TitanWall Stream Cipher & Block Cipher: An In-depth Analysis

Abstract

The TitanWall Stream Cipher & Block Cipher is an innovative encryption and decryption algorithm tailored for robust data security.
This document delves into the algorithm's design, principles, and implementation, providing a comprehensive understanding of its workings.

Introduction

The realm of modern cryptography has seen the evolution of numerous ciphering algorithms, each aiming to achieve robust security while maintaining computational efficiency.
Within this context, the TitanWall Cipher, as detailed in this document, serves as a paradigm of cutting-edge cryptographic design, amalgamating both block and stream cipher methodologies.
Its underlying architecture draws inspiration from various cryptographic primitives, including substitution boxes, polynomial arithmetic, and Galois field operations, among others.
This document endeavors to elucidate the intricate mechanisms of the TitanWall Cipher, delving deep into its mathematical foundations, Python and C++ implementations, and algorithmic intricacies.
Additionally, practical code implementations further demonstrate its applicability and performance in real-world scenarios.

1.Byte Substitution Box Generation

Multiplication in $GF(2^8)$:

Given two polynomials $a(x)$ and $b(x)$ in $GF(2^8)$, their product is:

$$c(x) = a(x) \times b(x)$$

To ensure the result remains within $GF(2^8)$, we use:

$$c'(x) = c(x) \mod m(x)$$

And, since we're in $GF(2)$, coefficients are either 0 or 1:

$$c''(x) = c'(x) \mod 2$$

Multiplicative Inverse in $GF(2^8)$:

The multiplicative inverse of a polynomial $a(x)$ in $GF(2^8)$ is found using the Extended Euclidean Algorithm. The goal is:

$$a(x) \times x(x) + m(x) \times y(x) = 1$$

Here, $x(x)$ is the multiplicative inverse of $a(x)$ in $GF(2^8)$.

1.1 Definitions Substitution Box A:

```cpp
/*
        This byte-substitution box: Strict avalanche criterion is satisfied !
        ByteDataSecurityTestData Transparency Order Is: 7.85956
        ByteDataSecurityTestData Nonlinearity Is: 112
        ByteDataSecurityTestData Propagation Characteristics Is: 8
        ByteDataSecurityTestData Delta Uniformity Is: 4
        ByteDataSecurityTestData Robustness Is: 0.984375
        ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 10.3062
        ByteDataSecurityTestData Absolute Value Indicatorer Is: 32
        ByteDataSecurityTestData Sum Of Square Value Indicator Is: 67584
        ByteDataSecurityTestData Algebraic Degree Is: 8
        ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
*/
static constexpr std::array<std::uint8_t, 256> ByteSubstitutionBoxA
{
        0xE2, 0x4E, 0x54, 0xFC, 0x94, 0xC2, 0x4A, 0xCC, 0x62, 0x0D, 0x6A, 0x46, 0x3C, 0x4D, 0x8B, 0xD1,
        0x5E, 0xFA, 0x64, 0xCB, 0xB4, 0x97, 0xBE, 0x2B, 0xBC, 0x77, 0x2E, 0x03, 0xD3, 0x19, 0x59, 0xC1,
        0x1D, 0x06, 0x41, 0x6B, 0x55, 0xF0, 0x99, 0x69, 0xEA, 0x9C, 0x18, 0xAE, 0x63, 0xDF, 0xE7, 0xBB,
        0x00, 0x73, 0x66, 0xFB, 0x96, 0x4C, 0x85, 0xE4, 0x3A, 0x09, 0x45, 0xAA, 0x0F, 0xEE, 0x10, 0xEB,
        0x2D, 0x7F, 0xF4, 0x29, 0xAC, 0xCF, 0xAD, 0x91, 0x8D, 0x78, 0xC8, 0x95, 0xF9, 0x2F, 0xCE, 0xCD,
        0x08, 0x7A, 0x88, 0x38, 0x5C, 0x83, 0x2A, 0x28, 0x47, 0xDB, 0xB8, 0xC7, 0x93, 0xA4, 0x12, 0x53,
        0xFF, 0x87, 0x0E, 0x31, 0x36, 0x21, 0x58, 0x48, 0x01, 0x8E, 0x37, 0x74, 0x32, 0xCA, 0xE9, 0xB1,
        0xB7, 0xAB, 0x0C, 0xD7, 0xC4, 0x56, 0x42, 0x26, 0x07, 0x98, 0x60, 0xD9, 0xB6, 0xB9, 0x11, 0x40,
        0xEC, 0x20, 0x8C, 0xBD, 0xA0, 0xC9, 0x84, 0x04, 0x49, 0x23, 0xF1, 0x4F, 0x50, 0x1F, 0x13, 0xDC,
        0xD8, 0xC0, 0x9E, 0x57, 0xE3, 0xC3, 0x7B, 0x65, 0x3B, 0x02, 0x8F, 0x3E, 0xE8, 0x25, 0x92, 0xE5,
        0x15, 0xDD, 0xFD, 0x17, 0xA9, 0xBF, 0xD4, 0x9A, 0x7E, 0xC5, 0x39, 0x67, 0xFE, 0x76, 0x9D, 0x43,
        0xA7, 0xE1, 0xD0, 0xF5, 0x68, 0xF2, 0x1B, 0x34, 0x70, 0x05, 0xA3, 0x8A, 0xD5, 0x79, 0x86, 0xA8,
        0x30, 0xC6, 0x51, 0x4B, 0x1E, 0xA6, 0x27, 0xF6, 0x35, 0xD2, 0x6E, 0x24, 0x16, 0x82, 0x5F, 0xDA,
        0xE6, 0x75, 0xA2, 0xEF, 0x2C, 0xB2, 0x1C, 0x9F, 0x5D, 0x6F, 0x80, 0x0A, 0x72, 0x44, 0x9B, 0x6C,
        0x90, 0x0B, 0x5B, 0x33, 0x7D, 0x5A, 0x52, 0xF3, 0x61, 0xA1, 0xF7, 0xB0, 0xD6, 0x3F, 0x7C, 0x6D,
        0xED, 0x14, 0xE0, 0xA5, 0x3D, 0x22, 0xB3, 0xF8, 0x89, 0xDE, 0x71, 0x1A, 0xAF, 0xBA, 0xB5, 0x81
};
```

1.2 Substitution Box A Generation:

Attention:
In our recent analysis, we revisited the generation process of Substitution Box A.
While the underlying principles of its construction were evident, mirroring those found in other cryptographic implementations, the exact parameters that were used in its original generation remained absent from our records.

It is worth noting that the principles of Substitution Box A generation, particularly in the context of the Galois Field GF(2^8) and polynomial arithmetic, are consistent across various cryptographic algorithms. However, the specific polynomials, seeds, and other parameters can vary, leading to different S-box outputs.

In our rigorous endeavor to reconstruct the generation process of Substitution Box A within the TitanWall block cipher, our team strictly adhered to time-honored cryptographic principles.
This pursuit, while enlightening, provided us with insights into potential generation methodologies and led to the derivation of plausible S-boxes.
However, it fell short of reproducing the authentic Substitution Box A.
It's of paramount importance to highlight that this was our maiden encounter with the differential power analysis parameters, particularly those possessing significant cryptographic attributes of this box.
Despite channeling extensive efforts to pinpoint and restore the exact parameters instrumental in crafting the original Substitution Box A, our attempts were met with challenges.
The inherent value and unparalleled uniqueness of Substitution Box A underscore its irreplaceability in our cryptographic research and applications.
Instead of forsaking it amidst these challenges, we are steadfast in our commitment to transparency and ethical standards.
Thus, we openly acknowledge our limitations in retrieving the foundational generation parameters. We earnestly seek understanding and respect for our unwavering dedication to uphold the sanctity of cryptographic principles, Kerckhoffs's principle being paramount among them.
It's imperative to draw a clear demarcation between the 'cryptographic attribute parameters that were subjected to analysis for Substitution Box A' and the 'parameters intrinsically employed in the construction of Substitution Box A'.
The two, while closely related, bear distinct connotations and implications in this scholarly discourse.

1. **Field**: Operations are performed in the Galois Field $GF(2^8)$.
2. **Modulus Polynomial**: $m(x) = x^8 + x^5 + x^4 + x^3 + x^2 + x + 1$. The value `0x13F` : This must be an irreducible polynomial.
3. **Forward Seed Polynomial**: $poly_{forward}(x) = x^7 + x^5 + x^4 + x^3 + x^2 + x + 1$.
4. **Backward Seed Polynomial**: $poly_{backward}(x) = x^7 + x^6 + x^5 + x^3 + x + 1$.
5. **Fixed Bytes**: `ByteForwardFixed = 0xE2` and `ByteBackwardFixed = 0xB8` .

Generated binary transformation matrix $GF(2^8)$:

$TransformationMatrix_{forward} = \text{init\_byte\_transform\_matrix}(poly_{backward})$
$TransformationMatrix_{backward} = \text{init\_byte\_transform\_matrix}(poly_{forward})$

$$TransformationMatrix_{forward} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

$$TransformationMatrix_{backward} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Algebraic Formulation
For the forward Substitution-box:

$$\text{Substitution-box}[byte] = byte\left(TransformationMatrix_{backward} \times binaryvector(byte^{-1}_{\pmod{m(x)}}) \oplus ByteForwardFixed\right)$$

For the backward Substitution-box:

$$\text{Substitution-box}^{-1}[byte] = byte\left((TransformationMatrix_{forward} \times binaryvector(byte) \oplus ByteBackwardFixed)^{-1}_{\pmod{m(x)}}\right)$$

ByteForwardFixed: `0xE2`
ByteBackwardFixed: `0xB8`

The forward fixed polynomial byte value and the reverse fixed polynomial byte value should satisfy the following relationship:

$$TransformationMatrix_{forward} \times_{\pmod{m(x)}} ByteForwardFixed = ByteBackwardFixed$$

$$TransformationMatrix_{backward} \times_{\pmod{m(x)}} ByteBackwardFixed = ByteForwardFixed$$

Where:

- $byte^{-1}_{\pmod{m(x)}}$ is the multiplicative inverse of $byte$ in $GF(2^8)$ with respect to $m(x)$.
- $TransformationMatrix_{forward}$ is the transformation matrix constructed from $poly_{forward}(x)$.
- $TransformationMatrix_{backward}$ is the transformation matrix constructed from $poly_{backward}(x)$.

Python Implementation:

```python
import numpy as np

def init_byte_transform_matrix(poly_init):
        column_number = len(poly_init)
        matrix = np.zeros((column_number, column_number), dtype=int)

        for cntY in range(column_number):
                for cntX in range(column_number):
                        i_actual_pos = (cntX + cntY) % column_number
                        matrix[cntY][i_actual_pos] = poly_init[cntX]

        return matrix

poly_forward_seed = [1, 1, 1, 1, 1, 1, 0, 1]
poly_backward_seed = [1, 1, 0, 1, 1, 1, 1, 1]

matrix_forward = init_byte_transform_matrix(poly_backward_seed)
matrix_backward = init_byte_transform_matrix(poly_forward_seed)

print("\nMatrix for Forward Seed:")
print(matrix_forward)
print("\nMatrix for Backward Seed:")
print(matrix_backward)

def is_identity(matrix):
        identity = np.eye(len(matrix), dtype=int)
        return np.array_equal(matrix, identity)

def check_matrices_product_identity(matrix1, matrix2):
        product = np.matmul(matrix1, matrix2)
        return is_identity(product)

# Multiply the matrices in the binary field
product = np.mod(np.matmul(matrix_forward, matrix_backward), 2)

# Check if the product is the identity matrix
identity = np.eye(8, dtype=int)
is_identity = np.array_equal(product, identity)

print("\nProduct:")
print(product)
print("\nIs the product the identity matrix?\n", is_identity)

def multiply_ints_as_polynomials(x, y):
        z = 0
        while x != 0:
                if x & 1 == 1:
                        z ^= y
                y <<= 1
                x >>= 1
        return z

def number_bits(x):
        nb = 0
        while x != 0:
                nb += 1
                x >>= 1
        return nb

def mod_int_as_polynomial(x, m):
        nbm = number_bits(m)
        while True:
                nbx = number_bits(x)
                if nbx < nbm:
                        return x
                mshift = m << (nbx - nbm)
                x ^= mshift

def galois_field_multiplication(x, y, mod_poly):
        z = multiply_ints_as_polynomials(x, y)
        return mod_int_as_polynomial(z, mod_poly)

# Polynomial division for a byte, returns quotient (a/b)
def polynomial_division(dividend, divisor):
        dividend_length = number_bits(dividend)
        divisor_length = number_bits(divisor)
        length_difference = dividend_length - divisor_length

        if dividend < divisor:  # If dividend is less than divisor
                if length_difference == 0:  # If both numbers have the same length, then the quotient is 1 and the remainder is the XOR result of the two
                        return (1, dividend ^ divisor)
                else:
                        return (0, dividend)  # If the length of the dividend is less than the divisor, then the quotient is 0 and the remainder is dividend

        highest_bit = 1
        highest_bit <<= (dividend_length - 1)
        divisor <<= length_difference  # Extend the length of divisor to match the length of dividend

        quotient = 0
        remainder = 0

        for i in range(length_difference):
                quotient <<= 1  # Shift quotient left by one bit, then determine if the new bit is 0 or 1 in the next loop
                if (highest_bit & dividend):  # If the highest bit of dividend is 1, it means the dividend can be divided by the divisor
                        quotient ^= 1  # If the above condition is true, then the quotient should be 1, so XOR with 1
                        dividend ^= divisor
                else:
                        dividend ^= 0
                highest_bit >>= 1  # Shift right by one bit for each iteration
                divisor >>= 1  # Same effect for divisor
        quotient <<= 1  # Shift quotient left by one bit, the new bit is 0
        if dividend < divisor:
                remainder = dividend
        else:
                quotient ^= 1
                remainder = dividend ^ divisor

        return quotient, remainder

# Extended Euclidean algorithm to find the multiplicative inverse of b in GF(2^8)
def galois_field_inverse(element, modulus_poly):
        if(element == 0):
                return 0

        initial_modulus = modulus_poly
        initial_element = element
        initial_multiplier = 0
        multiplier = 1
        quotient, intermediate_remainder = polynomial_division(initial_modulus, initial_element)

        next_multiplier = initial_multiplier ^ multiply_ints_as_polynomials(quotient, multiplier)
        while(1):
                if(intermediate_remainder == 0):
                        break
                initial_modulus = initial_element
                initial_element = intermediate_remainder
                quotient, intermediate_remainder = polynomial_division(initial_modulus, initial_element)
                initial_multiplier = multiplier
                multiplier = next_multiplier
                next_multiplier = initial_multiplier ^ multiply_ints_as_polynomials(quotient, multiplier)
        return multiplier

def dot_product(x, y):
        z = x & y
        dot = 0
        while z != 0:
                dot ^= z & 1
                z >>= 1
        return dot
```

```
def affine_transformation(A, x, v):
        y = 0
        for i in reversed(range(8)):
                row = (A >> 8 * i) & 0xff
                bit = dot_product(row, x)
                y ^= (bit << i)
        return y ^ v

def generate_byte(x, mod_poly, A, v):
        byte_gf_inverse = galois_field_inverse(x, mod_poly)
        # y = Ax + v
        return affine_transformation(A, byte_gf_inverse, v)

def generate_inverse_byte(y, mod_poly, A, v):
        # x = A^{-1}y + v^{-1}
        x = affine_transformation(A, y, v)
        return galois_field_inverse(x, mod_poly)

def check_sboxes_inverses(forward_sbox, backward_sbox):
        for x in range(256):
                if backward_sbox[forward_sbox[x]] != x or forward_sbox[backward_sbox[x]] != x:
                        return False
        return True

def make_forward_box(mod_poly, A, fixed_byte):
        positive_sbox = []
        for cntY in range(16):
                for cntX in range(16):
                        byte_source = (cntY << 4) | cntX
                        if byte_source == 0:
                                positive_sbox.append(fixed_byte)
                                continue
                        transformed_byte = generate_byte(byte_source, mod_poly, A, fixed_byte)
                        positive_sbox.append(transformed_byte)
        return positive_sbox

def make_backward_box(mod_poly, A, fixed_byte):
        reverse_sbox = []
        for cntY in range(16):
                for cntX in range(16):
                        byte_source = (cntY << 4) | cntX
                        transformed_byte = generate_inverse_byte(byte_source, mod_poly, A, fixed_byte)
                        if transformed_byte == 0:
                                reverse_sbox.append(0)
                                continue
                        reverse_sbox.append(transformed_byte)
        return reverse_sbox
```

```python
def matrix_to_binary_string(matrix):
        return ''.join([''.join(map(str, row)) for row in matrix])

# Convert matrices to binary strings
binary_string_forward = matrix_to_binary_string(matrix_forward)
binary_string_backward = matrix_to_binary_string(matrix_backward)

# Print the binary strings
print("Binary string for matrix_forward:", binary_string_forward)
print("Binary string for matrix_backward:", binary_string_backward)

# Parameters
ModulusPolynomial = int('100111111', 2)  # m(x) = x^8 + x^5 + x^4 + x^3 + x^2 + x + 1
TransformationMatrixForward = int(binary_string_forward, 2)
TransformationMatrixBackward = int(binary_string_backward, 2)
ByteForwardFixed = 0xE2
ByteBackwardFixed = 0xB8

# Generate the unique S-boxes
unique_forward_sbox = make_forward_box(ModulusPolynomial, TransformationMatrixForward, ByteForwardFixed)
unique_backward_sbox = make_backward_box(ModulusPolynomial, TransformationMatrixBackward, ByteBackwardFixed)

# Print the unique S-boxes in hexadecimal format
byte_counter = 0
print("Unique Forward S-box:")
for byte_val in unique_forward_sbox:
        print(format(byte_val, "02x"), end=', ')
        byte_counter += 1
        if byte_counter == 16:
                print('\n')
                byte_counter = 0
print()

byte_counter = 0
print("\nUnique Backward S-box:")
for byte_val in unique_backward_sbox:
        print(format(byte_val, "02x"), end=', ')
        byte_counter += 1
        if byte_counter == 16:
                print('\n')
                byte_counter = 0
print()

# Check if the S-boxes are inverses of each other
```

```
are_inverses = check_sboxes_inverses(unique_forward_sbox, unique_backward_sbox)
print("Are the S-boxes inverses of each other?", are_inverses)
```

Program Result:

```
Matrix for Forward Seed:
[[1 1 0 1 1 1 1 1]
 [1 1 1 0 1 1 1 1]
 [1 1 1 1 0 1 1 1]
 [1 1 1 1 1 0 1 1]
 [1 1 1 1 1 1 0 1]
 [1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1]
 [1 0 1 1 1 1 1 1]]

Matrix for Backward Seed:
[[1 1 1 1 1 1 0 1]
 [1 1 1 1 1 1 1 0]
 [0 1 1 1 1 1 1 1]
 [1 0 1 1 1 1 1 1]
 [1 1 0 1 1 1 1 1]
 [1 1 1 0 1 1 1 1]
 [1 1 1 1 0 1 1 1]
 [1 1 1 1 1 0 1 1]]

Product:
[[1 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 0 0]
 [0 0 1 0 0 0 0 0]
 [0 0 0 1 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 1 0 0]
 [0 0 0 0 0 0 1 0]
 [0 0 0 0 0 0 0 1]]

Is the product the identity matrix?
 True
Binary string for matrix_forward: 1101111111101111111101111111101111111101111111100111111110111111
Binary string for matrix_backward: 1111110111111110011111111011111111011111111011111111011111111011
Unique Forward S-box:
e2, 19, 9c, b6, 5e, d3, c8, c5, bc, ff, f9, 6c, 74, 1c, f2, f0,

cd, ec, ef, fc, 6f, 34, 26, 4a, 2a, 65, 9d, e9, 69, 98, eb, f3,

75, 12, e5, a9, 64, b5, 6e, c9, a7, f1, 89, 9f, 80, e6, 35, c3,

05, b8, a2, 1a, 5d, 17, e4, 7f, 27, 78, 5c, b9, 66, 8e, 6a, 52,

29, 8f, 9a, 08, 61, 10, 47, c6, a1, ce, 49, 94, a4, d6, 77, ba,

40, 24, e8, 2b, d4, 48, df, a3, 50, 6b, e0, c2, 8a, 3c, 72, 4e,
92, 96, cf, b2, 41, db, 1d, ad, 3d, 04, 9b, cb, 62, 7d, 2c, 2d,

83, fd, af, 13, 3e, fa, cc, e3, 23, 88, 57, 11, a6, 36, 39, 8b,

84, 3a, 54, d0, de, 09, 14, 51, a0, ed, 18, c4, b3, 4c, 73, 03,

c0, 16, f4, ea, b4, 4f, d9, fe, c1, 1f, f8, d8, ab, 87, 4d, 15,

30, 4b, 02, 31, e7, d2, 06, dc, 7a, da, b7, 2f, 7c, 90, 42, 0a,

bb, 58, a5, f6, 60, b0, 71, 68, 55, 8d, 0e, bd, aa, 07, 37, 3f,

59, 2e, 5b, 3b, f7, 46, ca, 97, 33, 01, 7e, 95, 9e, 82, 45, b1,

0d, 28, 91, 1e, dd, be, f5, 0c, 21, bf, ae, fb, 85, 32, 86, 53,

d1, 79, 6d, 63, c7, 93, 99, a8, 8c, 20, ee, 67, 76, 00, e1, d5,

81, 44, d7, 5f, 38, 5a, 1b, 70, 43, 22, 0b, 25, 0f, ac, 56, 7b,



Unique Backward S-box:
ed, c9, a2, 8f, 69, 30, a6, bd, 43, 85, af, fa, d7, d0, ba, fc,

45, 7b, 21, 73, 86, 9f, 91, 35, 8a, 01, 33, f6, 0d, 66, d3, 99,

e9, d8, f9, 78, 51, fb, 16, 38, d1, 40, 18, 53, 6e, 6f, c1, ab,

a0, a3, dd, c8, 15, 2e, 7d, be, f4, 7e, 81, c3, 5d, 68, 74, bf,

50, 64, ae, f8, f1, ce, c5, 46, 55, 4a, 17, a1, 8d, 9e, 5f, 95,

58, 87, 3f, df, 82, b8, fe, 7a, b1, c0, f5, c2, 3a, 34, 04, f3,

b4, 44, 6c, e3, 24, 19, 3c, eb, b7, 1c, 3e, 59, 0b, e2, 26, 14,

f7, b6, 5e, 8e, 0c, 20, ec, 4e, 39, e1, a8, ff, ac, 6d, ca, 37,

2c, f0, cd, 70, 80, dc, de, 9d, 79, 2a, 5c, 7f, e8, b9, 3d, 41,

ad, d2, 60, e5, 4b, cb, 61, c7, 1d, e6, 42, 6a, 02, 1a, cc, 2b,

88, 48, 32, 57, 4c, b2, 7c, 28, e7, 23, bc, 9c, fd, 67, da, 72,

b5, cf, 63, 8c, 94, 25, 03, aa, 31, 3b, 4f, b0, 08, bb, d5, d9,
90, 98, 5b, 2f, 8b, 07, 47, e4, 06, 27, c6, 6b, 76, 10, 49, 62,

83, e0, a5, 05, 54, ef, 4d, f2, 9b, 96, a9, 65, a7, d4, 84, 56,

5a, ee, 00, 77, 36, 22, 2d, a4, 52, 1b, 93, 1e, 11, 89, ea, 12,

0f, 29, 0e, 1f, 92, d6, b3, c4, 9a, 0a, 75, db, 13, 71, 97, 09,


Are the S-boxes inverses of each other? True


...Program finished with exit code 0
Press ENTER to exit console.
```

1.3 Definitions Substitution Box B:

```cpp
/*
	This byte-substitution box: Strict avalanche criterion is satisfied !
	ByteDataSecurityTestData Transparency Order Is: 7.85221
	ByteDataSecurityTestData Nonlinearity Is: 112
	ByteDataSecurityTestData Propagation Characteristics Is: 8
	ByteDataSecurityTestData Delta Uniformity Is: 4
	ByteDataSecurityTestData Robustness Is: 0.984375
	ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 9.23235
	ByteDataSecurityTestData Absolute Value Indicatorer Is: 32
	ByteDataSecurityTestData Sum Of Square Value Indicator Is: 67584
	ByteDataSecurityTestData Algebraic Degree Is: 8
	ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
*/
static constexpr std::array<std::uint8_t, 256> ByteSubstitutionBoxB
{
	0xE2, 0x0D, 0x3E, 0x94, 0x1D, 0x02, 0x48, 0x71, 0x1C, 0x93, 0xA8, 0x69, 0xB7, 0x90, 0xAA, 0x5C,
	0x37, 0x5A, 0xDB, 0x75, 0xFD, 0x64, 0x8D, 0xD3, 0x49, 0x12, 0xCB, 0xE0, 0xC6, 0x9A, 0x16, 0xDF,
	0x33, 0x08, 0xAE, 0xD0, 0xFF, 0xB3, 0x29, 0x34, 0x56, 0xE9, 0x20, 0x7F, 0x44, 0x2F, 0xFA, 0xDC,
	0x9C, 0x4E, 0x8A, 0x46, 0xDD, 0x42, 0xD9, 0x6A, 0x70, 0xF3, 0xF5, 0x8C, 0x09, 0x72, 0x7C, 0x9F,
	0xB0, 0x1B, 0x96, 0x62, 0x45, 0x10, 0xEA, 0xA0, 0x6D, 0xA7, 0xCA, 0x3F, 0xAC, 0x0B, 0x23, 0x57,
	0x28, 0x5B, 0xF7, 0xB4, 0x82, 0x9E, 0x17, 0xEC, 0x31, 0xA9, 0x14, 0xA2, 0xC5, 0x1E, 0x6C, 0x4F,
	0x4D, 0x55, 0x0F, 0xBB, 0xD7, 0xC0, 0x0A, 0xE1, 0x47, 0xAF, 0x89, 0x26, 0xC4, 0xCD, 0x9D, 0x2C,
	0x81, 0x3B, 0xEB, 0xF9, 0x53, 0x5E, 0x6F, 0x95, 0xBD, 0x27, 0xBA, 0xFB, 0x07, 0xA5, 0x5D, 0xED,
	0xDA, 0x2A, 0xA4, 0x99, 0x73, 0x01, 0x98, 0x13, 0x1A, 0xA3, 0xB1, 0xBF, 0xE7, 0x15, 0xF8, 0x78,
	0x0E, 0x9B, 0x6B, 0x67, 0xF6, 0xD8, 0x36, 0x61, 0x7E, 0xFC, 0x86, 0x40, 0x92, 0x52, 0x03, 0x97,
	0x87, 0xB9, 0x85, 0x8E, 0x68, 0x06, 0x59, 0xC9, 0xD2, 0xD1, 0x76, 0xC1, 0x22, 0x39, 0x5F, 0xE3,
	0x8B, 0xA6, 0xD6, 0x2B, 0x32, 0xBE, 0xC3, 0xE6, 0x60, 0x7A, 0x0C, 0xF4, 0x25, 0x41, 0x24, 0x54,
	0x1F, 0xF0, 0x38, 0xAB, 0x05, 0x83, 0xCF, 0x58, 0x79, 0x3C, 0xC8, 0x7D, 0xAD, 0x51, 0xF2, 0xB2,
	0x21, 0x43, 0x6E, 0xEF, 0xC7, 0x18, 0x3A, 0x88, 0x4B, 0x2E, 0x65, 0xDE, 0x66, 0xB6, 0x04, 0x30,
	0xC2, 0x4A, 0xB5, 0x19, 0xCC, 0xFE, 0xD5, 0x84, 0x80, 0x8F, 0x2D, 0xE8, 0x35, 0xF1, 0x63, 0x4C,
	0x77, 0x91, 0x11, 0xB8, 0xE4, 0xCE, 0xEE, 0xA1, 0x00, 0xD4, 0x50, 0xBC, 0x3D, 0x7B, 0x74, 0xE5
};
```

1.4 Substitution Box B Generation:

1. **Field**: Operations are performed in the Galois Field $GF(2^8)$.
2. **Modulus Polynomial**: $m(x) = x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1$. The value `0x1F9` : This polynomial is irreducible in $GF(2^8)$.
3. **Transformation Matrix**:

$$TransformationMatrix = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Fixed Byte: `ByteFixed = 0xE2` .

Algebraic Formulation

For Substitution Box B:

$$\text{S-box}[byte] = byte \left(TransformationMatrix \times binaryvector(byte^{-1}_{\pmod{m(x)}}) \oplus ByteFixed \right)$$

Where:

- $byte^{-1}_{\pmod{m(x)}}$ is the multiplicative inverse of $byte$ in $GF(2^8)$ with respect to $m(x)$.
- $TransformationMatrix$ is the matrix used for the affine transformation.
- $\oplus$ represents the bitwise XOR operation.

The combination of multiplicative inversion and affine transformation ensures the non-linearity of the S-box, which is crucial for cryptographic strength.

C++ Implementation:

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <unordered_set>

// Define polynomial modulus and GF(2^8) matrix
const int POLY_MOD = 0x1F9;
const int MATRIX[8][8] = {
        {1, 1, 1, 1, 1, 1, 0, 0},
        {1, 1, 1, 0, 1, 1, 1, 0},
        {1, 1, 1, 1, 0, 1, 1, 1},
        {1, 1, 1, 1, 1, 0, 1, 1},
        {0, 1, 1, 1, 1, 1, 0, 1},
        {1, 0, 1, 1, 1, 1, 1, 1},
        {1, 1, 0, 1, 1, 1, 1, 1},
        {1, 1, 1, 0, 1, 1, 1, 1}
};

// Multiplicative element on GF(2^8) finite field
int gf_multiplication(int a, int b) {
        int result = 0;
        while (b) {
                if (b & 1) result ^= a;
                a <<= 1;
                if (a & 0x100) a ^= POLY_MOD;
                b >>= 1;
        }
        return result;
}

// Multiplicative inverse element on GF(2^8) finite field
int gf_multiplication_inverse(int number) {
        if(!number)
                return 0;
        for (int i = 1; i < 256; ++i)
        {
                if(gf_multiplication(number, i) == 0x01)
                {
                        return i;
                }
        }
}


int binary_matrix_multiplication(int num) {
        int result = 0;
        for (int i = 0; i < 8; ++i) {
                int bit = 0;
                for (int j = 0; j < 8; ++j) {
                        bit ^= (num >> j & 1) & MATRIX[i][j];
                }
                result |= (bit << i);
        }
        return result;
}

int main() {
        int SubstitutionBox[256] = { 0 };

        std::cout << "const int SubstitutionBox[256] = {" << std::endl;
        for (int i = 0; i < 256; ++i) {
                int byte_gf_inverse = gf_multiplication_inverse(i);
                // 仿射变换 Affine Transform
                // y = Ax + v
                int sbox_value = binary_matrix_multiplication(byte_gf_inverse) ^ 0xE2;
                SubstitutionBox[i] = sbox_value;
                std::cout << "0x" << std::uppercase << std::setw(2) << std::setfill('0') << std::hex << sbox_value;
                if (i != 255) std::cout << ", ";
                if ((i + 1) % 16 == 0) std::cout << std::endl;
        }
        std::cout << "};" << std::endl;

        //Hash set
        std::unordered_set<int> unique_elements;
        for (int i = 0; i < 256; ++i) {
                unique_elements.insert(SubstitutionBox[i]);
        }

        if (unique_elements.size() == 256) {
                std::cout << "All elements are unique." << std::endl;
        } else {
                std::cout << "There are duplicate elements." << std::endl;
        }

        return 0;
}
```

Program Result:

```
const int SubstitutionBox[256] = {
0xE2, 0x0D, 0x3E, 0x94, 0x1D, 0x02, 0x48, 0x71, 0x1C, 0x93, 0xA8, 0x69, 0xB7, 0x90, 0xAA, 0x5C,
0x37, 0x5A, 0xDB, 0x75, 0xFD, 0x64, 0x8D, 0xD3, 0x49, 0x12, 0xCB, 0xE0, 0xC6, 0x9A, 0x16, 0xDF,
0x33, 0x08, 0xAE, 0xD0, 0xFF, 0xB3, 0x29, 0x34, 0x56, 0xE9, 0x20, 0x7F, 0x44, 0x2F, 0xFA, 0xDC,
0x9C, 0x4E, 0x8A, 0x46, 0xDD, 0x42, 0xD9, 0x6A, 0x70, 0xF3, 0xF5, 0x8C, 0x09, 0x72, 0x7C, 0x9F,
0xB0, 0x1B, 0x96, 0x62, 0x45, 0x10, 0xEA, 0xA0, 0x6D, 0xA7, 0xCA, 0x3F, 0xAC, 0x0B, 0x23, 0x57,
0x28, 0x5B, 0xF7, 0xB4, 0x82, 0x9E, 0x17, 0xEC, 0x31, 0xA9, 0x14, 0xA2, 0xC5, 0x1E, 0x6C, 0x4F,
0x4D, 0x55, 0x0F, 0xBB, 0xD7, 0xC0, 0x0A, 0xE1, 0x47, 0xAF, 0x89, 0x26, 0xC4, 0xCD, 0x9D, 0x2C,
0x81, 0x3B, 0xEB, 0xF9, 0x53, 0x5E, 0x6F, 0x95, 0xBD, 0x27, 0xBA, 0xFB, 0x07, 0xA5, 0x5D, 0xED,
0xDA, 0x2A, 0xA4, 0x99, 0x73, 0x01, 0x98, 0x13, 0x1A, 0xA3, 0xB1, 0xBF, 0xE7, 0x15, 0xF8, 0x78,
0x0E, 0x9B, 0x6B, 0x67, 0xF6, 0xD8, 0x36, 0x61, 0x7E, 0xFC, 0x86, 0x40, 0x92, 0x52, 0x03, 0x97,
0x87, 0xB9, 0x85, 0x8E, 0x68, 0x06, 0x59, 0xC9, 0xD2, 0xD1, 0x76, 0xC1, 0x22, 0x39, 0x5F, 0xE3,
0x8B, 0xA6, 0xD6, 0x2B, 0x32, 0xBE, 0xC3, 0xE6, 0x60, 0x7A, 0x0C, 0xF4, 0x25, 0x41, 0x24, 0x54,
0x1F, 0xF0, 0x38, 0xAB, 0x05, 0x83, 0xCF, 0x58, 0x79, 0x3C, 0xC8, 0x7D, 0xAD, 0x51, 0xF2, 0xB2,
0x21, 0x43, 0x6E, 0xEF, 0xC7, 0x18, 0x3A, 0x88, 0x4B, 0x2E, 0x65, 0xDE, 0x66, 0xB6, 0x04, 0x30,
0xC2, 0x4A, 0xB5, 0x19, 0xCC, 0xFE, 0xD5, 0x84, 0x80, 0x8F, 0x2D, 0xE8, 0x35, 0xF1, 0x63, 0x4C,
0x77, 0x91, 0x11, 0xB8, 0xE4, 0xCE, 0xEE, 0xA1, 0x00, 0xD4, 0x50, 0xBC, 0x3D, 0x7B, 0x74, 0xE5
};
All elements are unique.
```

2. Mathematical Functions

Algorithm:

```
Class CryptographicFunctions:

        Data:
                /*
                        Fibonacci numbers:
                        123581321345589144 = 0x1B70C8E97AD5F98

                        π binary:
                        Binary((π - 3) × 10^32) = 2611923443488327891 = 0x243F6A8885A308D3

                        ϕ binary:
                        Binary((ϕ - 1) × 10^32) = 11400714819323198485 = 0x9E3779B97F4A7C15

                        e binary:
                        Binary((e - 2) × 10^32) = 13249961062380153450 = 0xB7E151628AED2A6A
                */
                MathMagicNumbers = [0x01B70C8E,0x243F6A88,0x9E3779B9,0xB7E15162]
                ByteSubstitutionBoxA = [...]
                ByteSubstitutionBoxB = [...]
                KDSB = Array of size 128 of 32-bit numbers

        Function Bits32RotateLeft(number, bit):
                return (number << (bit % 32)) OR (number >> (32 - (bit % 32)))

        Function Bits32RotateRight(number, bit):
                return (number >> (bit % 32)) OR (number << (32 - (bit % 32)))

        Function FF(A, B, C, Index, ArraySize):
                if Index < (ArraySize / 4) * 3:
                        return A XOR B XOR C
                else:
                        return (A AND B) OR (A AND C) OR (B AND C)

        Function GG(A, B, C, Index, ArraySize):
                if Index < (ArraySize / 4) * 3:
                        return A XOR B XOR C
                else:
                        return (A AND B) OR (NOT A AND C)

        Function L(number):
                return number XOR Bits32RotateLeft(number, 2) XOR Bits32RotateLeft(number, 10) XOR Bits32RotateLeft(number, 18) XOR Bits32RotateLeft(number, 24)

        Function L2(number):
                return number XOR Bits32RotateLeft(number, 13) XOR Bits32RotateLeft(number, 23)

        Function NLFSR(Register):
            ... (operations inspired by KeeLoq algorithm)

        Function MixWithAddSubtract(Counter, RandomIndex):
            ... (operations involving MathMagicNumbers and KDSB)

        Function RandomAccessMix(Counter, RandomIndex):
            ... (operations involving MathMagicNumbers and KDSB)

        Function ComplexMix(Counter, RandomIndex):
            ... (operations involving MathMagicNumbers, KDSB, GG, FF, L, and L2)

        Function KeySchedule(KeyBytes):
            ... (key expansion using ByteSubstitutionBoxA, ByteSubstitutionBoxB, MixWithAddSubtract, RandomAccessMix, and ComplexMix)

    EndClass
```

Where $KDSB[number] = KDSB_{index}$

2.1. Bitwise Rotation

Left Rotation:

$$\text{Bits32RotateLeft}(number, bit) = (number \ll (bit \mod 32)) \vee (number \gg (32 - (bit \mod 32)))$$

Right Rotation:

$$\text{Bits32RotateRight}(number, bit) = (number \gg (bit \mod 32)) \vee (number \ll (32 - (bit \mod 32)))$$

Where $\ll$ denotes left bitwise shift, $\gg$ denotes right bitwise shift, $\lll$ denotes left bitwise rotate, $\ggg$ denotes right bitwise rotate, and $\vee$ denotes bitwise OR.

2.2. Nonlinear Boolean Function

$$\text{FF}(A, B, C, Index, ArraySize) = \begin{cases} A \oplus B \oplus C & \text{if } Index < \frac{3 \times ArraySize}{4} \\ (A \wedge B) \vee (A \wedge C) \vee (B \wedge C) & \text{otherwise} \end{cases}$$

$$\text{GG}(A, B, C, Index, ArraySize) = \begin{cases} A \oplus B \oplus C & \text{if } Index < \frac{3 \times ArraySize}{4} \\ (A \wedge B) \vee (\neg A \wedge C) & \text{otherwise} \end{cases}$$

Where $\oplus$ denotes bitwise XOR, $\wedge$ denotes bitwise AND, and $\vee$ denotes bitwise OR, $\neg$ denotes bitwise NOT.

2.3. Linear Transform Function

$$L(number) = number \oplus (number \lll 2) \oplus (number \lll 10) \oplus (number \lll 18) \oplus (number \lll 24)$$

$$L2(number) = number \oplus (number \lll 13) \oplus (number \lll 23)$$

2.4. NLFSR (Non-Linear Feedback Shift Register)

The NLFSR function operates on a 32-bit register, applying bitwise operations and feedback to generate a transformed register.
The exact operations are inspired by the KeeLoq algorithm and involve combinations of bit positions.

Feedback Function Definitions with Boolean Algebra:

- $\text{FeedBack0} = (B \oplus G) \oplus (A \oplus F)$
- $\text{FeedBack1} = (A \land D) \oplus (A \land G)$
- $\text{FeedBack2} = (B \land C) \oplus (B \land D) \oplus (B \land E)$
- $\text{FeedBack3} = (E \land F) \land (D \land F) \land (C \land F)$
- $\text{FeedBack4} = (F \land G) \land (E \land G) \land (D \land G)$
- $\text{FeedBack5} = (A \land B \land G) \oplus (A \land D \land G) \oplus (A \land F \land G)$
- $\text{FeedBack6} = (A \land B \land C) \oplus (B \land C \land D) \oplus (C \land D \land E) \oplus (D \land E \land F) \oplus (E \land F \land G)$
- $\text{FeedBack7} = (A \land C \land E \land G) \oplus (B \land D \land F)$

Algorithm:

```
Function NLFSR(Register: 32-bit integer):
        for NLFSR_ROUND from 0 to 63:

                // PrimerNumbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61
                // Extract specific bit positions from Register
                A = BitAt(Register, 31)
                B = BitAt(Register, 28) // Note: 29 - 1 = 28
                C = BitAt(Register, 23)
                D = BitAt(Register, 17) // Note: 19 - 2 = 17
                E = BitAt(Register, 13)
                F = BitAt(Register, 4) // Note: 7 - 3 = 4
                G = BitAt(Register, 1)
                H = BitAt(Register, 16) XOR LSB(Register)

                // Compute feedback using combinations of extracted bits
                FeedBack0 = (B XOR G) XOR (A XOR F)
                FeedBack1 = (A AND D) XOR (A AND G)
                FeedBack2 = (B AND C) XOR (B AND D) XOR (B AND E)
                FeedBack3 = (E AND F) AND (D AND F) AND (C AND F)
                FeedBack4 = (F AND G) AND (E AND G) AND (D AND G)
                FeedBack5 = (A AND B AND G) XOR (A AND D AND G) XOR (A AND F AND G)
                FeedBack6 = (A AND B AND C) XOR (B AND C AND D) XOR (C AND D AND E) XOR (D AND E AND F) XOR (E AND F AND G)
                FeedBack7 = (A AND C AND E AND G) XOR (B AND D AND F)

                FeedBack = H XOR FeedBack0 XOR FeedBack1 XOR FeedBack2 XOR FeedBack3 XOR FeedBack4 XOR FeedBack5 XOR FeedBack6 XOR FeedBack7

                // Right shift Register by 1 bit and inject Feedback into the leftmost bit
                Register = (Register >> 1) OR (FeedBack << 31)

        EndFor

        Return Register
EndFunction

Function BitAt(Number, Position):
        Return (Number >> Position) AND 1
EndFunction

Function LSB(Number):
        Return Number AND 1
EndFunction
```

2.5 MixWithAddSubtract

Mathematical Formulas:

$$\begin{gathered}\text{RandomIndex} = \text{RandomIndex} \\ \oplus (\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] + \text{MathMagicNumbers}[\text{Counter} \mod 4])\end{gathered}$$

$$\begin{gathered}\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] = \text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] \\ +\text{KDSB}[(\text{Counter}+1) \pmod{\text{KDSB.size()}}] \\ -\text{MathMagicNumbers}[(\text{RandomIndex}+\text{Counter}) \mod 4]\end{gathered}$$

$$\begin{gathered}\text{RandomIndex} = \text{RandomIndex} \\ \oplus (\text{KDSB}[(\text{Counter}+1) \pmod{\text{KDSB.size()}}] + \text{MathMagicNumbers}[(\text{RandomIndex}-\text{Counter}) \pmod 4])\end{gathered}$$

$$\begin{gathered}\text{KDSB}[(\text{Counter}+1) \pmod{\text{KDSB.size()}}] = \text{KDSB}[(\text{Counter}+1) \pmod{\text{KDSB.size()}}] \\ -\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] \\ +\text{MathMagicNumbers}[(\text{RandomIndex}+\text{Counter}) \mod 4]\end{gathered}$$

Algorithm:

```
Function MixWithAddSubtract(Counter, RandomIndex):
        RandomIndex = RandomIndex XOR (KDSB[Counter % KDSB.size()] + MathMagicNumbers[Counter % 4])
        KDSB[Counter % KDSB.size()] += KDSB[(Counter + 1) % KDSB.size()] - MathMagicNumbers[(RandomIndex + Counter) % 4]
        RandomIndex = RandomIndex XOR (KDSB[(Counter + 1) % KDSB.size()] + MathMagicNumbers[(RandomIndex - Counter) % 4])
        KDSB[(Counter + 1) % KDSB.size()] -= KDSB[Counter % KDSB.size()] + MathMagicNumbers[(RandomIndex + Counter) % 4]
EndFunction
```

2.6 RandomAccessMix

Mathematical Formulas:

$$\text{RandomPosition} = \text{RandomIndex} \pmod{\text{KDSB.size()}}$$

$$\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] = \text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] \oplus \text{KDSB}[\text{RandomPosition}]$$

$$\begin{gathered}\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] = \text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] \\ + (\text{KDSB}[\text{Counter} \pmod{\text{KDSB.size()}}] - \text{MathMagicNumbers}[\text{RandomPosition} \mod 4])\end{gathered}$$

Algorithm:

```
Function RandomAccessMix(Counter, RandomIndex):
        RandomPosition = RandomIndex % KDSB.size()
        KDSB[Counter % KDSB.size()] XOR= KDSB[RandomPosition]
        KDSB[Counter % KDSB.size()] += KDSB[Counter % KDSB.size()] - MathMagicNumbers[RandomPosition % 4]
EndFunction
```

2.7 ComplexMix

Mathematical Formulas:

1. Splitting and Processing `RandomIndex` :
   Let RL and RR be the left and right 32 bits of `RandomIndex` respectively.

$$RL = 32Bit(RandomIndex \gg 32)$$

$$RR = 32Bit(RandomIndex \land 0x00000000FFFFFFFF)$$

   Apply the NLFSR transformation:

$$RL' = \mathrm{NLFSR}(RL)$$

$$RR' = \mathrm{NLFSR}(RR)$$

   Combine RL' and RR' to form the new `RandomIndex` :

$$RandomIndex' = (64Bit(RR') \ll 32) \lor 64Bit(RL')$$

2. Updating `KDSB` using Previous Entries:

$$\begin{gathered}KDSB[Counter \mod N] = KDSB[Counter \mod N] \\ \oplus KDSB[(Counter - 2) \mod N] \oplus KDSB[(Counter - 1) \mod N]\end{gathered}$$

   where N is the size of `KDSB` .

3. Applying the Nonlinear Boolean Functions and Linear Transformation:

$$\begin{gathered}KDSB[Counter \mod N] = KDSB[Counter \mod N] \\ +GG(Counter, KDSB[(Counter - 1) \mod N], RandomIndex' \mod 2^{32}, Counter, N)\end{gathered}$$

$$KDSB[Counter \mod N] = KDSB[Counter \mod N] \\ - FF(KDSB[(Counter-3) \mod N], L(KDSB[(Counter-2) \mod N]), KDSB[(Counter-1) \mod N], Counter, N)$$

4. Further Mixing Using `L` , `L2` Functions and `MathMagicNumbers` :

$$RandomIndex'' = RandomIndex' \\ \oplus L(KDSB[Counter \mod N] - MathMagicNumbers[Counter \mod 4])$$

$$KDSB[(Counter+1) \mod N] = KDSB[(Counter+1) \mod N] \\ - L2(KDSB[Counter \mod N] + MathMagicNumbers[RandomIndex'' - Counter \mod 4])$$

$$RandomIndex''' = RandomIndex'' \\ \oplus L2(KDSB[Counter \mod N] - MathMagicNumbers[RandomIndex'' + Counter \mod 4])$$

$$KDSB[(Counter+2) \mod N] = KDSB[(Counter+2) \mod N] \\ + L(KDSB[(Counter+1) \mod N] + MathMagicNumbers[RandomIndex''' - Counter \mod 4])$$

The operations include bitwise XOR, modular addition, the GG, FF, L, and L2 functions, and feedback mechanisms.

Algorithm:

```
Function ComplexMix(Counter, RandomIndex):
        // 1. Split and Process RandomIndex
        RL = 32Bit(RandomIndex >> 32)
        RR = 32Bit(RandomIndex BIT_AND 0x00000000FFFFFFFF)
        RL = NLFSR(RL)
        RR = NLFSR(RR)
        RandomIndex = 64Bit(RR) << 32 BIT_OR 64Bit(RL)

        // 2. Update KDSB using Previous Entries
        N = SIZE_OF(KDSB)
        KDSB[Counter MOD N] = KDSB[Counter MOD N] XOR KDSB[(Counter - 2) MOD N] XOR KDSB[(Counter - 1) MOD N]

        // 3. Apply Nonlinear Boolean Functions and Linear Transformation
        KDSB[Counter MOD N] = KDSB[Counter MOD N] + GG(Counter, KDSB[(Counter - 1) MOD N], RandomIndex MOD MAX_UINT32, Counter, N)
        KDSB[Counter MOD N] = KDSB[Counter MOD N] - FF(KDSB[(Counter - 3) MOD N], L(KDSB[(Counter - 2) MOD N]), KDSB[(Counter - 1) MOD N], Counter, N)

        // 4. Mix Using L and L2 Functions and MathMagicNumbers
        RandomIndex = RandomIndex XOR L(KDSB[Counter MOD N] - MathMagicNumbers[Counter MOD 4])
        KDSB[(Counter + 1) MOD N] = KDSB[(Counter + 1) MOD N] - L2(KDSB[Counter MOD N] + MathMagicNumbers[RandomIndex - Counter MOD 4])
        RandomIndex = RandomIndex XOR L2(KDSB[Counter MOD N] - MathMagicNumbers[RandomIndex + Counter MOD 4])
        KDSB[(Counter + 2) MOD N] = KDSB[(Counter + 2) MOD N] + L(KDSB[(Counter + 1) MOD N] + MathMagicNumbers[RandomIndex - Counter MOD 4])
EndFunction
```

2.8 Key Schedule

2.8.1 System Endianness Check

Function:

$$\text{is\_system\_little\_endian}()$$

This function checks whether the system follows little-endian byte order.
In a little-endian system, the least significant byte is stored first.

Algorithm:

```
Function is_system_little_endian():
        Define value as 0x01
        Check the least significant byte of value's address
        If least significant byte is 0x01:
                Return True
        Else:
                Return False
EndFunction
```

2.8.2 Generate Subkeys

This function is responsible for deriving subkeys from the provided key bytes, using various cryptographic operations.

Mathematical Formulas:
For the byte order change operation:

$$\text{KeyBlock} = (\text{ByteD} \ll 24) \oplus (\text{ByteB} \ll 16) \oplus (\text{ByteC} \ll 8) \oplus \text{ByteA}$$

For the byte swap operation (in case of a big-endian system):

$$\text{KeyBlock} = ((\text{KeyBlock} \& 0x000000FF) \ll 24) \oplus ((\text{KeyBlock} \& 0x0000FF00) \ll 8) \oplus ((\text{KeyBlock} \& 0x00FF0000) \gg 8) \oplus ((\text{KeyBlock} \& 0xFF000000) \gg 24)$$

Algorithm:

```
Function KeySchedule(KeyBytes):
        If KeyBytes is empty:
                Return

        Initialize KeyBytesBuffer with KeyBytes
        While size of KeyBytesBuffer is not divisible by 4:
                Append 0x00 to KeyBytesBuffer

        Define KeyBlocks as an array of size (KeyBytesBuffer.size() / 4) filled with 0

        //Check system endianness and store the result in is_system_little_endian_flag
        is_system_little_endian_flag = is_system_little_endian()

        For i from 0 to size of KeyBytesBuffer in steps of 4:

                /*
                        Key preprocessing - Apply non-linear functions - Byte substitution box
                        The original byte is the number of rows, The first substitution finds the number of columns, and the second substitution finds the required value.
                */
                ByteA = ByteSubstitutionBoxA[ByteSubstitutionBoxA[KeyBytesBuffer[i]]]
                ByteB = ByteSubstitutionBoxA[ByteSubstitutionBoxA[KeyBytesBuffer[i + 1]]]
                ByteC = ByteSubstitutionBoxB[ByteSubstitutionBoxB[KeyBytesBuffer[i + 2]]]
                ByteD = ByteSubstitutionBoxB[ByteSubstitutionBoxB[KeyBytesBuffer[i + 3]]]

                // Fills the 32-bit number using the provided byte array taking into account system byte big/little endian order
                KeyBlock = (ByteD << 24) OR (ByteB << 16) OR (ByteC << 8) OR ByteA

                If not is_system_little_endian_flag:
                        KeyBlock = (KeyBlock AND 0x000000FF) << 24 OR (KeyBlock AND 0x0000FF00) << 8 OR (KeyBlock AND 0x00FF0000) >> 8 OR (KeyBlock AND 0xFF000000) >> 24

                Append KeyBlock to KeyBlocks

        Initialize RandomIndex to 0
        Define State as an array of size 128 of 32-bit numbers initialized with 0

        index = 0
        While index < size of KeyBlocks:
                offset = min((index + size of KDSB), size of KeyBlocks)

                If index > 0:
                        For i from 0 to size of KDSB:
                                j = index + i
                                KeyBlocks[j] = KeyBlocks[j] + (State[i] AND 0xFFFF0000)
                                KeyBlocks[j] = KeyBlocks[j] XOR (State[i] AND 0x0000FFFF)

                Copy block of KeyBlocks starting from index to KDSB
        For 4 rounds:
            For Counter from 0 to size of KDSB:
                Call MixWithAddSubtract(Counter, RandomIndex)
                Call RandomAccessMix(Counter, RandomIndex)
                Call ComplexMix(Counter, RandomIndex)

        If index >= size of KDSB:
            For i from index to offset:
                j = i - index
                State[j] = KDSB[j]
                State[j] = State[j] XOR (KeyBlocks[i] AND 0xFFFF0000)
                State[j] = State[j] + (KeyBlocks[i] AND 0x0000FFFF)

        index = index + size of KDSB

        If offset - index < size of KDSB
            Break loop

EndFunction
```

3. TitanWall Stream Cipher

Mathematical Functions:

Constructor (`TitanWallStreamCipher`):
Given a vector of bytes representing the key, it initializes the cipher by generating subkeys using the `KeySchedule` function. It also sets `IsKeyUsed` to true, indicating that a key has been set.

$$\text{Constructor}(KeyBytes) = \begin{matrix} \text{KeySchedule}(KeyBytes) \\ \text{IsKeyUsed} \leftarrow \text{true} \end{matrix}$$

Destructor (`~TitanWallStreamCipher`):
It resets the state of the cipher by calling the `ResetState` function.

$$\text{Destructor}() = \text{ResetState}()$$

GeneratePseudoRandomBytes:

3.1 Purpose:

The function aims to generate a sequence of pseudorandom bytes using initial states, cryptographic mixing, and transformation functions, and then fill the provided `Bytes` array with these pseudorandom bytes.

3.2 Mathematical Representation:

Let's represent the arrays and values used in the function:

Let B be the array `Bytes` .
Let K be the array `KDSB` .
Let S be the array `State` .

The size of K and S is 128, and each element is a 32-bit number.

The function operates in rounds, and in each round, elements of K are transformed using elements of S and vice versa, followed by cryptographic mixing to generate pseudorandomness.

Notation and Symbols:

- $\oplus$: Represents the XOR operation.
- $\land$: Represents the bitwise AND operation.
- $+$: Represents modular addition operation.

3.3 Operations:

a. Initial Checks:

If B is empty or `IsKeyUsed` is false:

- Exit the function.

b. Pseudorandom Number Generation:

Initialize RandomIndex to 0.

For each i in range of K:

- **Part A:** Update $K[i]$ using current S values:

$$K[i] = K[i] +_{\pmod{2^{32}}} (S[i] \land 0\text{xFFFF0000})$$

$$K[i] = K[i] \oplus (S[i] \land 0\text{x0000FFFF})$$

  These operations combine the left half of $S[i]$ with the right half via modular addition and XOR respectively.
- **Cryptographic Mixing:** This involves multiple rounds of cryptographic operations to generate pseudorandomness. These operations are `MixWithAddSubtract` , `RandomAccessMix` , and `ComplexMix` .
- **Part B:** Update $S[i]$ using the new $K[i]$ values:

$$S[i] = S[i] \oplus (K[i] \land 0\text{xFFFF0000})$$

$$S[i] = S[i] +_{\pmod{2^{32}}} (K[i] \land 0\text{x0000FFFF})$$

Similar to Part A, these operations combine the left half of $K[i]$ with the right half via XOR and modular addition respectively.

c. Filling Bytes Array:

For each i in range of S:

- Extract 4 bytes from $S[i]$ and store them in B. This is done by repeatedly taking the least significant byte from $S[i]$ and shifting $S[i]$ to the right.

4. InitialState:
It resets the current state using `ResetState`, then initializes the cipher with a new key using `KeySchedule`, and sets `IsKeyUsed` to true.

$$\text{InitialState}(KeyBytes) = \begin{array}{l} \text{ResetState}() \\ \text{KeySchedule}(KeyBytes) \\ \text{IsKeyUsed} \leftarrow \text{true} \end{array}$$

5. ResetState:
It resets the `State` and `KDSB` arrays to zero and sets `IsKeyUsed` to false.

$$\text{ResetState}() = \begin{array}{l} State, KDSB \leftarrow \text{Zero} \\ \text{IsKeyUsed} \leftarrow \text{false} \end{array}$$

Algorithm:

```
Class TitanWallStreamCipher InheritedFrom CryptographicFunctions

        Data:
                IsKeyUsed = Boolean
                State = Array of size 128 of 32-bit numbers

        Constructor(KeyBytes):
                Call KeySchedule(KeyBytes)
                IsKeyUsed = true

        Destructor():
                Call ResetState()

        Function GeneratePseudoRandomBytes(Bytes):

                // Check if Bytes is empty or no key has been used
                If Bytes is empty or not IsKeyUsed:
                        Return

                // Initialize RandomIndex
                RandomIndex = 0

                // Iterate over KDSB to generate pseudorandom 32-bit numbers
                For each i in KDSB:

                        // Update KDSB[i] using the current state
                        // Interleaved XOR and modulo addition operations on the left and right halves of a 32-bit number (Part A)
                        KDSB[i] += State[i] & 0xFFFF0000
                        KDSB[i] ^= State[i] & 0x0000FFFF

                        // Iterate and call cryptographic functions to generate pseudorandom 32-bit numbers
                        For Round from 0 to 3:
                                For Counter from 0 to size of KDSB:
                                        Call MixWithAddSubtract(Counter, RandomIndex)
                                        Call RandomAccessMix(Counter, RandomIndex)
                                        Call ComplexMix(Counter, RandomIndex)

                        // Update the internal state using KDSB
                        // Interleaved XOR and modulo addition operations on the left and right halves of a 32-bit number (Part B)
                        State[i] ^= KDSB[i] & 0xFFFF0000
                        State[i] += KDSB[i] & 0x0000FFFF

                // Fill Bytes array with the generated pseudorandom 32-bit numbers
                For i from 0 to size of State:
                        value = State[i]
                        For j from 0 to 3:
```

```
                        If (i * 4 + j) is less than size of Bytes:
                                Bytes[i * 4 + j] = value & 0xFF
                                value >>= 8

        Function InitialState(KeyBytes):
                Call ResetState()
                Call KeySchedule(KeyBytes)
                IsKeyUsed = true

        Function ResetState():
                Set all elements of State and KDSB to 0
                IsKeyUsed = false

EndClass
```

4. TitanWall Block Cipher

4.1 Class Structure:

```
Class TitanWallBlockCipher InheritedFrom CryptographicFunctions:

        // Private methods
        Function PHT(a, b) {...}
        Function InversePHT(a, b) {...}

        // Constructor
        Constructor(KeyBytes):
                Call KeySchedule(KeyBytes)

        // Destructor
        Destructor():
                Reset KDSB to zeros

        // Public methods
        Function Encrypt(plaintext) {...}
        Function Decrypt(ciphertext) {...}

EndClass
```

4.2 Mathematical Explanation:

Pseudo-Hadamard Transform (PHT) and Inverse PHT:

The Pseudo-Hadamard Transform (PHT) is a basic operation in block ciphers. Given two inputs a and b, the PHT produces two outputs a' and b'.

For $PHT(a, b)$:

$$a' = (a + b) \mod 2^{32}$$

$$b' = (a + 2b) \mod 2^{32}$$

For $InversePHT(a', b')$:

$$a = (2a' - b') \mod 2^{32}$$

$$b = (b' - a') \mod 2^{32}$$

4.2.1 Encryption:

The encryption function transforms a plaintext block into a ciphertext block.
It operates on 8 32-bit words of plaintext and uses several operations, including substitution, permutation, and the PHT. The encryption employs a series of rounds to mix the plaintext and produce the ciphertext.

1. **Initialization**:
   Given an input plaintext block, we first split it into eight 32-bit words A, B, C, D, E, F, G, H. The key scheduling values $KDSB$ are used in various operations throughout the encryption process.
2. **Addition with Key Values**:
   The values of B, D, F, and H are updated using:

$$B = B + KDSB[0]$$

$$D = D + KDSB[1]$$

$$F = F + KDSB[2]$$

$$H = H + KDSB[3]$$

3. **Iterative Rounds**:
   For 62 rounds, we undergo a series of operations, as follows (from j=1 to j=62):

- **Substitution**:
  We perform a nonlinear transformation on B, D, F, and H using:

$$t = B \oplus ((B \ll 1) + 1)$$

$$u = D \oplus ((D \ll 2) + 1)$$

$$v = F \oplus ((F \ll 3) + 1)$$

$$w = H \oplus ((H \ll 4) + 1)$$

- **Mixing with Rotational Operations**:
  We update the values of $A, C, E,$ and G using rotations:

$$A = ((A - t) \ggg (w \mod 32)) + KDSB[2 \times j]$$

$$C = ((C \oplus u) \ggg (v \mod 32)) + KDSB[2 \times j + 1]$$

$$E = ((E \oplus v) \ggg (u \mod 32)) + KDSB[2 \times j + 2]$$

$$G = ((G + w) \ggg (t \mod 32)) + KDSB[2 \times j + 3]$$

- **Update Using KDSB**:
  The values of $B, D, F,$ and H are further modified using:

$$B = B + KDSB[j \mod \text{size of KDSB}]$$

$$D = D \oplus KDSB[(j + 1) \mod \text{size of KDSB}]$$

$$F = F \oplus KDSB[(j + 2) \mod \text{size of KDSB}]$$

$$H = H - KDSB[(j + 3) \mod \text{size of KDSB}]$$

- **Permutation**:
  The eight words are permuted in the order $(F, D, B, H, A, G, E, C) = (A, B, C, D, E, F, G, H)$.
- **Pseudo-Hadamard Transform (PHT)**:
  PHT operations are applied pairwise on the words:

$$
\begin{aligned}
(A,B) &\rightarrow \mathrm{PHT}(A,B)\\
(C,D) &\rightarrow \mathrm{PHT}(C,D)\\
(E,F) &\rightarrow \mathrm{PHT}(E,F)\\
(G,H) &\rightarrow \mathrm{PHT}(G,H)
\end{aligned}
$$

4. **Final Addition with Key Values**:

$$A = A + KDSB[\mathrm{SubkeysSize} - 4]$$

$$C = C + KDSB[\mathrm{SubkeysSize} - 3]$$

$$E = E + KDSB[\mathrm{SubkeysSize} - 2]$$

$$G = G + KDSB[\mathrm{SubkeysSize} - 1]$$

5. **Output**:
   The resulting eight words A, B, C, D, E, F, G, H are the ciphertext.

4.2.2 Decryption:

The decryption function is the inverse of the encryption function.
It transforms a ciphertext block back into its original plaintext form. It also uses the inverse operations of the encryption, including the inverse PHT.

1. **Initialization**:
   Given an input ciphertext block, we first split it into eight 32-bit words A, B, C, D, E, F, G, H. The key scheduling values $KDSB$ are used in various operations throughout the decryption process.
2. **Subtraction with Key Values**:
   The values of $A, C, E,$ and G are updated using:

$$A = A - KDSB[\mathrm{SubkeysSize} - 4]$$

$$C = C - KDSB[\mathrm{SubkeysSize} - 3]$$

$$E = E - KDSB[\mathrm{SubkeysSize} - 2]$$

$$G = G - KDSB[\mathrm{SubkeysSize} - 1]$$

3. **Iterative Rounds**:
   For 62 rounds, we undergo a series of operations, as follows (from j=62 to j=1):

- **Inverse Pseudo-Hadamard Transform**:
  InversePHT operations are applied pairwise on the words:

$$(A,B) \to \text{InversePHT}(A,B)$$
$$(C,D) \to \text{InversePHT}(C,D)$$
$$(E,F) \to \text{InversePHT}(E,F)$$
$$(G,H) \to \text{InversePHT}(G,H)$$

- **Inverse Permutation**:
  The eight words are permuted back to the order $(A,B,C,D,E,F,G,H) = (F,D,B,H,A,G,E,C)$.
- **Update Using KDSB**:
  The values of B, D, F, and H are further modified using:

$$H = H + KDSB[(j+3) \mod \text{size of KDSB}]$$

$$F = F \oplus KDSB[(j+2) \mod \text{size of KDSB}]$$

$$D = D \oplus KDSB[(j+1) \mod \text{size of KDSB}]$$

$$B = B - KDSB[j \mod \text{size of KDSB}]$$

- **Inverse Substitution**:
  We perform a nonlinear transformation on B, D, F, and H using:

$$t = B \oplus ((B \ll 1) + 1)$$

$$u = D \oplus ((D \ll 2) + 1)$$

$$v = F \oplus ((F \ll 3) + 1)$$

$$w = H \oplus ((H \ll 4) + 1)$$

- **Mixing with Rotational Operations**:
  We update the values of A, C, E, and G using inverse rotations:

$$G = ((G - KDSB[2 \times j + 3]) \lll (t \mod 32)) - w$$

$$E = ((E - KDSB[2 \times j + 2]) \lll (u \mod 32)) \oplus v$$

$$C = ((C - KDSB[2 \times j + 1]) \lll (v \mod 32)) \oplus u$$

$$A = ((A - KDSB[2 \times j]) \lll (w \mod 32)) + t$$

4. **Final Subtraction with Key Values**:

$$H = H - KDSB[3]$$

$$F = F - KDSB[2]$$

$$D = D - KDSB[1]$$

$$B = B - KDSB[0]$$

5. **Output**:
   The resulting eight words A, B, C, D, E, F, G, H are the plaintext.

4.3 Algorithm:

```
Function PHT(a, b):
        temp_a = (a + b) MOD 2^32
        temp_b = (a + 2 * b) MOD 2^32
        return {temp_a, temp_b}

Function InversePHT(a, b):
        temp_b = (b - a) MOD 2^32
        temp_a = (2 * a - b) MOD 2^32
        return {temp_a, temp_b}
```

```
Function Encrypt(plaintext):
        If size of plaintext % 8 ≠ 0:
                Throw error("The number of data blocks cannot be aligned. Please perform preprocessing before proceeding.")

        Initialize ciphertext of size of plaintext

        For i from 0 to size of plaintext with step of 8:
                Load and Split plaintext[i to i+7] into A, B, C, D, E, F, G, H

                Update B, D, F, H with KDSB

                For j from 1 to 62:
                        Apply substitution on B, D, F, H
                        Update A, C, E, G using B, D, F, H and KDSB
                        Update B, D, F, H using KDSB

                        Permute A, B, C, D, E, F, G, H

                        Apply PHT on A, B; C, D; E, F; G, H

                Update A, C, E, G with last elements of KDSB

                Store A, B, C, D, E, F, G, H into ciphertext

        Return ciphertext
```

```
Function Decrypt(ciphertext):
        If size of ciphertext % 8 ≠ 0:
                Throw error("The number of data blocks cannot be aligned. Please perform preprocessing before proceeding.")

        Initialize plaintext of size of ciphertext

        For i from 0 to size of ciphertext with step of 8:
                Load and Split ciphertext[i to i+7] into A, B, C, D, E, F, G, H

                Update G, E, C, A using last elements of KDSB

                For j from 62 down to 1:
                        Apply InversePHT on A, B; C, D; E, F; G, H

                        Inverse permute A, B, C, D, E, F, G, H

                        Update B, D, F, H using KDSB
                        Update A, C, E, G using B, D, F, H and KDSB
                        Apply inverse substitution on B, D, F, H

                Update H, F, D, B using KDSB

                Store A, B, C, D, E, F, G, H into plaintext

        Return plaintext
```

4. Data Testing and Verification

1. **Mean**:
   Given a set of observed changes $X = \{x_1, x_2, \ldots, x_T\}$ over T trials:

$$\mu = \frac{1}{T}\sum_{t=1}^{T} x_t$$

2. **Variance**:
   Given the same set of observed changes and the computed mean μ:

$$\sigma^2 = \frac{1}{T}\sum_{t=1}^{T}(x_t - \mu)^2$$

3. **Differential Test**:
   For each trial, a bit is flipped in the input (either plaintext or key).
   The change in the output (either ciphertext or guessed plaintext) is observed and measured using the Hamming distance.

Now, let's define the Hamming distance:
Given two binary strings a and b of equal length, the Hamming distance $HD(a,b)$ is defined as the number of positions at which the corresponding bits are different.
Mathematically, the Hamming distance between two strings a and b of length n can be defined as:

$$HD(a,b)=\sum_{i=1}^{n}\delta(a_i,b_i)$$

Where:

$$\delta(x,y)=\begin{cases}1 & \text{if } x\neq y\\ 0 & \text{if } x=y\end{cases}$$

Here, δ is an indicator function that outputs 1 if the bits at the given position are different and 0 if they are the same.

4. **General Approach**:
    - For each test, multiple trials are conducted.
    - In each trial, a bit is flipped in either the plaintext or the key, depending on the test.
    - The output (either ciphertext or guessed plaintext) for the modified input is computed.
    - The Hamming distance between the original output and the new output is calculated.
    - The mean and variance of the number of bit changes across all trials are computed.
5. **Mathematical Formulation**:

Commonly recognized variable:

- T - number of trials
- P - the original plaintext
- P' - plaintext with one bit changed
- K - the original key.
- K' - key with one bit changed
- C - the original ciphertext
- C' - cipher text with one bit changed

Commonly recognized functions:

- HD - Hamming distance
- EF - Encryption function
- DF - Decryption function

(a). **Plain-to-Cipher Diffusivity Test** (`plain_for_cipher_diffusivity`)

This test evaluates how a change in one bit of the plaintext spreads to the entire ciphertext.

Formula:

$$\mu_{DPC} = \frac{1}{T}\sum_{t=1}^{T} HD(EF(P,K), EF(P',K))$$

$$\sigma^2_{PC} = \frac{1}{T}\sum_{t=1}^{T}(HD(EF(P,K), EF(P',K)) - \mu_{DPC})^2$$

Where:

- μ_{DPC} is the average diffusivity from plaintext to ciphertext.
- σ^2_{PC} is the variance of diffusivity from plaintext to ciphertext.

(b). **Key-to-Cipher Confusability Test** (`key_for_cipher_confusability`)

This test evaluates how a change in one bit of the key confuses the entire ciphertext.

Formula:

$$\mu_{CKC} = \frac{1}{T}\sum_{t=1}^{T} HD(EF(P,K), EF(P,K'))$$

$$\sigma^2_{KC} = \frac{1}{T}\sum_{t=1}^{T}(HD(EF(P,K), EF(P,K')) - \mu_{CKC})^2$$

Where:

- μ_{CKC} is the average confusability from key to ciphertext.
- σ^2_{KC} is the variance of confusability from key to ciphertext.

(c). **Cipher-to-Plain Diffusivity Test** (`cipher_for_plain_diffusivity`)

This test evaluates how a change in one bit of the ciphertext spreads to the entire plaintext.

Formula:

$$\mu_{DCP} = \frac{1}{T}\sum_{t=1}^{T} HD(DF(C,K), DF(C',K))$$

$$\sigma^2_{CP} = \frac{1}{T}\sum_{t=1}^{T}(HD(DF(C,K), DF(C',K)) - \mu D_{CP})^2$$

Where:

- μ_{DCP} is the average diffusivity from ciphertext to plaintext.
- σ^2_{CP} is the variance of diffusivity from ciphertext to plaintext.

(d). **Key-to-Plain Confusability Test** (`key_for_plain_confusability`)

This test evaluates how a change in one bit of the key confuses the entire plaintext.

Formula:

$$\mu_{C_{KP}} = \frac{1}{T}\sum_{t=1}^{T} HD(DF(C,K), DF(C,K'))$$

$$\sigma^2_{KP} = \frac{1}{T}\sum_{t=1}^{T}(HD(DF(C,K), DF(C,K')) - \mu C_{KP})^2$$

Where:

- $\mu_{C_{KP}}$ is the average confusability from key to plaintext.
- σ^2_{KP} is the variance of confusability from key to plaintext.

Test Data element size is: 10 Mega byte
Test Key element size is: 5120 byte

```
std::uint64_t RandomSeed64 = CommonSecurity::GenerateSecureRandomNumberSeed<std::uint64_t>(random_device_object);
 : 349210706352
TitanWallStreamCipher Cryptographic properties test result:
Testing ...... (1024/1024)
Cipher Diffusivity test (Difference testing based on changed plaintext data):
Mean number of bit changes: 1
Variance of number of bit changes: 0
密文扩散性测试 (基于明文的变化):
比特位数变化的平均数量: 1
比特位数变化的方差: 0
Testing ...... (1024/1024)
Cipher Confusion test (Difference testing based on changed key data):
Mean number of bit changes: 208.003
Variance of number of bit changes: 382839
密文混淆性测试 (基于密钥的变化):
比特位数变化的平均数量: 208.003
比特位数变化的方差: 382839
Testing ...... (1024/1024)
Plain Diffusivity test (Difference testing based on changed ciphertext data):
Mean number of bit changes: 1
Variance of number of bit changes: 0
明文扩散性测试 (基于密文的变化):
比特位数变化的平均数量: 1
比特位数变化的方差: 0
Testing ...... (1024/1024)
Plain Confusability test (Difference testing based on changed key data):
Mean number of bit changes: 207.252
Variance of number of bit changes: 380098
明文混淆性测试 (基于密钥的变化):
比特位数变化的平均数量: 207.252
比特位数变化的方差: 380098

TitanWallBlockCipher Cryptographic properties test result:
Testing ...... (1024/1024)
Cipher Diffusivity test (Difference testing based on changed plaintext data):
Mean number of bit changes: 64.2109
Variance of number of bit changes: 24.0746
密文扩散性测试 (基于明文的变化):
比特位数变化的平均数量: 64.2109
比特位数变化的方差: 24.0746
Testing ...... (1024/1024)
Cipher Confusion test (Difference testing based on changed key data):
Mean number of bit changes: 4.60576e+06
Variance of number of bit changes: 1.71841e+14
密文混淆性测试 (基于密钥的变化):
比特位数变化的平均数量: 4.60576e+06
```

```
比特位数变化的方差: 1.71841e+14
Testing ...... (1024/1024)
Plain Diffusivity test (Difference testing based on changed ciphertext data):
Mean number of bit changes: 3.9977e+07
Variance of number of bit changes: 37.8055
明文扩散性测试 (基于密文的变化):
比特位数变化的平均数量: 3.9977e+07
比特位数变化的方差: 37.8055
Testing ...... (1024/1024)
Plain Confusability test (Difference testing based on changed key data):
Mean number of bit changes: 4.01472e+07
Variance of number of bit changes: 1.1601e+12
明文混淆性测试 (基于密钥的变化):
比特位数变化的平均数量: 4.01472e+07
比特位数变化的方差: 1.1601e+12
```

5. Conclusion

- The TitanWall Cipher emerges as a formidable cryptographic solution, which is demonstrated by its intricate design and the rigorous testing it has undergone.
  With both block and stream cipher functionalities in its arsenal, it showcases versatility in application.
  The foundational principles, rooted in mathematical constructs such as Galois fields and Pseudo-Hadamard Transforms, offer a strong underpinning.
- A critical aspect of its strength lies in its diffusion and confusion properties.
  The diffusion tests, as elucidated, aim to measure the degree of change in output when a single bit of plaintext or ciphertext is altered, all while keeping the key constant.
  Ideal diffusion would ensure that changes in the input lead to indeterminate and widespread changes in the output.
  The results from the TitanWall Cipher indicate a non-zero mean bit change, suggesting commendable diffusion properties.
  It's imperative that this value doesn't gravitate too close to zero, as that would indicate predictability, potentially offering attackers discernible patterns.
  On the other hand, the confusion tests serve as a metric for evaluating the sensitivity of the cipher's output in response to perturbations in the key.
  A pronounced mean bit change, in this context, is indeed advantageous, suggesting that even infinitesimal modifications in the key can instigate substantial alterations in the output.
  Nonetheless, a disproportionately elevated confusion coefficient might be indicative of potential cyclical patterns within the cipher, a phenomenon that could undermine its cryptographic integrity.
  Furthermore, an element that cannot be overlooked is the extensive number of test rounds, quantified at 1024.
- Such rigorous testing not only imposes stringent demands on the cryptographic robustness of the algorithm but also becomes a discerning instrument to identify and rectify any suboptimal diffusion and confusion characteristics during iterative encryption and decryption processes.
  The testing parameters, involving 10 Mega bytes of data elements and 5120 bytes of key element per test round, further exemplify the rigorous nature of the evaluation.
  In summation, the TitanWall Cipher, with its promising diffusion and confusion coefficients, paints a picture of a robust cryptographic system.
  However, the nuanced balance between these coefficients is a testament to the intricate dance of cryptography — striving for security while avoiding potential pitfalls.
  As with any cryptographic endeavor, continuous scrutiny and real-world testing remain paramount to ascertain its resilience against evolving threats.

6. Appendix

C++ code:

```
#pragma once

#include <cstdint>
#include <cassert>

#include <iostream>
#include <iomanip>

#include <limits>
#include <exception>

#include <array>
#include <vector>
#include <tuple>

inline bool is_system_little_endian()
{
        const int value { 0x01 };
        const void * address { static_cast<const void *>(&value) };
        const unsigned char * least_significant_address { static_cast<const unsigned char *>(address) };

        return (*least_significant_address == 0x01);
}

struct CryptographicFunctions
{
protected:

        //Concatenation of Fibonacci numbers., π, ϕ, e
        const std::array<uint32_t, 4> MathMagicNumbers {0x01B70C8E,0x243F6A88,0x9E3779B9,0xB7E15162};

        /*
                This byte-substitution box: Strict avalanche criterion is satisfied !
                ByteDataSecurityTestData Transparency Order Is: 7.85956
                ByteDataSecurityTestData Nonlinearity Is: 112
                ByteDataSecurityTestData Propagation Characteristics Is: 8
                ByteDataSecurityTestData Delta Uniformity Is: 4
                ByteDataSecurityTestData Robustness Is: 0.984375
                ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 10.3062
                ByteDataSecurityTestData Absolute Value Indicatorer Is: 32
                ByteDataSecurityTestData Sum Of Square Value Indicator Is: 67584
                ByteDataSecurityTestData Algebraic Degree Is: 8
                ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
        */
        static constexpr std::array<std::uint8_t, 256> ByteSubstitutionBoxA
        {
```

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

/*
        This byte-substitution box: Strict avalanche criterion is satisfied !
        ByteDataSecurityTestData Transparency Order Is: 7.85221
        ByteDataSecurityTestData Nonlinearity Is: 112
        ByteDataSecurityTestData Propagation Characteristics Is: 8
        ByteDataSecurityTestData Delta Uniformity Is: 4
        ByteDataSecurityTestData Robustness Is: 0.984375
        ByteDataSecurityTestData Signal To Noise Ratio/Differential Power Analysis Is: 9.23235
        ByteDataSecurityTestData Absolute Value Indicatorer Is: 32
        ByteDataSecurityTestData Sum Of Square Value Indicator Is: 67584
        ByteDataSecurityTestData Algebraic Degree Is: 8
        ByteDataSecurityTestData Algebraic Immunity Degree Is: 4
*/
static constexpr std::array<std::uint8_t, 256> ByteSubstitutionBoxB
{
        0xE2, 0x0D, 0x3E, 0x94, 0x1D, 0x02, 0x48, 0x71, 0x1C, 0x93, 0xA8, 0x69, 0xB7, 0x90, 0xAA, 0x5C,
        0x37, 0x5A, 0xDB, 0x75, 0xFD, 0x64, 0x8D, 0xD3, 0x49, 0x12, 0xCB, 0xE0, 0xC6, 0x9A, 0x16, 0xDF,
        0x33, 0x08, 0xAE, 0xD0, 0xFF, 0xB3, 0x29, 0x34, 0x56, 0xE9, 0x20, 0x7F, 0x44, 0x2F, 0xFA, 0xDC,
        0x9C, 0x4E, 0x8A, 0x46, 0xDD, 0x42, 0xD9, 0x6A, 0x70, 0xF3, 0xF5, 0x8C, 0x09, 0x72, 0x7C, 0x9F,
        0xB0, 0x1B, 0x96, 0x62, 0x45, 0x10, 0xEA, 0xA0, 0x6D, 0xA7, 0xCA, 0x3F, 0xAC, 0x0B, 0x23, 0x57,
        0x28, 0x5B, 0xF7, 0xB4, 0x82, 0x9E, 0x17, 0xEC, 0x31, 0xA9, 0x14, 0xA2, 0xC5, 0x1E, 0x6C, 0x4F,
        0x4D, 0x55, 0x0F, 0xBB, 0xD7, 0xC0, 0x0A, 0xE1, 0x47, 0xAF, 0x89, 0x26, 0xC4, 0xCD, 0x9D, 0x2C,
        0x81, 0x3B, 0xEB, 0xF9, 0x53, 0x5E, 0x6F, 0x95, 0xBD, 0x27, 0xBA, 0xFB, 0x07, 0xA5, 0x5D, 0xED,
        0xDA, 0x2A, 0xA4, 0x99, 0x73, 0x01, 0x98, 0x13, 0x1A, 0xA3, 0xB1, 0xBF, 0xE7, 0x15, 0xF8, 0x78,
        0x0E, 0x9B, 0x6B, 0x67, 0xF6, 0xD8, 0x36, 0x61, 0x7E, 0xFC, 0x86, 0x40, 0x92, 0x52, 0x03, 0x97,
        0x87, 0xB9, 0x85, 0x8E, 0x68, 0x06, 0x59, 0xC9, 0xD2, 0xD1, 0x76, 0xC1, 0x22, 0x39, 0x5F, 0xE3,
        0x8B, 0xA6, 0xD6, 0x2B, 0x32, 0xBE, 0xC3, 0xE6, 0x60, 0x7A, 0x0C, 0xF4, 0x25, 0x41, 0x24, 0x54,
        0x1F, 0xF0, 0x38, 0xAB, 0x05, 0x83, 0xCF, 0x58, 0x79, 0x3C, 0xC8, 0x7D, 0xAD, 0x51, 0xF2, 0xB2,
        0x21, 0x43, 0x6E, 0xEF, 0xC7, 0x18, 0x3A, 0x88, 0x4B, 0x2E, 0x65, 0xDE, 0x66, 0xB6, 0x04, 0x30,
        0xC2, 0x4A, 0xB5, 0x19, 0xCC, 0xFE, 0xD5, 0x84, 0x80, 0x8F, 0x2D, 0xE8, 0x35, 0xF1, 0x63, 0x4C,
        0x77, 0x91, 0x11, 0xB8, 0xE4, 0xCE, 0xEE, 0xA1, 0x00, 0xD4, 0x50, 0xBC, 0x3D, 0x7B, 0x74, 0xE5
};

//Although it is a key-dependent substitution box, it is a 32-bit elemental pseudo-random number
//虽然是依赖密钥的替换盒，但是它是32比特的元素伪随机数
std::array<std::uint32_t, 128> KDSB {};

uint32_t Bits32RotateLeft(uint32_t number, uint32_t bit)
{
        return (number << (bit % 32)) | (number >> (32 - (bit % 32)));
}

uint32_t Bits32RotateRight(uint32_t number, uint32_t bit)
{
        return (number >> (bit % 32)) | (number << (32 - (bit % 32)));
}

//Nonlinear Boolean function
inline uint32_t FF(uint32_t A, uint32_t B, uint32_t C, size_t Index, size_t ArraySize)
{
        if(Index < ((ArraySize / 4) * 3))
                return A ^ B ^ C;
        else
                return (A & B) | (A & C) | (B & C);
}

inline uint32_t GG(uint32_t A, uint32_t B, uint32_t C, size_t Index, size_t ArraySize)
{
        if(Index < ((ArraySize / 4) * 3))
                return A ^ B ^ C;
        else
                return (A & B) | (~A & C);
}

//Linear Transform function
inline uint32_t L( uint32_t number )
{
        return number ^ Bits32RotateLeft( number, 2 ) ^ Bits32RotateLeft( number, 10 ) ^ Bits32RotateLeft( number, 18 ) ^ Bits32RotateLeft( number, 24 );
}

inline uint32_t L2( uint32_t number )
{
        return number ^ Bits32RotateLeft( number, 13 ) ^ Bits32RotateLeft( number, 23 );
}

void MixWithAddSubtract(size_t Counter, uint64_t& RandomIndex)
```

```
{
        //混合函数，模加和模减，数学常数
        RandomIndex ^= KDSB[Counter % KDSB.size()] + MathMagicNumbers[Counter % 4];
        KDSB[Counter % KDSB.size()] += KDSB[(Counter + 1) % KDSB.size()] - MathMagicNumbers[(RandomIndex + Counter) % 4];
        RandomIndex ^= KDSB[(Counter + 1) % KDSB.size()] + MathMagicNumbers[(RandomIndex - Counter) % 4];
        KDSB[(Counter + 1) % KDSB.size()] -= KDSB[Counter % KDSB.size()] - MathMagicNumbers[(RandomIndex + Counter) % 4];
}

void RandomAccessMix(size_t Counter, uint64_t& RandomIndex)
{
        //混合函数，模加和模减，数学常数
        //使用RandomPosition随机访问
        size_t RandomPosition = RandomIndex % KDSB.size();
        KDSB[Counter % KDSB.size()] ^= KDSB[RandomPosition];
        KDSB[Counter % KDSB.size()] += KDSB[Counter % KDSB.size()] - MathMagicNumbers[RandomPosition % 4];
}

uint32_t NLFSR(uint32_t Register)
{
        //Referenced: https://en.wikipedia.org/wiki/KeeLoq
        for ( size_t NLFSR_ROUND = 0; NLFSR_ROUND < 64; ++NLFSR_ROUND )
        {
                //PrimerNumbers: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61
                bool BinaryA = ((Register >> 31) & 1);
                bool BinaryB = ((Register >> 29 - 1) & 1);
                bool BinaryC = ((Register >> 23) & 1);
                bool BinaryD = ((Register >> 19 - 2) & 1);
                bool BinaryE = ((Register >> 13) & 1);
                bool BinaryF = ((Register >> 7 - 3) & 1);
                bool BinaryG = ((Register >> 1) & 1);

                bool BinaryH = ((Register >> 16) & 1) ^ (Register & 1);

                //Right Shift Register
                Register >>= 1;

                //Feedback Function
                bool FeedBack0 = (BinaryB != BinaryG) != (BinaryA != BinaryF);
                bool FeedBack1 = (BinaryA && BinaryD) != (BinaryA && BinaryG);
                bool FeedBack2 = (BinaryB && BinaryC) != (BinaryB && BinaryD) != (BinaryB && BinaryE);
                bool FeedBack3 = (BinaryE && BinaryF) == (BinaryD && BinaryF) == (BinaryC && BinaryF);
                bool FeedBack4 = (BinaryF && BinaryG) == (BinaryE && BinaryG) == (BinaryD && BinaryG);
                bool FeedBack5 = (BinaryA && BinaryB && BinaryG) != (BinaryA && BinaryD && BinaryG) != (BinaryA && BinaryF && BinaryG);
                bool FeedBack6 = (BinaryA && BinaryB && BinaryC) != (BinaryB && BinaryC && BinaryD) != (BinaryC && BinaryD && BinaryE) != (BinaryD && BinaryE && BinaryF) != (BinaryE && BinaryF &&
                bool FeedBack7 = (BinaryA && BinaryC && BinaryE && BinaryG) != (BinaryB && BinaryD && BinaryF);

                bool FeedBack = BinaryH != FeedBack0 != FeedBack1 != FeedBack2 != FeedBack3 != FeedBack4 != FeedBack5 != FeedBack6 != FeedBack7;
```

```
				Register ^= static_cast<uint64_t>( FeedBack ) << 31;
				//std::cout << "Register: "<< Register << std::endl;
				//std::cout << "Register (hex): " << std::hex << Register << std::endl;
			}

			return Register;
		}

		void ComplexMix(size_t Counter, uint64_t& RandomIndex)
		{
			uint32_t RegisterLeft = RandomIndex >> 32;
			uint32_t RegisterRight = RandomIndex & 0x00000000FFFFFFFF;

			RegisterLeft = NLFSR(RegisterLeft);
			RegisterRight = NLFSR(RegisterRight);

			RandomIndex = uint64_t(RegisterRight) << 32 | uint64_t(RegisterLeft);

			KDSB[Counter % KDSB.size()] ^= KDSB[(Counter - 2 + KDSB.size()) % KDSB.size()] ^ KDSB[(Counter - 1) % KDSB.size()];

			//GG, FF, L 函数
			KDSB[Counter % KDSB.size()] += GG(Counter, KDSB[(Counter - 1) % KDSB.size()], RandomIndex % std::numeric_limits<uint32_t>::max(), Counter, KDSB.size());
			KDSB[Counter % KDSB.size()] -= FF(KDSB[(Counter - 3) % KDSB.size()], L(KDSB[(Counter - 2) % KDSB.size()]), KDSB[(Counter - 1) % KDSB.size()], Counter, KDSB.size());

			//混合函数，模加和模减，数学常数
			//L, L2 函数
			RandomIndex ^= L(KDSB[Counter % KDSB.size()] - MathMagicNumbers[Counter % 4]);
			KDSB[(Counter + 1) % KDSB.size()] -= L2(KDSB[Counter % KDSB.size()] + MathMagicNumbers[(RandomIndex - Counter) % 4]);
			RandomIndex ^= L2(KDSB[Counter % KDSB.size()] - MathMagicNumbers[(RandomIndex + Counter) % 4]);
			KDSB[(Counter + 2) % KDSB.size()] += L(KDSB[(Counter + 1) % KDSB.size()] + MathMagicNumbers[(RandomIndex - Counter) % 4]);
		}

	public:

		void KeySchedule(const std::vector<std::uint8_t>& KeyBytes)
		{
			if (KeyBytes.empty())
			{
				return;
			}

			std::vector<std::uint8_t> KeyBytesBuffer(KeyBytes);

			while(KeyBytesBuffer.size() % 4 != 0)
			{
				KeyBytesBuffer.push_back(0x00);
			}
```

```
std::vector<std::uint32_t> KeyBlocks(KeyBytesBuffer.size() / 4, 0);

bool is_system_little_endian_flag = is_system_little_endian();

uint32_t KeyBlock = 0;
for ( size_t i = 0, j = 0; i < KeyBytesBuffer.size(); i += 4 )
{
        /*
                Key preprocessing - Apply non-linear functions - Byte substitution box
                密钥预处理 - 应用非线性函数 - 字节替换盒
        */

        uint8_t& ByteA = KeyBytesBuffer[i];
        uint8_t& ByteB = KeyBytesBuffer[i + 1];
        uint8_t& ByteC = KeyBytesBuffer[i + 2];
        uint8_t& ByteD = KeyBytesBuffer[i + 3];

        //The original byte is the number of rows, The first substitution finds the number of columns, and the second substitution finds the required value.
        //原字节是行数,第一次替代找到列数,第二次替代找到需要的值
        ByteA = ByteSubstitutionBoxA[ByteSubstitutionBoxA[ByteA]];
        ByteB = ByteSubstitutionBoxA[ByteSubstitutionBoxA[ByteB]];
        ByteC = ByteSubstitutionBoxB[ByteSubstitutionBoxB[ByteC]];
        ByteD = ByteSubstitutionBoxB[ByteSubstitutionBoxB[ByteD]];

        //Byte Order Change: D C B A -> D B C A

        //4 8-bits packing into 1 32-bits
        KeyBlock = uint32_t(ByteD) << 24 | uint32_t(ByteB) << 16 | uint32_t(ByteC) << 8 | ByteA;

        if(!is_system_little_endian_flag)
        {
                //Byte swap 32-bits
                KeyBlock = ((KeyBlock & 0x000000FF) << 24) | ((KeyBlock & 0x0000FF00) << 8) | ((KeyBlock & 0x00FF0000) >> 8) | ((KeyBlock & 0xFF000000) >> 24);
        }
        KeyBlocks[j] = KeyBlock;
        ++j;
}

uint64_t RandomIndex = 0;

//PRNG State Register
std::array<std::uint32_t, 128> State {};

size_t index = 0;
while (index < KeyBlocks.size())
{
```

```
// 每次从KeyBlocks中取出128个32位块放入KDSB
size_t offset = std::min(index + KDSB.size(), KeyBlocks.size());

//If the structure has indeed executed a round, the status is fed back to the next round of 128 32-bit key blocks.
//如果该结构确实已经执行了一轮，则将状态反馈到下一轮的128个32位密钥块。
if(index > 0)
{
        //Left half bits Mix use MODULAR-ADDITION
        //Right half bits Mix use XOR
        for(size_t i = 0, j = index; i < KDSB.size() && j < offset; ++i, ++j)
        {
                KeyBlocks[j] += State[i] & uint32_t(0xFFFF0000);
                KeyBlocks[j] ^= State[i] & uint32_t(0x0000FFFF);
        }
}

std::copy(KeyBlocks.begin() + index, KeyBlocks.begin() + offset, KDSB.begin());

//Pseudo Random Function
//伪随机函数
for (size_t Round = 0; Round < 4; Round++)
{
        for (size_t Counter = 0; Counter < KDSB.size(); Counter++)
        {
                MixWithAddSubtract(Counter, RandomIndex);
                RandomAccessMix(Counter, RandomIndex);
                ComplexMix(Counter, RandomIndex);
        }
}

if(index >= KDSB.size())
{
        //Left half bits Mix use XOR
        //Right half bits Mix use MODULAR-ADDITION
        for(size_t i = index, j = 0; i < offset && j < KDSB.size(); ++i, ++j)
        {
                State[j] = KDSB[j];

                State[j] ^= KeyBlocks[i] & uint32_t(0xFFFF0000);
                State[j] += KeyBlocks[i] & uint32_t(0x0000FFFF);
        }
}

index += KDSB.size();

// 如果不足128个32位块，执行一次4轮的伪随机函数后就结束
if (offset - index < KDSB.size())
```

```
                                break;
                        }
                }
};

class TitanWallStreamCipher : public CryptographicFunctions
{
private:

        bool IsKeyUsed = false;

        //PRNG State Register
        std::array<std::uint32_t, 128> State {};

public:

        TitanWallStreamCipher() = default;

        explicit TitanWallStreamCipher(const std::vector<std::uint8_t>& KeyBytes)
        {
                // 生成子密钥
                KeySchedule(KeyBytes);
                IsKeyUsed = true;
        }

        ~TitanWallStreamCipher()
        {
                ResetState();
        }

        void GeneratePseudoRandomBytes(std::vector<std::uint8_t>& Bytes)
        {
                if(Bytes.empty())
                        return;

                if(!IsKeyUsed)
                        return;

                uint64_t RandomIndex = 0;

                for ( size_t i = 0; i < KDSB.size(); i++ )
                {
                        KDSB[i] += State[i] & 0xFFFF0000;
                        KDSB[i] ^= State[i] & 0x0000FFFF;

                        //Pseudo Random Function
                        //伪随机函数
```

```
				for (size_t Round = 0; Round < 4; Round++)
				{
					for (size_t Counter = 0; Counter < KDSB.size(); Counter++)
					{
						MixWithAddSubtract(Counter, RandomIndex);
						RandomAccessMix(Counter, RandomIndex);
						ComplexMix(Counter, RandomIndex);
					}
				}

				State[i] ^= KDSB[i] & 0xFFFF0000;
				State[i] += KDSB[i] & 0x0000FFFF;
			}

			// 1 32-bits unpacking into 4 8-bits
			for (size_t i = 0; i < State.size() && (i * 4) < Bytes.size(); i++)
			{
				std::uint32_t value = State[i];
				for (int j = 0; j < 4 && ((i * 4) + j) < Bytes.size(); j++)
				{
					Bytes[i * 4 + j] = static_cast<std::uint8_t>(value & 0xFF);
					value >>= 8;
				}
			}
		}

		void InitialState(const std::vector<std::uint8_t>& KeyBytes)
		{
			ResetState();
			KeySchedule(KeyBytes);
			IsKeyUsed = true;
		}

		void ResetState()
		{
			const std::vector<uint32_t> ZeroNumbers(128, 0x00);

			memmove(State.data(), ZeroNumbers.data(), 128 * sizeof(uint32_t));
			memmove(KDSB.data(), ZeroNumbers.data(), 128 * sizeof(uint32_t));
			IsKeyUsed = false;
		}
	};

	class TitanWallBlockCipher : public CryptographicFunctions
	{
	private:
		// 伪哈达玛德变换
	void PHT(std::uint32_t& a, std::uint32_t& b)
	{
		std::uint32_t temp_a = a;
		// 32位模运算
		a = (a + b) & 0xFFFFFFFF;
		b = (temp_a + 2 * b) & 0xFFFFFFFF;
	}

	// 伪哈达玛德变换的逆变换
	void InversePHT(std::uint32_t& a, std::uint32_t& b)
	{
		std::uint32_t temp_a = a;
		// 32位模运算
		a = (2 * a - b) & 0xFFFFFFFF;
		b = (b - temp_a) & 0xFFFFFFFF;
	}

public:

	TitanWallBlockCipher() = default;

	explicit TitanWallBlockCipher(const std::vector<std::uint8_t>& KeyBytes)
	{
		// 生成子密钥
		KeySchedule(KeyBytes);
	}

	~TitanWallBlockCipher()
	{
		const std::vector<uint32_t> ZeroNumbers(128, 0x00);
		memmove(KDSB.data(), ZeroNumbers.data(), 128 * sizeof(uint32_t));
	}

	// 加密函数
	std::vector<std::uint32_t> Encrypt(const std::vector<std::uint32_t>& plaintext)
	{
		if(plaintext.size() % 8 != 0)
		{
			std::cout << "The number of data blocks cannot be aligned. Please perform preprocessing before proceeding." << std::endl;
			throw std::logic_error("The number of data blocks cannot be aligned. Please perform preprocessing before proceeding.");
		}

		std::vector<std::uint32_t> ciphertext(plaintext.size());

		const size_t SubkeysSize = KDSB.size();

		for (size_t i = 0; i < plaintext.size(); i += 8)
```

```
{
        uint32_t A = plaintext[i];
        uint32_t B = plaintext[i + 1];
        uint32_t C = plaintext[i + 2];
        uint32_t D = plaintext[i + 3];
        uint32_t E = plaintext[i + 4];
        uint32_t F = plaintext[i + 5];
        uint32_t G = plaintext[i + 6];
        uint32_t H = plaintext[i + 7];

        B += KDSB[0];
        D += KDSB[1];
        F += KDSB[2];
        H += KDSB[3];

        //128 - 1 = 127
        //(127 - 3) / 2 = 62
        for (size_t j = 1; j <= 62; j++)
        {
                // 替换函数S
                std::uint32_t t = B ^ ((B << 1) + 1);
                std::uint32_t u = D ^ ((D << 2) + 1);
                std::uint32_t v = F ^ ((F << 3) + 1);
                std::uint32_t w = H ^ ((H << 4) + 1);

                A = Bits32RotateRight(A - t, w % 32) + KDSB[2 * j];
                C = Bits32RotateRight(C ^ u, v % 32) + KDSB[2 * j + 1];
                E = Bits32RotateRight(E ^ v, u % 32) + KDSB[2 * j + 2];
                G = Bits32RotateRight(G + w, t % 32) + KDSB[2 * j + 3];

                B += KDSB[j % KDSB.size()];
                D ^= KDSB[(j + 1) % KDSB.size()];
                F ^= KDSB[(j + 2) % KDSB.size()];
                H -= KDSB[(j + 3) % KDSB.size()];

                // 置换函数P
                std::tie(F, D, B, H, A, G, E, C) = std::make_tuple(A, B, C, D, E, F, G, H);

                // 伪哈达玛德变换
                PHT(A, B);
                PHT(C, D);
                PHT(E, F);
                PHT(G, H);
        }

        A += KDSB[SubkeysSize - 4];
        C += KDSB[SubkeysSize - 3];
```

```
                E += KDSB[SubkeysSize - 2];
                G += KDSB[SubkeysSize - 1];

                ciphertext[i] = A;
                ciphertext[i + 1] = B;
                ciphertext[i + 2] = C;
                ciphertext[i + 3] = D;
                ciphertext[i + 4] = E;
                ciphertext[i + 5] = F;
                ciphertext[i + 6] = G;
                ciphertext[i + 7] = H;
        }

        return ciphertext;
}

// 解密函数
std::vector<std::uint32_t> Decrypt(const std::vector<std::uint32_t>& ciphertext)
{
        if(ciphertext.size() % 8 != 0)
        {
                std::cout << "The number of data blocks cannot be aligned. Please perform preprocessing before proceeding." << std::endl;
                throw std::logic_error("The number of data blocks cannot be aligned. Please perform preprocessing before proceeding.");
        }

        std::vector<std::uint32_t> plaintext(ciphertext.size());

        const size_t SubkeysSize = KDSB.size();

        for (size_t i = 0; i < ciphertext.size(); i += 8)
        {
                uint32_t A = ciphertext[i];
                uint32_t B = ciphertext[i + 1];
                uint32_t C = ciphertext[i + 2];
                uint32_t D = ciphertext[i + 3];
                uint32_t E = ciphertext[i + 4];
                uint32_t F = ciphertext[i + 5];
                uint32_t G = ciphertext[i + 6];
                uint32_t H = ciphertext[i + 7];

                G -= KDSB[SubkeysSize - 1];
                E -= KDSB[SubkeysSize - 2];
                C -= KDSB[SubkeysSize - 3];
                A -= KDSB[SubkeysSize - 4];

                //128 - 1 = 127
                //(127 - 3) / 2 = 62
```

```
            for (size_t j = 62; j >= 1; j--)
            {
                // 伪哈达玛德变换的逆变换
                InversePHT(A, B);
                InversePHT(C, D);
                InversePHT(E, F);
                InversePHT(G, H);

                // 置换函数P的逆操作
                std::tie(A, B, C, D, E, F, G, H) = std::make_tuple(F, D, B, H, A, G, E, C);

                // 替换函数S的逆操作
                H += KDSB[(j + 3) % KDSB.size()];
                F ^= KDSB[(j + 2) % KDSB.size()];
                D ^= KDSB[(j + 1) % KDSB.size()];
                B -= KDSB[j % KDSB.size()];

                std::uint32_t t = B ^ ((B << 1) + 1);
                std::uint32_t u = D ^ ((D << 2) + 1);
                std::uint32_t v = F ^ ((F << 3) + 1);
                std::uint32_t w = H ^ ((H << 4) + 1);

                G = Bits32RotateLeft(G - KDSB[2 * j + 3], t % 32) - w;
                E = Bits32RotateLeft(E - KDSB[2 * j + 2], u % 32) ^ v;
                C = Bits32RotateLeft(C - KDSB[2 * j + 1], v % 32) ^ u;
                A = Bits32RotateLeft(A - KDSB[2 * j], w % 32) + t;
            }

            H -= KDSB[3];
            F -= KDSB[2];
            D -= KDSB[1];
            B -= KDSB[0];

            plaintext[i] = A;
            plaintext[i + 1] = B;
            plaintext[i + 2] = C;
            plaintext[i + 3] = D;
            plaintext[i + 4] = E;
            plaintext[i + 5] = F;
            plaintext[i + 6] = G;
            plaintext[i + 7] = H;
        }

        return plaintext;
    }
};
```

```
void TestTitanWallBlockCipher()
{
        // 测试数据
        std::vector<std::uint32_t> plaintext = {0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210, 0xAAAAAAAA, 0x55555555, 0x80000000, 0x11111111, 0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210, 0xAAAAAAAA,
        std::vector<std::uint8_t> key = {0x01, 0x23, 0x45, 0x67, 0x89, 0xAB, 0xCD, 0xEF};

        TitanWallBlockCipher cipher(key);

        // 加密
        std::vector<std::uint32_t> ciphertext = cipher.Encrypt(plaintext);

        // 解密
        std::vector<std::uint32_t> decryptedText = cipher.Decrypt(ciphertext);

        // 验证
        if(plaintext != decryptedText)
        {
                std::cout << "测试失败！" << std::endl;
                assert(false);
        }

        std::cout << "加密前的数据：";
        for (auto val : plaintext)
                std::cout << std::hex << val << " ";
        std::cout << std::endl;

        std::cout << "加密后的数据：";
        for (auto val : ciphertext)
                std::cout << std::hex << val << " ";
        std::cout << std::endl;

        std::cout << "解密后的数据：";
        for (auto val : decryptedText)
                std::cout << std::hex << val << " ";
        std::cout << std::endl;

        std::cout << "测试成功！" << std::endl;

        // 输出KDSB的内容
        /*for (size_t i = 0; i < titanwall.KDSB.size(); ++i) {
                std::cout << std::dec << "KDSB[" << i << "] = " << std::hex << titanwall.KDSB[i] << std::endl;
        }*/
}

void TestTitanWallStreamCipher()
{
```

```cpp
// 1. Test Key Initialization
std::vector<std::uint8_t> key = {0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC, 0xDE, 0xF0};
TitanWallStreamCipher cipher(key);

// 2. Test Pseudo Random Byte Generation
std::vector<std::uint8_t> bytes(32, 0); // Initialize a vector of 32 bytes with zeros
cipher.GeneratePseudoRandomBytes(bytes);

// Ensure that the bytes have been modified
bool isModified = false;
for (auto byte : bytes)
{
        if (byte != 0)
        {
                isModified = true;
                break;
        }
}

assert(isModified); // Assert that the bytes have been modified

// 3. Test Reset State
cipher.ResetState();
cipher.InitialState(key);
std::vector<std::uint8_t> resetBytes(32, 0);
cipher.GeneratePseudoRandomBytes(resetBytes);

// Ensure that the bytes are still zeros after reset
bool isReset = true;
for (auto byte : resetBytes)
{
        if (byte != 0)
        {
                isReset = false;
                break;
        }
}

assert(!isReset); // Assert that the bytes have been modified even after reset

// 4. Encrypt data
std::string originalText = "Hello, TitanWall!";

// Print Original Text
std::cout << "Original Text: " << originalText << std::endl;

std::vector<std::uint8_t> data(originalText.begin(), originalText.end());
```

```cpp
        std::vector<std::uint8_t> random(originalText.size(), 0);
        std::vector<std::uint8_t> encryptedData(random.size(), 0);

        cipher.ResetState();
        cipher.InitialState(key);
        cipher.GeneratePseudoRandomBytes(random);
        for (size_t i = 0; i < data.size(); ++i)
        {
                encryptedData[i] = data[i] ^ random[i];
        }

        // Print encrypted data
        std::cout << "Encrypted Data: ";
        for (auto byte : encryptedData)
        {
                std::cout << std::hex << static_cast<int>(byte) << " ";
        }
        std::cout << std::endl;

        // 5. Decrypt data
        cipher.ResetState();
        cipher.InitialState(key);
        cipher.GeneratePseudoRandomBytes(random);
        std::vector<std::uint8_t> decryptedData(encryptedData.size(), 0);
        for (size_t i = 0; i < random.size(); ++i)
        {
                decryptedData[i] = encryptedData[i] ^ random[i];
        }

        // Print decrypted data
        std::cout << "Decrypted Data: ";
        for (auto byte : decryptedData)
        {
                std::cout << std::hex << static_cast<int>(byte) << " ";
        }
        std::cout << std::endl;

        std::string decryptedText(decryptedData.begin(), decryptedData.end());

        // Print Recovered Text
        std::cout << "Recovered Text: " << decryptedText << std::endl;

        // Ensure that the decrypted data matches the original plaintext
        assert(decryptedText == originalText);
}
```