+

The Algorithm OaldresPuzzle_Cryptic

Technical Details

A new cryptographically secure symmetric encryption-decryption algorithm to resist the impact of future quantum computers on data security

China Video Web Space: Twilight-Dream Sparkle-Magic Email: Link

Update content date: 2025–04–09

Abstract

本文介绍了一种新的对称加密-解密算法”OaldresPuzzle_Cryptic”, 旨在抵抗未来量子计算机对数据安全的影响. 该算法将利用和实施各种技术, 包括一个基于混沌理论系统的加密安全伪随机数生成器, 该系统模拟了双段摆锤的轨迹;1 个独立设计和实现的非线性反馈移位寄存器, 具有温和的混沌特性;1 个具有序列周期长度为 2 的 128 次方的线性反馈移位寄存器、2 对静态的字节替换盒, 用于模拟高强度的非线性颗粒度函数, 由伽罗瓦有限域中的计算和不可约的本原多项式生成;2 个动态的字节替换盒; 并且使用了线段数的数据结构并配合非线性反馈移位寄存器, 进一步的扰乱生成密钥数据的规律; 一个模仿 ZUC 序列密码设计的结构, 并且使用了动态的字节替换盒, 使每个生成的密钥不可预测.

此外还使用了线性代数的运算, 比如仿射变换、克朗克积、点积、解决转置和伴随矩阵, 以及矩阵的加、减和乘法. 包括使用了布尔运算 AND, OR, NOT, XOR, XNOR; 这些运算共同构成了本算法的子密钥生成模块和每轮轮回函数中使用的子密钥生成模块.

上述两个模块产生的子密钥数据被与 Lai-Massey Scheme 协调设计的单向函数所使用, 它们共同构建了一个抽象的、在计算上无法区分的安全伪随机函数.

尽管”OaldresPuzzle_Cryptic ” 具有抵御量子计算机攻击的潜力, 但需要注意的是, 其有效性尚未得到测试. 该算法可以被认为是对称加密-解密领域的一个微创新, 为量子计算在数据安全方面的挑战提供了一个新的解决方案.

Introduction

OaldresPuzzle_Cryptic Algorithm 是一种面向未来的对称（分组/数据块）加密和解密算法的微创新. 特别是, 它解决了量子计算机带来的潜在威胁. 随着技术的不断进步, 对更安全和强大的加密方法的需求变得越来越重要

传统的加密方法, 如 RSA 和 AES, 在量子计算机成熟之前的时代, 可以在传统的基于比特的计算机平台上使用. 可以实现与传统比特平台相同的量子加密水平, 代价是将密钥长度增加 2 倍. 以实现相同的量子比特安全水平. 我们不能只关注 Shor 的算法而忽视 Grover 算法的潜在威胁, 因此, 即使在量子计算机成熟之后, 也有必要开发新的方法来防御这些威胁, 这样才能保证经典比特计算机的数据安全

OaldresPuzzle_Cryptic 算法通过结合使用现有的技术和数学运算, 使用主密钥经过我们设计的子密钥生成模块和每轮的轮函数使用的子密钥生成模块, 来创建几乎不可能破解的对称的子密钥来解决此问题.

原理涉及利用混沌理论系统的伪随机数生成器、具有混沌特性的非线性反馈移位寄存器、线性反馈移位寄存器、2 对模拟非线性强函数的静态字节替换盒 (包含在 2^8 的伽罗瓦有限域)、2 个动态字节替换盒, 以及各种数学运算, 包括线性代数和布尔运算等等组合, 具备潜在的能力能够抵抗现在已知的和未来的量子计算机的攻击.

用于保护文件和小型磁盘的数据的理想解决方案. 该算法旨在高度安全, 使其适用于保护范围广泛的数据, 包括敏感和关键信息. 满足未来的数据安全需求, 可以保护数据免受未来量子计算机的影响.

在本文中, 我们详细描述了 OaldresPuzzle_Cryptic 算法及其各个组成部分. 此外, 我们将探讨 OaldresPuzzle_Cryptic 算法在数据安全领域的潜在未来发展和应用, 包括将其整合到现有系统所需的步骤以及使用该算法所需的基础设施. 该算法的潜在可扩展性以及其适应未来技术和量子计算发展的能力. 该算法的组成部分也被详细描述.

总之, 本文提出了一种新的、创新的加密-解密算法, 可以抵抗量子计算对数据安全的威胁.OaldresPuzzle_Cryptic 算法使用各种技术和数学运算来创建一个独特的加密-解密密钥, 几乎无法破解. 这种算法高度安全, 适合保护关键数据和信息免受未来量子计算的攻击.

目录

1 Topic

如果读者难以理解本文的介绍和摘要部分, 我有必要重申, 本研究中使用的理论框架与密码学领域有关, 涉及对称加密和解密的应用. 具体来说, 新的 OaldresPuzzle_Cryptic 算法中使用的对称加密和解密的框架是基于 Lai-Massey 方案的, 该方案拥有的特性使其能够抵抗基于量子的攻击. 这个框架已经得到了相关文献的支持 ([3] [13]) . 此外, 我已经用 C++ 代码完全实现了 Lai-Massey 方案的 F 函数和 H 函数. 我们将在后面提到 OaldresPuzzle_Cryptic 算法的公式.

2 Frequently Asked Questions

Q: 新算法的密钥长度是多少, 它与现有的对称算法相比有何不同?

A: 它的数据分块大小和密钥分块大小最低 512 比特. 目前我进行了的测试都应该在 512 比特以上, 但是实际上我并没有严格要求, 只是说它大于 512 比特并且是 8 的倍数就符合和后量子密码学的要求. 因为我使用的 c++ 语言是静态模板参数, 建议大家还是按照 64 比特的倍数传递模板参数.

Q: 在新算法的设计中使用了哪些数学和计算系统, 它们对其安全性有何贡献?

A: 基本上我的论文抽象和介绍, 已经说明了我使用的数学原理和计算方法. 这里我就不需要再次强调了. 如果想了解详细信息, 我应该会提供这个 OPC 算法模块的流程图.

Q: 与现有的对称算法相比, 新算法的优势和劣势是什么?

A: 灵活的密钥长度, 更长的密钥, 更好的安全性. 算法速度一直是个绕不开令人诟病的话题. 但是我是为了质量安全而牺牲速度的, 所以它的应用方面可能不是很广. 我建议把该算法应用在小型数据的处理上面, 发挥它的最佳性能.

Q: 该算法是否受到过任何攻击或安全评估, 其结果是什么?

A: 目前没有能力进行大规模设备的超级算力测试, 我也在全力的请求各界人士帮助, 如果你看到了这篇论文, 请你帮助我评估该算法的可能性.

Q: 新算法对密码学领域有什么影响, 它对该学科的发展有什么贡献?

A: 虽然可能我做的贡献非常的小, 但是我希望我的想法, 能够给未来研究密码学的更专业的学者, 提供更好的主意和帮助. 另外, 我竭尽全力的利用了那些以前前人们设计对称加密解密算法的思想以及所用到的数学方法. 每一次设计和实现 OPC 算法代码, 我都会谨慎的使用. 希望你们能给我信心, 稍后我在讲述完毕之后, 我会说明一部分这个算法会用到的数学公式.

本文作者设计了一种新的对称加密和解密算法, 并请求针对现有的最先进的计算系统测试其该算法的安全性. 为了实现这一目标, 作者建议将该算法置于超级计算机或量子计算机的攻击之下.

总之, 新设计的对称加解密算法有可能解决现有算法的缺陷, 提高对称加解密方法的安全性. 然而, 彻底评估该算法并回答有关其安全性和有效性的问题是很重要的. 该算法的作者正在寻求支持和资源, 使该算法受到来自量子计算机或超级计算机的攻击, 以全面评估其安全性. 需要进一步的研究和测试来确定这种新算法对密码学领域的潜在影响.

感谢你看到这里, 接下来我会讲述这个算法用到的一些数学公式以及如何构建这个算法. 如果你有兴趣, 请你认真的往下看, 谢谢.

3 Known existing symmetric encryption and decryption frameworks and comparisons

我们将描述这个特定的对称加密和解密框架的优点和缺点. 此外, 我们的目标是阐述本研究中使用的 Lai-Massey 方案框架与密码学领域专家公认的其他类似的对称加密和解密结构之间的区别.

1.Feistel Network(费斯特尔网络)

在密码学中, 费斯特尔密码 (又称卢比-拉科夫 (Luby–Rackoff) 分组块密码) 是一种用于构建区块密码的对称结构, 以德国出生的物理学家和密码学家霍斯特-费斯特尔命名, 他在为 IBM 工作时做了开创性的研究; 它也通常被称为费斯特尔网络. 在费斯特尔密码中, 加密和解密是非常相似的操作, 都是由反复运行一个被称为" 轮函数" 的函数的固定次数组成.

Feistel 网络的实现可以描述如下:

设 B 为输入块,$K_1, \ldots, K_n$ 为轮密钥. 输入块 B 首先被分成两个大小相等的一半 L 和 R. 应用轮函数时需要使用 i 轮的密钥.

其中一半应用轮函数操作完毕之后, 与另一半进行 $\oplus_{64}$(XOR) 异或操作, 把结果替换原有的一半, 然后互相交换数据; 由另一半重新应用轮函数进行操作, 与原一半进行 $\oplus_{64}$(XOR) 异或操作, 把结果替换原有的另一半, 然后互相交换数据.....

L_0 和 R_0,Feistel Network 的加密和解密, 定义如下: Luby–Rackoff: 7 Rounds Are Enough for $2n^{1-\epsilon}$ Security

For each round index i = 0, process i , . . . , **message_block_size**, compute

$$\textbf{FeistelNetworkEncryption}(L_i, R_i, K_i) = \begin{cases} L_{i+1} = R_i \\ R_{i+1} = L_i \oplus_{64} F(R_i, K_i) \end{cases}$$

For each round index i = **message_block_size**, process **message_block_size** - 1 , . . . , 0, compute

$$\textbf{FeistelNetworkDecryption}(R_{n+1}, L_{n+1}, K_i) = \begin{cases} R_i = L_{i+1} \\ L_i = R_{i+1} \oplus_{64} F(L_{i+1}, K_i) \end{cases}$$

其中 $\oplus_{64}$ 表示按位异或 (XOR),$F(R_{i-1}, K_i)$ 是应用于输入 R_{i-1} 和轮密钥 K_i 的轮函数.

Feistel 网络的输出是 R_n 和 L_n 的串联/拼接数据.

2.Substitution–Permutation Network(替换-置换/代换-排列网络)

在密码学中, 替换置换网络是用于分组密码算法的一系列链接数学运算, 例如以下加密解密算法使用了替换置换网络 AES (Rijndael) 3-Way Kalyna Kuznyechik PRESENT SAFER SHARK 和 Square.

这样的网络将明文块和密钥作为输入, 并交替应用几轮或几层替换框 (S-boxes) 和置换框 (P-boxes) 来生成密文块. S 盒和 P 盒将输入位的（子）块转换为输出位. 这些转换通常是在硬件中高效执行的操作, 例如异或 (XOR) 和按位旋转. 密钥在每一轮中引入, 通常以从中派生的“轮密钥”的形式出现.(在某些设计中,S-box 本身取决于密钥.)

SPN 的实现可以描述如下:

设 B 为输入块,$K_1, \ldots, K_n$ 为轮密钥. SPN 由 n 轮组成, 其中每一轮 i 将块 B_i 作为输入并输出 B_{i+1}.

SPN 定义如下:

For each round index i = 0, i ... **message__block__size**, compute

$$\textbf{EncryptionWithSPN}(B_i, K_i)$$
$$B_{i+1} = \mathbf{P}(\mathbf{S}(B_i \oplus_{64} K_i))$$

$$\textbf{DecryptionWithSPN}(B_i, K_i)$$
$$B_{i+1} = \mathbf{S^{-1}}(\mathbf{P^{-1}}(B_i)) \oplus_{64} K_i$$

其中 $\oplus_{64}$ 表示按位异或 (XOR),P 是置换函数,$S_1, \ldots, S_m$ 是应用于输入 $B_i \oplus_{64} K_i$ 的 S-boxes. SPN 的输出是 B_n.

S 可以被看作为一个替换函数 (举个例子, 在本文中 (OPC algorithm - The bytes data secure substitution layer), 就已经解释了这个 S 函数的具体实现过程.)

就是每一个语句的

$$DataArray_{index} := SubstitutionBox_{DataArray_{index}}$$

3.Lai–Massey Scheme(莱-马西方案) ([27] [11] [21])

在密码学中,Lai–Massey Scheme 在设计上类似于 Feistel Network. 使用一个轮函数和一个半轮函数. 轮函数是一个接受两个输入, 一个子密钥和一个数据块, 并返回一个与数据块等长的输出的函数. 半轮函数接受两个输入, 并将其转化为两个输出. 对于任何给定的回合, 输入被分成两半, 即左和右.

最初, 输入被传递给半轮函数. 在每一轮中, 输入之间的差异与一个子密钥一起被传递给轮函数, 然后轮函数的结果被加到每个输入中. 然后, 输入被传递到半轮函数中. 然后重复固定的次数, 最后的输出是加密的数据.

由于它的设计, 它比 Substitution–Permutation Network 更有优势, 因为轮函数不需要是双射性质的, 可以为单射性质–只需要半轮函数满足双射性质–使它更容易被反转, 并使轮函数可以任意地复杂. 加

密和解密过程相当相似, 解密则需要颠倒密钥计划, 求半轮函数的反函数, 以及轮函数的输出被减去而不是增加. 由于二进制异或运算的自反性质, 我们可以把所有的加减法全部替换为二进制的异或运算.

L_0 和 R_0,Lai–Massey Scheme 的加密和解密, 定义如下:

令 F 为轮函数,H 为双射性质的半轮函数, 令 $K_0, K_1, \ldots, K_n$ 为轮次的子密钥分别为 $0, 1, \ldots, n$, 输入块 B 首先被分成两个大小相等的一半 L 和 R.

For each round index is i = 0, process i , . . . , **message_block_size**, compute

$$\begin{aligned}
&\textbf{LaiMasseySchemeEncryption}(L_i, R_i, K_i)\\
&\{L_i', R_i'\} = \mathbf{H}(L_i, R_i)\\
&TK_i = \mathbf{F}(L_i' - R_i', K_i)\\
&L_i'' = L_i' + TK_i\\
&R_i'' = R_i' + TK_i
\end{aligned}$$

For each round index is i = **message_block_size**, process **message_block_size** - 1 , . . . , 0, compute

$$\begin{aligned}
&\textbf{LaiMasseySchemeDecryption}(L_{n+1}, R_{n+1}, K_i)\\
&TK_i = \mathbf{F}(L_i'' - R_i'', K_i)\\
&L_i' = L_i' - TK_i\\
&R_i' = R_i' - TK_i\\
&\{L_i, R_i\} = \mathbf{H^{-1}}(L_i', R_i')
\end{aligned}$$

4 OaldresPuzzle_Cryptic Algorithm

我们采用从底层设计到表层实现的讲解方法.

OPC 加密函数和 OPC 解密函数其实设计结构很简单, 只需使用下列公式就可以解释:

$$\begin{aligned}
&Subkeys = \textbf{GenerateSubkeys}(Keys)\\
&RoundSubkeys = \textbf{GenerateRoundSubkeys}(Subkeys)\\
&\textbf{EncryptionWithOPC}(PlainDataVector, RoundSubkeys)\\
&\textbf{DecryptionWithOPC}(CipherDataVector, RoundSubkeys)
\end{aligned}$$

由 GenerateSubkeys 和 GenerateRoundSubkeys 函数组成的密钥生成系统, 是本文讨论的框架的一个组成部分. 这些函数负责生成子密钥和轮回函数所需的密钥, 这将在后续章节中详细讨论.

我们的密钥生成系统中使用的伪随机数生成器（PRNGs）可以分为三种类型.

第一种类型是线性反馈移位寄存器（LFSR）, 序列周期长度为 2^{128}.

第二种类型是我们设计的非线性反馈移位寄存器（NLFSR）, 它表现出混沌特性. 非线性反馈移位寄存器有两种不同的实现方式, 我们称之为" 大版本" 和" 小版本". 这两个版本背后的理论是不同的, 小版本是一个真正的 NLFSR. 相比之下, 大版本利用超越数或无理数, 选择小数点后的随机数字, 并将其转

换为 64 位的二进制表示. 经过几次多项式计算、数据扩散操作和比特操作, 一个不可预测的比特序列就产生了.

第三种类型是利用模拟物理双段摆锤的运动现象的混沌理论系统原理来产生安全的伪随机数序列. 然而, 这种方法中使用的系统是基于模拟双摆的物理现象. 输入的密钥经过一系列的变换, 得到一组可以被混沌系统使用的系统参数. 此外, 由于混沌系统的特征行为, 对于不同的输入参数, 输出状态会有所不同.

这三种类型的 PRNG 算法的具体实现和结构将在题为 (使用的 PRNG 详细组件实现) 一节中讨论. [15]

4.1 Predependent algorithms for key generation systems

在深入研究 OaldresPuzzle_Cryptic 算法的复杂性之前, 有必要在 Lai-Massey 方案的框架内研究上述算法的 F 函数. 值得注意的是, 该 F 函数所需的密钥生成系统是以其他算法的公式为前提的.

使用的线性反馈移位寄存器的函数如下:

Algorithm 1 OPC core algorithm - LFSR

Define variable state: $state \leftarrow \left[a, b\right]$
where $a, b \in [0, 2^{64} - 1]$

function INITIALIZE_BITS(seed)
    a := 0
    b := seed
    GENERATE_BITS(64)
    GENERATE_BITS(64)
end function

function GENERATE_BITS(bits_size)
    a $\leftrightarrow$ $state_0$
    b $\leftrightarrow$ $state_1$
    current_random_bit = 0
    answer = 128
    for *round_counter* := 0; **to** *bits_size* − 1; *round_counter* := *round_counter* + 1 **do**
        current_random_bit := POLYNOMIAL(a, b) $\wedge_{64}$ 1
        answer := answer $\ll_{64}$ 1
        answer := answer $\oplus_{64}$ current_random_bit
        b := b $\gg_{64}$ 1
        b := ((a $\wedge_{64}$ 1) $\ll_{64}$ 63) $\vee_{64}$ b
        a := a $\gg_{64}$ 1
        a := (current_random_bit $\ll_{64}$ 63) $\vee_{64}$ a
    end for
    return answer

24: **end function**

25: **function** POLYNOMIAL(a, b)

26: **return** $b \oplus_{64} (a \gg_{64} 23) \oplus_{64} (a \gg_{64} 25) \oplus_{64} (a \gg_{64} 63)$ ▷ This is irreducible and primitive polynomial: $x^{128} \oplus_{128} x^{41} \oplus_{128} x^{39} \oplus_{128} x \oplus_{128} 1$

27: **end function**

使用的自主设计的非线性反馈移位寄存器的函数如下:

Algorithm 2 OPC core algorithm - NLFSR

1: Define variable state: $state \leftarrow \left[a, b, c, d\right]$

2: where $a, b, c, d \in [0, 2^{64} - 1]$

3: **function** __RANDOM__BITS__($number, select$) ▷ Compute pseudo-random bit sequences in binary

4: **Input**: a number *number* and an integer *select* in the range [0, 8].

5: **Output**: a new value for *number*.

6: $result := \neg_{64}(number) + 1$

7: **if** $select = 0$ **then**

8: $result := result \wedge_{64} (2^{23} \vee_{64} 2^{10} \vee_{64} 2^{9} \vee_{64} 2^{8} \vee_{64} 2^{6} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 1)$

9: **else if** $select = 1$ **then**

10: $result := result \wedge_{64} (2^{54} \vee_{64} 2^{10} \vee_{64} 2^{9} \vee_{64} 2^{8} \vee_{64} 2^{7} \vee_{64} 2^{6} \vee_{64} 2^{5} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 2^{2})$

11: **else if** $select = 2$ **then**

12: $result := result \wedge_{64} (2^{47} \vee_{64} 2^{11} \vee_{64} 2^{10} \vee_{64} 2^{8} \vee_{64} x^{5} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 1)$

13: **else if** $select = 3$ **then**

14: $result := result \wedge_{64} (2^{30} \vee_{64} 2^{9} \vee_{64} 2^{8} \vee_{64} 2^{7} \vee_{64} 2^{5} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 2^{2})$

15: **else if** $select = 4$ **then**

16: $result := result \wedge_{64} (2^{63} \vee_{64} 2^{12} \vee_{64} 2^{9} \vee_{64} 2^{8} \vee_{64} 2^{5} \vee_{64} 2^{2})$

17: **else if** $select = 5$ **then**

18: $result := result \wedge_{64} (2^{26} \vee_{64} 2^{10} \vee_{64} 2^{3} \vee_{64} 2^{2} \vee_{64} 2 \vee_{64} 1)$

19: **else if** $select = 6$ **then**

20: $result := result \wedge_{64} (2^{6} \vee_{64} 1)$

21: **else if** $select = 7$ **then**

22: $result := result \wedge_{64} (2^{15} \vee_{64} 2^{10} \vee_{64} 2^{7} \vee_{64} 2^{5} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 2^{2} \vee_{64} 2^{1} \vee_{64} 1)$

23: **else**

24: $result := result \wedge_{64} (2^{41} \vee_{64} 2^{11} \vee_{64} 2^{10} \vee_{64} 2^{8} \vee_{64} 2^{6} \vee_{64} 2^{5} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 2^{2} \vee_{64} 2^{1})$

25: **end if**

26: **end function**

27: **function** RANDOM__BITS($number, select, bit$)

28: $number :=$ __RANDOM__BITS__($number \wedge 1$, $select$) ▷ I have combined different degrees of linear feedback shift registers here, They form a nonlinear feedback shift register, and the numbers generated by mixing these states are not predictable

29: $number := number \gg_{64} 1$

30: **return** *number*

31: **end function**

32: **function** INITIALIZE(*seed*)

33: **if** $seed \neq 0$ **then**

34: $a \leftrightarrow state_0$

35: $b \leftrightarrow state_1$

36: $c \leftrightarrow state_2$

37: $d \leftrightarrow state_3$

38: $a := seed$

39: $b := seed \boxtimes_{64} 2 \boxplus_{64} 1$

40: $c := seed \boxtimes_{64} 3 \boxplus_{64} 2$

41: $d := seed \boxtimes_{64} 4 \boxplus_{64} 3$

42: $a := a \boxplus_{64} ((b \oplus_{64} c) \oplus_{64} (\neg d))$

43: $b := b \boxminus_{64} ((b \wedge_{64} d) \vee_{64} a)$

44: $c := c \boxplus_{64} ((d \oplus_{64} a) \oplus_{64} (\neg b))$

45: $d := d \boxminus_{64} ((a \vee_{64} b) \wedge_{64} c)$

46: $state_3 := d \times (seed \ll_{64} 48) \wedge_{64} 4294967295$

47: $state_2 := c \times (seed \ll_{64} 32) \wedge_{64} 4294967295$

48: $state_1 := b \times (seed \ll_{64} 16) \wedge_{64} 4294967295$

49: $state_0 := a \times (seed) \wedge_{64} 4294967295$

50: **for** $round = 128$ **to** 1, $round := round - 1$ **do**

51: $c := state_2 \oplus_{64}$ RANDOM_BITS$(a, ((a \gg_{64} 6 \oplus_{64} b) \oplus_{64} d \oplus_{64} seed) \bmod 9, b \wedge_{64} 1)$

52: $d := state_3 \oplus_{64}$ RANDOM_BITS$(b, ((b \ll_{64} 57 \oplus_{64} a) \oplus_{64} c \oplus_{64} seed) \bmod 9, a \wedge_{64} 1)$

53: $a := state_0 \oplus_{64}$ RANDOM_BITS$(c, ((c \gg_{64} 24 \oplus_{64} d) \oplus_{64} b \oplus_{64} seed) \bmod 9, d \wedge_{64} 1)$

54: $b := state_1 \oplus_{64}$ RANDOM_BITS$(d, ((d \ll_{64} 37 \oplus_{64} c) \oplus_{64} a \oplus_{64} seed) \bmod 9, c \wedge_{64} 1)$

55: $bit := (a \wedge_{64} 1) \oplus_{64} (b \wedge_{64} 1) \oplus_{64} (c \wedge_{64} 1) \oplus_{64} (d \wedge_{64} 1)$

56: $temporary\_state \leftarrow (a \oplus_{64} b) \wedge_{64} c \vee_{64} d$

57: $seed := (seed \ggg_{64} 49) \times_{64} (state_0 \lll_{64} 13)$

58: $state_0 := state_1$

59: $state_1 := state_2$

60: $state_2 := state_3$

61: $state_3 := temporary\_state$

62: **if** temporary_state $\wedge_{64} 1 = 1$ **then**

63: seed' := seed $\vee_{64}$ (bit $\ll_{64} 63$)

64: **else if** temporary_state $\wedge_{64} 1 = 0$ **then**

65: seed' := seed $\vee_{64}$ (bit $\wedge_{64} 1$)

66: **end if**

67: **end for**

```
    end if
    return random_numbers
end function

function generate_chaotic_number( 𝔽_2^64 execute_count)                ▷ This is big version
    fibonacci_bits := Bits64(123581321345589144)
    pi_bits := Bits64 ((π − 3) × 10^64)
    euler_bits := Bits64 ((e − 2) × 10^64)
    gold_ratio_bits := Bits64 ((ϕ − 1) × 10^64)
    if execute_count ≥ 8 then
        AA ↔ state_0
        BB ↔ state_1
        CC ↔ state_2
        DD ↔ state_3
        answer := 0
        bit := 0
        for round = 0 to execute_count − 1, round := round + 1 do
            bit := (AA ⊕_64 BB ⊕_64 CC ⊕_64 DD) ∧_64 1
            answer := answer ≪_64 1
            answer := answer ∨_64 bit
            if HammingWeights(answer) ∧_64 1 ≠ 0 then
                answer := answer ⊕_64 pi_bits
            else
                bytes0 := Bits64ToBytes(answer)
                if (answer ⊕_64 BB) ∧_64 1 = 1 then
                    sequence_bytes := fibonacci_bits
                else
                    sequence_bytes := gold_ratio_bits
                end if
                repeat
                    bytes0 := GaloisFiniteField256_Multiplication(bytes0, sequence_bytes)   ▷ bytes0_index ×_GF sequence_bytes_index
                until executed 8 count
                answer := answer ⊕_64 Bits64FromBytes(bytes0)
            end if
            if HammingWeights(CC) ∧_64 1 = 0 then
                bytes1 := Bits64ToBytes(CC)
                if (answer ⊕_64 DD) ∧_64 1 = 1 then
                    sequence_bytes := euler_bits
                else
```

```
106:                 sequence_bytes := pi_bits
107:             end if
108:             repeat
109:                 bytes1 := GaloisFiniteField256_Multiplication(bytes1, sequence_bytes) ▷ bytes1_index ×_GF sequence_bytes_index
110:             until executed 8 count
111:             CC := CC ⊕64 Bits64FromBytes(bytes1)
112:             if CC ∧64 1 = 0 then
113:                 CC := CC ⊕64 fibonacci_bits
114:             end if
115:         else
116:             CC ⟵ CC ⊕64 (gold_ratio_bits ⊕64 answer)
117:             if (CC ∧64 1) ≠ 0 then
118:                 CC ⟵ CC ⊕64 pi_bits
119:             end if
120:         end if
121:         if (round mod 2) = 0 then
122:             random_number := ((answer ≫64 17) ⊕64 BB)
123:             AA := (AA ∧64 DD)
124:             if AA = 0 then
125:                 AA := AA + (CC × 2)
126:             end if
127:             answer := answer ⊕64 random_bits(AA, random_number mod 9, (DD ∧64 1) ⊕64 bit)
128:             DD := (DD ∧64 AA)
129:             if DD = 0 then
130:                 DD := DD - (BB × 2)
131:             end if
132:         else
133:             BB := BB ⊕64 ( (answer ⊕64 AA) ⋙64 (DD - CC) mod 64)
134:             CC := CC ⊕64 (BB ≪64 (DD + AA) mod 64)
135:             DD := DD ⊕64 (CC ≪64 (BB + AA) mod 64)
136:             AA := AA ⊕64 ( (answer ⊕64 DD) ⋘64 (BB - CC) mod 64)
137:             aa,bb := PseudoHadamardForwardTransform(AA, BB)
138:             if aa = 0 then
139:                 aa := bit
140:             else if bb = 0 then
141:                 bb := bit
142:             end if
143:             cc,dd := PseudoHadamardBackwardTransform(CC, DD)
```

144: **if** cc = 0 **then**
145: cc := bit
146: **else if** dd = 0 **then**
147: dd := bit
148: **end if**
149: AA := AA $\oplus_{64}$ aa
150: BB := BB $\oplus_{64}$ bb
151: CC := CC $\oplus_{64}$ cc
152: DD := DD $\oplus_{64}$ dd
153: aa,bb,cc,dd := 0
154: answer := answer $\oplus_{64}$ (AA $\oplus_{64}$ BB $\oplus_{64}$ CC $\oplus_{64}$ DD)
155: **end if**
156: **end for**
157: **end if**
158: **return** answer $\oplus_{64}$ ((answer $\ll_{64}$ 17) $\vee_{64}$ (answer $\gg_{64}$ 42))
159: **end function**

160: **function unpredictable_bits**($\mathbb{F}_2^{64}$ base_number, $\mathbb{F}_2^{64}$ number_bits) ▷ **This is little version**
161: $answer = base\_number$
162: $current\_random\_bit = 0$
163: $current\_random\_bits = \{0, 0, 0, 0 | \forall element \in \mathbb{F}_2^8\}$
164: **for** $round\_counter = 0$ **to** $number\_bits - 1$, $round\_counter := round\_counter + 1$ **do**
165: $current\_random\_bit := ((state_0 \oplus_{64} state_1 \oplus_{64} state_2 \oplus_{64} state_3) \gg_{64} 63) \wedge_{64} 1$
166: $answer := answer \ll_{64} 1$ ▷ Discard the highest bit of the answer random number, the lowest bit is complemented by '0'
167: $answer := answer \vee_{64} current_random_bit$ ▷ The answer random number is 0 or 1
168: $state_0$:= RANDOM_BITS($state_0$, $(state_3 \oplus_{64} state_2) \pmod 9$, current_random_bit)
169: $current\_random\_bits_0 := current\_random\_bits_0 \oplus_{64} (state_0 \wedge_{64} 1)$ ▷ Only one binary random bit is switched
170: $state_1$:= RANDOM_BITS($state_1$, $(state_2 \oplus_{64} state_1) \pmod 9$, current_random_bit)
171: $current\_random\_bits_1 := current\_random\_bits_2 \oplus_{64} (state_1 \wedge_{64} 1)$ ▷ Only one binary random bit is switched
172: $state_2$:= RANDOM_BITS($state_2$, $(state_1 \oplus_{64} state_0) \pmod 9$, current_random_bit)
173: $current\_random\_bits_2 := current\_random\_bits_2 \oplus_{64} (state_2 \wedge_{64} 1)$ ▷ Only one binary random bit is switched
174: $state_3$:= RANDOM_BITS($state_3$, $(state_0 \oplus_{64} state_3) \pmod 9$, current_random_bit)
175: $current\_random\_bits_3 := current\_random\_bits_3 \oplus_{64} (state_3 \wedge_{64} 1)$ ▷ Only one binary random bit is switched
176: $value_a \rightarrow current\_random\_bits_0 \vee_{64} current\_random\_bits_1$
177: $value_b \rightarrow current\_random\_bits_1 \wedge_{64} current\_random\_bits_2$

178: $value_c \to current\_random\_bits_2 \vee_{64} current\_random\_bits_3$

179: $value_d \to current\_random\_bits_3 \wedge_{64} current\_random\_bits_0$ ▷ The temporary values

180: $current\_random\_bit := value_a \oplus_{64} value_b \oplus_{64} value_c \oplus_{64} value_d$ ▷ This is Nonlinear boolean function

181: $answer := answer \ll_{64} 1$ ▷ Discard the highest bit of the answer random number, the lowest bit is complemented by '0'

182: $answer := answer \vee_{64} current_random_bit$ ▷ The answer random number is 0 or 1

183: $value\_a \to state_0 \pmod 4$

184: $value\_b \to state_1 \pmod 4$

185: $value\_c \to state_2 \pmod 4$

186: $value\_d \to state_3 \pmod 4$ ▷ The temporary values

187: SWAP($current\_random\_bits_{value\_a}$, $current\_random\_bits_3$)

188: SWAP($current\_random\_bits_{value\_b}$, $current\_random\_bits_3$)

189: SWAP($current\_random\_bits_{value\_c}$, $current\_random\_bits_3$)

190: SWAP($current\_random\_bits_{value\_d}$, $current\_random\_bits_3$) ▷ Pseudo Shuffle the elements of current_random_bits array

191: $state_1 := state_1 \gg_{64} 1$

192: $state_1 := state_1 \vee_{64} ((state_0 \wedge_{64} 1) \ll_{64} 63)$

193: $state_2 := state_2 \gg_{64} 1$

194: $state_2 := state_2 \vee_{64} ((state_1 \wedge_{64} 1) \ll_{64} 63)$

195: $state_3 := state_3 \gg_{64} 1$

196: $state_3 := state_3 \vee_{64} ((state_2 \wedge_{64} 1) \ll_{64} 63)$;

197: $state_0 := state_0 \gg_{64} 1$

198: $state_0 := state_0 \vee_{64} ((state_3 \wedge_{64} 1) \ll_{64} 63)$ ▷ Get the lowest bit of the bit sequence according to the current state and set that bit to the highest bit of the next state

199: **end for**

200: **return** answer

201: **end function**

使用的混沌理论系统的公式如下:

gravity_coefficient = 9.8

$$\theta_1' := \frac{-\text{gravity\_coefficient} \times (2 \times mass_1 + mass_2) \times \sin(\theta_1) - mass_2 \times \text{gravity\_coefficient} \times \sin(\theta_1 - 2 \times \theta_2)}{length_1 \times (2 \times mass_1 + mass_2 - mass_2 \times \cos(2 \times \theta_1 - 2 \times \theta_2))}$$
$$- \frac{2 \times \sin(\theta_1 - \theta_2) \times mass_2 \times ({\theta_2}^2 \times length_2) + ({\theta_1}^2 \times length_1 \times \cos(\theta_1 - \theta_2))}{length_1 \times (2 \times mass_1 + mass_2 - mass_2 \times \cos(2 \times \theta_1 - 2 \times \theta_2))}$$
$$\theta_2' := \frac{2 \times \sin(\theta_1' - \theta_2) \times \left[{\theta_1'}^2 \times length_1 \times (mass_1 + mass_2)\right]}{length_2 \times (2 \times mass_1 + mass_2 - mass_2 \times \cos(2 \times \theta_1' - 2 \times \theta_2))}$$
$$+ \frac{\text{gravity\_coefficient} \times (mass_1 + mass_2) \times \cos(\theta_1') + \left[{\theta_2}^2 \times length_2 \times mass_2 \cos(\theta_1' - \theta_2)\right]}{length_2 \times (2 \times mass_1 + mass_2 - mass_2 \times \cos(2 \times \theta_1' - 2 \times \theta_2))}$$

4.2 Workflow detail - Round funtion

本节深入探讨 OaldresPuzzle_Cryptic 算法中使用的轮函数的复杂性, 以及相关公式的实现. 此外, 一对字节替换框被用来建立一个数据替换层, 提供扩散性、混淆和非线性规律性. 需要注意的是, 这方面不属于我们对 Lai-Massey 方案框架的研究范围, 而是对该框架最终结果的一种适应.

对于 EncryptionWithOPC 和 DecryptionWithOPC, 2 个轮函数的实现, 我们可以简单的分成 2 种结构.

Algorithm 3 OPC core algorithm - The encrytion and decryption

Require: None

Ensure: None

1: **function EncryptionWithOPC**($PlainDataVetcor$)
2: **repeat**
3: **EncrytionByLaiMasseyFramework**($PlainDataVetcor, RoundSubkeys$)
4: **ForwardBytesSubstitution**($PlainDataVetcor$)
5: **until** executed 16 round
6: **end function**

7: **function DecryptionWithOPC**($CipherDataVector$)
8: **repeat**
9: **BackwardBytesSubstitution**($CipherDataVector$)
10: **DecrytionByLaiMasseyFramework**($CipherDataVector, RoundSubkeys$)
11: **until** executed 16 round
12: **end function**

EncryptionWithOPC 和 DecryptionWithOPC 函数的实现还没有完成, 因为 GenerateSubKeys 和 GenerateRoundSubKeys 函数必须在 Lai-Massey 方案框架完全建立之前最终完成. 然而, 我们将首先利用两个内部函数的实现来研究上述两个函数, 即 EncrytionByLaiMasseyFramework 和 DecrytionByLaiMasseyFramework. 一旦完成, 我们将继续实现 GenerateSubKeys 和 GenerateRoundSubKeys 函数.

我们提出的方案与 Lai-Massey 方案在结构上有相似之处, 主要区别在于 F 和 H 函数的排序. 如果读者需要复习一下这个框架的工作原理, 我们将引导他们注意题为 (已知的现有对称加密和解密框架及比较) 的章节.

Algorithm 4 OPC core algorithm - Round functions use a Modified lai–massey scheme

Require: $WordDatas \in \mathbb{F}_2^{64}$, $WordKeyMaterial \in \mathbb{F}_2^{64}$

Ensure: Updated $WordData$

1: The SecureRoundSubkeyGeneratationModule is class, The Instance Object Alias Name is SRSGM
2: $LeftWordData \in \mathbb{F}_2^{32}$ and $RightWordData \in \mathbb{F}_2^{32}$ from the RoundFunction

function EncrytionByLaiMasseyFramework($WordData$, $WordKeyMaterial$)
    if Data endian order is big **then**
        ByteSwap(WordData)
    end if
    $\{LeftWordData, RightWordData\}$ = **Split**($WordData$)
    $TransformKey$ = SRSGM.**CrazyTransformAssociatedWord**($LeftWordData \oplus_{32} RightWordData, Wo$
    $LeftWordData := LeftWordData \oplus_{32} TransformKey$
    $RightWordData := RightWordData \oplus_{32} TransformKey$
    $\{LeftWordData, RightWordData\}$:= SRSGM.**ForwardTransform**($LeftWordData, RightWordData$)
    $WordData$:= **Concatenate**($LeftWordData, RightWordData$)
    if Data endian order is big **then**
        ByteSwap($WordData$)
    end if
end function

function DecrytionByLaiMasseyFramework($WordData$, $WordKeyMaterial$)
    if Data endian order is big **then**
        ByteSwap(WordData)
    end if
    $\{LeftWordData, RightWordData\}$ = **Split**($WordData$)
    $\{LeftWordData, RightWordData\}$:= SRSGM.**BackwardTransform**($LeftWordData, RightWordDat$
    $TransformKey$ = SRSGM.**CrazyTransformAssociatedWord**($LeftWordData \oplus_{32} RightWordData, Wo$
    $LeftWordData := LeftWordData \oplus_{32} TransformKey$
    $RightWordData := RightWordData \oplus_{32} TransformKey$
    $WordData$:= **Concatenate**($LeftWordData, RightWordData$)
    if Data endian order is big **then**
        ByteSwap($WordData$)
    end if
end function

除了上述两个函数外, 我们还将解决另外两个内部函数的实现, 即 ForwardBytesSubstitution 和 BackwardBytesSubstitution. 这些函数需要四个字节置换盒, 其中包括两组前向和后向的密码学上的稳健的非线性函数. 然后, 字节置换盒数据被用来定义一个能够安全置换字节数据的函数.

Algorithm 5 OPC algorithm - The bytes data secure substitution layer

Require: $EachRoundDatas$ is byte array, $EachRoundDatas \in \{\mathbb{F}_2^8\}$

Ensure: Updated $EachRoundDatas$

The StateDataWorker is class, The Instance Object Alias Name is SDW
using ForwardSubstitutionBox0 ▷ AES Forward SubstitutionBox Modified
using BackwardSubstitutionBox0 ▷ AES Backward SubstitutionBox Modified

using ForwardSubstitutionBox1 ▷ China ZUC Stream Cipher Forward SubstitutionBox

using BackwardSubstitutionBox1 ▷ China ZUC Stream Cipher Backward SubstitutionBox

function SDW.**ForwardBytesSubstitution**(*EachRoundDatas*)
    if EachRoundDatas.size() is not a multiple of 8 **then**
        return
    end if
    for Index = 0; Index < EachRoundDatas.size(); Index = Index + 8 **do**
        $EachRoundDatas_{Index} := ForwardSubstitutionBox1_{EachRoundDatas_{Index}}$
        $EachRoundDatas_{Index+1} := ForwardSubstitutionBox0_{EachRoundDatas_{Index+1}}$
        $EachRoundDatas_{Index+2} := BackwardSubstitutionBox1_{EachRoundDatas_{Index+2}}$
        $EachRoundDatas_{Index+3} := BackwardSubstitutionBox0_{EachRoundDatas_{Index+3}}$
        $EachRoundDatas_{Index+4} := ForwardSubstitutionBox0_{EachRoundDatas_{Index+4}}$
        $EachRoundDatas_{Index+5} := BackwardSubstitutionBox1_{EachRoundDatas_{Index+5}}$
        $EachRoundDatas_{Index+6} := ForwardSubstitutionBox0_{EachRoundDatas_{Index+6}}$
        $EachRoundDatas_{Index+7} := BackwardSubstitutionBox1_{EachRoundDatas_{Index+7}}$
    end for
end function

function SDW.**BackwardBytesSubstitution**(*EachRoundDatas*)
    if EachRoundDatas.size() is not a multiple of 8 **then**
        return
    end if
    for Index = 0; Index < EachRoundDatas.size(); Index = Index + 8 **do**
        $EachRoundDatas_{Index} := BackwardSubstitutionBox1_{EachRoundDatas_{Index}}$
        $EachRoundDatas_{Index+1} := BackwardSubstitutionBox0_{EachRoundDatas_{Index+1}}$
        $EachRoundDatas_{Index+2} := ForwardSubstitutionBox1_{EachRoundDatas_{Index+2}}$
        $EachRoundDatas_{Index+3} := ForwardSubstitutionBox0_{EachRoundDatas_{Index+3}}$
        $EachRoundDatas_{Index+4} := BackwardSubstitutionBox0_{EachRoundDatas_{Index+4}}$
        $EachRoundDatas_{Index+5} := ForwardSubstitutionBox1_{EachRoundDatas_{Index+5}}$
        $EachRoundDatas_{Index+6} := BackwardSubstitutionBox0_{EachRoundDatas_{Index+6}}$
        $EachRoundDatas_{Index+7} := ForwardSubstitutionBox1_{EachRoundDatas_{Index+7}}$
    end for
end function ▷ Similar to AES bytes substitution step, where Index is the row and $EachRoundDatas_{Index}$ is the column

---

```
        //ForwardSubstitutionBox0, BackwardSubstitutionBox0, ForwardSubstitutionBox1, BackwardSubstitutionBox1
        //These ForwardSubstitutionBox0_Index, BackwardSubstitutionBox0_Index ∈ F_2^8 and all is static constant

        //Primitive polynomial degree is 8
        //Generator: x^8 ⊕8 x^7 ⊕8 x^6 ⊕8 x^5 ⊕8 x^4 ⊕8 x^3 ⊕8 1
        ForwardSubstitutionBox0
        {
```

```
        0x7F, 0x84, 0x01, 0x2B, 0xC3, 0x4E, 0x55, 0x58, 0x21, 0x62, 0x64, 0xF1, 0xE9, 0x81, 0x6F, 0x6D,
        0x50, 0x71, 0x72, 0x61, 0xF2, 0xA9, 0xBB, 0xD7, 0xB7, 0xF8, 0x00, 0x74, 0xF4, 0x05, 0x76, 0x6E,
        0xE8, 0x8F, 0x78, 0x34, 0xF9, 0x28, 0xF3, 0x54, 0x3A, 0x6C, 0x14, 0x02, 0x1D, 0x7B, 0xA8, 0x5E,
        0x98, 0x25, 0x3F, 0x87, 0xC0, 0x8A, 0x79, 0xE2, 0xBA, 0xE5, 0xC1, 0x24, 0xFB, 0x13, 0xF7, 0xCF,
        0xB4, 0x12, 0x07, 0x95, 0xFC, 0x8D, 0xDA, 0x5B, 0x3C, 0x53, 0xD4, 0x09, 0x39, 0x4B, 0xEA, 0x27,
        0xDD, 0xB9, 0x75, 0xB6, 0x49, 0xD5, 0x42, 0x3E, 0xCD, 0xF6, 0x7D, 0x5F, 0x17, 0xA1, 0xEF, 0xD3,
        0x0F, 0x0B, 0x52, 0x2F, 0xDC, 0x46, 0x80, 0x30, 0xA0, 0x99, 0x06, 0x56, 0xFF, 0xE0, 0xB1, 0xB0,
        0x1E, 0x60, 0x32, 0x8E, 0xA3, 0x67, 0x51, 0x7E, 0xBE, 0x15, 0xCA, 0x8C, 0x3B, 0xAB, 0xA4, 0x16,
        0x19, 0xA7, 0xC9, 0x4D, 0x43, 0x94, 0x89, 0xCC, 0x3D, 0x70, 0x85, 0x59, 0x2E, 0xD1, 0xEE, 0x9E,
        0x5D, 0x8B, 0x69, 0x77, 0x29, 0xD2, 0x44, 0x63, 0x5C, 0x82, 0x65, 0x45, 0x36, 0x1A, 0xD0, 0x88,
        0xAD, 0xD6, 0x9F, 0xAC, 0x7A, 0x4F, 0x9B, 0x41, 0xE7, 0x47, 0x2A, 0xB2, 0xE1, 0x0D, 0xDF, 0x97,
        0x26, 0xC5, 0x38, 0x6B, 0xFD, 0x2D, 0xEC, 0xF5, 0xC8, 0x10, 0x93, 0x20, 0x37, 0x9A, 0xAA, 0xA2,
        0xC4, 0xB3, 0xC6, 0xA6, 0x6A, 0xDB, 0x57, 0x0A, 0xAE, 0x9C, 0xE3, 0x08, 0x03, 0x1F, 0xD8, 0x2C,
        0x90, 0xB5, 0x0C, 0x83, 0x40, 0x23, 0x68, 0x91, 0xBC, 0x22, 0x33, 0x66, 0x18, 0xAF, 0x1B, 0xCE,
        0x4C, 0xE4, 0xF0, 0xFE, 0x5A, 0x0E, 0x04, 0x35, 0x11, 0xBD, 0x73, 0xFA, 0xEB, 0x9D, 0x7C, 0x48,
        0x1C, 0xD9, 0x4A, 0xC2, 0xA5, 0xC7, 0x86, 0xED, 0xDE, 0xBF, 0x96, 0xB8, 0x92, 0x31, 0xCB, 0xE6
    }

    //Primitive polynomial degree is 8
    //Generator: x^8 ⊕8 x^7 ⊕8 x^6 ⊕8 x^5 ⊕8 x^4 ⊕8 x^3 ⊕8 1
    BackwardSubstitutionBox0
    {
        0x1A, 0x02, 0x2B, 0xCC, 0xE6, 0x1D, 0x6A, 0x42, 0xCB, 0x4B, 0xC7, 0x61, 0xD2, 0xAD, 0xE5, 0x60,
        0xB9, 0xE8, 0x41, 0x3D, 0x2A, 0x79, 0x7F, 0x5C, 0xDC, 0x80, 0x9D, 0xDE, 0xF0, 0x2C, 0x70, 0xCD,
        0xBB, 0x08, 0xD9, 0xD5, 0x3B, 0x31, 0xB0, 0x4F, 0x25, 0x94, 0xAA, 0x03, 0xCF, 0xB5, 0x8C, 0x63,
        0x67, 0xFD, 0x72, 0xDA, 0x23, 0xE7, 0x9C, 0xBC, 0xB2, 0x4C, 0x28, 0x7C, 0x48, 0x88, 0x57, 0x32,
        0xD4, 0xA7, 0x56, 0x84, 0x96, 0x9B, 0x65, 0xA9, 0xEF, 0x54, 0xF2, 0x4D, 0xE0, 0x83, 0x05, 0xA5,
        0x10, 0x76, 0x62, 0x49, 0x27, 0x06, 0x6B, 0xC6, 0x07, 0x8B, 0xE4, 0x47, 0x98, 0x90, 0x2F, 0x5B,
        0x71, 0x13, 0x09, 0x97, 0x0A, 0x9A, 0xDB, 0x75, 0xD6, 0x92, 0xC4, 0xB3, 0x29, 0x0F, 0x1F, 0x0E,
        0x89, 0x11, 0x12, 0xEA, 0x1B, 0x52, 0x1E, 0x93, 0x22, 0x36, 0xA4, 0x2D, 0xEE, 0x5A, 0x77, 0x00,
        0x66, 0x0D, 0x99, 0xD3, 0x01, 0x8A, 0xF6, 0x33, 0x9F, 0x86, 0x35, 0x91, 0x7B, 0x45, 0x73, 0x21,
        0xD0, 0xD7, 0xFC, 0xBA, 0x85, 0x43, 0xFA, 0xAF, 0x30, 0x69, 0xBD, 0xA6, 0xC9, 0xED, 0x8F, 0xA2,
        0x68, 0x5D, 0xBF, 0x74, 0x7E, 0xF4, 0xC3, 0x81, 0x2E, 0x15, 0xBE, 0x7D, 0xA3, 0xA0, 0xC8, 0xDD,
        0x6F, 0x6E, 0xAB, 0xC1, 0x40, 0xD1, 0x53, 0x18, 0xFB, 0x51, 0x38, 0x16, 0xD8, 0xE9, 0x78, 0xF9,
        0x34, 0x3A, 0xF3, 0x04, 0xC0, 0xB1, 0xC2, 0xF5, 0xB8, 0x82, 0x7A, 0xFE, 0x87, 0x58, 0xDF, 0x3F,
        0x9E, 0x8D, 0x95, 0x5F, 0x4A, 0x55, 0xA1, 0x17, 0xCE, 0xF1, 0x46, 0xC5, 0x64, 0x50, 0xF8, 0xAE,
        0x6D, 0xAC, 0x37, 0xCA, 0xE1, 0x39, 0xFF, 0xA8, 0x20, 0x0C, 0x4E, 0xEC, 0xB6, 0xF7, 0x8E, 0x5E,
        0xE2, 0x0B, 0x14, 0x26, 0x1C, 0xB7, 0x59, 0x3E, 0x19, 0x24, 0xEB, 0x3C, 0x44, 0xB4, 0xE3, 0x6C
    }

    ForwardSubstitutionBox1
    {
        0x55, 0xC2, 0x63, 0x71, 0x3B, 0xC8, 0x47, 0x86, 0x9F, 0x3C, 0xDA, 0x5B, 0x29, 0xAA, 0xFD, 0x77,
        0x8C, 0xC5, 0x94, 0x0C, 0xA6, 0x1A, 0x13, 0x00, 0xE3, 0xA8, 0x16, 0x72, 0x40, 0xF9, 0xF8, 0x42,
        0x44, 0x26, 0x68, 0x96, 0x81, 0xD9, 0x45, 0x3E, 0x10, 0x76, 0xC6, 0xA7, 0x8B, 0x39, 0x43, 0xE1,
        0x3A, 0xB5, 0x56, 0x2A, 0xC0, 0x6D, 0xB3, 0x05, 0x22, 0x66, 0xBF, 0xDC, 0x0B, 0xFA, 0x62, 0x48,
        0xDD, 0x20, 0x11, 0x06, 0x36, 0xC9, 0xC1, 0xCF, 0xF6, 0x27, 0x52, 0xBB, 0x69, 0xF5, 0xD4, 0x87,
        0x7F, 0x84, 0x4C, 0xD2, 0x9C, 0x57, 0xA4, 0xBC, 0x4F, 0x9A, 0xDF, 0xFE, 0xD6, 0x8D, 0x7A, 0xEB,
        0x2B, 0x53, 0xD8, 0x5C, 0xA1, 0x14, 0x17, 0xFB, 0x23, 0xD5, 0x7D, 0x30, 0x67, 0x73, 0x08, 0x09,
        0xEE, 0xB7, 0x70, 0x3F, 0x61, 0xB2, 0x19, 0x8E, 0x4E, 0xE5, 0x4B, 0x93, 0x8F, 0x5D, 0xDB, 0xA9,
        0xAD, 0xF1, 0xAE, 0x2E, 0xCB, 0x0D, 0xFC, 0xF4, 0x2D, 0x46, 0x6E, 0x1D, 0x97, 0xE8, 0xD1, 0xE9,
        0x4D, 0x37, 0xA5, 0x75, 0x5E, 0x83, 0x9E, 0xAB, 0x82, 0x9D, 0xB9, 0x1C, 0xE0, 0xCD, 0x49, 0x89,
        0x01, 0xB6, 0xBD, 0x58, 0x24, 0xA2, 0x5F, 0x38, 0x78, 0x99, 0x15, 0x90, 0x50, 0xB8, 0x95, 0xE4,
        0xD0, 0x91, 0xC7, 0xCE, 0xED, 0x0F, 0xB4, 0x6F, 0xA0, 0xCC, 0xF0, 0x02, 0x4A, 0x79, 0xC3, 0xDE,
        0xA3, 0xEF, 0xEA, 0x51, 0xE6, 0x6B, 0x18, 0xEC, 0x1B, 0x2C, 0x80, 0xF7, 0x74, 0xE7, 0xFF, 0x21,
        0x5A, 0x6A, 0x54, 0x1E, 0x41, 0x31, 0x92, 0x35, 0xC4, 0x33, 0x07, 0x0A, 0xBA, 0x7E, 0x0E, 0x34,
        0x88, 0xB1, 0x98, 0x7C, 0xF3, 0x3D, 0x60, 0x6C, 0x7B, 0xCA, 0xD3, 0x1F, 0x32, 0x65, 0x04, 0x28,
        0x64, 0xBE, 0x85, 0x9B, 0x2F, 0x59, 0x8A, 0xD7, 0xB0, 0x25, 0xAC, 0xAF, 0x12, 0x03, 0xE2, 0xF2
    }

    BackwardSubstitutionBox1
```

```
        {
            0x17, 0xA0, 0xBB, 0xFD, 0xEE, 0x37, 0x43, 0xDA, 0x6E, 0x6F, 0xDB, 0x3C, 0x13, 0x85, 0xDE, 0xB5,
            0x28, 0x42, 0xFC, 0x16, 0x65, 0xAA, 0x1A, 0x66, 0xC6, 0x76, 0x15, 0xC8, 0x9B, 0x8B, 0xD3, 0xEB,
            0x41, 0xCF, 0x38, 0x68, 0xA4, 0xF9, 0x21, 0x49, 0xEF, 0x0C, 0x33, 0x60, 0xC9, 0x88, 0x83, 0xF4,
            0x6B, 0xD5, 0xEC, 0xD9, 0xDF, 0xD7, 0x44, 0x91, 0xA7, 0x2D, 0x30, 0x04, 0x09, 0xE5, 0x27, 0x73,
            0x1C, 0xD4, 0x1F, 0x2E, 0x20, 0x26, 0x89, 0x06, 0x3F, 0x9E, 0xBC, 0x7A, 0x52, 0x90, 0x78, 0x58,
            0xAC, 0xC3, 0x4A, 0x61, 0xD2, 0x00, 0x32, 0x55, 0xA3, 0xF5, 0xD0, 0x0B, 0x63, 0x7D, 0x94, 0xA6,
            0xE6, 0x74, 0x3E, 0x02, 0xF0, 0xED, 0x39, 0x6C, 0x22, 0x4C, 0xD1, 0xC5, 0xE7, 0x35, 0x8A, 0xB7,
            0x72, 0x03, 0x1B, 0x6D, 0xCC, 0x93, 0x29, 0x0F, 0xA8, 0xBD, 0x5E, 0xE8, 0xE3, 0x6A, 0xDD, 0x50,
            0xCA, 0x24, 0x98, 0x95, 0x51, 0xF2, 0x07, 0x4F, 0xE0, 0x9F, 0xF6, 0x2C, 0x10, 0x5D, 0x77, 0x7C,
            0xAB, 0xB1, 0xD6, 0x7B, 0x12, 0xAE, 0x23, 0x8C, 0xE2, 0xA9, 0x59, 0xF3, 0x54, 0x99, 0x96, 0x08,
            0xB8, 0x64, 0xA5, 0xC0, 0x56, 0x92, 0x14, 0x2B, 0x19, 0x7F, 0x0D, 0x97, 0xFA, 0x80, 0x82, 0xFB,
            0xF8, 0xE1, 0x75, 0x36, 0xB6, 0x31, 0xA1, 0x71, 0xAD, 0x9A, 0xDC, 0x4B, 0x57, 0xA2, 0xF1, 0x3A,
            0x34, 0x46, 0x01, 0xBE, 0xD8, 0x11, 0x2A, 0xB2, 0x05, 0x45, 0xE9, 0x84, 0xB9, 0x9D, 0xB3, 0x47,
            0xB0, 0x8E, 0x53, 0xEA, 0x4E, 0x69, 0x5C, 0xF7, 0x62, 0x25, 0x0A, 0x7E, 0x3B, 0x40, 0xBF, 0x5A,
            0x9C, 0x2F, 0xFE, 0x18, 0xAF, 0x79, 0xC4, 0xCD, 0x8D, 0x8F, 0xC2, 0x5F, 0xC7, 0xB4, 0x70, 0xC1,
            0xBA, 0x81, 0xFF, 0xE4, 0x87, 0x4D, 0x48, 0xCB, 0x1E, 0x1D, 0x3D, 0x67, 0x86, 0x0E, 0x5B, 0xCE
        }
```

本文中规定的两对字节替换盒的数据和顺序的重要性. 这是由于这些字节替换盒的实现, 本质上是一个数学上被严格证明的非线性函数, 如果在没有彻底理解基本数学原理的情况下, 不能证明修改后的实现具有同等非线性粒度, 不要试图修改或优化该实现. 为了更深入地了解这个问题, 关于密码学安全的替换盒的实现的所有评测标准, 详细请见文献. [1] [8] [16]

作者善意地提醒读者, 文件中的源代码块并不意味着可以作为实际代码来编译和执行. 相反, 它们作为一种视觉表现来解释各种概念和想法. 作者强调了彻底阅读所有附带的解释和数学公式的重要性, 以便充分理解所提出的概念. 如果读者不考虑源代码块的全部内容, 就可能表明错过或忽略了一个细节. 可能会误会本文的意思. 另外, 如果本文中有任何错误, 欢迎联系作者的邮箱.

4.3 Workflow detail - Key generation system

对于 GenerateSubkeys 和 GenerateRoundSubkeys 函数的实现，我们仍有很多工作要做

接下来，我们需要定义一个名为（CommonStateData）的数据结构，它将在后面解释密钥生成系统时使用

首先，这个数据结构需要使用 2 个不可变的 $\mathbb{F}_2^{32}$ 整数，第一个整数是 DataBlockSize，它代表数据块的元素大小；第二个整数是 KeyBlockSize，它代表主密钥块的元素大小

$DataBlockSize \pmod{16} = 0 \; and \; not(DataBlockSize < 2)$ 原因: (128 Bit ÷ 8 Bit(1 Byte) = 16 Bytes, 16 Bytes ÷ 8 Bytes (1 QuadWords = 2 QuadWords)

$KeyBlockSize \pmod{32} = 0 \; and \; not(KeyBlockSize < 4)$ 原因: (256 Bit ÷ 8 Bit(1 Byte) = 32 Bytes, 32 Bytes ÷ 8 Bytes (1 QuadWords = 4 QuadWords)

$KeyBlockSize \geq DataBlockSize \; and \; KeyBlockSize \pmod{DataBlockSize} = 0$

为了满足未来抗量子密码的要求，建议 DataBlockSize 大于或等于 4，KeyBlockSize 大于或等于 8；因为 4 个元素的 64 比特等于 256 比特，然后 8 个元素的 64 比特等于 512 比特

此外，在这个数据结构中，我们之前在（密钥生成系统的前置算法）中提到的三种伪随机数生成器算法需要这三种算法数据结构对象的实例，即 LFSR、NLFSR 和 SDP

此外，我们需要定义两个状态数据，分别代表该数据结构中子密钥数据和轮密钥数据的状态矩阵。这两个矩阵都是行和列一致的方形矩阵。它们的长度由基于前面要求的两个不可变的整数的简单计算决定

下面是如何计算分块大小的:

$KeyRows = KeyBlockSize \times 2, KeyColumns = KeyBlockSize \times 2$

现在定义 2 个矩阵:

$$\forall RandomQuadWordMatrix_{Row,Column} \in \mathbb{F}_2^{64}$$

$$\textbf{RandomQuadWordMatrix}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$\forall TransformedSubkeyMatrix_{Row,Column} \in \mathbb{F}_2^{64}$$

$$\textbf{TransformedSubkeyMatrix}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

此外，这个数据结构包括一个伯努利分布的对象实例，它负责调整伪随机数发生器结果的位级概率（通常是 64 位的输入和输出数据）。产生 0 和 1 比特的概率被设置为 50%

我们可以用数学方法表达如下:

$$BernoulliDistributionObject(x, probability) = \begin{cases} probability & \text{if } x = 1 \\ 1 - probability & \text{if } x = 0 \end{cases}$$

这里，x 是一个二元随机变量，其值为 0 或 1，而 $probability$ 表示该变量取值为 1 的概率

接下来，在数据结构中定义了一个向量，用来存储用于定义前面提到的两个矩阵的行和列的 Index。这些 Index 用于访问矩阵，存储在这个向量中的数据将在某个时候被洗牌

$$\textbf{MatrixOffsetWithRandomIndices} := \{0, 1, 2, 3, 4, 5, 6, 7 \dots KeyBlockSize \times 2 - 1 | \forall element \in \mathbb{F}_2^{32}\}$$

这就是我们使用的洗牌算法

注意: 与原始的 Fisher-Yates Shuffle 算法相比，Flavor Water 伪随机发生器产生的结果不能直接利用，而是必须先经过一个特定范围的统一整数分布，然后才能利用

Algorithm 6 Fisher-Yates Shuffle

Require: Random-access iterators $first$ and $last$ that denote the range to be shuffled, and a uniform random bit generator $functionRNG$
Ensure: The range $[first, last)$ is shuffled in place

```
function ShuffleRangeData(first, last, functionRNG)
    distance = last − first
    for index = 1 to distance − 1 do
        random_index = UniformIntegerDistribution(functionRNG, Param) ▷ Param is UniformIntegerDistributionParam(min: 0, max: index)
        swap(Datas_{first+index}, Datas_{first+random_index})
    end for
    return next(first, last)
end function
```

然后在数据结构中定义一个数组，用来存储主密钥数据。在算法运行了 N 轮之后，这个向量数据将被修改。当我们讨论算法的最外层封装函数时，将详细解释它被修改的伪代码

$$\textbf{WordKeyDataVector} = \{0_0, 0_1, 0_2, 0_3 \dots 0_{KeyBlockSize-1} | \forall element \in \mathbb{F}_2^{64}\}$$

最后，在数据结构中，定义了一个空向量来存储伪随机性的初始数据集，该向量由其他字节数据向量填充。虽然它的长度是可变的，但其他字节数据向量的长度必须是 $DataBlockSize \pmod 8 = 0$.

$$\forall WordDataInitialVector_{Row} \in \mathbb{F}_2^{32}$$

$$\textbf{WordDataInitialVector} = \begin{bmatrix} \end{bmatrix}$$

这里规定一下 IntegerToBytes 和 IntegerFromBytes 的函数的意义, 以后不再重复.

而且本文接下来类似的结构也会使用到.

$ExampleBytes = \textbf{IntegerToBytes}(ExampleInteger)$ 函数输入成倍数关系数量的字节数据, 然后输出这个字节数据所对应倍数大小的整数数据.(而且要注意计算机的字节序)

比如说把 8 个 8 比特字节数据转成 1 个 64 比特的整数数据, 如果输入输出双方都是数组, 那就重复执行这个操作

$ExampleInteger = \textbf{IntegerFromBytes}(ExampleBytes)$ 函数输入成倍数关系数量的整数数据, 然后输出这个整数数据所对应倍数大小的字节数据.(而且要注意计算机的字节序)

比如说把 1 个 64 比特整数数据转成 8 个 8 比特的字节数据, 如果输入输出双方都是数组, 那就重复执行这个操作

我们还需要定义矩阵和向量之间的运算符, 以便在之后的算法展示中使用.

其中 $+_{MATRIX}$ 代表矩阵加法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $-_{MATRIX}$ 代表矩阵减法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{MATRIX}$ 代表矩阵乘法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $+_{VECTOR}$ 表示向量加法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $-_{VECTOR}$ 代表向量减法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{VECTOR}$ 代表向量乘法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{\mathtt{VEW}}$ 代表向量元素相乘, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{\mathtt{MVE}}$ 代表矩阵-向量乘法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{SCALAR}$ 代表矩阵或向量与标量的乘法, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{KRONECKER}$ 代表克朗克积运算, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $\times_{DOT}$ 代表点积运算, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $NameMatrix^{Transpose}$ 表示求解 NameMatrix 的转置矩阵, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

其中 $NameMatrix^{HermitianTranspose}$ 表示求解 NameMatrix 的共轭转置矩阵, 但这个 $result$ 仍然属于伽罗瓦有限域 2^{bit_count}

4.3.1 Pre-process stage: Use seed initialize PRNGs then fill MatrixA with initial vector bytes

提供 3 个（不同/相同的）种子，用于初始化 3 个伪随机数生成器

$$SeedValue \neq 0, SeedValue \in \mathbb{F}_2^{64}$$

$$CommonStateData.LFSR.\textbf{seed}(1\ or\ SeedValue \in \mathbb{F}_2^{64})$$

$$CommonStateData.NLFSR.\textbf{seed}(1\ or\ SeedValue \in \mathbb{F}_2^{64})$$

$$CommonStateData.SDP.\textbf{seed}(13249961062380153450\ \textbf{or}\ SeedValue \geq 10000000000\ \textbf{and}\ SeedValue \in \mathbb{F}_2^{64})$$

提供初始向量量数据 (注意：这个数据必须是独立的，不应该与明文、密文或主密钥有关)

$$WordDataInitialVector := \textbf{IntegerFromBytes}(BytesData)$$

$$WordDataInitialVector \xrightarrow[\textbf{ApplyWordDataInitialVector}(WordDataInitialVector)]{WordDataInitialVector_{row} \in \mathbb{F}_2^{32}} CommonStateData.MatrixA$$

接下来我们将详细展示算法中的 ApplyWordDataInitialVector 函数

Algorithm 7 Apply Word Data Initial Vector

```
function ApplyWordDataInitialVector(WordDataInitialVector)
    RandomQuadWordMatrix = ReferenceObject(CommonStateData.RandomQuadWordMatrix)   ▷ Initial sampling of Word data (Use 32Bit Word Data - Initial Vector)
    Word32Bit_ExpandedInitialVector = Word32Bit_ExpandKey(WordDataInitialVector)
    Index = Word32Bit_ExpandedInitialVector.size()
    MatrixRow = KeyRows from RandomQuadWordMatrix
    MatrixColumn = KeyColumns from RandomQuadWordMatrix
    Flag Use32BitData
    while MatrixRow > 0 do                                   ▷ Iterate through each column of the matrix in descending order
        while MatrixColumn > 0 do                            ▷ Iterate through each row of the matrix in descending order
            if Index = 0 then
                break
```

end if
$\mathbb{F}_2^{64} RandomValue = Word32Bit\_ExpandedInitialVector_{Index-1}$
$\mathbb{F}_2^{64} RotatedBits = (RandomValue \ll_{64} 7) \vee_{64} (RandomValue \gg_{64} 1)$
$Position \to \{MatrixRow - 1, MatrixColumn - 1\}$
$MatrixValue \leftrightarrow RandomQuadWordMatrix_{Position}$ ▷ Access the value reference from the state key MatrixA
$MatrixValue := RandomValue \oplus_{64} (RandomValue \wedge_{64} RotatedBits)$ ▷ Random bits
$MatrixValue := MatrixValue \oplus_{64} (1 \ll_{64} (RandomValue \pmod{64}))$ ▷ Switch bit
$RandomValue := RandomValue \boxplus_{64} MatrixValue$
$MatrixValue := MatrixValue \boxplus_{64} (2 \boxtimes_{64} RandomValue \boxplus_{64} MatrixValue)$
$Index := Index - 1$
$MatrixColumn := MatrixColumn - 1$
end while
$MatrixRow := MatrixRow - 1$
$MatrixColumn := KeyColumns\ from\ RandomQuadWordMatrix$
end while
if $MatrixRow = 0$ and $MatrixColumn = 0$ and $Index > 0$ **then**
$MatrixRow := KeyRows\ from\ RandomQuadWordMatrix$
$MatrixColumn := KeyColumns\ from\ RandomQuadWordMatrix$
goto Use32BitData
end if

$$\textbf{Word32Bit\_ExpandedInitialVector} := \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyBlockSize-1} \end{bmatrix}$$

end function

Algorithm 8 Word32Bit ExpandKey

Require: $NeedHashDataWords$ is a vector span view, each $element \in \mathbb{F}_2^{32}$, and $element$ is constant
Ensure: $ProcessedWordKeys$ is expanded keys vector, each $element \in \mathbb{F}_2^{32}$
function WORD32BIT_EXPANDKEY($NeedHashDataWords$)

$$\textbf{ProcessedWordKeys} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{NeedHashDataWords.size() \times 12-1} \end{bmatrix}$$

NeedHashDataIndex = 0
while NeedHashDataIndex < NeedHashDataWords.size() **do**
$\mathbb{F}_2^{32} RestructedWordKey$ = WORDBITRESTRUCT($NeedHashDataWords_{NeedHashDataIndex}$) ▷ Data word do bit reorganization
if Data endian order is big **then**
BYTESWAP(RestructedWordKey)
end if
$\mathbb{F}_2^{32} UpPartWord, DownPartWord, LeftPartWord, RightPartWord = 0$
$UpPartWord := (RestructedWordKey \gg_{32} 16)$ ▷ Data words do bit splitting: Reserve the **High 16 bits**
$DownPartWord := (RestructedWordKey \ll_{32} 16) \gg_{32} 16$ ▷ Data words do bit splitting: Reserve the **Low 16 bits**
$LeftPartWord := (RestructedWordKey \wedge_{32} 0xF000'0000) \vee_{32} ((RestructedWordKey \wedge_{32} 0x00F0'0000) \ll_{32} 4) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'F000) \ll_{32} 8) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'00F0) \ll_{32} 12)$ ▷ Data words do bit splitting: Concatenate all data at bit positions **28∼31, 20∼23, 12∼15, 4∼7**
$RightPartWord := ((RestructedWordKey \wedge_{32} 0x0F00'0000) \ll_{32} 4) \vee_{32} ((RestructedWordKey \wedge_{32} 0x000F'0000) \ll_{32} 8) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'0F00U) \ll_{32} 12) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'000F) \ll_{32} 14)$ ▷ Data words do bit splitting: Concatenate all data at bit positions **24∼27, 16∼19, 8∼11, 0∼3**
$\mathbb{F}_2^{32} DiffusionResult0, DiffusionResult1, DiffusionResult2, DiffusionResult3, DiffusionResult4, DiffusionResult5 = 0$
$DiffusionResult0 := UpPartWord \oplus_{32} DownPartWord$
$DiffusionResult1 := LeftPartWord \oplus_{32} RightPartWord$
$DiffusionResult2 := UpPartWord \oplus_{32} LeftPartWord$
$DiffusionResult3 := DownPartWord \oplus_{32} RightPartWord$
$DiffusionResult4 := UpPartWord \oplus_{32} RightPartWord$
$DiffusionResult5 := DownPartWord \oplus_{32} LeftPartWord$
$\mathbb{F}_2^{32} KeyIndex = 0$
while KeyIndex < ProcessedWordKeys.size() **do**
$\mathbb{F}_2^{32} Prime0, Prime1, Prime2, Prime3, Prime4, Prime5 = 0$
$\mathbb{F}_2^{32} Prime6, Prime7, Prime8, Prime9, Prime10, Prime11 = 0$
$Prime0 = 286331173$
$Prime1 = 3676758703$
$Prime2 = 4123665971$
$Prime3 = 3193679207$

$Prime4 = 339204479$
$Prime5 = 2017551733$
$Prime6 = 3451580309$
$Prime7 = 2711043323$
$Prime8 = 45676697$
$Prime9 = 1066195267$
$Prime10 = 4172536373$
$Prime11 = 3285900997$
$Key0 \leftrightarrow ProcessedWordKeys_{KeyIndex}, Key1 \leftrightarrow ProcessedWordKeys_{KeyIndex+1}$
$Key2 \leftrightarrow ProcessedWordKeys_{KeyIndex+2}, Key3 \leftrightarrow ProcessedWordKeys_{KeyIndex+3}$
$Key4 \leftrightarrow ProcessedWordKeys_{KeyIndex+4}, Key5 \leftrightarrow ProcessedWordKeys_{KeyIndex+5}$
$Key6 \leftrightarrow ProcessedWordKeys_{KeyIndex+6}, Key7 \leftrightarrow ProcessedWordKeys_{KeyIndex+7}$
$Key8 \leftrightarrow ProcessedWordKeys_{KeyIndex+8}, Key9 \leftrightarrow ProcessedWordKeys_{KeyIndex+9}$
$Key10 \leftrightarrow ProcessedWordKeys_{KeyIndex+10}, Key11 \leftrightarrow ProcessedWordKeys_{KeyIndex+11}$ ▷ Define: Key0,Key1,Key2,Key3,Key4,Key5,Key6,Key7,Key8 Key9,Key10,Key11 and are used as aliases for the following data references for accessing arrays
Key0 := Key0 $\oplus_{32}$ ((DiffusionResult0 $\lll_{32}$ 8 $\vee_{32}$ DiffusionResult4) $\boxplus_{32}$ Prime0)
Key1 := Key1 $\oplus_{32}$ ((DiffusionResult0 $\vee_{32}$ DiffusionResult4 $\ggg_{32}$ 24) $\boxminus_{32}$ Prime1)
Key2 := Key2 $\oplus_{32}$ ((DiffusionResult5 $\lll_{32}$ 16 $\vee_{32}$ DiffusionResult1) $\boxtimes_{32}$ Prime2)
Key3 := (DiffusionResult5 $\vee_{32}$ DiffusionResult1 $\ggg_{32}$ 16) (mod $Prime3$)
Key4 := Key4 $\oplus_{32}$ ((DiffusionResult2 $\lll_{32}$ 24 $\vee_{32}$ DiffusionResult3) $\boxtimes_{32}$ Prime4)
Key5 := Key5 $\oplus_{32}$ ((DiffusionResult2 $\vee_{32}$ DiffusionResult3 $\ggg_{32}$ 8) $\boxplus_{32}$ Prime5)
Key6 := (DiffusionResult0 $\ggg_{32}$ 24 $\vee_{32}$ DiffusionResult4) (mod $Prime6$)
Key7 := Key7 $\oplus_{32}$ ((DiffusionResult0 $\vee_{32}$ DiffusionResult4 $\lll_{32}$ 8) $\boxminus_{32}$ Prime7)
Key8 := Key8 $\oplus_{32}$ ((DiffusionResult5 $\ggg_{32}$ 16 $\vee_{32}$ DiffusionResult1) $\boxtimes_{32}$ Prime8)
Key9 := Key9 $\oplus_{32}$ ((DiffusionResult5 $\vee_{32}$ DiffusionResult1 $\lll_{32}$ 16) $\boxminus_{32}$ Prime9)
Key10 := (DiffusionResult2 $\ggg_{32}$ 8 $\vee_{32}$ DiffusionResult3) (mod $Prime10$)
Key11 := Key11 $\oplus_{32}$ ((DiffusionResult2 $\vee_{32}$ DiffusionResult3 $\lll_{32}$ 24) $\boxplus_{32}$ Prime11)
For all the elements in the array, move the loop to the right 1 time ▷ Example: {A,B,C,D,E,F,G,H,I,J,K,L} → {L,A,B,C,D,E,F,G,H,I,J,K}
DiffusionResult0 := DiffusionResult0 $\boxminus_{32}$ (Key0 $\vee_{32}$ Key11)
DiffusionResult5 := DiffusionResult5 $\boxplus_{32}$ (Key1 $\wedge_{32}$ Key10)
DiffusionResult1 := DiffusionResult1 $\boxminus_{32}$ (Key2 $\vee_{32}$ Key9)
DiffusionResult4 := DiffusionResult4 $\boxplus_{32}$ (Key3 $\wedge_{32}$ Key8)
DiffusionResult2 := DiffusionResult2 $\boxminus_{32}$ (Key4 $\vee_{32}$ Key7)
DiffusionResult3 := DiffusionResult3 $\boxplus_{32}$ (Key5 $\wedge_{32}$ Key6)
For all the elements in the array, move the loop to the right 1 time ▷ Example: {L,A,B,C,D,E,F,G,H,I,J,K} → {K,L,A,B,C,D,E,F,G,H,I,J}
DiffusionResult0 := WordBitRestruct(DiffusionResult0)
DiffusionResult1 := WordBitRestruct(DiffusionResult1)
DiffusionResult2 := WordBitRestruct(DiffusionResult2)
DiffusionResult3 := WordBitRestruct(DiffusionResult3)
DiffusionResult4 := WordBitRestruct(DiffusionResult4)
DiffusionResult5 := WordBitRestruct(DiffusionResult5)
KeyIndex = KeyIndex + 12
end while ▷ Data words do byte mixing and number expansions
$DiffusionResult0, DiffusionResult1, DiffusionResult2, DiffusionResult3, DiffusionResult4, DiffusionResult5 := 0$
$UpPartWord, DownPartWord, LeftPartWord, RightPartWord := 0$ ▷ Temporary data zeroing to prevent analysis
NeedHashDataIndex = NeedHashDataIndex + 1
return ProcessedWordKeys
end while
end function

Require: $WordKey \in \mathbb{F}_2^{32}$
Ensure: $WordKey$ after the single-bit restructuring
function WordBitRestruct($WordKey$)
$WordKey$:= SwapBits($WordKey, 0, 9$)
$WordKey$:= SwapBits($WordKey, 1, 18$)
$WordKey$:= SwapBits($WordKey, 2, 27$) ▷ Green Step 1
$WordKey$:= SwapBits($WordKey, 5, 28$)
$WordKey$:= SwapBits($WordKey, 6, 21$)

83: $WordKey := \text{SwapBits}(WordKey, 7, 14)$ ▷ Green Step 2
84: $WordKey := \text{SwapBits}(WordKey, 10, 24)$
85: $WordKey := \text{SwapBits}(WordKey, 11, 25)$
86: $WordKey := \text{SwapBits}(WordKey, 12, 30)$
87: $WordKey := \text{SwapBits}(WordKey, 13, 31)$ ▷ Orange Step
88: $WordKey := \text{SwapBits}(WordKey, 19, 4)$
89: $WordKey := \text{SwapBits}(WordKey, 20, 3)$ ▷ Red Step
90: $WordKey := \text{SwapBits}(WordKey, 17, 2)$
91: $WordKey := \text{SwapBits}(WordKey, 22, 5)$ ▷ Yellow Step
92: $WordKey := \text{SwapBits}(WordKey, 27, 15)$
93: $WordKey := \text{SwapBits}(WordKey, 28, 8)$ ▷ Blue Step
94: **return** $WordKey$
95: **end function**

96: **function** SwapBits($Word, BitPosition, BitPosition2$)
97: $BitMask := ((Word \gg_{32} BitPosition) \wedge_{32} 1) \oplus ((Word \gg_{32} BitPosition2) \wedge_{32} 1)$ ▷ Calculate the bit mask to swap the bits
98: **if** $BitMask = 0$ **then**
99: **return** $Word$ ▷ Return the word as it is if bits are same
100: **end if**
101: $BitMask := (BitMask \ll_{32} BitPosition) \vee_{32} (BitMask \ll_{32} BitPosition2)$ ▷ Create the bit mask to swap the bits
102: **return** $Word \oplus_{32} BitMask$ ▷ Return the swapped word
103: **end function**

在这个阶段完成后，我们将不再使用外部提供的初始矢量数据。在主密钥数据被分块之前，再将其分块到一个向量量视图或真实向量中，其长度每次都由 CommmonStateData 类决定。这被用来表示每个主密钥的分块数据

4.3.2 Work stage: Compute the key state MatrixA and MatrixB by using the MainKeyBlockData selection function

这实际上是 **GenerateSubkeys** 函数的实现，最外层的封装函数，它将是第一个使用该函数的函数，我们将在此详细讨论其流程
如果 MainKeyBlockData 的大小为空，那么只执行更新函数，否则先执行初始化函数，然后再执行更新函数

Initialization algorithm block: 使用主密钥的一大块数据和一个复杂的单向函数来改变矩阵

$$MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)$$
$$MainKeyBlockData_{Row} \in \mathbb{F}_2^{64}$$
$$WordKeyResistQC = \{0, 0, 0, 0, \ldots \mid \forall element \in \mathbb{F}_2^{64}\}$$
$$MainKeyBlockData \xrightarrow[SubkeyMatrixOperationObject.\textbf{InitializationState}(WordKeyResistQC)]{\textbf{LatticeCryptographyAndHash}(MainKeyBlockData,\ WordKeyResistQC)} MatrixA$$

接下来我们将详细展示算法中的 LatticeCryptographyAndHash 函数

Algorithm 9 Complex one-way functions using lattice cryptography and my sponge structure hash class

Require: $InputKeys$ is a vector span view, each $element \in \mathbb{F}_2^{64}$, and $element$ is constant
Ensure: $OutputKeys$ is a vector span view, each $element \in \mathbb{F}_2^{64}$
1: **function** LatticeCryptographyAndHash($InputKeys$, $OutputKeys$)
2: $SDP = \textbf{ReferenceObject}(CommonStateData.SDP)$
3: $HashMixedIntegerVector \in \mathbb{F}_2^{64}$
4:

$$\textbf{HashMixedIntegerVector} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

5: $HashMixedIntegerVector := InputKeys$
6:

$$\textbf{PseudoRandomNumberMatrix}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

7: **for** $Row = 0$ to $KeyRows - 1$ **do**

for $Column = 0$ to $KeyColumns - 1$ **do**

$PseudoRandomNumberMatrix_{Row,\ Column}$:= SDP(min : 0, max: 18446744073709551615)

end for

end for ▷ Fill the matrix with elements, each of which is an absolutely 64 bits data with of uniform pseudo-random

$HashMixedIntegerVector$:= SecureHash(PseudoRandomNumberMatrix, HashMixedIntegerVector)

$\mathbb{F}_2^{64} PrimeNumber = 18446744073709551557$

for $Index = 0$ to $HashMixedIntegerVector.size() - 1$ **do**

$a = InputKeys_{Index \pmod{InputKeys.size()}}$

$b = HashMixedIntegerVector_{Index}$

$c \leftrightarrow OutputKeys_{Index}$

if c = 0 **then**

c := if a + b ≥ PrimeNumber, then return a + b - PrimeNumber, else return a + b

else

$\mathbb{F}_2^{64} d = 0$

d := if a + b ≥ PrimeNumber, then return a + b - PrimeNumber, else return a + b

c := if c + c ≥ PrimeNumber, then return c + d - PrimeNumber, else return c + d

end if

$$\textbf{HashMixedIntegerVector} := \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

end for ▷ The original vector data and the hashed vector data are added with a large integer with a large modulus, and then become a hash-mixed vector

Ensure that the status vector is securely cleaned ▷ Fill zero to HashMixedIntegerVector

end function

Require: $KeyMatrix$, $KeyVector$ is a matrix vector, each $element \in \mathbb{F}_2^{64}$, and $element$ is constant

Ensure: $Hashed$ is a vector, each $element \in \mathbb{F}_2^{64}$

function SecureHash($KeyMatrix$, $KeyVector$)

$$\textbf{MA}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$\textbf{VA} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

$$\textbf{MB}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

$$\textbf{VB} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

for $Index = 0$ to $KeyRows \times KeyColumns - 1$ **do**

$\mathbb{F}_2^{64} value = KeyMatrix_{Index}$

$MA_{\{Index \div KeyColumns, Index \pmod{KeyColumns}\}} := value \gg_{64} 32$

$MB_{\{Index \div KeyColumns, Index \pmod{KeyColumns}\}} := value \wedge_{64} 0x00000000FFFFFFFF$

end for ▷ Matrix Element is 64 bits data, split into high and low 32 bits Data and stored as 64 bits data

for $Index = 0$ to $KeyRows - 1$ **do**

$\mathbb{F}_2^{64} value = KeyVector_{Index}$

$VA_{\{Index \div KeyColumns, Index \pmod{KeyColumns}\}} := value \gg_{64} 32$

$VB_{\{Index \div KeyColumns, Index \pmod{KeyColumns}\}} := value \wedge_{64} 0x00000000FFFFFFFF$

end for ▷ Vector Element is 64 bits data, split into high and low 32 bits Data and stored as 64 bits data

$ResultA = MA \times_{\mathbb{MVE}} VA$ ▷ Matrix-vector multiplication using split 32-bit data in stored 64-bit data without any computational overflow

$ResultB = MB \times_{\mathbb{MVE}} VB$ ▷ Matrix-vector multiplication using split 32-bit data in stored 64-bit data without any computational overflow

$SpanVectorA \leftrightarrow \{ResultA_0, ResultA_1, ResultA_2 \ldots ResultA_{ResultA.size()-1}\}$

$SpanVectorB \leftrightarrow \{ResultB_0, ResultB_1, ResultB_2 \ldots ResultA_{ResultB.size()-1}\}$

$$\textbf{CustomHashed} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

$$\textbf{Hashed} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}$$

49: $HashObject$ = **MySpongeStructureHashClass**($HashBitSize$: ($KeyRows \times 64$) ÷ 2) ▷ The implementation of this class, we have the actual code to refer to in the appendix of this paper.
50: HashObject.Execute(SpanVectorA, $\{CustomHashed_0 \dots CustomHashed_{KeyRows \div 2}\}$)
51: HashObject.Execute(SpanVectorB, $\{CustomHashed_{KeyRows \div 2} \dots CustomHashed_{KeyRows \div 2}\}$)
52: $\mathbb{F}_2^{64} PrimeNumber = 18446744073709551557$
53: $\mathbb{F}_2^{64} HashedValue = 0$
54: **for** $Index = 0$ to $KeyRows - 1$ **do**
55: $HashedValue = (ResultA_{Index} \pmod{PrimeNumber}) + (ResultB_{Index} \pmod{PrimeNumber}) \pmod{PrimeNumber}$ ▷ (A + B) mod PrimeNumber = ((A mod PrimeNumber) + (B mod PrimeNumber)) mod PrimeNumber
56: $Hashed_{Index} := (CustomHashed_{Index} \pmod{PrimeNumber}) + (HashedValue \pmod{PrimeNumber}) \pmod{PrimeNumber}$
57: **end for**▷ After splitting, the hashed and matrix-vector multiplication results on both sides are combined using this addition. If there is a calculation overflow, it is guaranteed to use a large prime number for modulo, and the result will not overflow.
58: Ensure that the status matrix and vector is securely cleaned ▷ Fill zero to MA, MB, VA, VB, ResultA, ResultB
59: **return** Hashed
60: **end function**

接下来我们将详细展示算法中的 InitializationState 函数

这需要定义一个长度为 256 字节的向量，作为一个非线性函数的实现来利用。这个向量包含可变值，类似于之前的字节替换盒，而不是静态不变的值。因此，它可以代表字节替换盒的当前状态

然而，在提出上述功能之前，我们还必须建立一个专门的算法，能够从当前的字节替换盒数据中生成新的替换盒数据。这个算法应该采用伪随机数发生器产生的数字和一个基于位操作原理的数据结构，即线段树。该数据结构的代码实现包含在本论文的附录中

而且，这个功能采用了最初由中国研究人员开发的 ZUC 流密码算法。然而，我们对原始算法进行了修改，我们将在稍后阶段比较和对比两种 ZUC 算法的区别

```
        //MaterialSubstitutionBox0, MaterialSubstitutionBox1
        //These MaterialSubstitutionBox0_Index, MaterialSubstitutionBox1_Index ∈ F_2^8 and all is vector element

        /*
            This byte-substitution box: Strict avalanche criterion is satisfied !
            ByteDataSecurityTestData Transparency Order Is: 7.81299
            ByteDataSecurityTestData Nonlinearity Is: 94
            ByteDataSecurityTestData Propagation Characteristics Is: 8
            ByteDataSecurityTestData Delta Uniformity Is: 10
            ByteDataSecurityTestData Robustness Is: 0.960938
            ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 9.29288
            ByteDataSecurityTestData Absolute Value Indicatorer Is: 120
            ByteDataSecurityTestData Sum Of Square Value Indicator Is: 244160
            ByteDataSecurityTestData Algebraic Degree Is: 8
            ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
        */
        MaterialSubstitutionBox0
        {
            0xF4, 0x53, 0x75, 0x96, 0xBE, 0x6F, 0x66, 0x11, 0x80, 0xC8, 0x5C, 0xDF, 0xF7, 0xAE, 0xC6, 0x93,
            0xF1, 0x2F, 0x5F, 0x47, 0xB8, 0xF2, 0x71, 0x30, 0x1E, 0x87, 0x32, 0x0A, 0xCA, 0x6E, 0x16, 0xCB,
            0x65, 0x2C, 0x35, 0x0D, 0x8C, 0x1C, 0x3A, 0xA8, 0xC4, 0x84, 0xC7, 0x46, 0x0B, 0xCE, 0xFC, 0xB1,
            0x62, 0x5A, 0x59, 0x6D, 0x42, 0x3D, 0xA9, 0xAA, 0xD6, 0x14, 0x88, 0x02, 0xE8, 0x82, 0x9A, 0x7E,
            0xF6, 0x9E, 0x43, 0x27, 0x33, 0x4C, 0x57, 0x01, 0x8B, 0x25, 0x79, 0xB0, 0x18, 0xB9, 0xB2, 0x9D,
            0xAF, 0x0E, 0xD4, 0xE1, 0x2E, 0x0C, 0xDB, 0x8E, 0x1D, 0xE2, 0x00, 0x51, 0xB3, 0xF3, 0x7F, 0x99,
            0xA5, 0xCD, 0x77, 0xB4, 0xD9, 0x61, 0x76, 0x70, 0x40, 0x9F, 0x5E, 0xFF, 0x4D, 0xF9, 0x86, 0xAB,
            0xD3, 0x41, 0xB5, 0x2B, 0xA1, 0x39, 0x63, 0xC9, 0x6C, 0x73, 0x9B, 0xBB, 0x7B, 0xD0, 0xAD, 0x7C,
            0xEE, 0xDE, 0xF8, 0xD8, 0xB6, 0xED, 0x98, 0x19, 0xFA, 0x8F, 0x92, 0xAC, 0x12, 0xC2, 0x05, 0xCF,
            0xC0, 0xEF, 0x08, 0xFE, 0xDD, 0x50, 0x23, 0x4B, 0xC3, 0x15, 0xE5, 0xD5, 0x3E, 0xE0, 0x2A, 0x52,
            0x95, 0x44, 0x72, 0x56, 0x0F, 0x1B, 0xF5, 0x90, 0xE3, 0x58, 0x69, 0x8D, 0x48, 0x26, 0xD2, 0xA2,
            0x7A, 0x38, 0x49, 0xEC, 0x13, 0x67, 0x07, 0x81, 0xE9, 0xD1, 0x34, 0x36, 0x85, 0xA3, 0x5D, 0x22,
            0x24, 0x6B, 0xBA, 0x37, 0x7D, 0xBF, 0x6A, 0x2D, 0x45, 0x3C, 0x55, 0x5B, 0x74, 0xF0, 0xDA, 0x83,
            0xDC, 0x4A, 0x91, 0x31, 0x97, 0xA4, 0xE6, 0x1A, 0x1F, 0x4F, 0xC5, 0x54, 0xFD, 0x17, 0x06, 0x89,
            0x60, 0xA6, 0xB7, 0x3B, 0xA7, 0xFB, 0x78, 0x94, 0xBD, 0xA0, 0xE7, 0xD7, 0xEB, 0x21, 0xE4, 0xEA,
            0x09, 0xC1, 0x03, 0xBC, 0xCC, 0x68, 0x20, 0x04, 0x28, 0x9C, 0x4E, 0x3F, 0x10, 0x29, 0x8A, 0x64,
        }

        /*
```

```
            This byte-substitution box: Strict avalanche criterion is satisfied !
            ByteDataSecurityTestData Transparency Order Is: 7.80907
            ByteDataSecurityTestData Nonlinearity Is: 94
            ByteDataSecurityTestData Propagation Characteristics Is: 8
            ByteDataSecurityTestData Delta Uniformity Is: 12
            ByteDataSecurityTestData Robustness Is: 0.953125
            ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 9.25523
            ByteDataSecurityTestData Absolute Value Indicatorer Is: 96
            ByteDataSecurityTestData Sum Of Square Value Indicator Is: 199424
            ByteDataSecurityTestData Algebraic Degree Is: 8
            ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
        */
        MaterialSubstitutionBox1
        {
            0x88, 0xB4, 0x21, 0xF9, 0xC9, 0xBC, 0x7C, 0x5D, 0xAB, 0x7D, 0x04, 0x69, 0x96, 0x8E, 0x00, 0x71,
            0x94, 0xB0, 0xFB, 0xE1, 0xD6, 0xA2, 0xD5, 0xE6, 0x74, 0x6C, 0xB9, 0x31, 0xAE, 0xDD, 0x49, 0x19,
            0x02, 0x75, 0x34, 0x33, 0x46, 0x0A, 0xA9, 0x54, 0x1F, 0x5F, 0xCA, 0x56, 0xD2, 0xD8, 0x41, 0xD9,
            0x0D, 0x47, 0xF0, 0xB3, 0x62, 0x8F, 0x52, 0x08, 0x3F, 0x4C, 0x84, 0x1C, 0xA8, 0x3A, 0x7A, 0xCE,
            0x22, 0x2C, 0x1B, 0x4D, 0xFA, 0x30, 0x2F, 0x80, 0x3B, 0x55, 0x91, 0x05, 0x61, 0x03, 0x64, 0x87,
            0xFF, 0xE0, 0x26, 0xBE, 0x68, 0x0E, 0x50, 0xC3, 0x29, 0x42, 0x6F, 0x2B, 0x53, 0x79, 0xB5, 0x27,
            0x77, 0x97, 0x32, 0x38, 0x07, 0xBB, 0xF7, 0xF5, 0x28, 0x11, 0x36, 0x9B, 0x5C, 0x81, 0x65, 0x6A,
            0xEB, 0xE5, 0x17, 0xF4, 0x3C, 0xE9, 0x39, 0x58, 0xF8, 0x66, 0x15, 0xC6, 0xA4, 0xEA, 0xE2, 0xDF,
            0xCC, 0xFD, 0x3D, 0xEF, 0x1A, 0x24, 0x4A, 0xBF, 0xB6, 0x67, 0xF6, 0x45, 0xB7, 0x4B, 0xB2, 0x5E,
            0x60, 0x7F, 0x89, 0x76, 0xD4, 0x59, 0xE4, 0xAD, 0xCB, 0xA3, 0xFC, 0x7B, 0xBD, 0x35, 0x51, 0xC7,
            0xA0, 0xA1, 0x8C, 0x13, 0x83, 0xA5, 0xCF, 0x44, 0x95, 0xDE, 0x9E, 0xF3, 0x1D, 0x40, 0x2E, 0x0F,
            0x72, 0xD0, 0x6E, 0x8A, 0xAF, 0x6D, 0x16, 0xC1, 0xE7, 0x43, 0x8B, 0x9C, 0x4F, 0x82, 0x10, 0xDA,
            0x57, 0x0C, 0xCD, 0x63, 0x9F, 0xBA, 0x0B, 0x4E, 0x90, 0x93, 0xAA, 0xF2, 0xC0, 0x20, 0x14, 0x78,
            0xEE, 0xA7, 0x85, 0x3E, 0x5A, 0x2D, 0x01, 0xED, 0xC4, 0xAC, 0x25, 0x73, 0x5B, 0x98, 0x06, 0xEC,
            0xDC, 0x12, 0xB8, 0xD3, 0xD7, 0xC5, 0xE3, 0x9A, 0xF1, 0xD1, 0xE8, 0x6B, 0xB1, 0x48, 0xFE, 0x86,
            0x70, 0xA6, 0x9D, 0x18, 0xC2, 0x99, 0x1E, 0x09, 0x7E, 0x37, 0x2A, 0xDB, 0x8D, 0xC8, 0x23, 0x92,
        }
```

Algorithm 10 Line-segment tree use bitwise operation

Require: DataType is an integral data type and ArraySize is a power of 2

Ensure: A line-segment tree data structure

Nodes

function INITIALIZE($Size$)
  if Checks if $Size$ is an integral power of two = false **then**
    ProgramError
  end if
  $Nodes = \{0, 0, 0, 0, 0 \ldots 0_{Size-1}\}$
end function

function SET(Position) ▷ Increment the count at position Position by 1
  for CurrentNode = N $\vee$ Position; $CurrentNode \neq 0; CurrentNode := CurrentNode \gg 1$ **do**
    $Nodes_{CurrentNode} := Nodes_{CurrentNode} + 1$
  end for
end function

function GET(Order) ▷ Find the position of the Order-th smallest element
  $CurrentNode = 1$
  for $CurrentLeftSize = \text{N} \gg 1, LeftTotal = 0; CurrentLeftSize \neq 0; CurrentLeftSize := CurrentLeftSize \gg 1$ **do**
    $CurrentLeftCount = CurrentLeftSize - Nodes_{CurrentNode \ll 1}$
    if $LeftTotal + CurrentLeftCount > Order$ **then**
      $CurrentNode := CurrentNode \ll 1$
    else
      $CurrentNode := CurrentNode \ll 1 \vee 1$
      $LeftTotal := LeftTotal + CurrentLeftCount$
    end if

23: **end for**
24: **return** $CurrentNode \oplus N$
25: **end function**

26: **function** CLEAR() ▷ Set all counts to 0
27: $Nodes := \{0, 0, 0, 0, 0 \dots 0_{Size-1}\}$
28: **end function**

Algorithm 11 Regeneration material byte substitution box with use Pseudo-random number generator and line-segment tree

Require: $OldBox$ is a vector span view, from Substitution boxes, each $element \in \mathbb{F}_2^8$, and $element$ is constant
Ensure: $NewBox$ is a vector, each $element \in \mathbb{F}_2^8$
1: **function** REGENERATIONRANDOMMATERIALSUBSTITUTIONBOX($OldBox$)
2: $NLFSR =$ **ReferenceObject**($CommonStateData.NLFSR$)
3: $LineSegmentTreeObject = LineSegmentTree.Initialize(Size : 256)$
4: **NewBox** $= \{0_0, 0_1, 0_2, 0_3 \dots 0_{255} | \forall element \in \mathbb{F}_2^8\}$
5: $\mathbb{F}_2^{64} Index = 0, Index2 = 0$
6: **while** $Index < OldDataArraySize$ **and** $Index2 < NewDataArraySize$ **do**
7: **if** $Index = OldDataArraySize - 1$ **and** $OldDataBox_{Index} = LineSegmentTreeObject$.GET(0) **then**
8: **NewBox** $:= \{0_0, 0_1, 0_2, 0_3 \dots 0_{255} | \forall element \in \mathbb{F}_2^8\}$
9: LineSegmentTreeObject.CLEAR()
10: $Index := 0, Index2 := 0$
11: **end if**
12: $\mathbb{F}_2^{64} Order =$ NLFSR.GENERATE_CHAOTIC_NUMBER(8) $\pmod{OldBox.size() - Index}$
13: $\mathbb{F}_2^{64} Position =$ LINESEGMENTTREEOBJECT.GET($Order$)
14: **while** $OldBox_{Index} = Position$ **do**
15: $Order :=$ NLFSR.GENERATE_CHAOTIC_NUMBER(8) $\pmod{OldBox.size() - Index}$
16: $Position :=$ LINESEGMENTTREEOBJECT.GET($Order$)
17: **end while**
18: $NewBox_{Index2} = Position$
19: LINESEGMENTTREEOBJECT.SET($Position$)
20: Index := Index + 1, Index2 := Index2 + 1
21: **end while**
22: **return** NewBox
23: **end function**

我们之前提到，我们修改了 ZUC 流密码算法。然而，让我们首先介绍一下原始算法，然后讨论我们修改的部分。值得注意的是，修改后的 ZUC 流密码算法使用了前面提到的 2 个动态字节替换框，而内部寄存器初始化函数的差异是很大的

```
ZUC_Box0
{
    0x3E, 0x72, 0x5B, 0x47, 0xCA, 0xE0, 0x00, 0x33, 0x04, 0xD1, 0x54, 0x98, 0x09, 0xB9, 0x6D, 0xCB,
    0x7B, 0x1B, 0xF9, 0x32, 0xAF, 0x9D, 0x6A, 0xA5, 0xB8, 0x2D, 0xFC, 0x1D, 0x08, 0x53, 0x03, 0x90,
    0x4D, 0x4E, 0x84, 0x99, 0xE4, 0xCE, 0xD9, 0x91, 0xDD, 0xB6, 0x85, 0x48, 0x8B, 0x29, 0x6E, 0xAC,
    0xCD, 0xC1, 0xF8, 0x1E, 0x73, 0x43, 0x69, 0xC6, 0xB5, 0xBD, 0xFD, 0x39, 0x63, 0x20, 0xD4, 0x38,
    0x76, 0x7D, 0xB2, 0xA7, 0xCF, 0xED, 0x57, 0xC5, 0xF3, 0x2C, 0xBB, 0x14, 0x21, 0x06, 0x55, 0x9B,
    0xE3, 0xEF, 0x5E, 0x31, 0x4F, 0x7F, 0x5A, 0xA4, 0x0D, 0x82, 0x51, 0x49, 0x5F, 0xBA, 0x58, 0x1C,
    0x4A, 0x16, 0xD5, 0x17, 0xA8, 0x92, 0x24, 0x1F, 0x8C, 0xFF, 0xD8, 0xAE, 0x2E, 0x01, 0xD3, 0xAD,
    0x3B, 0x4B, 0xDA, 0x46, 0xEB, 0xC9, 0xDE, 0x9A, 0x8F, 0x87, 0xD7, 0x3A, 0x80, 0x6F, 0x2F, 0xC8,
    0xB1, 0xB4, 0x37, 0xF7, 0x0A, 0x22, 0x13, 0x28, 0x7C, 0xCC, 0x3C, 0x89, 0xC7, 0xC3, 0x96, 0x56,
    0x07, 0xBF, 0x7E, 0xF0, 0x0B, 0x2B, 0x97, 0x52, 0x35, 0x41, 0x79, 0x61, 0xA6, 0x4C, 0x10, 0xFE,
    0xBC, 0x26, 0x95, 0x88, 0x8A, 0xB0, 0xA3, 0xFB, 0xC0, 0x18, 0x94, 0xF2, 0xE1, 0xE5, 0xE9, 0x5D,
    0xD0, 0xDC, 0x11, 0x66, 0x64, 0x5C, 0xEC, 0x59, 0x42, 0x75, 0x12, 0xF5, 0x74, 0x9C, 0xAA, 0x23,
    0x0E, 0x86, 0xAB, 0xBE, 0x2A, 0x02, 0xE7, 0x67, 0xE6, 0x44, 0xA2, 0x6C, 0xC2, 0x93, 0x9F, 0xF1,
    0xF6, 0xFA, 0x36, 0xD2, 0x50, 0x68, 0x9E, 0x62, 0x71, 0x15, 0x3D, 0xD6, 0x40, 0xC4, 0xE2, 0x0F,
    0x8E, 0x83, 0x77, 0x6B, 0x25, 0x05, 0x3F, 0x0C, 0x30, 0xEA, 0x70, 0xB7, 0xA1, 0xE8, 0xA9, 0x65,
    0x8D, 0x27, 0x1A, 0xDB, 0x81, 0xB3, 0xA0, 0xF4, 0x45, 0x7A, 0x19, 0xDF, 0xEE, 0x78, 0x34, 0x60
}

ZUC_Box1
{
    0x55, 0xC2, 0x63, 0x71, 0x3B, 0xC8, 0x47, 0x86, 0x9F, 0x3C, 0xDA, 0x5B, 0x29, 0xAA, 0xFD, 0x77,
```

```
                0x8C, 0xC5, 0x94, 0x0C, 0xA6, 0x1A, 0x13, 0x00, 0xE3, 0xA8, 0x16, 0x72, 0x40, 0xF9, 0xF8, 0x42,
                0x44, 0x26, 0x68, 0x96, 0x81, 0xD9, 0x45, 0x3E, 0x10, 0x76, 0xC6, 0xA7, 0x8B, 0x39, 0x43, 0xE1,
                0x3A, 0xB5, 0x56, 0x2A, 0xC0, 0x6D, 0xB3, 0x05, 0x22, 0x66, 0xBF, 0xDC, 0x0B, 0xFA, 0x62, 0x48,
                0xDD, 0x20, 0x11, 0x06, 0x36, 0xC9, 0xC1, 0xCF, 0xF6, 0x27, 0x52, 0xBB, 0x69, 0xF5, 0xD4, 0x87,
                0x7F, 0x84, 0x4C, 0xD2, 0x9C, 0x57, 0xA4, 0xBC, 0x4F, 0x9A, 0xDF, 0xFE, 0xD6, 0x8D, 0x7A, 0xEB,
                0x2B, 0x53, 0xD8, 0x5C, 0xA1, 0x14, 0x17, 0xFB, 0x23, 0xD5, 0x7D, 0x30, 0x67, 0x73, 0x08, 0x09,
                0xEE, 0xB7, 0x70, 0x3F, 0x61, 0xB2, 0x19, 0x8E, 0x4E, 0xE5, 0x4B, 0x93, 0x8F, 0x5D, 0xDB, 0xA9,
                0xAD, 0xF1, 0xAE, 0x2E, 0xCB, 0x0D, 0xFC, 0xF4, 0x2D, 0x46, 0x6E, 0x1D, 0x97, 0xE8, 0xD1, 0xE9,
                0x4D, 0x37, 0xA5, 0x75, 0x5E, 0x83, 0x9E, 0xAB, 0x82, 0x9D, 0xB9, 0x1C, 0xE0, 0xCD, 0x49, 0x89,
                0x01, 0xB6, 0xBD, 0x58, 0x24, 0xA2, 0x5F, 0x38, 0x78, 0x99, 0x15, 0x90, 0x50, 0xB8, 0x95, 0xE4,
                0xD0, 0x91, 0xC7, 0xCE, 0xED, 0x0F, 0xB4, 0x6F, 0xA0, 0xCC, 0xF0, 0x02, 0x4A, 0x79, 0xC3, 0xDE,
                0xA3, 0xEF, 0xEA, 0x51, 0xE6, 0x6B, 0x18, 0xEC, 0x1B, 0x2C, 0x80, 0xF7, 0x74, 0xE7, 0xFF, 0x21,
                0x5A, 0x6A, 0x54, 0x1E, 0x41, 0x31, 0x92, 0x35, 0xC4, 0x33, 0x07, 0x0A, 0xBA, 0x7E, 0x0E, 0x34,
                0x88, 0xB1, 0x98, 0x7C, 0xF3, 0x3D, 0x60, 0x6C, 0x7B, 0xCA, 0xD3, 0x1F, 0x32, 0x65, 0x04, 0x28,
                0x64, 0xBE, 0x85, 0x9B, 0x2F, 0x59, 0x8A, 0xD7, 0xB0, 0x25, 0xAC, 0xAF, 0x12, 0x03, 0xE2, 0xF2
        }
```

Algorithm 12 The Original ZUC Sequence/Stream Data cipher [14]

using ZUC_Box0
using ZUC_Box1

$StateDataRegister = \{0, 0\}, \forall element \in \mathbb{F}_2^{32}$

Require: ZUC 31-bit LFSR state
Ensure: Four 32-bit Words data

function BitRestructure() ▷ Note: Using the linear feedback shift register of the original ZUC algorithm, after initializing the register state and updating the register state, 128 bits can be extracted and composed of 4 words of data in the following manner, and the size of each word is 32 bits.
    StateWithLFSR $\in \mathbb{F}_2^{32}$, and size 16
    WordData = $(StateWithLFSR_{15} \wedge_{32}$ 0x7fff8000) $\vee_{32}$ (Is binary concatenate) $(StateWithLFSR_{14} \wedge_{32}$ 0x0000ffff)
    WordData1 = $(StateWithLFSR_{11} \wedge_{32}$ 0x0000ffff) $\vee_{32}$ (Is binary concatenate) $(StateWithLFSR_{9} \wedge_{32}$ 0x7fff8000)
    WordData2 = $(StateWithLFSR_{7} \wedge_{32}$ 0x0000ffff) $\vee_{32}$ (Is binary concatenate) $(StateWithLFSR_{5} \wedge_{32}$ 0x7fff8000)
    WordData3 = $(StateWithLFSR_{2} \wedge_{32}$ 0x0000ffff) $\vee_{32}$ (Is binary concatenate) $(StateWithLFSR_{0} \wedge_{32}$ 0x7fff8000)
    return $\{WordData, WordData1, WordData2, WordData3\}$
end function

function ApplySubstitutionBox($RegisterValue0$, $RegisterValue1$) ▷ Register data using non-linear data for byte substitution operation
    $Bytes0 \rightarrow (RegisterValue0 \gg_{32} 24) \wedge_{32} 0xFF$
    $Bytes1 \rightarrow (RegisterValue0 \gg_{32} 16) \wedge_{32} 0xFF$
    $Bytes2 \rightarrow (RegisterValue0 \gg_{32} 8) \wedge_{32} 0xFF$
    $Bytes3 \rightarrow RegisterValue0 \wedge_{32} 0xFF$
    $Bytes4 \rightarrow (RegisterValue1 \gg_{32} 24) \wedge_{32} 0xFF$
    $Bytes5 \rightarrow (RegisterValue1 \gg_{32} 16) \wedge_{32} 0xFF$
    $Bytes6 \rightarrow (RegisterValue1 \gg_{32} 8) \wedge_{32} 0xFF$
    $Bytes7 \rightarrow RegisterValue1 \wedge_{32} 0xFF$ ▷ Temporary values
    $StateDataRegister_0 := (ZUC\_Box0_{Bytes0} \ll_{32} 24) \vee_{32} (ZUC\_Box1_{Bytes1} \ll_{32} 16) \vee_{32} (ZUC\_Box0_{Bytes2} \ll_{32} 8) \vee_{32} ZUC\_Box1_{Bytes3}$
    $StateDataRegister_1 := (ZUC\_Box0_{Bytes4} \ll_{32} 24) \vee_{32} (ZUC\_Box1_{Bytes5} \ll_{32} 16) \vee_{32} (ZUC\_Box0_{Bytes6} \ll_{32} 8) \vee_{32} ZUC\_Box1_{Bytes7}$
end function

function GenerateKeyStream($WordMaterial$) ▷ Non-linear function for generating key streams
    if $WordMaterial.size() \neq 4$ **then**
        ProgramError
    end if
    $\mathbb{F}_2^{32} WordData = (WordMaterial_0 \oplus_{32} DataRegister_0) \boxplus_{32} DataRegister_1$
    $\mathbb{F}_2^{32} WordData1 = DataRegister_0 \boxplus_{32} WordMaterial_1$
    $\mathbb{F}_2^{32} WordData2 = DataRegister_1 \oplus_{32} WordMaterial_2$
    $\mathbb{F}_2^{32} WordDataA = \mathbf{WT_1}(RandomWordData1, RandomWordData2)$
    $\mathbb{F}_2^{32} WordDataB = \mathbf{WT_2}(RandomWordData1, RandomWordData2)$
    $StateDataRegister_0 := \mathbf{LT_1}(WordDataA)$
    $StateDataRegister_1 := \mathbf{LT_2}(WordDataB)$ ▷ The function of WT is to split binary data into two halves and concatenate them interleaved, The LT function is a linear transformation. ▷ $\mathbf{WT_1}(Word1, Word2) = (\mathbf{Low16BitOnly}(Word1) \ll_{32} 16) \vee_{32} (\mathbf{High16BitOnly}(Word2) \gg_{32} 16)$ ▷

$\mathbf{WT_2}(Word1, Word2) = (\mathbf{Low16BitOnly}(Word2) \lll_{32} 16) \vee_{32} (\mathbf{High16BitOnly}(Word1) \ggg_{32} 16)$ ▷

$\mathbf{LT_1}(Word) = Word \oplus_{32} (Word \lll_{32} 2) \oplus_{32} (Word \lll_{32} 10) \oplus_{32} (Word \lll_{32} 18) \oplus_{32} (Word \lll_{32} 24)$ ▷

$\mathbf{LT_2}(Word) = Word \oplus_{32} (Word \lll_{32} 8) \oplus_{32} (Word \lll_{32} 14) \oplus_{32} (Word \lll_{32} 22) \oplus_{32} (Word \lll_{32} 30)$

35: ApplySubstitutionBox($StateDataRegister_0, StateDataRegister_1$))
36: **return** WordData
37: **end function**

38: **function** KeyWithStreamCipher($WordMaterial$) ▷ WordMaterial $\in \mathbb{F}_2^{32}$, and size 4
39: $\{WordData, WordData1, WordData2, WordData3\}$ = **BitRestructure**()
40: **return GenerateKeyStream**($WordMaterial_0, WordMaterial_1, WordMaterial_2$) $\oplus_{32} WordMaterial_3$
41: **end function**

Algorithm 13 The Modified ZUC Sequence/Stream Data cipher

1: **using** MaterialSubstitutionBox0
2: **using** MaterialSubstitutionBox1

3: $StateDataRegister = \{0, 0\}, \forall element \in \mathbb{F}_2^{32}$

Require: LFSR, NLFSR, SDP
Ensure: Two 32-bit Words of State Data Register
4: **function** InitializeDataRegister() ▷ It takes input from three different objects, which are accessed through a CommonStateData, namely an LFSR (Linear Feedback Shift Register) object, an NLFSR (Non-Linear Feedback Shift Register) object, and an SDP (SimulateDoublePendulum) object. These objects generate chaotic numbers that are used as a basis for generating pseudo-random bits. The function also uses an array of two 32-bit state registers(StateDataRegister), to store the generated pseudo-random bits.The output of this function is two 32-bit numbers, which are generated by combining the generated pseudo-random bits using bitwise operations. The first 32-bit number is stored in StateValue0, and the second 32-bit number is stored in StateValue1.
5: $LFSR$ = **ReferenceObject**($CommonStateData.LFSR$)
6: $NLFSR$ = **ReferenceObject**($CommonStateData.NLFSR$)
7: SDP = **ReferenceObject**($CommonStateData.SDP$)
8: $StateValue0 \leftrightarrow StateDataRegister_0$
9: $StateValue1 \leftrightarrow StateDataRegister_1$
10: $\mathbb{F}_2^{64} BaseNumber$ = NLFSR.generate_chaotic_number(8) $\oplus_{64}$ SDP($min : 0, max : 18446744073709551615$)
11: $\mathbb{F}_2^{64} RandomNumber = 0$
12: **for** Round = 129 **to** 1, Round := Round - 1 **do**
13: $BaseNumber$:= NLFSR.unpredictable_bits($BaseNumber \pmod{18446744073709551615}, 64$) $\oplus_{64}$ LFSR()
14: **end for**
15: $RandomNumber$:= NLFSR.generate_chaotic_number(8) $\oplus_{64}$ ($\neg_{64}$(LFSR.generate_bits(63) $\oplus_{64}$ $BaseNumber$))
16: $StateValue0$:= **Hight32BitOnly**($RandomNumber$)
17: $StateValue1$:= **Low32BitOnly**($RandomNumber$)
18: $RandomNumber$:= 0
19: **end function**

20: **function** ApplySubstitutionBox($RegisterValue0$, $RegisterValue1$) ▷ Register data using non-linear data for byte substitution operation
21: The function definition is the same as the original ZUC sequence/stream data cipher, But change ZUC_Box0 to MaterialSubstitutionBox0 and change ZUC_Box1 to MaterialSubstitutionBox1
22: **end function**

23: **function** GenerateKeyStream($WordMaterial$) ▷ Non-linear function for generating key streams
24: The function definition is the same as the original ZUC sequence/stream data cipher
25: **end function**

26: **function** KeyWithStreamCipher($WordMaterial$)
27: The function definition is the same as the original ZUC sequence/stream data cipher
28: **end function**

在确保所有必要的准备工作都已完成之后，我们现在可以开始建立 InitializationState 函数了

Algorithm 14 InitializationState from the name of the class object in the author's code is SecureSubkeyGeneratationModuleObject

Require: $HashedKeys$ is a vector span view, from Complex one-way functions, each $element \in \mathbb{F}_2^{64}$, and $element$ is constant
Ensure: Change MatrixA, each $element \in \mathbb{F}_2^{64}$

```
using MaterialSubstitutionBox0
using MaterialSubstitutionBox1
using ModifiedZUC

function InitializationState(HashedKeys)
    BernoulliDistribution = ReferenceObject(CommonStateData.BernoulliDistributionObject)
    MatrixA = ReferenceObject(CommonStateData.RandomQuadWordMatrix)
    LFSR = ReferenceObject(CommonStateData.LFSR)
    ByteKeys = {∅|∀element ∈ F_2^8}
    ByteKeys := IntegerToBytes(HashedKeys)
    for ByteKey in Ranges(ByteKeys) do                ▷ For each element, Byte data substitution operation via material substitution box 0
        TemporaryByte = MaterialSubstitutionBox0_ByteKey
        ByteKey := MaterialSubstitutionBox0_TemporaryByte
    end for
    Word32Bit_Key = {∅|∀element ∈ F_2^32}
    Word32Bit_Key := IntegerFromBytes(ByteKeys)
    ByteKeys all element reset to 0
    Word32Bit_ExpandedKey = Word32Bit_ExpandKey(Word32Bit_Key)
    Word32Bit_ExpandedKeySpan                         ▷ Define an object called Word32Bit_ExpandedKeySpan, which can directly access the data in the (span range/sub-collection) of Word32Bit_ExpandedKey, and it can also retrieve a reference to the data in the (sub-span range/sub-collection).
    Word32Bit_Random = {0, 0, 0, 0, ... |∀element ∈ F_2^32, ∀element = 0}        ▷ Size is Word32Bit_ExpandedKey.size() ÷ 4
    Index = 0, OffsetIndex = 0
    while OffsetIndex + 4 < Word32Bit_ExpandedKeySpan.size() and Index < Word32Bit_Random.size() do
        Word32Bit_ExpandedKeySubSpan = {Word32Bit_ExpandedKeySpan_OffsetIndex ... Word32Bit_ExpandedKeySpan_OffsetIndex+3}
    ▷ Subspan is (sub-span range/sub-collection) range of Word32Bit_ExpandedKey, elements size is 4
        OffsetIndex := OffsetIndex + 4, Index := Index + 1
        F_2^32 RandomWord = ModifiedZUC.GenerateKeyStream(Word32Bit_ExpandedKeySubSpan) ⊕_32 Word32Bit_ExpandedKeySubSpan_3
        Word32Bit_Random_Index := RandomWord
        RandomWord := 0
    end while
    ByteKeys := IntegerToBytes(Word32Bit_Random)
    Word32Bit_ExpandedKey and Word32Bit_Random and Word32Bit_Key all element reset to 0
    for ByteKey in Ranges(ByteKeys) do                ▷ For each element, Byte data substitution operation via material substitution box 1
        TemporaryByte = MaterialSubstitutionBox1_ByteKey
        ByteKey := MaterialSubstitutionBox1_TemporaryByte
    end for
    Word64Bit_ProcessedKey = {∅|∀element ∈ F_2^64}
    Word64Bit_ProcessedKey = IntegerFromBytes(ByteKeys)
    ByteKeys all element reset to 0
    F_2^1 Word64Bit_KeyUsed = false
    RandomBitsArray = {0, 0, 0, 0, 0, ... |∀element ∈ F_2^1, ∀element = 0}        ▷ RandomBitsArray elements size is 64
    for Row = 0, Row to KeyRows - 1, Row := Row + 1 do
        for Column = 0, Column to KeyColumns - 1, Column := Column + 1 do
            if Column + 1 = Word64Bit_ProcessedKey.size() or Column + 1 = KeyColumns then
                Word64Bit_KeyUsed := true
            end if
            if Column + 1 = Word64Bit_ProcessedKey.size() or Column + 1 = KeyColumns then
                MatrixA_{Row,Column} := MatrixA_{Row,Column} − Word64Bit_ProcessedKey_Column
            else
                while Column < KeyColumns do
                    F_2^64 RandomNumber = 0
                    for RandomBit in Ranges(RandomBitsArray) do        ▷ Using an instance object of the Bernoulli distribution class and an instance object of the linear feedback shift register class, each generates a pseudo-random 64-bit word, and then uses bitwise exclusive or to compute a superimposed 64-bit word result.
                        RandomNumber := BernoulliDistribution(LFSR) ⊕_64 LFSR.generate_bits(63)
                        RandomBit := RandomNumber ∧_64 1
                    end for
                    for BitIndex = 0, BitIndex to RandomBitsArray.size() - 1, BitIndex := BitIndex + 1 do
                        if RandomBitsArray_BitIndex = 1 then
                            RandomNumber := RandomNumber ∨_64 (RandomBitsArray_BitIndex ≪_64 BitIndex)
```

```
                    else
                        BitIndex := BitIndex + 1
                    end if
                    MatrixA_{Row,Column} := MatrixA_{Row,Column} + RandomNumber
                    RandomNumber := 0
                    Column := Column + 1
                end for
            end while
            if Column + 1 < Word64Bit_ProcessedKey.size() then
                Word64Bit_KeyUsed := false
            end if
        end if
    end for
end for
RandomBitsArray all element reset to 0
MaterialSubstitutionBox0 := RegenerationRandomMaterialSubstitutionBox(MaterialSubstitutionBox0)
MaterialSubstitutionBox1 := RegenerationRandomMaterialSubstitutionBox(MaterialSubstitutionBox1)
end function
```

Update algorithm block: 混合 MatrixA 和 MatrixB，然后洗牌 Index (密钥混淆层)

$$
\begin{aligned}
&MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)\\
&MatrixB = \textbf{ReferenceObject}(CommonStateData.TransformedSubkeyMatrix)\\
&MatrixA_{\{Row,Column\}},\ MatrixB_{\{Row,Column\}} \in \mathbb{F}_2^{64}\\
&CommonStateData \xrightarrow[SubkeyMatrixOperationObject.\textbf{UpdateState}()]{MatrixA,\ MatrixB} CommonStateData'
\end{aligned}
$$

接下来我们将详细展示算法中的 UpdateState 函数

Algorithm 15 UpdateState from the name of the class object in the author's code is SecureSubkeyGeneratationModuleObject

```
using CommonStateData.RandomQuadWordMatrix
using CommonStateData.TransformedSubkeyMatrix
using CommonStateData.MatrixOffsetWithRandomIndices
```

Require: RandomQuadWordMatrix, TransformedSubkeyMatrix, MatrixOffsetWithRandomIndices

Ensure: Mixed RandomQuadWordMatrix, TransformedSubkeyMatrix, and shffled MatrixOffsetWithRandomIndices, each $element \in \mathbb{F}_2^{64}$

```
function UpdateState
    NLFSR = ReferenceObject(CommonStateData.NLFSR)
    SDP = ReferenceObject(CommonStateData.SDP)
```

$$
\textbf{RandomVector} = \begin{pmatrix} 0_0 \\ 0_1 \\ \vdots \\ 0_{KeyColumns-1} \end{pmatrix}
$$

$$
\textbf{RandomVector2} = \begin{bmatrix} 0_0 & 0_1 & \cdots & 0_{KeyRows-1} \end{bmatrix}
$$

```
    F_2^64 BaseNumber = 0                                              ▷ 64-bit Counter
    for Rows in RandomVector.rowwise() do      ▷ Iterate over each row from the matrix to access the vector for each column
        for MatrixValue in Rows do                ▷ Iterate through each element(alias name) in this column vector
            MatrixValue := NLFSR.unpredictable_bits(BaseNumber ∧_64 1, 64)
            BaseNumber := BaseNumber + 1
        end for
    end for
    for Columns in RandomVector2.columnwise() do   ▷ Iterate over each column from the matrix to access the vector for each row
        for MatrixValue in Columns do             ▷ Iterate through each element(alias name) in this row vector
            MatrixValue := NLFSR.unpredictable_bits(BaseNumber ∧_64 1, 64)
```

BaseNumber := BaseNumber + 1
end for
end for
$BaseNumber := 0$
$LeftMatrix = (RandomQuadWordMatrix.rowwise() \times_{\mathbb{VEW}} RandomVector)$
$LeftMatrix := LeftMatrix.columnwise() +_{VECTOR} RandomVector2$
$RightMatrix = (RandomQuadWordMatrix.columnwise() \times_{\mathbb{VEW}} RandomVector2)$
$RightMatrix := RightMatrix.rowwise() -_{VECTOR} RandomVector$ ▷ Applying the affine transformation element-wise on each element of the matrix
$\mathbb{F}_2^{64} MatrixRow = 0, MatrixColumn = 0$
$\mathbb{F}_2^{64} ValueA = 0, ValueB = 0$
while MatrixRow < KeyRows **do** ▷ Iterate through each row of the matrix in ascending order
while MatrixColumn < KeyColumns **do** ▷ Iterate through each column of the matrix in ascending order
$Position \rightarrow \{MatrixRow, MatrixColumn\}$
$ValueA = LeftMatrix_{Position} \oplus_{64} (RandomQuadWordMatrix_{Position} \wedge_{64} TransformedSubkeyMatrix_{Position})$
$ValueB = RightMatrix_{Position} \oplus_{64} (RandomQuadWordMatrix_{Position} \vee_{64} TransformedSubkeyMatrix_{Position})$
$RandomQuadWordMatrix_{Position} := RandomQuadWordMatrix_{Position} \oplus_{64} ((ValueA \ggg_{64} 1) + (ValueB \lll_{64} 63))$
MatrixColumn := MatrixColumn + 1
end while
MatrixRow : = MatrixRow + 1
end while
for Rows **in** RandomVector.rowwise() **do** ▷ Iterate over each row from the matrix to access the vector for each column
for MatrixValue **in** Rows **do** ▷ Iterate through each element(alias name) in this column vector
$MatrixValue :=$ SDP($min : 0, max : 18446744073709551615$)
BaseNumber := BaseNumber + 1
end for
end for
for Columns **in** RandomVector2.columnwise() **do** ▷ Iterate over each column from the matrix to access the vector for each row
for MatrixValue **in** Columns **do** ▷ Iterate through each element(alias name) in this row vector
$MatrixValue :=$ SDP($min : 0, max : 18446744073709551615$)
BaseNumber := BaseNumber + 1
end for
end for
$KroneckerProductMatrix = RandomVector \times_{KRONECKER} RandomVector2$
$\mathbb{F}_2^{64} DotProduct = RandomVector \times_{DOT} RandomVector2$
$TransformedSubkeyMatrix := RandomQuadWordMatrix \times_{MATRIX} (KroneckerProductMatrix \times_{SCALAR} DotProduct)$
$DotProduct := 0$
$first =$ **begin**$(MatrixOffsetWithRandomIndices), last =$ **end**$(MatrixOffsetWithRandomIndices)$ ▷ The first and last are iterators from the range
ShuffleRangeData(first, last, CommonStateData.NLFSR)
RandomVector, RandomWordVector2, LeftMatrix, RightMatrix, KroneckerProductMatrix all element reset to 0
end function

4.3.3 Post-process stage: Use MatrixA and MatrixB of common state data to generate subkey vectors of round functions (Key diffusion layer)

这实际上是 **GenerateRoundSubkeys** 函数的实现，最外层的封装函数，它将是第一个使用该函数的函数，我们将在此详细讨论其流程

$$MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)$$
$$MatrixB = \textbf{ReferenceObject}(CommonStateData.TransformedSubkeyMatrix)$$
$$MatrixC = \textbf{ReferenceObject}(RoundSubkeyGeneratationModuleObject.GeneratedMatrix)$$
$$RoundSubkeyGeneratationModuleObject.Matrix_{\{Row,Column\}} \in \mathbb{F}_2^{64}$$
$$MatrixA, MatrixB \xrightarrow[RoundSubkeyGeneratationModuleObject.\textbf{GenerationRoundSubkeys}()]{} MatrixC$$

接下来我们将详细展示算法中的 RoundSubkeyGeneratationModule 类

Algorithm 16 GenerationRoundSubkeys from the name of the class object in the author's code is SecureRoundSubkeyGeneratationModuleObject

1:

$$\textbf{GeneratedMatrix}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

▷ $\forall element \in \mathbb{F}_2^{64}$

2: **GeneratedVetcor** $= \{0_0, 0_1, 0_2, 0_3 \ldots 0_{KeyRows \times KeyColums-1} | \forall element \in \mathbb{F}_2^{64}\}$

3: $\mathbb{F}_2^{64} AlgorithmCounter = 0$

Require: RandomQuadWordMatrix, TransformedSubkeyMatrix

Ensure: GeneratedMatrix, each $element \in \mathbb{F}_2^{64}$

4: **function** OPC_MatrixTransformation ▷ OaldresPuzzle_Cryptic - Unpredictable matrix transformation

5: $MatrixA =$ **ReferenceObject**$(CommonStateData.RandomQuadWordMatrix)$

6: $MartixB =$ **ReferenceObject**$(CommonStateData.TransformedSubkeyMatrix)$

7: $MartixC =$ **ReferenceObject**$(GeneratedMatrix)$

8:

$$\textbf{TemporaryIntegerMatrix}_{KeyRows \times KeyColumns} = \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

▷ $\forall element \in \mathbb{F}_2^{64}$

9: $Temporary0 \rightarrow MartixB^{Transpose}$

10: $Temporary1 \rightarrow MatrixA^{Transpose}$

11: $Temporary2 \rightarrow MatrixA +_{MATRIX} Temporary0$

12: $Temporary3 \rightarrow MartixB -_{MATRIX} Temporary1$

13: $Temporary4 \rightarrow Temporary2 \times_{MATRIX} Temporary3$ ▷ Temporary values

14: $TemporaryIntegerMatrix := Temporary4^{HermitianTranspose}$

15: $MartixC := MartixC \ +_{MATRIX} \ (TemporaryIntegerMatrix \times_{MATRIX} MatrixA \times_{MATRIX} MatrixB)$

16:

$$\textbf{TemporaryIntegerMatrix}_{KeyRows \times KeyColumns} := \begin{bmatrix} 0 & 0 & \cdots & 0 \\ 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix}$$

▷ Ensure that the status matrix is securely cleaned

17: **end function**

Require: GenerateMatrix

Ensure: GeneratedVector, each $element \in \mathbb{F}_2^{64}$

18: **function** GenerationRoundSubkeys ▷ Take the old QuadWord subkey matrix and the QuadWord subkey matrix used for the round function, perform one-way transformation and operation, and generate a new QuadWord subkey matrix and subkey vector, and use them as the RoundSubkey of the round function

19: **if** AlgorithmCounter = 0 **then**

20: GeneratedMatrix, GeneratedVector all element reset to 0

21: **end if**

22: OPC_MatrixTransformation()

23: $\mathbb{F}_2^{64} Index = 0$

24: **while** (Index < GeneratedVector.size() **do**

25: $GeneratedVector_{Index} := eneratedVector_{Index} \oplus_{64} GeneratedMatrix_{\{Index \div KeyColumns, Index \pmod{KeyColumns}\}}$

26: **end while** ▷ Key whitening

27: $TransformedVector = \{0_0, 0_1, 0_2, 0_3, 0_4, \ldots 0_{GeneratedVetcor.size()-1} | \forall element \in \mathbb{F}_2^{64}\}$

28: $NewRoundSubkeyVectorSpan \leftrightarrow \{TransformedVector_0 \ldots TransformedVector_{TransformedVector.size()-1}\}$ ▷ Define an object called NewRoundSubkeyVectorSpan, which can directly access the data in the TransformedVector (span range/collection range), and it can also take out a reference to the data (sub-span range/sub-collection range).

29: $RoundSubkeyVectorSpan \leftrightarrow \{GeneratedVector_0 \ldots GeneratedVector_{GeneratedVector.size()-1}\}$ ▷ Define an object called RoundSubkeyVectorSpan, which can directly access the data in the GeneratedVector (span range/collection range), and it can also take out a reference to the data (sub-span range/sub-collection range).

30: **for** Index = 0, Index < RoundSubkeyVectorSpan.size(), Index = Index + 32 **do**

31: $X \leftrightarrow \{RoundSubkeyVectorSpan_{Index} \ldots NewRoundSubkeyVectorSpan_{Index+32}\}$

32: $Y \leftrightarrow \{NewRoundSubkeyVectorSpan_{Index} \ldots NewRoundSubkeyVectorSpan_{Index+32}\}$ ▷ KeyStateX, KeyStateY are subspan views of GeneratedVector, TransformedVector

33: $Y_0 := X_{24} \oplus_{64} X_8 \oplus_{64} X_6 \oplus_{64} X_1 \oplus_{64} X_9 \oplus_{64} X_4 \oplus_{64} X_{10} \oplus_{64} X_3 \oplus_{64} X_{26} \oplus_{64} X_2 \oplus_{64} X_5 \oplus_{64} X_{15} \oplus_{64} X_{17} \oplus_{64} X_{13} \oplus_{64} X_{23} \oplus_{64} X_{12}$

34: $Y_1 := X_{19} \oplus_{64} X_{11} \oplus_{64} X_{22} \oplus_{64} X_{14} \oplus_{64} X_{25} \oplus_{64} X_{31} \oplus_{64} X_7 \oplus_{64} X_0 \oplus_{64} X_{30} \oplus_{64} X_{21} \oplus_{64} X_{28} \oplus_{64} X_{20} \oplus_{64} X_{18} \oplus_{64} X_{27} \oplus_{64} X_{29} \oplus_{64} X_{16}$

35: $Y_2 := X_4 \oplus_{64} X_{18} \oplus_{64} X_{10} \oplus_{64} X_{26} \oplus_{64} X_1 \oplus_{64} X_{22} \oplus_{64} X_{30} \oplus_{64} X_{21} \oplus_{64} X_{20} \oplus_{64} X_5 \oplus_{64} X_{23} \oplus_{64} X_{12} \oplus_{64} X_{17} \oplus_{64} X_6 \oplus_{64} X_3 \oplus_{64} X_{25}$

36: $Y_3 := X_{11} \oplus_{64} X_{19} \oplus_{64} X_{24} \oplus_{64} X_{16} \oplus_{64} X_0 \oplus_{64} X_7 \oplus_{64} X_{28} \oplus_{64} X_{13} \oplus_{64} X_{29} \oplus_{64} X_{14} \oplus_{64} X_2 \oplus_{64} X_{15} \oplus_{64} X_{27} \oplus_{64} X_8 \oplus_{64} X_{31} \oplus_{64} X_9$

37: $Y_4 := X_{21} \oplus_{64} X_{13} \oplus_{64} X_{28} \oplus_{64} X_4 \oplus_{64} X_7 \oplus_{64} X_{24} \oplus_{64} X_{25} \oplus_{64} X_9 \oplus_{64} X_{16} \oplus_{64} X_5 \oplus_{64} X_6 \oplus_{64} X_{19} \oplus_{64} X_{23} \oplus_{64} X_{31} \oplus_{64} X_{27} \oplus_{64} X_1$

38: $Y_5 := X_{15} \oplus_{64} X_3 \oplus_{64} X_{11} \oplus_{64} X_2 \oplus_{64} X_{12} \oplus_{64} X_{20} \oplus_{64} X_{17} \oplus_{64} X_{30} \oplus_{64} X_{10} \oplus_{64} X_{22} \oplus_{64} X_8 \oplus_{64} X_0 \oplus_{64} X_{18} \oplus_{64} X_{26} \oplus_{64} X_{29} \oplus_{64} X_{14}$

39: $Y_6 := X_{16} \oplus_{64} X_{24} \oplus_{64} X_{21} \oplus_{64} X_{25} \oplus_{64} X_{18} \oplus_{64} X_{10} \oplus_{64} X_{30} \oplus_{64} X_{22} \oplus_{64} X_0 \oplus_{64} X_6 \oplus_{64} X_{27} \oplus_{64} X_1 \oplus_{64} X_{23} \oplus_{64} X_4 \oplus_{64} X_{28} \oplus_{64} X_3$

40: $Y_7 := X_{12} \oplus_{64} X_{20} \oplus_{64} X_{14} \oplus_{64} X_{31} \oplus_{64} X_{15} \oplus_{64} X_2 \oplus_{64} X_9 \oplus_{64} X_8 \oplus_{64} X_{29} \oplus_{64} X_{11} \oplus_{64} X_5 \oplus_{64} X_{19} \oplus_{64} X_{26} \oplus_{64} X_{13} \oplus_{64} X_{17} \oplus_{64} X_7$

41: $Y_8 := X_7 \oplus_{64} X_{31} \oplus_{64} X_8 \oplus_{64} X_{24} \oplus_{64} X_2 \oplus_{64} X_9 \oplus_{64} X_3 \oplus_{64} X_{22} \oplus_{64} X_{14} \oplus_{64} X_6 \oplus_{64} X_4 \oplus_{64} X_{20} \oplus_{64} X_{27} \oplus_{64} X_{17} \oplus_{64} X_{26} \oplus_{64} X_{21}$

42: $Y_9 := X_{19} \oplus_{64} X_{23} \oplus_{64} X_{15} \oplus_{64} X_{28} \oplus_{64} X_5 \oplus_{64} X_0 \oplus_{64} X_1 \oplus_{64} X_{10} \oplus_{64} X_{25} \oplus_{64} X_{30} \oplus_{64} X_{13} \oplus_{64} X_{12} \oplus_{64} X_{18} \oplus_{64} X_{16} \oplus_{64} X_{29} \oplus_{64} X_{11}$

43: $Y_{10} := X_{25} \oplus_{64} X_9 \oplus_{64} X_{30} \oplus_{64} X_{22} \oplus_{64} X_{14} \oplus_{64} X_3 \oplus_{64} X_{10} \oplus_{64} X_{18} \oplus_{64} X_{12} \oplus_{64} X_4 \oplus_{64} X_{26} \oplus_{64} X_{21} \oplus_{64} X_{27} \oplus_{64} X_{24} \oplus_{64} X_8 \oplus_{64} X_{28}$

44: $Y_{11} := X_0 \oplus_{64} X_{17} \oplus_{64} X_1 \oplus_{64} X_{19} \oplus_{64} X_{11} \oplus_{64} X_{13} \oplus_{64} X_5 \oplus_{64} X_7 \oplus_{64} X_{29} \oplus_{64} X_{15} \oplus_{64} X_6 \oplus_{64} X_{20} \oplus_{64} X_{16} \oplus_{64} X_{31} \oplus_{64} X_{23} \oplus_{64} X_2$

45: $Y_{12} := X_9 \oplus_{64} X_{17} \oplus_{64} X_{13} \oplus_{64} X_5 \oplus_{64} X_7 \oplus_{64} X_2 \oplus_{64} X_{28} \oplus_{64} X_{30} \oplus_{64} X_{11} \oplus_{64} X_4 \oplus_{64} X_{24} \oplus_{64} X_0 \oplus_{64} X_{26} \oplus_{64} X_{23} \oplus_{64} X_{16} \oplus_{64} X_{22}$

46: $Y_{13} := X_{12} \oplus_{64} X_{20} \oplus_{64} X_{27} \oplus_{64} X_{19} \oplus_{64} X_8 \oplus_{64} X_6 \oplus_{64} X_{21} \oplus_{64} X_{25} \oplus_{64} X_3 \oplus_{64} X_{10} \oplus_{64} X_{31} \oplus_{64} X_1 \oplus_{64} X_{18} \oplus_{64} X_{14} \oplus_{64} X_{29} \oplus_{64} X_{15}$

47: $Y_{14} := X_7 \oplus_{64} X_3 \oplus_{64} X_{11} \oplus_{64} X_{30} \oplus_{64} X_{28} \oplus_{64} X_{18} \oplus_{64} X_{10} \oplus_{64} X_{25} \oplus_{64} X_1 \oplus_{64} X_{24} \oplus_{64} X_{16} \oplus_{64} X_{22} \oplus_{64} X_{26} \oplus_{64} X_9 \oplus_{64} X_{13} \oplus_{64} X_8$

48: $Y_{15} := X_{20} \oplus_{64} X_{12} \oplus_{64} X_{21} \oplus_{64} X_{23} \oplus_{64} X_{31} \oplus_{64} X_{15} \oplus_{64} X_6 \oplus_{64} X_2 \oplus_{64} X_{29} \oplus_{64} X_{19} \oplus_{64} X_4 \oplus_{64} X_0 \oplus_{64} X_{14} \oplus_{64} X_{17} \oplus_{64} X_{27} \oplus_{64} X_5$

▷ Vectorα := Part A of Matrix × Vectorα, This use $\mathbb{F}_2^{64}$ multiplication, implemented as a bitwise operation of the form

49: $Y_{16} := X_7 \oplus_{64} X_{31} \oplus_{64} X_8 \oplus_{64} X_{24} \oplus_{64} X_2 \oplus_{64} X_9 \oplus_{64} X_3 \oplus_{64} X_{22} \oplus_{64} X_{14} \oplus_{64} X_6 \oplus_{64} X_4 \oplus_{64} X_{20} \oplus_{64} X_{27} \oplus_{64} X_{17} \oplus_{64} X_{26} \oplus_{64} X_{21}$

50: $Y_{17} := X_{19} \oplus_{64} X_{23} \oplus_{64} X_{15} \oplus_{64} X_{28} \oplus_{64} X_5 \oplus_{64} X_0 \oplus_{64} X_1 \oplus_{64} X_{10} \oplus_{64} X_{25} \oplus_{64} X_{30} \oplus_{64} X_{13} \oplus_{64} X_{12} \oplus_{64} X_{18} \oplus_{64} X_{16} \oplus_{64} X_{29} \oplus_{64} X_{11}$

51: $Y_{18} := X_{25} \oplus_{64} X_9 \oplus_{64} X_{30} \oplus_{64} X_{22} \oplus_{64} X_{14} \oplus_{64} X_3 \oplus_{64} X_{10} \oplus_{64} X_{18} \oplus_{64} X_{12} \oplus_{64} X_4 \oplus_{64} X_{26} \oplus_{64} X_{21} \oplus_{64} X_{27} \oplus_{64} X_{24} \oplus_{64} X_8 \oplus_{64} X_{28}$

52: $Y_{19} := X_0 \oplus_{64} X_{17} \oplus_{64} X_1 \oplus_{64} X_{19} \oplus_{64} X_{11} \oplus_{64} X_{13} \oplus_{64} X_5 \oplus_{64} X_7 \oplus_{64} X_{29} \oplus_{64} X_{15} \oplus_{64} X_6 \oplus_{64} X_{20} \oplus_{64} X_{16} \oplus_{64} X_{31} \oplus_{64} X_{23} \oplus_{64} X_2$

53: $Y_{20} := X_9 \oplus_{64} X_{17} \oplus_{64} X_{13} \oplus_{64} X_5 \oplus_{64} X_7 \oplus_{64} X_2 \oplus_{64} X_{28} \oplus_{64} X_{30} \oplus_{64} X_{11} \oplus_{64} X_4 \oplus_{64} X_{24} \oplus_{64} X_0 \oplus_{64} X_{26} \oplus_{64} X_{23} \oplus_{64} X_{16} \oplus_{64} X_{22}$

54: $Y_{21} := X_{12} \oplus_{64} X_{20} \oplus_{64} X_{27} \oplus_{64} X_{19} \oplus_{64} X_8 \oplus_{64} X_6 \oplus_{64} X_{21} \oplus_{64} X_{25} \oplus_{64} X_3 \oplus_{64} X_{10} \oplus_{64} X_{31} \oplus_{64} X_1 \oplus_{64} X_{18} \oplus_{64} X_{14} \oplus_{64} X_{29} \oplus_{64} X_{15}$

55: $Y_{22} := X_7 \oplus_{64} X_3 \oplus_{64} X_{11} \oplus_{64} X_{30} \oplus_{64} X_{28} \oplus_{64} X_{18} \oplus_{64} X_{10} \oplus_{64} X_{25} \oplus_{64} X_1 \oplus_{64} X_{24} \oplus_{64} X_{16} \oplus_{64} X_{22} \oplus_{64} X_{26} \oplus_{64} X_9 \oplus_{64} X_{13} \oplus_{64} X_8$

56: $Y_{23} := X_{20} \oplus_{64} X_{12} \oplus_{64} X_{21} \oplus_{64} X_{23} \oplus_{64} X_{31} \oplus_{64} X_{15} \oplus_{64} X_6 \oplus_{64} X_2 \oplus_{64} X_{29} \oplus_{64} X_{19} \oplus_{64} X_4 \oplus_{64} X_0 \oplus_{64} X_{14} \oplus_{64} X_{17} \oplus_{64} X_{27} \oplus_{64} X_5$

57: $Y_{24} := X_{31} \oplus_{64} X_7 \oplus_{64} X_{23} \oplus_{64} X_6 \oplus_{64} X_{10} \oplus_{64} X_2 \oplus_{64} X_5 \oplus_{64} X_8 \oplus_{64} X_{15} \oplus_{64} X_{24} \oplus_{64} X_9 \oplus_{64} X_{12} \oplus_{64} X_{16} \oplus_{64} X_{27} \oplus_{64} X_{14} \oplus_{64} X_{30}$

58: $Y_{25} := X_0 \oplus_{64} X_4 \oplus_{64} X_{20} \oplus_{64} X_{13} \oplus_{64} X_1 \oplus_{64} X_{22} \oplus_{64} X_{26} \oplus_{64} X_3 \oplus_{64} X_{28} \oplus_{64} X_{25} \oplus_{64} X_{17} \oplus_{64} X_{21} \oplus_{64} X_{18} \oplus_{64} X_{11} \oplus_{64} X_{29} \oplus_{64} X_{19}$

59: $Y_{26} := X_{18} \oplus_{64} X_{10} \oplus_{64} X_2 \oplus_{64} X_{15} \oplus_{64} X_8 \oplus_{64} X_{28} \oplus_{64} X_{25} \oplus_{64} X_3 \oplus_{64} X_{21} \oplus_{64} X_9 \oplus_{64} X_{14} \oplus_{64} X_{30} \oplus_{64} X_{16} \oplus_{64} X_7 \oplus_{64} X_{31} \oplus_{64} X_{13}$

60: $Y_{27} := X_{17} \oplus_{64} X_1 \oplus_{64} X_{22} \oplus_{64} X_{27} \oplus_{64} X_{19} \oplus_{64} X_0 \oplus_{64} X_4 \oplus_{64} X_5 \oplus_{64} X_{29} \oplus_{64} X_{20} \oplus_{64} X_{24} \oplus_{64} X_{12} \oplus_{64} X_{11} \oplus_{64} X_{23} \oplus_{64} X_{26} \oplus_{64} X_6$

61: $Y_{28} := X_{27} \oplus_{64} X_2 \oplus_{64} X_4 \oplus_{64} X_{13} \oplus_{64} X_5 \oplus_{64} X_6 \oplus_{64} X_{17} \oplus_{64} X_{25} \oplus_{64} X_{19} \oplus_{64} X_9 \oplus_{64} X_7 \oplus_{64} X_1 \oplus_{64} X_{14} \oplus_{64} X_{26} \oplus_{64} X_{11} \oplus_{64} X_{10}$

62: $Y_{29} := X_{28} \oplus_{64} X_{12} \oplus_{64} X_{16} \oplus_{64} X_{24} \oplus_{64} X_0 \oplus_{64} X_{31} \oplus_{64} X_{21} \oplus_{64} X_{30} \oplus_{64} X_8 \oplus_{64} X_3 \oplus_{64} X_{23} \oplus_{64} X_{22} \oplus_{64} X_{18} \oplus_{64} X_{15} \oplus_{64} X_{29} \oplus_{64} X_{20}$

63: $Y_{30} := X_{13} \oplus_{64} X_5 \oplus_{64} X_3 \oplus_{64} X_{19} \oplus_{64} X_{25} \oplus_{64} X_8 \oplus_{64} X_{18} \oplus_{64} X_{28} \oplus_{64} X_{22} \oplus_{64} X_7 \oplus_{64} X_{11} \oplus_{64} X_{10} \oplus_{64} X_{14} \oplus_{64} X_2 \oplus_{64} X_{17} \oplus_{64} X_{31}$

64: $Y_{31} := X_{21} \oplus_{64} X_6 \oplus_{64} X_{30} \oplus_{64} X_{12} \oplus_{64} X_{20} \oplus_{64} X_{24} \oplus_{64} X_{23} \oplus_{64} X_{26} \oplus_{64} X_{29} \oplus_{64} X_0 \oplus_{64} X_9 \oplus_{64} X_1 \oplus_{64} X_{15} \oplus_{64} X_{27} \oplus_{64} X_{16} \oplus_{64} X_4$

▷ Vectorβ := Part B of Matrix × Vectorβ, This use $\mathbb{F}_2^{64}$ multiplication, implemented as a bitwise operation of the form

65: **end for** ▷ Bits data diffusion layer - Data avalanche effect for diffusion

66: ▷ The choice of the X constant subscript is generated using the cryptographically secure pseudo-random number generator ISAAC 64 plus bit version in conjunction with a duplicate element removal hash table, generated by shuffling through a jumbled array. (We implement this GenerateDiffusionLayerPermuteIndices function in the appendix.)

67: GeneratedVector := TransformedVector

68: AlgorithmCounter := AlgorithmCounter + 1

69: **end function**

我们已经充分解释了 **GenerateSubkeys** 和 **GenerateRoundSubkeys** 数学抽象函数中涉及的所有模块

4.4 Implementation of the lai-massey scheme after modified the execution order of the F and H functions

Algorithm 17 OPC core algorithm - Modified lai–massey scheme details - Apply PRNG keys for round function

Require: $WordDatas \in \mathbb{F}_2^{64}$, $WordKeyMaterial \in \mathbb{F}_2^{64}$

Ensure: Updated $WordData$

1: **using** CommonStateData.MatrixOffsetWithRandomIndices

2: **using** SecureRoundSubkeyGeneratationModule.GeneratedRoundSubkeyMatrix

3: $LeftWordData \in \mathbb{F}_2^{32}$ and $RightWordData \in \mathbb{F}_2^{32}$ from the RoundFunction

4: **function** SRSGM.**ForwardTransform**($LeftWordData$, $RightWordData$) ▷ The H-function encode described by Lai–Massey Scheme

5: $LeftWordData' = LeftWordData \boxplus_{32} RightWordData$

6: $RightWordData' = LeftWordData \boxplus_{32} 2 \boxtimes_{32} RightWordData$

7: $RightWordData' := RightWordData' \oplus_{32} (LeftWordData' \lll_{32} 1)$
8: $LeftWordData' := LeftWordData' \oplus_{32} (RightWordData' \ggg_{32} 63)$
9: **return** $\{LeftWordData', RightWordData'\}$
10: **end function**

11: **function** SRSGM.**BackwardTransform**($LeftWordData'$, $RightWordData'$) ▷ The H-function decode described by Lai–Massey Scheme
12: $LeftWordData' := LeftWordData' \oplus_{32} (RightWordData' \ggg_{32} 63)$
13: $RightWordData' := RightWordData' \oplus_{32} (LeftWordData' \lll_{32} 1)$
14: $RightWordData = RightWordData' \boxminus_{32} LeftWordData'$
15: $LeftWordData = 2 \boxtimes_{32} LeftWordData' \boxminus_{32}$ RightWordData'
16: **return** $\{LeftWordData, RightWordData\}$
17: **end function**

18: **function** SRSGM.**CrazyTransformAssociatedWord**($AssociatedWordData$, $WordKeyMaterial$) ▷ The F-function described by Lai-Massey Scheme
19: BitReorganizationWord $\in \mathbb{F}_2^{32}$
20: BitReorganizationWord $= \{0, 0\}$
21: $WordA \longleftrightarrow BitReorganizationWord_0$
22: $WordB \longleftrightarrow BitReorganizationWord_1$
23: $\{LeftWordKey, RightWordKey\} =$ **Split**($WordKeyMaterial$)
24: ▷ LeftWordKey and RightWordKey are constant 32-bits word, $LeftWordKey, RightWordKey \in \mathbb{F}_2^{32}$
25: $\mathbb{F}_2^{64} PseudoRandomValue = ((WordKeyMaterial \oplus_{64} AssociatedWordData) \lll_{64} 32) \vee_{64} ((WordKeyMaterial \odot_{64} AssociatedWordData) \ggg_{64} 32)$
26: $\mathbb{F}_2^{32} WordC = PseudoRandomValue \lll_{64} (WordKeyMaterial \pmod{64}) \ggg_{64} 32$
27: $\mathbb{F}_2^{32} WordD = PseudoRandomValue \ggg_{64} (WordKeyMaterial \pmod{64})$
28: $WordC := (AssociatedWordData \vee_{32} LeftWordKey) \wedge_{32} WordC$
29: $WordD := (AssociatedWordData \wedge_{32} RightWordKey) \vee_{32} WordD$
30: $WordA := WordA \oplus_{32} WordC$
31: $WordB := WordB \oplus_{32} WordD$
32: $WordB := (WordA \boxplus_{32} LeftWordKey) \lll_{32} (PseudoRandomValue \pmod{32})$
33: $WordA := (WordB \boxplus_{32} RightWordKey) \ggg_{32} (PseudoRandomValue \pmod{32})$
34: $WordC := (WordB \wedge_{32} LeftWordKey) \oplus_{32} (WordD \vee_{32} AssociatedWordData)$
35: $WordD := (WordA \wedge_{32} RightWordKey) \oplus_{32} (WordC \vee_{32} AssociatedWordData)$
36: $WordA := WordA \oplus_{32} WordC$
37: $WordB := WordB \oplus_{32} WordD$
38: $MatrixRow = MatrixOffsetWithRandomIndices_{WordA \pmod{MatrixOffsetWithRandomIndices.size()}}$
39: $MatrixColumn = MatrixOffsetWithRandomIndices_{WordB \pmod{MatrixOffsetWithRandomIndices.size()}}$
40: ▷ MatrixRow and MatrixColumn are constant 32-bits word
41: $\mathbb{F}_2^{32} ShiftAmount = WordA \boxplus_{32} WordB$
42: $\mathbb{F}_2^{32} ShiftAmount2 = WordA \boxplus_{32} WordB \boxtimes_{32} 2$
43: $\mathbb{F}_2^{32} RotateAmount = MatrixColumn \boxminus_{32} MatrixRow$
44: $\mathbb{F}_2^{32} RotateAmount2 = 2 \boxtimes_{32} MatrixRow \boxminus_{32} MatrixColumn$
45: $RoundSubkey \in \mathbb{F}_2^{64}$
46: $RoundSubkey = GeneratedRoundSubkeyMatrix_{\{MatrixRow, MatrixColumn\}}$
47: $\mathbb{F}_2^{64} Bit = (RoundSubkey \ggg_{64} ShiftAmount \pmod{64}) \wedge_{64} 1$
48: $\mathbb{F}_2^{64} Bit2 = (RoundSubkey \ggg_{64} ShiftAmount2 \pmod{64}) \wedge_{64} 1$
49: $\mathbb{F}_2^{64} LeftRotatedMask = Bit \lll_{64} RotateAmount \pmod{64}$
50: $\mathbb{F}_2^{64} RightRotatedMask = Bit2 \ggg_{64} RotateAmount2 \pmod{64}$
51: $\mathbb{F}_2^{64} BitMask = LeftRotatedMask \oplus_{64} RightRotatedMask \pmod{64}$
52: **if** BitMask $= 0$ **then**
53: $BitMask := BitMask \vee_{64} (1 \lll_{64} ((MatrixRow \boxplus_{32} MatrixColumn) \boxtimes_{64} 2) \pmod{64})$
54: **end if**
55: $RoundSubkey := RoundSubkey \wedge_{64} (\neg_{64} BitMask)$
56: $\{aa, bb\} :=$ **Split**($RoundSubkey$)
57: WordA := WordA $\oplus_{32}$ aa
58: WordB := WordB $\oplus_{32}$ bb
59: AssociatedWordData := AssociatedWordData $\oplus_{32}$ ($WordA \oplus_{32} WordB$)
60: **return** $AssociatedWordData$
61: **end function**

4.5 Workflow detail - OaldresPuzzle_Cryptic Algorithm wrapper class - StateDataWorker(SDW)

感谢你阅读这篇论文，现在我们只需要按照之前的架构和提供的算法来实现 OaldresPuzzle_Cryptic 的算法框架。OPC 算法建立在 StateDataWorker 类中，其伪代码如下所示

Algorithm 18 OPC algorithm - Round function (Encrypting and Decrypting) mode

$SRSGM$ = **ReferenceObject**($SecureRoundSubkeyGeneratationModuleObject$)

function SDW.**EncryptingRound**($EachRoundDatas$)
  if EachRoundDatas is not DataBlockSize **then**
    return
  end if
  RoundSubkeyVector := SRSGM.$GeneratedVector$ ▷ Is Object Reference
  $BytesDatas = \{0, 0, 0, 0, 0, \ldots | \forall element \in \mathbb{F}_2^8, \forall element = 0\}$
  $BytesData$ size is EachRoundDatas.size() × 8 ▷ A quadword data is eight byte data
  $KeyIndex = 0$
  SRSGM.**GenerationRoundSubkeys**()
  for RoundCounter = 0; RoundCounter < 16; RoundCounter := RoundCounter + 1 **do**
    Flag **DoEncryptionDataBlock**
    for Index = 0; Index < EachRoundDatas.size(); Index := Index + 1 **do**
      $EachRoundDatas_{Index}$:= **EncrytionByLaiMasseyFramework**($EachRoundDatas_{Index}, RoundSubkeyVector_{KeyIndex}$)
      if KeyIndex < RoundSubkeyVector.size() **then**
        $KeyIndex := KeyIndex + 1$
      end if
    end for
    if KeyIndex < RoundSubkeyVector.size() **then**
      goto DoEncryptionDataBlock
    else
      $KeyIndex := 0$
    end if
    $BytesData$:= **IntegerToBytes**($EachRoundDatas$)
    SDW.**ForwardBytesSubstitution**($BytesData$)
    $EachRoundDatas$:= **IntegerFromBytes**($BytesData$)
  end for
end function

function SDW.**DecryptingRound**($EachRoundDatas$)
  if EachRoundDatas is not DataBlockSize **then**
    return
  end if
  RoundSubkeyVector := SRSGM.$GeneratedVector$ ▷ Is Object Reference
  $BytesDatas = \{0, 0, 0, 0, 0, \ldots | \forall element \in \mathbb{F}_2^8, \forall element = 0\}$
  $BytesData$ size is EachRoundDatas.size() × 8 ▷ A quadword data is eight byte data
  $KeyIndex = 0$
  SRSGM.**GenerationRoundSubkeys**()
  for RoundCounter = 0; RoundCounter < 16; RoundCounter := RoundCounter + 1 **do**
    $BytesData$:= **IntegerToBytes**($EachRoundDatas$)
    SDW.**BackwardBytesSubstitution**($BytesData$)
    $EachRoundDatas$:= **IntegerFromBytes**($BytesData$)
    Flag **DoDecryptionDataBlock**
    for Index = EachRoundDatas.size(); Index > 0; Index := Index - 1 **do**
      $EachRoundDatas_{Index}$:= **DecrytionByLaiMasseyFramework**($EachRoundDatas_{Index-1}, RoundSubkeyVector_{KeyIndex-1}$)
      if (KeyIndex - 1) > 0 **then**
        KeyIndex := KeyIndex - 1
      end if
    end for
    if (KeyIndex - 1) > 0 **then**
      goto DoDecryptionDataBlock
    else
      $KeyIndex$:= $RoundSubkeyVector.size()$

53: **end if**
54: **end for**
55: **end function**

Applied encryption functions

ScryptKDF_AlgorithmClass KeyDerivationFunctionObject
Define Class Member Function:

$KeyDerivationFunctionObject$.**GenerateKeys**($\mathbb{F}_2^8 SecretBytes, \mathbb{F}_2^8 SaltBytes, ResultByteSize, ResourceCost, BlockSize, ParallelizationCount$)

Algorithm 19 OPC algorithm - Encrypt data wrapper funtion

1: $SSGM$ = **ReferenceObject**($StateDataWorker.SecureSubkeyGeneratationModuleObject$)

Require: PlainText 64 bits array array and Keys 64 bits array
Ensure: CipherText 64 bits array
2: $PlainText \in \mathbb{F}_2^{64}$ or $CipherText \in \mathbb{F}_2^{64}$ and $Keys \in \mathbb{F}_2^{64}$
3: The CommonStatedata is class, The Instance Object Alias Name is CSD
4: $\mathbb{F}_2^{64} RoundSubkeysCounter = 0$

5: **function** SDW.**SplitDataBlockToEncrypt**($PlainText$, $Keys$)
6: **if** PlainText.size() (mod $DataBlockSize$) $\neq 0$ **then**
7: **return**
8: **end if**
9: **if** Keys.size() (mod $DataBlockSize$) $\neq 0$ **then**
10: **return**
11: **end if**
12: $Key\_OffsetIndex = 0$
13: $KeyDataVector := CSD.WordKeyDataVector$ ▷ Is Object Reference
14: $KeyDataVector_0 \dots KeyDataVector_{KeyDataVector.size()} := Keys_0 \dots Keys_{0+KeyDataVector.size()}$
15: $Key\_OffsetIndex = Key\_OffsetIndex + KeyBlockSize$
16: $RandomKeyDataVector = \{0, 0, 0, 0, 0, \dots | \forall element \in \mathbb{F}_2^{64}, \forall element = 0\}$
17: $RandomKeyDataVector$ size is KeyBlockSize × 2
18: $\mathbb{F}_2^1$ CCFlag = **true** ▷ The Condition Control Flag
19: MersenneTwister64Bit
20: **for** DataBlockOffset = 0; DataBlockOffset < PlainTextSize; DataBlockOffset := DataBlockOffset + DataBlockSize **do**
21: **if** $Key\_OffsetIndex <$ Keys.size() **then**
22: $KeySpan \longleftrightarrow \{Keys_{Key\_OffsetIndex} \dots Keys_{Key\_OffsetIndex+KeyBlockSize}\}$
23: **for** Index = 0; Index < KeySpan.size() and Index < KeyDataVector.size(); Index := Index + 1 **do**
24: **if** $KeyDataVector_{Index} = KeySpan_{Index}$ **then**
25: $KeyDataVector_{Index} := \neg_{64}(KeyDataVector_{Index} \boxplus_{64} KeySpan_{Index})$
26: **else**
27: $KeyDataVector_{Index} := KeyDataVector_{Index} \oplus_{64} KeySpan_{Index}$
28: **end if**
29: **end for**
30: $Key\_OffsetIndex := Key\_OffsetIndex + KeyBlockSize$
31: SSGM.**GenerationSubkeys**($KeyDataVector$)
32: $RoundSubkeysCounter := RoundSubkeysCounter + 1$
33: **else**
34: **if** CCFlag or ((RoundSubkeysCounter (mod 2048 × 4)) == 0) **then**
35: **for** KeyRound = 0; KeyRound < 16; KeyRound := KeyRound + 1 **do**
36: **for** i = 0; i < WordKeyDataVector.size(); i := i + 1 **do**
37: $\mathbb{F}_2^{64} a = WordKeyDataVector_i \gg_{64} 32$
38: $\mathbb{F}_2^{64} b = WordKeyDataVector_i \wedge_{64} 0x00000000FFFFFFFF$
39: $a := a \oplus_{64} b$
40: $a := \neg_{64} a$
41: $b := b \oplus_{64} a$
42: $b := b \lll_{64} 19$
43: $a := a \oplus_{64} b$
44: $a := a \lll_{64} 13$
45: $b := b \oplus_{64} a$
46: $b := \neg_{64} b$

$a := a \oplus_{64} b$
$a := a \lll_{64} 27$
$b := b \oplus_{64} a$
$b := b \lll_{64} 23$ ▷ Apply bitwise operations to diffuse bits
$WordKeyDataVector_i := (a \ll_{64} 32) \vee_{64} b$
KeyBytes = $\{0, 0, 0, 0, 0 \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$ size is $KeyBlockSize \times 8$ ▷ 8 is the number of bytes in each 64 bits.
$KeyBytes :=$ **IntegerToBytes**($WordKeyDataVector$)
SDW.**ForwardBytesSubstitution**($KeyBytes$) ▷ Call Byte-level data confusion algorithm
$WordKeyDataVector :=$ **IntegerFromBytes**($KeyBytes$)
end for ▷ Bit-level data diffusion algorithm
end for
SSGM.**GenerationSubkeys**($WordKeyDataVector$)
CCFlag = false
$RoundSubkeysCounter := RoundSubkeysCounter + 1$
Continue
end if
if RoundSubkeysCounter (mod 2048) = 0 **then**
SaltWordData size is 16, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^{64}\}$
for Index = 0; Index < 16; Index := Index + 1 **do**
$SaltWordData_{index} :=$ **MersenneTwister64Bit**()
end for
if RoundSubkeysCounter (mod 2048 × 3) = 0 **then**
SaltData size is 16 × 8, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$
SaltData := **IntegerToBytes**($SaltWorData$)
MaterialKeys = **IntegerToBytes**($RandomKeyDataVector$)
GeneratedSecureKeys size is 0, $GeneratedSecureKeys = \{\varnothing | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$
GeneratedSecureKeys := KeyDerivationFunctionObject.**GenerateKeys**($MaterialKeys, SaltData, RandomKeyDataVector.size() \times 8, 1024, 8, 16$)
RandomKeyDataVector = **IntegerFromBytes**($GeneratedSecureKeys$)
SDW.**GenerationSubkeys**(RandomKeyDataVector)
else if RoundSubkeysCounter (mod 2048 × 2) = 0 **then**
SaltData size is 16 × 8, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$
SaltData := **IntegerToBytes**($SaltWorData$)
MaterialKeys = **IntegerToBytes**($RandomKeyDataVector$)
GeneratedSecureKeys size is 0, $GeneratedSecureKeys = \{\varnothing | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$
GeneratedSecureKeys := KeyDerivationFunctionObject.**GenerateKeys**($MaterialKeys, SaltData, RandomKeyDataVector.size() \times 8, 1024, 8, 16$)
RandomKeyDataVector = **IntegerFromBytes**($GeneratedSecureKeys$)
SDW.**GenerationSubkeys**(RandomKeyDataVector)
Seeds size is KeyBlockSize × 2, $Seeds = \{RandomKeyDataVector_0 \ldots RandomKeyDataVector_{KeyBlockSize-1} | \forall element = 0, \forall element \in \mathbb{F}_2^{64}\}$
MersenneTwister64Bit.**Seed**($Seeds$)
end if
SDW.**GenerationSubkeys**($\varnothing$)
end if
$RoundSubkeysCounter := RoundSubkeysCounter + 1$
end if
$DataSpan \longleftrightarrow \{PlainText_{DataBlockOffset} \ldots PlainText_{DataBlockOffset+DataBlockSize}\}$
SDW.**EncryptingRound**(DataSpan)
end for
if PlainText.size() == DataBlockSize **then**
SDW.**EncryptingRound**(PlainText)
end if
end function

Applied decryption functions

ScryptKDF_AlgorithmClass KeyDerivationFunctionObject
Define Class Member Function:

$KeyDerivationFunctionObject$.**GenerateKeys**($\mathbb{F}_2^8 SecretBytes, \mathbb{F}_2^8 SaltBytes, ResultByteSize, ResourceCost, BlockSize, ParallelizationCount$)

Algorithm 20 OPC algorithm - Decrypt data wrapper funtion

$SSGM =$ **ReferenceObject**($StateDataWorker.SecureSubkeyGeneratationModuleObject$)

Require: CipherText 64 bits array and Keys 64 bits array
Ensure: PlainText 64 bits array
$PlainText \in \mathbb{F}_2^{64}$ or $CipherText \in \mathbb{F}_2^{64}$ and $Keys \in \mathbb{F}_2^{64}$
The CommonStatedata is class, The Instance Object Alias Name is CSD
$\mathbb{F}_2^{64} RoundSubkeysCounter = 0$

function SDW.**SplitDataBlockToDecrypt**($CipherText$, $Keys$)
  if CipherText.size() (mod $DataBlockSize$) $\neq 0$ **then**
    return
  end if
  if Keys.size() (mod $DataBlockSize$) $\neq 0$ **then**
    return
  end if
  $Key\_OffsetIndex = 0$
  $KeyDataVector := CSD.WordKeyDataVector$ ▷ Is Object Reference
  $KeyDataVector_0 \dots KeyDataVector_{KeyDataVector.size()} := Keys_0 \dots Keys_{0+KeyDataVector.size()}$
  $Key\_OffsetIndex = Key\_OffsetIndex + KeyBlockSize$
  $RandomKeyDataVector = \{0, 0, 0, 0, 0, \dots | \forall element \in \mathbb{F}_2^{64}, \forall element = 0\}$
  $RandomKeyDataVector$ size is KeyBlockSize × 2
  $\mathbb{F}_2^1$ CCFlag = **true** ▷ The Condition Control Flag
  MersenneTwister64Bit
  for DataBlockOffset = 0; DataBlockOffset < PlainTextSize; DataBlockOffset := DataBlockOffset + DataBlockSize **do**
    if $Key\_OffsetIndex <$ Keys.size() **then**
      $KeySpan \longleftrightarrow \{Keys_{Key\_OffsetIndex} \dots Keys_{Key\_OffsetIndex+KeyBlockSize}\}$
      for Index = 0; Index < KeySpan.size() and Index < KeyDataVector.size(); Index := Index + 1 **do**
        if $KeyDataVector_{Index} = KeySpan_{Index}$ **then**
          $KeyDataVector_{Index} := \neg_{64}(KeyDataVector_{Index} \boxplus_{64} KeySpan_{Index})$
        else
          $KeyDataVector_{Index} := KeyDataVector_{Index} \oplus_{64} KeySpan_{Index}$
        end if
      end for
      $Key\_OffsetIndex := Key\_OffsetIndex + KeyBlockSize$
      SSGM.**GenerationSubkeys**($KeyDataVector$)
      $RoundSubkeysCounter := RoundSubkeysCounter + 1$
    else
      if CCFlag or ((RoundSubkeysCounter (mod 2048 × 4)) == 0) **then**
        for KeyRound = 0; KeyRound < 16; KeyRound := KeyRound + 1 **do**
          for i = 0; i < WordKeyDataVector.size(); i := i + 1 **do**
            $\mathbb{F}_2^{64} a = WordKeyDataVector_i \gg_{64} 32$
            $\mathbb{F}_2^{64} b = WordKeyDataVector_i \wedge_{64} 0x00000000FFFFFFFF$
            $a := a \oplus_{64} b$
            $a := \neg_{64} a$
            $b := b \oplus_{64} a$
            $b := b \lll_{64} 19$
            $a := a \oplus_{64} b$
            $a := a \lll_{64} 13$
            $b := b \oplus_{64} a$
            $b := \neg_{64} b$
            $a := a \oplus_{64} b$
            $a := a \lll_{64} 27$
            $b := b \oplus_{64} a$
            $b := b \lll_{64} 23$ ▷ Apply bitwise operations to diffuse bits
            $WordKeyDataVector_i := (a \lll_{64} 32) \vee_{64} b$
            KeyBytes = $\{0, 0, 0, 0, 0 \dots | \forall element = 0, \forall element \in \mathbb{F}_2^8\}$ size is $KeyBlockSize \times 8$ ▷ 8 is the number of bytes in each 64 bits.
            $KeyBytes :=$ **IntegerToBytes**($WordKeyDataVector$)
            SDW.**ForwardBytesSubstitution**($KeyBytes$) ▷ Call Byte-level data confusion algorithm

$WordKeyDataVector$:= **IntegerFromBytes**($KeyBytes$)
end for ▷ Bit-level data diffusion algorithm
end for
SSGM.**GenerationSubkeys**($WordKeyDataVector$)
CCFlag = false
$RoundSubkeysCounter := RoundSubkeysCounter + 1$
Continue
end if
if RoundSubkeysCounter (mod 2048) = 0 **then**
SaltWordData size is 16, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^{64}\}$
for Index = 0; Index < 16; Index := Index + 1 **do**
$SaltWordData_{index}$:= **MersenneTwister64Bit**()
end for
if RoundSubkeysCounter (mod 2048 × 3) = 0 **then**
SaltData size is 16 × 8, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^{8}\}$
SaltData := **IntegerToBytes**($SaltWorData$)
MaterialKeys = **IntegerToBytes**($RandomKeyDataVector$)
GeneratedSecureKeys size is 0, $GeneratedSecureKeys = \{\varnothing | \forall element = 0, \forall element \in \mathbb{F}_2^{8}\}$
GeneratedSecureKeys := KeyDerivationFunctionObject.**GenerateKeys**($MaterialKeys, SaltData, RandomKeyDataVector.size() \times 8, 1024, 8, 16$)
RandomKeyDataVector = **IntegerFromBytes**($GeneratedSecureKeys$)
SDW.**GenerationSubkeys**(RandomKeyDataVector)
else if RoundSubkeysCounter (mod 2048 × 2) = 0 **then**
SaltData size is 16 × 8, $SaltData = \{0, 0, 0, 0, 0, \ldots | \forall element = 0, \forall element \in \mathbb{F}_2^{8}\}$
SaltData := **IntegerToBytes**($SaltWorData$)
MaterialKeys = **IntegerToBytes**($RandomKeyDataVector$)
GeneratedSecureKeys size is 0, $GeneratedSecureKeys = \{\varnothing | \forall element = 0, \forall element \in \mathbb{F}_2^{8}\}$
GeneratedSecureKeys := KeyDerivationFunctionObject.**GenerateKeys**($MaterialKeys, SaltData, RandomKeyDataVector.size() \times 8, 1024, 8, 16$)
RandomKeyDataVector = **IntegerFromBytes**($GeneratedSecureKeys$)
SDW.**GenerationSubkeys**(RandomKeyDataVector)
Seeds size is KeyBlockSize × 2, $Seeds = \{RandomKeyDataVector_0 \ldots RandomKeyDataVector_{KeyBlockSize-1} | \forall element = 0, \forall element \in \mathbb{F}_2^{64}\}$
MersenneTwister64Bit.**Seed**($Seeds$)
end if
SDW.**GenerationSubkeys**($\varnothing$)
end if
$RoundSubkeysCounter := RoundSubkeysCounter + 1$
end if
$DataSpan \longleftrightarrow \{CipherText_{DataBlockOffset} \ldots CipherText_{DataBlockOffset+DataBlockSize}\}$
SDW.**DecryptingRound**(DataSpan)
end for
if CipherText.size() == DataBlockSize **then**
SDW.**DecryptingRound**(CipherText)
end if
end function

5 Previous studies and discussion

目前在抗量子密码的设计和研究NIST Post-Quantum Cryptography, 以及人们的各种想法都在非对称领域热点中的热点的时候: Post-Quantum Cryptography Standardization,PQC Algorithm Round 1 Submissions,PQC Algorithm Round 2 Submissions,PQC Algorithm Round 3 Submissions,PQC Algorithm Round 4 Submissions. 却很少有人关注抗量子密码在对称领域的影响.

本文作者是一个追求和了解密码学及其发展和设计原则的人. 通过在线课程和入门书籍研究密码学了知识, 包括"CRYPTOGRAPHY I" 和"A Graduate Course in Applied Cryptography (Dan Boneh and Victor Shoup)". 尽管教育和资源有限, 作者还是独立设计并实现了一种对称的加密和解密算法, 被称为 OaldresPuzzle_Cryptic 算法.

另外关于量子计算机分析现有的对称密码学的一些的影响, 因为前人所尝试过使用量子计算机的以及相关的算法, 虽然目前对称领域的加密和解密算法可能不需要太多改进, 但在该领域仍有很多好奇心和实验, 比如使用量子计算机的算力进行密码分析 ([9], [10],[7] [17])

用改进的攻击算法, 针对于量子计算机上运行的平台, 对传统密码的分析 (比如 Simon[22] 和 VQA[26]) 进行密码分析). 本文作者引用了一个综述和结论 ([12] [24]), 使用量子计算机可以对称密码学产生重大影响. 因为由于量子计算机能够比传统计算机代表更多的信息位, 破解的时间复杂度可以滑落到多项式水平, 因此本文对于目前的形式有必要开发更安全的算法. 本文作者参考过到对称领域的各种类型的加密和解密算法, 包括 AES, ARIA, BLOWFISH, TWOFISH, THREEFISH, CAMELLIA, DES(TRIPLE_DES), SERPENT, SM4, IDEA, RC6, CHACHA20, SALEA20, RC4, TRIVIUM, ZUC 等, 他认为这些目前的短密钥的对称加密和解密算法的实现方式, 可能不适合未来的密码学的发展.

本文作者还提供了一份文件 ([23]), 描述了用于加密的对称或非对称密钥的比特长度的演变. 他们强调了量子计算机对对称密码学构成的潜在威胁, 并建议需要新的算法, 使其更加安全并能抵御量子攻击.

本文作者目前需要向专家和公众寻求对其 OPC 算法的评价和反馈. 尽管缺乏深厚的知识和人脉, 但是希望通过足够的计算能力获得实验数据, 比如使用量子计算机或者超级计算机来攻击自己的算法, 这样可以更好地了解其 OPC 算法的缺点. 他们相信在密码学中不断改进和微创新的重要性, 并相信他们的算法虽然为安全牺牲了一些速度, 但是却是向后量子计算机时代的未来密码学发展迈出了微小的一步.

OPC 算法可能不是最快的, 在 2022 年的计算机上用 5KB 的密钥加密或解密 10M 的数据需要 40 多秒, 在 2013 年的计算机上需要 70 多秒,OPC 算法将优先考虑安全性, 而不是考虑运行速度. 预计该算法的性能将在未来 5-10 年内随着计算机性能的增长而提高.

我们设计一种新设计的对称加密和解密算法, 希望其他有能力的人可以进行全面分析. 本研究的目的是评估该算法的有效性和安全性, 并提出支持和反对其设计的核心论点. 在保持技术发展的前提下, 现有的对称算法, 在未来已经不够使用了, 其新设计可以解决这些缺陷. 从展望未来的形式的观点, 现有的对称算法需要改进, 并尝试解决这一问题. 我们将会概述支持和反对该 OPC 算法的论点, 并试图回答任何可能出现的关于其安全性和有效性的问题.

支持该算法的论点:

1. 解决了现有对称算法的不足之处:

密码学依赖于伪随机数发生器的使用, 这些伪随机数发生器是由抽象的、计算上不可区分的单向函数构建的. 一个安全的伪随机数生成器是一个在计算上不可区分的, 这意味着很难预测它的输出. 这是 Lai-Massey 框架的前提, 该框架由两个抽象函数组成：H 函数（代表一个双射转换）和 F 函数（代表一个单射转换）.F-函数的不可预测性通过使用每轮密钥, 允许创建一个安全的子密钥, 使得对手很难在多项式时间内检索到原始的密钥数据. 作者还指出, 由于对正在处理的数据的每一半使用相同的子密钥以及 F-函数的不可预测性, 被操作的数据被认为是计算安全的. 关于该框架的详细信息, 请翻阅之后介绍 Lai-Massey 框架的章节附带的参考文献.

2. 注重安全:

作者在设计这个算法时把安全作为优先考虑的因素. 他们为了安全而牺牲了速度, 试图创造一个更强大、更可靠的加密和解密方法.

该算法遵守了密码学的框架, 基于密码学最基本的单向功能, 一步步建立起来的.

利用他们的密码学知识和对数学函数属性的基本理解, 设计了一种建立在不可预测的伪随机数生成器并且配合莱马西对称加密解密框架的算法.

这为该算法提供了一个坚实的基础, 有助于其安全性的评估.

根据上面一点, 只要在 Lai-Massey 框架内设计的 F 函数在计算上是不可区分的, 整个框架就可以被认为是安全的, 由于 F 函数中每轮应用该 OPC 算法的生成密钥的所有框架. 使用生成的不同的密钥. 导致正在处理的数据的不可预测性, 使加密和解密过程变得安全. 即使在计算上对加密数据和随机数据进行区分, 也无法在多项式复杂时间内区分.

反对该算法的论点:

1. 缺少测试:

作者没有机会使用量子计算机或超级计算机来测试该算法, 这限制了他们全面评估其安全性的能力.

与现有的算法相比没有明显的改进: 目前还不清楚新算法是否比现有的对称加密和解密算法有任何重大改进.

这让人对这种新算法的必要性及其对密码学领域的潜在影响产生疑问.

2. 知识和经验不足:

作者在密码学方面的知识和经验有限, 这让人对该算法的有效性产生了担忧. 作者希望有关专业人士能够提供帮助

要求说明理由:

本文作者正在寻求密码学领域专业人士的帮助, 以评估他们独立设计的一种新的对称加密和解密算法.

尽管资源和人脉有限, 但本文提出的 OPC 算法是向微创新迈出的一步, 是跟上技术快速发展步伐的需要.

作者认识到, 他们在密码学方面的知识和经验可能不如该领域的其他人丰富, 他们希望对其算法的公开评估将帮助他们更好地了解其优势和局限. 作者请求密码学领域的专业人士花时间评估他们的算法, 并对其有效性提供反馈.

这种反馈可以包括改进的建议, 对其安全性和速度的严格评估, 或任何其他有助于作者更好地了解其算法的优势和劣势的相关信息.

测试如果是在算力很小的传统计算机上, 用 10MB 的数据和 5KB 的主密钥作为输入然后输出改变过的 10M 数据是比较合适的. 针对超级计算机或量子计算机测试可以尝试更大的数据, 更长的密钥. 便于与找到这个算法的缺点和安全性.

他们正在寻求支持和资源来进行这种类型的测试, 他们认为这对于全面评估他们算法的安全性和促进密码学的成长和发展至关重要.

总之, 作者正在寻求密码学界的支持和指导, 以帮助他们进一步了解密码学和他们在其中的地位. 作者希望他们的工作是朝着开发更安全、更高效的对称加密和解密算法迈出的重要一步, 他们希望得到必要的支持和指导, 以使其取得成果.

5.1 Try to prove, using mathematics, why OPC symmetric encryption and decryption algorithm, is cryptographically secure?

A Documents referenced

参考文献

[1] Fırat Artuğer and Fatih Özkaynak. A method for generation of substitution box based on random selection. *Egyptian Informatics Journal*, 23(1):127–135, 2022.

[2] Fabio Borges, Paulo Ricardo Reis, and Diogo Pereira. A comparison of security and its performance for key agreements in post-quantum cryptography. *IEEE Access*, 8:142413–142422, 2020.

[3] Amit Kumar Chauhan and Somitra Sanadhya. Quantum security of fox construction based on lai-massey scheme. Cryptology ePrint Archive, Paper 2022/1001, 2022. https://eprint.iacr.org/2022/1001.

[4] Guillermo Cotrina, Alberto Peinado, and Andrés Ortiz. Gaussian pseudorandom number generator using linear feedback shift registers in extended fields. *Mathematics*, 9(5):556, 2021.

[5] Joan Daemen and Vincent Rijmen. *The design of Rijndael*, volume 2. Springer, 2002.

[6] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Collision-free hashing from lattice problems. *IACR Cryptol. ePrint Arch.*, 1996:9, 1996.

[7] Marc Kaplan, Gaëtan Leurent, Anthony Leverrier, and María Naya-Plasencia. Breaking symmetric cryptosystems using quantum period finding. In *Advances in Cryptology – CRYPTO 2016*, pages 207–237. Springer Berlin Heidelberg, 2016.

[8] Kazys Kazlauskas and Jaunius Kazlauskas. Key-dependent s-box generation in aes block cipher system. *Informatica, Lith. Acad. Sci.*, 20:23–34, 01 2009.

[9] Hidenori Kuwakado and Masakatu Morii. Quantum distinguisher between the 3-round feistel cipher and the random permutation. In *2010 IEEE International Symposium on Information Theory*, pages 2682–2685, 2010.

[10] Hidenori Kuwakado and Masakatu Morii. Security on the quantum-type even-mansour cipher. In *2012 International Symposium on Information Theory and its Applications*, pages 312–316, 2012.

[11] Yiyuan Luo, Xuejia Lai, and Yujie Zhou. Generic attacks on the lai–massey scheme. *Designs, Codes and Cryptography*, 83, 05 2017.

[12] Ashwini Kumar Malviya, Namita Tiwari, and Meenu Chawla. Quantum cryptanalytic attacks of symmetric ciphers: A review. *Computers and Electrical Engineering*, 101:108122, 2022.

[13] Shuping Mao, Tingting Guo, Peng Wang, and Lei Hu. Quantum attacks on lai-massey structure. Cryptology ePrint Archive, Paper 2022/986, 2022. https://eprint.iacr.org/2022/986.

[14] Chandra Sekhar Mukherjee, Dibyendu Roy, and Subhamoy Maitra. Design specification of zuc stream cipher. In *Design and Cryptanalysis of ZUC*, pages 43–62. Springer, 2021.

[15] Chokri Nouar and Z. Guennoun. A pseudo-random number generator using double pendulum. *Applied Mathematics Information Sciences*, 14:977–984, 11 2020.

[16] Stjepan Picek, Lejla Batina, Domagoj Jakobović, Barış Ege, and Marin Golub. S-box, SET, Match: A Toolbox for S-box Analysis. In David Naccache and Damien Sauveron, editors, *8th IFIP International Workshop on Information Security Theory and Practice (WISTP)*, volume LNCS-8501 of *Information Security Theory and Practice. Securing the Internet of Things*, pages 140–149, Heraklion, Crete, Greece, June 2014. Springer. Part 5: Short Papers.

[17] Richard Preston. Applying grover's algorithm to hash functions: A software perspective, 2022.

[18] Tomasz Rachwalik, Janusz Szmidt, Robert Wicik, and Janusz Zablocki. Generation of nonlinear feedback shift registers with special-purpose hardware. Cryptology ePrint Archive, Paper 2012/314, 2012. https://eprint.iacr.org/2012/314.

[19] Maximilian Richter, Magdalena Bertram, Jasper Seidensticker, and Alexander Tschache. A mathematical perspective on post-quantum cryptography. *Mathematics*, 10(15), 2022.

[20] Ronald L Rivest, Matthew JB Robshaw, Ray Sidney, and Yiqun Lisa Yin. The rc6tm block cipher. In *First advanced encryption standard (AES) conference*, page 16, 1998.

[21] M. R. Mirzaee Shamsabad and S. M. Dehnavi. Lai-massey scheme revisited. Cryptology ePrint Archive, Paper 2020/005, 2020. https://eprint.iacr.org/2020/005.

[22] D.R. Simon. On the power of quantum computation. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 116–123, 1994.

[23] Nigel P Smart and Emmanuel Thomé. History of Cryptographic Key Sizes. In Joppe Bos and Martijn Stam, editors, *Computational Cryptography*, volume 469 of *London Mathematical Society Lecture Note Series*. Cambridge University Press, December 2021.

[24] Han SUI and Wenling WU. Research status and development trend of post-quantum symmetric cryptography. *Journal of Electronics & Information Technology*, 42(190667):287, 2020.

[25] Kethan Vooke, Nithin Kumar Toramamidi, Kalyan Kumar Thodeti, and Sangeeta Singh. Design of pseudo-random number generator using non-linear feedback shift register. In *2022 First International Conference on Electrical, Electronics, Information and Communication Technologies (ICEEICT)*, pages 1–5, 2022.

[26] Zeguo Wang, Shijie Wei, Gui Long, and L. Hanzo. Variational quantum attacks threaten advanced encryption standard based symmetric cryptography. *Science China Information Sciences*, 65:200503, 10 2022.

[27] Aaram Yun, Je Hong Park, and Jooyoung Lee. Lai-massey scheme and quasi-feistel networks. Cryptology ePrint Archive, Paper 2007/347, 2007. https://eprint.iacr.org/2007/347.

B Theories referenced and used

本文将用到以下知识点、技术和内容. 请读者自行理解相关概念后, 自行查找参考资料、书籍资料. 开始阅读本文.

Algorithms and Data Structures: 算法是一组指令或一个用于解决问题或完成任务的分步程序. 在密码学方面, 算法是用于加密和解密数据的数学程序. 不同的加密算法使用不同的技术, 如替换、换位和模块化算术, 将明文转换为密文, 反之亦然. 数据结构用于以特定方式组织和存储数据, 使其易于访问、修改和处理数据. 在密码学的背景下, 数据结构可用于存储加密密钥、中间值以及加密和解密过程中使用的其他数据. 加密学中使用的数据结构的例子包括数组、链接列表和树.OaldresPuzzle_Cryptic 算法利用各种数据结构和算法来创建一个几乎不可破解的独特的加密-解密密钥. 它使用行树数据结构和动态生成的字节替换框, 使每个生成的密钥都是不可预测的. 它还利用各种数学运算和算法, 包括线性代数, 如仿生变换、克朗克积、点积、求解转置和邻接矩阵, 以及矩阵的加、减、乘法. 这些子密钥生成模块是由 Lai-Massey 计划协调和设计的.

Ciphers: 密码器是一种数学算法, 用于加密和解密数据. 密码被用来将明文转换为密文, 并将密文转换为明文. 它们被用来确保数据的保密性和完整性, 以便只有授权方能够阅读和理解原始信息.

Plaintext and Ciphertext: 明文是要传输或存储的未加密的原始信息. 它可以是任何形式的数据, 如文本、图像或音频. 密文是使用特定加密算法和密钥对明文进行加密的结果. 密文是明文的加扰版本, 如果没有相应的解密密钥或算法, 就很难或无法阅读或理解. 加密的主要目标是将明文转化为密文, 使只有授权方能通过解密来读取原始信息. 将明文转化为密文的过程称为加密, 而将密文转化回明文的反向过程称为解密.

The relationship between Ciphers for encryption and decryption: 密码器是用于加密和解密数据的数学算法. 加密过程是使用特定的加密算法和密钥将明文转换为密文的过程. 解密过程是加密的反向过程, 它是使用特定的解密算法和密钥将密码文本转换为明文的过程. 加密和解密过程密切相关, 因为加密是将明文转换为密码文本的过程, 解密是将密码文本转换为明文的过程. 加密过程和解密过程通常由不同的一方进行, 发送方对信息进行加密, 接收方对信息进行解密. 加密和解密过程使用相同的数学算法, 但用于加密的密钥与用于解密的密钥不同. 例如, 在对称密钥密码中, 加密和解密使用相同的密钥, 而在非对称密钥密码中, 使用两个不同的密钥, 一个用于加密, 另一个用于解密.OaldresPuzzle_Cryptic 算法是一种对称密钥密码算法, 它使用一个密钥来加密明文, 并使用同一密钥来解密密文.

Keys and Subkeys: 在加密方面, 密钥是一个值或一组值, 用于加密和解密数据. 密钥在加密算法中用于将明文转化为密文, 反之亦然. 密钥是加密过程中的一个关键因素, 因为它决定了加密的安全程度. 子密钥也称为圆密钥, 是由秘密密钥衍生出来的, 通常是通过一个密钥计划算法. 它们被用于许多加密算法中, 特别是那些使用多轮加密的算法. 子密钥用于在加密过程的不同阶段对数据进行加密, 通过使攻击者更难确定用于加密数据的密钥, 为加密算法增加了更多的安全性. 子密钥是一个派生密钥, 用于在加密算法的特定回合中对数据进行加密. 子密钥通常是通过密钥计划算法从主密钥衍生出来的, 该算法用于为每一轮加密生成新的密钥. 子密钥的生成方式与主密钥不同, 使攻击者更难确定用于加密数据的主密钥.

Symmetric-key ciphers: 对称密钥密码使用相同的密钥进行加密和解密. 对称密钥密码的例子包括 AES、DES 和 Blowfish.

Asymmetric-key ciphers: 非对称密钥密码使用两个不同的密钥, 一个用于加密, 一个用于解密. 非对称密钥密码的例子包括 RSA 和椭圆曲线加密法（ECC）.

Block ciphers: 分块密码是一种对称密钥密码, 它一次加密一个固定大小的数据块, 而不是一个数据流. 分块密码被广泛用于各种应用, 包括文件加密和安全通信. 它们适用于加密大量数据, 如文件或磁盘分区. 分块密码的例子包括 AES [5] 和 DES, RC6[20].

Stream ciphers: 流密码是一种对称密钥密码, 每次对一个比特或字节的数据流进行加密. 流密码被用于各种应用, 包括无线通信和移动网络. 它们适用于加密实时数据流, 如音频或视频. 流密码的例子包括 RC4 和 Salsa20.

Key size: 密钥大小是对加密密钥中的比特数的一种衡量. 密钥大小是决定加密算法安全性的一个重要因素, 因为较大的密钥大小可以使通过暴力攻击破解加密更加困难.

Chaos theory: 混沌理论是数学的一个分支, 研究对初始条件高度敏感的动态系统的行为, 也被称为蝴蝶效应. 混沌理论已被应用于各个领域, 包括密码学, 它可以被用来生成难以预测的伪随机数.

Cryptography based on chaos theory: 是一个新的研究领域, 利用混沌系统的特性来产生安全的加密密钥.

Cryptography based on lattices: 格密码学是一种基于格的数学属性的密码学形式, 格是多维空间中的离散点集. 格密码学被认为是一种很有前途的后量子密码学方法, 这意味着它被认为可以抵御量子计算机攻击. 在格密码学中, 加密和解密过程都是基于格的运算, 例如最短向量问题（SVP）和最近向量问题（CVP）. 这些问题对于量子计算机来说是很难解决的, 这使得基于格的加密方案能够抵抗量子攻击. 基于格的密码学的一个例子是错误学习

(LWE) 问题, 它是许多加密方案（例如 NTRU 和 Ring-LWE）的基础.Learing With Error 问题基于求解晶格上的线性方程组的难度, 其中方程是随机选择的. 另一个例子是 Ring-LWE, 这是一种基于格的加密方案, 它基于解决多项式环上 LWE 问题变体的难度. [4] [2]

Ajtai's hash function: Ajtai 的哈希函数是 Miklos Ajtai 在 1996 年提出的一种密码学哈希函数. 它是一种单向函数, 它接受任意长度的输入并产生固定长度的输出, 称为哈希或摘要. 输出被设计成独一无二的, 这意味着即使对输入进行微小的更改也会导致完全不同的输出.Ajtai 的哈希函数基于抗碰撞函数的概念, 这意味着要找到产生相同输出的两个输入在计算上是不可行的. 它还被设计为具有原像抗性, 这意味着找到产生特定输出的输入在计算上是不可行的. [6]

Lai-Massey Scheme: Lai-Massey sheme 是一种用于设计密码系统的技术, 它是设计密码系统和分析其安全性的有力工具.

Linear algebra: 线性代数是数学的一个分支, 涉及矢量空间和线性变换. 线性代数被用于数学和科学的许多领域, 包括密码学、计算机图形学和机器学习. 线性代数操作可用于以各种方式处理矩阵和向量, 如解决线性方程组、计算行列式和特征值, 以及进行矩阵乘法和矩阵乘法逆元.

Affine transformations: 仿射变换是线性代数中的一种变换类型, 它保留了碰撞性（即在同一直线上的点仍然在同一直线上）和距离比（即保留了各点之间的距离比）的事实. 仿生变换由一个矩阵和一个矢量定义, 可以包括诸如平移、旋转、缩放和剪切等操作.

Kronecker product: 克罗内克积, 又称张量积, 是矩阵的二元运算, 通过一个矩阵的每个元素与另一个矩阵的每个元素的外积产生一个新矩阵. 克朗克积可以用符号 $\otimes$ 表示, 其定义为: $C = A \otimes B = [a_{i,j}B]$ (我们在本文中不这样表示, 因为数学图形符号太少, 容易产生歧义.) 其中 A 是一个 m × n 矩阵,B 是一个 p × q 矩阵, 而 C 是一个 mp × nq 矩阵.A 和 B 的克朗克乘积是通过将矩阵 B 沿行复制 m × p 次, 沿列复制 n × q 次, 然后将结果与矩阵 A 的元素相乘而形成的. 克朗克乘积是一个强大的运算, 可用于建立广泛的数学和物理系统模型, 包括线性系统、非线性系统和信号处理系统.

Dot product: 点积, 也被称为标量积, 是两个向量之间的一种运算, 其结果是一个标量值. 两个向量的点积是通过乘以相应的条目, 然后将结果相加来计算的. 两个向量的点积可用于确定它们之间的角度, 并可用于各种数学运算.

Transpose and adjoint matrices: 矩阵的转置是一个新的矩阵, 它是由原矩阵围绕其主对角线翻转而形成的. 矩阵的邻接点是矩阵的共轭转置. 这些操作可以用来将矩阵转换为不同的形式, 以便于进行特定的操作.

Group theory: 群论是数学的一个分支, 涉及对群的研究, 群是具有特定运算的元素集合, 满足某些属性. 群论被用于数学和科学的许多领域, 包括密码学. 在密码学中, 群论被用于设计和分析各种类型的加密算法, 如对称密钥密码和公开密钥密码. 例如, 许多对称密钥密码的安全性是基于解决某些属于特定组的数学问题的难度.

Finite Field: 有限域, 也被称为伽罗瓦域, 是一种数学结构, 由有限数量的元素和一组可对这些元素进行的数学运算组成. 有限域被用于数学的许多领域, 包括数论、编码理论和密码学.

Information theory: 信息理论是数学的一个分支, 涉及到信息的表示、传输、处理和解释. 它与密码学密切相关, 因为它涉及熵和信息熵的概念, 这在分析加密算法时非常重要.

Shift Register: 移位寄存器是一种数字电路, 可用于存储和操作多个数据位. 移位寄存器经常被用于数字电路和密码学中, 作为一种简单而有效的方法来生成一串伪随机数.

Feedback shift register (FSR): 反馈移位寄存器是一种具有反馈回路的移位寄存器, 最后一级的输出被反馈为第一级的输入. 反馈移位寄存器经常被用来产生伪随机数或伪随机位序列.

Linear feedback shift register (LFSR): 线性反馈移位寄存器是一个具有线性反馈功能的移位寄存器. 线性反馈移位寄存器经常被用于数字电路和密码学中, 作为一种简单而有效的方法来生成一串伪随机数.

Linear systems and Nonlinear systems: 线性系统和非线性系统是两种类型的数学系统, 它们描述了不同的物理和数学现象的行为. 线性系统是遵循线性方程的系统, 这些方程的特性是两个解的和也是一个解, 而一个解与一个标量的乘积也是一个解. 线性系统具有简单的数学结构, 它们相对容易分析和控制. 线性系统的例子包括线性微分方程、线性微分代数方程、线性差分方程和线性代数方程. 另一方面, 非线性系统是遵循非线性方程的系统, 这些方程不具有线性方程的特性. 非线性系统具有更复杂的数学结构, 它们更难控制. 这不能用线性数学来建模或分析. 非线性系统的例子包括非线性微分方程、非线性微分-代数方程、非线性差分方程和非线性代数方程. 它们表现出复杂的行为, 通常以多个平衡点和极限循环的存在为特征. 在密码学中, 线性系统可能容易受到线性密码分析和代数攻击, 而非线性系统对这些类型的攻击更有抵抗力.OaldresPuzzle_Cryptic 算法利用一个具有混沌特性的非线性反馈移位寄存器, 一个静态字节替换盒来模拟非线性强函数, 以及一个动态字节替换盒. 此外, 它还利用了各种数学运算, 包括线性代数, 如仿射变换、克朗克积、点积、解决转置和邻接矩阵, 以及矩阵的加减和乘法. 这些设计选择使 OaldresPuzzle_Cryptic 算法成为一个非线性系统, 对线性和代数攻击的抵抗力更强.

Nonlinear feedback shift register (NLFSR): 非线性反馈移位寄存器（NLFSR）是一种具有非线性反馈功能的移位寄存器. 与具有线性反馈功能的线性反馈移位寄存器（LFSRs）不同,NLFSRs 使用非线性功能来生成序列中的下一个位.NLFSRs 可以产生更复杂和更难预测的比特序列, 使它们更难预测, 更适合用于加密应用. 它们可以被设计成表现出混沌行为, 使它们更适合用于基于混沌的密码学. 该算法还利用了一个线性反馈移位寄存器, 其序列周期长度为 2 到 128 次方.LFSR 和 NLFSR 的结合创造了一个更加稳健和不可预测的比特序列, 可以作为加密的密钥. [25] [4] [18]

Pseudo-random number generators (PRNGs): 伪随机数生成器是产生与真正随机数序列在统计学上相似的数字序列的算法.PRNG 经常被用于密码学中, 以生成加密密钥.

Pseudorandomness: 伪随机数列是一个看起来是随机的, 但却是由一个确定的过程产生的. 伪随机数在密码学中被广泛使用, 它们被用来生成加密密钥.

Byte substitution box (S-box): 字节置换盒是许多加密算法的一个组成部分, 它使用一个固定的表格将输入值映射到输出值.S-boxes 经常被用来在加密算法中提供扩散和混乱, 通过使攻击者难以确定明文和密码文本之间的关系.

Confusion and diffusion: 混乱和扩散是加密算法的两个重要属性. 混淆指的是明文和密文之间的关系是复杂和难以确定的, 而扩散指的是明文的小变化会导致密文的大变化.

Key schedule: 密钥安排是一种算法, 用于将短的加密密钥扩展为长的密钥, 以便在区块密码中使用. 密钥计划是区块密码的一个重要组成部分, 因为它可以影响密码的安全性.

ZUC sequence cipher design: ZUC 是一种用于无线通信和移动网络的流密码, 由中国人创造, 用于商业. 它是基于一个序列发生器, 通过使用非线性运算、位运算和模块化加法等操作产生一个密钥流.[14]

Line segment tree data structure: 线段树是一种数据结构, 用来表示元素的序列. 它是前缀树的一个概括. 它是一种树形数据结构, 可以用来表示元素的序列, 通常是字符或单词.

Evaluation and testing of encryption-decryption algorithms: 为了评估加解密算法的安全性和有效性, 必须使用各种参数对其进行测试, 如密钥大小、加解密时间和对各种已知攻击的抵抗力, 包括量子计算攻击.

Encryption-decryption time: 加密-解密时间是衡量使用特定加密算法对信息进行加密或解密所需的时间. 这是选择加密算法时要考虑的一个重要因素, 因为更快的加密-解密时间对某些应用来说可能更实用.

Cryptographic secureness: 加密系统的一种属性, 确保攻击者在不掌握某些秘密信息（如密钥）的情况下, 从密码文本或所用密钥中恢复明文在计算上是不可行的.

Methods of attacking ciphers: 有几种方法可以用来攻击密码并试图恢复加密过程中使用的明文或密钥. 一些常见的方法包括. 蛮力攻击. 蛮力攻击是一种攻击类型, 攻击者尝试所有可能的密钥, 直到找到正确的密钥. 这种方法非常耗时, 但如果密钥空间较小, 它也是有效的. 已知明文攻击. 已知明文攻击是一种攻击类型, 其中攻击者可以获得密码文本和相应的明文. 攻击者使用这些信息来试图确定加密过程中使用的密钥. 选择明文攻击是一种攻击类型, 攻击者可以选择要加密的明文, 然后试图确定加密过程中使用的密钥. 差分密码分析是一种攻击类型, 它使用两个明文和其相应的密码文之间的差异, 试图确定加密过程中使用的密钥. 线性密码分析是一种攻击类型, 它使用加密函数的线性近似, 试图确定加密过程中使用的密钥. 代数攻击是对加密算法的一种攻击, 它利用了算法的数学结构. 这些攻击可以包括线性和微分密码分析等技术, 以及对一个区块的密钥安排的代数攻击. 量子攻击. 量子计算机有能力解决某些问题, 比经典计算机快得多, 这对经典加密算法构成了威胁. 量子攻击包括 Shor 的算法,Grover 的算法和其他. 侧信道攻击是一种利用从加密系统的物理实现中泄露的信息的攻击, 如定时信息、功耗或电磁辐射. 这些攻击可以让攻击者提取加密过程中使用的密钥. 社会工程攻击. 社会工程攻击是一种攻击类型, 它使用心理操纵来欺骗用户透露敏感信息, 如加密密钥或密码. 字典攻击. 词典攻击是一种攻击类型, 攻击者使用预先计算好的常用单词、短语和模式的词典, 试图找到加密密钥. 重要的是要注意, 密码的安全性不仅由加密算法本身的强度决定, 而且还由加密过程中使用的密钥的强度、算法的实现以及使用该算法的整个系统的安全性决定.

Resistance to quantum computing attacks: 由于量子计算机有能力比经典计算机更快地解决某些问题, 因此设计能抵抗量子计算攻击的加密算法很重要. 这可以通过使用在量子计算机上难以解决的数学运算来实现通过使用足够大的加密密钥, 使暴力攻击不可行. [2] [19]

B.1 Definition of necessary concepts and mathematical symbols

为了我们的算法流程, 可以被方便的解释, 我们还需要定义以下基本概念:

1. 字节和比特

假设 values 是一个有限元素的一维集合而且大小于 9 , 但是每一个元素只能是 0 或者 1 . 这个 values 集合的左边是最高位, values 集合的右边是最低位. 当其中一个元素超过了 1 , 那么这个元素需要变为 0 , 接着把 1 加法运算给下一个高位.(依次类推). 在 values 集合中的数字元素 0 和 1. 它们建立了满二进一的关系, 这个就是二进制的表示法,0 和 1 是最基本的表示单位被称为比特. values 集合就是二进制表示的 8 个比特, 然后组成的 1 个字节. 这一种表达的方式可以表示为 0 到 $2^8 - 1$ 的数字.

$$values = \{0,0,0,0,0,0,0,0\} \quad or \quad values = \{1,1,1,1,1,1,1,1\}$$

$$values \in \{0,1\} \quad and \quad (\text{values size} < 9)$$

Example:

$$\{0,0,0,0,0,0,1,0\} = values = \{0,0,0,0,0,0,0,1\} + \{0,0,0,0,0,0,0,1\}$$

$$\{0,0,0,0,0,0,0,0\} = values = \{0,0,0,0,0,0,0,1\} - \{0,0,0,0,0,0,0,1\}$$

2. 比特字

1 个比特字是 16 个比特单位组成的有限集合, 或者 2 个字节元素拼接的有限集合. 1 个比特双字是 32 个比特单位组成的有限集合, 或者 4 个字节元素拼接的有限集合. 1 个比特四字是 64 个比特单位组成的有限集合, 或者 8 个字节元素拼接的有限集合.

3. 十六进制

在数学和计算中, 十六进制 (也称为 16 进制或简称十六进制) 数字系统是一种位置数字系统, 它使用基数（基数）为 16 来表示数字.

与使用 10 个符号表示数字的十进制系统不同, 十六进制使用 16 个不同的符号, 最常见的符号"0"-"9" 表示值 0 到 9,"A"-"F"(或替代 "a" - "f) 表示 10 到 15 之间的值.

软件开发人员和系统设计人员广泛使用十六进制数, 因为它们提供了二进制编码值的人性化表示. 每个十六进制数字代表四个位 (二进制数字), 也称为半字节 (或 nybble).

例如, 一个 8 位字节的值范围从 00000000_2 到 11111111_2 的二进制形式, 可以方便地表示为十六进制的 00_{16} 到 FF_{16}

在数学中, 下标通常用于指定基数. 例如, 十进制值 43838 将以十六进制表示为 $AB3E_{16}$.

在计算机编程语言中, 许多符号用于表示十六进制数字, 通常涉及前缀.

前缀在 C/C++ 编程语言中有广泛使用, 0xAB3E 表示此值为 43838

注意:

以上的下标的定义, 来区分这个值是用什么进制的系统, 容易和我们即将要定义运算符号发生意思冲突. 所以我们基本上会使用前缀的方法来解释这个值的进制系统到底是什么.

0b 表示二进制的前缀, 0b1010110001011001 表示十进制数字 44121 .

0x 表示十六进制的前缀, 0x123456 表示十进制数字 1193046 .

为了我们的算法流程, 可以被方便的解释, 我们还需要定义以下符号:

这些运算的操作对象 a, b, c. 大小都是 1 个比特字或者 1 个比特双字或者 1 个比特四字 (32 是比特单位的大小, 可以是 8, 16, 32, 64 这些数字).

$left \pmod{right}$ 它表示通过模数 right, 由 left 除以 right 然后获取余数, 而且 $right > 0$.
举个例子: $255 \equiv 4 \pmod{251}4$ 是余数
$c = a \boxplus_{32} b$: 它表示带有模数的加法运算. $c = a + b \pmod{2^{32}}$
$c = a \boxminus_{32} b$: 它表示带有模数的减法运算. $c = a - b \pmod{2^{32}}$
$c = a \boxtimes_{32} b$: 它表示带有模数的乘法运算. $c = a \times b \pmod{2^{32}}$

我们定义二进制的**比特按位运算**.

以下表达中后缀 SN 意思是有符号数类型 (可能是正数或者负数) 使用最高的比特位来表示符号位, 如果值为 1 就是负数, 否则值为 0 就是正数, 后缀 USN 意思是无符号数类型 (一定是正数).

$$bits(\text{bits size} < 9) \in \{-128, 127\}(SN) \quad bits(\text{bits size} < 9) \in \{0, 255\}(USN)$$

其中按位运算的过程将会把一个或者两个比特位集合的操作数, 作为纯比特位数据集合进行运算处理举个例子: 两个操作数, 定义为 a 和 b, 结果为 c. (条件一: 必须两个比特集合大小相等)

$$a = \{1, 0, 1, 0, 1, 1, 0, 0\}(172USN) \quad (\text{a size} < 9) \quad b = \{0, 1, 0, 0, 0, 1, 0, 1\}(69USN) \quad (\text{b size} < 9)$$
$$a = \{1, 0, 1, 0, 1, 1, 0, 0\}(-84SN) \quad (\text{a size} < 9) \quad b = \{0, 1, 0, 0, 0, 1, 0, 1\}(69SN) \quad (\text{b size} < 9)$$

那么无论 a 和 b 之前的类型是 SN 还是 USN, 运算结果都不由 a 和 b 之前所代表的类型决定, 而是取决于 c 本身属于有符号类型还是无符号类型, 来决定运算结果是正数还是负数. (条件二: 需要运算左右两边的两个比特都在同一个比特集合的位置 $bits_{index} \quad operator \quad bits_{index}$)
只有满足条件一和条件二的情况下, 以下运算才能成立.

$c = a \wedge_{32} b$: 它表示二进制的**与运算**.
具体规则是, 当两个比特都是 1 的时候, 它的运算结果就为 1; 当两个比特都不是 1 的时候, 它的运算结果就为 0 . 详细操作请看公式例子说明.

$$c = \{0, 0, 0, 0, 0, 1, 0, 0\}(4USN \quad or \quad SN) = a \wedge_8 b$$

$c = a \vee_{32} b$: 它表示二进制的**或运算**.
具体规则是, 当两个比特中任意一个比特是 1 的时候, 它的运算结果就为 1; 当两个比特都是 0 的时候, 它的运算结果就为 0 . 详细操作请看公式例子说明.

$$c = \{1, 1, 1, 0, 1, 1, 0, 1\}(237USN \quad or \quad -19SN) = a \wedge_8 b$$

$bit' = \neg_{32} bit$: 它表示二进制的**非运算**.
具体规则是, 当前比特中是 1 的时候, 它的运算结果就为 0 ; 当前比特中是 0 的时候, 它的运算结果就为 1 . 详细操作请看公式例子说明.

$$bits' = \{1, 0, 0, 0, 1, 1, 1, 0\}(142USN) = \neg_{32} bits\{0, 1, 1, 1, 0, 0, 0, 1\}(113USN)$$
$$bits' = \{0, 1, 0, 1, 0, 0, 1, 1\}(83SN) = \neg_{32} bits\{1, 0, 1, 0, 1, 1, 0, 0\}(-84SN)$$

$c = a \oplus_{32} b$: 它表示二进制的**异或运算**.
具体规则是, 需要运算左右两边的两个比特都在同一个比特集合的位置, 当两个比特相同的时候, 它的运算结果就为 0; 当两个比特不同的时候, 它的运算结果就为 1. 详细操作请看公式例子说明.

$$c = \{1, 1, 1, 0, 1, 0, 0, 1\}(233USN \quad or \quad -23SN) = a \oplus_8 b$$

$c = a \odot_{32} b$: 它表示二进制的**同或运算**.
具体规则是, 当两个比特相同的时候, 它的运算结果就为 1 ; 当两个比特不同的时候, 它的运算结果就为 0 . 详细操作请看公式例子说明.

$$c = \{0, 0, 0, 1, 0, 1, 1, 0\}(22USN \quad or \quad SN) = a \odot_8 b$$
$$c = a \odot_8 b = \neg_8(a \oplus_8 b) = (a \oplus_8 \neg_8 b) = (\neg_8 a \oplus_8 b)$$

我们定义二进制的**比特移位运算**.

如果操作的这个数是有符号数, 那么左移操作将会丢弃最高位 (符号位) 和左移动比特数据, 反之右移操作就会保留最高位 (符号位) 和右移动比特数据并且丢弃最低位, 这个运算被称为**算术移位**.

如果操作的这个数是有无符号数, 那么左移操作将会丢弃最高位和左移动比特数据, 反之右移操作就会保留符号位和右移动比特数据并且丢弃最低位, 这个运算被称为**逻辑移位**.

以上操作定义, 我们都在一个比特集合里面进行操作, 如果这个操作导致超出了比特有限集合的大小范围, 那么这些比特会将会被丢弃.

$bits' = bits \ll_{32} number$：它表示向左移位运算.

具体规则是,bits 是比特字以及类似单位的比特集合, number 是向左移位的比特数量. 如果要防止这个运算的结果未定义, 在这个例子中 number 必须满足 $number = number \pmod{32}$, 详细操作请看公式例子说明.

$$bits = \{0,1,0,1,0,0,0,1\}(81USN) \quad (\text{bits size} < 9)$$
$$bits' = \{0,1,0,0,0,1,0,0\}(68USN) = bits \ll_8 2$$
$$bits = \{0,1,0,1,0,0,0,1\}(81SN) \quad (\text{bits size} < 9)$$
$$bits' = \{1,0,1,0,0,0,1,0\}(-94SN) = bits \ll_8 1$$

$bits' = bits \gg_{32} number$：它表示向右移位运算.

具体规则是,bits 是比特字以及类似单位的比特集合, number 是向右移位的比特数量. 如果要防止这个运算的结果未定义, 在这个例子中 number 必须满足 $number = number \pmod{32}$, 详细操作请看公式例子说明.

$$bits = \{1,1,0,1,0,1,1,0\}(214USN) \quad (\text{bits size} < 9)$$
$$bits' = \{0,0,1,1,0,1,0,1\}(53USN) = bits \gg_8 2$$
$$bits = \{1,1,0,1,0,1,1,0\}(-42SN) \quad (\text{bits size} < 9)$$
$$bits' = \{1,1,1,0,1,0,1,1\}(-21SN) = bits \gg_8 1$$

我们定义二进制的**循环比特移位运算**.

运算过程与前面的**比特移位运算**很像, 但是**不丢弃任何一个比特位**

$bits' = bits \lll_{32} number$：它表示向左循环移位运算.

具体规则是,bits 是比特字以及类似单位的比特集合, number 是向左移位的比特数量. 如果要防止这个运算的结果未定义, 在这个例子中 number 必须满足 $number = number \pmod{32}$ 详细操作请看公式例子说明.

$$bits = \{1,0,1,1,0,1,0,1\}(181USN) \quad (\text{bits size} < 9)$$
$$bits' = \{1,0,1,1,0,1,0,1\}(181USN) = bits \lll_8 8$$
$$bits = \{1,0,1,0,0,1,0,1\}(-91SN) \quad (\text{bits size} < 9)$$
$$bits' = \{1,1,0,1,0,0,1,0\}(-46SN) = bits \lll_8 7$$

运算过程与前面的**比特移位运算**很像, 但是**不丢弃任何一个比特位**

$bits' = bits \ggg_{32} number$：它表示向右循环移位运算.

具体规则是,bits 是比特字以及类似单位的比特集合, number 是向左移位的比特数量. 如果要防止这个运算的结果未定义, 在这个例子中 number 必须满足 $number = number \pmod{32}$ 详细操作请看公式例子说明.

$$bits = \{1,1,0,1,0,1,1,0\}(214USN) \quad (\text{bits size} < 9)$$
$$bits' = \{1,1,0,1,0,1,1,0\}214USN) = bits \ggg_8 8$$
$$bits = \{1,1,0,1,0,1,1,0\}(-42SN) \quad (\text{bits size} < 9)$$
$$bits' = \{0,1,1,0,1,0,1,1\}(107SN) = bits \ggg_8 7$$

我们定义**赋值运算**.

$a := b$ 只有满足条件一情况下, 意思是 b 的数据或者值复制給 a

C Used PRNG Detail Component Implementation

Structured Pseudocode 1: Linear Feedback Shift Register (python)

Input: **Seed** $\in \mathbb{F}_2^{64}$ (The $\mathbb{F}_2^{64}$ is a collection of integers ranging from 0 to 18446744073709551616 - 1)

Output: The updated **StateArray** and **PseudoRandomNumber** $\in \mathbb{F}_2^{64}$

```
1   import numpy as np
2
3   class LinearFeedbackShiftRegister:
4
5       """
6       Array position 0 is is the current random number seed
```

```
        Array position 1 the current random number
        """
        state = [np.uint64(0),np.uint64(0)]

        def __init__(self, seed: np.uint64):
            self.seed(seed)

        def seed(self, seed) -> None:
            self.state[0] = 0
            self.state[1] = seed;
            self.generate_bits(63)
            self.generate_bits(63)

        def generate_bits(self, bits_size: np.uint64) -> np.uint64:
            NumberA = np.uint64(self.state[0])
            NumberB = np.uint64(self.state[1])

            # The initial value of the polynomial can be: 128, 126, 101, 99
            answer = np.uint64(128)

            '''
            ? : polynomial power coefficient
            64(bits need shift amount, in 64-bit data) + 64 == 128(>= 64) == 128 - ?
            ? = 0
            63(bits need shift amount, in 64-bit data) + 64 == 127(>= 64) == 128 - ?
            ? = 1
            25(bits need shift amount, in 64-bit data) + 64 == 89(>= 64) == 128 - ?
            ? = 39
            23(bits need shift amount, in 64-bit data) + 64 == 87(>= 64) == 128 - ?
            ? = 41
            0(bits need shift amount, in 64-bit data) (< 64) = 128 - ?
            ? = 128
            '''

            for round_counter in range(bits_size):
                # Compute pseudo-random bit sequences in binary
                # This polynomial is : x^128 ⊕128 x^41 ⊕128 x^39 ⊕128 x ⊕128 1
                # As an example, the highest coefficient of this polynomial is 128.
                irreducible_primitive_polynomial = NumberB ^ (NumberA >> np.uint64(23)) ^ (NumberA >> np.uint64(25)) ^ (NumberA
                ↪  >> np.uint64(63))

                # Only one binary random bit is retained
                current_random_bit = irreducible_primitive_polynomial & np.uint64(0x01)

                # Discard the highest bit of the answer random number, the lowest bit is complemented by '0'
                answer <<= 1

                # The answer random number BIT_OR 0ULL || 1ULL
                answer |= current_random_bit

                # Discard the lowest bit of the random number seed, the highest bit is complemented by '0'
                NumberB >>= np.uint64(1)

                # Random number seed of the current state BIT_OR 0ULL || 0xFFFF'FFFF'FFFF'FFFFULL
                NumberB |= (NumberA & np.uint64(0x01)) << np.uint64(63)

                # Discard the lowest bit of the random number, the highest bit is complemented by '0'
                NumberA >>= np.uint64(1)

                # Random number of the current state BIT_OR 0ULL || 0xFFFF'FFFF'FFFF'FFFFULL
                NumberA |= current_random_bit << np.uint64(63)
```

```
67
68              self.state[0] = NumberA
69              self.state[1] = NumberB
70
71              return answer
72
73          def discard(self, round_number: np.uint64)-> None:
74              for i in range(0, round_number)
75                  self.generate_bits(63)
76
77          def __call__(self):
78              return self.generate_bits(63)
79
80          def __del__(self):
81              check_pointer = ctypes.c_void_p()
82              ctypes.memset(ctypes.byref(state), 0, ctypes.sizeof(state))
83              check_pointer = None
```

Structured Pseudocode 2: Twilight-Dream Nonlinear Feedback Shift Register (python)

Input: **Seed** $\in \mathbb{F}_2^{64}$

Output: The **StateArray** Updated and **PseudoRandomNumber** $\in \mathbb{F}_2^{64}$

```
1       import numpy as np
2
3       """
4       A random number generator using non-linear feedback shift register algorithm
5       """
6       class NonlinearFeedbackShiftRegister:
7
8           """
9           Array position 0 is is the current random number seed
10          Array position 1,2,3 the current random number
11          """
12          state = [np.uint64(0),np.uint64(0),np.uint64(0),np.uint64(0)]
13
14          def __init__(self, seed: np.uint64):
15              self.seed(seed)
16
17          def seed(self, seed) -> None:
18              if seed == 0:
19                  seed += 1
20
21              # Initial state
22              self.state[0] = seed
23              self.state[1] = (seed * 2) + 1
24              self.state[2] = (seed * 3) + 2
25              self.state[3] = (seed * 4) + 3
26
27              # Mix state (stage 1/2)
28              self.state[0] += (self.state[1] ^ self.state[2]) ^ ~(self.state[3])
29              self.state[1] -= (self.state[2] & self.state[3]) | self.state[0]
30              self.state[2] += (self.state[3] ^ self.state[0]) ^ ~(self.state[1])
31              self.state[3] -= (self.state[0] | self.state[1]) & self.state[2]
32
33              # Mix state (stage 2/2)
34              self.state[3] *= (seed << 48) & 0xffffffff
35              self.state[2] *= (seed << 32) & 0xffffffff
36              self.state[1] *= (seed << 16) & 0xffffffff
37              self.state[0] *= (seed) & 0xffffffff
38
```

```
39              # Update state
40              for initial_round in range(128, 0, -1):
41                  self.state[2] ^= self.random_bits(self.state[0], ((self.state[0] >> 6) ^ self.state[1] ^ self.state[3] ^ seed)
                    ↪  % 9, self.state[1] & 0x01)
42                  self.state[3] ^= self.random_bits(self.state[1], ((self.state[1] << 57) ^ self.state[0] ^ self.state[2] ^ seed)
                    ↪  % 9, self.state[0] & 0x01)
43                  self.state[0] ^= self.random_bits(self.state[2], ((self.state[2] >> 24) ^ self.state[3] ^ self.state[1] ^ seed)
                    ↪  % 9, self.state[3] & 0x01)
44                  self.state[1] ^= self.random_bits(self.state[3], ((self.state[3] << 37) ^ self.state[2] ^ self.state[0] ^ seed)
                    ↪  % 9, self.state[2] & 0x01)
45
46                  # Current random bit
47                  bit = (self.state[0] & 0x01) ^ (self.state[1] & 0x01) ^ (self.state[2] & 0x01) ^ (self.state[3] & 0x01)
48
49                  # Perform the nonlinear feedback function
50                  temporary_state = (self.state[0] ^ self.state[1]) & self.state[2] | self.state[3]
51
52                  # Override seed number values
53                  seed = (seed >> 49 | seed << 15) * (self.state[0] << 13 | self.state[0] >> 51)
54
55                  # Shift the values in the state array
56                  self.state[0], self.state[1], self.state[2], self.state[3] = self.state[1], self.state[2], self.state[3],
                    ↪  temporary_state
57
58                  # In the (MSB/LSB) position, set a random bit
59                  seed |= (bit << 63) if (temporary_state & 0x01) else (bit & 0x01)
60
61          """
62          Apply complex properties of irreducible primitive polynomials to generate
63          nonlinear random bit streams of numbers.
64
65          Parameters:
66          state_number (int): A 64-bit unsigned integer representing the current state value.
67          irreducible_polynomial_count (int): An integer representing the degree of the primitive polynomial.
68          bit (int): A value of either 0 or 1 representing the bit to XOR with the output.
69
70          Returns:
71          A tuple containing the updated state number and the output bit.
72          """
73          def random_bits(state_number, irreducible_polynomial_count, bit) -> np.uint64:
74              # Binary polynomial data source: https://users.ece.cmu.edu/~koopman/lfsr/index.html
75              # x is 2, for example: x ^ 3 = 2 * 2 * 2;
76
77              switcher = {
78              # Primitive polynomial degree is 24
79              # x^23 + x^10 + x^9 + x^8 + x^6 + x^4 + x^3 + 1
80              0: 0x80_0759
81
82              # Primitive polynomial degree is 55
83              # x^54 - x^10 - x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 - x^2
84              1: 0x40_0000_0000_07FC,
85
86              # Primitive polynomial degree is 48
87              # x^47 + x^11 + x^10 + x^8 + x^5 + x^4 + x^3 + 1
88              2: 0x8000_0000_0D39,
89
90              # Primitive polynomial degree is 31
91              # x^30 - x^9 - x^8 - x^7 - x^5 - x^4 - x^3 - x^2 - x - 1
92              3: 0x4000_03BF,
93
94              # Primitive polynomial degree is 64
```

```
            # x^63 + x^12 + x^9 + x^8 + x^5 + x^2
            4: 0x8000_0000_0000_1324,

            # Primitive polynomial degree is 27
            # x^26 - x^10 - x^3 - x^2 - x - 1
            5: 0x400_040F,

            # Primitive polynomial degree is 7
            # x^6 + 1
            6: 0x41,

            # Primitive polynomial degree is 16
            # x^15 - x^10 - x^7 - x^5 - x^4 - x^3 - x^2 - x
            7: 0x84BE,

            #Primitive polynomial degree is 42
            #x^41 + x^11 + x^10 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x
            8: 0x200_0000_0D7E
            }

            primitive_polynomial = switcher.get(irreducible_polynomial_count, 0)
            lowest_bit = state_number & 0x01
            state_number >>= 1
            state_number ^= ((~(lowest_bit) + 1) & primitive_polynomial)

            return state_number ^ bit

        """
        Reference URL:
        http://www.numberworld.org/constants.html
        https://www.exploringbinary.com/pi-and-e-in-binary/
        https://oeis.org/A001113
        https://oeis.org/A001622
        https://oeis.org/A000796

        Combination of the values of the Fibonacci sequence
        123581321345589144 == 0x1B70C8E97AD5F98

        PI Approximately equal to 3.14159265358979323846264338327950288419716939937510582097494459230781

        Circumference is a mathematical constant that is the ratio of the circumference of a circle to its diameter
        Binary format: 11.001001000011111101101010100010001000010110100011000010001101001 1

        The binary numbers are stripped of the floating point portion and converted to hexadecimal, i.e: 0x243F6A8885A308D3

        e Approximately equal to 2.71828182845904523536028747135266249775724709369995957496696762772407

        The Euler number is the base of the natural logarithm, not to be confused with the Euler-Mascheroni constant
        Binary format: 10.101101111110000101010001011000101000101011101101001010100110101 0

        The binary numbers are stripped of the floating point portion and converted to hexadecimal, i.e: 0xB7E151628AED2A6A

        phi Approximately equal to 1.61803398874989484820458683436563861803398874989484820458683436563 8

        In mathematics, two quantities are in the golden ratio if their ratio is the same as the ratio of their sum to the larger of the two quantities. Expressed algebraically, for quantities.
        Expressed algebraically, for quantities a and b with a>b>0
        where the Greek letter phi denotes the golden ratio.
        The constant phi satisfies the quadratic equation phi^2 = phi + 1, and is an irrational number with a value of phi = (1 + sqrt(5)) / 2
        Binary format: 01.10011110001101110111100110111001011111110100101001111100000101 01
```

```
154
155         The binary numbers are stripped of the floating point portion and converted to hexadecimal, i.e: 0x9E3779B97F4A7C15
156         """
157         def generate_chaotic_number(self, algorithm_execute_count: np.uint64) -> np.uint64:
158             """
159             Hamming weights (number of bits with 1)
160             bin(value).count("1")
161             """
162             FibonacciSequence = 0x1B70C8E97AD5F98
163             CircumferenceSequence = 0x243F6A8885A308D3
164             GoldenRatioSequence = 0x9E3779B97F4A7C15
165             EulerNumberSequence = 0xB7E151628AED2A6A
166
167             FibonacciSequenceBytes = unpack_8byte(FibonacciSequence)
168             CircumferenceSequenceBytes = unpack_8byte(CircumferenceSequence)
169             GoldenRatioSequenceBytes = unpack_8byte(GoldenRatioSequence)
170             EulerNumberSequenceBytes = unpack_8byte(EulerNumberSequence)
171             Number2Power64Modulus = np.uint64(2**64 - 1)
172
173             if algorithm_execute_count < 8:
174                 algorithm_execute_count = 8
175
176             answer = 0
177
178             for round_counter in range(algorithm_execute_count):
179                 bit = (self.state[0] ^ self.state[1] ^ self.state[2] ^ self.state[3]) & 0x01
180
181                 answer <<= 1
182                 answer |= bit
183
184                 if (bin(answer).count('1') & 0x01) != 0:
185                     answer ^= CircumferenceSequence
186                 else:
187                     multiplied_number_byte_span = memory_data_format_exchanger.Unpacker_8Byte(answer)
188
189                 SequenceBytes = FibonacciSequenceBytes if (answer ^ self.state[1]) & 0x01 else GoldenRatioSequenceBytes
190
191                 for index in range(sizeof(np.uint64)):
192                     multiplied_number_byte_span[index] = GF256_Instance.multiplication(multiplied_number_byte_span[index],
                        ↪  SequenceBytes[index])
193
194                 answer ^= memory_data_format_exchanger.Packer_8Byte(multiplied_number_byte_span)
195
196                 if (bin(self.state[2]).count('1') & 0x01) == 0:
197                     multiplied_number_byte_span = memory_data_format_exchanger.Unpacker_8Byte(self.state[2])
198
199                 SequenceBytes = EulerNumberSequenceBytes if (answer ^ self.state[3]) & 0x01 else CircumferenceSequenceBytes
200
201                 for index in range(sizeof(np.uint64)):
202                     multiplied_number_byte_span[index] = GF256_Instance.multiplication(multiplied_number_byte_span[index],
                        ↪  SequenceBytes[index])
203
204                 self.state[2] ^= memory_data_format_exchanger.Packer_8Byte(multiplied_number_byte_span)
205
206                 if (self.state[2] & 0x01) == 0:
207                     self.state[2] ^= FibonacciSequence
208                 else:
209                     self.state[2] ^= GoldenRatioSequence ^ answer
210
211                 if (self.state[2] & 0x01) != 0:
212                     self.state[2] ^= CircumferenceSequence
```

```
                if round_counter % 2 == 0:
                    value_0, value_1, value_2, value_3 = state

                    # Binary hash processing that can cause an avalanche effect
                    # When this function is called frequently, it consumes a lot of CPU computing power
                    random_number = int(((answer >> 17) ^ value_1) ^ value_2)

                    value_0 &= value_3
                    if value_0 == 0:
                        value_0 += (value_2 * 2)

                    answer ^= self.random_bits(value_0, random_number % 9, np.uint64((value_3 & 0x01) ^ bit))

                    value_3 &= value_0
                    if value_3 == 0:
                        value_3 -= value_1 * 2
                else:
                    value_0, value_1, value_2, value_3 = state

                    # Bit Data Mixing Function
                    value_1 ^= ((answer ^ value_0) >> (value_3 - value_2)) & Number2Power64Modulus
                    value_2 ^= (value_1 << ((value_0 + value_3) & Number2Power64Modulus)) & Number2Power64Modulus
                    value_3 ^= (value_2 >> ((value_1 + value_0) & Number2Power64Modulus)) & Number2Power64Modulus
                    value_0 ^= ((answer ^ value_3) << (value_1 - value_2)) & Number2Power64Modulus

                    # Pseudo-Hadamard Transform
                    value_a = bit if value_0 + value_1 == 0 else value_0 + value_1
                    value_b = bit if value_0 + value_1 * 2 == 0 else value_0 + value_1 * 2
                    value_c = bit if value_3 - value_2 == 0 else value_3 - value_2
                    value_d = bit if value_2 * 2 - value_3 == 0 else value_2 * 2 - value_3

                    # Forward form
                    value_0 ^= value_a
                    value_1 ^= value_b

                    # Backward form
                    value_2 ^= value_c
                    value_3 ^= value_d

                    value_a = value_b = value_c = value_d = 0

                bit = 0

            """
            Important Notes:
            The two step constants here, 17 and 42, can swap positions; bitwise left shifts (<<) and bitwise right shifts (>>),
    ↪  can also swap positions.
            Note that this bitwise exclusive-or operation cannot be removed, and the operand must be a variable ANSWER!
            Although the two step constants can be any number of step  [0 , 63], they must be unequal and need to be 1 odd and 1
    ↪  even!
            """
            return (answer ^ ( (answer << 17) | (answer >> 42) ));

        def unpredictable_bits(self, base_number: np.uint64, number_bits: np.uint64) -> np.uint64:

            """
            Generate unpredictable bit sequences.

            Using the same numeric seed, construct an object of a nonlinear feedback shift register and call this function.
```

```
271             Depending on whether the (base_number) argument is odd or even, it determines one of the two different bit sequences
    ↪ that will be generated.
272
273             However, there is an exception to this rule
274             If the (number_bit) parameter is greater than or equal to 64
275             the linear feedback shift register (result value - answer) is broken because the number of bits shifted right or
    ↪ left is greater than 64
276             Then the sequence will be chaotic in a way that even the linear feedback shift register is not known.
277             Even though all the parameters provided and the internal state are the same, you can restore these sequences
278
279             When the sequence is in a chaotic state, it may be in between linear and non-linear states, so please record all the
    ↪ provided parameters and numerical seeds yourself.
280
281             Args:
282                 base_number: An integer to determine which bit sequence will be generated.
283                 number_bits: The number of bits to generate.
284
285             Returns:
286                 An integer representing the generated unpredictable bit sequence.
287
288             """
289
290             answer = base_number
291             current_random_bit = 0
292
293             current_random_bits = [0, 0, 0, 0]
294
295             for round_counter in range(number_bits):
296                 current_random_bit = ((self.state[0] ^ self.state[1] ^ self.state[2] ^ self.state[3]) >> 63) & 0x01
297
298                 # Discard the highest bit of the answer random number, the lowest bit is complemented by '0'
299                 answer <<= 1
300
301                 # The answer random number BIT_OR 0 or 1
302                 answer ^= current_random_bit
303
304                 # Compute pseudo-random bit sequences in binary
305
306                 # I have combined different degrees of linear feedback shift registers here
307                 # They form a nonlinear feedback shift register, and the numbers generated by mixing these states are not
                    ↪  predictable
308                 self.state[0] = self.random_bits(self.state[0], (self.state[3] ^ self.state[2]) % 9, current_random_bit)
309
310                 # Only one binary random bit is retained
311                 current_random_bits[0] ^= self.state[0] & 0x01
312
313                 self.state[1] = self.random_bits(self.state[1], (self.state[2] ^ self.state[1]) % 9, current_random_bit)
314
315                 # Only one binary random bit is retained
316                 current_random_bits[1] ^= self.state[1] & 0x01
317
318                 self.state[2] = self.random_bits(self.state[2], (self.state[1] ^ self.state[0]) % 9, current_random_bit)
319
320                 # Only one binary random bit is retained
321                 current_random_bits[2] ^= self.state[2] & 0x01
322
323                 self.state[3] = self.random_bits(self.state[3], (self.state[0] ^ self.state[3]) % 9, current_random_bit)
324
325                 # Only one binary random bit is retained
326                 current_random_bits[3] ^= self.state[3] & 0x01
327
```

```
                current_random_bit = (current_random_bits[0] | current_random_bits[1])
                ^ (current_random_bits[1] & current_random_bits[2])
                ^ (current_random_bits[2] | current_random_bits[3])
                ^ (current_random_bits[3] & current_random_bits[0])

                # Discard the highest bit of the answer random number, the lowest bit is complemented by '0'
                answer <<= 1

                answer |= current_random_bit

                swap(self.state[0 + self.state[0] % len(current_random_bits)], current_random_bits[3])
                swap(self.state[0 + self.state[1] % len(current_random_bits)], current_random_bits[3])
                swap(self.state[0 + self.state[2] % len(current_random_bits)], current_random_bits[3])
                swap(self.state[0 + self.state[3] % len(current_random_bits)], current_random_bits[3])

                # Get the lowest bit of the bit sequence according to the current state (random number seed or random number);
                # and set that bit to the highest bit of the next state (random number seed or random number)

                self.state[1] >>= 1
                self.state[1] |= (self.state[0] & 0x01) << 63

                self.state[2] >>= 1
                self.state[2] |= (self.state[1] & 0x01) << 63

                self.state[3] >>= 1
                self.state[3] |= (self.state[2] & 0x01) << 63

                self.state[0] >>= 1
                self.state[0] |= (self.state[3] & 0x01) << 63

            check_pointer = ctypes.c_void_p()
            ctypes.memset(ctypes.byref(current_random_bits), 0, ctypes.sizeof(current_random_bits))
            check_pointer = None

            return answer

        def discard(self, round_number: np.uint64)-> None:
            if round_number == 0
                round_number = 1;

        self.generate_chaotic_number(round_number * 2)

        def __call__(self) -> np.uint64:
            return self.generate_chaotic_number(8)

        def __del__(self):
            check_pointer = ctypes.c_void_p()
            ctypes.memset(ctypes.byref(state), 0, ctypes.sizeof(state))
            check_pointer = None
```

There are note function **RandomBits** form **Structured Pseudocode 2**

These fixed constants are the result of calculating polynomials composed of binary data.

$x^{23} \oplus_{64} x^{10} \oplus_{64} x^{9} \oplus_{64} x^{8} \oplus_{64} x^{6} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} 1$ (Primitive polynomial degree is 24)

$x^{54} \oplus_{64} x^{10} \oplus_{64} x^{9} \oplus_{64} x^{8} \oplus_{64} x^{7} \oplus_{64} x^{6} \oplus_{64} x^{5} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} x^{2}$ (Primitive polynomial degree is 55)

$x^{47} \oplus_{64} x^{11} \oplus_{64} x^{10} \oplus_{64} x^{8} \oplus_{64} x^{5} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} 1$ (Primitive polynomial degree is 48)

$x^{30} \oplus_{64} x^{9} \oplus_{64} x^{8} \oplus_{64} x^{7} \oplus_{64} x^{5} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} x^{2} \oplus_{64} x \oplus_{64} 1$ (Primitive polynomial degree is 30)

$x^{63} \oplus_{64} x^{12} \oplus_{64} x^{9} \oplus_{64} x^{8} \oplus_{64} x^{5} \oplus_{64} x^{2}$ (Primitive polynomial degree is 63)

$x^{26} \oplus_{64} x^{10} \oplus_{64} x^{3} \oplus_{64} x^{2} \oplus_{64} x \oplus_{64} 1$ (Primitive polynomial degree is 27)

$x^{6} \oplus_{64} 1$ (Primitive polynomial degree is 6)

$x^{15} \oplus_{64} x^{10} \oplus_{64} x^{7} \oplus_{64} x^{5} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} x^{2} \oplus_{64} x$ (Primitive polynomial degree is 16)

$x^{41} \oplus_{64} x^{11} \oplus_{64} x^{10} \oplus_{64} x^{8} \oplus_{64} x^{6} \oplus_{64} x^{5} \oplus_{64} x^{4} \oplus_{64} x^{3} \oplus_{64} x^{2} \oplus_{64} x$ (Primitive polynomial degree is 42)

Structured Pseudocode 3: CSPRNG based on chaos theory, using simulated double pendulum motion. (python)

```python
import numpy as np
"""
Simulate a two-segment pendulum physical system to generate pseudo-random numbers based on a binary key
https://zh.wikipedia.org/wiki/%E5%8F%8C%E6%91%86
https://en.wikipedia.org/wiki/Double_pendulum
https://www.researchgate.net/publication/345243089_A_Pseudo-Random_Number_Generator_Using_Double_Pendulum

Please refer to the citation <A pseudo-random number generator using double pendulum> for the contents

Or refer to the implementation of the c++ programming language
https://github.com/robinsandhu/DoublePendulumPRNG/blob/master/prng.cpp
https://github.com/Twilight-Dream-Of-Magic/TDOM-EncryptOrDecryptFile-Reborn
/blob/ExperimentalFeatureTesting/include/CommonSecurity/SecureRandomUtilLibrary.hpp#L3804
"""

class SimulateDoublePendulum:
    gravity_coefficient = 9.8
    hight = 0.002

    BackupTensions = [0.0, 0.0]
    BackupVelocitys = [0.0, 0.0]

    def __init__(self, number):
        self.BackupTensions = np.zeros(2)
        self.BackupVelocitys = np.zeros(2)
        self.SystemData = np.zeros(10)
        self.seed(number)

    def run_system(self, is_initialize_mode, time):
        gravity_coefficient = 9.81
        length1 = self.SystemData[0]
        length2 = self.SystemData[1]
        mass1 = self.SystemData[2]
        mass2 = self.SystemData[3]
        tension1 = self.SystemData[4]
        tension2 = self.SystemData[5]

        velocity1 = self.SystemData[8]
        velocity2 = self.SystemData[9]

        for counter in range(time):
            denominator = 2 * mass1 + mass2 - mass2 * math.cos(2 * tension1 - 2 * tension2)

            alpha1 = -1 * gravity_coefficient * (2 * mass1 + mass2) * math.sin(tension1) \
                    - mass2 * gravity_coefficient * math.sin(tension1 - 2 * tension2) \
                    - 2 * math.sin(tension1 - tension2) * mass2 \
                    * (velocity2 * velocity2 * length2 + velocity1 * velocity1 * length1 * math.cos(tension1 - tension2))

            alpha1 /= length1 * denominator

            alpha2 = 2 * math.sin(tension1 - tension2) \
                    * (velocity1 * velocity1 * length1 * (mass1 + mass2) + gravity_coefficient * (mass1 + mass2) *
                    ↪  math.cos(tension1) \
                    + velocity2 * velocity2 * length2 * mass2 * math.cos(tension1 - tension2))

            alpha2 /= length2 * denominator

            velocity1 += self.hight * alpha1
```

```
            velocity2 += self.hight * alpha2
            tension1 += self.hight * velocity1
            tension2 += self.hight * velocity2

        if is_initialize_mode:
            self.BackupTensions[0] = tension1
            self.BackupTensions[1] = tension2

            self.BackupVelocitys[0] = velocity1
            self.BackupVelocitys[1] = velocity2

    def initialize(self, binary_key_sequence):
        if not binary_key_sequence:
            raise ValueError("RNG_ChaoticTheory::SimulateDoublePendulum: This binary key sequence must be not empty!")

        binary_key_sequence_size = len(binary_key_sequence)
        binary_key_sequence_2d = [[] for _ in range(4)]
        for index in range(binary_key_sequence_size // 4):
            binary_key_sequence_2d[0].append(binary_key_sequence[index])
            binary_key_sequence_2d[1].append(binary_key_sequence[binary_key_sequence_size // 4 + index])
            binary_key_sequence_2d[2].append(binary_key_sequence[binary_key_sequence_size // 2 + index])
            binary_key_sequence_2d[3].append(binary_key_sequence[binary_key_sequence_size * 3 // 4 + index])

        binary_key_sequence_2d_param = [[] for _ in range(7)]
        key_outer_round_count = 0
        key_inner_round_count = 0
        while key_outer_round_count < 64:
            while key_inner_round_count < binary_key_sequence_size // 4:
                binary_key_sequence_2d_param[0].append(binary_key_sequence_2d[0][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[1][key_inner_round_count])
                binary_key_sequence_2d_param[1].append(binary_key_sequence_2d[0][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[2][key_inner_round_count])
                binary_key_sequence_2d_param[2].append(binary_key_sequence_2d[0][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[3][key_inner_round_count])
                binary_key_sequence_2d_param[3].append(binary_key_sequence_2d[1][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[2][key_inner_round_count])
                binary_key_sequence_2d_param[4].append(binary_key_sequence_2d[1][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[3][key_inner_round_count])
                binary_key_sequence_2d_param[5].append(binary_key_sequence_2d[2][key_inner_round_count] ^
                ↪  binary_key_sequence_2d[3][key_inner_round_count])
                binary_key_sequence_2d_param[6].append(binary_key_sequence_2d[0][key_inner_round_count])

                key_inner_round_count += 1
                key_outer_round_count += 1
                if key_outer_round_count >= 64:
                    break
            key_inner_round_count = 0
        key_outer_round_count = 0

        radius = self.SystemData[6]
        current_binary_key_sequence_size = self.SystemData[7]

        for i in range(64):
            for j in range(6):
                if binary_key_sequence_2d_param[j][i] == 1:
                    self.SystemData[j] += 1 * pow(2.0, 0 - i)
            if binary_key_sequence_2d_param[6][i] == 1:
                radius += 1 * pow(2.0, 4 - i)

        current_binary_key_sequence_size = float(binary_key_sequence_size)

```

```
            # This is initialize mode
            self.run_system(True, round(radius * current_binary_key_sequence_size))

        def seed_with_binary_string(self, binary_key_sequence_string: str):
            binary_key_sequence = []
            binary_zero_string = '0'
            binary_one_string = '1'
            for data in binary_key_sequence_string:
                if data != binary_zero_string and data != binary_one_string:
                    continue

                binary_key_sequence.append(0 if data == binary_zero_string else 1)

            if not binary_key_sequence:
                return
            else:
                self.initialize(binary_key_sequence)

        def seed(self, seed_value):
            if isinstance(seed_value, int):
                if seed_value < 0:
                    binary_string = format(seed_value & (2**32-1), '032b')
                else:
                    binary_string = format(seed_value, '032b')
                self.seed_with_binary_string(binary_string)
            elif isinstance(seed_value, str):
                self.seed_with_binary_string(seed_value)
            else:
                raise ValueError("Seed value must be an integer or string.")

        # Interleaved concatenate one-by-one bits
        def concat(a: np.int32, b: np.int32) -> np.int64:
            result_binary_string = ""
            for i in range(32):
                result_binary_string += "1" if (b % 2) == 1 else "0"
                b //= 2
                result_binary_string += "1" if (a % 2) == 1 else "0"
                a //= 2
            concate_bitset = np.int64(result_binary_string[::-1], 2)
            c = concate_bitset
            return c

        def generate(self) -> np.int64:
            # This is generate mode
            self.run_system(False, 1)

            temporary_floating_a = 0.0
            temporary_floating_b = 0.0

            left_number = 0
            right_number = 0

            temporary_floating_a = self.SystemData[0] * sin(self.SystemData[4]) + self.SystemData[1] * sin(self.SystemData[5])
            temporary_floating_b = -(self.SystemData[0]) * sin(self.SystemData[4]) - self.SystemData[1] *
            ↪  sin(self.SystemData[5])

            left_number = floor(math.fmod(temporary_floating_a * 1000, 1.0) * 4294967296)
            right_number = floor(math.fmod(temporary_floating_b * 1000, 1.0) * 4294967296)

            return self.concat(int(left_number), int(right_number))

```

```
    def __call__(self, generated_count: int, min_number: int, max_number: int) -> List[np.uint64]:
        modulus = np.int64(max_number) - np.int64(min_number) + 1

        random_numbers = [0] * generated_count
        for i in range(generated_count):
            temporary_random_number = self.generate()

            if modulus != 0:
                temporary_random_number %= modulus

            if temporary_random_number < 0:
                temporary_random_number += modulus

            random_numbers[i] = np.uint64(np.int64(min_number) + temporary_random_number)

        return random_numbers

    def __call__(self, min_number: np.uint64, max_number: np.uint64) -> np.uint64:
        modulus = np.int64(max_number) - np.int64(min_number) + 1

        random_number = 0
        temporary_random_number = self.generate()

        if modulus != 0:
            temporary_random_number %= modulus

        if temporary_random_number < 0:
            temporary_random_number += modulus

        random_numbers = np.uint64(np.int64(min_number) + temporary_random_number)

        return random_number

    def __del__(self):
        self.BackupVelocitys.fill(0.0)
        self.BackupTensions.fill(0.0)
        self.SystemData.fill(0.0)
```

D Specific implementation of some of the algorithms of this project in programming language

关于更多本项目的算法的 c++ 语言的具体实现, 详细请见文件: For more details on the implementation of the algorithms in c++ for this project, pleas

Modules_OaldresPuzzle_Cryptic.hpp
OaldresPuzzle_Cryptic.cpp
OPC_MainAlgorithm_Worker.cpp

Code block 1: Computational classes belonging to the Galois finite field (2^8) byte data (p

```
import math

class GaloisFiniteField256:

    _LogarithmicTable =
    [
        0x00, 0x00, 0x01, 0x19, 0x02, 0x32, 0x1a, 0xc6, 0x03, 0xdf, 0x33, 0xee, 0x1b, 0x68, 0xc7, 0x4b,
        0x04, 0x64, 0xe0, 0x0e, 0x34, 0x8d, 0xef, 0x81, 0x1c, 0xc1, 0x69, 0xf8, 0xc8, 0x08, 0x4c, 0x71,
        0x05, 0x8a, 0x65, 0x2f, 0xe1, 0x24, 0x0f, 0x21, 0x35, 0x93, 0x8e, 0xda, 0xf0, 0x12, 0x82, 0x45,
        0x1d, 0xb5, 0xc2, 0x7d, 0x6a, 0x27, 0xf9, 0xb9, 0xc9, 0x9a, 0x09, 0x78, 0x4d, 0xe4, 0x72, 0xa6,
        0x06, 0xbf, 0x8b, 0x62, 0x66, 0xdd, 0x30, 0xfd, 0xe2, 0x98, 0x25, 0xb3, 0x10, 0x91, 0x22, 0x88,
```

```
            0x36, 0xd0, 0x94, 0xce, 0x8f, 0x96, 0xdb, 0xbd, 0xf1, 0xd2, 0x13, 0x5c, 0x83, 0x38, 0x46, 0x40,
            0x1e, 0x42, 0xb6, 0xa3, 0xc3, 0x48, 0x7e, 0x6e, 0x6b, 0x3a, 0x28, 0x54, 0xfa, 0x85, 0xba, 0x3d,
            0xca, 0x5e, 0x9b, 0x9f, 0x0a, 0x15, 0x79, 0x2b, 0x4e, 0xd4, 0xe5, 0xac, 0x73, 0xf3, 0xa7, 0x57,
            0x07, 0x70, 0xc0, 0xf7, 0x8c, 0x80, 0x63, 0x0d, 0x67, 0x4a, 0xde, 0xed, 0x31, 0xc5, 0xfe, 0x18,
            0xe3, 0xa5, 0x99, 0x77, 0x26, 0xb8, 0xb4, 0x7c, 0x11, 0x44, 0x92, 0xd9, 0x23, 0x20, 0x89, 0x2e,
            0x37, 0x3f, 0xd1, 0x5b, 0x95, 0xbc, 0xcf, 0xcd, 0x90, 0x87, 0x97, 0xb2, 0xdc, 0xfc, 0xbe, 0x61,
            0xf2, 0x56, 0xd3, 0xab, 0x14, 0x2a, 0x5d, 0x9e, 0x84, 0x3c, 0x39, 0x53, 0x47, 0x6d, 0x41, 0xa2,
            0x1f, 0x2d, 0x43, 0xd8, 0xb7, 0x7b, 0xa4, 0x76, 0xc4, 0x17, 0x49, 0xec, 0x7f, 0x0c, 0x6f, 0xf6,
            0x6c, 0xa1, 0x3b, 0x52, 0x29, 0x9d, 0x55, 0xaa, 0xfb, 0x60, 0x86, 0xb1, 0xbb, 0xcc, 0x3e, 0x5a,
            0xcb, 0x59, 0x5f, 0xb0, 0x9c, 0xa9, 0xa0, 0x51, 0x0b, 0xf5, 0x16, 0xeb, 0x7a, 0x75, 0x2c, 0xd7,
            0x4f, 0xae, 0xd5, 0xe9, 0xe6, 0xe7, 0xad, 0xe8, 0x74, 0xd6, 0xf4, 0xea, 0xa8, 0x50, 0x58, 0xaf
        ]

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

        def addition_or_subtraction(self, left: np.uint8, right: np.uint8):
            return left ^ right

        def multiplication(self, left, right):
            if left == 0x00 or right == 0x00:
                return 0x00

            integer_a = left
            integer_b = right

            integer_a = GaloisFiniteField256._LogarithmicTable[integer_a]
            integer_b = GaloisFiniteField256._LogarithmicTable[integer_b]

            value = np.uint32(integer_a + integer_b) % np.uint32(255)

            return GaloisFiniteField256._ExponentialTable[value]

        def division(self, left: np.uint8, right: np.uint8):
            if left == 0x00:
                return 0x00

            if right == 0x00:
                assert False, "GaloisFiniteField256: divide by zero"

            integer_a = left
            integer_b = right

            integer_a = GaloisFiniteField256._LogarithmicTable[integer_a]
```

```
            integer_b = GaloisFiniteField256._LogarithmicTable[integer_b]

            value = np.uint32(integer_a - integer_b) % np.uint32(255)
            if value < 0:
                value += 255

            return GaloisFiniteField256._ExponentialTable[value]

        get_instance_instance = GaloisFiniteField256()

        @staticmethod
        def get_instance():
            return get_instance_instance

```

Code block 2: SegmentTree Class And Caller This Class Funtion (python)

```
    class SegmentTree:
        def __init__(self, array_size):
            assert (array_size & (array_size - 1) == 0) and array_size > 0, \
                "array_size must be a power of 2"
            self.n = array_size
            self.nodes = [0] * (n << 1)

        def set(self, position):
            """Sets the value at position to 1."""
            current_node = N | position
            while current_node:
                self.nodes[current_node] += 1
                current_node >>= 1

        def get(self, order):
            """Returns the index of the element with the given order"""
            current_node = 1
            current_left_size = N >> 1
            left_total = 0

            while current_left_size:
                current_left_count = current_left_size - self.nodes[current_node << 1]

                if left_total + current_left_count > order:
                    current_node = current_node << 1
                else:
                    current_node = current_node << 1 | 1
                    left_total += current_left_count

                current_left_size >>= 1

            return current_node ^ N

        def clear(self):
            """Clears all elements in the tree."""
            self.nodes = [0] * (n << 1)

        def __del__(self):
            """Clears all elements in the tree upon deletion."""
            self.clear()

    #Note: This Member Funtion From MixTransformationUtil class
    def RegenerationRandomMaterialSubstitutionBox(old_data_box) -> None:
        """
        Regenerate a random material substitution box based on the old box.
```

```
46
47          Args:
48          - old_data_box: a list of 256 bytes representing the old substitution box
49
50          Returns:
51          - a list of 256 bytes representing the new substitution box
52          """
53
54          check_pointer = ctypes.c_void_p()
55
56          # initialize the NLFSR and the segment tree
57          nlfsr_object = CommonStateData.nlfsr
58          old_data_array_size = len(old_data_box)
59          segment_tree_object = SegmentTree(256)
60
61          new_data_box = [0] * 256
62          new_data_array_size = len(new_data_box)
63
64          index = 0
65          index2 = 0
66
67          while index < old_data_array_size and index2 < new_data_array_size:
68              if index == old_data_array_size - 1 and old_data_box[index] == segment_tree_object.get(0):
69                  # Need to re-operate data
70                  check_pointer = ctypes.memset(ctypes.byref(new_data_box), 0, ctypes.sizeof(new_data_box))
71                  check_pointer = None
72                  segment_tree_object.clear()
73                  index = 0
74                  index2 = 0
75                  continue
76
77              order = nlfsr_object() % (old_data_array_size - index)
78              position = segment_tree_object.get(order)
79
80              while old_data_box[index] == position:
81                  order = nlfsr_object() % (old_data_array_size - index)
82                  position = segment_tree_object.get(order)
83
84              new_data_box[index2] = position
85              segment_tree_object.set(position)
86              index += 1
87              index2 += 1
88
89          return new_data_box
```

Code block 2-1: SegmentTree Class And Caller This Class Funtion (c++)

```
1   template<std::integral DataType, std::size_t ArraySize>
2   class SegmentTree
3   {
4
5       /*
6           std::has_single_bit(ArraySize)
7           ArraySize != 0 && (ArraySize ^ (ArraySize & -ArraySize) == 0)
8       */
9
10  private:
11
12      static constexpr std::size_t N = std::has_single_bit(ArraySize) ? ArraySize : 0;
13      std::array<DataType, N << 1> Nodes {};
14
15  public:
```

```
        void Set(std::size_t Position)
        {
            for(std::size_t CurrentNode = N | Position; CurrentNode; CurrentNode >>= 1)
                this->Nodes[CurrentNode]++;
        }

        DataType Get(std::size_t Order)
        {
            std::size_t CurrentNode = 1;
            for(std::size_t CurrentLeftSize = N >> 1, LeftTotal = 0; CurrentLeftSize; CurrentLeftSize >>= 1)
            {
                std::size_t CurrentLeftCount = CurrentLeftSize - this->Nodes[CurrentNode << 1];
                if(LeftTotal + CurrentLeftCount > Order)
                    CurrentNode = CurrentNode << 1;
                else
                    CurrentNode = CurrentNode << 1 | 1, LeftTotal += CurrentLeftCount;
            }
            return static_cast<DataType>(CurrentNode ^ N);
        }

        void Clear()
        {
            volatile void* CheckPointer = nullptr;
            CheckPointer = memory_set_no_optimize_function<0x00>(this->Nodes.data(), this->Nodes.size() * sizeof(DataType));
            CheckPointer = nullptr;
        }

        ~SegmentTree()
        {
            volatile void* CheckPointer = nullptr;
            CheckPointer = memory_set_no_optimize_function<0x00>(this->Nodes.data(), this->Nodes.size() * sizeof(DataType));
            CheckPointer = nullptr;
        }
    };

    //Note: This Member Funtion From MixTransformationUtil class
    std::array<std::uint8_t, 256> RegenerationRandomMaterialSubstitutionBox(std::span<const std::uint8_t> OldDataBox)
    {
        volatile void* CheckPointer = nullptr;

        auto& NLFSR_Object = *(CommonStateDataPointerObject.AccessReference().NLFSR_ClassicPointer);

        const std::size_t OldDataArraySize = OldDataBox.size();
        SegmentTree<std::uint8_t, 256> SegmentTreeObject;

        std::array<std::uint8_t, 256> NewDataBox;
        const std::size_t NewDataArraySize = NewDataBox.size();

        for(std::size_t Index = 0, Index2 = 0; Index < OldDataArraySize && Index2 < NewDataArraySize; Index++, Index2++)
        {
            if(Index == OldDataArraySize - 1 && OldDataBox[Index] == SegmentTreeObject.Get(0))
            {
                //Need to re-operate data
                CheckPointer = memory_set_no_optimize_function<0x00>(NewDataBox.data(), NewDataBox.size());
                CheckPointer = nullptr;
                SegmentTreeObject.Clear();
                Index = 0;
                Index2 = 0;
                continue;
            }

```

```
            std::size_t Order = NLFSR_Object() % (OldDataArraySize - Index), Position = SegmentTreeObject.Get(Order);
            while (OldDataBox[Index] == Position)
                Order = NLFSR_Object() % (OldDataArraySize - Index), Position = SegmentTreeObject.Get(Order);
            NewDataBox[Index2] = Position, SegmentTreeObject.Set(Position);
        }

        return NewDataBox;
    }
```

Code block 3: Custom Secure Hash Class Based Sponge Function Structure (python)

```

    """
    https://en.wikipedia.org/wiki/Sponge_function

    基于海绵结构的密码学哈希函数，使用的伪随机置换函数由 Twilight-Dream 设计
    Cryptographic hash function based on sponge structure using a pseudo-random permutation function designed by Twilight-Dream
    """
    class CustomSecureHash:

        """
        Hash state bits size =  Bits rate size + Bits capacity size

        Example :
        The security of a sponge function depends on the length of its internal state and the length of the blocks.
        If message blocks are r-bit long and the internal state is w-bit long, then there are c = w − r bits of the internal state that can' t be modified by message blocks.
        The value of c is called a sponge' s capacity, and the security level guaranteed by the sponge function is c/2. For example, to reach 256-bit security with 64-bit message blocks, the internal state should be w = 2 × 256 + 64 = 576 bits.
        Of course, the security level also depends on the length, n, of the hash value. The complexity of a collision attack is therefore the smallest value between 2^{n/2} and 2^{c/2}, while the complexity of a second preimage attack is the smallest value between 2^n and 2^{c/2}.

        To be secure, the permutation P should behave like a random permutation, without statistical bias and without a mathematical structure that would allow an attacker to predict outputs.
        As in compression function-based hashes, sponge functions also pad messages, but the padding is simpler because it doesn' t need to include the message' s length.
        """

        BITS_STATE_SIZE = 2 * HashBitSize + 64
        BITS_RATE = HashBitSize
        BITS_CAPACITY = BITS_STATE_SIZE - BITS_RATE

        BYTES_RATE = BITS_RATE // 8
        BITWORDS_RATE = BYTES_RATE // 8
        BYTES_CAPACITY = BITS_CAPACITY // 8
        BITWORDS_CAPACITY = BYTES_CAPACITY // 8

        PAD_BYTE_DATA = 0b00000001
        PAD_BITSWORD_DATA = 0b0000000100000001000000010000000100000001000000010000000100000001

        BitsHashState = [0] * (BITS_STATE_SIZE // 64)

        StateBufferIndices = GenerateHashStateBufferIndices(BITS_STATE_SIZE)

        MoveBitCounts = [0] * 63
        HashStateIndices = [0] * (BITS_STATE_SIZE // 64)
        LeftRotatedStateBufferIndices = [0] * (BITS_STATE_SIZE // 128)
        RightRotatedStateBufferIndices = [0] * (BITS_STATE_SIZE // 128)

        StateCurrentCounter = 1

```

```
        def __init__(self, HashBitSize):
            self.HashBitSize = HashBitSize
            self.MoveBitCounts = self.GenerateRandomMoveBitCounts()
            self.HashStateIndices = self.GenerateRandomHashStateIndices()

            assert HashBitSize >= 128 and HashBitSize % 8 == 0

            self.LeftRotatedStateBufferIndices = [0] * STATE_BUFFER_SIZE
            self.RightRotatedStateBufferIndices = [0] * STATE_BUFFER_SIZE
            for i in range(STATE_BUFFER_SIZE):
                self.LeftRotatedStateBufferIndices[i] = StateBufferIndices[(i + 1) % STATE_BUFFER_SIZE]
                self.RightRotatedStateBufferIndices[i] = StateBufferIndices[(i - 1 + STATE_BUFFER_SIZE) % STATE_BUFFER_SIZE]

        def GenerateRandomMoveBitCounts():
            CSPRNG = CommonSecurity.RNG_ISAAC.isaac64(1946379852749613)
            CSPRNG.discard(1024)

            move_bit_counts = list(range(1, 64))

            for index in range(len(move_bit_counts)):
                new_index = (index + CSPRNG()) % len(move_bit_counts)
                move_bit_counts[index], move_bit_counts[new_index] = move_bit_counts[new_index], move_bit_counts[index]

            return move_bit_counts

        def GenerateRandomHashStateIndices():
            CSPRNG = CommonSecurity.RNG_ISAAC.isaac64(1946379852749613)
            CSPRNG.discard(2048)

            random_hash_state_indices = list(range(BITS_STATE_SIZE // 64))

            for index in range(len(random_hash_state_indices)):
                new_index = (index + CSPRNG()) % len(random_hash_state_indices)
                random_hash_state_indices[index], random_hash_state_indices[new_index] = random_hash_state_indices[new_index],
                ↪  random_hash_state_indices[index]

            return random_hash_state_indices

        """
        This corresponds to the mathematical abstraction of the F function in the structure of the sponge function
        (it is supposed to be a safe pseudo-random permutation function).
        It has the following steps:
        1. Hash state mixing
        2. Apply linear function
        3. Apply bit pseudo-random permutation (P function)
        4. Apply nonlinear functions
        5. Each round requires a mix of hash state and constants used by the hash
        """
        def TransformState(self, Counter):
            from CommonSecurity import Binary_LeftRotateMove, Binary_RightRotateMove
            BITS_STATE_SIZE = self.BITS_STATE_SIZE
            BitsHashState = self.BitsHashState
            BitsHashState_size = self.BitsHashState.size()
            HASH_ROUND_CONSTANTS = self.HASH_ROUND_CONSTANTS
            StateBufferIndices = self.StateBufferIndices
            LeftRotatedStateBufferIndices = self.LeftRotatedStateBufferIndices
            RightRotatedStateBufferIndices = self.RightRotatedStateBufferIndices
            HashStateIndices = self.HashStateIndices
            MoveBitCounts = self.MoveBitCounts
            StateBuffer = [0] * (BITS_STATE_SIZE // 2)
            StateBuffer2 = [0] * (BITS_STATE_SIZE // 2)
```

```
106             StateBuffer3 = [0] * BITS_STATE_SIZE
107 
108             for RoundIndex in range(BitsHashState_size - 1 - Counter, BitsHashState_size):
109                 # Step 1
110                 while self.StateCurrentCounter % BitsHashState_size != 0:
111                     StateBuffer[self.StateCurrentCounter % (BITS_STATE_SIZE // 2)] = BitsHashState[self.StateCurrentCounter %
                        ↪  BitsHashState_size] ^ BitsHashState[(self.StateCurrentCounter + 1) % BitsHashState_size]
112                     self.StateCurrentCounter += 1
113                     StateBuffer[self.StateCurrentCounter % (BITS_STATE_SIZE // 2)] = BitsHashState[(self.StateCurrentCounter +
                        ↪  2) % BitsHashState_size] ^ BitsHashState[(self.StateCurrentCounter + 3) % BitsHashState_size]
114                     self.StateCurrentCounter += 1
115 
116                 # Step 2
117                 for StateBufferIndex in range(len(StateBufferIndices)):
118                     StateBuffer2[StateBufferIndex] = StateBuffer[RightRotatedStateBufferIndices[StateBufferIndex]] ^
                        ↪  Binary_RightRotateMove(StateBuffer[LeftRotatedStateBufferIndices[StateBufferIndex]], 1)
119 
120                 # Step 3
121                 StateBuffer3[0] = BitsHashState[0] ^ StateBuffer2[0]
122                 for StateBufferIndex in range(1, len(StateBuffer3)):
123                     StateBuffer3[HashStateIndices[StateBufferIndex]] = Binary_RightRotateMove(BitsHashState[StateBufferIndex] ^
                        ↪  StateBuffer2[StateBufferIndex % len(StateBuffer2)], MoveBitCounts[self.StateCurrentCounter %
                        ↪  len(MoveBitCounts)])
124                     self.StateCurrentCounter += 1
125 
126                 # Step 4
127                 for StateBufferIndex in range(len(StateBuffer3)):
128                     BitsHashState[StateBufferIndex] = StateBuffer3[StateBufferIndex] ^ (~(StateBuffer3[(StateBufferIndex + 1) %
                        ↪  len(StateBuffer3)]) & StateBuffer3[(StateBufferIndex + 2) % len(StateBuffer3)])
129 
130                 # Step 5
131                 BitsHashState[0] ^= HASH_ROUND_CONSTANTS[RoundIndex % len(HASH_ROUND_CONSTANTS)]
132                 BitsHashState[BitsHashState_size - 1] ^= HASH_ROUND_CONSTANTS[(len(HASH_ROUND_CONSTANTS) - 1 - RoundIndex) %
                    ↪  len(HASH_ROUND_CONSTANTS)]
133 
134         def AbsorbInputData(self, ByteDatas: bytes) -> None:
135             BitWords = [0] * (len(ByteDatas) // 8)
136 
137             for i in range(0, len(ByteDatas), 8):
138                 BitWords[i//8] = int.from_bytes(ByteDatas[i:i+8], byteorder='little')
139 
140             for InputBytesIndex in range(BITWORDS_RATE):
141                 for OutputBytesIndex in range(0, len(BitWords)):
142                     if InputBytesIndex >= BITWORDS_RATE:
143                         InputBytesIndex = 0
144                     self.BitsHashState[InputBytesIndex] ^= BitWords[OutputBytesIndex]
145 
146                     # State permutation and transformation (string of information entropy pool)
147                     self.TransfromState(len(BitsHashState))
148 
149             # Clear sensitive information from BitWords
150             for i in range(len(BitWords)):
151                 BitWords[i] = 0
152 
153         def AbsorbInputData(self, BitWordDatas: List[int]) -> None:
154             for InputBitsIndex in range(BITWORDS_RATE):
155                 for OutputBitsIndex in range(len(BitWordDatas)):
156                     if InputBitsIndex >= BITWORDS_RATE:
157                         InputBitsIndex = 0
158                     self.BitsHashState[InputBitsIndex] ^= BitWordDatas[OutputBitsIndex]
159 
```

```
                # State permutation and transformation (string of information entropy pool)
                self.TransfromState(len(BitsHashState))

    def SqueezeOutputData(self, byte_datas):
        bit_words = [0] * (self.hash_bit_size // 64)
        bits_index_offset = 0
        for bits_index in range(len(bit_words)):
            bit_words[bits_index] = self.bits_hash_state[bits_index_offset]

            if bit_index >= bits_index_offset:
                # State permutation and transformation (string of information entropy pool)
                self.TransfromState(len(BitsHashState))
                bits_index_offset = 0

        for i in range(len(byte_datas) // 8):
            word = bit_words[i].to_bytes(8, byteorder='little')
            byte_datas[i*8:(i+1)*8] = word

    def SqueezeOutputData(self, word_datas):
        bits_index_offset = 0
        for bits_index in range(self.hash_bit_size // 64):
            word_datas[bits_index] = self.bits_hash_state[bits_index_offset]

            if bit_index >= bits_index_offset:
                # State permutation and transformation (string of information entropy pool)
                self.TransfromState(len(BitsHashState))
                bits_index_offset = 0

    def SecureHash(self, InputData, OuputData):
        BlockDataBuffer = list(InputData)

        # Pad data and Absorbing data stage
        if len(BlockDataBuffer) % self.BYTES_RATE != 0:
            for PadCount in range(len(BlockDataBuffer) % self.BYTES_RATE):
                BlockDataBuffer.append(self.PAD_BYTE_DATA)
        self.AbsorbInputData(BlockDataBuffer)

        # Squeeze data stage
        self.SqueezeOutputData(OuputData)

        # Clear the BlockDataBuffer
        for i in range(len(BlockDataBuffer)):
            BlockDataBuffer[i] = 0x00

        # If the hash summary data has been generated, the current state must be completely reset and cleaned up.
        # If you don't reset and clean, you will affect the quality of the hash function
        self.Reset()

    def SecureHash(self, InputData, OuputData):
        BlockDataBuffer = list(InputData)

        # Pad data and Absorbing data stage
        if len(BlockDataBuffer) % self.BITWORDS_RATE != 0:
            for PadCount in range(len(BlockDataBuffer) % self.BYTES_RATE):
                BlockDataBuffer.append(self.PAD_BITSWORD_DATA)
        self.AbsorbInputData(BlockDataBuffer)

        # Squeeze data stage
        self.SqueezeOutputData(OuputData)

        # Clear the BlockDataBuffer
```

```
            for i in range(len(BlockDataBuffer)):
                BlockDataBuffer[i] = 0x00

            # If the hash summary data has been generated, the current state must be completely reset and cleaned up.
            # If you don't reset and clean, you will affect the quality of the hash function
            self.Reset()

        def Reset(self):
            self.StateCurrentCounter = 0
            self.BitsHashState = [0] * (self.HashBitSize // 64)
```

Code block 3-1: Custom Secure Hash Class Based Sponge Function Structure (c++)

```
    template<std::uint64_t HashBitSize>
    /*
        https://en.wikipedia.org/wiki/Sponge_function

        基于海绵结构的密码学哈希函数，使用的伪随机置换函数由 Twilight-Dream 设计
        Cryptographic hash function based on sponge structure using a pseudo-random permutation function designed by Twilight-Dream

        Reference: https://locklessinc-com.translate.goog/articles/crypto_hash/?_x_tr_sl=en&_x_tr_tl=zh-CN&_x_tr_hl=zh-CN&_x_tr_pto=sc
    */
    class CustomSecureHash
    {

    private:

        /*
            哈希状态比特大小 = 比特率大小 + 比特容量大小

            例子:
            海绵函数的安全性取决于其内部状态的长度和块的长度。
            如果信息块的长度为 r 位，内部状态的长度为 w 位，那么有 c = w - r 位的内部状态不能被信息块所修改。
            c 的值被称为海绵的容量，海绵函数所保证的安全级别是 c/2。例如，要达到 256 位的安全与 64 位的消息块，内部状态应该是 w=2×256+64=576 位。
            当然，安全级别也取决于哈希值的长度 n。因此，碰撞攻击的复杂性是 2^{n/2} 和 2^{c/2} 之间的最小值，而第二次预像攻击的复杂性是 2^n 和 2^{c/2} 之间的最小值。

            为了安全起见，排列组合 P 应该表现得像一个随机排列组合，没有统计上的偏差，也没有让攻击者预测输出的数学结构。
            与基于压缩函数的哈希值一样，海绵函数也会对信息进行填充，但填充更简单，因为它不需要包括信息的长度。
            最后一个信息位只是由一个 1 位和尽可能多的 0 跟在后面。

            Hash state bits size =  Bits rate size + Bits capacity size

            Example :
            The security of a sponge function depends on the length of its internal state and the length of the blocks.
            If message blocks are r-bit long and the internal state is w-bit long, then there are c = w − r bits of the internal state that can' t be modified by message blocks.
            The value of c is called a sponge' s capacity, and the security level guaranteed by the sponge function is c/2. For example, to reach 256-bit security with 64-bit message blocks, the internal state should be w = 2 × 256 + 64 = 576 bits.
            Of course, the security level also depends on the length, n, of the hash value. The complexity of a collision attack is therefore the smallest value between 2^{n/2} and 2^{c/2}, while the complexity of a second preimage attack is the smallest value between 2^n and 2^{c/2}.

            To be secure, the permutation P should behave like a random permutation, without statistical bias and without a mathematical structure that would allow an attacker to predict outputs.
            As in compression function–based hashes, sponge functions also pad messages, but the padding is simpler because it doesn' t need to include the message' s length.
            The last message bit is simply followed by a 1 bit and as many zeroes as necessary.
        */
```

```
40
41          static constexpr std::uint64_t BITS_STATE_SIZE = 2 * HashBitSize + std::numeric_limits<std::uint64_t>::digits;
42          static constexpr std::uint64_t BITS_RATE = HashBitSize;
43          static constexpr std::uint64_t BITS_CAPACITY = BITS_STATE_SIZE - BITS_RATE;
44
45          static constexpr std::uint64_t BYTES_RATE = BITS_RATE / std::numeric_limits<std::uint8_t>::digits;
46          static constexpr std::uint64_t BITWORDS_RATE = BYTES_RATE / sizeof(std::uint64_t);
47          static constexpr std::uint64_t BYTES_CAPACITY = BITS_CAPACITY / std::numeric_limits<std::uint8_t>::digits;
48          static constexpr std::uint64_t BITWORDS_CAPACITY = BYTES_CAPACITY / sizeof(std::uint64_t);
49
50          static constexpr std::uint8_t PAD_BYTE_DATA = 0b00000001;
51          static constexpr std::uint64_t PAD_BITSWORD_DATA = 0b0000000100000001000000010000000100000001000000010000000100000001;
52
53          std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> BitsHashState {};
54
55          static constexpr std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2>
    ↪ StateBufferIndices = GenerateHashStateBufferIndices<BITS_STATE_SIZE>();
56
57          const std::array<std::uint32_t, 63> MoveBitCounts {};
58          const std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> HashStateIndices {};
59          std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2>
    ↪ LeftRotatedStateBufferIndices;
60          std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2>
    ↪ RightRotatedStateBufferIndices;
61
62          std::size_t StateCurrentCounter = 1;
63
64          /*
65              这里的伪随机数来源，可以是大素数的立方根或者平方根所生成的比特位，也可以是无理数所生成的比特位，也可以是严格设计的比特掩码
66              The source of the pseudo-random numbers here can be the bits generated by the cube root or square root of a large
    ↪ prime number, the bits generated by an irrational number, or a strictly designed bit mask
67          */
68          static constexpr std::array<std::uint64_t, 64> HASH_ROUND_CONSTANTS
69          {
70
    ↪ 0xe02d51d52e6988abULL,0xfc48780c20090b50ULL,0xc6144c4d89151352ULL,0xb98669bb3a32a8f1ULL,0xd4786928fe033c03ULL,0xaebb38f01d73faabUL
71
    ↪ 0x06d5b3dbf088ae77ULL,0x7e2be74e7f525e23ULL,0xe5459a079549e2e3ULL,0x352ba71a6a95e6d6ULL,0x7b40c16d92d5e43bULL,0xa559af839ba27363UL
72
    ↪ 0x9ab94838ff7737c6ULL,0x718d70cd883014f9ULL,0x0bda9af50ba21d4dULL,0xd88cb07c07a814d5ULL,0xa6c8d66f9b3d8933ULL,0x80643413e011c839UL
73
    ↪ 0x19224d7b455813b1ULL,0xb1dbd44f138bac7fULL,0x2ba9107bb26a6134ULL,0x48297fe2c4167b76ULL,0x776528a5edb8a68eULL,0x2381e0eb054681a8UL
74
    ↪ 0x655f38e3d5446574ULL,0xd8093b5a1172958cULL,0x28880627fe4c014bULL,0x0459d6592d1b2b51ULL,0x2aeb8df1c83b63beULL,0xcba3ca8c513a8205UL
75
    ↪ 0xdf8ee44352384448ULL,0xff38527afa3b13a2ULL,0x9ff904a86c03fe22ULL,0xe81a56aef956f93fULL,0x3c13136bf0612494ULL,0xca9b0621705e9748UL
76
    ↪ 0xd249f4efd3685008ULL,0xda2779c07b0e4a43ULL,0x1cc1bd402438ea81ULL,0x7b090a135f97ba29ULL,0xd25e80bc98b09e4bULL,0xeea820f2885ac1f8UL
77
    ↪ 0x75208f3a3cb244dfULL,0x20f74f61571512b4ULL,0xfd526ef256343eb7ULL,0x753082ea79791d09ULL,0x41a3a000a8c7ae30ULL,0xb2a056be3a257d27UL
78          };
79
80          std::array<std::uint32_t, 63>
81          GenerateRandomMoveBitCounts()
82          {
83              CommonSecurity::RNG_ISAAC::isaac64<8> CSPRNG = CommonSecurity::RNG_ISAAC::isaac64<8>(1946379852749613ULL);
84
85              CSPRNG.discard(1024);
86
87              std::array<std::uint32_t, 63> MoveBitCounts {};
88
```

```
89              std::iota(MoveBitCounts.begin(), MoveBitCounts.end(), 1);
90
91              for(std::uint64_t Index = 0; Index < MoveBitCounts.size(); ++Index)
92              {
93                  std::swap(MoveBitCounts[Index], MoveBitCounts[(Index + CSPRNG()) % MoveBitCounts.size()]);
94              }
95
96              return MoveBitCounts;
97          }
98
99          std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits>
100         GenerateRandomHashStateIndices()
101         {
102             CommonSecurity::RNG_ISAAC::isaac64<8> CSPRNG = CommonSecurity::RNG_ISAAC::isaac64<8>(1946379852749613ULL);
103
104             CSPRNG.discard(2048);
105
106             std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> RandomHashStateIndices {};
107
108             std::iota(RandomHashStateIndices.begin(), RandomHashStateIndices.end(), 0);
109
110             for(std::uint64_t Index = 0; Index < RandomHashStateIndices.size(); ++Index)
111             {
112                 std::swap(RandomHashStateIndices[Index], RandomHashStateIndices[(Index + CSPRNG()) %
    ↪ RandomHashStateIndices.size()]);
113             }
114
115             return RandomHashStateIndices;
116         }
117
118         /*
119             这个对应了海绵函数结构中的数学抽象的 F 函数（它应该是一个安全的伪随机置换函数）。
120
121             它有以下几个步骤：
122             1. 哈希状态混合
123             2. 应用线性函数
124             3. 应用比特伪随机置换（P 函数）
125             4. 应用非线性函数
126             5. 每一轮需要哈希状态和哈希使用的常量进行混合
127
128             This corresponds to the mathematical abstraction of the F function in the structure of the sponge function (it is
    ↪ supposed to be a safe pseudo-random permutation function).
129
130             It has the following steps:
131             1. Hash state mixing
132             2. Apply linear function
133             3. Apply bit pseudo-random permutation (P function)
134             4. Apply nonlinear functions
135             5. Each round requires a mix of hash state and constants used by the hash
136         */
137         void TransfromState(std::size_t Counter)
138         {
139             using CommonSecurity::Binary_LeftRotateMove;
140             using CommonSecurity::Binary_RightRotateMove;
141
142             std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> StateBuffer {};
143             std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> StateBuffer2 {};
144             std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> StateBuffer3 {};
145
146             for(std::size_t RoundIndex = BitsHashState.size() - 1 - Counter; RoundIndex < BitsHashState.size(); ++RoundIndex)
147             {
```

```
148                 //Step 1
149                 while(StateCurrentCounter % BitsHashState.size() != 0)
150                 {
151                     StateBuffer[StateCurrentCounter % StateBuffer.size()] = BitsHashState[StateCurrentCounter %
    ↪ BitsHashState.size()] ^ BitsHashState[(StateCurrentCounter + 1) % BitsHashState.size()];
152                     ++StateCurrentCounter;
153                     StateBuffer[StateCurrentCounter % StateBuffer.size()] = BitsHashState[(StateCurrentCounter + 2) %
    ↪ BitsHashState.size()] ^ BitsHashState[(StateCurrentCounter + 3) % BitsHashState.size()];
154                     ++StateCurrentCounter;
155                 }
156
157                 //Step 2
158                 for(std::size_t StateBufferIndex = 0; StateBufferIndex < StateBufferIndices.size(); ++StateBufferIndex)
159                 {
160                     StateBuffer2[StateBufferIndex] = StateBuffer[RightRotatedStateBufferIndices[StateBufferIndex]] ^
    ↪ Binary_RightRotateMove<std::uint64_t>(StateBuffer[LeftRotatedStateBufferIndices[StateBufferIndex]], 1);
161                 }
162
163                 //Step 3
164                 StateBuffer3[0] = BitsHashState[0] ^ StateBuffer2[0];
165
166                 for(std::size_t StateBufferIndex = 1; StateBufferIndex < StateBuffer3.size(); ++StateBufferIndex)
167                 {
168                     StateBuffer3[HashStateIndices[StateBufferIndex]] =
    ↪ Binary_RightRotateMove<std::uint64_t>(BitsHashState[StateBufferIndex] ^ StateBuffer2[StateBufferIndex %
    ↪ StateBuffer2.size()], MoveBitCounts[StateCurrentCounter % MoveBitCounts.size()]);
169                     ++StateCurrentCounter;
170                 }
171
172                 //Step 4
173                 for(std::size_t StateBufferIndex = 0; StateBufferIndex < StateBuffer3.size(); ++StateBufferIndex)
174                 {
175                     BitsHashState[StateBufferIndex] = StateBuffer3[StateBufferIndex] ^ ( ~(StateBuffer3[(StateBufferIndex + 1)
    ↪ % StateBuffer3.size()]) & StateBuffer3[(StateBufferIndex + 2) % StateBuffer3.size()] );
176                 }
177
178                 //Step 5
179                 BitsHashState[0] ^= HASH_ROUND_CONSTANTS[RoundIndex % HASH_ROUND_CONSTANTS.size()];
180                 BitsHashState[BitsHashState.size() - 1] ^= HASH_ROUND_CONSTANTS[(HASH_ROUND_CONSTANTS.size() - 1 - RoundIndex)
    ↪ % HASH_ROUND_CONSTANTS.size()];
181             }
182         }
183
184         void AbsorbInputData(std::span<const std::uint8_t> ByteDatas)
185         {
186             using CommonToolkit::IntegerExchangeBytes::MessagePacking;
187
188             std::vector<std::uint64_t> BitWords(ByteDatas.size() / sizeof(std::uint64_t), 0);
189
190             MessagePacking<std::uint64_t, std::uint8_t>(ByteDatas, BitWords.data());
191
192             for(std::uint64_t InputBytesIndex = 0, OutputBytesIndex = 0; OutputBytesIndex < BitWords.size(); ++InputBytesIndex,
    ↪ ++OutputBytesIndex)
193             {
194                 if(InputBytesIndex >= BITWORDS_RATE)
195                     InputBytesIndex = 0;
196                 BitsHashState[InputBytesIndex] ^= BitWords[OutputBytesIndex];
197
198                 //状态排列和变换 (信息熵池搅拌)
199                 //State permutation and transformation (string of information entropy pool)
200                 this->TransfromState(BitsHashState.size());
```

```
201             }
202 
203             memory_set_no_optimize_function<0x00>(BitWords.data(), BitWords.size() * sizeof(std::uint64_t));
204         }
205 
206         void AbsorbInputData(std::span<const std::uint64_t> BitWordDatas)
207         {
208             for(std::uint64_t InputBitsIndex = 0, OutputBitsIndex = 0; OutputBitsIndex < BitWordDatas.size(); ++InputBitsIndex,
    ↪  ++OutputBitsIndex)
209             {
210                 if(InputBitsIndex >= BITWORDS_RATE)
211                     InputBitsIndex = 0;
212                 BitsHashState[InputBitsIndex] ^= BitWordDatas[OutputBitsIndex];
213 
214                 //状态排列和变换 (信息熵池搅拌)
215                 //State permutation and transformation (string of information entropy pool)
216                 this->TransfromState(BitsHashState.size());
217             }
218         }
219 
220         void SqueezeOutputData(std::span<std::uint8_t> ByteDatas)
221         {
222             using CommonToolkit::IntegerExchangeBytes::MessageUnpacking;
223 
224             std::vector<std::uint64_t> BitWords(HashBitSize / std::numeric_limits<std::uint64_t>::digits, 0);
225 
226             size_t BitsIndexOffest = 0;
227 
228             for(std::uint64_t BitsIndex = 0; BitsIndex < BitWords.size(); ++BitsIndex)
229             {
230                 BitWords[BitsIndex] = BitsHashState[BitsIndexOffest];
231 
232                 if(BitsIndexOffest >= BITWORDS_RATE)
233                 {
234                     //状态排列和变换 (信息熵池搅拌)
235                     //State permutation and transformation (string of information entropy pool)
236                     this->TransfromState(BitsHashState.size());
237 
238                     BitsIndexOffest = 0;
239                 }
240             }
241 
242             MessageUnpacking<std::uint64_t, std::uint8_t>(BitWords, ByteDatas.data());
243         }
244 
245         void SqueezeOutputData(std::span<std::uint64_t> WordDatas)
246         {
247             size_t BitsIndexOffest = 0;
248 
249             for(std::uint64_t BitsIndex = 0; BitsIndex < (HashBitSize / std::numeric_limits<std::uint64_t>::digits);
    ↪  ++BitsIndex)
250             {
251                 WordDatas[BitsIndex] = BitsHashState[BitsIndexOffest];
252 
253                 if(BitsIndexOffest >= BITWORDS_RATE)
254                 {
255                     //状态排列和变换 (信息熵池搅拌)
256                     //State permutation and transformation (string of information entropy pool)
257                     this->TransfromState(BitsHashState.size());
258 
259                     BitsIndexOffest = 0;
```

```
260                 }
261             }
262         }
263 
264     public:
265 
266         void Reset()
267         {
268             this->StateCurrentCounter = 0;
269             memory_set_no_optimize_function<0x00>(BitsHashState.data(), BitsHashState.size() * sizeof(std::uint64_t));
270         }
271 
272         //不提供外部数据的测试
273         //Tests that do not provide external data
274         std::vector<std::uint64_t> Test()
275         {
276             for(std::size_t BlockCounter = 0; BlockCounter < (HashBitSize / std::numeric_limits<std::uint64_t>::digits);
    ↪ BlockCounter++)
277             {
278                 this->TransfromState(BlockCounter);
279             }
280 
281             std::vector<std::uint64_t> TestData(HashBitSize / std::numeric_limits<std::uint64_t>::digits, 0);
282             this->SqueezeOutputData(TestData);
283 
284             this->Reset();
285 
286             return TestData;
287         }
288 
289         void SecureHash
290         (
291             std::span<const std::uint8_t> InputData,
292             std::span<std::uint8_t> OuputData
293         )
294         {
295             std::vector<std::uint8_t> BlockDataBuffer(InputData.begin(), InputData.end());
296 
297             //填充数据和吸收数据阶段
298             //Pad data and Absorbing data stage
299             if(BlockDataBuffer.size() % BYTES_RATE != 0)
300             {
301                 for(std::size_t PadCount = 0; PadCount < BlockDataBuffer.size() % BYTES_RATE; ++PadCount)
302                 {
303                     BlockDataBuffer.push_back(PAD_BYTE_DATA);
304                 }
305             }
306             this->AbsorbInputData(BlockDataBuffer);
307 
308             //挤压数据阶段
309             //squeeze data stage
310             this->SqueezeOutputData(OuputData);
311 
312             memory_set_no_optimize_function<0x00>(BlockDataBuffer.data(), BlockDataBuffer.size());
313 
314             //如果已经生成哈希摘要数据，就必须把当前状态全部重置和清理
315             //如果不重置和清理，你将会影响哈希函数的质量
316             //If the hash summary data has been generated, the current state must be completely reset and cleaned up.
317             //If you don't reset and clean, you will affect the quality of the hash function
318             this->Reset();
319         }
```

```
320
321         void SecureHash
322         (
323             std::span<const std::uint64_t> InputData,
324             std::span<std::uint64_t> OuputData
325         )
326         {
327             std::vector<std::uint64_t> BlockDataBuffer(InputData.begin(), InputData.end());
328
329             //填充数据和吸收数据阶段
330             //Pad data and Absorbing data stage
331             if(BlockDataBuffer.size() % BITWORDS_RATE != 0)
332             {
333                 for(std::size_t PadCount = 0; PadCount < BlockDataBuffer.size() % BYTES_RATE; ++PadCount)
334                 {
335                     BlockDataBuffer.push_back(PAD_BITSWORD_DATA);
336                 }
337             }
338             this->AbsorbInputData(BlockDataBuffer);
339
340             //挤压数据阶段
341             //squeeze data stage
342             this->SqueezeOutputData(OuputData);
343
344             memory_set_no_optimize_function<0x00>(BlockDataBuffer.data(), BlockDataBuffer.size() * sizeof(std::uint64_t));
345
346             //如果已经生成哈希摘要数据，就必须把当前状态全部重置和清理
347             //如果不重置和清理，你将会影响哈希函数的质量
348             //If the hash summary data has been generated, the current state must be completely reset and cleaned up.
349             //If you don't reset and clean, you will affect the quality of the hash function
350             this->Reset();
351         }
352
353         CustomSecureHash()
354             :
355             MoveBitCounts(GenerateRandomMoveBitCounts()), HashStateIndices(GenerateRandomHashStateIndices())
356         {
357             static_assert(HashBitSize >= 128 && HashBitSize % 8 == 0, "");
358
359             std::ranges::rotate_copy(StateBufferIndices.begin(), StateBufferIndices.begin() + 1, StateBufferIndices.end(),
    ↪ LeftRotatedStateBufferIndices.begin());
360             std::ranges::rotate_copy(StateBufferIndices.begin(), StateBufferIndices.end() - 1, StateBufferIndices.end(),
    ↪ RightRotatedStateBufferIndices.begin());
361         }
362     };
```

Code block 4: ISAAC PRNG (c++)

```
1   /*
2       RNG_ISAAC contains code common to isaac and isaac64.
3       It uses CRTP (a.k.a. 'static polymorphism') to invoke specialized methods in the derived class templates,
4       avoiding the cost of virtual method invocations and allowing those methods to be placed inline by the compiler.
5       Applications should not specialize or instantiate this template directly.
6   */
7
8   template<std::size_t Alpha, class T>
9   class RNG_ISAAC
10  {
11  public:
12      using result_type = T;
13
14      static constexpr std::size_t state_size = 1 << Alpha;
```

```
15
16          static constexpr result_type default_seed = 0;
17
18          RNG_ISAAC()
19          {
20              seed(default_seed);
21          }
22
23          explicit RNG_ISAAC(result_type seed_number)
24              : issac_base_member_counter(state_size)
25          {
26              seed(seed_number);
27          }
28
29          template <typename SeedSeq>
30          requires( not std::convertible_to<SeedSeq, result_type> )
31          explicit RNG_ISAAC( SeedSeq& number_sequence )
32              : issac_base_member_counter(state_size)
33          {
34              seed(number_sequence);
35          }
36
37          RNG_ISAAC(const std::vector<result_type>& seed_vector)
38              : issac_base_member_counter(state_size)
39          {
40              seed(seed_vector);
41          }
42
43          template<class IteratorType>
44          RNG_ISAAC
45          (
46              IteratorType begin,
47              IteratorType end,
48              typename std::enable_if
49              <
50                      std::is_integral<typename std::iterator_traits<IteratorType>::value_type>::value &&
51                      std::is_unsigned<typename std::iterator_traits<IteratorType>::value_type>::value
52              >::type* = nullptr
53          )
54          : issac_base_member_counter(state_size)
55          {
56              seed(begin, end);
57          }
58
59          RNG_ISAAC(std::random_device& random_device_object)
60              : issac_base_member_counter(state_size)
61          {
62              seed(random_device_object);
63          }
64
65          RNG_ISAAC(const RNG_ISAAC& other)
66              : issac_base_member_counter(state_size)
67          {
68              for (std::size_t index = 0; index < state_size; ++index)
69              {
70                  issac_base_member_result[index] = other.issac_base_member_result[index];
71                  issac_base_member_memory[index] = other.issac_base_member_memory[index];
72              }
73              issac_base_member_register_a = other.issac_base_member_register_a;
74              issac_base_member_register_b = other.issac_base_member_register_b;
75              issac_base_member_register_c = other.issac_base_member_register_c;
```

```
76              issac_base_member_counter = other.issac_base_member_counter;
77          }
78
79      public:
80
81          static constexpr result_type min()
82          {
83              return std::numeric_limits<result_type>::min();
84          }
85          static constexpr result_type max()
86          {
87              return std::numeric_limits<result_type>::max();
88          }
89
90          inline void seed(result_type seed_number)
91          {
92              for (std::size_t index = 0; index < state_size; ++index)
93              {
94                  issac_base_member_result[index] = seed_number;
95              }
96              init();
97          }
98
99          template <typename SeedSeq>
100         requires( not std::convertible_to<SeedSeq, result_type> )
101         constexpr void seed( SeedSeq& number_sequence )
102         {
103             std::seed_seq my_seed_sequence(number_sequence.begin(), number_sequence.end());
104             std::array<result_type, state_size> seed_array;
105             my_seed_sequence.generate(seed_array.begin(), seed_array.end());
106             for (std::size_t index = 0; index < state_size; ++index)
107             {
108                 issac_base_member_result[index] = seed_array[index];
109             }
110             init();
111         }
112
113         template<class IteratorType>
114         inline typename std::enable_if
115         <
116             std::is_integral<typename std::iterator_traits<IteratorType>::value_type>::value &&
117             std::is_unsigned<typename std::iterator_traits<IteratorType>::value_type>::value, void
118         >::type
119         seed(IteratorType begin, IteratorType end)
120         {
121             IteratorType iterator = begin;
122             for (std::size_t index = 0; index < state_size; ++index)
123             {
124                 if (iterator == end)
125                 {
126                     iterator = begin;
127                 }
128                 issac_base_member_result[index] = *iterator;
129                 ++iterator;
130             }
131             init();
132         }
133
134         void seed(std::random_device& random_device_object)
135         {
136             std::vector<result_type> random_seed_vector;
```

```
137             random_seed_vector.reserve(state_size);
138             for (std::size_t round = 0; round < state_size; ++round)
139             {
140                 result_type seed_number_value = GenerateSecureRandomNumberSeed<result_type>(random_device_object);
141 
142                 std::size_t bytes_filled{sizeof(std::random_device::result_type)};
143                 while(bytes_filled < sizeof(result_type))
144                 {
145                     result_type seed_number_value2 = GenerateSecureRandomNumberSeed<result_type>(random_device_object);
146 
147                     seed_number_value <<= (sizeof(std::random_device::result_type) * 8);
148                     seed_number_value |= seed_number_value2;
149                     bytes_filled += sizeof(std::random_device::result_type);
150                 }
151                 random_seed_vector.push_back(seed_number_value);
152             }
153             seed(random_seed_vector.begin(), random_seed_vector.end());
154         }
155 
156         inline result_type operator()()
157         {
158             if(issac_base_member_counter - 1 == std::numeric_limits<std::size_t>::max())
159                 issac_base_member_counter = state_size - 1;
160 
161             return (!issac_base_member_counter--) ? (do_isaac(), issac_base_member_result[issac_base_member_counter]) :
    ↪  issac_base_member_result[issac_base_member_counter];
162         }
163 
164         inline void discard(unsigned long long z)
165         {
166             for (; z; --z) operator()();
167         }
168 
169         ~RNG_ISAAC() = default;
170 
171     private:
172 
173         /*
174             ISAAC (Indirection, Shift, Accumulate, Add, and Count) generates 32-bit random numbers.
175             Averaged out, it requires 18.75 machine cycles to generate each 32-bit value.
176             Cycles are guaranteed to be at least 2(^)40 values long, and they are 2(^)8295 values long on average.
177             The results are uniformly distributed, unbiased, and unpredictable unless you know the seed.
178         */
179 
180         void implementation_isaac()
181         {
182             /*
183                 Modulo a power of two, the following works (assuming twos complement representation):
184 
185                 i mod n == i & (n-1) when n is a power of two and mod is the aforementioned positive mod.
186                 (FYI: modulus is the common mathematical term for the "divisor" when a modulo operation is considered).
187 
188                 return i & (n-1);
189 
190                 auto lambda_Modulo = [](result_type value, result_type modulo_value)
191                 {
192                     return modulo_value & ( modulo_value - 1) ? value % modulo_value : value & ( modulo_value - 1);
193                 };
194             */
195 
196             result_type index = 0, x = 0, y = 0, state_random_value = 0;
```

```
197
198             result_type accumulate = this->issac_base_member_register_a;
199             result_type bit_result = this->issac_base_member_register_b + (++(this->issac_base_member_register_c)); //b ← (c +
    ↪  1)
200
201             for (index = 0; index < this->state_size; ++index)
202             {
203                 //x ← state[index]
204                 x = this->issac_base_member_memory[index];
205                 /*
206                     //barrel shift
207
208                     function(a, index)
209                     {
210                         if index   0 mod 4
211                             return a ^= a << 13
212                         if index   1 mod 4
213                             return a ^= a << 6
214                         if index   2 mod 4
215                             return a ^= a << 2
216                         if index   3 mod 4
217                             return a ^= a << 16
218                     }
219
220                     mix_index ← function(a, index);
221                 */
222                 switch (index & 3)
223                 {
224                     case 0:
225                         accumulate ^= accumulate << 13;
226                         break;
227                     case 1:
228                         accumulate ^= accumulate >>  6;
229                         break;
230                     case 2:
231                         accumulate ^= accumulate <<  2;
232                         break;
233                     case 3:
234                         accumulate ^= accumulate >> 16;
235                         break;
236                 }
237                 // a(mix_index) + state[index] + 128 mod 256
238                 accumulate += this->issac_base_member_memory[ (index + this->state_size / 2) & (this->state_size - 1) ];
239                 //state[index] ← a(mix_index)   b + (state[x] >>> 2) mod 256
240                 //y == state[index]
241                 state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(x, 2) &
    ↪  (this->state_size - 1) ];
242                 y = accumulate ^ bit_result + state_random_value;
243                 this->issac_base_member_memory[index] = y;
244                 //result[index] ← x + a(mix_index)   (state[state[index]] >>> 10) mod 256
245                 //b == result[index]
246                 state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(y, 10) &
    ↪  (this->state_size - 1) ];
247                 bit_result = x + accumulate ^ state_random_value;
248                 this->issac_base_member_result[index] = bit_result;
249             }
250         }
251
252         /*
253             ISAAC-64 generates a different sequence than ISAAC, but it uses the same principles. It uses 64-bit arithmetic.
```

```
254             It generates a 64-bit result every 19 instructions. All cycles are at least 2(^)72 values, and the average cycle
    ↪  length is 2(^)16583.
255
256             The following files implement ISAAC-64.
257             The constants were tuned for a 64-bit machine, and a complement was thrown in so that all-zero states become nonzero
    ↪  faster.
258         */
259
260         void implementation_isaac64()
261         {
262             /*
263                 Modulo a power of two, the following works (assuming twos complement representation):
264
265                 i mod n == i & (n-1) when n is a power of two and mod is the aforementioned positive mod.
266                 (FYI: modulus is the common mathematical term for the "divisor" when a modulo operation is considered).
267
268                 return i & (n-1);
269
270                 auto lambda_Modulo = [](result_type value, result_type modulo_value)
271                 {
272                     return modulo_value & ( modulo_value - 1) ? value % modulo_value : value & ( modulo_value - 1);
273                 };
274             */
275
276             result_type index = 0, x = 0, y = 0, state_random_value = 0;
277
278             result_type accumulate = this->issac_base_member_register_a;
279             result_type bit_result = this->issac_base_member_register_b + (++(this->issac_base_member_register_c)); //b ← (c +
    ↪  1)
280
281             for (index = 0; index < this->state_size; ++index)
282             {
283                 //x ← state[index]
284                 x = this->issac_base_member_memory[index];
285                 /*
286                     //barrel shift
287
288                     function(a, index)
289                     {
290                         if index  0 mod 4
291                             return a ^= ~(a << 21)
292                         if index  1 mod 4
293                             return a ^= a << 5
294                         if index  2 mod 4
295                             return a ^= a << 12
296                         if index  3 mod 4
297                             return a ^= a << 33
298                     }
299
300                     mix_index ← function(a, index);
301                 */
302                 switch (index & 3)
303                 {
304                     case 0:
305                         accumulate ^= ~(accumulate << 21);
306                         break;
307                     case 1:
308                         accumulate ^= accumulate >>  5;
309                         break;
310                     case 2:
311                         accumulate ^= accumulate << 12;
```

```
312                         break;
313                     case 3:
314                         accumulate ^= accumulate >> 33;
315                         break;
316                 }
317                 // a(mix_index) + state[index] + 128 mod 256
318                 accumulate += this->issac_base_member_memory[ (index + this->state_size / 2) & (this->state_size - 1) ];
319                 //state[index] ← a(mix_index)   b + (state[x] >>> 2) mod 256
320                 //y == state[index]
321                 state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(x, 2) &
    ↪  (this->state_size - 1) ];
322                 y = accumulate ^ bit_result + state_random_value;
323                 this->issac_base_member_memory[index] = y;
324                 //result[index] ← x + a(mix_index)   (state[state[index]] >>> 10) mod 256
325                 //b == result[index]
326                 state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(y, 10) &
    ↪  (this->state_size - 1) ];
327                 bit_result = x + accumulate ^ state_random_value;
328                 this->issac_base_member_result[index] = bit_result;
329             }
330         }
331 
332         void init()
333         {
334             result_type a = golden();
335             result_type b = golden();
336             result_type c = golden();
337             result_type d = golden();
338             result_type e = golden();
339             result_type f = golden();
340             result_type g = golden();
341             result_type h = golden();
342 
343             issac_base_member_register_a = 0;
344             issac_base_member_register_b = 0;
345             issac_base_member_register_c = 0;
346 
347             /* scramble it */
348             for (std::size_t index = 0; index < 4; ++index)
349             {
350                 mix(a,b,c,d,e,f,g,h);
351             }
352 
353             /* initialize using the contents of issac_base_member_result[] as the seed */
354             for (std::size_t index = 0; index < state_size; index += 8)
355             {
356                 a += issac_base_member_result[index];
357                 b += issac_base_member_result[index+1];
358                 c += issac_base_member_result[index+2];
359                 d += issac_base_member_result[index+3];
360                 e += issac_base_member_result[index+4];
361                 f += issac_base_member_result[index+5];
362                 g += issac_base_member_result[index+6];
363                 h += issac_base_member_result[index+7];
364 
365                 mix(a,b,c,d,e,f,g,h);
366 
367                 issac_base_member_memory[index] = a;
368                 issac_base_member_memory[index+1] = b;
369                 issac_base_member_memory[index+2] = c;
370                 issac_base_member_memory[index+3] = d;
```

```
                issac_base_member_memory[index+4] = e;
                issac_base_member_memory[index+5] = f;
                issac_base_member_memory[index+6] = g;
                issac_base_member_memory[index+7] = h;
            }

            /* do a second pass to make all of the seed affect all of issac_base_member_memory */
            for (std::size_t index = 0; index < state_size; index += 8)
            {
                a += issac_base_member_memory[index];
                b += issac_base_member_memory[index+1];
                c += issac_base_member_memory[index+2];
                d += issac_base_member_memory[index+3];
                e += issac_base_member_memory[index+4];
                f += issac_base_member_memory[index+5];
                g += issac_base_member_memory[index+6];
                h += issac_base_member_memory[index+7];

                mix(a,b,c,d,e,f,g,h);

                issac_base_member_memory[index] = a;
                issac_base_member_memory[index+1] = b;
                issac_base_member_memory[index+2] = c;
                issac_base_member_memory[index+3] = d;
                issac_base_member_memory[index+4] = e;
                issac_base_member_memory[index+5] = f;
                issac_base_member_memory[index+6] = g;
                issac_base_member_memory[index+7] = h;
            }

            /* fill in the first set of results */
            do_isaac();
        }

        inline void do_isaac()
        {
            if constexpr(std::same_as<result_type,std::uint32_t>)
                this->implementation_isaac();
            else if constexpr(std::same_as<result_type,std::uint64_t>)
                this->implementation_isaac64();
        }

        /* the golden ratio */
        inline result_type golden()
        {
            if constexpr(std::same_as<result_type,std::uint32_t>)
                return static_cast<std::uint32_t>(0x9e3779b9);
            else if constexpr(std::same_as<result_type,std::uint64_t>)
                return static_cast<std::uint64_t>(0x9e3779b97f4a7c13);
        }

        inline void mix(result_type& a, result_type& b, result_type& c, result_type& d, result_type& e, result_type& f,
↪  result_type& g, result_type& h)
        {
            if constexpr(std::same_as<result_type,std::uint32_t>)
            {
                a ^= b << 11;
                d += a;
                b += c;

                b ^= c >> 2;
```

```
431                 e += b;
432                 c += d;
433 
434                 c ^= d << 8;
435                 f += c;
436                 d += e;
437 
438                 d ^= e >> 16;
439                 g += d;
440                 e += f;
441 
442                 e ^= f << 10;
443                 h += e;
444                 f += g;
445 
446                 f ^= g >> 4;
447                 a += f;
448                 g += h;
449 
450                 g ^= h << 8;
451                 b += g;
452                 h += a;
453 
454                 h ^= a >> 9;
455                 c += h;
456                 a += b;
457             }
458             else if constexpr(std::same_as<result_type,std::uint64_t>)
459             {
460                 a -= e;
461                 f ^= h >> 9;
462                 h += a;
463 
464                 b -= f;
465                 g ^= a << 9;
466                 a += b;
467 
468                 c -= g;
469                 h ^= b >> 23;
470                 b += c;
471 
472                 d -= h;
473                 a ^= c << 15;
474                 c += d;
475 
476                 e -= a;
477                 b ^= d >> 14;
478                 d += e;
479 
480                 f -= b;
481                 c ^= e << 20;
482                 e += f;
483 
484                 g -= c;
485                 d ^= f >> 17;
486                 f += g;
487 
488                 h -= d;
489                 e ^= g << 14;
490                 g += h;
491             }
```

```
        }

        std::array<result_type, state_size> issac_base_member_result {};
        std::array<result_type, state_size> issac_base_member_memory {};
        result_type issac_base_member_register_a = 0;
        result_type issac_base_member_register_b = 0;
        result_type issac_base_member_register_c = 0;
        std::size_t^^Iissac_base_member_counter = 0;
    };

    template<std::size_t Alpha = 8>
    using isaac = RNG_ISAAC<Alpha, std::uint32_t>;

    template<std::size_t Alpha = 8>
    using isaac64 = RNG_ISAAC<Alpha, std::uint64_t>;
```

Code block 5: X constant subscript generation used by GenerationRoundSubkeys functio

```
    void GenerateDiffusionLayerPermuteIndices()
    {
        std::array<std::unordered_set<std::uint32_t>, 16> DiffusionLayerMatrixIndex
        {
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
            std::unordered_set<std::uint32_t>{},
        };

        std::array<std::uint32_t, 32> ArrayIndexData
        {
            //0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
            25,9,27,18,11,2,26,7,12,24,5,17,6,1,10,3,21,30,8,20,0,29,4,13,19,14,23,16,22,31,28,15
        };

        std::vector<std::uint32_t> VectorIndexData(ArrayIndexData.begin(), ArrayIndexData.end());

        CommonSecurity::RNG_ISAAC::isaac64<8> CSPRNG;
        CommonSecurity::RND::UniformIntegerDistribution<std::uint32_t> UniformDistribution;

        for(std::size_t Round = 0; Round < 10223; ++Round)
        {
            for(std::size_t X = 0; X < DiffusionLayerMatrixIndex.size(); ++X )
            {
                std::unordered_set<std::uint32_t> HashSet;
                while(HashSet.size() != 16)
                {
                    std::uint32_t RandomIndex = UniformDistribution(CSPRNG) % 32;
                    while (RandomIndex >= VectorIndexData.size())
                    {
                        RandomIndex = UniformDistribution(CSPRNG) % 32;
                    }
```

```
                HashSet.insert(VectorIndexData[RandomIndex]);
                VectorIndexData.erase(VectorIndexData.begin() + RandomIndex);

                if(VectorIndexData.empty())
                {
                    CommonSecurity::ShuffleRangeData(ArrayIndexData.begin(), ArrayIndexData.end(), CSPRNG);
                    VectorIndexData = std::vector<std::uint32_t>(ArrayIndexData.begin(), ArrayIndexData.end());
                }
            }
            DiffusionLayerMatrixIndex[X] = HashSet;

            if(VectorIndexData.empty())
            {
                CommonSecurity::ShuffleRangeData(ArrayIndexData.begin(), ArrayIndexData.end(), CSPRNG);
                VectorIndexData = std::vector<std::uint32_t>(ArrayIndexData.begin(), ArrayIndexData.end());
            }
        }
    }

    for( std::size_t X = DiffusionLayerMatrixIndex.size(); X > 0; --X )
    {
        for(const auto& Value : DiffusionLayerMatrixIndex[X - 1] )
            std::cout << "KeyStateX" << "[" << Value << "]" << ", ";

        std::cout << "\n";
    }

    std::cout << std::endl;

    for(std::size_t Round = 0; Round < 10223; ++Round)
    {
        for(std::size_t X = DiffusionLayerMatrixIndex.size(); X > 0; --X )
        {
            std::unordered_set<std::uint32_t> HashSet;
            while(HashSet.size() != 16)
            {
                std::uint32_t RandomIndex = UniformDistribution(CSPRNG) % 32;
                while (RandomIndex >= VectorIndexData.size())
                {
                    RandomIndex = UniformDistribution(CSPRNG) % 32;
                }
                HashSet.insert(VectorIndexData[RandomIndex]);
                VectorIndexData.erase(VectorIndexData.begin() + RandomIndex);

                if(VectorIndexData.empty())
                {
                    CommonSecurity::ShuffleRangeData(ArrayIndexData.begin(), ArrayIndexData.end(), CSPRNG);
                    VectorIndexData = std::vector<std::uint32_t>(ArrayIndexData.begin(), ArrayIndexData.end());
                }
            }
            DiffusionLayerMatrixIndex[X - 1] = HashSet;

            if(VectorIndexData.empty())
            {
                CommonSecurity::ShuffleRangeData(ArrayIndexData.begin(), ArrayIndexData.end(), CSPRNG);
                VectorIndexData = std::vector<std::uint32_t>(ArrayIndexData.begin(), ArrayIndexData.end());
            }
        }
    }

    for( std::size_t X = 0; X < DiffusionLayerMatrixIndex.size(); ++X )
```

```
106         {
107             for(const auto& Value : DiffusionLayerMatrixIndex[X] )
108                 std::cout << "KeyStateX" << "[" << Value << "]" << ", ";
109 
110             std::cout << "\n";
111         }
112 
113         std::cout << std::endl;
114     }
```