The Pathosis Hash Function: Approximating the Dirichlet Function for Secure Hash Generation

Introduction:

In the world of digital security, truly random numbers with a uniform distribution are a precious commodity.
While pseudorandom number generators, like the MT19937 algorithm, offer us an efficient and rapid means of generating random numbers, we still harbor reservations about whether the numbers they produce truly follow a uniform distribution.
To ensure the randomness of a uniform distribution, we turn to the chaotic systems found in physics — specifically, the double pendulum system.
The motion of a double pendulum is rife with chaos and unpredictability, providing us with a unique method to simulate randomness from the real world.
By marrying this physical simulation with digital algorithms, we aim to construct a hash function that's both more secure and dependable.

Part One: From Finite Field Elements to Real Numbers

The Pathosis hash function begins with an element `x` from the finite field $\mathbb{F}_2^{64}$, typically generated by the Mersenne Twister algorithm. This element is a pseudorandom 64-bit integer, which needs to be converted into a uniformly distributed real number.

As previously mentioned, we employ a double pendulum system, simulated through the `SimulateDoublePendulum` class, to generate a real number mapped between 0 and 1 (inclusive). The initial seed for this pseudorandom number generator is our finite field element `x`.

Now, to map `x` to a uniformly distributed real number x' within the real domain $\mathbb{R}$, we normalize the output of the `SimulateDoublePendulum` class. This conversion is illustrated as follows:

$$
\begin{aligned}
pendulum\_min &= 1024 \times 10^{-64} \\
pendulum\_max &= 1024 \times 10^{64} \\
x' = pendulum\_min + (pendulum\_max - pendulum\_min) &\times \frac{\sin(\text{SimulateDoublePendulum.operator}()(x)) + 1.0}{2.0}
\end{aligned}
$$

A key aspect to consider is that while the pseudorandom number generator and double pendulum simulation are deterministic, the sensitive dependence on initial conditions, commonly known as chaos, makes it nearly impossible to predict x' from x without executing the entire computational sequence.

Moreover, due to the complex, nonlinear dynamics of the double pendulum system, the mapping from x to x' is highly irregular; slight variations in x can lead to significant differences in x'. This characteristic is crucial for providing the required uniform distribution of x' values.

To summarize, the input x and its transformation into the real number x' through a pseudorandom number generator (PRNG) and mapping function are as follows:

- x is an element from the finite field $\mathbb{F}_2^{64}$, represented as a 64-bit binary number.
- x' is the result of processing x through a pseudorandom number generator and mapping function, which maps it onto a uniformly distributed real number.

Part Two: Using `x'` with the Dirichlet Function for the Pathosis Hash Function

The real number `x'` generated in the interval `[pendulum_min, pendulum_max]` is now ready to be input into the next part of the Pathosis hash function. This second part involves creating a hash function based on the Dirichlet function, which is renowned for its property of being discontinuous everywhere. `y` belongs to the real numbers, and we ensure that `y'` falls within the interval `[0, 1]`.

The Dirichlet function is typically defined as:

$$
D(x) = \begin{cases} 1 & \text{if } x \in \mathbb{Q} \text{ "x is rational"} \\ 0 & \text{if } x \notin \mathbb{Q} \text{ "x is irrational"} \end{cases}
$$

Initial Mapping

Firstly, the transformation from the finite field element `x` to the uniformly distributed real number `x'` in the interval `[0,1]` was described in Part One. It looks like this:

$$x \in \mathbb{F}_2^{64}$$
$$x' \in \mathbb{R}$$
$$x' = mapping(x) = SimulateDoublePendulum.operator()(PRNG(x))$$

Next, we examine how to use this `x'` to generate the final hash value.

Modified Dirichlet Function

Initially, the traditional Dirichlet function $D(x)$ is mathematically defined to take the value 1 on rationals and 0 on irrationals, thus exhibiting a form of extreme discontinuity. In the fields of cryptography and data security, such discontinuity is considered an ideal characteristic for constructing secure hash functions because it increases the hash function's sensitivity to minor changes in input.
However, for our Pathosis hash function, we require a different approach to simulate the discontinuity of the Dirichlet function and how to apply this property to generate a cryptographically secure hash value.
To achieve a similar discontinuity as the Dirichlet function while being suitable for digital security and hash algorithms, we have reimagined the original concept.

Specifically, we define a new function $y = D(x')$ computed as:

$$y = \lim_{j \to n} \lim_{k \to n} \cos\left(j! \pi x'\right)^{2k}$$

Here, the parameters j, k, and n are large numbers to ensure that the function's output fluctuates between 0 and 1, which is achieved through complex domain trigonometric functions and limit operations.

The essence of this method lies in using factorial-growth angular frequencies $j!\pi$, along with squaring to increase output stability and reduce numerical errors.

Mathematical Explanation of the Computation Process

In the practical code implementation, we first define a complex number z with a real part of 0 and an imaginary part of $j!\pi x'$, i.e., $z = 0 + i \cdot j!\pi x'$. Next, we compute e^z, or $\exp(0 + i \cdot j!\pi x')$.
According to Euler's formula, $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, meaning $\exp(z)$'s real part is $\cos(j!\pi x')$, and the imaginary part is $\sin(j!\pi x')$.

However, we further take $\cos(j!\pi x')^{2k}$, which is done to magnify the computational effects of discontinuity, making the output extremely sensitive to any slight changes in x'.
When j and k are sufficiently large, this expression will oscillate rapidly between 0 and 1, which is the desired "hash collision resistance" characteristic, where different input values yield vastly different output hash values.

Handling Numerical Stability

In actual programming implementation, special attention must be paid to numerical stability issues.
As the code indicates, when results are NaN (Not a Number) or infinite, the function directly returns 0.0, which handles potential anomalies caused by extreme inputs or computational errors.
Furthermore, when results are below the machine's floating-point precision (epsilon), it also returns 0.0 to avoid errors due to floating-point precision issues.

Remapping to Uniform Distribution

Since the calculation of y might lead to biased output values, we need to ensure that the final hash value is uniformly distributed.
This can be achieved by remapping y to a new

uniformly distributed y', ensuring each bit is independently and uniformly generated.

Typically, such a mapping can be defined as:

$$y' = Uniform01(y)$$

Here, the $Uniform01(y)$ function should be one that can map y to a uniformly distributed (fairly) [0,1] interval on y'.
For example, this could be a simple normalization step or a more complex statistical transformation to ensure output distribution uniformity.

Further Discussion on the Basic Details of Part One and Part Two

Pseudorandom Number Generation Using the Mersenne Twister (PRNG):

The initial step of the Pathosis hash function is to generate pseudorandom numbers using the 64-bit version of the Mersenne Twister algorithm (MT19937). Recognized for its reliability, long period of 2^19937-1, and impressive speed and space efficiency, this PRNG forms the foundation of our unique hash function.

The seed for the PRNG is set through the `void seed()` function, typically corresponding to the input value of the hash function. Pseudorandom numbers are generated by the `uint64_t prng()` function.

The MT19937 algorithm operates based on recursive relationships as follows:

$$x_{k+n} = x_{k+m} \oplus ((x_k^u | x_{k+1}^l)A)$$

where:

- $x_k^u = x_k \gg r$ represents the upper $w - r$ bits of x_k,
- $x_{k+1}^l = x_{k+1} \ll (w - r)$ represents the lower r bits of x_{k+1},
- $\oplus$ is the bitwise XOR operation,
- $|$ is the bitwise OR operation,
- $\gg$ and $\ll$ are bitwise right and left shifts,
- A is a matrix designed to ensure the algorithm's long period and complex structure.

Here, w, n, m, and r represent the word size (in bits), degree of recursion, middle word, and a separation point of a word (where one word equals w bits), respectively, with values $w = 64$, $n = 312$, $m = 156$, and $r = 31$.

Utilizing Chaos Theory for Uniform Distribution

We've covered the basic framework of generating pseudorandom numbers using the Mersenne Twister algorithm. This section focuses on ensuring that the input to the hash function is transformed into an unpredictable output through a highly complex computational process.

Chaos theory in mathematics and physics describes the behavior of a class of systems that are extremely sensitive to initial conditions, where even minor changes can lead to vastly different outcomes, known as the "butterfly effect." In cryptography, this characteristic of chaos theory can be utilized to enhance the security of hash functions by making the output highly sensitive to the input, thereby increasing the hash's unpredictability and non-determinism.

The Pathosis hash function employs the chaotic properties of a double pendulum system to enhance its randomness. A double pendulum system is a classic physical example used to demonstrate chaos theory.
A double pendulum consists of two pendulum arms, where one arm is connected to another, with each arm's motion influenced not only by gravity but also by the motion of the other arm.
The dynamics of such a system are highly complex, where even minimal disturbances can lead to significant changes in system behavior.

Integration of the Double Pendulum System's Mathematical Model with the Hash Algorithm

In the `SimulateDoublePendulum` class, we simulate a double pendulum system by precisely calculating the tensions and velocities of the two pendulum arms to implement a continuously updating chaotic system. The mathematical description of this system includes:

1. **System Parameters**: The length (`length1` , `length2`), mass (`mass1` , `mass2`), and swinging angles (`tension1` , `tension2`) of each pendulum arm.
2. **Dynamical Equations**: Using Newton's second law and the dynamics of pendulum motion, combined with the effects of gravity and inter-arm interaction forces, calculate each time step's arm accelerations and velocities.
3. **Numerical Solution Method**: Iteratively update the state of the double pendulum system using a simple Euler integration method (with a time step size of `hight`).

The system's initial state is set by a value generated from a binary key sequence. The key sequence directly impacts the initial angles and speeds of the pendulum, thus determining the trajectory of system evolution.
Each time the `generate()` function is called, the double pendulum system runs for a specified number of time steps based on the current state, then outputs a pseudorandom number based on the current state of the double pendulum.
This process ensures that even with the same input value, the results of each run will differ due to the chaotic nature of the system, depending on the numerical value of the simulation time.

In the `run_system()` function of the `SimulateDoublePendulum` class, the core simulation of the double pendulum system is conducted by numerically solving the dynamical equations of the two interacting pendulum arms

. These equations, based on Newton's second law and the dynamics of the double pendulum, require the calculation of each arm's angular acceleration (alpha1 and alpha2), angular velocity (velocity1 and velocity2), and angular position (tension1 and tension2).

Dynamics Equation Derivation

The double pendulum system includes two pendulum arms, each able to rotate freely. Assuming:

- θ_1 and θ_2 are the angles of the first and second pendulum arms with the vertical line,
- L_1 and L_2 are the lengths of the pendulum arms,
- m_1 and m_2 are the masses of the pendulum arms.

The dynamical equations for the double pendulum system can be derived using the Lagrangian mechanics formula. These equations consider the effects of gravity, tension, and the inertial forces caused by swinging. For each pendulum arm, we have:

1. **Angular acceleration of the first pendulum arm (α_1)**:

$$\alpha_1 = \frac{-g(2m_1 + m_2)\sin(\theta_1) - m_2 g \sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)m_2(\omega_2^2 L_2 + \omega_1^2 L_1 \cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

   Here g is the acceleration due to gravity, ω_1 and ω_2 are the angular velocities of the pendulum arms.
2. **Angular acceleration of the second pendulum arm (α_2)**:

$$\alpha_2 = \frac{2\sin(\theta_1 - \theta_2)(\omega_1^2 L_1(m_1 + m_2) + g(m_1 + m_2)\cos(\theta_1) + \omega_2^2 L_2 m_2 \cos(\theta_1 - \theta_2))}{L_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

Numerical Implementation of the Double Pendulum Chaotic System

In the `run_system()` function, the above equations are numerically solved through the following steps:

1. **Initialization**: Set the lengths and masses of the pendulum arms, and the initial tensions and velocities.
2. **Time Iteration**: For a specified number of time steps, use the simple Euler method to update the state of the pendulum arms:
   - Calculate the next time step's angular accelerations based on the current angles (tensions) and angular velocities.
   - Update angular velocities: $\omega_{new} = \omega + \alpha \times \text{time step length}$
   - Update angles: $\theta_{new} = \theta + \omega_{new} \times \text{time step length}$
3. **State Saving**: In initialization mode, the system's final state (e.g., angles and velocities) is saved as a backup, so it can be reset to these values.

System Initialization

The `initialize` function is a key component of the `SimulateDoublePendulum` class, used to convert a binary key sequence into system parameters and set initial conditions, which will influence the trajectory and behavior of the double pendulum system. This function's implementation reflects the complex process of encoding binary data into a physical model.

1. Splitting the Binary Key Sequence

- Input: A `std::vector<std::int8_t>` array containing only 0s or 1s, representing the binary key sequence.
- First, this sequence is evenly split into four parts, stored in `binary_key_sequence_2d` , a two-dimensional vector array. Each part contains 1/4 of the total length.

2. Generating Parameterized Key Sequences

- New binary sequences `binary_key_sequence_2d_param` are generated using the XOR operation. In this step, the four parts of the original key sequence are combined in pairs through the XOR operation to generate new sequences.
- For example, `binary_key_sequence_2d_param[0]` is obtained by XORing the corresponding bits of `binary_key_sequence_2d[0]` and `binary_key_sequence_2d[1]` .

3. Setting System Data with Parameterized Key Sequences

- This step converts

the binary sequence into actual parameters used to simulate the double pendulum system.

- By iterating through each bit of `binary_key_sequence_2d_param` , binary bits are converted into system parameters such as length and mass.
- For instance, the value of `SystemData[j]` is updated using the following formula:

$$\text{SystemData}[j] + 1 \times 2^{-i}, \text{ if } \text{binary\_key\_sequence\_2d\_param}[j][i] \text{ is } 1$$

where i is the index of the bit, j is the parameter index.

4. Calculating System Running Parameters and Initializing the System

- `radius` is calculated using a specific formula based on the content of `binary_key_sequence_2d_param[6]`:

$$\text{radius} + 1 \times 2^{4-i}, \text{ if } \text{binary\_key\_sequence\_2d\_param}[6][i] \text{ is } 1$$

  Here i is the index of the bit.
- `current_binary_key_sequence_size` is set to the length of the input binary sequence, converted into a long double-precision floating-point number.
- The `run_system` function is called in initialization mode, with the time parameter being the product of `radius` and `current_binary_key_sequence_size`, rounded to the nearest integer.

Part Three: Secure Hashing Using the Dirichlet Function

In the previous sections, we discussed the initial steps of the Pathosis hash function, involving pseudorandom number generation and mapping finite field elements to uniformly distributed real numbers. Now, let's delve deeper into the final part of the algorithm, which employs the Dirichlet function for secure hash generation.

Hash Space Mapping H_D

The design of the hash function H_D aims to map the continuous `y'` to the discrete elements of the hash space H.
Here, each bit of the hash value $H_D(y')_i$ is defined as follows:

$$H_D(y')_i = [y' \geq 0.5]$$

The indicator function $[y' \geq 0.5]$ generates a binary sequence that conforms to the Bernoulli distribution.
Where i is the index of the bit (from 0 to 63), and the Iverson bracket $[y'_i \geq 0.5]$ is an indicator function that results in 1 if y'_i is equal to or greater than 0.5 (threshold), otherwise 0.
In this way, each bit of y' is mapped to a binary bit, which is then combined to form the final 64-bit hash value.
For instance, the threshold for each bit could be uniformly distributed from 0 to 1, so that minor changes in `y'` can significantly affect the final hash value.

"Stretching" y' the from interval [0, 1] to [0, h], and rounding ensures we land on some integer within this range, thus mapping `y'` to an element of our hash space H.

Generation of the Hash Value

The final hash value $hash$ is determined by each bit of $H_D(y')_i$ as follows:

$$x' \in \mathbb{R}, hash \in \mathbb{F}_2^{64}$$
$$hash = H_D(y') = \sum_{i=0}^{63} 2^i \cdot H_D(y')_i$$

Addressing Bit Distribution Imbalance with the Pathosis Hash Function

Since the Pathosis hash function utilizes a chaotic system close to a true random number sequence, the generated bit sequence may have a certain bias in the number of 1s and 0s. Ideally, a good hash function should produce a bit sequence that is nearly uniformly distributed to ensure cryptographic security. For our system, issues of uneven distribution can be addressed by combining multiple hash values and using XOR operations, thus enhancing the randomness and security of the hash function.

Specifically, when encountering output biases due to the inherent properties of the chaotic system, a strategy of hashing multiple times and XORing the results can be employed to balance the distribution of 0s and 1s. This method is highly effective in enhancing the randomness of hash outputs, as the XOR operation is essentially a binary addition that can offset biases in multiple hashing results to produce a more uniform outcome.

Algorithm Description

At this stage, we handle multiple inputs x'_i, which can be sequential or random numbers produced within a finite field or ring. Each x'_i is processed through our hash function H_D to generate a preliminary hash value. These hash values are then XORed together to form the final secure hash.

The formula is as follows:

$$secure\_hash = \bigoplus_{i=0}^{N} hash_i$$

Here, $\bigoplus$ represents the XOR operation, N is the total number of inputs, and each $hash_i$ is the hash value for the individual input x_i'.

Implementation Steps

1. **Generating the Input Sequence**: Depending on the required level of security, generate a series of x_i'. These x_i' can be random or sequential numbers from a finite field or ring.
2. **Individual Hash Processing**: Use the Pathosis hash function H_D for each x_i' to compute their respective hash values.
3. **XOR Combination**: XOR all the individual hash values to generate the final secure hash value.

Enhancing Pathosis Hash Function's Handling of Non-Uniform Hash Values Using ARX Operations

In cryptography and the design of hash algorithms, ARX (Modular Addition, Rotation, and XOR) operations are a common technique used to build complex cryptographic functions. Utilizing the nonlinearity and the difficulty of reversing these operations, we can further enhance the security and randomness of our hash function. By combining multiple non-uniform Pathosis hash values, we can generate a more secure, random, and uniform final hash value through ARX operations.

Specific Implementation of ARX Operations

Given the potential non-uniform hash values produced by the Pathosis hash function, we can opt to generate four such hash values and then mix them through the following ARX operations:

1. **Modular Addition**: Add two hash values together, ensuring the result stays within a fixed numeric range, like using a 32-bit integer range.
2. **Rotation**: Rotate the bits of the hash value, which helps to disperse information across the bits, increasing the complexity of the hash function.
3. **XOR**: Perform an XOR operation between two hash values, a common method used to create new randomness and unpredictability in hash values.

Process of Mixing Hash Values

We first generate four Pathosis hash values $hash_0, hash_1, hash_2, hash_3$, and then mix them using ARX operations as follows:

1. **Step One**: $result1 = (hash_0 + hash_1) \mod 2^{64}$
2. **Step Two**: $result2 = (hash_2 \oplus hash_3)$
3. **Step Three**: $result3 = result1 \oplus (result1 \lll r) \oplus result2$, where r is the number of bits to rotate, adjusted as needed.

Pathosis哈希函数：用于安全哈希生成的狄利克雷函数近似

引言：

在数字安全的世界中，均匀分布的真正随机数是宝贵的资源。
尽管伪随机数生成器，如MT19937算法，为我们提供了高效和快速的随机数生成方法，但我们仍然对其产生的数字是否真正服从均匀分布持有一定的怀疑。
为了确保均匀分布的随机性，我们转向物理学中的混沌系统——特别是双摆系统。
双摆的运动是充满混沌和不可预测性的，它为我们提供了一个独特的方法来模拟真实世界的随机性。
通过将这种物理模拟与数字算法相结合，我们旨在创建一个更加安全和可靠的哈希函数。

第一部分：从有限域元素到实数

Pathosis哈希函数从有限域 $\mathbb{F}_2^{64}$ 中的元素 `x` 开始，这通常是由Mersenne Twister算法生成的。这个元素是一个伪随机的64位整数，需要转换为一个均匀分布的实数。

正如前文所提到的，我们使用一个双摆系统，通过 `SimulateDoublePendulum` 类进行模拟，来生成一个映射到0和1（包含）之间的实数。这个伪随机数生成器的初始种子就是我们的有限域元素 `x` 。

现在，为了将 `x` 映射到均匀分布的实数域 $\mathbb{R}$ 中的实数 `x'`，我们对 `SimulateDoublePendulum` 类的输出进行标准化。
这个转换如下所示：

$$pendulum\_min = 1024 \times 10^{-64}$$
$$pendulum\_max = 1024 \times 10^{64}$$
$$x' = pendulum\_min + (pendulum\_max - pendulum\_min) \times \frac{sin(\text{SimulateDoublePendulum.operator}()(x)) + 1.0}{2.0}$$

需要考虑的关键方面是，虽然伪随机数生成器和双摆模拟是确定的，但初始条件的敏感依赖性，即所谓的混沌，使得从 `x` 预测 `x'` 几乎不可能，除非执行整个计算序列。

此外，由于双摆系统的复杂、非线性动力学，从 `x` 到 `x'` 的映射是高度不规则的， `x` 的微小变化可能导致 `x'` 的显著差异。这一特性对于提供 `x'` 值的所需均匀分布至关重要。

输入 x 和它通过伪随机数生成器（PRNG）和映射函数转换成实数 x' 的过程：

- x 是一个来自有限域 $\mathbb{F}_2^{64}$ 的元素，表示为一个64位的二进制数。
- x' 是 x 经过一个伪随机数生成器和映射函数处理后的结果，映射到一个均匀分布的实数上。

第二部分：使用 `x'` 和狄利克雷函数为Pathosis哈希函数

在区间 `[pendulum_min, pendulum_max]` 中生成的实数 `x'` 现在已经准备好作为Pathosis哈希函数的下一部分的输入。
第二部分涉及基于狄利克雷函数创建一个哈希函数，该函数因其在任何地方都不连续的属性而著称。
`y` 是属于实数 。 我们确保 `y‘ 落在 [0, 1]` 的区间内。

狄利克雷函数通常定义为：

$$D(x) = \begin{cases} 1 & \text{如果 } x \in \mathbb{Q} \text{ "x是有理数"} \\ 0 & \text{如果 } x \notin \mathbb{Q} \text{ "x是无理数"} \end{cases}$$

初始映射

首先，我们从有限域元素 `x` 到 `[0,1]` 区间的均匀分布实数 `x'` 的转换已经在第一部分描述。

它看起来像这样：

$$x \in \mathbb{F}_2^{64}$$
$$x' \in \mathbb{R}$$
$$x' = mapping(x) = SimulateDoublePendulum.operator()(PRNG(x))$$

接下来是如何利用这个 `x'` 来生成最终的哈希值。

修改后的狄利克雷函数

首先，传统的狄利克雷函数 $D(x)$ 在数学上定义为在有理数上取值1，而在无理数上取值0，从而展示了一种极端的非连续性。
而在加密学和数据安全的领域，这种非连续性特性被认为是构建安全哈希函数的一个理想特性，因为它可以增加哈希函数对输入值微小变化的敏感性。
但对于我们的Pathosis哈希函数，我们需要一种不同的方法来模拟狄利克雷函数的非连续性，以及如何将这种性质应用于生成一个加密安全的哈希值。
为了实现类似狄利克雷函数的非连续性，但同时又适用于数字安全和哈希算法的需要，我们对原始的概念进行了改造。

具体地，我们定义一个新的函数 $y = D(x')$，计算方法为：

$$y = \lim_{j \to n} \lim_{k \to n} \cos\left(j! \pi x'\right)^{2k}$$

在这里，参数 j、k 和 n 是大数，以确保函数的输出在0和1之间波动，这是通过复数域的三角函数和极限运算实现的。

这种方法的核心在于使用阶乘增长的角频率 $j!\pi$，以及通过平方的方式增加输出稳定性和减少数值误差。

计算过程的数学解释

在实际的代码实现中，我们首先定义一个复数 z ，其实部为0，虚部为 $j!\pi x'$，即 $z = 0 + i \cdot j!\pi x'$。接着，计算 e^z，也就是 $\exp(0 + i \cdot j!\pi x')$ 。
根据欧拉公式，$e^{i\theta} = \cos(\theta) + i\sin(\theta)$，这就意味着 $\exp(z)$ 的实部为 $\cos(j!\pi x')$，而虚部为 $\sin(j!\pi x')$。

然而，我们进一步取 $\cos(j!\pi x')^{2k}$，这样做的目的是为了放大计算中的非连续性效应，使得输出值对输入 x' 的任何微小变化都极其敏感。
当 j 和 k 足够大时，这个表达式的值将在0与1之间快速振荡，这就是我们所希望的“哈希碰撞抵抗”特性，即不同的输入值会产生迥然不同的输出哈希值。

对数值稳定性的处理

在实际的编程实现中，需要特别注意数值稳定性问题。
如代码所示，当计算结果为NaN（不是一个数）或无穷大时，函数直接返回0.0，这是为了处理可能因极端输入或计算误差导致的异常情况。
此外，当计算得到的结果小于计算机浮点数的最小精度（epsilon）时，同样返回0.0，以避免因浮点数精度问题带来的误差。

重映射到均匀分布

由于 y 的计算可能导致输出值的偏差，我们需要确保最终的哈希值是均匀分布的。
这可以通过重新将 y 映射到一个新的均匀分布的y'来实现，确保每个比特都是独立且均匀生成的。

通常这样的映射可以通过如下方式定义：

$$y' = Unifrom01(y)$$

这里的 $Unifrom01(y)$ 函数应该是一个能将 y 映射到一个均匀分布(公平地)的 [0,1] 区间内的 y' 上。
例如，这可以是一个简单的归一化步骤，或者一个更复杂的统计变换，确保输出分布的均匀性。

确信读者会对上述内容感到困惑，所以让我们继续探讨第一部分和第二部分的基本细节。

使用Mersenne Twister的伪随机数生成 (PRNG)：

Pathosis哈希函数的初始步骤是生成伪随机数，这是使用Mersenne Twister算法的64位版本（MT19937）实现的。
因其可靠性、长周期2^19937-1以及令人印象深刻的速度和空间效率而获得认可，这个PRNG构成了我们独特哈希函数的基础。

PRNG的种子，通过 `void seed()` 函数设置，通常对应于哈希函数的输入值。伪随机数是通过 `uint64_t prng()` 函数生成的。

MT19937算法基于递归关系原理运作，如下所示：

$$x_{k+n} = x_{k+m} \oplus ((x_k^u | x_{k+1}^l)A)$$

其中:

- $x_k^u = x_k \gg r$，表示 x_k 的上 $w-r$ 位，
- $x_{k+1}^l = x_{k+1} \ll (w-r)$，表示 x_{k+1} 的下 r 位，
- $\oplus$ 是位异或操作，
- $|$ 是位或操作，
- $\gg$ 和 $\ll$ 分别是位右移和位左移，
- A 是一个矩阵，旨在确保算法的长周期和复杂结构。

这里，w，n，m 和 r 分别是字大小（以比特数表示），递归度，中间字和一个字的分隔点（其中一个字等于 w 比特），其值分别为 $w = 64$，$n = 312$，$m = 156$ 和 $r = 31$。

利用混沌理论进行均匀分布

我们已经了解了使用Mersenne Twister算法生成伪随机数的基本框架，这一部分重点在于确保哈希函数的输入能够通过高度复杂的计算过程，转换成难以预测的输出

混沌理论在数学和物理学中描述了一类系统的行为，这些系统对初态非常敏感，即使是微小的变化也能导致迥然不同的结果，这种特性被称为“蝴蝶效应”。
在密码学中，混沌理论的这种特性可以被用来增加哈希函数的安全性，通过使输出对于输入极其敏感，从而提高哈希的非确定性和不可预测性。

Pathosis哈希函数利用双摆系统的混沌特性来增强其随机性。双摆系统是一个经典的物理例子，用以演示混沌理论。
一个双摆由两个摆臂组成，其中一个摆臂连接到另一个摆臂，每个摆臂的运动不仅受到重力的影响，还受到另一个摆臂运动的影响。
这种系统的动力学非常复杂，即使是极小的变动也能导致系统行为的巨大变化。

双摆系统的数学模型与哈希算法的整合

在 `SimulateDoublePendulum` 类中，我们模拟了一个双摆系统，通过精确计算两个摆臂的张力和速度，来实现一个持续更新状态的混沌系统。此系统的数学描述如下：

1. **系统参数**：每个摆臂的长度（ `length1` ， `length2` ）、质量（ `mass1` ， `mass2` ）以及摆动的角度（ `tension1` ， `tension2` ）。
2. **动力学方程**：利用牛顿第二定律和摆动的动力学方程，结合重力和摆臂间相互作用力的作用，计算每个时间步的摆臂加速度和速度。
3. **数值解法**：通过一个简单的欧拉积分方法（时间步长为 `hight` ），对双摆系统的状态进行迭代更新。

系统的初始状态由一个由二进制密钥序列生成的数值设定。密钥序列直接影响了双摆的初始角度和速度，进而决定了系统演化的轨迹。
每次调用 `generate()` 函数时，双摆系统根据当前状态运行一定的时间步，然后输出一个基于双摆当前状态的伪随机数。
这一过程保证了即使是同一个输入值，只要所在的模拟时间数值不同，由于系统的混沌性，每次运行的结果也会有所不同。

在 `SimulateDoublePendulum` 类的 `run_system()` 函数中，模拟双摆系统的核心是通过数值方法求解两个相互作用的摆臂的动力学方程。这些方程基于牛顿第二定律和双摆的动力学特性，我们需要计算每个摆臂的角加速度（alpha1 和 alpha2），角速度（velocity1 和 velocity2），以及角位置（tension1 和 tension2）。

对系统进行运行

动力学方程的推导

双摆系统包含两个摆臂，每个摆臂可以自由旋转。假设：

- θ_1 和 θ_2 分别是第一和第二摆臂与垂直线的夹角，
- L_1 和 L_2 是摆臂的长度，
- m_1 和 m_2 是摆臂的质量。

双摆系统的动力学方程可以通过应用拉格朗日力学公式得到。这些方程考虑了重力、张力以及由于摆动造成的惯性力。对于每个摆臂，我们有：

1. **第一摆臂的角加速度** α_1：

$$\alpha_1 = \frac{-g(2m_1 + m_2)\sin(\theta_1) - m_2 g \sin(\theta_1 - 2\theta_2) - 2\sin(\theta_1 - \theta_2)m_2(\omega_2^2 L_2 + \omega_1^2 L_1 \cos(\theta_1 - \theta_2))}{L_1(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

   这里 g 是重力加速度，ω_1 和 ω_2 是摆臂的角速度。
2. **第二摆臂的角加速度** α_2：

$$\alpha_2 = \frac{2\sin(\theta_1 - \theta_2)(\omega_1^2 L_1(m_1 + m_2) + g(m_1 + m_2)\cos(\theta_1) + \omega_2^2 L_2 m_2 \cos(\theta_1 - \theta_2))}{L_2(2m_1 + m_2 - m_2\cos(2\theta_1 - 2\theta_2))}$$

双段摆锤混沌系统的数值实现

在 `run_system()` 函数中，上述方程通过以下步骤数值求解：

1. **初始化**：设置摆臂的长度、质量、初始张力和速度。
2. **时间迭代**：对于指定的时间步数，使用简单的欧拉方法更新摆臂的状态：
   - 根据当前的角度（张力）和角速度计算下一个时间步的角加速度。
   - 更新角速度：$\omega_{new} = \omega + \alpha \times \text{时间步长}$
   - 更新角度：$\theta_{new} = \theta + \omega_{new} \times \text{时间步长}$
3. **保存状态**：在初始化模式下，系统的最终状态（例如，角度和速度）保存为备份，以便可以重置到这些值。

对系统进行初始化

`initialize` 函数是 `SimulateDoublePendulum` 类的关键组成部分，用于将二进制密钥序列转换成系统参数，并设定初始条件，这些参数和条件将影响双摆系统的运动轨迹和行为。
此函数的实现体现了将二进制数据编码到一个物理模型中的复杂过程。

1. 分割二进制密钥序列

- 输入：一个仅包含0或1的 `std::vector<std::int8_t>` 数组，代表二进制密钥序列。
- 首先，将这个序列均匀分割成四个部分，存储到 `binary_key_sequence_2d` ， 一个二维向量数组中。每个部分包含总长度的1/4。

2. 生成参数化密钥序列

- 利用异或操作生成新的二进制序列 `binary_key_sequence_2d_param` 。此步骤中，原始密钥序列的四个部分两两组合，通过异或操作生成新的序列。
- 例如， `binary_key_sequence_2d_param[0]` 是由 `binary_key_sequence_2d[0]` 和 `binary_key_sequence_2d[1]` 的对应位异或得到的。

3. 用参数化密钥序列设置系统数据

- 该步骤将二进制序列转换为用于模拟双摆系统的实际参数。
- 通过迭代 `binary_key_sequence_2d_param` 的每一位，将二进制位转换为系统参数，如长度、质量等。
- 例如， `SystemData[j]` 的值通过以下公式更新：

$$\text{SystemData}[j] + 1 \times 2^{-i}，\text{如果 binary\_key\_sequence\_2d\_param}[j][i] \text{ 是 } 1$$

其中 i 是位的索引，j 是参数索引。

4. 计算系统的运行参数和初始化系统

- `radius` 通过一个特定的公式计算，该公式基于 `binary_key_sequence_2d_param[6]` 的内容：

$$\text{radius} + 1 \times 2^{4-i}，\text{如果 binary\_key\_sequence\_2d\_param}[6][i] \text{ 是 } 1$$

这里 i 是位的索引。

- `current_binary_key_sequence_size` 设置为输入二进制序列的长度，转换成长双精度浮点数。
- `run_system` 函数以初始化模式调用，时间参数是 `radius` 与 `current_binary_key_sequence_size` 的乘积，四舍五入到最接近的整数。

第三部分：使用狄利克雷函数进行安全哈希

在前面的部分中，我们讨论了Pathosis哈希函数的初始步骤，涉及伪随机数生成和将有限域元素映射到均匀分布的实数。
现在，我们深入探讨算法的最后部分，该部分使用狄利克雷函数进行安全哈希生成。

哈希空间映射 H_D

哈希函数 H_D 的设计旨在将连续的 `y'` 映射到哈希空间 H 的离散元素。
在这里，我们需要注意的是，每个比特的计算需要独立进行，以确保整个哈希值的安全性和随机性。
我们定义每个哈希值的比特 $H_D(y')_i$ 如下：

$$H_D(y')_i = [y' \geq 0.5]$$

指示函数$[y' \geq 0.5]$生成了符合伯努利分布的二进制序列。
其中 i 是比特的索引（从0到63），这里的 Iverson 括号 $[y'_i \geq 0.5]$ 是一个指示函数，如果 y'_i 的值大于或等于 0.5(阈值)，则结果为 1，否则为 0。
通过这种方式，我们将 y' 的每一位映射为一个二进制位，然后组合这些位以形成最终的 64 位哈希值。
例如，每个比特的阈值可以从0到1均匀分布，这样 `y'` 的小变化可以显著影响最终的哈希值。

将y'从区间 [0, 1] "拉伸" 到 [0, h]，取整确保我们落在这个范围内的某个整数上，从而将 `y'` 映射到我们的哈希空间 H 中的一个元素。

哈希值的生成

通过 $H_D(y')_i$ 的每一位来决定最终的哈希值 $hash$，如下：

$$x' \in \mathbb{R}, hash \in \mathbb{F}_2^{64}$$
$$hash = H_D(y') = \sum_{i=0}^{63} 2^i \cdot H_D(y')_i$$

使用Pathosis哈希函数处理不均匀比特分布问题

由于Pathosis哈希函数使用的是接近于真随机数序列的混沌系统，因此生成的比特序列中1和0的数量可能会有一定的偏差。在理想情况下，一个良好的哈希函数应该产生一个近乎均匀分布的比特序列，以保证密码学上的安全性。对于我们的系统，可以通过结合多个哈希值并使用异或运算来解决这种不均匀分布的问题，从而增强哈希函数的随机性和安全性。

具体地说，当我们遇到由于混沌系统的固有性质导致输出偏差时，可以采用多次哈希并将结果进行异或处理的策略，从而均衡0和1的分布。这种方法在增强哈希输出的随机性方面非常有效，因为异或操作本质上是一种二进制加法，能够在多次哈希的结果中抵消偏差，生成更加均匀的结果。

算法描述

在这个阶段，我们将处理多个输入 x_i'，这些输入可以是有限域或有限环中产生的顺序数字或随机数字。每一个 x_i' 都通过我们的哈希函数 H_D 进行处理，生成一个初步的哈希值。然后，我们将这些哈希值进行异或处理，以形成最终的安全哈希值。

公式如下：

$$secure\_hash = \bigoplus_{i=0}^{N} hash_i$$

这里，$\bigoplus$ 表示异或操作，N 是输入值的总数，每个 $hash_i$ 是单独输入 x_i' 的哈希值。

实施步骤

1. **生成输入序列**: 根据需要的安全级别，选择生成一系列的 x_i'。这些 x_i' 可以是有限域或有限环中的随机或顺序数字。
2. **单独哈希处理**: 对每个 x_i' 使用Pathosis哈希函数 H_D，计算得到各自的哈希值。
3. **异或合并**: 将所有单独的哈希值进行异或运算，生成最终的安全哈希值。

使用ARX操作加强Pathosis哈希函数的不均匀哈希值处理

在密码学和哈希算法的设计中，ARX（Modular Addition, Rotation, and XOR）操作是一种常见的方法，用于构建复杂的加密函数。利用这些操作的非线性和难以逆向的特性，我们可以进一步增强哈希函数的安全性和随机性。通过结合多个不均匀的Pathosis哈希值，我们可以通过ARX操作来生成一个更加安全、随机且均匀的最终哈希值。

ARX操作的具体实施

考虑到Pathosis哈希函数可能生成的不均匀哈希值，我们可以选择生成四个这样的哈希值，然后通过以下ARX操作将它们混合：

1. **模加（Modular Addition）**：将两个哈希值相加，并确保结果在一个固定的数值范围内，例如使用32位整数范围。
2. **旋转（Rotation）**：将哈希值的比特位进行循环位移，这可以帮助分散比特位上的信息，提高哈希函数的复杂性。
3. **异或（XOR）**：将两个哈希值进行异或操作，这是一种常见的用于创建新的随机性和非预测性的哈希值的方法。

哈希值的混合过程

我们首先生成四个Pathosis哈希值 $hash_0, hash_1, hash_2, hash_3$，然后将它们通过ARX操作混合，具体步骤如下：

1. **第一步**：$result1 = hash_0 + hash_1 \mod 2^{64}$
2. **第二步**：$result2 = hash_2 \oplus hash_3$
3. **第三步**：$result3 = result1 \oplus (result1 \lll r) \oplus result2$，其中 r 是旋转的位数，根据需要调整。

c++ code (stdandard 2020 year):

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

#include <cmath>

#include <string>
#include <vector>
#include <array>
#include <algorithm>

#include <limits>
#include <numeric>

#include <random>
#include <complex>
#include <bitset>

#include <bit>

//https://zh.wikipedia.org/wiki/%E6%B7%B7%E6%B2%8C%E7%90%86%E8%AE%BA
//https://en.wikipedia.org/wiki/Chaos_theory
namespace ChaoticTheory
{
	//模拟双段摆锤物理系统，根据二进制密钥生成伪随机实数
	//Simulate a two-segment pendulum physical system to generate pseudo-random real numbers based on a binary key
	//Reference:
	//https://zh.wikipedia.org/wiki/%E5%8F%8C%E6%91%86
	//https://en.wikipedia.org/wiki/Double_pendulum
	//https://www.myphysicslab.com/pendulum/double-pendulum-en.html
	//https://www.researchgate.net/publication/345243089_A_Pseudo-Random_Number_Generator_Using_Double_Pendulum
	//https://github.com/robinsandhu/DoublePendulumPRNG/blob/master/prng.cpp
	class SimulateDoublePendulum
	{

	private:

		std::array<long double, 2> BackupTensions{};
		std::array<long double, 2> BackupVelocitys{};
		std::array<long double, 10> SystemData{};

		static constexpr long double gravity_coefficient = 9.8;
		static constexpr long double hight = 0.002;

		void run_system(bool is_initialize_mode, std::uint64_t time)
		{
			const long double& length1 = this->SystemData[0];
			const long double& length2 = this->SystemData[1];
			const long double& mass1 = this->SystemData[2];
			const long double& mass2 = this->SystemData[3];
			long double& tension1 = this->SystemData[4];
			long double& tension2 = this->SystemData[5];

			long double& velocity1 = this->SystemData[8];
			long double& velocity2 = this->SystemData[9];

			for (std::uint64_t counter = 0; counter < time; ++counter)
			{
				long double denominator = 2 * mass1 + mass2 - mass2 * ::cos(2 * tension1 - 2 * tension2);

				long double alpha1 = -1 * gravity_coefficient * (2 * mass1 + mass2) * ::sin(tension1)
					- mass2 * gravity_coefficient * ::sin(tension1 - 2 * tension2)
					- 2 * ::sin(tension1 - tension2) * mass2
					* (velocity2 * velocity2 * length2 + velocity1 * velocity1 * length1 * ::cos(tension1 - tension2))

				alpha1 /= length1 * denominator;

				long double alpha2 = 2 * ::sin(tension1 - tension2)
```

```
                        * (velocity1 * velocity1 * length1 * (mass1 + mass2) + gravity_coefficient * (mass1 + mass2) * ::c

                alpha2 /= length2 * denominator;

                velocity1 += hight * alpha1;
                velocity2 += hight * alpha2;
                tension1 += hight * velocity1;
                tension2 += hight * velocity2;
        }

        if (is_initialize_mode)
        {
                this->BackupTensions[0] = tension1;
                this->BackupTensions[1] = tension2;

                this->BackupVelocitys[0] = velocity1;
                this->BackupVelocitys[1] = velocity2;
        }
}

void initialize(std::vector<std::int8_t>& binary_key_sequence)
{
        if (binary_key_sequence.empty())
                return;

        const std::size_t binary_key_sequence_size = binary_key_sequence.size();
        std::vector<std::vector<std::int8_t>> binary_key_sequence_2d(4, std::vector<std::int8_t>());
        for (std::size_t index = 0; index < binary_key_sequence_size / 4; index++)
        {
                binary_key_sequence_2d[0].push_back(binary_key_sequence[index]);
                binary_key_sequence_2d[1].push_back(binary_key_sequence[binary_key_sequence_size / 4 + index]);
                binary_key_sequence_2d[2].push_back(binary_key_sequence[binary_key_sequence_size / 2 + index]);
                binary_key_sequence_2d[3].push_back(binary_key_sequence[binary_key_sequence_size * 3 / 4 + index]);
        }

        std::vector<std::vector<std::int8_t>> binary_key_sequence_2d_param(7, std::vector<std::int8_t>());
        std::int32_t key_outer_round_count = 0;
        std::int32_t key_inner_round_count = 0;
        while (key_outer_round_count < 64)
        {
                while (key_inner_round_count < binary_key_sequence_size / 4)
                {
                        binary_key_sequence_2d_param[0].push_back(binary_key_sequence_2d[0][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[1].push_back(binary_key_sequence_2d[0][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[2].push_back(binary_key_sequence_2d[0][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[3].push_back(binary_key_sequence_2d[1][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[4].push_back(binary_key_sequence_2d[1][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[5].push_back(binary_key_sequence_2d[2][key_inner_round_count] ^ binar
                        binary_key_sequence_2d_param[6].push_back(binary_key_sequence_2d[0][key_inner_round_count]);

                        ++key_inner_round_count;
                        ++key_outer_round_count;
                        if (key_outer_round_count >= 64)
                        {
                                break;
                        }
                }
                key_inner_round_count = 0;
        }
        key_outer_round_count = 0;

        long double& radius = this->SystemData[6];
        long double& current_binary_key_sequence_size = this->SystemData[7];

        for (std::int32_t i = 0; i < 64; i++)
        {
                for (std::int32_t j = 0; j < 6; j++)
```

```
                {
                        if (binary_key_sequence_2d_param[j][i] == 1)
                                this->SystemData[j] += 1 * ::powl(2.0, 0 - i);
                }
                if (binary_key_sequence_2d_param[6][i] == 1)
                        radius += 1 * ::powl(2.0, 4 - i);
        }

        current_binary_key_sequence_size = static_cast<long double>(binary_key_sequence_size);

        //This is initialize mode
        this->run_system(true, static_cast<std::uint64_t>(::round(radius * current_binary_key_sequence_size)));
    }

    long double generate()
    {
        //This is generate mode
        this->run_system(false, 1);

        long double temporary_floating_a = 0.0;
        long double temporary_floating_b = 0.0;

        std::int64_t left_number = 0, right_number = 0;

        temporary_floating_a = this->SystemData[0] * ::sin(this->SystemData[4]) + this->SystemData[1] * ::sin(this->System
        temporary_floating_b = -(this->SystemData[0]) * ::sin(this->SystemData[4]) - this->SystemData[1] * ::sin(this->Sys

        temporary_floating_a = fmod(temporary_floating_a * 1000.0, 1.0) * 4294967296;
        temporary_floating_b = fmod(temporary_floating_b * 1000.0, 1.0) * 4294967296;

        if (std::isnan(temporary_floating_a) || std::isnan(temporary_floating_b))
        {
                return 0.0;
        }

        if (std::isinf(temporary_floating_a) || std::isinf(temporary_floating_b))
        {
                return 0.0;
        }

        //std::cout << "temporary_floating_a: " << temporary_floating_a << std::endl;
        //std::cout << "temporary_floating_b: " << temporary_floating_b << std::endl;

        //This is generate mode
        this->run_system(false, 1);
        if (temporary_floating_a * 2.0 + temporary_floating_b >= 0.0)
        {
                return temporary_floating_a;
        }
        else
        {
                return temporary_floating_b;
        }
    }

public:

    using result_type = long double;

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

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

```
};

result_type operator()()
{
        return this->generate();
}

void reset()
{
        this->SystemData[4] = this->BackupTensions[0];
        this->SystemData[5] = this->BackupTensions[1];
        this->SystemData[8] = this->BackupVelocitys[0];
        this->SystemData[9] = this->BackupVelocitys[1];
}

void seed_with_binary_string(std::string binary_key_sequence_string)
{
        std::vector<int8_t> binary_key_sequence;
        std::string_view view_only_string(binary_key_sequence_string);
        const char binary_zero_string = '0';
        const char binary_one_string = '1';
        for (const char& data : view_only_string)
        {
                if (data != binary_zero_string && data != binary_one_string)
                        continue;

                binary_key_sequence.push_back(data == binary_zero_string ? 0 : 1);
        }

        if (binary_key_sequence.empty())
                return;
        else
                this->initialize(binary_key_sequence);
}

void seed(std::int32_t seed_value)
{
        this->seed_with_binary_string(std::bitset<32>(seed_value).to_string());
}

void seed(std::uint32_t seed_value)
{
        this->seed_with_binary_string(std::bitset<32>(seed_value).to_string());
}

void seed(std::int64_t seed_value)
{
        this->seed_with_binary_string(std::bitset<64>(seed_value).to_string());
}

void seed(std::uint64_t seed_value)
{
        this->seed_with_binary_string(std::bitset<64>(seed_value).to_string());
}

void seed(const std::string& seed_value)
{
        this->seed_with_binary_string(seed_value);
}

SimulateDoublePendulum()
{
        this->seed(static_cast<uint64_t>(1));
}

explicit SimulateDoublePendulum(auto number)
{
```

```
                this->seed(number);
        }

        ~SimulateDoublePendulum()
        {
                this->BackupVelocitys.fill(0.0);
                this->BackupTensions.fill(0.0);
                this->SystemData.fill(0.0);
        }
    };
}

class Dirichlet
{

private:
    long double real_number = 0.0;
    uint64_t limit_count = 0;
    const long double pi = 3.141592653589793;
public:
    // 构造函数，初始化随机数生成器
    Dirichlet() = default;

    using result_type = long double;

    static constexpr result_type min()
    {
        return -2.0;
    }

    static constexpr result_type max()
    {
        return 2.0;
    };

    void reset(long double real_number, uint64_t limit_count)
    {
        this->real_number = real_number;
        this->limit_count = limit_count;
    }

    // 计算Dirichlet函数
    long double operator()()
    {
        if (real_number == 0.0)
        {
            return real_number;
        }

        //y = D(x) =limit[k -> ∞]limit[j -> ∞] cos(k! πx)^{2j}
        for (uint64_t k = 1; k <= limit_count; ++k)
        {
            std::complex<long double> z(0, k * std::tgamma(k + 1) * pi * real_number);  // z(real: 0,imag: j!πx)
            std::complex<long double> e_to_z = std::exp(z);  // e^z(real: 0,imag: j!πx)
            long double result = std::pow(e_to_z.real(), 2 * limit_count);

            if (std::isnan(result) || std::isinf(result))
            {
                // 处理数值不稳定的情况
                return 0.0;
            }
            if (std::isinf(result))
            {
                return 1.0;
            }
            if (std::abs(result) > std::numeric_limits<long double>::epsilon())
            {
```

```
                        return result;
                }
        }
        return 0.0; // 默认返回值
    }
};

uint64_t pathosis_hash(uint64_t seed, uint64_t limit_count)
{
    // Seed the random number generator
    std::mt19937_64 prng(1);
    prng.seed(seed);
    // Generate real numbers between lowest and highest double
    ChaoticTheory::SimulateDoublePendulum SDP(static_cast<uint64_t>(prng()));

    //Here, to facilitate the use of c++'s pseudo-random number class, the Dirichlet class function is also encapsulated in the form c
    //这里为了方便使用c++的伪随机数类，所以把Dirichlet函数也封装成了类的形式。
    Dirichlet dirichlet;

    uint64_t result = 0;

    for (int i = 0; i < 128; ++i)
    {
#if 1
        long double real_number = (std::sin(SDP()) + 1.0 / 2.0) * (1024.0e64 - 1024.0e-64) + 1024.0e-64;
#else
        long double real_number = SDP() * (1024.0e64 - 1024.0e-64) + 1024.0e-64;
#endif
        //std::cout << "real_number: "<< real_number << std::endl;

        dirichlet.reset(real_number, limit_count);
        long double dirichlet_value = dirichlet();

        //Uniform01
        long double uniform01_value = 0.0;
        if (dirichlet_value >= 0.0 && dirichlet_value <= 1.0)
        {
            if ((dirichlet_value - 0.0 > std::numeric_limits<long double>::epsilon()) || (dirichlet_value - 1.0 > std::numeric
            {
                uniform01_value = dirichlet_value;
            }
        }
        else
        {
            if (dirichlet_value < 0.0)
            {
                uniform01_value = fmod(-dirichlet_value * 4294967296, 1.0);
            }
            else if (dirichlet_value > 1.0)
            {
                uniform01_value = fmod(dirichlet_value * 4294967296, 1.0);
            }
        }

        //if(uniform01_value < 0.0 || uniform01_value > 1.0)
            //std::cerr << "uniform01_value range is error! " << std::endl;
        //std::cout << "uniform01_value: " << uniform01_value << std::endl;

        result |= (uint64_t)(uniform01_value >= 0.5) << i;
    }

    return result;
}

inline void print_pathosis_hash()
{
    //std::cout << std::setiosflags(std::ios::fixed);
```

```
        //std::cout << std::setprecision(24);
        std::mt19937_64 prng(2);
        uint64_t rounds = 256, sub_rounds = 102400;

        uint64_t hash_number_a = 0;
        uint64_t hash_number_b = 0;
        std::string binary_string_a = "";
        std::string binary_string_b = "";
        std::string binary_string_a_xor_b = "";

        for (uint64_t round = 0; round < rounds; round++)
        {
                //std::cout << pathosis_hash(prng(), sub_rounds) << std::endl;
                hash_number_a = pathosis_hash(prng(), sub_rounds);
                hash_number_b = pathosis_hash(prng(), sub_rounds);

                std::cout << "pathosis_hash a: " << hash_number_a << std::endl;
                std::cout << "pathosis_hash b: " << hash_number_b << std::endl;

                binary_string_a = std::bitset<64>(hash_number_a).to_string();
                std::cout << "binary_string a: " << binary_string_a << std::endl;
                binary_string_b = std::bitset<64>(hash_number_b).to_string();
                std::cout << "binary_string b: " << binary_string_b << std::endl;
                binary_string_a_xor_b = (std::bitset<64>(hash_number_a) ^ std::bitset<64>(hash_number_b)).to_string();
                std::cout << "binary_string a xor b: " << binary_string_a_xor_b << std::endl;
                //sub_rounds++;
        }

}

inline void hash_chain_with_pathosis_hash()
{
        //std::cout << std::setiosflags(std::ios::fixed);
        //std::cout << std::setprecision(24);
        std::mt19937_64 prng(2);
        uint64_t rounds = 256, sub_rounds = 102400;

        uint64_t hash_number_a = 0;
        uint64_t hash_number_b = 0;
        std::string binary_string_a = "";
        std::string binary_string_b = "";
        std::string binary_string_a_xor_b = "";

        for (uint64_t round = 0; round < rounds; round++)
        {
                if (round == 0)
                {
                        hash_number_a = pathosis_hash(prng(), sub_rounds);
                        hash_number_b = pathosis_hash(prng(), sub_rounds);
                }
                else
                {
                        hash_number_a = pathosis_hash(hash_number_b, sub_rounds);
                        hash_number_b = pathosis_hash(hash_number_a, sub_rounds);
                }

                std::cout << "pathosis_hash a: " << hash_number_a << std::endl;
                std::cout << "pathosis_hash b: " << hash_number_b << std::endl;

                binary_string_a = std::bitset<64>(hash_number_a).to_string();
                std::cout << "binary_string a: " << binary_string_a << std::endl;
                binary_string_b = std::bitset<64>(hash_number_b).to_string();
                std::cout << "binary_string b: " << binary_string_b << std::endl;
                binary_string_a_xor_b = (std::bitset<64>(hash_number_a) ^ std::bitset<64>(hash_number_b)).to_string();
                std::cout << "binary_string a xor b: " << binary_string_a_xor_b << std::endl;
        }
}
```

```
inline void arx_with_pathosis_hash()
{
	//std::cout << std::setiosflags(std::ios::fixed);
	//std::cout << std::setprecision(24);
	std::mt19937_64 prng(2);
	uint64_t rounds = 256, sub_rounds = 102400;

	uint64_t hash_number_a = pathosis_hash(prng(), sub_rounds);
	uint64_t hash_number_b = pathosis_hash(prng(), sub_rounds);
	uint64_t hash_number_c = pathosis_hash(prng(), sub_rounds);
	uint64_t hash_number_d = pathosis_hash(prng(), sub_rounds);

	std::cout << "pathosis_hash a: " << hash_number_a << std::endl;
	std::cout << "pathosis_hash b: " << hash_number_b << std::endl;
	std::cout << "pathosis_hash c: " << hash_number_c << std::endl;
	std::cout << "pathosis_hash d: " << hash_number_d << std::endl;

	std::string binary_string_a = "";
	std::string binary_string_b = "";
	std::string binary_string_a_xor_b = "";

	for (uint64_t round = 0; round < rounds; round++)
	{
		//Add-(Bits)Rotate-Xor Example:
		uint64_t OperatorA = hash_number_a + hash_number_b;
		uint64_t OperatorX = hash_number_c ^ hash_number_d;
		uint64_t OperatorR = OperatorA ^ std::rotl(OperatorA, 47) ^ OperatorX;

		hash_number_a = OperatorR;
		hash_number_b = OperatorX;
		hash_number_c = OperatorA;
		hash_number_d = std::rotl((OperatorR + OperatorX), 64 - 47) ^ OperatorA;

		std::cout << "arx_with_pathosis_hash a: " << hash_number_a << std::endl;
		std::cout << "arx_with_pathosis_hash b: " << hash_number_b << std::endl;
		std::cout << "arx_with_pathosis_hash c: " << hash_number_c << std::endl;
		std::cout << "arx_with_pathosis_hash d: " << hash_number_d << std::endl;

		binary_string_a = std::bitset<64>(hash_number_a).to_string();
		std::cout << "binary_string a: " << binary_string_a << std::endl;
		binary_string_b = std::bitset<64>(hash_number_b).to_string();
		std::cout << "binary_string b: " << binary_string_b << std::endl;
		binary_string_a = std::bitset<64>(hash_number_c).to_string();
		std::cout << "binary_string c: " << binary_string_a << std::endl;
		binary_string_b = std::bitset<64>(hash_number_d).to_string();
		std::cout << "binary_string d: " << binary_string_b << std::endl;

		hash_number_a = pathosis_hash(hash_number_a, sub_rounds);
		hash_number_b = pathosis_hash(hash_number_b, sub_rounds);
		hash_number_c = pathosis_hash(hash_number_c, sub_rounds);
		hash_number_d = pathosis_hash(hash_number_d, sub_rounds);
	}
}

int main()
{
	arx_with_pathosis_hash();

	return 0;
}
```

本文介绍了一种基于混沌理论和狄利克雷函数的哈希函数设计，称为Pathosis哈希函数。
以下是对文章中每个部分的核心数学目标和方法的总结：

第一部分：从有限域元素到实数

数学目的：将来自有限域 $\mathbb{F}_2^{64}$ 的64位伪随机整数 x 转换为均匀分布的实数 x'。
数学表述：
利用双摆系统的模拟输出，通过正弦函数和线性变换将 x 映射到一个均匀分布的实数：

$$x' = pendulum\_min + (pendulum\_max - pendulum\_min) \times \frac{\sin(\text{SimulateDoublePendulum.operator}()(x)) + 1.0}{2.0}$$

这里 x' 代表映射后的实数，而 x 是原始的伪随机数。

第二部分：从实数 x' 到非均匀映射 $y \in \mathbb{R}\,[0,1]$

数学目的：利用改良的狄利克雷函数对 x' 进行处理，生成具有非连续性特点的中间值 y。
数学表述：
定义函数 $y = D(x')$ 通过使用复数的三角函数和极限来实现复杂的非连续性：

$$y = \lim_{j \to n} \lim_{k \to n} \cos\left(j! \pi x'\right)^{2k}$$

此函数设计通过增加角频率 $j!\pi$ 和平方项 $2k$ 来增强输出的波动性和敏感性，这增加了哈希过程的安全性。

第三部分：从非均匀映射 y 到均匀分布 y' 和哈希空间映射 H_D

数学目的：首先将 y 调整为一个真正均匀分布的 y'，然后将 y' 映射到哈希空间 H 的离散元素上。
数学表述：

1. **均匀分布映射**：
   通过累积分布函数的逆来将 y 映射到 y'：

   $$y' = F_Y^{-1}(y)$$

   其中 F_Y^{-1} 是 y 的累积分布函数的逆函数，用于确保 y' 在 [0,1] 区间内均匀分布。
   在实际实现过程中，因为第一部分产生的随机浮点数是比较规范化的，没有奇怪的浮点值。我们可以用浮点数模运算 `fmod` 直接来解决这个问题。
2. **哈希空间映射**：
   利用 y' 计算离散的哈希值 $H_D(y')$：

   $$hash = H_D(y')_i = [y' \geq 0.5]$$

   在此步骤中，每个 y' 的比特 i 根据 y' 是否大于或等于0.5来决定其值，0或1。最终的哈希值 $H_D(y')$ 由64位的二进制序列构成，每一位都是根据 y' 的相应阈值独立计算得出。