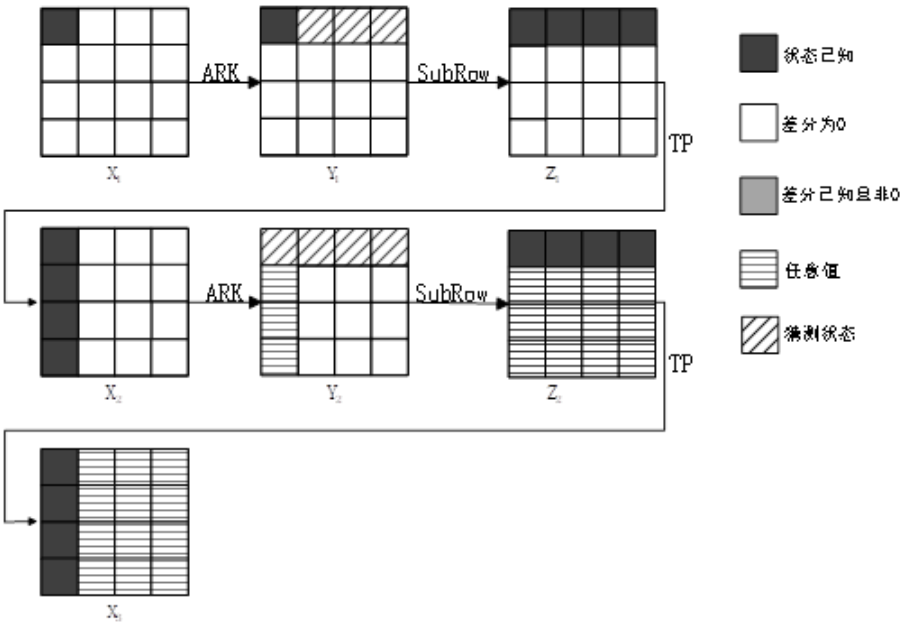


图 34　SPRING-128-128 的2轮中间相遇区分器

该两轮区分器需要$2^{48} \cdot 2^{8} = 2^{56}$个单字节的存储空间。由于算法加密过程中涉及到 32 字节的 S 盒替换以及转置，因此区分器每增加一轮，则确定的位置会增加 16 字节。在该区分器增加一轮后，复杂度将达到2^{184}。

因此，对于 SPRING-128-128 而言，中间相遇攻击的区分器至多为 2 轮。

根据性质 4，我们区分器前后各加一轮，形成5轮 SPRING-128-128 的中间相遇攻击，攻击过程如下所述。

a. 离线预计算阶段

根据性质 4，计算由δ-集在 $X_3[0]$处差分值的所有 2^{48} 种可能，并将所有差分存储在表 T_1 中。

b. 在线攻击阶段：

（1）选取 2^{32} 明文，这些明文满足在第一行遍历 2^{32} 个取值，其余 12 字节相等，如图 2 中 X_0 所示；猜测密钥$k_0[0,4,8,12]$，计算$X_1[0]$，从 2^{32} 个明文中选出 256 个明文$\{P_0, P_1, \dots, P_{255}\}$，使其在在 X_1 处的值形成δ-集；

（2）用 5 轮 SPRING-128-128 加密 256 个明文，得到密文$\{C_0, C_1, .. C_{255}\}$；

（3）猜测密钥$k_4[0,1,2,3]$，部分解密$\{C_0, C_1, .. C_{255}\}$，如图所示，可求得

$\{Y_3^0[0], Y_3^1[0], \ldots, Y_3^{255}[0]\}$;

（4）判断差分多重集$\{X_3^0[0] \oplus X_3^0[0], X_3^1[0] \oplus X_3^0[0], \ldots, X_3^{255}[0] \oplus X_3^0[0]\}$是否在表中；若在 T_1 中，判定猜测密钥为正确密钥；否则为错误密钥。

上述五轮 SPRING-128-128 的攻击过程中，选择明文量为2^{32}，存储复杂度为2^{64}，猜测密钥量 8 个字节，即时间复杂度为2^{64}。

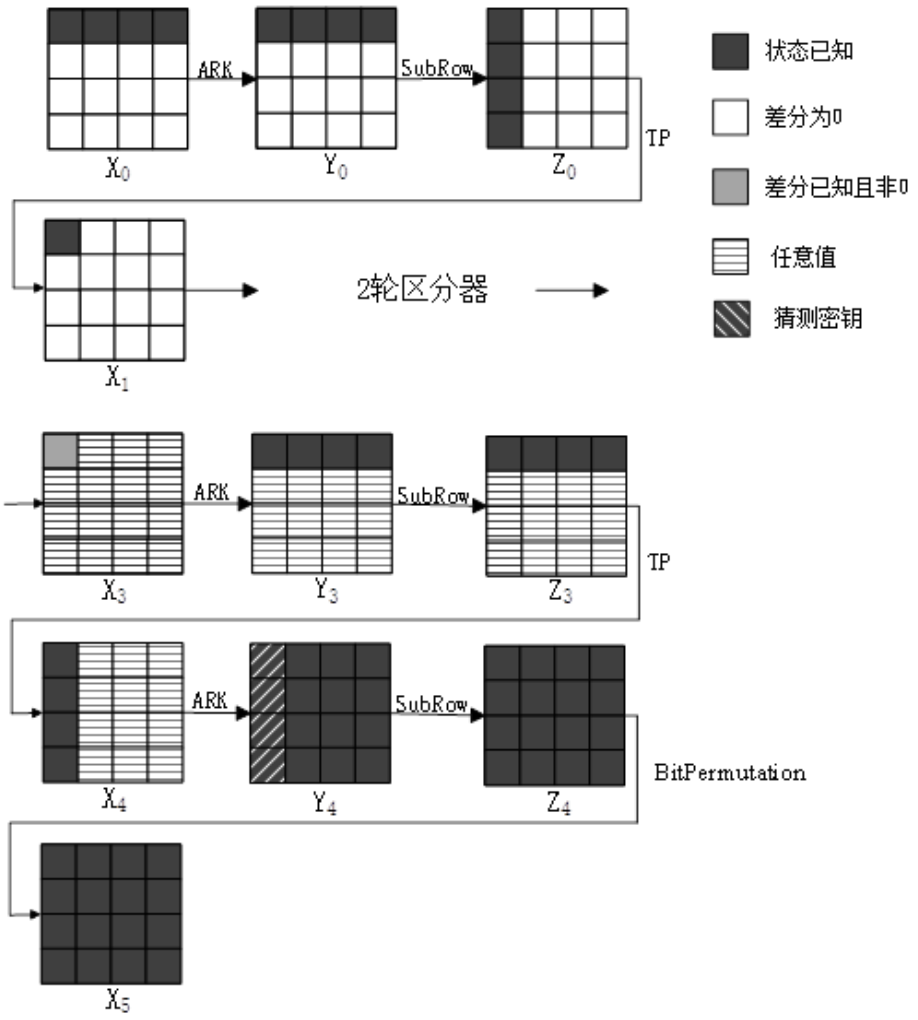


图 35　五轮 SPRING-128-128 的中间相遇攻击

2.7.2 SPRING-128-256 的3轮中间相遇区分器和7轮中间相遇攻击

性质 5 经过 3 轮的 SPRING-128-256 加密后，第 0 字节活跃的$\delta-$集$(P_0, P_1, \ldots, P_{255})$满足如下图所示的差分传播路径，则 (P_0, P_i) $(i = 0,1, \ldots,255)$在

$X_4[0]$处的差分形成的多重集至多有2^{176}种可能。

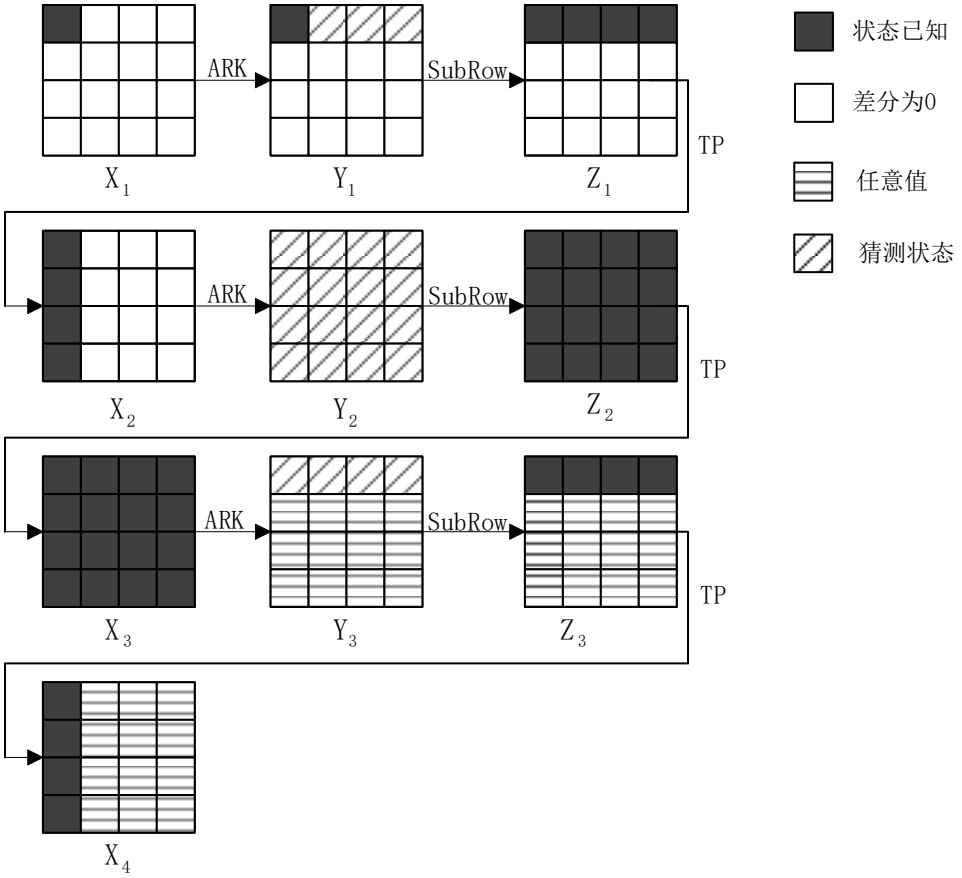


图 36 SPRING-128-256 的 3 轮中间相遇区分器

该三轮区分器需要$2^{176} \cdot 2^8 = 2^{184}$个单字节的存储空间。该区分器增加一轮后，复杂度将达到2^{312}。

因此，SPRING-128-256 中间相遇攻击的区分器至多为三轮。

由性质 **5**，在区分器前加一轮，区分器后加三轮，可形成七轮 SPRING-128-256 的中间相遇攻击，攻击过程如下所述。

a. 离线预存储阶段:

由性质 **5**，计算由δ-集在 $X_4[0]$处差分值的所有 2^{176} 种可能，并将所有差分存储在表 T_2 中。

b. 在线攻击阶段:

（1）选取 2^{32} 明文，这些明文满足在第一行遍历 2^{32} 个取值，其余 12 字节

相等，如图中 X_0 所示；猜测密钥$k_0[0,4,8,12]$，计算$X_1[0]$，从 2^{32} 个明文中选出 256 个明文$\{P_0,P_1,\dots,P_{255}\}$，使其在在 X_1 处的值形成δ-集；

（2）用 7 轮 SPRING-128-256 加密 256 个明文，得到密文$\{C_0,C_1,..C_{255}\}$；

（3）猜测密钥$k_5[0,1,2,3]$以及全部的k_6，部分解密$\{C_0,C_1,..C_{255}\}$，如图所示，可求得$\{Y_4^0[0],Y_4^1[0],\dots,Y_4^{255}[0]\}$；

（4）查询表 T_2，判断$\{X_4^0[0]\oplus X_4^0[0],X_4^1[0]\oplus X_4^0[0],\dots,X_4^{255}[0]\oplus X_4^0[0]\}$是否在表中；若在 T_2 中，则判定所猜测密钥为正确密钥；否则为错误密钥。

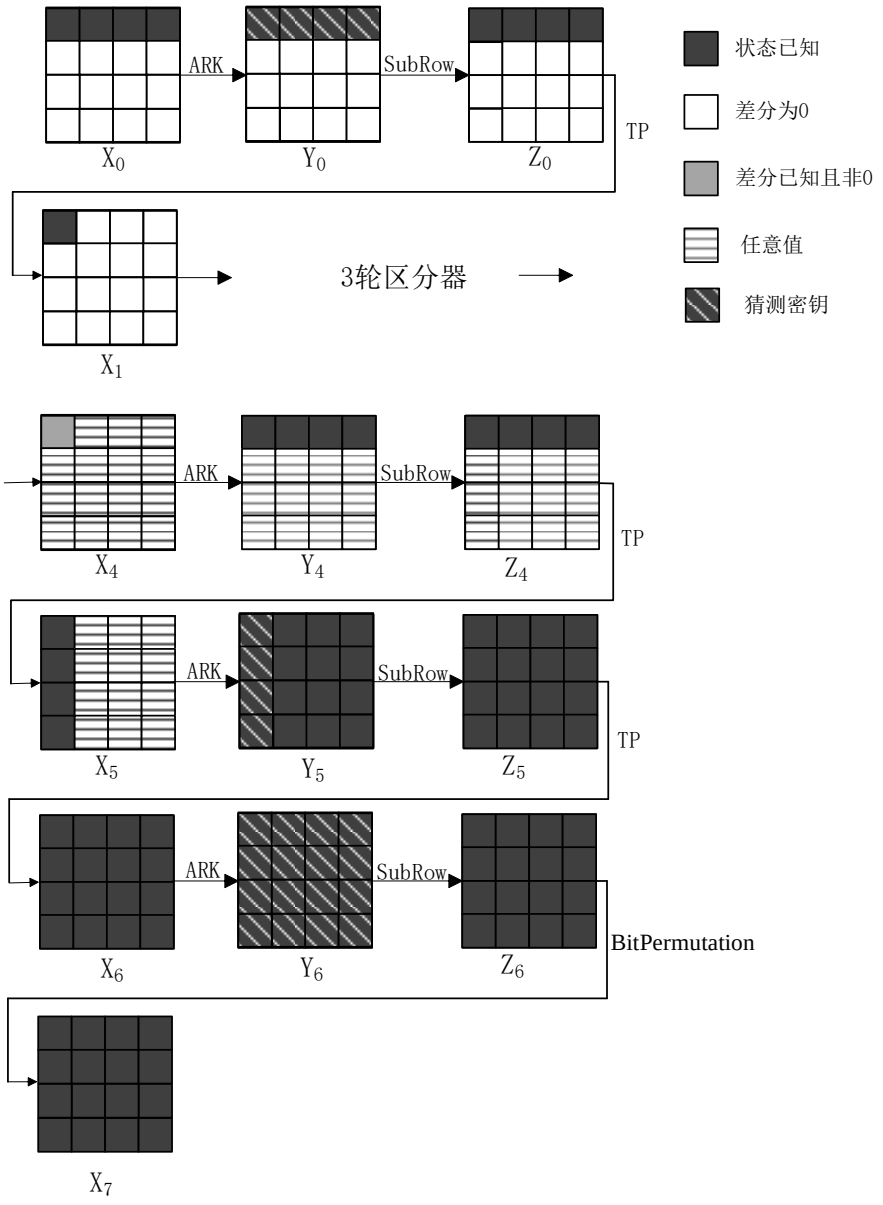


图 37　SPRING-128-256 的 7 轮中间相遇攻击

上述七轮 SPRING-128-256 的攻击过程中，选择明文量为2^{32}，存储复杂度为2^{184}，猜测密钥量 24 个字节，即时间复杂度为2^{192}。

2.7.3 SPRING-256-256 的3轮中间相遇区分器和7轮中间相遇攻击

性质 6 经过 3 轮的 SPRING-256-256 加密后，第 0 字节活跃的$\delta-$集$(P_0,P_1,\dots,P_{255})$满足如下图所示的差分传播路径，则(P_0,P_i) $(i=0,1,\dots,255)$在$X_4[0]$处的差分形成的多重集至多有2^{176}种可能。

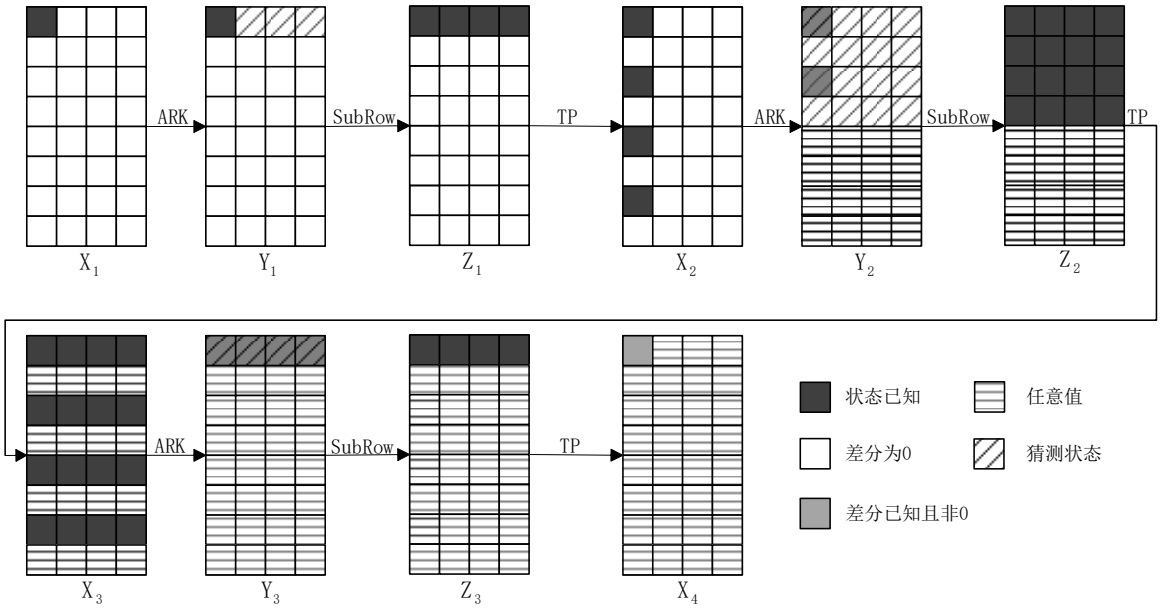


图 38　SPRING-256-256 的 3 轮中间相遇区分器

该三轮区分器需要$2^{176}\cdot 2^8=2^{184}$个单字节的存储空间。在增加一轮后，要确定X_5处的多重集，则需要确定的字节为X_2的第 0,2,4,6 行的 16 个字节，X_3前四行的 16 个字节，X_4第 0 行的 4 个字节，此时，四轮区分器的复杂度将达到2^{312}。

因此 SPRING-256-256 中间相遇攻击的区分器至多为三轮。

由性质 6，在区分器前加一轮，区分器后加两轮，可形成六轮 SPRING-128-256 的中间相遇攻击，攻击过程如下所述。

a. 离线预存储阶段：

由性质 6，计算由δ-集在 $X_4[0]$处差分值的所有 2^{176} 种可能，并将差分结果存储在表 T_3 中。

b. 在线攻击阶段：

（1）选取 2^{32} 明文，这些明文满足在第一行遍历 2^{32} 个取值，其余 28 字节相等，如图中 X_0 所示；猜测密钥$k_0[0,8,16,24]$，计算$Y_1[0]$，从 2^{32} 个明文中选出 256 个明文$\{P_0,P_1,\dots,P_{255}\}$，使其在在 X_1 处的值形成δ-集；

（2）用 7 轮 SPRING-256-256 加密 256 个明文，得到密文$\{C_0,C_1,..C_{255}\}$；

（3）猜测密钥$k_5[0,2,4,6],k_6[0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23]$，部分解密$\{C_0,C_1,..C_{255}\}$，如图所示，可求得$\{Y_4^0[0],Y_4^1[0],\dots,Y_4^{255}[0]\}$；

（4）查询表 T_3，判断$\{X_4^0[0]\oplus X_4^0[0],X_4^1[0]\oplus X_4^0[0],\dots,X_4^{255}[0]\oplus X_4^0[0]\}$是否在表中；若在 T_3 中，则判定所猜测密钥为正确密钥；否则为错误密钥。

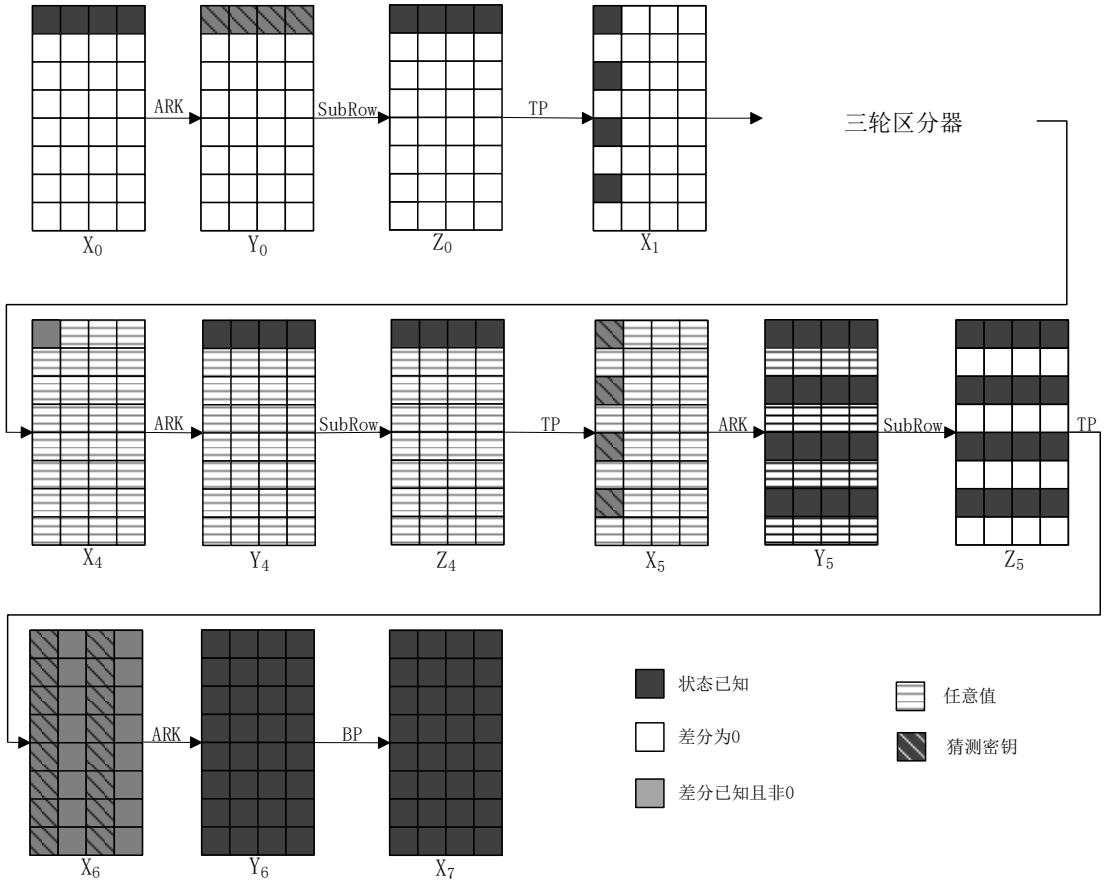


图 39　SPRING-256-256 的 7 轮中间相遇攻击

上述七轮 SPRING-256-256 的攻击过程中，选择明文量为2^{32}，存储复杂度为2^{184}，猜测密钥量 24 个字节，即时间复杂度为2^{192}。

3. 优缺点声明

优点：由于 SPRING 系列算法的 S-盒和密钥扩展算法均是基于非线性反馈移位寄存器设计的，算法硬件实现面积小，并且根据不同的应用可以调节实现面积与速率。根据我们的评估，工艺库为 TSMC 16nm 时，SPRING-128-128基于轮的实现面积只有约1046平方微米；如果将 SPRING-128-128的10轮全部展开实现，面积大约为8079平方微米左右，加密速率为2084Mbps。

缺点：暂时无法给出非线性部件 NFSR-SR 更新32拍的完整差分分布表和线性逼近概率表。

4. 参考文献

[1] Xiao-Xin Zhao, Tian Tian, and Wen-Feng Qi. A ring-like cascade connection and a class of nfsrs with the same cycle structures. Des. Codes Cryptography, 86(12):2775–2790, 2018.

[2] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Jung Hee Cheon and Tsuyoshi Takagi, editors, Advances in Cryptology -ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I, volume 10031 of Lecture Notes in Computer Science, pages 648–678, 2016.

[3] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT: A small present - towards reaching the limit of lightweight encryption. In Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings, pages 321–345, 2017.

[4] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August

14-18, 2016, Proceedings, Part II, volume 9815 of Lecture Notes in Computer Science, pages 123–153. Springer, 2016.

[5] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, Louis Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013).

[6] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In Tetsu Iwata and Jung Hee Cheon, editors, Advances in Cryptology - ASIACRYPT 2015 - 21st International Conference on the Theory and Application of Cryptology and Information Security, Auckland, New Zealand, November 29 - December 3, 2015, Proceedings, Part II, volume 9453 of Lecture Notes in Computer Science, pages 411–436. Springer, 2015.

[7] Hüseyin Demirci and Adi Aydm Selçuk. A meet-in-the-middle attack on 8-round AES. In: K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 116- 126, 2008.

5. 附录

各输入字节到各输出字节最大差分概率特征具体情况如下：

第一字节到第一字节：

输入差分	输出差分
0x42	0x07

第一字节到第二字节：

输入差分	输出差分
0x3f	0xd9
0x94	0x2a
0xc2	0x81
0xd2	0x55
0xf0	0x30
0xfb	0xbc

第一字节到第三字节：

输入差分	输出差分
0x02	0x1d
0x03	0x5b

0x09	0x16
0x12	0x2c
0x19	0xe0
0x27	0x7c
0x59	0xef
0x6a	0x70
0x6e	0x86
0x9b	0xfc
0x9d	0x2b
0xf9	0x69
0xfd	0x36

第一字节到第四字节：

输入差分	输出差分
0x12	0xa3
0x24	0x46

第二字节到第一字节：

输入差分	输出差分
0xe2	0x28

第二字节到第二字节：

输入差分	输出差分
0x3d	0xb8

第二字节到第三字节：

输入差分	输出差分
0x39	0x0c
0x3b	0x95
0x3d	0xe4
0x6d	0x59
0x97	0x88
0xa8	0xf1
0xc3	0x3a
0xd1	0xcd
0xe1	0x2f
0xeb	0x3e

0xfa	0xcf
0xfb	0x14

第二字节到第四字节：

输入差分	输出差分
0x1d	0x8c
0x87	0xa4

第三字节到第一字节：

输入差分	输出差分
0x20	0xe4

第三字节到第二字节：

输入差分	输出差分
0xa2	0xa1

第三字节到第三字节：

输入差分	输出差分
0x0a	0x49
0x30	0x82
0x38	0x75
0x5d	0x8f
0x86	0x07
0x8e	0xe0
0x97	0xb3
0xb6	0xef
0xc1	0xc6
0xde	0x5d
0xfa	0x3b

第三字节到第四字节：

输入差分	输出差分
0x51	0x54

第四字节到第一字节：

输入差分	输出差分
0x2a	0xf3
0x4a	0x41
0x4d	0xae

0x5f	0x46
0x68	0x89
0x6e	0x0c
0xab	0x44
0xda	0xa9

第四字节到第二字节：

输入差分	输出差分
0xc4	0xf4
0xe2	0xfa

第四字节到第三字节：

输入差分	输出差分
0x06	0x6a

第四字节到第四字节：

输入差分	输出差分
0x33	0x30