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

This paper presents a new symmetric encryption-decryption algorithm, "OaldresPuzzle_Cryptic", designed to resist the impact of future quantum computers on data security. The algorithm will utilize and implement various techniques, including a cryptographically secure pseudo-random number generator based on a chaos-theoretic system that simulates the trajectory of a two-segment pendulum; 1 independently designed and implemented nonlinear feedback shift register with mild chaotic properties; 1 linear feedback shift register with a sequence period length of 2 to the 128th power, 2 pairs of static byte substitution boxes for simulating A high-strength nonlinear granularity function generated by computational and integrable primitive polynomials in Galois finite fields; 2 dynamic byte substitution boxes; and the use of a line-number data structure with nonlinear feedback shift registers to further disrupt the regularity of the generated key data; a structure mimicking the ZUC sequence cipher design, and the use of dynamic byte substitution boxes to make each generated key unpredictable.

In addition, linear algebra operations such as affine transformations, Kronecker products, dot products, solution transpositions and accompanying matrices, as well as matrix addition, subtraction and multiplication are used. Boolean operations AND, OR, NOT, XOR, XNOR are used; these operations together form the subkey generation module of this algorithm and the subkey generation module used in each round of the round function.

The subkey data generated by the above two modules are used by the one-way functions designed in coordination with the Lai-Massey Scheme, which together construct an abstract and computationally indistinguishable secure pseudo-random function.

Despite the potential of ”OaldresPuzzle_Cryptic” to resist quantum computer attacks, it is important to note that its effectiveness has not been tested. The algorithm can be considered as a micro-innovation in the field of symmetric encryption-decryption, offering a new solution to the challenges of quantum computing in terms of data security.

Introduction

English:

OaldresPuzzle_Cryptic Algorithm is a future-oriented micro-innovation of symmetric (group/data block) encryption and decryption algorithms. In particular, it addresses the potential threats posed by quantum computers. As technology continues to advance, the need for more secure and robust encryption methods becomes increasingly important.

Traditional encryption methods, such as RSA and AES, were available on traditional bit-based computer platforms in the era before the maturity of quantum computers. It was possible to achieve the same level of quantum encryption as the traditional bit-based platforms at the cost of increasing the key length by a factor of two. to achieve the same level of quantum bit security. We cannot focus on Shor's algorithm and ignore the potential threat of Grover's algorithm, so it is necessary to develop new ways to defend against these threats, even after the maturity of quantum computers, so that the data security of classical bit-based computers can be guaranteed

OaldresPuzzle_Cryptic The algorithm solves this problem by using a combination of existing techniques and mathematical operations to create symmetric subkeys that are almost impossible to break, using a master key that goes through a subkey generation module of our design and a subkey generation module that is used in each round of the round function.

The principle involves a pseudo-random number generator using a chaos-theoretic system, a nonlinear feedback shift register with chaotic properties, a linear feedback shift register, two pairs of static byte substitution boxes simulating nonlinear strong functions (contained in the Galois finite field of 2^8), two dynamic byte substitution boxes, and various combinations of mathematical operations, including linear algebra and Boolean operations, etc., with the potential ability to resist known and future attacks on quantum computers.

An ideal solution for protecting data in files and small disks. The algorithm is designed to be highly secure, making it suitable for protecting a wide range of data, including sensitive and critical information. Meeting future data security needs will protect data from future quantum computers.

In this paper, we describe in detail the OaldresPuzzle_Cryptic algorithm and its various components. In addition, we will explore potential future developments and applications of the OaldresPuzzle_Cryptic algorithm in the field of data security, including the steps required to integrate it into existing systems and the infrastructure needed to use the algorithm. The potential scalability of the algorithm and its ability to adapt to future technological and quantum computing developments. The components of the algorithm are also described in detail.

In conclusion, this paper presents a new and innovative encryption-decryption algorithm that can resist quantum computing threats to data security.The OaldresPuzzle_Cryptic algorithm uses various techniques and mathematical operations to create a unique encryption-decryption key that is virtually unbreakable. This algorithm is highly secure and suitable for protecting critical data and information from future quantum computing attacks.

目录

1 Topic

In the event that the reader has difficulty understanding the introduction and abstract sections of this paper, it is necessary for me to reiterate that the theoretical framework used in this study is related to the field of cryptography and involves the application of symmetric encryption and decryption. Specifically, the framework used in the new OaldresPuzzle_Cryptic Algorithm for symmetric encryption and decryption is based on the Lai-Massey scheme, which possesses properties that make it resistant to quantum-based attacks. This framework has been supported by citations provided: ([3] [13]). In addition, I have fully implemented the F-function and H-function of the Lai-Massey scheme using C++ code. We will mention the formula of the OaldresPuzzle_Cryptic algorithm later.

2 Frequently Asked Questions

Q: What is the key length of the new algorithm and how does it differ from existing symmetric algorithms?

A: The data block size and key block size of the algorithm are both at least 512 bits. All of the tests I have conducted so far should be above 512 bits, but I have not strictly required this. I only specified that it should be greater than 512 bits and a multiple of 8 to meet the requirements of post-quantum cryptography. Because the C++ language I used is a static template parameter, I recommend passing template parameters in multiples of 64 bits.

Q: What mathematical and computational systems were used in the design of the new algorithm, and what contributions do they make to its security?

A: The mathematical principles and computational methods I used are basically explained in the abstract and introduction of my paper. I don't need to emphasize them again here. If you want more detailed information, I may provide a flowchart of the OPC algorithm module.

Q: What are the advantages and disadvantages of the new algorithm compared to existing symmetric algorithms?

A: The algorithm has flexible key lengths, longer keys, and better security. Algorithm speed has always been a controversial topic that cannot be avoided. However, I sacrificed speed for quality and security, so its application may not be very widespread. I recommend using this algorithm for processing small amounts of data to achieve its best performance.

Q: Has the algorithm been subjected to any attacks or security evaluations, and what were the results?

A: Currently, there is no ability to perform large-scale supercomputer tests, and I am actively seeking help from people from all walks of life to evaluate the feasibility of the algorithm.

Q: What is the impact of the new algorithm on the field of cryptography, and what contribution does it make to the development of this discipline?

A: Although my contribution may be very small, I hope my ideas can provide better suggestions and help to future professionals who study cryptography. In addition, I have made every effort to use the ideas and mathematical methods used by previous designers of symmetric encryption and decryption algorithms. Every time I design and implement the OPC algorithm code, I use it with caution. I hope you can give me confidence. Later, after I finish my explanation, I will explain some of the mathematical formulas that this algorithm will use.

The author of this paper has designed a new symmetric encryption and decryption algorithm and is requesting that it be tested for security against the most advanced existing computing systems. To achieve this goal, the author suggests subjecting the algorithm to attacks by supercomputers or quantum computers.

In summary, the newly designed symmetric encryption and decryption algorithm may address the shortcomings of existing algorithms and improve the security of symmetric encryption and decryption methods. However, it is important to thoroughly evaluate the security and effectiveness of the algorithm. The author of the algorithm is seeking support and resources to subject the algorithm to attacks from quantum computers or supercomputers in order to fully evaluate its security. Further research and testing are needed to determine the potential impact of this new algorithm on the field of cryptography.

Thank you for reading this far, I will now explain some of the mathematical formulas used by this algorithm and how to construct it. If you are interested, please continue reading. Very Thank you.

3 Known existing symmetric encryption and decryption frameworks and comparisons

We shall delineate the benefits and drawbacks of this particular framework for symmetric encryption and decryption. Additionally, our objective is to expound upon the distinctions between the Lai-Massey scheme framework utilized in this study and other comparable structures for symmetric encryption and decryption that have been acknowledged by experts in the field of cryptography.

1. Feistel Network

In cryptography, a Feistel cipher (also known as Luby–Rackoff block cipher) is a symmetric structure used to construct block ciphers. It was named after Horst Feistel, a German-born physicist and cryptographer who made groundbreaking research while working for IBM, and is often referred to as a Feistel network. In a Feistel cipher, encryption and decryption are very similar operations consisting of a fixed number of rounds, each of which involves running a function called the "round function".

The implementation of a Feistel network can be described as follows: Let B be the input block, $K_1, \ldots, K_n$ be the round keys. The input block B is first split into two halves of equal size, L and R. The round function is applied using the i-th round key. After one half is operated on by the round function, it is XORed with the other half using $\oplus_{64}$, the result replaces the original half, and the halves are exchanged. Then the other half is operated on by the round function, it is XORed with the original half using $\oplus_{64}$, the result replaces the original other half, and the halves are exchanged again...

L_0 and R_0, the encryption and decryption of a Feistel network, are defined as follows: Luby–Rackoff: 7 Rounds Are Enough for $2n^{1-\epsilon}$ Security

For each round index $i = 0, \ldots,$ **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

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

Where $\oplus_{64}$ denotes bitwise XOR, and $F(R_{i-1}, K_i)$ is the round function applied to input R_{i-1} and round key K_i.

The output of a Feistel network is the concatenation of R_n and L_n.

2. Substitution-Permutation Network

In cryptography, a substitution-permutation network (SPN) is a series of linked mathematical operations used in block cipher algorithms. Examples of encryption/decryption algorithms that use SPN are AES (Rijndael), 3-Way, Kalyna, Kuznyechik, PRESENT, SAFER, SHARK, and Square. Such networks take plaintext blocks and keys as input and apply several rounds or layers of substitution boxes (S-boxes) and permutation boxes (P-boxes) to produce ciphertext blocks. The S-boxes and P-boxes transform sub-blocks of the input bits into output bits. These transformations are typically efficiently implemented operations in hardware, such as XOR and bitwise rotation. The key is introduced in each round, usually in the form of ”round keys” derived from it. (In some designs, the S-box itself depends on the key.)

The implementation of an SPN can be described as follows: Let B be the input block, $K_1, \ldots, K_n$ be the round keys. An SPN consists of n rounds, each of which takes block B_i as input and outputs B_{i+1}.

The SPN is defined as follows:

For each round index $i = 0, \ldots,$ **message_block_size**, compute:

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

The symbol $\oplus_{64}$ denotes bitwise XOR. P represents a permutation function, and $S_1, \ldots, S_m$ are S-boxes applied to the input $B_i \oplus_{64} K_i$. The output of SPN is B_n.

The S-box can be viewed as a substitution function. For instance, in this paper (OPC algorithm - The bytes data secure substitution layer), the specific implementation process of this S function has been explained.

That is, each statement is of the form

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

3. Lai-Massey Scheme ([27] [11] [21])

In cryptography, Lai-Massey Scheme is similar in design to the Feistel Network. It uses a round function and a half-round function. The round function is a function that takes two inputs, a subkey and a data block, and returns an output of the same length as the data block. The half-round function takes two inputs and transforms them into two outputs. For any given round, the input is divided into two halves, left and right.

Initially, the input is passed to the half-round function. In each round, the difference between the inputs, along with a subkey, is passed to the round function, and the result of the round function is added to each input. Then the input is passed to the half-round function again. This process is repeated for a fixed number of times, and the final output is the encrypted data.

Due to its design, it has an advantage over Substitution-Permutation Network, as the round function does not need to have the bijection property. It can have the injection property, and the half-round function only needs to satisfy the bijection property. This makes it easier to invert and allows the round function to be arbitrarily complex. The decryption process is quite similar, except that the key schedule is reversed, the inverse function of the half-round function is used, and the output of the round function is subtracted instead of added. Due to the reflexive property of binary XOR operation, all addition and subtraction operations can be replaced by binary XOR operation.

The encryption and decryption of Lai-Massey Scheme, L_0 and R_0, are defined as follows:

Let F be the round function, H be the bijection property of the half-round function, and let $K_0, K_1, \ldots, K_n$ be the subkeys of each round, numbered $0, 1, \ldots, n$. The input block B is first split into two equal-sized halves, L and R. For each round index $i = 0, \ldots,$ **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

We adopt an approach that explains the design from the bottom up to the top-level implementation.

The structure of the OPC encryption and decryption functions is actually very simple and can be explained using the following formula:

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

The key generation system, comprising of the GenerateSubkeys and GenerateRoundSubkeys functions, is an integral part of the framework discussed in this article. These functions are responsible for generating the subkeys and keys required for the round functions, which will be discussed in greater detail in subsequent sections.

The pseudo-random number generators (PRNGs) used in our key generation system can be categorized into three types.

The first type is a linear feedback shift register (LFSR) with a sequence period length of 2^{128}.

The second type is a non-linear feedback shift register (NLFSR) that we designed, which exhibits chaotic properties. There are two different implementations of the NLFSR, which we refer to as the "big version" and the "little version". The theories behind the two versions are different, and the small version is a true NLFSR. In contrast, the large version utilizes transcendental or irrational numbers, selecting random digits after the decimal point and converting them to a binary representation of 64 bits. After several polynomial calculations, data diffusion operations, and bit manipulations, an unpredictable bit sequence is generated.

The third type utilizes chaotic theory to generate secure pseudo-random number sequences. However, the system used in this approach is based on simulating the physical phenomenon of a double-pendulum. The input key undergoes a series of transformations to obtain a set of system parameters that can be used by the chaotic system. Moreover, the output states will be different for different input parameters, owing to the characteristic behavior of chaotic systems.

The specific implementation and structure of the PRNG algorithms for these three types will be discussed in the section entitled "Used PRNG Detail Component Implementation". [15]

4.1 Predependent algorithms for key generation systems

Prior to delving into the intricacies of the OaldresPuzzle_Cryptic algorithm, it is pertinent to examine the F-function of said algorithm within the framework of the Lai-Massey scheme. Notably, the key generation system required for this F-function is contingent upon the formulas of other algorithms.

The functions of the linear feedback shift register used are as follows:

Algorithm 1 OPC core algorithm - LFSR

1: Define variable state: $state \leftarrow \left[a, b\right]$
2: 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
end function

function POLYNOMIAL(a, b)
  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: $x^{128} \oplus_{128} x^{41} \oplus_{128} x^{39} \oplus_{128} x \oplus_{128} 1$
end function

The function of the self-designed nonlinear feedback shift register used is as follows:

Algorithm 2 OPC core algorithm - NLFSR

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

function __RANDOM__BITS__($number, select$) ▷ Compute pseudo-random bit sequences in binary
  Input: a number *number* and an integer *select* in the range [0, 8].
  Output: a new value for *number*.
  $result := \neg_{64}(number) + 1$
  if $select = 0$ **then**
    $result := result \wedge_{64} (2^{23} \vee_{64} 2^{10} \vee_{64} 2^{9} \vee_{64} 2^{8} \vee_{64} 2^{6} \vee_{64} 2^{4} \vee_{64} 2^{3} \vee_{64} 1)$

else if $select = 1$ **then**
$result := result \wedge_{64} (2^{54} \vee_{64} 2^{10} \vee_{64} 2^9 \vee_{64} 2^8 \vee_{64} 2^7 \vee_{64} 2^6 \vee_{64} 2^5 \vee_{64} 2^4 \vee_{64} 2^3 \vee_{64} 2^2)$
else if $select = 2$ **then**
$result := result \wedge_{64} (2^{47} \vee_{64} 2^{11} \vee_{64} 2^{10} \vee_{64} 2^8 \vee_{64} x^5 \vee_{64} 2^4 \vee_{64} 2^3 \vee_{64} 1)$
else if $select = 3$ **then**
$result := result \wedge_{64} (2^{30} \vee_{64} 2^9 \vee_{64} 2^8 \vee_{64} 2^7 \vee_{64} 2^5 \vee_{64} 2^4 \vee_{64} 2^3 \vee_{64} 2^2)$
else if $select = 4$ **then**
$result := result \wedge_{64} (2^{63} \vee_{64} 2^{12} \vee_{64} 2^9 \vee_{64} 2^8 \vee_{64} 2^5 \vee_{64} 2^2)$
else if $select = 5$ **then**
$result := result \wedge_{64} (2^{26} \vee_{64} 2^{10} \vee_{64} 2^3 \vee_{64} 2^2 \vee_{64} 2 \vee_{64} 1)$
else if $select = 6$ **then**
$result := result \wedge_{64} (2^6 \vee_{64} 1)$
else if $select = 7$ **then**
$result := result \wedge_{64} (2^{15} \vee_{64} 2^{10} \vee_{64} 2^7 \vee_{64} 2^5 \vee_{64} 2^4 \vee_{64} 2^3 \vee_{64} 2^2 \vee_{64} 2^1 \vee_{64} 1)$
else
$result := result \wedge_{64} (2^{41} \vee_{64} 2^{11} \vee_{64} 2^{10} \vee_{64} 2^8 \vee_{64} 2^6 \vee_{64} 2^5 \vee_{64} 2^4 \vee_{64} 2^3 \vee_{64} 2^2 \vee_{64} 2^1)$
end if
end function

function RANDOM_BITS($number, select, bit$)
$number :=$ __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
$number := number \gg_{64} 1$
return $number$
end function

function INITIALIZE($seed$)
if $seed \neq 0$ **then**
$a \leftrightarrow state_0$
$b \leftrightarrow state_1$
$c \leftrightarrow state_2$
$d \leftrightarrow state_3$
$a := seed$
$b := seed \boxtimes_{64} 2 \boxplus_{64} 1$
$c := seed \boxtimes_{64} 3 \boxplus_{64} 2$
$d := seed \boxtimes_{64} 4 \boxplus_{64} 3$
$a := a \boxplus_{64} ((b \oplus_{64} c) \oplus_{64} (\neg d))$
$b := b \boxminus_{64} ((b \wedge_{64} d) \vee_{64} a)$
$c := c \boxplus_{64} ((d \oplus_{64} a) \oplus_{64} (\neg b))$

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

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

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

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

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

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

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

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

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

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

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

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

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

$state_0 := state_1$

$state_1 := state_2$

$state_2 := state_3$

$state_3 := temporary\_state$

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

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

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

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

end if

end for

end if

return *random_numbers*

end function

function generate_chaotic_number($\mathbb{F}_2^{64}$ execute_count) ▷ **This is big version**

fibonacci_bits := **Bits64**(123581321345589144)

pi_bits := **Bits64** $((\pi - 3) \times 10^{64})$

euler_bits := **Bits64** $((e - 2) \times 10^{64})$

gold_ratio_bits := **Bits64** $((\phi - 1) \times 10^{64})$

if execute_count ≥ 8 **then**

AA $\leftrightarrow$ *state_0*

BB $\leftrightarrow$ *state_1*

CC $\leftrightarrow$ *state_2*

DD $\leftrightarrow$ *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
                    sequence_bytes := pi_bits
                end if
                repeat
                    bytes1 := GaloisFiniteField256_Multiplication(bytes1, sequence_bytes)   ▷ bytes1_index ×_GF sequence_bytes_index
                until executed 8 count
                CC := CC ⊕64 Bits64FromBytes(bytes1)
                if CC ∧64 1 = 0 then
                    CC := CC ⊕64 fibonacci_bits
                end if
            else
                CC ⟵ CC ⊕64 (gold_ratio_bits ⊕64 answer)
                if (CC ∧64 1) ≠ 0 then
                    CC ⟵ CC ⊕64 pi_bits
                end if
            end if
            if (round mod 2) = 0 then
```

random_number := ((answer $\gg_{64}$ 17) $\oplus_{64}$ BB)

AA := (AA $\wedge_{64}$ DD)

if AA = 0 **then**

AA := AA + (CC $\times$ 2)

end if

answer := answer $\oplus_{64}$ RANDOM_BITS(AA, random_number mod 9, (DD $\wedge_{64}$ 1) $\oplus_{64}$ bit)

DD := (DD $\wedge_{64}$ AA)

if DD = 0 **then**

DD := DD - (BB $\times$ 2)

end if

else

BB := BB $\oplus_{64}$ ((answer $\oplus_{64}$ AA) $\ggg_{64}$ (DD - CC) mod 64)

CC := CC $\oplus_{64}$ (BB $\ll_{64}$ (DD + AA) mod 64)

DD := DD $\oplus_{64}$ (CC $\ll_{64}$ (BB + AA) mod 64)

AA := AA $\oplus_{64}$ ((answer $\oplus_{64}$ DD) $\lll_{64}$ (BB - CC) mod 64)

aa,bb := PSEUDOHADAMARDFORWARDTRANSFORM(AA, BB)

if aa = 0 **then**

aa := bit

else if bb = 0 **then**

bb := bit

end if

cc,dd := PSEUDOHADAMARDBACKWARDTRANSFORM(CC, DD)

if cc = 0 **then**

cc := bit

else if dd = 0 **then**

dd := bit

end if

AA := AA $\oplus_{64}$ aa

BB := BB $\oplus_{64}$ bb

CC := CC $\oplus_{64}$ cc

DD := DD $\oplus_{64}$ dd

aa,bb,cc,dd := 0

answer := answer $\oplus_{64}$ (AA $\oplus_{64}$ BB $\oplus_{64}$ CC $\oplus_{64}$ DD)

end if

end for

end if

return answer $\oplus_{64}$ ((answer $\ll_{64}$ 17) $\vee_{64}$ (answer $\gg_{64}$ 42))

end function

function unpredictable_bits($\mathbb{F}_2^{64}$ base_number, $\mathbb{F}_2^{64}$ number_bits) ▷ **This is little version**

$answer = base\_number$

$current\_random\_bit = 0$

$current\_random\_bits = \{0, 0, 0, 0 | \forall element \in \mathbb{F}_2^8\}$

for $round\_counter = 0$ **to** $number\_bits - 1$, $round\_counter := round\_counter + 1$ **do**

$current\_random\_bit := ((state_0 \oplus_{64} state_1 \oplus_{64} state_2 \oplus_{64} state_3) \gg_{64} 63) \wedge_{64} 1$

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

$answer := answer \vee_{64} current_{r}andom_{b}it$ ▷ The answer random number is 0 or 1

$state_0 :=$ RANDOM_BITS($state_0$, $(state_3 \oplus_{64} state_2) \pmod 9$, current_random_bit)

$current\_random\_bits_0 := current\_random\_bits_0 \oplus_{64} (state_0 \wedge_{64} 1)$ ▷ Only one binary random bit is switched

$state_1 :=$ RANDOM_BITS($state_1$, $(state_2 \oplus_{64} state_1) \pmod 9$, current_random_bit)

$current\_random\_bits_1 := current\_random\_bits_2 \oplus_{64} (state_1 \wedge_{64} 1)$ ▷ Only one binary random bit is switched

$state_2 :=$ RANDOM_BITS($state_2$, $(state_1 \oplus_{64} state_0) \pmod 9$, current_random_bit)

$current\_random\_bits_2 := current\_random\_bits_2 \oplus_{64} (state_2 \wedge_{64} 1)$ ▷ Only one binary random bit is switched

$state_3 :=$ RANDOM_BITS($state_3$, $(state_0 \oplus_{64} state_3) \pmod 9$, current_random_bit)

$current\_random\_bits_3 := current\_random\_bits_3 \oplus_{64} (state_3 \wedge_{64} 1)$ ▷ Only one binary random bit is switched

$value_a \rightarrow current\_random\_bits_0 \vee_{64} current\_random\_bits_1$

$value_b \rightarrow current\_random\_bits_1 \wedge_{64} current\_random\_bits_2$

$value_c \rightarrow current\_random\_bits_2 \vee_{64} current\_random\_bits_3$

$value_d \rightarrow current\_random\_bits_3 \wedge_{64} current\_random\_bits_0$ ▷ The temporary values

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

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

$answer := answer \vee_{64} current_{r}andom_{b}it$ ▷ The answer random number is 0 or 1

$value\_a \rightarrow state_0 \pmod 4$

$value\_b \rightarrow state_1 \pmod 4$

$value\_c \rightarrow state_2 \pmod 4$

$value\_d \rightarrow state_3 \pmod 4$ ▷ The temporary values

$\textsc{Swap}(current\_random\_bits_{value\_a}, current\_random\_bits_3)$

$\textsc{Swap}(current\_random\_bits_{value\_b}, current\_random\_bits_3)$

$\textsc{Swap}(current\_random\_bits_{value\_c}, current\_random\_bits_3)$

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

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

$state_1 := state_1 \vee_{64} ((state_0 \wedge_{64} 1) \ll_{64} 63)$
$state_2 := state_2 \gg_{64} 1$
$state_2 := state_2 \vee_{64} ((state_1 \wedge_{64} 1) \ll_{64} 63)$
$state_3 := state_3 \gg_{64} 1$
$state_3 := state_3 \vee_{64} ((state_2 \wedge_{64} 1) \ll_{64} 63)$;
$state_0 := state_0 \gg_{64} 1$
$state_0 := state_0 \vee_{64} ((state_3 \wedge_{64} 1) \ll_{64} 63)$ ▷ 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
end for
return answer
end function

The equation of the chaos theory system used is as follows:

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

This section delves into the intricacies of the wheel functions employed in the OaldresPuzzle_Cryptic algorithm, along with the implementation of the relevant formulas. Furthermore, a pair of byte substitution boxes are utilized to establish a data substitution layer that affords diffusivity, obfuscation, and nonlinear regularity. It should be noted that this aspect falls outside the purview of our investigation into the Lai-Massey scheme framework, and is instead an adaptation made to the ultimate outcome of said framework.

For EncryptionWithOPC and DecryptionWithOPC, the implementation of the 2 round functions, we can simply divide into 2 structures.

Algorithm 3 OPC core algorithm - The encrytion and decryption

Require: None
Ensure: None
function EncryptionWithOPC($PlainDataVetcor$)
repeat

EncrytionByLaiMasseyFramework($PlainDataVetcor, RoundSubkeys$)
ForwardBytesSubstitution($PlainDataVetcor$)
until executed 16 round
end function

function DecryptionWithOPC($CipherDataVector$)
repeat
BackwardBytesSubstitution($CipherDataVector$)
DecrytionByLaiMasseyFramework($CipherDataVector, RoundSubkeys$)
until executed 16 round
end function

The implementation of the EncryptionWithOPC and DecryptionWithOPC round functions is not yet complete, as the GenerateSubKeys and GenerateRoundSubKeys functions must be finalized before the Lai-Massey scheme framework can be fully built. However, we will first examine the above two functions using the implementation of two internal functions, EncrytionByLaiMasseyFramework and DecrytionByLaiMasseyFramework.Once this is complete, we will move on to the implementation of the GenerateSubKeys and GenerateRoundSubKeys functions.

Our proposed scheme shares a structural similarity with the Lai-Massey scheme, with the primary distinction lying in the sequencing of the F and H functions. In case the reader requires a refresher on the workings of this framework, we would direct their attention to the section titled (Known existing symmetric encryption and decryption frameworks and comparison).

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 $WordData$

The SecureRoundSubkeyGeneratationModule is class, The Instance Object Alias Name is SRSGM
$LeftWordData \in \mathbb{F}_2^{32}$ 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**($LeftWordData \oplus_{32} RightWordData, W$
$LeftWordData := LeftWordData \oplus_{32} TransformKey$
$RightWordData := RightWordData \oplus_{32} TransformKey$
$\{LeftWordData, RightWordData\} :=$ 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

Apart from the aforementioned two functions, we will also address the implementation of two additional internal functions, ForwardBytesSubstitution and BackwardBytesSubstitution. These functions entail four byte substitution boxes that encompass two sets of cryptographically robust nonlinear functions in both forward and backward directions. The byte substitution box data is then employed to define a function that enables the secure substitution of bytes data.

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
    }
```

The importance of the data and order of the two pairs of byte substitution boxes specified in this paper. This is due to the fact that the implementation of these byte substitution boxes is, by nature, a mathematically rigorously proven nonlinear function, and do not attempt to modify or optimize the implementation without a thorough understanding of the underlying mathematical principles and without being able to demonstrate that the modified implementation has equivalent nonlinear granularity. For a deeper understanding of this issue, see the literature for details of all the evaluation criteria for cryptographically secure replacement box implementations. [1] [8] [16]

The authors kindly remind the reader that the source code blocks in the files are not meant to be compiled and executed as actual code. Instead, they serve as a visual representation to explain various concepts and ideas. The authors emphasize the importance of thoroughly reading all accompanying explanations and mathematical formulas in order to fully understand the concepts presented. If the reader does not consider the source code block in its entirety, it may indicate that a detail has been missed or overlooked. The meaning of this document may be misunderstood. Also, if there are any errors in this paper, please feel free to contact the author at his email address.

4.3 Workflow detail - Key generation system

For the implementation of the GenerateSubkeys and GenerateRoundSubkeys functions, we still have a lot of work to do.

Next, we need to define a data structure called (CommonStateData), which will be used later when explaining the key generation system.

First, this data structure needs to use 2 immutable $\mathbb{F}_2^{32}$ integers, the first integer is DataBlockSize, which represents the element size of the data block; the second integer is KeyBlockSize, which represents the element size of the master key block.

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

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

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

To meet the requirements of future quantum-resistant ciphers, DataBlockSize is recommended to be greater than or equal to 4, and KeyBlockSize is recommended to be greater than or equal to 8; because 64 bits of 4 elements are equal to 256 bits, and then 64 bits of 8 elements are equal to 512 bits.

In addition, in this data structure, the three pseudo-random number generator algorithms that we mentioned before in (Predecessor algorithms for key generation systems) require instances of these three algorithm data structure objects, namely LFSR, NLFSR and SDP.

Moreover, we need to define two state data, representing the state matrices of the subkey data and the round key data, respectively, in this data structure. Both matrices are square matrices with consistent rows and columns. Their lengths are determined by a simple calculation based on the two immutable integers requested earlier.

Here is how the chunk size is computed:

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

Now define 2 matrices:

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

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

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

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

In addition, this data structure includes an object instance of the Bernoulli distribution, which is responsible for adjusting the bit-level probabilities of the pseudo-random number generator results (typically 64 bits of data for input and output). The probability of generating 0 and 1 bits is set to 50%

We can express it mathematically as follows:

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

Here, x is a binary random variable whose value is 0 or 1, and *probability* denotes the probability that the variable takes the value 1.

Next, a vector is defined in the data structure to store the index numbers of the rows and columns used to define the two matrices mentioned earlier. These index numbers are used to access the matrices, and the data stored in this vector will be shuffled at some point.

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

This is the shuffling algorithm we use.

Note: In contrast to the original Fisher-Yates Shuffle algorithm, the results generated by the Flavor Water pseudo-random generator are not utilized directly but must first undergo a uniform integer distribution with a specific range of numbers before they can be utilized.

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
```

Then an array is defined in the data structure, which is used to store the master key data. After the algorithm has run for N rounds, this vector data will be modified. The pseudocode for when it is modified will be explained in detail when we discuss the outermost wrapper function of the algorithm.

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

Finally, in the data structure, an empty vector is defined to store the initial data set for pseudo-randomness, which is populated by other byte data vectors. Although its length is variable, the length of the other byte data vector must be $DataBlockSize \pmod 8 = 0$.

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

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

The meaning of the **IntegerToBytes** and **IntegerFromBytes** functions is stipulated here, and will not be repeated in the future.

And a similar structure will be used later in this article.

The $ExampleBytes =$ **IntegerToBytes**($ExampleInteger$) function inputs a multiple of byte data, and then outputs the integer data of the multiple size corresponding to the byte data.(and pay attention to the byte endianness of the computer)

For example, convert eight 8-bit byte data into one 64-bit integer data, if both input and output are arrays, then repeat this operation

The $ExampleInteger =$ **IntegerFromBytes**($ExampleBytes$) function inputs the integer data of the number of multiples, and then outputs the byte data of the multiple size corresponding to the integer data.(and pay attention to the byte endianness of the computer)

For example, convert one 64-bit integer data into eight 8-bit byte data, if both input and output are arrays, then repeat this operation

We also need to define the operators between matrices and vectors to be used later in the presentation of the algorithm.

Where $+_{MATRIX}$ represents the matrix addition, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $-_{MATRIX}$ represents the matrix subtraction, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $\times_{MATRIX}$ represents the matrix multiplication, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $+_{VECTOR}$ represents the vector addition, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $-_{VECTOR}$ represents the vector subtraction, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $\times_{VECTOR}$ represents the vector multiplication, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $\times_{\mathbb{VEW}}$ represents the vector element-wise multiplication, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $\times_{\mathbb{MVE}}$ represents matrix-vector multiplication, but this *result* still belongs to the Galois finite field 2^{bit_count}

Where $\times_{SCALAR}$ represents the multiplication of a matrix or vector with a scalar, but this *result* still belongs to the Galois finite field 2^{bit_count}

Where $\times_{KRONECKER}$ represents the Kronecker product operation, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $\times_{DOT}$ represents the dot product operation, But this *result* still belongs to Galois finite field 2^{bit_count}

Where $NameMatrix^{Transpose}$ means to solve the transpose matrix of NameMatrix,But this *result* still belongs to Galois finite field 2^{bit_count}

Where $NameMatrix^{HermitianTranspose}$ means solving the conjugate transpose matrix of NameMatrix,But this *result* still belongs to Galois finite field 2^{bit_count}

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

Provide 3 (different/same) seeds for initializing 3 pseudo-random number generators.

$$SeedValue \neq 0, SeedValue \in \mathbb{F}_2^{64}$$
$$CommonStateData.LFSR.\textbf{seed}(1 \ or \ SeedValue \in \mathbb{F}_2^{64})$$
$$CommonStateData.NLFSR.\textbf{seed}(1 \ or \ SeedValue \in \mathbb{F}_2^{64})$$
$$CommonStateData.SDP.\textbf{seed}(13249961062380153450 \ \textbf{or} \ SeedValue \geq 10000000000 \ \textbf{and} \ SeedValue \in \mathbb{F}_2^{64})$$

Provide initial vector data (note: this data must be independent, and should not be related to plaintext, ciphertext, or master key).

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

We will show the ApplyWordDataInitialVector function from algorithm in detail next.

Algorithm 7 Apply Word Data Initial Vector

1: **function** ApplyWordDataInitialVector($WordDataInitialVector$)
2: $RandomQuadWordMatrix =$ **ReferenceObject**($CommonStateData.RandomQuadWordMatrix$) ▷ Initial sampling of Word data (Use 32Bit Word Data - Initial Vector)
3: $Word32Bit\_ExpandedInitialVector =$ Word32Bit_ExpandKey($WordDataInitialVector$)
4: $Index = Word32Bit\_ExpandedInitialVector.size()$
5: $MatrixRow = KeyRows \ from \ RandomQuadWordMatrix$
6: $MatrixColumn = KeyColumns \ from \ RandomQuadWordMatrix$
7: Flag **Use32BitData**
8: **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
$\mathbb{F}_2^{64} RandomValue = Word32Bit\_ExpandedInitialVector_{Index-1}$
$\mathbb{F}_2^{64} RotatedBits = (RandomValue \lll_{64} 7) \vee_{64} (RandomValue \ggg_{64} 1)$
$Position \to \{MatrixRow - 1, MatrixColumn - 1\}$
$MatrixValue \leftrightarrow RandomQuadWordMatrix_{Position}$ ▷ Access the value reference from the state key MatrixA
$MatrixValue := RandomValue \oplus_{64} (RandomValue \wedge_{64} RotatedBits)$ ▷ Random bits
$MatrixValue := MatrixValue \oplus_{64} (1 \lll_{64} (RandomValue \pmod{64}))$ ▷ 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**
$\mathbb{F}_2^{32} RestructedWordKey = \textsc{WordBitRestruct}(NeedHashDataWords_{NeedHashDataIndex})$ ▷ Data word do bit reorganization
if Data endian order is big **then**
BYTESWAP(RestructedWordKey)
end if
$\mathbb{F}_2^{32} UpPartWord, DownPartWord, LeftPartWord, RightPartWord = 0$
$UpPartWord := (RestructedWordKey \ggg_{32} 16)$ ▷ Data words do bit splitting: Reserve the **High 16 bits**
$DownPartWord := (RestructedWordKey \lll_{32} 16) \ggg_{32} 16$ ▷ Data words do bit splitting: Reserve the **Low 16 bits**
$LeftPartWord := (RestructedWordKey \wedge_{32} 0xF000'0000) \vee_{32} ((RestructedWordKey \wedge_{32} 0x00F0'0000) \lll_{32} 4) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'F000) \lll_{32} 8) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'00F0) \lll_{32} 12)$ ▷ Data words do bit splitting: Concatenate all data at bit positions **28∼31, 20∼23, 12∼15, 4∼7**
$RightPartWord := ((RestructedWordKey \wedge_{32} 0x0F00'0000) \lll_{32} 4) \vee_{32} ((RestructedWordKey \wedge_{32} 0x000F'0000) \lll_{32} 8) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'0F00U) \lll_{32} 12) \vee_{32} ((RestructedWordKey \wedge_{32} 0x0000'000F) \lll_{32} 14)$ ▷ 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) $\pmod{Prime3}$
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) $\pmod{Prime6}$
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) $\pmod{Prime10}$
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($WordKey, 1, 18$)

$WordKey := \textsc{SwapBits}(WordKey, 2, 27)$ ▷ Green Step 1
$WordKey := \textsc{SwapBits}(WordKey, 5, 28)$
$WordKey := \textsc{SwapBits}(WordKey, 6, 21)$
$WordKey := \textsc{SwapBits}(WordKey, 7, 14)$ ▷ Green Step 2
$WordKey := \textsc{SwapBits}(WordKey, 10, 24)$
$WordKey := \textsc{SwapBits}(WordKey, 11, 25)$
$WordKey := \textsc{SwapBits}(WordKey, 12, 30)$
$WordKey := \textsc{SwapBits}(WordKey, 13, 31)$ ▷ Orange Step
$WordKey := \textsc{SwapBits}(WordKey, 19, 4)$
$WordKey := \textsc{SwapBits}(WordKey, 20, 3)$ ▷ Red Step
$WordKey := \textsc{SwapBits}(WordKey, 17, 2)$
$WordKey := \textsc{SwapBits}(WordKey, 22, 5)$ ▷ Yellow Step
$WordKey := \textsc{SwapBits}(WordKey, 27, 15)$
$WordKey := \textsc{SwapBits}(WordKey, 28, 8)$ ▷ Blue Step
return $WordKey$
end function

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

After this stage is completed, we will not use the initial vector data provided externally. Before the master key data is chunked, it is then chunked into a vector view or real vector whose length has been determined each time by the CommmonStateData class. This is used to represent the chunked data for each of the master keys.

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

This is actually the implementation of the **GenerateSubkeys** function, The outermost wrapper function, which will be the first to use this function, and we will discuss its flow in detail here.

If the size of MainKeyBlockData is empty, then only the update function is executed, otherwise the initialization function is executed first, and then the update function is executed.

Initialization algorithm block: Use a chunk of data from the master key and a complex one-way function to change the matrix.

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

We will show the LatticeCryptographyAndHash function from algorithm in detail next.

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 $element \in \mathbb{F}_2^{64}$
function $\textsc{LatticeCryptographyAndHash}(InputKeys, OutputKeys)$
$SDP = \textbf{ReferenceObject}(CommonStateData.SDP)$
$HashMixedIntegerVector \in \mathbb{F}_2^{64}$

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

$HashMixedIntegerVector := InputKeys$

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

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

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

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($KeyMatrix$, $KeyVector$)

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

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

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

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

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

$SpanVectorA \leftrightarrow \{ResultA_0, ResultA_1, ResultA_2 \dots ResultA_{ResultA.size()-1}\}$
$SpanVectorB \leftrightarrow \{ResultB_0, ResultB_1, ResultB_2 \dots ResultA_{ResultB.size()-1}\}$

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

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

$HashObject =$ **MySpongeStructureHashClass**$(HashBitSize : (KeyRows \times 64) \div 2)$ ▷ The implementation of this class, we have the actual code to refer to in the appendix of this paper.
HashObject.Execute(SpanVectorA, $\{CustomHashed_0 \dots CustomHashed_{KeyRows \div 2}\}$)
HashObject.Execute(SpanVectorB, $\{CustomHashed_{KeyRows \div 2} \dots CustomHashed_{KeyRows \div 2}\}$)
$\mathbb{F}_2^{64} PrimeNumber = 18446744073709551557$
$\mathbb{F}_2^{64} HashedValue = 0$
for $Index = 0$ to $KeyRows - 1$ **do**
$HashedValue = (ResultA_{Index} \pmod{PrimeNumber}) + (ResultB_{Index} \pmod{PrimeNumber}) \pmod{PrimeNumber}$ ▷ (A + B) mod PrimeNumber = ((A mod PrimeNumber) + (B mod PrimeNumber)) mod PrimeNumber
$Hashed_{Index} := (CustomHashed_{Index} \pmod{PrimeNumber}) + (HashedValue \pmod{PrimeNumber}) \pmod{PrimeNumber}$
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.
Ensure that the status matrix and vector is securely cleaned ▷ Fill zero to MA, MB, VA, VB, ResultA, ResultB
return Hashed
end function

We will show the InitializationState function from algorithm in detail next.

This need define a vector of 256 bytes in length to be utilized as an implementation of a non-linear function. This vector contains variable values, similar to the previous byte substitution box, rather than static invariant values. Therefore, it can represent the current state of the byte substitution box.

However, prior to presenting the aforementioned function, we must also establish a dedicated algorithm that can generate new substitution box data from the current byte substitution box data. This algorithm should employ the numbers produced by the pseudo-random number generator and a data structure based on the principle of bitwise operations known as a line-segment tree. The code implementation of this data structure is included in the appendix of this thesis.

And, This function incorporates the ZUC stream cipher algorithm, originally developed by Chinese researchers. However, we have made modifications to the original algorithm, and we will compare and contrast the differences between the two ZUC algorithms at a later stage.

```
//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
  $Nodes = \{0, 0, 0, 0, 0 ... 0_{Size-1}\}$
end function

function Set(Position) ▷ Increment the count at position Position by 1
  for CurrentNode = N ∨ 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 $CurrentLeftSize = \mathrm{N} \gg 1, LeftTotal = 0; CurrentLeftSize \neq 0; CurrentLeftSize := CurrentLeftSize \gg 1$ **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
    end for
    return $CurrentNode \oplus N$
end function

function CLEAR() ▷ Set all counts to 0
    $Nodes := \{0, 0, 0, 0, 0 \ldots 0_{Size-1}\}$
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 $element \in \mathbb{F}_2^8$
function REGENERATIONRANDOMMATERIALSUBSTITUTIONBOX(*OldBox*)
    $NLFSR =$ **ReferenceObject**$(CommonStateData.NLFSR)$
    $LineSegmentTreeObject = LineSegmentTree.Initialize(Size : 256)$
    NewBox $= \{0_0, 0_1, 0_2, 0_3 \ldots 0_{255} | \forall element \in \mathbb{F}_2^8\}$
    $\mathbb{F}_2^{64} Index = 0, Index2 = 0$
    while $Index < OldDataArraySize$ **and** $Index2 < NewDataArraySize$ **do**
        if $Index = OldDataArraySize - 1$ **and** $OldDataBox_{Index} = LineSegmentTreeObject.$GET(0) **then**
            NewBox $:= \{0_0, 0_1, 0_2, 0_3 \ldots 0_{255} | \forall element \in \mathbb{F}_2^8\}$
            LineSegmentTreeObject.CLEAR()
            $Index := 0, Index2 := 0$
        end if
        $\mathbb{F}_2^{64} Order =$ NLFSR.GENERATE__CHAOTIC__NUMBER(8) $\pmod{OldBox.size() - Index}$
        $\mathbb{F}_2^{64} Position =$ LINESEGMENTTREEOBJECT.GET(*Order*)
        while $OldBox_{Index} = Position$ **do**
            $Order :=$ NLFSR.GENERATE__CHAOTIC__NUMBER(8) $\pmod{OldBox.size() - Index}$
            $Position :=$ LINESEGMENTTREEOBJECT.GET(*Order*)
        end while
        $NewBox_{Index2} = Position$
        LINESEGMENTTREEOBJECT.SET(*Position*)
        Index := Index + 1, Index2 := Index2 + 1
    end while
    return NewBox
end function

We previously mentioned that we modified the ZUC stream cipher algorithm. However, let us first present the original algorithm and then discuss the parts we modified. It is worth noting that the modified ZUC stream cipher algorithm uses the 2 dynamic byte substitution boxes mentioned earlier, while the differences in the internal register initialization functions are significant.

```
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. ▷ $\mathbf{WT_1}(Word1, Word2) = (\mathbf{Low16BitOnly}(Word1) \ll_{32} 16) \vee_{32} (\mathbf{High16BitOnly}(Word2) \gg_{32} 16)$ ▷ $\mathbf{WT_2}(Word1, Word2) = (\mathbf{Low16BitOnly}(Word2) \ll_{32} 16) \vee_{32} (\mathbf{High16BitOnly}(Word1) \gg_{32} 16)$ ▷ $\mathbf{LT_1}(Word) = Word \oplus_{32} (Word \lll_{32} 2) \oplus_{32} (Word \lll_{32} 10) \oplus_{32} (Word \lll_{32} 18) \oplus_{32} (Word \lll_{32} 24)$ ▷ $\mathbf{LT_2}(Word) = Word \oplus_{32} (Word \lll_{32} 8) \oplus_{32} (Word \lll_{32} 14) \oplus_{32} (Word \lll_{32} 22) \oplus_{32} (Word \lll_{32} 30)$
ApplySubstitutionBox($StateDataRegister_0, StateDataRegister_1$)
return WordData
end function

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

Algorithm 13 The Modified ZUC Sequence/Stream Data cipher

using MaterialSubstitutionBox0
using MaterialSubstitutionBox1

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

Require: LFSR, NLFSR, SDP
Ensure: Two 32-bit Words of State Data Register
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.
$LFSR = \mathbf{ReferenceObject}(CommonStateData.LFSR)$
$NLFSR = \mathbf{ReferenceObject}(CommonStateData.NLFSR)$
$SDP = \mathbf{ReferenceObject}(CommonStateData.SDP)$
$StateValue0 \leftrightarrow StateDataRegister_0$
$StateValue1 \leftrightarrow StateDataRegister_1$
$\mathbb{F}_2^{64} BaseNumber =$ NLFSR.generate_chaotic_number(8) $\oplus_{64}$ SDP($min : 0, max : 18446744073709551615$)
$\mathbb{F}_2^{64} RandomNumber = 0$
for Round = 129 **to** 1, Round := Round - 1 **do**
$BaseNumber :=$ NLFSR.unpredictable_bits($BaseNumber \pmod{18446744073709551615}, 64$) $\oplus_{64}$ LFSR()
end for
$RandomNumber :=$ NLFSR.generate_chaotic_number(8) $\oplus_{64}$ ($\neg_{64}$(LFSR.generate_bits(63) $\oplus_{64}$ $BaseNumber$))
$StateValue0 := \mathbf{Hight32BitOnly}(RandomNumber)$
$StateValue1 := \mathbf{Low32BitOnly}(RandomNumber)$
$RandomNumber := 0$
end function

function ApplySubstitutionBox($RegisterValue0$, $RegisterValue1$) ▷ Register data using non-linear data for byte substitution operation
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
end function

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

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

Having ensured that all necessary preparations have been made, we are now able to proceed with build the InitializationState function.

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**($CommonStateData.LFSR$)
$ByteKeys = \{\varnothing | \forall element \in \mathbb{F}_2^8\}$
$ByteKeys$:= 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
$Word32Bit\_Key = \{\varnothing | \forall element \in \mathbb{F}_2^{32}\}$
$Word32Bit\_Key$:= 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).
$Word32Bit\_Random = \{0, 0, 0, 0, \dots | \forall element \in \mathbb{F}_2^{32}, \forall element = 0\}$ ▷ Size is Word32Bit_ExpandedKey.size() ÷ 4
$Index = 0, OffsetIndex = 0$
while OffsetIndex + 4 < Word32Bit_ExpandedKeySpan.size() and Index < Word32Bit_Random.size() **do**
$Word32Bit\_ExpandedKeySubSpan = \{Word32Bit\_ExpandedKeySpan_{OffsetIndex} \dots Word32Bit\_ExpandedKeySpan_{OffsetIndex+3}\}$
▷ Subspan is (sub-span range/sub-collection) range of Word32Bit_ExpandedKey, elements size is 4
OffsetIndex := OffsetIndex + 4, Index := Index + 1
$\mathbb{F}_2^{32} RandomWord$ = ModifiedZUC.GenerateKeyStream($Word32Bit\_ExpandedKeySubSpan$)$\oplus_{32} Word32Bit\_ExpandedKeySubSpan_3$
$Word32Bit\_Random_{Index} := RandomWord$
$RandomWord := 0$
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
$Word64Bit\_ProcessedKey = \{\varnothing | \forall element \in \mathbb{F}_2^{64}\}$
$Word64Bit\_ProcessedKey = IntegerFromBytes(ByteKeys)$
ByteKeys all element reset to 0
$\mathbb{F}_2^1$ Word64Bit_KeyUsed = **false**
$RandomBitsArray = \{0, 0, 0, 0, 0, \dots | \forall element \in \mathbb{F}_2^1, \forall element = 0\}$ ▷ 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**

$\mathbb{F}_2^{64}$ 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($LFSR$) $\oplus_{64}$ LFSR.GENERATE_BITS(63)

$RandomBit := RandomNumber \wedge_{64} 1$

end for

for BitIndex = 0, BitIndex **to** RandomBitsArray.size() - 1, BitIndex := BitIndex + 1 **do**

if $RandomBitsArray_{BitIndex} = 1$ **then**

$RandomNumber := RandomNumber \vee_{64} (RandomBitsArray_{BitIndex} \lll_{64} BitIndex)$

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: Mix MatrixA and MatrixB then shuffle indices (Key confusion layer).

$$MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)$$

$$MatrixB = \textbf{ReferenceObject}(CommonStateData.TransformedSubkeyMatrix)$$

$$MatrixA_{\{Row,Column\}},\ MatrixB_{\{Row,Column\}} \in \mathbb{F}_2^{64}$$

$$CommonStateData \xrightarrow[SubkeyMatrixOperationObject.\textbf{UpdateState}()]{MatrixA,\ MatrixB} CommonStateData'$$

We will show the UpdateState function from algorithm in detail next.

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

$NLFSR = \textbf{ReferenceObject}(CommonStateData.NLFSR)$

$SDP = \textbf{ReferenceObject}(CommonStateData.SDP)$

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

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

$\mathbb{F}_2^{64} BaseNumber = 0$ ▷ 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$(BaseNumber \wedge_{64} 1, 64)$
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$(BaseNumber \wedge_{64} 1, 64)$
BaseNumber := BaseNumber + 1
end for
end for
$BaseNumber := 0$
$LeftMatrix = (RandomQuadWordMatrix.rowwise() \times_{\text{VEW}} RandomVector)$
$LeftMatrix := LeftMatrix.columnwise() +_{VECTOR} RandomVector2$
$RightMatrix = (RandomQuadWordMatrix.columnwise() \times_{\text{VEW}} RandomVector2)$
$RightMatrix := RightMatrix.rowwise() -_{VECTOR} RandomVector$ ▷ 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)

This is actually the implementation of the **GenerateRoundSubkeys** function, The outermost wrapper function, which will be the first to use this function, and we will discuss its flow in detail here.

$$MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)$$

$$MatrixB = \textbf{ReferenceObject}(CommonStateData.TransformedSubkeyMatrix)$$

$$MatrixC = \textbf{ReferenceObject}(RoundSubkeyGeneratationModuleObject.GeneratedMatrix)$$

$$RoundSubkeyGeneratationModuleObject.Matrix_{\{Row,Column\}} \in \mathbb{F}_2^{64}$$

$$MatrixA, MatrixB \xrightarrow[RoundSubkeyGeneratationModuleObject.\textbf{GenerationRoundSubkeys}()]{} MatrixC$$

We will show the RoundSubkeyGeneratationModule class from algorithm in detail next.

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

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

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

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

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

Require: RandomQuadWordMatrix, TransformedSubkeyMatrix

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

function OPC_MatrixTransformation ▷ OaldresPuzzle_Cryptic - Unpredictable matrix transformation

    $MatrixA = \textbf{ReferenceObject}(CommonStateData.RandomQuadWordMatrix)$

    $MartixB = \textbf{ReferenceObject}(CommonStateData.TransformedSubkeyMatrix)$

    $MartixC = \textbf{ReferenceObject}(GeneratedMatrix)$

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

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

    $Temporary0 \rightarrow MartixB^{Transpose}$

    $Temporary1 \rightarrow MatrixA^{Transpose}$

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

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

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

    $TemporaryIntegerMatrix := Temporary4^{HermitianTranspose}$

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

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

▷ Ensure that the status matrix is securely cleaned

end function

Require: GenerateMatrix

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

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

    if AlgorithmCounter $= 0$ **then**

        GeneratedMatrix, GeneratedVector all element reset to 0

    end if

    OPC_MatrixTransformation()

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

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

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

end while ▷ Key whitening

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

$NewRoundSubkeyVectorSpan \leftrightarrow \{TransformedVector_0 \ldots TransformedVector_{TransformedVector.size()-1}\}$ ▷ 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).

$RoundSubkeyVectorSpan \leftrightarrow \{GeneratedVector_0 \ldots GeneratedVector_{GeneratedVector.size()-1}\}$ ▷ 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end for ▷ Bits data diffusion layer - Data avalanche effect for diffusion

▷ 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.)

GeneratedVector := TransformedVector

AlgorithmCounter := AlgorithmCounter + 1

end function

We have fully explained all the modules involved in the **GenerateSubkeys** and **GenerateRoundSubkeys** mathematical abstraction functions.

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 $WordData$

using CommonStateData.MatrixOffsetWithRandomIndices
using SecureRoundSubkeyGeneratationModule.GeneratedRoundSubkeyMatrix

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

function SRSGM.**ForwardTransform**($LeftWordData$, $RightWordData$) ▷ The H-function encode described by Lai–Massey Scheme
  $LeftWordData' = LeftWordData \boxplus_{32} RightWordData$
  $RightWordData' = LeftWordData \boxplus_{32} 2 \boxtimes_{32} RightWordData$
  $RightWordData' := RightWordData' \oplus_{32} (LeftWordData' \lll_{32} 1)$
  $LeftWordData' := LeftWordData' \oplus_{32} (RightWordData' \ggg_{32} 63)$
  return $\{LeftWordData', RightWordData'\}$
end function

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

function SRSGM.**CrazyTransformAssociatedWord**($AssociatedWordData$, $WordKeyMaterial$) ▷ The F-function described by Lai-Massey Scheme
  BitReorganizationWord $\in \mathbb{F}_2^{32}$
  BitReorganizationWord $= \{0, 0\}$
  $WordA \longleftrightarrow BitReorganizationWord_0$
  $WordB \longleftrightarrow BitReorganizationWord_1$
  $\{LeftWordKey, RightWordKey\} = \textbf{Split}(WordKeyMaterial)$
  ▷ LeftWordKey and RightWordKey are constant 32-bits word, $LeftWordKey, RightWordKey \in \mathbb{F}_2^{32}$
  $\mathbb{F}_2^{64} PseudoRandomValue = ((WordKeyMaterial \oplus_{64} AssociatedWordData) \ll_{64} 32) \vee_{64} ((WordKeyMaterial \odot_{64} AssociatedWordData) \gg_{64} 32)$
  $\mathbb{F}_2^{32} WordC = PseudoRandomValue \ll_{64} (WordKeyMaterial \pmod{64}) \gg_{64} 32$
  $\mathbb{F}_2^{32} WordD = PseudoRandomValue \gg_{64} (WordKeyMaterial \pmod{64})$
  $WordC := (AssociatedWordData \vee_{32} LeftWordKey) \wedge_{32} WordC$
  $WordD := (AssociatedWordData \wedge_{32} RightWordKey) \vee_{32} WordD$
  $WordA := WordA \oplus_{32} WordC$
  $WordB := WordB \oplus_{32} WordD$
  $WordB := (WordA \boxplus_{32} LeftWordKey) \lll_{32} (PseudoRandomValue \pmod{32})$
  $WordA := (WordB \boxplus_{32} RightWordKey) \ggg_{32} (PseudoRandomValue \pmod{32})$
  $WordC := (WordB \wedge_{32} LeftWordKey) \oplus_{32} (WordD \vee_{32} AssociatedWordData)$
  $WordD := (WordA \wedge_{32} RightWordKey) \oplus_{32} (WordC \vee_{32} AssociatedWordData)$
  $WordA := WordA \oplus_{32} WordC$
  $WordB := WordB \oplus_{32} WordD$
  $MatrixRow = MatrixOffsetWithRandomIndices_{WordA \pmod{MatrixOffsetWithRandomIndices.size()}}$
  $MatrixColumn = MatrixOffsetWithRandomIndices_{WordB \pmod{MatrixOffsetWithRandomIndices.size()}}$
  ▷ MatrixRow and MatrixColumn are constant 32-bits word
  $\mathbb{F}_2^{32} ShiftAmount = WordA \boxplus_{32} WordB$
  $\mathbb{F}_2^{32} ShiftAmount2 = WordA \boxplus_{32} WordB \boxtimes_{32} 2$
  $\mathbb{F}_2^{32} RotateAmount = MatrixColumn \boxminus_{32} MatrixRow$
  $\mathbb{F}_2^{32} RotateAmount2 = 2 \boxtimes_{32} MatrixRow \boxminus_{32} MatrixColumn$
  $RoundSubkey \in \mathbb{F}_2^{64}$
  $RoundSubkey = GeneratedRoundSubkeyMatrix_{\{MatrixRow, MatrixColumn\}}$
  $\mathbb{F}_2^{64} Bit = (RoundSubkey \gg_{64} ShiftAmount \pmod{64}) \wedge_{64} 1$
  $\mathbb{F}_2^{64} Bit2 = (RoundSubkey \gg_{64} ShiftAmount2 \pmod{64}) \wedge_{64} 1$
  $\mathbb{F}_2^{64} LeftRotatedMask = Bit \lll_{64} RotateAmount \pmod{64}$
  $\mathbb{F}_2^{64} RightRotatedMask = Bit2 \ggg_{64} RotateAmount2 \pmod{64}$
  $\mathbb{F}_2^{64} BitMask = LeftRotatedMask \oplus_{64} RightRotatedMask \pmod{64}$

```
    if BitMask = 0 then
        BitMask := BitMask ∨64 (1 ≪64 ((MatrixRow ⊞32 MatrixColumn) ⊠64 2) (mod 64))
    end if
    RoundSubkey := RoundSubkey ∧64 (¬64 BitMask)
    {aa, bb} := Split(RoundSubkey)
    WordA := WordA ⊕32 aa
    WordB := WordB ⊕32 bb
    AssociatedWordData := AssociatedWordData ⊕32 (WordA ⊕32 WordB)
    return AssociatedWordData
end function
```

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

Thank you for reading this papers, Now we just need to follow the previous architecture and the provided algorithm to implement the algorithmic framework of the OaldresPuzzle_Cryptic. The OPC algorithm is built in the StateDataWorker class, and the pseudo-code is shown below.

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, . . . |∀element ∈ F_2^8, ∀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, . . . |∀element ∈ F_2^8, ∀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
$KeyIndex := RoundSubkeyVector.size()$
end if
end for
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

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

Require: PlainText 64 bits array array and Keys 64 bits array
Ensure: CipherText 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.**SplitDataBlockToEncrypt**($PlainText$, $Keys$)
if PlainText.size() $\pmod{DataBlockSize} \neq 0$ **then**
return
end if
if Keys.size() $\pmod{DataBlockSize} \neq 0$ **then**
return
end if
$Key\_OffsetIndex = 0$
$KeyDataVector := CSD.WordKeyDataVector$ ▷ Is Object Reference
$KeyDataVector_0 \ldots KeyDataVector_{KeyDataVector.size()} := Keys_0 \ldots Keys_{0+KeyDataVector.size()}$
$Key\_OffsetIndex = Key\_OffsetIndex + KeyBlockSize$
$RandomKeyDataVector = \{0, 0, 0, 0, 0, \ldots | \forall element \in \mathbb{F}_2^{64}, \forall element = 0\}$
$RandomKeyDataVector$ 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**
$KeySpan \longleftrightarrow \{Keys_{Key\_OffsetIndex} \ldots Keys_{Key\_OffsetIndex+KeyBlockSize}\}$
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 $\pmod{2048 \times 4}$) == 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
    $WordKeyDataVector_i := (a \ll_{64} 32) \vee_{64} b$
    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.**DecryptingRound**(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() $\pmod{DataBlockSize} \neq 0$ **then**
return
end if
if Keys.size() $\pmod{DataBlockSize} \neq 0$ **then**
return
end if
$Key\_OffsetIndex = 0$
$KeyDataVector := CSD.WordKeyDataVector$ ▷ Is Object Reference
$KeyDataVector_0 \ldots KeyDataVector_{KeyDataVector.size()} := Keys_0 \ldots Keys_{0+KeyDataVector.size()}$
$Key\_OffsetIndex = Key\_OffsetIndex + KeyBlockSize$
$RandomKeyDataVector = \{0, 0, 0, 0, 0, \ldots | \forall element \in \mathbb{F}_2^{64}, \forall element = 0\}$
$RandomKeyDataVector$ size is KeyBlockSize $\times$ 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**
$KeySpan \longleftrightarrow \{Keys_{Key\_OffsetIndex} \ldots Keys_{Key\_OffsetIndex+KeyBlockSize}\}$
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 $\pmod{2048 \times 4}$) == 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
$WordKeyDataVector_i := (a \ll_{64} 32) \vee_{64} b$
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 $\pmod{2048} = 0$ **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 $\pmod{2048 \times 3} = 0$ **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 $\pmod{2048 \times 2} = 0$ **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

Currently, in the field of post-quantum cryptography, the design and research of quantum-resistant cryptography is a hot topic, including the NIST Post-Quantum Cryptography project (source), and various ideas are being explored in the hotspots of the asymmetric field, such as Post-Quantum Cryptography Standardization (source), PQC Algorithm Round 1 Submissions (source), PQC Algorithm Round 2 Submissions (source), PQC Algorithm Round 3 Submissions (source), and PQC Algorithm Round 4 Submissions (source). However, there is little attention paid to the impact of post-quantum cryptography in the field of symmetric cryptography.

The author of this paper is someone who seeks to understand cryptography and its development and design principles. Through online courses and introductory books, the author has studied cryptographic knowledge, including "CRYPTOGRAPHY I" (source) and "A Graduate Course in Applied Cryptography (Dan Boneh and Victor Shoup)" (source). Despite limited education and resources, the author independently designed and implemented a symmetric encryption and decryption algorithm called the OaldresPuzzle$_Cryptic algorithm$.

In addition, the impact of quantum computers on existing symmetric cryptography has been studied by previous researchers and algorithms have been developed for using the computational power of quantum computers for password analysis ([9], [10], [7], [17]).

Furthermore, improved attack algorithms have been proposed for analyzing traditional passwords such as Simon [22] and VQA [26] on platforms running on quantum computers. A review and conclusion ([12], [24]) cited by the author of this paper suggests that quantum computers can have a significant impact on symmetric cryptography. Because quantum computers can represent more information bits than traditional computers, the time complexity of cracking can be reduced to polynomial level. Therefore, it is necessary to develop more secure algorithms for the current form of cryptography. The author of this paper has referred to various types of symmetric encryption and decryption algorithms in the symmetric field, including AES, ARIA, BLOWFISH, TWOFISH, THREEFISH, CAMELLIA, DES (TRIPLE_DES), SERPENT, SM4, IDEA, RC6, CHACHA20, SALEA20, RC4, TRIVIUM, ZUC, and believes that the implementation of these short-key symmetric encryption and decryption algorithms may not be suitable for the future development of cryptography.

The author of this paper also provides a document ([23]) that describes the evolution of bit lengths for symmetric or asymmetric keys used for encryption. They emphasize the potential threat posed by quantum computers to symmetric cryptography and suggest the need for new algorithms to make it more secure and resistant to quantum attacks.

The author of this paper is currently seeking evaluation and feedback on their OPC algorithm from experts and the public. Despite a lack of deep knowledge and connections, they hope to obtain experimental data through sufficient computational power, such as using a quantum computer or a supercomputer to attack their own algorithm, in order to better understand its weaknesses. They believe in the importance of continuous improvement and innovation in cryptography, and although their algorithm sacrifices some speed for security, they see it as a small step towards the future development of cryptography in the age of quantum computing.

It should be noted that the OPC algorithm may not be the fastest, as encrypting or decrypting 10MB of data using a 5KB key on a computer from 2022 takes more than 40 seconds, and on a computer from 2013, it takes more than 70 seconds. The OPC algorithm prioritizes security over speed, and it is expected that its performance will improve over the next 5-10 years with the growth of computer capabilities.

We propose a novel design for a symmetric encryption and decryption algorithm, and invite experts in the field to conduct a comprehensive analysis. The purpose of this research is to evaluate the effectiveness and security of this algorithm, and to present core arguments in support of or against its design. With the development of technology, existing symmetric algorithms may not be sufficient for future use, and our new design aims to address these shortcomings. From a forward-looking perspective, existing symmetric algorithms need improvement to address this issue. We will outline the arguments for and against the OPC algorithm, and attempt to address any potential questions regarding its security and effectiveness.

Supporting arguments for this algorithm:

1. Addresses the deficiencies of existing symmetric algorithms:

Cryptography relies on the use of pseudo-random number generators constructed by abstract, computationally indistinguishable one-way functions. A secure pseudo-random number generator is computationally indistinguishable, meaning it is difficult to predict its output. This is the premise of the Lai-Massey framework, which consists of two abstract functions: the H function (representing a bijective transformation) and the F function (representing an injective transformation). The unpredictability of the F function, achieved by using a secure subkey with each round key, makes it difficult for an opponent to retrieve the original key data in polynomial time. The data being operated on is considered computationally secure because the same subkeys are used for each half of the data being processed and the unpredictability of the F function. For more information about this framework, please refer to the references accompanying the section introducing the Lai-Massey framework.

2. Emphasizes security:

The authors prioritized security in designing this algorithm, sacrificing speed in an attempt to create a stronger and more reliable encryption and decryption method.

This algorithm follows the framework of cryptography, built step by step on the most basic one-way functions of cryptography.

Using their knowledge of cryptography and a basic understanding of mathematical function properties, they designed an algorithm based on an unpredictable pseudo-random number generator and the Lai-Massey symmetric encryption and decryption framework.

This provides a solid foundation for evaluating the security of the algorithm.

Based on the above, as long as the F function designed within the Lai-Massey framework is computationally indistinguishable, the entire framework can be considered secure. The use of different keys generated by the OPC algorithm for each application of the F function in each round results in the unpredictability of the data being processed, making the encryption and decryption process secure. Even if the encrypted data and random data are computationally distinguishable, it is impossible to distinguish between them in polynomial time.

Counterarguments against the algorithm:

1. Lack of testing:

The authors did not have the opportunity to test the algorithm using a quantum computer or supercomputer, which limits their ability to comprehensively evaluate its security.

There is also no clear evidence of any significant improvement over existing symmetric encryption and decryption algorithms. This raises questions about the necessity of this new algorithm and its potential impact on the field of cryptography.

2. Insufficient knowledge and experience:

The authors have limited knowledge and experience in cryptography, which raises concerns about the effectiveness of the algorithm. They hope that professionals in the field can provide assistance.

Justification for Request:

The authors of this paper are seeking help from experts in the field of cryptography to evaluate a new independently designed symmetric encryption and decryption algorithm. Although their resources and network are limited, the OPC algorithm proposed in this paper is a step towards incremental innovation and a need to keep up with the rapid development of technology.

The authors recognize that their knowledge and experience in cryptography may not be as extensive as others in the field, and they hope that the open evaluation of their algorithm will help them better understand its strengths and limitations. The authors request that professionals in the field of cryptography take the time to evaluate their algorithm and provide feedback on its effectiveness.

This feedback may include suggestions for improvement, a rigorous assessment of its security and speed, or any other relevant information that can help the authors better understand the strengths and weaknesses of their algorithm.

Testing on traditional computers with limited computing power using 10MB of data and a 5KB master key as input and outputting modified 10M data would be appropriate. Testing on supercomputers or quantum computers can attempt larger data and longer keys, making it easier to find the algorithm's weaknesses and security.

They are seeking support and resources to conduct this type of testing, which they believe is crucial for a comprehensive evaluation of their algorithm's security and for promoting the growth and development of cryptography.

In summary, the authors are seeking support and guidance from the cryptography community to help them further their understanding of cryptography and their place in it. They hope that their work is an important step towards developing more secure and efficient symmetric encryption and decryption algorithms and that they receive the necessary support and guidance to achieve their goals.

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 Jaković, Bariş 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

The following knowledge points, technologies and contents will be used in this paper. Readers are asked to find references, books and materials after understanding the relevant concepts by themselves. Start reading this paper.

Algorithms and Data Structures: An algorithm is a set of instructions or a step-by-step procedure used to solve a problem or accomplish a task. In the context of cryptography, an algorithm is a mathematical procedure used to encrypt and decrypt data. Different encryption algorithms use different techniques, such as substitution, transposition, and modular arithmetic, to convert plaintext into ciphertext and vice versa.Data structures are used to organize and store data in a specific way, making it easy to access, modify, and process the data. In the context of cryptography, data structures can be used to store encryption keys, intermediate values, and other data used in the encryption and decryption process. Examples of data structures used in cryptography include arrays, linked lists, and trees.The OaldresPuzzle_Cryptic algorithm utilizes various data structures and algorithms to create a unique encryption-decryption key that is virtually unbreakable. It uses a line tree data structure and dynamically generated byte substitution boxes to make each generated key unpredictable. It also makes use of various mathematical operations and algorithms, including linear algebra such as affine transformations, Kronecker product, dot product, solving transpose and adjoint matrices, and addition, subtraction, and multiplication of matrices. These subkey generation modules are coordinated and designed by the Lai-Massey program.

Ciphers: A cipher is a mathematical algorithm that is used to encrypt and decrypt data. Ciphers are used to convert plaintext into ciphertext, and ciphertext back into plaintext. They are used to ensure the confidentiality and integrity of data, so that only authorized parties can read and understand the original message.

Plaintext and Ciphertext: Plaintext is the original unencrypted message that is to be transmitted or stored. It can be any form of data, such as text, images, or audio. Ciphertext is the result of encrypting plaintext using a specific encryption algorithm and key. Ciphertext is a scrambled version of the plaintext, which is difficult or impossible to read or understand without the corresponding decryption key or algorithm. The main goal of encryption is to convert plaintext into ciphertext in such a way that only authorized parties can read the original message by decrypting it. The transformation of plaintext to ciphertext is called encryption, and the reverse process of transforming ciphertext back to plaintext is called decryption.

The relationship between Ciphers for encryption and decryption: Ciphers are mathematical algorithms that are used to encrypt and decrypt data. The encryption process is the process of converting plaintext into ciphertext using a specific encryption algorithm and key. The decryption process is the reverse process of encryption, it is the process of converting ciphertext back into plaintext using a specific decryption algorithm and key.The encryption and decryption process are closely related, as encryption is the process of converting plaintext into ciphertext and decryption is the process of converting ciphertext back into plaintext. The encryption process and decryption process are usually performed by different parties, the sender encrypts the message, and the receiver decrypts the message.The encryption and decryption process use the same mathematical algorithm, but the key used for encryption is different from the key used for decryption. For example, in symmetric-key ciphers, the same key is used for encryption and decryption, while in asymmetric-key ciphers, two different keys are used, one for encryption and another one for decryption.The OaldresPuzzle_Cryptic algorithm is a symmetric-key cipher algorithm, it uses a key to encrypt the plaintext and the same key to decrypt the ciphertext.

Keys and Subkeys: In the context of encryption, a key is a value or a set of values that are used to encrypt and decrypt data. The key is used in the encryption algorithm to transform plaintext into ciphertext and vice versa. The key is a critical element of the encryption process, as it determines the level of security of the encryption.Subkeys, also called round keys, are derived from a secret key, usually through a key schedule algorithm. They are used in many encryption algorithms, particularly those that use multiple rounds of encryption. Subkeys are used to encrypt the data at different stages of the encryption process, adding more security to the encryption algorithm by making it more difficult for an attacker to determine the key used to encrypt the data.A subkey is a derived key, which is used to encrypt data in a specific round of the encryption algorithm. The subkey is usually derived from the main key through a key schedule algorithm, which is used to generate a new key for each round of encryption. The subkeys are generated in a way that is different from the main key, making it more difficult for an attacker to determine the main key used to encrypt the data.

Symmetric-key ciphers: Symmetric-key ciphers use the same key for encryption and decryption. Examples of symmetric-key ciphers include AES, DES, and Blowfish.

Asymmetric-key ciphers: Asymmetric-key ciphers use two different keys, one for encryption and one for decryption. Examples of asymmetric-key ciphers include RSA and Elliptic Curve Cryptography (ECC).

Block ciphers: A block cipher is a type of symmetric key cipher that encrypts a fixed-size block of data at a time, rather than a stream of data. Block ciphers are widely used in various applications, including file encryption and secure communications.They are suitable for encrypting large amounts of data, such as files or disk partitions. Examples of block ciphers include AES [5] and DES, RC6[20].

Stream ciphers: A stream cipher is a type of symmetric key cipher that encrypts a stream of data one bit or byte at a time. Stream ciphers are used in various applications, including wireless communications and mobile networks.They are suitable for encrypting real-time data streams, such as audio or video. Examples of stream ciphers include RC4 and Salsa20.

Key size: Key size is a measure of the number of bits in an encryption key. Key size is an important factor in determining the security of an encryption algorithm, as larger key sizes can make it more difficult to break the encryption through brute force attacks.

Chaos theory: Chaos theory is a branch of mathematics that studies the behavior of dynamic systems that are highly sensitive to initial conditions, also known as the butterfly effect. Chaos theory has been applied in various fields, including cryptography, where it can be used to generate pseudo-random numbers that are difficult to predict.

Cryptography based on chaos theory: is a new field of research that exploits the properties of chaotic systems to generate secure keys for encryption.

Cryptography based on lattices: Lattice cryptography is a form of cryptography that is based on the mathematical properties of lattices, which are discrete sets of points in a multidimensional space. Lattice cryptography is considered as a promising post-quantum cryptography method, meaning that it is believed to be secure against quantum computer attacks.In lattice cryptography, the encryption and decryption process are based on the operations of the lattice, such as the shortest vector problem (SVP) and the closest vector problem (CVP). These problems are hard to solve for a quantum computer, which makes lattice-based encryption schemes resistant to quantum attacks.One example of lattice-based cryptography is the Learning with Errors (LWE) problem, which is the foundation of many encryption schemes such as NTRU and Ring-LWE. The LWE problem is based on the difficulty of solving a system of linear equations over a lattice, where the equations are chosen at random.Another example is the Ring-LWE, a lattice-based encryption scheme that is based on the difficulty of solving a variant of the LWE problem over a polynomial ring. [4] [2]

Ajtai's hash function: Ajtai's hash function is a cryptographic hash function proposed by Miklos Ajtai in 1996. It is a one-way function that takes an input of arbitrary length and produces a fixed-length output, called a hash or digest. The output is designed to be unique, meaning that even small changes to the input will result in a completely different output.Ajtai's hash function is based on the concept of a collision-resistant function, which means that it is computationally infeasible to find two inputs that produce the same output. It is also designed to be preimage-resistant, meaning that it is computationally infeasible to find an input that produces a specific output. [6]

Lai-Massey Scheme: The Lai-Massey sheme is a technique used to design cryptographic systems, it is a powerful tool for designing cryptographic systems and analyze their security.

Linear algebra: Linear algebra is the branch of mathematics that deals with vector spaces and linear transformations. Linear algebra is used in many areas of mathematics and science, including cryptography, computer graphics, and machine learning.Linear algebra operations can be used to manipulate matrices and vectors in various ways, such as solving systems of linear equations, calculating determinants and eigenvalues, and performing matrix multiplication and inversion.

Affine transformations: An affine transformation is a type of transformation in linear algebra that preserves collinearity (i.e. the fact that points that are on the same line remain on the same line) and ratios of distances (i.e. the fact that the ratio of distances between points is preserved). Affine transformations are defined by a matrix and a vector, and can include operations such as translation, rotation, scaling, and shearing.

Kronecker product: The Kronecker product, also known as the tensor product, is a binary operation on matrices that produces a new matrix by taking the outer product of each element of one matrix with each element of the other matrix. The Kronecker product can be represented using the symbol $\otimes$ and it is defined as:$C = A \otimes B = [a_{i,j}B]$ (We do not represent it this way in this paper because there are too few mathematical graphical symbols and it is easy to create ambiguity.)where A is an m $\times$ n matrix, B is a p $\times$ q matrix, and C is an mp $\times$ nq matrix. The Kronecker product of A and B is formed by taking the matrix B and replicating it m $\times$ p times along the rows and n $\times$ q times along the columns, and then element-wise multiplying the result with the elements of matrix A.The Kronecker product is a powerful operation that can be used to model a wide range of mathematical and physical systems, including linear systems, nonlinear systems, and signal processing systems.

Dot product: The dot product, also known as the scalar product, is a type of operation between two vectors that results in a scalar value. The dot product of two vectors is calculated by multiplying the corresponding entries and then summing the results. The dot product of two vectors can be used to determine the angle between them and can be used in various mathematical operations.

Transpose and adjoint matrices: The transpose of a matrix is a new matrix that is formed by flipping the original matrix about its main diagonal. The adjoint of a matrix is the conjugate transpose of the matrix. These operations can be used to transform the matrix into a different form that may be easier to work with for a specific operation.

Group theory: Group theory is a branch of mathematics that deals with the study of groups, which are sets of elements with a specific operation that satisfies certain properties. Group theory is used in many areas of mathematics and science, including cryptography.In cryptography, group theory is used in the design and analysis of various types of encryption algorithms, such as symmetric-key ciphers and public-key ciphers. For example, the security of many symmetric-key ciphers is based on the difficulty of solving a Certain mathematical problems that belong to a specific group.

Finite Field: A finite field, also known as Galois field, is a mathematical structure that consists of a finite number of elements and a set of mathematical operations that can be performed on those elements. Finite fields are used in many areas of mathematics, including number theory, coding theory and cryptography.

Information theory: Information theory is the branch of mathematics that deals with the representation, transmission, processing, and interpretation of information. It is closely related to cryptography, as it deals with the concepts of entropy and information entropy which are important in the analysis of encryption algorithms.

Shift Register: A shift register is a digital circuit that can be used to store and manipulate multiple bits of data. Shift registers are often used in digital circuits and in cryptography as a simple and efficient way to generate a sequence of pseudo-random numbers.

Feedback shift register (FSR): A feedback shift register is a type of shift register that has a feedback loop. The output of the last stage is fed back as input to the first stage. FSRs are often used to generate pseudo-random numbers or pseudo-random bit sequences.

Linear feedback shift register (LFSR): A linear feedback shift register is a shift register that has a linear feedback function. LFSRs are often used in digital circuits and in cryptography as a simple and efficient way to generate a sequence of pseudo-random numbers.

Linear systems and Nonlinear systems: Linear systems and nonlinear systems are two types of mathematical systems that describe the behavior of different physical and mathematical phenomena.Linear systems are systems that follow linear equations, which are equations that have the property that the sum of two solutions is also a solution, and that the product of a solution by a scalar is also a solution. Linear systems have a simple mathematical structure and they are relatively easy to analyze and control.Examples of linear systems include linear differential equations, linear differential-algebraic equations, linear difference equations, and linear algebraic equations.On the other hand, nonlinear systems are systems that follow nonlinear equations, which are equations that do not have the properties of linear equations. Nonlinear systems have a more complex mathematical structure and they are more difficult to control.That cannot be modeled or analyzed using linear mathematics. Examples of nonlinear systems include nonlinear differential equations, nonlinear differential-algebraic equations, nonlinear difference equations, and nonlinear algebraic equations. They exhibit complex behavior and are often characterized by multiple equilibrium points and the existence of limit cycles.In cryptography, linear systems can be vulnerable to linear cryptanalysis and algebraic attacks, while nonlinear systems are more resistant to these types of attacks.The OaldresPuzzle_Cryptic algorithm utilizes a nonlinear feedback shift register with chaotic properties, a static byte substitution box to simulate nonlinear strong functions, and a dynamic byte substitution box. Furthermore, it makes use of various mathematical operations including linear algebra such as affine transformations, Kronecker product, dot product, solving transpose and adjoint matrices, and addition, subtraction, and multiplication of matrices. These design choices make the OaldresPuzzle_Cryptic algorithm a nonlinear system, which is more resistant to linear and algebraic attacks.

Nonlinear feedback shift register (NLFSR): A nonlinear feedback shift register (NLFSR) is a type of shift register that has a nonlinear feedback function. Unlike linear feedback shift registers (LFSRs) which have a linear feedback function, NLFSRs use a nonlinear function to generate the next bit in the sequence. NLFSRs can generate more complex and less predictable sequences of bits, making them more difficult to predict and more suitable for use in cryptographic applications. They can be designed to exhibit chaotic behavior, making them more suitable for use in chaos-based cryptography. The algorithm also utilizes a linear feedback shift register with a sequence period length of 2 to the 128th power. The combination of LFSR and NLFSR creates a more robust and unpredictable sequence of bits that can be used as a key for encryption. [25] [4] [18]

Pseudo-random number generators (PRNGs): Pseudo-random number generators are algorithms that produce sequences of numbers that are statistically similar to sequences of truly random numbers. PRNGs are often used in cryptography to generate encryption keys.

Pseudorandomness: A pseudorandom sequence of numbers is one that appears to be random, but is generated by a deterministic process. Pseudorandom numbers are widely used in cryptography, where they are used to generate encryption keys.

Byte substitution box (S-box): A byte substitution box is a component of many encryption algorithms that maps input values to output values using a fixed table. S-boxes are often used to provide diffusion and confusion in encryption algorithms, by making it difficult for an attacker to determine the relationship between the plaintext and the ciphertext. [8] [16]

Confusion and diffusion: Confusion and diffusion are two important properties of encryption algorithms. Confusion refers to the property that the relationship between the plaintext and the ciphertext is complex and difficult to determine, while diffusion refers to the property that small changes in the plaintext result in large changes in the ciphertext.

Key schedule: A key schedule is an algorithm that is used to expand a short encryption key into a longer key for use in a block cipher. The key schedule is an important component of a block cipher, as it can affect the security of the cipher.

ZUC sequence cipher design: ZUC is a stream cipher used in wireless communications and mobile networks and created by chinese for commercial. It is based on a sequence generator that produces a key stream by using operations such as non-linear operations, bitwise operations, and modular addition. [14]

Line segment tree data structure: Line segment tree is a data structure that is used to represent a sequence of elements. It is a generalization of the prefix tree. It is a tree data structure that can be used to represent a sequence of elements, usually characters or words.

Evaluation and testing of encryption-decryption algorithms: To evaluate the security and effectiveness of an encryption-decryption algorithm, it is important to test it using various parameters such as key size, encryption-decryption time, and resistance to various known attacks, including quantum computing attacks.

Encryption-decryption time: Encryption-decryption time is a measure of how long it takes to encrypt or decrypt a message using a particular encryption algorithm. This is an important factor to consider when choosing an encryption algorithm, as a faster encryption-decryption time can be more practical for some applications.

Cryptographic secureness: A property of a cryptographic system that ensures that it is computationally infeasible for an attacker to recover the plaintext from the ciphertext or the key used, without possessing some secret information, such as the key.

Methods of attacking ciphers: There are several methods that can be used to attack a cipher and try to recover the plaintext or key used in the encryption process. Some common methods include: Brute force attack: A brute force attack is a type of attack in which an attacker tries all possible keys until the correct one is found. This method is highly time-consuming, but it can be effective if the key space is small.Known plaintext attack: A known plaintext attack is a type of attack in which an attacker has access to both the ciphertext and the corresponding plaintext. The attacker uses this information to try to determine the key used in the encryption process.Chosen plaintext attack: A chosen plaintext attack is a type of attack in which an attacker can choose the plaintext that is to be encrypted and then tries to determine the key used in the encryption process.Differential cryptanalysis: A differential cryptanalysis is a type of attack that uses the difference between two plaintexts and their corresponding ciphertexts to try to determine the key used in the encryption process.Linear cryptanalysis: A linear cryptanalysis is a type of attack that uses linear approximations of the encryption function to try to determine the key used in the encryption process.Algebraic attacks: Algebraic attacks are a type of attack on encryption algorithms that exploit the mathematical structure of the algorithm. These attacks can include techniques such as linear and differential cryptanalysis, and algebraic attacks on the block key schedule of a block.Quantum attacks: Quantum computers have the ability to solve certain problems much faster than classical computers, which poses a threat to classical encryption algorithms. Quantum attacks include Shor's algorithm, Grover's algorithm and others.Side-channel attacks: Side-channel attacks are a type of attack that exploit information leaked from the physical implementation of a cryptographic system, such as timing information, power consumption, or electromagnetic emissions. These attacks can allow an attacker to extract the key used in the encryption process.Social engineering attacks: Social engineering attacks are a type of attack that use psychological manipulation to trick users into revealing sensitive information, such as encryption keys or passwords.Dictionary attacks: Dictionary attacks are a type of attack in which an attacker uses a pre-computed dictionary of commonly used words, phrases, and patterns in attempts to find the encryption key.It's important to note that the security of a cipher is determined not only by the strength of the encryption algorithm itself, but also by the strength of the key used in the encryption process, the implementation of the algorithm, and the security of the entire system in which the algorithm is used.

Resistance to quantum computing attacks: As quantum computers have the ability to solve certain problems much faster than classical computers, it is important to design encryption algorithms that are resistant to quantum computing attacks. This can be done by using mathematical operations that are difficult to solve on a quantum computer By using encryption keys that are large enough to make brute force attacks infeasible. [2] [19]

B.1 Definition of necessary concepts and mathematical symbols

To aid in the interpretation of our algorithmic process, we need to define the following basic concepts:

1. Bytes and bits

Let "values" be a one-dimensional set of finite elements with a size of 9, where each element can only be 0 or 1. The leftmost element in the "values" set is the most significant bit, and the rightmost element is the least significant bit. When an element exceeds 1, it becomes 0, and the value 1 is carried over to the next bit in the higher position (and so on). The binary representation is based on the numbers 0 and 1, which establish a one-to-one correspondence, and each of these basic representation units is called a bit. The "values" set comprises 8 bits in binary representation, forming one byte. This representation method can express numbers from 0 to $2^8 - 1$.

$$\begin{aligned}
&values = \{0,0,0,0,0,0,0,0\} \quad or \quad values = \{1,1,1,1,1,1,1,1\} \\
&values \in \{0,1\} \quad and \quad (\text{values size} < 9) \\
&\text{Example:} \\
&\{0,0,0,0,0,0,1,0\} = values = \{0,0,0,0,0,0,0,1\} + \{0,0,0,0,0,0,0,1\} \\
&\{0,0,0,0,0,0,0,0\} = values = \{0,0,0,0,0,0,0,1\} - \{0,0,0,0,0,0,0,1\}
\end{aligned}$$

2. Byte and Bit

Let values be a one-dimensional set of finite elements of size 9, where each element can only be 0 or 1. The leftmost element of the values set represents the highest bit, while the rightmost element represents the lowest bit. When any of the elements exceeds 1, it needs to be converted to 0 and 1 is added to the next higher bit (and so on). The numeric elements 0 and 1 in the values set establish a one-to-one relationship, which is the binary representation. 0 and 1 are the most basic representation units, known as bits. The values set consists of eight bits represented in binary, and then combined into one byte. This form of expression can represent numbers from 0 to $2^8 - 1$.

3. Hexadecimal

In mathematics and computing, the hexadecimal (also called base 16 or simply hex) numeral system is a positional numeral system that uses a radix (base) of 16 to represent numbers.

Unlike the decimal system that uses ten symbols to represent numbers, hexadecimal uses 16 different symbols, with the most common being "0"-"9" to represent values 0 to 9, and "A"-"F" (or alternately "a"-"f") to represent values 10 to 15.

Hexadecimal numbers are widely used by software developers and system designers because they provide a human-readable representation of binary-encoded values. Each hexadecimal digit represents four bits (binary digits), also known as a nybble.

For example, a binary value ranging from 00000000_2 to 11111111_2 in an 8-bit byte can be conveniently represented as the hexadecimal range of 00_{16} to FF_{16}.

In mathematics, subscripts are often used to specify the radix. For example, the decimal value 43838 would be represented in hexadecimal as $AB3E_{16}$.

In computer programming languages, many symbols are used to represent hexadecimal digits, often involving prefixes.

The prefix "0x" is widely used in C/C++ programming languages to indicate hexadecimal values, where 0xAB3E indicates a value of 43838.

Note:

The above definition of subscripts, which is used to distinguish what base the value is in, may cause confusion with the operators we are about to define. Therefore, we will primarily use the prefix method to indicate what base system the value is in.

The prefix "0b" indicates binary, and 0b1010110001011001 indicates the decimal number 44121.

The prefix "0x" indicates hexadecimal, and 0x123456 indicates the decimal number 1193046.

To aid in the interpretation of our algorithmic process, we need to define the following notations:

These operations operate on operands a, b, c. Their sizes are either 1 byte, 2 bytes, or 4 bytes (32-bit unit size, which can be 8, 16, 32, or 64)

$left \pmod{right}$ represents the modulo operation, which computes the remainder when left is divided by right, and $right > 0$. For example, $255 \equiv 4 \pmod{251} 4$ is the remainder.
$c = a \boxplus_{32} b$ represents addition with modulo. $c = a + b \pmod{2^{32}}$
$c = a \boxminus_{32} b$ represents subtraction with modulo. $c = a - b \pmod{2^{32}}$
$c = a \boxtimes_{32} b$ represents multiplication with modulo. $c = a \times b \pmod{2^{32}}$

We define the **bitwise operations** in binary representation.

The suffix SN in the following expressions refers to signed number types (which can be positive or negative), where the highest bit is used to represent the sign. If the value is 1, it is negative; otherwise, it is positive. The suffix USN refers to unsigned number types (which are always positive).

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

The process of bitwise operations involves performing operations on pure bit data sets using one or two bit sets as operands.
Example: Given two bit sets a and b as operands, the result is c. (Condition 1: The two bit sets must be of the same size.)

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

Regardless of whether the types of 'a' and 'b' are SN or USN, the result 'c' depends on whether 'c' itself belongs to a signed or unsigned type to determine whether the result is positive or negative. (Condition 2: both 'a' and 'b' need to be in the same bit position within their respective bit sets: $bits_{index} \quad operator \quad bits_{index}$).
Only when both condition 1 and condition 2 are met, can the following operations be carried out.

$c = a \wedge_{32} b$: This represents the **bitwise AND operation** in binary.

Specifically, when both bits are 1, the operation result is 1; when both bits are not 1, the operation result is 0. Please see the example formula for detailed operations.

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

$c = a \vee_{32} b$: This represents the **bitwise OR operation** in binary.

Specifically, when either bit is 1, the operation result is 1; when both bits are 0, the operation result is 0. Please see the example formula for detailed operations.

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

$bit' = \neg_{32} bit$: This represents the **bitwise NOT operation** in binary.

Specifically, when the bit is 1, the operation result is 0; when the bit is 0, the operation result is 1. Please see the example formula for detailed operations.

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

$c = a \oplus_{32} b$: This represents the **bitwise XOR operation** in binary.

Specifically, when both bits are in the same position, if they are the same, the operation result is 0; if they are different, the operation result is 1. Please see the example formula for detailed operations.

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

$c = a \odot_{32} b$: This represents the **bitwise XNOR operation** in binary.

Specifically, when both bits are the same, the operation result is 1; when they are different, the operation result is 0. Please see the example formula for detailed operations.

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

We define the **bitwise shift operation** in binary.

If the number being operated on is a signed number, then the left shift operation will discard the most significant bit (the sign bit) and shift the bit data to the left; conversely, the right shift operation will preserve the most significant bit (the sign bit) and shift the bit data to the right while discarding the least significant bit. This operation is called **arithmetic shift**.

If the number being operated on is an unsigned number, then the left shift operation will discard the most significant bit and shift the bit data to the left; conversely, the right shift operation will preserve the sign bit and shift the bit data to the right while discarding the least significant bit. This operation is called **logical shift**.

For all the above-defined operations, we perform them within a set of bits. If an operation causes the result to exceed the size limit of the bit set, those extra bits will be discarded.

$bits' = bits \ll_{32} number$: This represents the left bitwise shift operation.

Specifically, bits is a bit string or a bit set with similar units, and number is the number of bits to be shifted to the left. To prevent the operation from having undefined results, in this example number must satisfy $number = number \pmod{32}$. Please see the example formula for detailed operations.

$$bits = \{0,1,0,1,0,0,0,1\}(81USN) \quad \text{(bits size} < 9)$$
$$bits' = \{0,1,0,0,0,1,0,0\}(68USN) = bits \lll_8 2$$
$$bits = \{0,1,0,1,0,0,0,1\}(81SN) \quad \text{(bits size} < 9)$$
$$bits' = \{1,0,1,0,0,0,1,0\}(-94SN) = bits \lll_8 1$$

$bits' = bits \gg_{32} number$: This represents the right bitwise shift operation.

Specifically, bits is a bit string or a bit set with similar units, and number is the number of bits to be shifted to the right. To prevent the operation from having undefined results, in this example number must satisfy $number = number \pmod{32}$. Please see the example formula for detailed operations.

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

The process of the operation is similar to the previous **bit shift operation**, but it **does not discard any bit**.

$bits' = bits \lll_{32} number$: It represents the left circular shift operation.

The specific rule is that bits is a bit string or a similar unit of bit set, and number is the number of bits to be shifted to the left. To prevent the result of this operation from being undefined, in this example, number must satisfy $number = number \pmod{32}$. Please refer to the formula example for detailed operations.

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

The process of the operation is similar to the previous **bit shift operation**, but it **does not discard any bit**.

$bits' = bits \ggg_{32} number$: It represents the right circular shift operation.

The specific rule is that bits is a bit string or a similar unit of bit set, and number is the number of bits to be shifted to the right. To prevent the result of this operation from being undefined, in this example, number must satisfy $number = number \pmod{32}$. Please refer to the formula example for detailed operations.

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

We define the **assignment operation** in binary arithmetic.

$a := b$ means that the value or data of b is copied to a, but only when condition one is satisfied.

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** $\in \mathbb{F}_2^{64}$

```python
import numpy as np

class LinearFeedbackShiftRegister:

    """
    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)

```

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

        current_random_bit = 0

        # 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 : $x^{128} \oplus_{128} x^{41} \oplus_{128} x^{39} \oplus_{128} x \oplus_{128} 1$
            # 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)

        self.state[0] = NumberA
        self.state[1] = NumberB

        return answer

    def discard(self, round_number: np.uint64)-> None:
        for i in range(0, round_number)
            self.generate_bits(63)

    def __call__(self):
```

```python
            return self.generate_bits(63)

    def __del__(self):
        check_pointer = ctypes.c_void_p()
        ctypes.memset(ctypes.byref(state), 0, ctypes.sizeof(state))
        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** $\in \mathbb{F}_2^{64}$

```python
import numpy as np

"""
A random number generator using non-linear feedback shift register algorithm
"""
class NonlinearFeedbackShiftRegister:

    """
    Array position 0 is is the current random number seed
    Array position 1,2,3 the current random number
    """
    state = [np.uint64(0),np.uint64(0),np.uint64(0),np.uint64(0)]

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

    def seed(self, seed) -> None:
        if seed == 0:
            seed += 1

        # Initial state
        self.state[0] = seed
        self.state[1] = (seed * 2) + 1
        self.state[2] = (seed * 3) + 2
        self.state[3] = (seed * 4) + 3

        # Mix state (stage 1/2)
        self.state[0] += (self.state[1] ^ self.state[2]) ^ ~(self.state[3])
        self.state[1] -= (self.state[2] & self.state[3]) | self.state[0]
        self.state[2] += (self.state[3] ^ self.state[0]) ^ ~(self.state[1])
        self.state[3] -= (self.state[0] | self.state[1]) & self.state[2]

        # Mix state (stage 2/2)
        self.state[3] *= (seed << 48) & 0xffffffff
        self.state[2] *= (seed << 32) & 0xffffffff
        self.state[1] *= (seed << 16) & 0xffffffff
        self.state[0] *= (seed) & 0xffffffff

        # Update state
        for initial_round in range(128, 0, -1):
            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)
            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)
            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)
            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)

```

```python
            # Current random bit
            bit = (self.state[0] & 0x01) ^ (self.state[1] & 0x01) ^ (self.state[2] & 0x01) ^ (self.state[3] & 0x01)

            # Perform the nonlinear feedback function
            temporary_state = (self.state[0] ^ self.state[1]) & self.state[2] | self.state[3]

            # Override seed number values
            seed = (seed >> 49 | seed << 15) * (self.state[0] << 13 | self.state[0] >> 51)

            # Shift the values in the state array
            self.state[0], self.state[1], self.state[2], self.state[3] = self.state[1], self.state[2], self.state[3], temporary_state

            # In the (MSB/LSB) position, set a random bit
            seed |= (bit << 63) if (temporary_state & 0x01) else (bit & 0x01)

    """
    Apply complex properties of irreducible primitive polynomials to generate
    nonlinear random bit streams of numbers.

    Parameters:
    state_number (int): A 64-bit unsigned integer representing the current state value.
    irreducible_polynomial_count (int): An integer representing the degree of the primitive polynomial.
    bit (int): A value of either 0 or 1 representing the bit to XOR with the output.

    Returns:
    A tuple containing the updated state number and the output bit.
    """
    def random_bits(state_number, irreducible_polynomial_count, bit) -> np.uint64:
        # Binary polynomial data source: https://users.ece.cmu.edu/~koopman/lfsr/index.html
        # x is 2, for example: x ^ 3 = 2 * 2 * 2;

        switcher = {
        # Primitive polynomial degree is 24
        # x^23 + x^10 + x^9 + x^8 + x^6 + x^4 + x^3 + 1
        0: 0x80_0759

        # Primitive polynomial degree is 55
        # x^54 - x^10 - x^9 - x^8 - x^7 - x^6 - x^5 - x^4 - x^3 - x^2
        1: 0x40_0000_0000_07FC,

        # Primitive polynomial degree is 48
        # x^47 + x^11 + x^10 + x^8 + x^5 + x^4 + x^3 + 1
        2: 0x8000_0000_0D39,

        # Primitive polynomial degree is 31
        # x^30 - x^9 - x^8 - x^7 - x^5 - x^4 - x^3 - x^2 - x - 1
        3: 0x4000_03BF,

        # 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
        12358132134558914 4 == 0x1B70C8E97AD5F98

        PI Approximately equal to 3.1415926535897932384626433832795028841971693993751058209749445923078

        Circumference is a mathematical constant that is the ratio of the circumference of a circle to its diameter
        Binary format: 11.00100100001111110110101010001000100001011010001100001000110100110

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

        e Approximately equal to 2.7182818284590452353602874713526624977572470936999595749669676277240

        The Euler number is the base of the natural logarithm, not to be confused with the Euler-Mascheroni constant
        Binary format: 10.1011011111100001010100010110001010001010111011010010101001101010

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

        phi Approximately equal to 1.6180339887498948482045868343656386180339887498948482045868343656386

        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: 01.1001111000110111011110011011100101111111010010100111110000010101

        The binary numbers are stripped of the floating point portion and converted to hexadecimal, i.e: 0x9E3779B97F4A7C15
        """
        def generate_chaotic_number(self, algorithm_execute_count: np.uint64) -> np.uint64:
            """
            Hamming weights (number of bits with 1)
            bin(value).count("1")
            """
            FibonacciSequence = 0x1B70C8E97AD5F98
            CircumferenceSequence = 0x243F6A8885A308D3
            GoldenRatioSequence = 0x9E3779B97F4A7C15
```

```python
        EulerNumberSequence = 0xB7E151628AED2A6A

        FibonacciSequenceBytes = unpack_8byte(FibonacciSequence)
        CircumferenceSequenceBytes = unpack_8byte(CircumferenceSequence)
        GoldenRatioSequenceBytes = unpack_8byte(GoldenRatioSequence)
        EulerNumberSequenceBytes = unpack_8byte(EulerNumberSequence)
        Number2Power64Modulus = np.uint64(2**64 - 1)

        if algorithm_execute_count < 8:
            algorithm_execute_count = 8

        answer = 0

        for round_counter in range(algorithm_execute_count):
            bit = (self.state[0] ^ self.state[1] ^ self.state[2] ^ self.state[3]) & 0x01

            answer <<= 1
            answer |= bit

            if (bin(answer).count('1') & 0x01) != 0:
                answer ^= CircumferenceSequence
            else:
                multiplied_number_byte_span = memory_data_format_exchanger.Unpacker_8Byte(answer)

            SequenceBytes = FibonacciSequenceBytes if (answer ^ self.state[1]) & 0x01 else GoldenRatioSequenceBytes

            for index in range(sizeof(np.uint64)):
                multiplied_number_byte_span[index] = GF256_Instance.multiplication(multiplied_number_byte_span[index],
                ↪ SequenceBytes[index])

            answer ^= memory_data_format_exchanger.Packer_8Byte(multiplied_number_byte_span)

            if (bin(self.state[2]).count('1') & 0x01) == 0:
                multiplied_number_byte_span = memory_data_format_exchanger.Unpacker_8Byte(self.state[2])

            SequenceBytes = EulerNumberSequenceBytes if (answer ^ self.state[3]) & 0x01 else CircumferenceSequenceBytes

            for index in range(sizeof(np.uint64)):
                multiplied_number_byte_span[index] = GF256_Instance.multiplication(multiplied_number_byte_span[index],
                ↪ SequenceBytes[index])

            self.state[2] ^= memory_data_format_exchanger.Packer_8Byte(multiplied_number_byte_span)

            if (self.state[2] & 0x01) == 0:
                self.state[2] ^= FibonacciSequence
            else:
                self.state[2] ^= GoldenRatioSequence ^ answer

            if (self.state[2] & 0x01) != 0:
                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)
```

```python

                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.
        Depending on whether the (base_number) argument is odd or even, it determines one of the two different bit sequences that will be generated.

        However, there is an exception to this rule
        If the (number_bit) parameter is greater than or equal to 64
        the linear feedback shift register (result value - answer) is broken because the number of bits shifted right or left is greater than 64
        Then the sequence will be chaotic in a way that even the linear feedback shift register is not known.
        Even though all the parameters provided and the internal state are the same, you can restore these sequences

        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.
```

```
        Args:
            base_number: An integer to determine which bit sequence will be generated.
            number_bits: The number of bits to generate.

        Returns:
            An integer representing the generated unpredictable bit sequence.

        """

        answer = base_number
        current_random_bit = 0

        current_random_bits = [0, 0, 0, 0]

        for round_counter in range(number_bits):
            current_random_bit = ((self.state[0] ^ self.state[1] ^ self.state[2] ^ self.state[3]) >> 63) & 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 0 or 1
            answer ^= current_random_bit

            # Compute pseudo-random bit sequences in binary

            # 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
            self.state[0] = self.random_bits(self.state[0], (self.state[3] ^ self.state[2]) % 9, current_random_bit)

            # Only one binary random bit is retained
            current_random_bits[0] ^= self.state[0] & 0x01

            self.state[1] = self.random_bits(self.state[1], (self.state[2] ^ self.state[1]) % 9, current_random_bit)

            # Only one binary random bit is retained
            current_random_bits[1] ^= self.state[1] & 0x01

            self.state[2] = self.random_bits(self.state[2], (self.state[1] ^ self.state[0]) % 9, current_random_bit)

            # Only one binary random bit is retained
            current_random_bits[2] ^= self.state[2] & 0x01

            self.state[3] = self.random_bits(self.state[3], (self.state[0] ^ self.state[3]) % 9, current_random_bit)

            # Only one binary random bit is retained
            current_random_bits[3] ^= self.state[3] & 0x01

            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)

```
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
```

```python

    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

```

```python
    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

```

```python
            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:
```

```python
                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

For more details on the implementation of the algorithms in c++ for this project, please see the documentation:

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

```python
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)

```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.

    Args:
    - old_data_box: a list of 256 bytes representing the old substitution box

    Returns:
    - a list of 256 bytes representing the new substitution box
    """

    check_pointer = ctypes.c_void_p()

    # initialize the NLFSR and the segment tree
```

```
    nlfsr_object = CommonStateData.nlfsr
    old_data_array_size = len(old_data_box)
    segment_tree_object = SegmentTree(256)

    new_data_box = [0] * 256
    new_data_array_size = len(new_data_box)

    index = 0
    index2 = 0

    while index < old_data_array_size and index2 < new_data_array_size:
        if index == old_data_array_size - 1 and old_data_box[index] == segment_tree_object.get(0):
            # Need to re-operate data
            check_pointer = ctypes.memset(ctypes.byref(new_data_box), 0, ctypes.sizeof(new_data_box))
            check_pointer = None
            segment_tree_object.clear()
            index = 0
            index2 = 0
            continue

        order = nlfsr_object() % (old_data_array_size - index)
        position = segment_tree_object.get(order)

        while old_data_box[index] == position:
            order = nlfsr_object() % (old_data_array_size - index)
            position = segment_tree_object.get(order)

        new_data_box[index2] = position
        segment_tree_object.set(position)
        index += 1
        index2 += 1

    return new_data_box
```

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

```cpp
template<std::integral DataType, std::size_t ArraySize>
class SegmentTree
{

    /*
        std::has_single_bit(ArraySize)
        ArraySize != 0 && (ArraySize ^ (ArraySize & -ArraySize) == 0)
    */

private:

    static constexpr std::size_t N = std::has_single_bit(ArraySize) ? ArraySize : 0;
    std::array<DataType, N << 1> Nodes {};

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)

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

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]
```

```python

        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)
            StateBuffer3 = [0] * BITS_STATE_SIZE

            for RoundIndex in range(BitsHashState_size - 1 - Counter, BitsHashState_size):
                # Step 1
                while self.StateCurrentCounter % BitsHashState_size != 0:
                    StateBuffer[self.StateCurrentCounter % (BITS_STATE_SIZE // 2)] = BitsHashState[self.StateCurrentCounter %
                    ↪  BitsHashState_size] ^ BitsHashState[(self.StateCurrentCounter + 1) % BitsHashState_size]
                    self.StateCurrentCounter += 1
                    StateBuffer[self.StateCurrentCounter % (BITS_STATE_SIZE // 2)] = BitsHashState[(self.StateCurrentCounter +
                    ↪  2) % BitsHashState_size] ^ BitsHashState[(self.StateCurrentCounter + 3) % BitsHashState_size]
                    self.StateCurrentCounter += 1

```

```python
            # Step 2
            for StateBufferIndex in range(len(StateBufferIndices)):
                StateBuffer2[StateBufferIndex] = StateBuffer[RightRotatedStateBufferIndices[StateBufferIndex]] ^ Binary_RightRotateMove(StateBuffer[LeftRotatedStateBufferIndices[StateBufferIndex]], 1)

            # Step 3
            StateBuffer3[0] = BitsHashState[0] ^ StateBuffer2[0]
            for StateBufferIndex in range(1, len(StateBuffer3)):
                StateBuffer3[HashStateIndices[StateBufferIndex]] = Binary_RightRotateMove(BitsHashState[StateBufferIndex] ^ StateBuffer2[StateBufferIndex % len(StateBuffer2)], MoveBitCounts[self.StateCurrentCounter % len(MoveBitCounts)])
                self.StateCurrentCounter += 1

            # Step 4
            for StateBufferIndex in range(len(StateBuffer3)):
                BitsHashState[StateBufferIndex] = StateBuffer3[StateBufferIndex] ^ (~(StateBuffer3[(StateBufferIndex + 1) % len(StateBuffer3)]) & StateBuffer3[(StateBufferIndex + 2) % len(StateBuffer3)])

            # Step 5
            BitsHashState[0] ^= HASH_ROUND_CONSTANTS[RoundIndex % len(HASH_ROUND_CONSTANTS)]
            BitsHashState[BitsHashState_size - 1] ^= HASH_ROUND_CONSTANTS[(len(HASH_ROUND_CONSTANTS) - 1 - RoundIndex) % len(HASH_ROUND_CONSTANTS)]

    def AbsorbInputData(self, ByteDatas: bytes) -> None:
        BitWords = [0] * (len(ByteDatas) // 8)

        for i in range(0, len(ByteDatas), 8):
            BitWords[i//8] = int.from_bytes(ByteDatas[i:i+8], byteorder='little')

        for InputBytesIndex in range(BITWORDS_RATE):
            for OutputBytesIndex in range(0, len(BitWords)):
                if InputBytesIndex >= BITWORDS_RATE:
                    InputBytesIndex = 0
                self.BitsHashState[InputBytesIndex] ^= BitWords[OutputBytesIndex]

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

        # Clear sensitive information from BitWords
        for i in range(len(BitWords)):
            BitWords[i] = 0

    def AbsorbInputData(self, BitWordDatas: List[int]) -> None:
        for InputBitsIndex in range(BITWORDS_RATE):
            for OutputBitsIndex in range(len(BitWordDatas)):
                if InputBitsIndex >= BITWORDS_RATE:
                    InputBitsIndex = 0
                self.BitsHashState[InputBitsIndex] ^= BitWordDatas[OutputBitsIndex]

                # 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++)

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

    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:

    /*
        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.
    */

    static constexpr std::uint64_t BITS_STATE_SIZE = 2 * HashBitSize + std::numeric_limits<std::uint64_t>::digits;
    static constexpr std::uint64_t BITS_RATE = HashBitSize;
    static constexpr std::uint64_t BITS_CAPACITY = BITS_STATE_SIZE - BITS_RATE;

    static constexpr std::uint64_t BYTES_RATE = BITS_RATE / std::numeric_limits<std::uint8_t>::digits;
    static constexpr std::uint64_t BITWORDS_RATE = BYTES_RATE / sizeof(std::uint64_t);
    static constexpr std::uint64_t BYTES_CAPACITY = BITS_CAPACITY / std::numeric_limits<std::uint8_t>::digits;
    static constexpr std::uint64_t BITWORDS_CAPACITY = BYTES_CAPACITY / sizeof(std::uint64_t);

    static constexpr std::uint8_t PAD_BYTE_DATA = 0b00000001;
    static constexpr std::uint64_t PAD_BITSWORD_DATA = 0b0000000100000001000000010000000100000001000000010000000100000001;

    std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> BitsHashState {};

    static constexpr std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> StateBufferIndices = GenerateHashStateBufferIndices<BITS_STATE_SIZE>();

    const std::array<std::uint32_t, 63> MoveBitCounts {};
    const std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> HashStateIndices {};
    std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> LeftRotatedStateBufferIndices;
    std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> RightRotatedStateBufferIndices;

    std::size_t StateCurrentCounter = 1;

```

```cpp
        /*
            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
        */
        static constexpr std::array<std::uint64_t, 64> HASH_ROUND_CONSTANTS
        {

        ↪ 0xe02d51d52e6988abULL,0xfc48780c20090b50ULL,0xc6144c4d89151352ULL,0xb98669bb3a32a8f1ULL,0xd4786928fe033c03ULL,0xaebb38f01d73faabUL

        ↪ 0x06d5b3dbf088ae77ULL,0x7e2be74e7f525e23ULL,0xe5459a079549e2e3ULL,0x352ba71a6a95e6d6ULL,0x7b40c16d92d5e43bULL,0xa559af839ba27363UL

        ↪ 0x9ab94838ff7737c6ULL,0x718d70cd883014f9ULL,0x0bda9af50ba21d4dULL,0xd88cb07c07a814d5ULL,0xa6c8d66f9b3d8933ULL,0x80643413e011c839UL

        ↪ 0x19224d7b455813b1ULL,0xb1dbd44f138bac7fULL,0x2ba9107bb26a6134ULL,0x48297fe2c4167b76ULL,0x776528a5edb8a68eULL,0x2381e0eb054681a8UL

        ↪ 0x655f38e3d5446574ULL,0xd8093b5a1172958cULL,0x28880627fe4c014bULL,0x0459d6592d1b2b51ULL,0x2aeb8df1c83b63beULL,0xcba3ca8c513a8205UL

        ↪ 0xdf8ee44352384448ULL,0xff38527afa3b13a2ULL,0x9ff904a86c03fe22ULL,0xe81a56aef956f93fULL,0x3c13136bf0612494ULL,0xca9b0621705e9748UL

        ↪ 0xd249f4efd3685008ULL,0xda2779c07b0e4a43ULL,0x1cc1bd402438ea81ULL,0x7b090a135f97ba29ULL,0xd25e80bc98b09e4bULL,0xeea820f2885ac1f8UL

        ↪ 0x75208f3a3cb244dfULL,0x20f74f61571512b4ULL,0xfd526ef256343eb7ULL,0x753082ea79791d09ULL,0x41a3a000a8c7ae30ULL,0xb2a056be3a257d27UL
        };

        std::array<std::uint32_t, 63>
        GenerateRandomMoveBitCounts()
        {
            CommonSecurity::RNG_ISAAC::isaac64<8> CSPRNG = CommonSecurity::RNG_ISAAC::isaac64<8>(1946379852749613ULL);

            CSPRNG.discard(1024);

            std::array<std::uint32_t, 63> MoveBitCounts {};

            std::iota(MoveBitCounts.begin(), MoveBitCounts.end(), 1);

            for(std::uint64_t Index = 0; Index < MoveBitCounts.size(); ++Index)
            {
                std::swap(MoveBitCounts[Index], MoveBitCounts[(Index + CSPRNG()) % MoveBitCounts.size()]);
            }

            return MoveBitCounts;
        }

        std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits>
        GenerateRandomHashStateIndices()
        {
            CommonSecurity::RNG_ISAAC::isaac64<8> CSPRNG = CommonSecurity::RNG_ISAAC::isaac64<8>(1946379852749613ULL);

            CSPRNG.discard(2048);

            std::array<std::uint32_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> RandomHashStateIndices {};

            std::iota(RandomHashStateIndices.begin(), RandomHashStateIndices.end(), 0);

            for(std::uint64_t Index = 0; Index < RandomHashStateIndices.size(); ++Index)
            {
                std::swap(RandomHashStateIndices[Index], RandomHashStateIndices[(Index + CSPRNG()) %
        ↪ RandomHashStateIndices.size()]);
            }

            return RandomHashStateIndices;
```

```cpp
        }

        /*
            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
        */
        void TransfromState(std::size_t Counter)
        {
            using CommonSecurity::Binary_LeftRotateMove;
            using CommonSecurity::Binary_RightRotateMove;

            std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> StateBuffer {};
            std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits / 2> StateBuffer2 {};
            std::array<std::uint64_t, BITS_STATE_SIZE / std::numeric_limits<std::uint64_t>::digits> StateBuffer3 {};

            for(std::size_t RoundIndex = BitsHashState.size() - 1 - Counter; RoundIndex < BitsHashState.size(); ++RoundIndex)
            {
                //Step 1
                while(StateCurrentCounter % BitsHashState.size() != 0)
                {
                    StateBuffer[StateCurrentCounter % StateBuffer.size()] = BitsHashState[StateCurrentCounter % BitsHashState.size()] ^ BitsHashState[(StateCurrentCounter + 1) % BitsHashState.size()];
                    ++StateCurrentCounter;
                    StateBuffer[StateCurrentCounter % StateBuffer.size()] = BitsHashState[(StateCurrentCounter + 2) % BitsHashState.size()] ^ BitsHashState[(StateCurrentCounter + 3) % BitsHashState.size()];
                    ++StateCurrentCounter;
                }

                //Step 2
                for(std::size_t StateBufferIndex = 0; StateBufferIndex < StateBufferIndices.size(); ++StateBufferIndex)
                {
                    StateBuffer2[StateBufferIndex] = StateBuffer[RightRotatedStateBufferIndices[StateBufferIndex]] ^ Binary_RightRotateMove<std::uint64_t>(StateBuffer[LeftRotatedStateBufferIndices[StateBufferIndex]], 1);
                }

                //Step 3
                StateBuffer3[0] = BitsHashState[0] ^ StateBuffer2[0];

                for(std::size_t StateBufferIndex = 1; StateBufferIndex < StateBuffer3.size(); ++StateBufferIndex)
                {
                    StateBuffer3[HashStateIndices[StateBufferIndex]] = Binary_RightRotateMove<std::uint64_t>(BitsHashState[StateBufferIndex] ^ StateBuffer2[StateBufferIndex % StateBuffer2.size()], MoveBitCounts[StateCurrentCounter % MoveBitCounts.size()]);
                    ++StateCurrentCounter;
                }

                //Step 4
                for(std::size_t StateBufferIndex = 0; StateBufferIndex < StateBuffer3.size(); ++StateBufferIndex)
                {
                    BitsHashState[StateBufferIndex] = StateBuffer3[StateBufferIndex] ^ ( ~(StateBuffer3[(StateBufferIndex + 1) % StateBuffer3.size()]) & StateBuffer3[(StateBufferIndex + 2) % StateBuffer3.size()] );
                }

                //Step 5
```

```
                BitsHashState[0] ^= HASH_ROUND_CONSTANTS[RoundIndex % HASH_ROUND_CONSTANTS.size()];
                BitsHashState[BitsHashState.size() - 1] ^= HASH_ROUND_CONSTANTS[(HASH_ROUND_CONSTANTS.size() - 1 - RoundIndex) % HASH_ROUND_CONSTANTS.size()];
            }
        }

        void AbsorbInputData(std::span<const std::uint8_t> ByteDatas)
        {
            using CommonToolkit::IntegerExchangeBytes::MessagePacking;

            std::vector<std::uint64_t> BitWords(ByteDatas.size() / sizeof(std::uint64_t), 0);

            MessagePacking<std::uint64_t, std::uint8_t>(ByteDatas, BitWords.data());

            for(std::uint64_t InputBytesIndex = 0, OutputBytesIndex = 0; OutputBytesIndex < BitWords.size(); ++InputBytesIndex, ++OutputBytesIndex)
            {
                if(InputBytesIndex >= BITWORDS_RATE)
                    InputBytesIndex = 0;
                BitsHashState[InputBytesIndex] ^= BitWords[OutputBytesIndex];

                //State permutation and transformation (string of information entropy pool)
                this->TransfromState(BitsHashState.size());
            }

            memory_set_no_optimize_function<0x00>(BitWords.data(), BitWords.size() * sizeof(std::uint64_t));
        }

        void AbsorbInputData(std::span<const std::uint64_t> BitWordDatas)
        {
            for(std::uint64_t InputBitsIndex = 0, OutputBitsIndex = 0; OutputBitsIndex < BitWordDatas.size(); ++InputBitsIndex, ++OutputBitsIndex)
            {
                if(InputBitsIndex >= BITWORDS_RATE)
                    InputBitsIndex = 0;
                BitsHashState[InputBitsIndex] ^= BitWordDatas[OutputBitsIndex];

                //State permutation and transformation (string of information entropy pool)
                this->TransfromState(BitsHashState.size());
            }
        }

        void SqueezeOutputData(std::span<std::uint8_t> ByteDatas)
        {
            using CommonToolkit::IntegerExchangeBytes::MessageUnpacking;

            std::vector<std::uint64_t> BitWords(HashBitSize / std::numeric_limits<std::uint64_t>::digits, 0);

            size_t BitsIndexOffest = 0;

            for(std::uint64_t BitsIndex = 0; BitsIndex < BitWords.size(); ++BitsIndex)
            {
                BitWords[BitsIndex] = BitsHashState[BitsIndexOffest];

                if(BitsIndexOffest >= BITWORDS_RATE)
                {
                    //State permutation and transformation (string of information entropy pool)
                    this->TransfromState(BitsHashState.size());

                    BitsIndexOffest = 0;
                }
```

```cpp
            }

            MessageUnpacking<std::uint64_t, std::uint8_t>(BitWords, ByteDatas.data());
        }

        void SqueezeOutputData(std::span<std::uint64_t> WordDatas)
        {
            size_t BitsIndexOffest = 0;

            for(std::uint64_t BitsIndex = 0; BitsIndex < (HashBitSize / std::numeric_limits<std::uint64_t>::digits); ++BitsIndex)
            {
                WordDatas[BitsIndex] = BitsHashState[BitsIndexOffest];

                if(BitsIndexOffest >= BITWORDS_RATE)
                {
                    //State permutation and transformation (string of information entropy pool)
                    this->TransfromState(BitsHashState.size());

                    BitsIndexOffest = 0;
                }
            }
        }

    public:

        void Reset()
        {
            this->StateCurrentCounter = 0;
            memory_set_no_optimize_function<0x00>(BitsHashState.data(), BitsHashState.size() * sizeof(std::uint64_t));
        }

        //Tests that do not provide external data
        std::vector<std::uint64_t> Test()
        {
            for(std::size_t BlockCounter = 0; BlockCounter < (HashBitSize / std::numeric_limits<std::uint64_t>::digits); BlockCounter++)
            {
                this->TransfromState(BlockCounter);
            }

            std::vector<std::uint64_t> TestData(HashBitSize / std::numeric_limits<std::uint64_t>::digits, 0);
            this->SqueezeOutputData(TestData);

            this->Reset();

            return TestData;
        }

        void SecureHash
        (
            std::span<const std::uint8_t> InputData,
            std::span<std::uint8_t> OuputData
        )
        {
            std::vector<std::uint8_t> BlockDataBuffer(InputData.begin(), InputData.end());

            //Pad data and Absorbing data stage
            if(BlockDataBuffer.size() % BYTES_RATE != 0)
            {
                for(std::size_t PadCount = 0; PadCount < BlockDataBuffer.size() % BYTES_RATE; ++PadCount)
```

```
            {
                BlockDataBuffer.push_back(PAD_BYTE_DATA);
            }
        }
        this->AbsorbInputData(BlockDataBuffer);

        //squeeze data stage
        this->SqueezeOutputData(OuputData);

        memory_set_no_optimize_function<0x00>(BlockDataBuffer.data(), BlockDataBuffer.size());

        //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
        this->Reset();
    }

    void SecureHash
    (
        std::span<const std::uint64_t> InputData,
        std::span<std::uint64_t> OuputData
    )
    {
        std::vector<std::uint64_t> BlockDataBuffer(InputData.begin(), InputData.end());

        //Pad data and Absorbing data stage
        if(BlockDataBuffer.size() % BITWORDS_RATE != 0)
        {
            for(std::size_t PadCount = 0; PadCount < BlockDataBuffer.size() % BYTES_RATE; ++PadCount)
            {
                BlockDataBuffer.push_back(PAD_BITSWORD_DATA);
            }
        }
        this->AbsorbInputData(BlockDataBuffer);

        //squeeze data stage
        this->SqueezeOutputData(OuputData);

        memory_set_no_optimize_function<0x00>(BlockDataBuffer.data(), BlockDataBuffer.size() * sizeof(std::uint64_t));

        //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
        this->Reset();
    }

    CustomSecureHash()
        :
        MoveBitCounts(GenerateRandomMoveBitCounts()), HashStateIndices(GenerateRandomHashStateIndices())
    {
        static_assert(HashBitSize >= 128 && HashBitSize % 8 == 0, "");

        std::ranges::rotate_copy(StateBufferIndices.begin(), StateBufferIndices.begin() + 1, StateBufferIndices.end(), LeftRotatedStateBufferIndices.begin());
        std::ranges::rotate_copy(StateBufferIndices.begin(), StateBufferIndices.end() - 1, StateBufferIndices.end(), RightRotatedStateBufferIndices.begin());
    }
};
```

Code block 4: ISAAC PRNG (c++)

```
/*
    RNG_ISAAC contains code common to isaac and isaac64.
    It uses CRTP (a.k.a. 'static polymorphism') to invoke specialized methods in the derived class templates,
```

```
    avoiding the cost of virtual method invocations and allowing those methods to be placed inline by the compiler.
    Applications should not specialize or instantiate this template directly.
*/

template<std::size_t Alpha, class T>
class RNG_ISAAC
{
public:
    using result_type = T;

    static constexpr std::size_t state_size = 1 << Alpha;

    static constexpr result_type default_seed = 0;

    RNG_ISAAC()
    {
        seed(default_seed);
    }

    explicit RNG_ISAAC(result_type seed_number)
        : issac_base_member_counter(state_size)
    {
        seed(seed_number);
    }

    template <typename SeedSeq>
    requires( not std::convertible_to<SeedSeq, result_type> )
    explicit RNG_ISAAC( SeedSeq& number_sequence )
        : issac_base_member_counter(state_size)
    {
        seed(number_sequence);
    }

    RNG_ISAAC(const std::vector<result_type>& seed_vector)
        : issac_base_member_counter(state_size)
    {
        seed(seed_vector);
    }

    template<class IteratorType>
    RNG_ISAAC
    (
        IteratorType begin,
        IteratorType end,
        typename std::enable_if
        <
                std::is_integral<typename std::iterator_traits<IteratorType>::value_type>::value &&
                std::is_unsigned<typename std::iterator_traits<IteratorType>::value_type>::value
        >::type* = nullptr
    )
    : issac_base_member_counter(state_size)
    {
        seed(begin, end);
    }

    RNG_ISAAC(std::random_device& random_device_object)
        : issac_base_member_counter(state_size)
    {
        seed(random_device_object);
    }

```

```
        RNG_ISAAC(const RNG_ISAAC& other)
            : issac_base_member_counter(state_size)
        {
            for (std::size_t index = 0; index < state_size; ++index)
            {
                issac_base_member_result[index] = other.issac_base_member_result[index];
                issac_base_member_memory[index] = other.issac_base_member_memory[index];
            }
            issac_base_member_register_a = other.issac_base_member_register_a;
            issac_base_member_register_b = other.issac_base_member_register_b;
            issac_base_member_register_c = other.issac_base_member_register_c;
            issac_base_member_counter = other.issac_base_member_counter;
        }

    public:

        static constexpr result_type min()
        {
            return std::numeric_limits<result_type>::min();
        }
        static constexpr result_type max()
        {
            return std::numeric_limits<result_type>::max();
        }

        inline void seed(result_type seed_number)
        {
            for (std::size_t index = 0; index < state_size; ++index)
            {
                issac_base_member_result[index] = seed_number;
            }
            init();
        }

        template <typename SeedSeq>
        requires( not std::convertible_to<SeedSeq, result_type> )
        constexpr void seed( SeedSeq& number_sequence )
        {
            std::seed_seq my_seed_sequence(number_sequence.begin(), number_sequence.end());
            std::array<result_type, state_size> seed_array;
            my_seed_sequence.generate(seed_array.begin(), seed_array.end());
            for (std::size_t index = 0; index < state_size; ++index)
            {
                issac_base_member_result[index] = seed_array[index];
            }
            init();
        }

        template<class IteratorType>
        inline typename std::enable_if
        <
            std::is_integral<typename std::iterator_traits<IteratorType>::value_type>::value &&
            std::is_unsigned<typename std::iterator_traits<IteratorType>::value_type>::value, void
        >::type
        seed(IteratorType begin, IteratorType end)
        {
            IteratorType iterator = begin;
            for (std::size_t index = 0; index < state_size; ++index)
            {
                if (iterator == end)
                {
```

```
                    iterator = begin;
                }
                issac_base_member_result[index] = *iterator;
                ++iterator;
            }
            init();
        }

        void seed(std::random_device& random_device_object)
        {
            std::vector<result_type> random_seed_vector;
            random_seed_vector.reserve(state_size);
            for (std::size_t round = 0; round < state_size; ++round)
            {
                result_type seed_number_value = GenerateSecureRandomNumberSeed<result_type>(random_device_object);

                std::size_t bytes_filled{sizeof(std::random_device::result_type)};
                while(bytes_filled < sizeof(result_type))
                {
                    result_type seed_number_value2 = GenerateSecureRandomNumberSeed<result_type>(random_device_object);

                    seed_number_value <<= (sizeof(std::random_device::result_type) * 8);
                    seed_number_value |= seed_number_value2;
                    bytes_filled += sizeof(std::random_device::result_type);
                }
                random_seed_vector.push_back(seed_number_value);
            }
            seed(random_seed_vector.begin(), random_seed_vector.end());
        }

        inline result_type operator()()
        {
            if(issac_base_member_counter - 1 == std::numeric_limits<std::size_t>::max())
                issac_base_member_counter = state_size - 1;

            return (!issac_base_member_counter--) ? (do_isaac(), issac_base_member_result[issac_base_member_counter]) : issac_base_member_result[issac_base_member_counter];
        }

        inline void discard(unsigned long long z)
        {
            for (; z; --z) operator()();
        }

        ~RNG_ISAAC() = default;

    private:

        /*
            ISAAC (Indirection, Shift, Accumulate, Add, and Count) generates 32-bit random numbers.
            Averaged out, it requires 18.75 machine cycles to generate each 32-bit value.
            Cycles are guaranteed to be at least 2(^)40 values long, and they are 2(^)8295 values long on average.
            The results are uniformly distributed, unbiased, and unpredictable unless you know the seed.
        */

        void implementation_isaac()
        {
            /*
                Modulo a power of two, the following works (assuming twos complement representation):

                i mod n == i & (n-1) when n is a power of two and mod is the aforementioned positive mod.
```

```
            (FYI: modulus is the common mathematical term for the "divisor" when a modulo operation is considered).

            return i & (n-1);

            auto lambda_Modulo = [](result_type value, result_type modulo_value)
            {
                return modulo_value & ( modulo_value - 1) ? value % modulo_value : value & ( modulo_value - 1);
            };
        */

        result_type index = 0, x = 0, y = 0, state_random_value = 0;

        result_type accumulate = this->issac_base_member_register_a;
        result_type bit_result = this->issac_base_member_register_b + (++(this->issac_base_member_register_c)); //b ← (c + 1)

        for (index = 0; index < this->state_size; ++index)
        {
            //x ← state[index]
            x = this->issac_base_member_memory[index];
            /*
                //barrel shift

                function(a, index)
                {
                    if index   0 mod 4
                        return a ^= a << 13
                    if index   1 mod 4
                        return a ^= a << 6
                    if index   2 mod 4
                        return a ^= a << 2
                    if index   3 mod 4
                        return a ^= a << 16
                }

                mix_index ← function(a, index);
            */
            switch (index & 3)
            {
                case 0:
                    accumulate ^= accumulate << 13;
                    break;
                case 1:
                    accumulate ^= accumulate >>  6;
                    break;
                case 2:
                    accumulate ^= accumulate <<  2;
                    break;
                case 3:
                    accumulate ^= accumulate >> 16;
                    break;
            }
            // a(mix_index) + state[index] + 128 mod 256
            accumulate += this->issac_base_member_memory[ (index + this->state_size / 2) & (this->state_size - 1) ];
            //state[index] ← a(mix_index)   b + (state[x] >>> 2) mod 256
            //y == state[index]
            state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(x, 2) & (this->state_size - 1) ];
            y = accumulate ^ bit_result + state_random_value;
            this->issac_base_member_memory[index] = y;
            //result[index] ← x + a(mix_index)   (state[state[index]] >>> 10) mod 256
```

```
                //b == result[index]
                state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(y, 10) & (this->state_size - 1) ];
                bit_result = x + accumulate ^ state_random_value;
                this->issac_base_member_result[index] = bit_result;
            }
        }

        /*
            ISAAC-64 generates a different sequence than ISAAC, but it uses the same principles. It uses 64-bit arithmetic.
            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.

            The following files implement ISAAC-64.
            The constants were tuned for a 64-bit machine, and a complement was thrown in so that all-zero states become nonzero faster.
        */

        void implementation_isaac64()
        {
            /*
                Modulo a power of two, the following works (assuming twos complement representation):

                i mod n == i & (n-1) when n is a power of two and mod is the aforementioned positive mod.
                (FYI: modulus is the common mathematical term for the "divisor" when a modulo operation is considered).

                return i & (n-1);

                auto lambda_Modulo = [](result_type value, result_type modulo_value)
                {
                    return modulo_value & ( modulo_value - 1) ? value % modulo_value : value & ( modulo_value - 1);
                };
            */

            result_type index = 0, x = 0, y = 0, state_random_value = 0;

            result_type accumulate = this->issac_base_member_register_a;
            result_type bit_result = this->issac_base_member_register_b + (++(this->issac_base_member_register_c)); //b ← (c + 1)

            for (index = 0; index < this->state_size; ++index)
            {
                //x ← state[index]
                x = this->issac_base_member_memory[index];
                /*
                    //barrel shift

                    function(a, index)
                    {
                        if index   0 mod 4
                            return a ^= ~(a << 21)
                        if index   1 mod 4
                            return a ^= a << 5
                        if index   2 mod 4
                            return a ^= a << 12
                        if index   3 mod 4
                            return a ^= a << 33
                    }

                    mix_index ← function(a, index);
                */
```

```
                switch (index & 3)
                {
                    case 0:
                        accumulate ^= ~(accumulate << 21);
                        break;
                    case 1:
                        accumulate ^= accumulate >>  5;
                        break;
                    case 2:
                        accumulate ^= accumulate << 12;
                        break;
                    case 3:
                        accumulate ^= accumulate >> 33;
                        break;
                }
                // a(mix_index) + state[index] + 128 mod 256
                accumulate += this->issac_base_member_memory[ (index + this->state_size / 2) & (this->state_size - 1) ];
                //state[index] ← a(mix_index)   b + (state[x] >>> 2) mod 256
                //y == state[index]
                state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(x, 2) &
↪   (this->state_size - 1) ];
                y = accumulate ^ bit_result + state_random_value;
                this->issac_base_member_memory[index] = y;
                //result[index] ← x + a(mix_index)   (state[state[index]] >>> 10) mod 256
                //b == result[index]
                state_random_value = this->issac_base_member_memory[ Binary_RightRotateMove<result_type>(y, 10) &
↪   (this->state_size - 1) ];
                bit_result = x + accumulate ^ state_random_value;
                this->issac_base_member_result[index] = bit_result;
            }
        }

        void init()
        {
            result_type a = golden();
            result_type b = golden();
            result_type c = golden();
            result_type d = golden();
            result_type e = golden();
            result_type f = golden();
            result_type g = golden();
            result_type h = golden();

            issac_base_member_register_a = 0;
            issac_base_member_register_b = 0;
            issac_base_member_register_c = 0;

            /* scramble it */
            for (std::size_t index = 0; index < 4; ++index)
            {
                mix(a,b,c,d,e,f,g,h);
            }

            /* initialize using the contents of issac_base_member_result[] as the seed */
            for (std::size_t index = 0; index < state_size; index += 8)
            {
                a += issac_base_member_result[index];
                b += issac_base_member_result[index+1];
                c += issac_base_member_result[index+2];
                d += issac_base_member_result[index+3];
                e += issac_base_member_result[index+4];
```

```
            f += issac_base_member_result[index+5];
            g += issac_base_member_result[index+6];
            h += issac_base_member_result[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;
        }

        /* 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;
                e += b;
                c += d;

                c ^= d << 8;
                f += c;
                d += e;

                d ^= e >> 16;
                g += d;
                e += f;

                e ^= f << 10;
                h += e;
                f += g;

                f ^= g >> 4;
                a += f;
                g += h;

                g ^= h << 8;
                b += g;
                h += a;

                h ^= a >> 9;
                c += h;
                a += b;
            }
            else if constexpr(std::same_as<result_type,std::uint64_t>)
            {
                a -= e;
                f ^= h >> 9;
                h += a;

                b -= f;
                g ^= a << 9;
                a += b;

                c -= g;
                h ^= b >> 23;
                b += c;

                d -= h;
                a ^= c << 15;
                c += d;

                e -= a;
                b ^= d >> 14;
                d += e;

                f -= b;
                c ^= e << 20;
```

```
            e += f;

            g -= c;
            d ^= f >> 17;
            f += g;

            h -= d;
            e ^= g << 14;
            g += h;
        }
    }

    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::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 )
        {
            for(const auto& Value : DiffusionLayerMatrixIndex[X] )
                std::cout << "KeyStateX" << "[" << Value << "]" << ", ";

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

        std::cout << std::endl;
    }
```