Technical Details of The Algorithm Little OaldresPuzzle_Cryptic

Twilight-Dream

February 16, 2024

1 Abstract

This paper introduces a symmetric sequence cryptographic algorithm, ”Little OaldresPuzzle_Cryptic”. The algorithm combines the speed of symmetric cryptography with an extra layer of security through sequence-based encryption. Central to its design is a cryptographically secure pseudo-random number generator, ”XorConstantRotation” (XCR), which greatly increases the complexity of the encryption process. The paper explores the algorithm’s technical aspects, illustrating its capability to balance speed and security in high-speed data transmission scenarios.

Contents

2 Introduction

This paper presents "Little OaldresPuzzle_Cryptic", a symmetric sequence cryptographic algorithm tailored for the demands of our digital age. As data generation and consumption rates increase, so do threats to data integrity and security. This scenario requires cryptographic algorithms capable of ensuring secure data exchange and high-speed performance.

Our algorithm uses the same cryptographic key for both encryption and decryption, offering significant speed advantages over asymmetric alternatives. The sequence cryptographic approach adds complexity to the encryption and decryption process, making it harder for unauthorized entities to interpret encrypted data.

The algorithm combines speed and security, leveraging a cryptographically secure pseudo-random number generator (CSPRNG), "XorConstantRotation" (XCR), which generates highly unpredictable number sequences.

The remainder of this paper will examine the algorithm's mechanics, demonstrating how it employs mathematical constants, bitwise operations, and sequence-based cryptography to ensure speed and security in data transmission.

3 XCR/Little OaldresPuzzle_Cryptic Stream Cipher Design Background:

3.1 What's ARX structure and Salsa20, Chacha20 Algorithm ?

ARX, or Addition/Bit-Rotation/Exclusive-OR, is a class of symmetric key algorithms constructed using the following simple operations: modulo addition, bit rotation, and Exclusive-OR. Unlike S-box based designs, where the only nonlinear element is the substitution box (S-box), ARX designs rely on nonlinear hybrid functions such as addition and rotation. [Ranea et al., 2022] [Liu et al., 2021] These functions are easy to implement in both software and hardware and have good diffusivity and resistance to differential and linear analysis. [Fei, 2012] [Aumasson et al., 2007] There are some ECRYPT PPTs that mention this design structure in summary. ARX-based Cryptography

Salsa20 is a stream cipher algorithm proposed by Daniel J. Bernstein in 2005, which utilizes the ARX structure.Salsa20/8 and Salsa20/12 are two variants of Salsa20 that run 8 and 12 rounds of encryption respectively. [Bernstein, 2005] [Bernstein, 2008] These algorithms were evaluated in the eSTREAM project and accepted as finalists in 2008. [Tsunoo et al., 2007] Salsa20 was designed with a focus on simplicity and efficiency, and it is favored in the cryptography community for its speed and security. [Maitra et al., 2015] [Ghafoori and Miyaji, 2022]

ChaCha20 [Bernstein et al., 2008a] [Bernstein et al., 2008b] is an ARX-based high-speed stream cipher proposed by Daniel J. Bernstein in 2008 as an improved version of Salsa20 .ChaCha20 uses a 512-bit permutation function to convert a 512-bit input vector into a 512-bit output vector, and then The output vector is added to the input vector to obtain a 512-bit keystream block. The input vector consists of four parts: a constant, a key, a counter, and a random number.The security of ChaCha20 is based on the complexity and irreversibility of the substitution function, as well as the randomness and uniqueness of the input vector. ChaCha20 has been used in a wide variety of applications, such as TLS v1.3, SSH, IPsec, WireGuard, etc. [Serrano et al., 2022] [Cai et al., 2022].

3.2 About the XCR Structure of Our Lightweight Symmetric Encryption Technology

3.2.1 Designing XCR from the ARX Structure: Insights from NIST and Lightweight Cryptography

The National Institute of Standards and Technology (NIST) has acknowledged the urgent need for lightweight encryption algorithms to address the security challenges posed by the proliferation of resource-constrained devices, particularly in the context of the Internet of Things (IoT). NIST-LCS Website These devices, such as sensors and RFID tags, demand encryption algorithms that are both efficient and secure, a balance that traditional cryptographic methods often fail to achieve due to their high computational overhead.

Our approach to designing the XCR structure is referenced in the insights provided by the paper "State of the Art in Lightweight Symmetric Cryptography" by Biryukov and Perrin [Biryukov and Perrin, 2017]. This seminal work outlines the design constraints and trends in lightweight symmetric cryptography, emphasizing the importance of algorithms tailored to specific hardware and use cases. It highlights the critical trade-offs between performance, security, and resource consumption, which are central to the development of lightweight encryption algorithms.

The paper sets forth criteria for lightweight cryptographic algorithms, particularly in resource-constrained environments. It advocates for a small block size, ideally 64-bit or less, and a small key size of at least 80 bits to balance security and efficiency. The round function should be simple, relying on straightforward operations that are easy to implement on low-power devices. A straightforward key scheduling mechanism is also essential to avoid vulnerabilities and complexity.

The potential risks of poorly implemented lightweight cryptography are underscored by the study of "Speck-R: an ultra-lightweight encryption scheme for the Internet of Things" ([Sleem and Couturier, 2021]). This paper and the references it cites illustrate the disastrous consequences of not using lightweight encryption, which can lead to security breaches and even denial-of-service attacks on small devices. This paper serves as a cautionary tale, emphasizing the need for careful design and implementation of such schemes.

Our "Little OaldresPuzzle_Cryptic" algorithm is designed with these considerations in mind, adhering to the standards set by the state-of-the-art in lightweight cryptography. It aims to provide a robust and efficient solution for secure data transmission in IoT and other resource-constrained environments.

By leveraging the ARX structure, known for its simplicity and efficiency, we have crafted the XCR to generate a sequence of pseudo-random numbers that are both unpredictable and computationally intensive. This structure allows for a consistent architecture adaptable across various device types, enabling faster encryption and decryption processes.

4 XCR Structure Overview

The XCR structure is a ARX novel design that we propose to enhance the randomness and security of the CSPRNG. The XCR structure consists of three main operations: Exclusive-OR, constant addition, and bitwise rotation. (We chose the operations because they are simple and efficient bitwise and arithmetic operations that improve randomness and security without increasing complexity.The Exclusive-OR operation enables state mixing or data mixing of inputs and outputs, constant addition introduces randomness of irrational numbers, and bitwise rotation allows for spreading and scrambling of bits.) The XOR operation combines the CSPRNG state with a random data, which can be derived from the key, the nonce, or the counter. The constant addition operation adds a mathematical constant to the XOR output, which can be chosen from well-known irrational numbers, such as π or e (We choose round constants from irrational numbers because they have infinitely acyclic fractional parts (or infinitely decimal part) and can provide a high-quality source of random numbers.) The bitwise rotation operation shifts the bits of the constant addition output by a certain amount, which can be determined by the key or the counter, but we chose the constant. The XCR structure produces a highly unpredictable and non-repeating output, which can be used as a keystream for encryption or a tag for authentication.

The XCR structure can be formally defined as follows:

- Define x, y and $state$ be 64-bit unsigned integers, where x, y represent random input/output states of the XCR structure. Initially, $x = y = 0$ and $state$ is the internal state of the CSPRNG.

- Define $number\_once$ be a 64-bit unsigned integer, representing the round number of the XCR structure. Initially, $number\_once = 0$. The value of $number\_once$ can be derived from the key, the nonce, the counter.

- Define $\oplus$, $\boxplus_{64}$, and $\lll$, $\ggg$ denote the bitwise XOR, addition modulo 2^{64}, and left/right rotation operations, respectively.

- Define $Constants$ be an array of 64-bit unsigned integers, representing the round constants of the XCR structure. The elements of $Constants$ can be chosen from well-known irrational numbers.

- Define I and O be a 64-bit unsigned integer, representing the non-random/random input and random output of the XCR structure. The value of I can be derived from the plain-text, the cipher-text, the random data. The value of O is obtained by applying the XCR structure to I, which is defined as follows:

$$O = XCR(I, number\_once) = I \oplus PRF(x, y, state, number\_once) = I \oplus y$$

  which consists of three main operations: XOR, constant addition, and bitwise rotation. The XCR structure produces a highly unpredictable and non-repeating output, which can be used as a keystream for encryption or a tag for authentication.

- Define $r1$, $r2$, $r3$ as the rotation amounts of the bitwise operations, which indicate the number of bits moved in each rotation. To ensure the maximum randomness of the XCR structure, we require that the bit lengths of $r1$, $r2$, $r3$ must be mutually (prime/relatively prime). This means that $r1$, $r2$, $r3$ have no common factors, except for 1. This can avoid the repetition of patterns caused by the rotation operations, and fully utilize the uniform distribution randomness of mod primes. For example, if the bit lengths of $r1$, $r2$, $r3$ are all even, then the rotation operations will only affect the even bits, while the odd bits remain unchanged, which reduces the randomness of the XCR structure. Therefore, we choose the bit lengths of $r1$, $r2$, $r3$ such that they have no common divisors, except for 1.

- The XCR structure can be expressed as a function $XCR(number\_once)$, which takes the number once $number\_once$ as input and returns the output y as follows:

$$\begin{aligned} y &= (x \oplus (state \lll r1)) \oplus (state \lll r2) \\ x &= \begin{cases} Constants[number\_once \bmod |Constants|] & \text{if } x = 0 \\ x \boxplus_{64} ((x \lll r3) \oplus Constants[number\_once \bmod |Constants|]) \oplus number\_once & \text{otherwise} \end{cases} \\ state &= state \boxplus_{64} (x \oplus y) \\ O &= I \oplus y \end{aligned}$$

  Used here only, where $|Constants|$ denotes the size of the array $Constants$.

The XCR's design exemplifies the adaptability of ARX-based algorithms, contributing to the field of lightweight cryptography by offering a solution that balances speed and security in high-speed data transmission scenarios.

5 Little OaldresPuzzle_Cryptic Algorithm Description

Building upon the foundation laid by the ARX structure and the innovative XCR design, the "Little OaldresPuzzle_Cryptic" algorithm is a sophisticated cryptographic tool designed for high-speed data transmission with enhanced security. The precomputed constants used in each round of this algorithm are composed of complex mathematical principles and operations, such as bitwise operations, hexadecimal representation, irrational and transcendental numbers, sequence generation, pseudo-random number generation, etc, a combination of the above methods generate.

First, We need to specifically outline an algorithm for generator — computing the hexadecimal representation of the XCR round constants, which serves as a cornerstone for our cryptographic functions. This algorithm leverages the inherent unpredictability of several mathematical constants, including Euler's number, Pi, the Golden ratio, the square roots of 2 and 3(the cube roots of 2 and 3), Euler-Mascheroni constant, Feigenbaum constant, and the Plastic number. These constants, known for their irrational and transcendental characteristics, inject a fundamental element of randomness, thus augmenting the cryptographic strength of our algorithm. This binary sequence of constants, generated by a nonlinear Boolean function, is further segmented into 64-bit groups and returned as a hexadecimal string.

Second, in the previous introduction we illustrate our cryptographically secure pseudo-random number generator (CSPRNG), called "XorConstantRotation" or "XCR structural design". The XCR algorithm relies heavily on the constant data for each round that has been precomputed by the above process.

Subsequently, The heart of the discussion is devoted to our symmetric sequence cryptographic algorithm, the "Little OaldresPuzzle_Cryptic". The algorithm, designed as a class, employs the XorConstantRotation for its operations. In the coming sections, the paper lays out pseudo-codes for these algorithms, bringing their mechanics to light.

6 Flow of the algorithms

Algorithm 1 Generator - Computing XCR Round Constant Hexadecimal Representation

```
1:  function GenerateRoundConstant                        ▷ This function generates the round constant
2:      $e \leftarrow 2.71828182845904523536028747135266249775724709369995957496696762772407663035354759457138217852516642 7$ ▷ Euler's number
3:      $\pi \leftarrow 3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 8$ ▷ Pi
4:      $\phi \leftarrow 1.61803398874989484820458683436563811772030917980576286213544862270526046281890244970720720418939113 7$ ▷ Golden ratio $\frac{1+\sqrt{5}}{2}$
5:      $\sqrt{2} \leftarrow 1.41421356237309504880168872420969807856967187537694807317667973799073247846210703885038753432764157 3$ ▷ Square root of 2
6:      $\sqrt{3} \leftarrow 1.73205080756887729352744634150587236694280525381038062805580697945193301690880003708114618675724857 6$ ▷ Square root of 3
7:      $\gamma \leftarrow 0.5772156649$                           ▷ Euler-Mascheroni constant
8:      $\delta \leftarrow 4.6692016091$                           ▷ Feigenbaum constant
9:      $\rho \leftarrow 1.3247179572$                             ▷ Plastic number $\sqrt[3]{\frac{9+\sqrt{69}}{18}} + \sqrt[3]{\frac{9-\sqrt{69}}{18}}$

10:     $x \leftarrow 1$
11:     $binary\_string \leftarrow$ ""
12:     for $index \leftarrow 0$ to 140 do
13:         $result \leftarrow (e^x - \cos(\pi x)) \times (\phi x^2 - \phi x - 1) \times (\sqrt[2]{2}x - \lfloor \sqrt[2]{2}x \rfloor) \times (\sqrt[2]{3}x - \lfloor \sqrt[2]{3}x \rfloor) \times \ln(1+x) \times (x\delta - \lfloor x\delta \rfloor) \times (x\rho - \lfloor x\rho \rfloor)$
                ▷ Original plan: $(e^x - \cos(\pi x)) \times (\phi x^2 - \phi x - 1) \times (\sqrt[2]{x} - \lfloor \sqrt[2]{x} \rfloor) \times (\sqrt[3]{x} - \lfloor \sqrt[3]{x} \rfloor) \times \ln(1+x) \times (x\delta - \lfloor x\delta \rfloor) \times (x\rho - \lfloor x\rho \rfloor)$

14:         $fractional\_part \leftarrow result - \lfloor result \rfloor$    ▷ Isolate the fractional part
15:         $binary\_fractional\_part \leftarrow$ Binary of $(fractional\_part \times 2^{128})$
16:         $hexadecimal\_fractional\_part \leftarrow$ Hexadecimal of $(fractional\_part \times 2^{128})$
17:         $integer\_part \leftarrow \lfloor result \rfloor$
18:         $x \leftarrow x + 1$
19:         $binary\_string \leftarrow binary\_string.$Append$(binary\_fractional\_part)$
20:     end for
21:     $integer\_value \leftarrow$ Integer of $binary\_string$          ▷ print result
22:     $hexadecimal\_string \leftarrow$ Hexadecimal of $integer\_value$
23:     return $hexadecimal\_string$        ▷ Return the hexadecimal string of the round constant
24: end function
```

Algorithm 2 Cryptographically Secure Pseudo-Random Number Generator - XorConstantRotation (XCR)

```
function XCR INITIALIZE($seed$)
    $seed \in \mathbb{F}_2^{64}$
    $state \in \mathbb{F}_2^{64}$
    $x, y \in \mathbb{F}_2^{64}$
    $ROUND\_CONSTANT_i \in \mathbb{F}_2^{64}$
    $x, y = 0$
    $state = seed$                                          ▷ Initial state is the seed
end function

function XCR GENERATION($number\_once$)
    $y \leftarrow (x \oplus (state \lll 32)) \oplus (state \lll 19))$
    if $x == 0$ then
        $x \leftarrow$ ROUND_CONSTANT[$round$ (mod size(ROUND_CONSTANT)])
    else
        $x \leftarrow x \boxplus_{64} ((x \lll 7) \oplus$ ROUND_CONSTANT[$round$ (mod size(ROUND_CONSTANT)])) $\oplus$ $number\_once$
    end if
    $state \leftarrow (x \oplus y) \boxplus_{64} state$
    return $y$
end function
```

Algorithm 3 LittleOaldresPuzzle_Cryptic Class Implementation

```
$seed \leftarrow$ initial seed value                         ▷ Set the initial seed for CSPRNG
$csprng \leftarrow$ XORCONSTANTROTATION($number\_once$)      ▷ The XorConstantRotation CSPRNG Instance
$rounds \leftarrow 8, 16, 32, 64 \ldots$                      ▷ Set the number of rounds
function KEYSTATE($subkey, choise\_function, bit\_rotation\_amount\_a, bit\_rotation\_amount\_b$)
    return KeyState with the following attributes:
            subkey = $subkey$
            choise_function = $choise\_function$
            bit_rotation_amount_a = $bit\_rotation\_amount\_a$
            bit_rotation_amount_b = $bit\_rotation\_amount\_b$
end function
function ENCRYPTION($data, key, number\_once$)
    $result \leftarrow data$                                  ▷ Initialize result with the data
    for $round \leftarrow 0$ to $rounds - 1$ do
        $key\_state \leftarrow$ KeyState$()_{round}$              ▷ Initialize KeyState
        $key\_state.subkey \leftarrow key \oplus$ CSPRNG($number\_once \oplus round$)   ▷ Generate subkey using PRNG
        $key\_state.choise\_function \leftarrow$ CSPRNG($key\_state.subkey \oplus (key \gg 1)$)   ▷ Generate choice function
        $key\_state.bit\_rotation\_amount\_a \leftarrow$ CSPRNG($key\_state.subkey \oplus key\_state.choise\_function$)   ▷ Calculate bit rotation amount
        $key\_state.bit\_rotation\_amount\_b \leftarrow (key\_state.bit\_rotation\_amount\_a \gg 6) \mod 64$   ▷ Select bit position 6 to 11
        $key\_state.bit\_rotation\_amount\_a \leftarrow key\_state.bit\_rotation\_amount\_a \mod 64$   ▷ Select bit position 0 to 5
        $key\_state.choise\_function \leftarrow key\_state.choise\_function \mod 4$   ▷ Ensure choice function is in range [0, 3]
    end for
    for $round \leftarrow 0$ to $rounds - 1$ do
        $key\_state \leftarrow key\_state_{round}$                ▷ Get KeyState
        if $key\_state.choise\_function = 0$ then
            $result \leftarrow result \oplus key\_state.subkey$     ▷ XOR operation
        else if $key\_state.choise\_function = 1$ then
            $result \leftarrow \neg(result \oplus key\_state.subkey)$  ▷ Negation(Bitwise NOT) after XOR operation
        else if $key\_state.choise\_function = 2$ then
            $result \leftarrow (result \lll key\_state.bit\_rotation\_amount\_b)$  ▷ Left bitwise rotation
        else if $key\_state.choise\_function = 3$ then
            $result \leftarrow (result \ggg key\_state.bit\_rotation\_amount\_b)$  ▷ Right bitwise rotation
        end if
        $result \leftarrow result \oplus (1 \ll key\_state.bit\_rotation\_amount\_a)$
        $result \leftarrow result \boxplus_{64} ((key \ggg 3) \oplus (key\_state.subkey \ggg 11))$
    end for
    return $result$
end function
function DECRYPTION($data, key, number\_once$)
    $result \leftarrow data$  ▷ Initialize result with the data
    for $round \leftarrow 0$ to $rounds - 1$ do
        $key\_state \leftarrow \text{KeyState}()_{round}$  ▷ Initialize KeyState
        $key\_state.subkey \leftarrow key \oplus \text{CSPRNG}(number\_once \oplus round)$
        $key\_state.choise\_function \leftarrow \text{CSPRNG}(key\_state.subkey \oplus (key \gg 1))$
        $key\_state.bit\_rotation\_amount\_a \leftarrow \text{CSPRNG}(key\_state.subkey \oplus key\_state.choise\_function)$
        $key\_state.bit\_rotation\_amount\_b \leftarrow (key\_state.bit\_rotation\_amount\_a \gg 6) \mod 64$
        $key\_state.bit\_rotation\_amount\_a \leftarrow key\_state.bit\_rotation\_amount\_a \mod 64$
        $key\_state.choise\_function \leftarrow key\_state.choise\_function \mod 4$
    end for
    for $round \leftarrow rounds$ down to $1$ do
        $key\_state \leftarrow key\_state_{round-1}$  ▷ Get KeyState
        $result \leftarrow result \boxminus_{64} ((key \ggg 3) \oplus (key\_state.subkey \ggg 11))$
        $result \leftarrow result \oplus (1 \ll key\_state.bit\_rotation\_amount\_a)$
        if $key\_state.choise\_function = 0$ then
            $result \leftarrow result \oplus key\_state.subkey$  ▷ XOR operation
        else if $key\_state.choise\_function = 1$ then
            $result \leftarrow \neg result$  ▷ Negation(Bitwise NOT)
            $result \leftarrow result \oplus key\_state.subkey$  ▷ XOR operation
        else if $key\_state.choise\_function = 2$ then
            $result \leftarrow (result \ggg key\_state.bit\_rotation\_amount\_b)$  ▷ Left bitwise rotation
        else if $key\_state.choise\_function = 3$ then
            $result \leftarrow (result \lll key\_state.bit\_rotation\_amount\_b)$  ▷ Right bitwise rotation
        end if
    end for
    return $result$
end function
function RESETPRNG($seed$)
    XORCONSTANTROTATION.SEED($seed$)  ▷ Reset the PRNG with a new seed
end function
```

We believe the readers are acquainted with the three closely related algorithms aforementioned. However, they might now be grappling with several questions regarding their composition and purpose. Thus, it is essential to address these queries and provide a more profound understanding of our design intentions.

The three algorithms, specifically, The **XCR Round Constant Generator**, Cryptographically Secure Pseudo-Random Number Generator - **XorConstantRotation (XCR)**, and **LittleOaldresPuzzle_Cryptic**, are not standalone entities but integral components of our cryptographic architecture. Each algorithm serves a purpose, and they function in unison to create a secure, robust cryptographic system.

The **XCR** Round Constant Generator lays the foundation for our cryptographic functions. This generator algorithm employs a fascinating assortment of mathematical constants, each known for its irrational and transcendental nature. This rich mix instills a degree of randomness and complexity that strengthens the cryptographic robustness of our algorithm.

Next is the Cryptographically Secure Pseudo-Random Number Generator - **XorConstantRotation (XCR)**. This algorithm is the backbone of our cryptographic system, generating sequences that exhibit randomness on the surface yet are entirely predictable given the known seed. We designed the **XCR** algorithm to leverage two-bit operations: XOR and rotating-left. These operations exhibit exceptional confusion and diffusion properties, effectively scattering changes in the input evenly across the output. This feature significantly enhances the system's security.

The final piece of the puzzle is the **LittleOaldresPuzzle_Cryptic** algorithm. This algorithm is a symmetric sequential cipher algorithm that is dependent on **XCR**. It encrypts input data through a series of complex bit operations, each adding a layer of complexity and security. The key to the steps is the generation of a subkey using **XCR**, which drives the choice of operation to be performed on the data. The decryption function of **LittleOaldresPuzzle_Cryptic** mirrors the steps of the encryption function, effectively undoing all operations executed during the encryption phase.

In this design, we purposefully harnessed certain inherent characteristics of mathematics and computer science, such as non-linearity, confusion, diffusion, and pseudorandom number generation. The aim was to create a cryptographic system that offers enhanced security, unpredictability, and resistance to attacks. Together, these components provide a formidable foundation for cryptographic protection. As such, our system stands as a testament to the strength of intertwining various elements of mathematics and computer science in creating a robust, secure cryptographic system.

7 Performance Evaluation and Security Evaluation

7.1 Performance Evaluation

This document's appendices present **Tables 1 to 3**, meticulously detailing the performance and operational characteristics of lightweight cryptographic algorithms. These tables are designed to provide an exhaustive comparison of algorithm features, offering experts in cryptography a comprehensive understanding of their performance and functionality.

A comprehensive performance evaluation methodology has been established to assess the efficiency of lightweight cryptographic algorithms, with a particular focus on the ASCON algorithm. The goal is to offer a comparative analysis of encryption and decryption operations across various data sizes, incorporating insights from multiple lightweight algorithms.

Experimental Setup:

The experimental setup involved the implementation of the **ASCON** cryptographic algorithm, utilizing a fixed key (0x0123456789ABCDEF, 0xFEDCBA9876543210) and nonce (0x0000000000000000, 0x0000000000000000). The associated data was set to a constant value of 0x12345678. For the **XCR/Little_OaldresPuzzle_Cryptic** algorithm, a 64-bit nonce was employed in counter mode, with no associated data and the same key. As for the Chacha20 algorithm, a predetermined key (0x00000000,0x00000000,0x00000000,0x00000000,0x01234567,0x89ABCDEF,0xFEDCBA98,0x76543210) and nonce (0x00000000, 0x00000000) were utilized.

To broaden the study's scope, lightweight encryption and decryption algorithms were also included in the evaluation. The experimental setup aimed to compare **ASCON** with these algorithms under identical conditions. The MT19937-64Bit pseudorandom number generator, seeded with 1, was used to generate 2^n bits of random data for each iteration.

Data Size Variation:

Ensuring a comprehensive analysis, the performance evaluation spanned a range of data sizes, from 128 bits to 671088640 bits (10 GB). The data sizes were selected in a doubling pattern (128, 256, 512, 1024, and so on) to represent a diverse array of input sizes.

After an extensive comparison, taking into account the simplicity of software implementation, alignment with standardization efforts, and an acute awareness of our limitations, we have selected the algorithms detailed in **Tables 4 to 6** for comparative performance analysis. This selection is based on performance metrics derived from our evaluation methodology, emphasizing the significance of both efficiency and reliability in cryptographic algorithms.

The performance evaluation was conducted on a CPU platform of Intel64 Family 6 Model 151 Stepping 2 GenuineIntel, operating at approximately 3610 MHz, with a RAM capacity of 65,277 MB.

This hardware configuration was chosen to ensure that the results are representative of a modern computing environment, providing a realistic benchmark for the algorithms' performance.

Evaluation Result:

The comprehensive analysis of the **XCR/Little_OaldresPuzzle_Cryptic** algorithm in comparison to **ASCON** and **ChaCha20** algorithms, as detailed in our internal data and graphical representations, reveals a nuanced performance landscape.

The **XCR/Little_OaldresPuzzle_Cryptic** algorithm, while consistently slower than **ASCON**, exhibits a manageable performance degradation, particularly in the context of larger data sizes. Specifically, our measurements indicate that the **XCR/Little_OaldresPuzzle_Cryptic** algorithm operates at approximately half the speed of **ASCON** under certain time measurement conditions, yet it remains competitive, suggesting that its utility in practical applications is not significantly hindered by this discrepancy.

The performance metrics, as depicted in **Figures 1, 2**, further support these findings. The scalability and efficiency of **ASCON** are underscored, especially when handling larger data sizes, while **ChaCha20** shows a similar trend, albeit with different performance characteristics. The **XCR/Little_OaldresPuzzle_Cryptic** algorithm, despite its slower performance, maintains a steady pace, indicating a potential for use in environments where computational overhead is a consideration.

In conclusion, the performance evaluation positions **ASCON** as a leading candidate for lightweight cryptographic applications, particularly where rapid processing is critical. The **XCR/Little_OaldresPuzzle_Cryptic** algorithm, while not matching the speed of **ASCON** or **ChaCha20**, remains a viable alternative, especially in scenarios where a balance between speed and computational resources is essential. The choice between these algorithms should be guided by the specific requirements of the application, with careful consideration of the trade-offs between speed, efficiency, and resource utilization.

It is important to acknowledge that the performance metrics presented are based on a controlled environment and may vary under different conditions or with different hardware configurations. Future research should explore the robustness of these algorithms across a broader range of scenarios and hardware platforms to provide a more comprehensive understanding of their practical applicability.

The findings contribute to the ongoing discourse on the selection and optimization of cryptographic algorithms for modern security applications, emphasizing the need for algorithms that balance performance, efficiency, and resource consumption. The results provide valuable insights for developers and researchers in the field of cryptography, aiding in the development of secure and efficient systems.

The study's limitations, including the specific hardware used for the evaluation and the potential for varying results under different conditions, should be recognized. Further research is recommended to validate these findings and to explore the performance of these algorithms in a wider array of environments. This will ensure that the algorithms' performance is thoroughly understood in diverse settings, allowing for informed decisions in the implementation of cryptographic solutions.

7.2 Security Evaluation with Statistical Tests

In the context of data security assessments, our expertise in advanced mathematical techniques is not extensive; however, we employ a methodology that emulates the properties of uniform randomness. This necessitates the application of an efficacious and robust statistical analysis to substantiate the unpredictability of the subsequent bit generation within our algorithm. The methodology entails restricting each encryption operation to a 128-bit data segment, subsequently writing the resultant samples into files, each comprising 128 kilobytes, for a total of 128 sample files. The evaluation process is bifurcated into two distinct phases. In the **first phase**, the plaintext is uniformly set to all zeroes, while the key is utilized as a unique incrementing counter. Conversely, in the **second phase**, the plaintext is derived from a sequence of random data generated by the MT19937-64Bit algorithm, with the key maintaining its role as a unique incrementing counter

7.3 Statistical Tests for Randomness Assessment

The suite of statistical tests employed in this study is designed to rigorously evaluate the randomness of the bit generation process within our encryption algorithm. These tests are predicated on the assumption that true randomness is characterized by an absence of patterns and a uniform distribution of outcomes. Herein, we delineate the statistical properties each test assesses and their respective applications in the context of cryptographic security.

Monobit Frequency Test

This test evaluates the balance between the occurrences of 0s and 1s in the generated binary sequence. It is predicated on the hypothesis that a truly random sequence should exhibit an equal frequency of both bits.

Block Frequency Test (m=10000)

This test examines the frequency distribution of 1s within blocks of a specified size (m=10000). It is used to detect any deviation from the expected uniform distribution, which could indicate a lack of randomness.

Poker Test (m=4, m=8)

The Poker Test assesses the frequency of specific subsequence patterns within the binary sequence. For m=4 and m=8, it evaluates the occurrence of 2-bit and 4-bit patterns, respectively, to ensure their distribution is uniform.

Overlapping Subsequence Test (m=3, P1, P2; m=5, P1, P2)

This test analyzes the frequency of overlapping subsequences of length m (3 or 5) and their permutations (P1, P2). It is designed to detect any non-random clustering of bit patterns.

Run Tests (Run Count, Run Distribution)

Run Tests measure the total number of runs (sequences of consecutive identical bits) and their distribution across the sequence. These tests are sensitive to the presence of long runs, which may indicate a deviation from randomness.

Longest Run Test (m=10000)

Specifically, the Longest Run Test identifies the longest run of 1s (or 0s) within blocks of size m=10000. It is used to detect any anomalies in the distribution of run lengths.

Binary Derivative Test (k=3, k=7)

The Binary Derivative Test evaluates the randomness of the sequence by considering the differences between consecutive bits (k=3, k=7). It is based on the principle that a random sequence should exhibit no correlation between adjacent bits.

Autocorrelation Test (d=1, d=2, d=8, d=16)

Autocorrelation Tests analyze the correlation between a sequence and its shifted versions (with delays d=1, d=2, d=8, d=16). A random sequence should exhibit minimal autocorrelation.

Matrix Rank Test

This test involves constructing a matrix from the binary sequence and determining its rank. The rank is then compared against expected values for a random sequence, providing insights into the sequence's complexity.

Cumulative Sum Test (Forward, Backward)

Cumulative Sum Tests assess the distribution of the running sum of the binary sequence in both forward and backward directions. These tests are sensitive to the presence of systematic trends in the sequence.

Approximate Entropy Test (m=2, m=5)

Approximate Entropy Tests measure the unpredictability of the sequence by comparing the frequency of patterns of length m (2 or 5) with their overlapping counterparts. It is a measure of the sequence's complexity and unpredictability.

Linear Complexity Test (m=500, m=1000)

Linear Complexity Tests estimate the shortest linear feedback shift register (LFSR) that can generate the sequence. A higher complexity indicates a more random sequence.

Maurer's Universal Statistical Test (L=7, Q=1280)

This test evaluates the sequence against a universal statistical model, comparing the observed frequencies with those expected from a truly random sequence. It is a comprehensive test that assesses multiple statistical properties.

Discrete Fourier Transform Test

The Discrete Fourier Transform Test analyzes the frequency spectrum of the sequence. A random sequence should exhibit a uniform frequency distribution across all frequencies.

In summary, these statistical tests provide a comprehensive and systematic framework for assessing the randomness of our encryption algorithm's output. By ensuring that the generated bits pass these rigorous evaluations, we can assert the cryptographic strength of our algorithm, thereby safeguarding the security of the data it encrypts.

7.4 The Credibility of Statistical Tests and the Rationale for Their Use

The statistical tests mentioned above are widely recognized and utilized in the field of cryptography and information security due to their ability to provide a quantitative measure of randomness. These tests are designed to detect patterns and biases that may indicate a lack of randomness, which is a critical property for secure cryptographic operations.

The credibility of these tests stems from their mathematical rigor and empirical validation. They are based on well-established statistical principles and have been extensively analyzed and tested across various scenarios. The tests are not only theoretically sound but also practically effective, having been applied in numerous security evaluations and standards, such as the National Institute of Standards and Technology (NIST) guidelines and China Randomness test specification(GM/T 0005-2021 standards).

The rationale for employing these tests in our evaluation process is multifaceted. Firstly, they offer an objective and systematic approach to assessing the randomness of our encryption algorithm's output. By subjecting the generated bits to a battery of tests, we can gain confidence in the algorithm's ability to produce unpredictable sequences, which is essential for thwarting potential attacks that rely on the predictability of the encryption process.

Secondly, the use of multiple tests provides a comprehensive assessment. Each test targets a different aspect of randomness, and together they form a robust evaluation framework. This ensures that any weaknesses in the algorithm's randomness are likely to be detected, as no single test can cover all possible scenarios.

Lastly, the choice of tests and parameters is informed by the specific requirements of our encryption algorithm and the nature of the data being encrypted. For instance, the block size of 128 bits and the use of a unique incrementing counter as a key are tailored to the algorithm's design, and the tests are selected to effectively evaluate this configuration.

7.5 Understanding Conditional Probability in Statistical Analysis

In the context of cryptographic analysis, understanding conditional probability, denoted as $P(p_1|p_2)$, holds pivotal significance. This probability measures the likelihood of a specific event p_1 occurring given that another event p_2 has already taken place. For instance, in the analysis of encryption algorithms' bit frequency distributions, $P(p_1|p_2)$ elucidates the probability of observing a certain bit pattern p_1 in the encrypted output, conditioned on the occurrence of another specific pattern p_2.

By comprehensively evaluating conditional probabilities across various scenarios, analysts can discern intricate relationships between different bit patterns within the encrypted data. This understanding enables a nuanced assessment of the algorithm's behavior, elucidating potential dependencies or correlations between specific bits. Such insights are invaluable in identifying irregularities or vulnerabilities that adversaries might exploit to compromise the encryption scheme's security.

Moreover, conditional probability analysis augments the robustness of cryptographic evaluations by facilitating a deeper understanding of the algorithm's underlying randomness properties. By scrutinizing conditional probabilities, researchers can uncover subtle deviations from expected randomness, thereby pinpointing areas for algorithmic refinement or strengthening cryptographic protocols.

Thus, integrating conditional probability analysis into cryptographic assessments enhances the rigor and comprehensiveness of security evaluations, empowering stakeholders to make informed decisions regarding algorithm selection, configuration, and deployment in real-world scenarios.

7.5.1 Test Result With Phases 1

In Phase 1 of the **Chacha20** evaluation, wherein the data is set to all zeros and the key operates in Counter Mode, all statistical tests conducted failed to pass, regardless of attempts at both 32-bit and 64-bit trials. This highlights Chacha20's suboptimal performance when subjected to non-random data, particularly in the 64-bit scenario.

Our test results reveal that the **XCR/Little_OaldresPuzzle_Cryptic** algorithm not only closely mirrors the performance characteristics of the **ASCON** algorithm, as inferred from the aforementioned conclusions, but also maintains a level of random uniformity indistinguishable from the **ASCON** algorithm. This assertion will be visually substantiated through a table presented in the **Figure 3 to 6** included in the appendix.

It is essential to note that due to the extensive nature of our testing, yielding 128 individual data files, we have chosen the final binary file as our reference dataset. Consequently, all data presented in the tables within our screenshots is derived exclusively from the results of this last binary file. The comprehensive statistical test results, including the complete Excel spreadsheet, will be made available in our code repository. HereIsLink

7.5.2 Test Result With Phases 2

Transitioning to Phase 2, where data is generated using MT19937-64Bit with Seed 1 and the key is a random 64-bit value (32-bit * 2), **Chacha20** exhibits significantly improved performance. In this phase, all tests yielded satisfactory or excellent results.

It becomes evident that **Chacha20** encounters challenges in maintaining security when confronted with non-random input data, underscoring the algorithm's dependence on randomness, especially in the 64-bit context.

Chacha20's strengths lie in its capacity to provide ample bit distribution, approaching pseudo-randomness without fully achieving it, even with minimal input data. It excels in scenarios where the input data exhibits sufficient chaos, showcasing the intricacies of the algorithm.

In conclusion, while **Chacha20** demonstrates merits, especially in Phase 2 where randomness is better preserved, the **XCR/Little_OaldresPuzzle_Cryptic** algorithm emerges as the superior choice when **Chacha20** struggles to uphold security and robustness, particularly in non-random data scenarios. The test results will be reflected in **Figure 7 to 10**.

7.5.3 Evaluate Test Results

Upon synthesizing the insights garnered from both Phase 1 and Phase 2 evaluations, a nuanced appreciation of the Chacha20 algorithm's efficacy is achieved. Phase 1 revealed its vulnerability to non-random data, which resulted in suboptimal outcomes across multiple iterations. Conversely, Phase 2 demonstrated a significant enhancement in performance, particularly when the input data conformed to a more stochastic distribution.

XCR/Little_OaldresPuzzle_Cryptic algorithm has consistently exhibited competitive performance, maintaining a level of randomness akin to the ASCON algorithm. This resilience to data patterns underscores its potential in environments where randomness is paramount.

Given the observed performance of **Chacha20**, it becomes apparent that it may not be an ideal candidate for consideration in subsequent evaluations. Consequently, **Chacha20** has been excluded from our pool of reference algorithms for further assessment. This decision narrows our focus to the **ASCON** algorithm and the **XCR/Little_OaldresPuzzle_Cryptic** algorithm as the primary contenders for comparison and evaluation. The upcoming analysis will scrutinize and compare the strengths and weaknesses of **ASCON** and **XCR/Little_OaldresPuzzle_Cryptic** to determine their suitability for our intended application.

In addressing concerns regarding the perceived issues with **Chacha20**, it is imperative to substantiate our assertions with robust and compelling data. To enhance the persuasiveness of our findings, comprehensive statistical results are meticulously presented in the tables within the appendix, accompanied by graphical representations in **Figures 11 to 14**. This transparent approach aims to provide readers with direct access to the raw data, fostering a clearer understanding of the intricacies and nuances of Chacha20's performance. We believe that this meticulous presentation of data in our supplementary materials will dispel any reservations and contribute to the confidence in the validity of our analysis.

8 Conclusion

In the holistic development of our algorithm, **XCR/Little_OaldresPuzzle_Cryptic**, we drew inspiration from the proven efficiency of the **ASCON** algorithm in real-world test evaluations and its pseudo-random indistinguishability, closely approaching a uniform random distribution. Leveraging these insights, we ensured our algorithm provides robust security while addressing the challenge of a substantial number of constant arrays needed for the values of nonlinear pseudo-random functions. To streamline the design structure, we focused on concise enhancements, distinguishing our approach from both the **ASCON** algorithm and **Chacha20**. This strategic refinement empowers users to make informed and balanced decisions based on their specific requirements.

This paper introduces our innovative symmetric sequence cryptographic algorithm, **XCR/Little_OaldresPuzzle_Cryptic**, showcasing its exceptional speed in encryption and decryption along with robust security features. In the ever-evolving digital landscape, our algorithm emerges as a cutting-edge solution for securely and efficiently managing large-scale binary data files. Notably, the inclusion of a quantum-resistant block cipher algorithm in the same repository highlights our commitment to anticipating and addressing future cryptographic challenges.

A About Git repository and run test

While our focus in this paper revolves around the symmetric sequence algorithm, our repository offers a comprehensive perspective on our cryptographic contributions. We invite interested readers and researchers to explore both algorithms, contribute insights, and collaborate on potential enhancements. In the dynamic realm of cryptography, collective efforts and continuous innovation are indispensable for staying prepared for future challenges.

The repository encompasses not only the OaldresPuzzle_Cryptic algorithm discussed in this paper but also its quantum-resistant counterpart, designed for future cryptographic needs. Despite being slower in packet encryption and decryption (based on test data of 10MB and a key size of 5120 bytes), it contributes to the overall resilience of the cryptographic system. This section is intended to guide users in testing these algorithms using the code available in the repository.

Link: README.md

A.1 Environment Setup

Before you begin, ensure you have an environment compatible with the code. Clone the repository using the following link:

XorConstantRotation.cpp
LittleOaldresPuzzle_Cryptic.cpp
gist

Next, install the required software and libraries as stated in the README.md file..

A.2 Running the Tests

Within the repository, you will find two primary directories, *OOP* and *Template*. The *OOP* directory offers an object-oriented version of the algorithm, ideal for developers familiar with object-oriented programming. Conversely, the *Template* directory presents a simplified implementation, suitable for beginners or those seeking a more straightforward understanding of the algorithm.

Choose the implementation that suits your requirements, navigate to the appropriate directory, and follow the README.md instructions to compile and execute the tests. These tests will furnish a holistic understanding of the algorithm's cryptographic robustness, speed, and overall performance.

A.3 Unique Features and Precautions

It's crucial to note the peculiar characteristics and necessary precautions while using these algorithms. For instance, encryption and decryption operations demand a reset of the internal key state after each use. Therefore, if an encryption operation follows a decryption operation (or vice versa), the internal key state must be reset first to ensure correct functioning.

A.4 Interpreting the Results

Once the tests conclude, you will obtain a set of numerical performance metrics. These results, available in the README.md file, enable a performance comparison between the Algorithm Little OaldresPuzzle_Cryptic and other prevalent algorithms. However, while comparing, remember that an algorithm's efficacy isn't only about speed but also about security. Hence, consider these test results thoughtfully, factoring in the complexity of the sequences generated by the algorithm and its resilience against attacks.

A.5 Contributions

We value your feedback and contributions. If you encounter possible improvements or any issues, feel free to submit a pull request or open an issue in the GitHub repository.

B Additional Quantum-Resistant OaldresPuzzle_Cryptic Block Cipher Algorithm

This repository houses an additional robust block cipher algorithm, developed independently from the symmetric sequence algorithm that is the primary focus of this paper. Conceived with the anticipation of future cryptographic challenges, particularly those presented by quantum computing, this block cipher algorithm offers several defining characteristics. It is designed to mitigate risks associated with brute force attacks, withstand analytical attacks on the key, and resist potential quantum computer intrusions.

Although slower in encrypting and decrypting packets (as evidenced by tests with 10MB data packets and a 5120-byte key, which required approximately one and a half minutes to execute), the algorithm confers a significant advantage by future-proofing cryptographic systems against potential advancements in quantum computing. This symmetric block cipher cryptographic algorithm has been tailored to meet contemporary cryptographic requirements. It prioritizes unpredictability and a high level of analytical complexity, making it suitable for managing protected, large-scale binary data files while ensuring requisite cryptographic robustness.

For a more comprehensive understanding, detailed information regarding the implementation and unique characteristics of this block cipher algorithm can be found in the corresponding directory of the repository. Despite being outside the primary scope of this paper, which is dedicated to the symmetric sequence algorithm, we encourage interested readers and researchers to explore this quantum-resistant block cipher algorithm and its potential applications.

C Documents referenced

References

[Aumasson et al., 2007] Aumasson, J.-P., Fischer, S., Khazaei, S., Meier, W., and Rechberger, C. (2007). New features of latin dances: Analysis of salsa, chacha, and rumba. Cryptology ePrint Archive, Paper 2007/472. https://eprint.iacr.org/2007/472.

[Bernstein, 2005] Bernstein, D. J. (2005). Salsa20 design. *Department of Mathematics, Statistics, and Computer Science. The University of Illinois at Chicago. Chicago.*

[Bernstein, 2008] Bernstein, D. J. (2008). The salsa20 family of stream ciphers. In *New stream cipher designs: the eSTREAM finalists*, pages 84–97. Springer.

[Bernstein et al., 2008a] Bernstein, D. J. et al. (2008a). Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5. Citeseer.

[Bernstein et al., 2008b] Bernstein, D. J. et al. (2008b). The chacha family of stream ciphers. In *Workshop record of SASC*. Citeseer.

[Biryukov and Perrin, 2017] Biryukov, A. and Perrin, L. (2017). State of the art in lightweight symmetric cryptography. Cryptology ePrint Archive, Paper 2017/511. https://eprint.iacr.org/2017/511.

[Cai et al., 2022] Cai, W., Chen, H., Wang, Z., and Zhang, X. (2022). Implementation and optimization of chacha20 stream cipher on sunway taihulight supercomputer. *The Journal of Supercomputing*, 78(3):4199–4216.

[Fei, 2012] Fei, D. (2012). Research on safety of arx structures. Master's thesis, Xi'an University of Electronic Science and Technology, China.

[Ghafoori and Miyaji, 2022] Ghafoori, N. and Miyaji, A. (2022). Differential cryptanalysis of salsa20 based on comprehensive analysis of pnbs. In Su, C., Gritzalis, D., and Piuri, V., editors, *Information Security Practice and Experience*, pages 520–536, Cham. Springer International Publishing.

[Liu et al., 2021] Liu, J., Rijmen, V., Hu, Y., Chen, J., and Wang, B. (2021). Warx: efficient white-box block cipher based on arx primitives and random mds matrix. *Science China Information Sciences*, 65(3):132302.

[Maitra et al., 2015] Maitra, S., Paul, G., and Meier, W. (2015). Salsa20 cryptanalysis: New moves and revisiting old styles. Cryptology ePrint Archive, Paper 2015/217. https://eprint.iacr.org/2015/217.

[Ranea et al., 2022] Ranea, A., Vandersmissen, J., and Preneel, B. (2022). Implicit white-box implementations: White-boxing arx ciphers. In Dodis, Y. and Shrimpton, T., editors, *Advances in Cryptology – CRYPTO 2022*, pages 33–63, Cham. Springer Nature Switzerland.

[Serrano et al., 2022] Serrano, R., Duran, C., Sarmiento, M., Pham, C.-K., and Hoang, T.-T. (2022). Chacha20-poly1305 authenticated encryption with additional data for transport layer security 1.3. *Cryptography*, 6(2).

[Sleem and Couturier, 2021] Sleem, L. and Couturier, R. (2021). Speck-R: An ultra light-weight cryptographic scheme for Internet of Things. *Multimedia Tools and Applications*, 80(11):17067 – 17102.

[Tsunoo et al., 2007] Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T., and Nakashima, H. (2007). Differential cryptanalysis of salsa20/8. In *Workshop Record of SASC*, volume 28. Citeseer.

D Data Images

E Data Tables

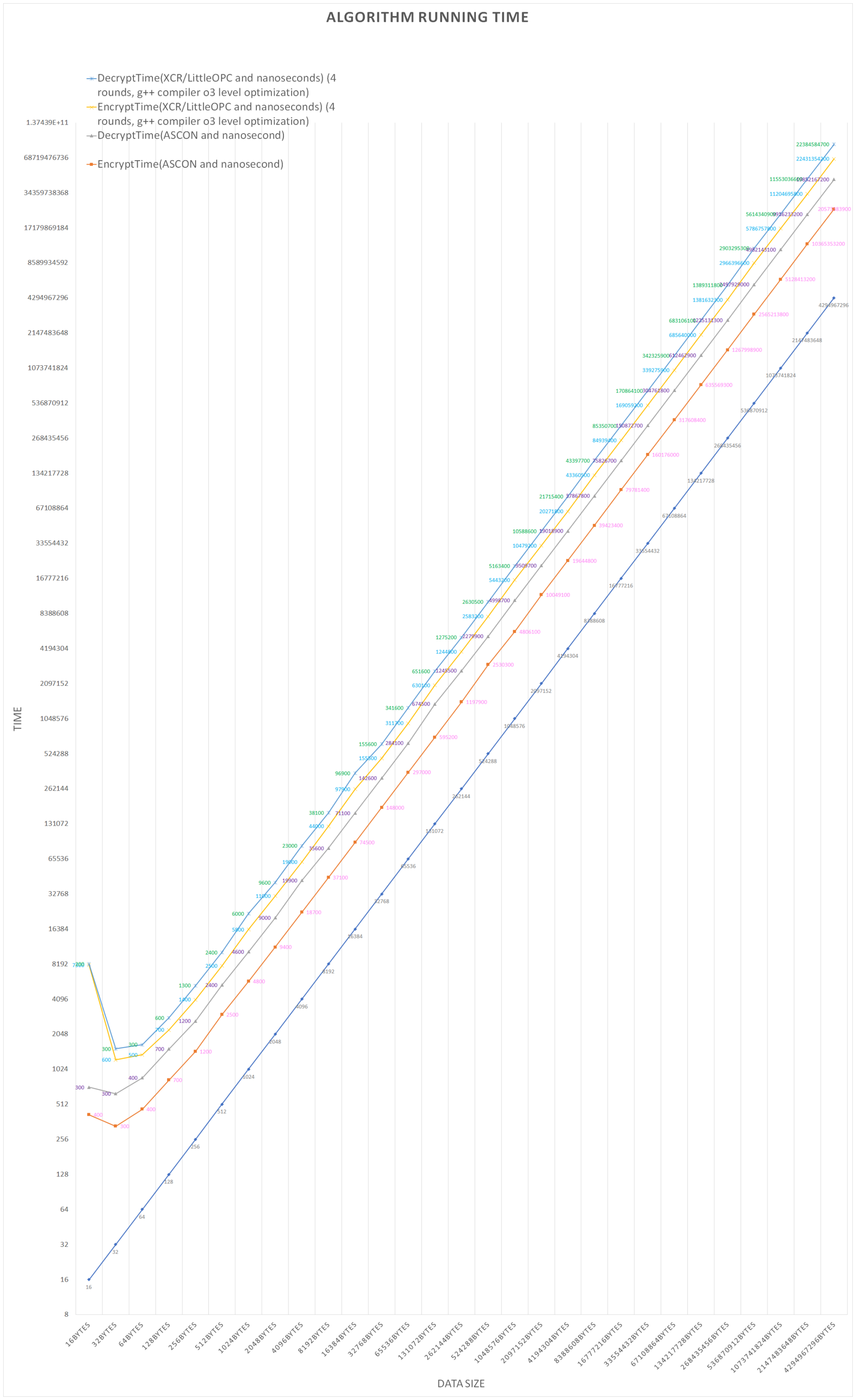


Figure 1: ASCON vs XCR/Little_OaldresPuzzle_Cryptic Algorithm Benchmark

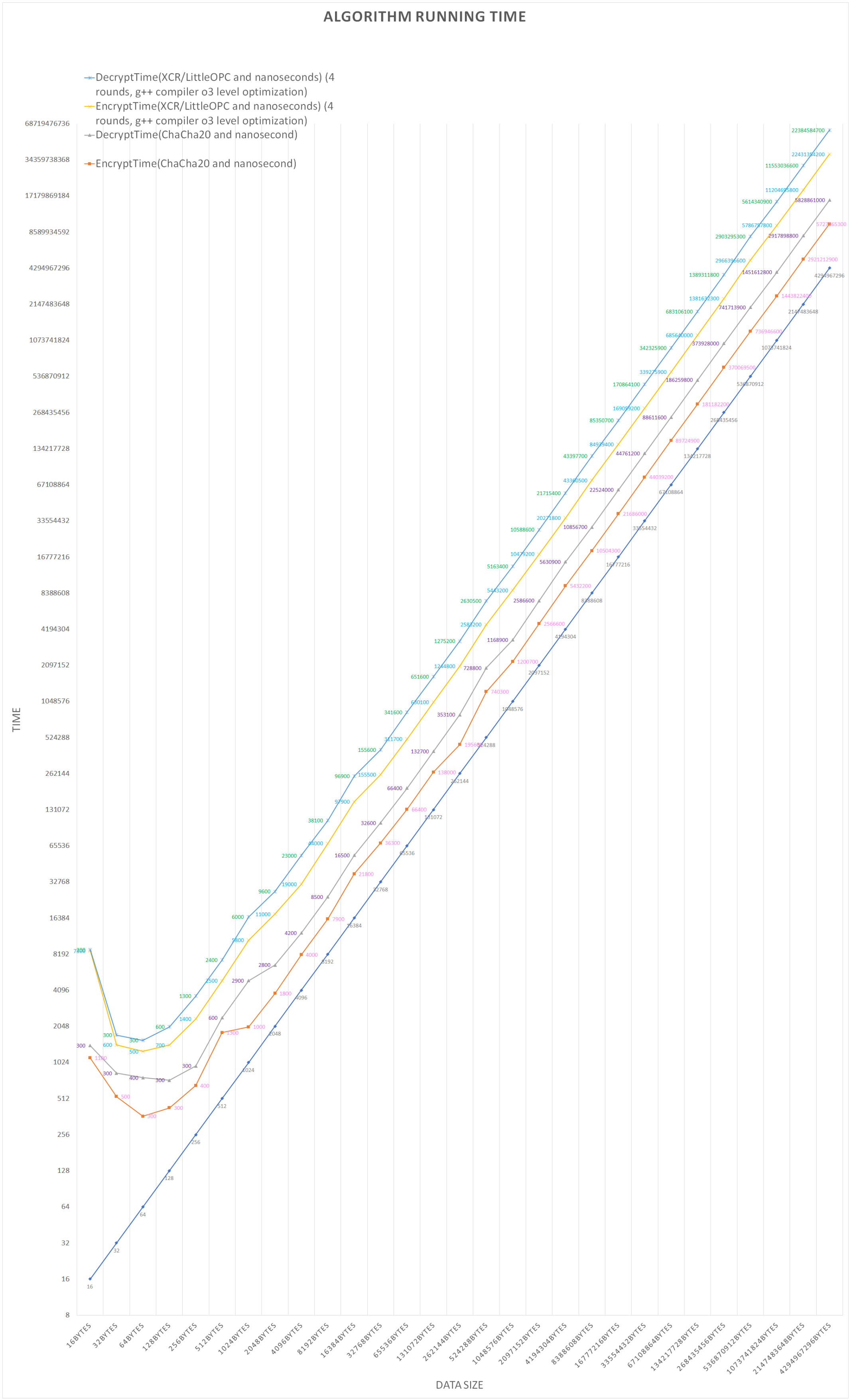
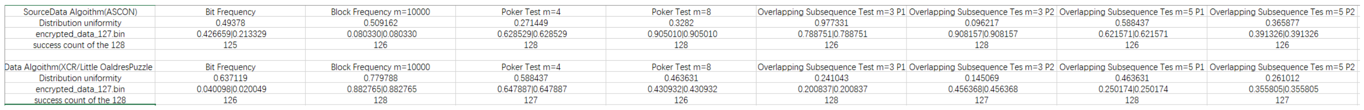
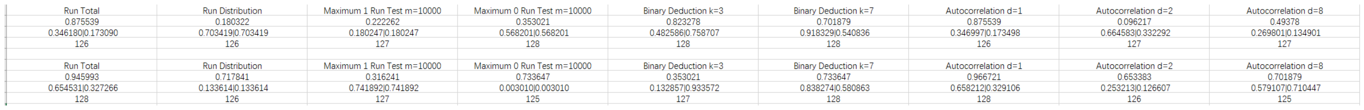


Figure 2: Chacha20 vs XCR/Little_OaldresPuzzle_Cryptic Algorithm Benchmark



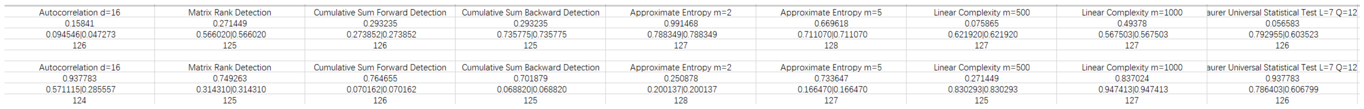
SourceData Algoithm(ASCON)	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0.49378	0.509162	0.271449	0.3282	0.977331	0.096217	0.588437	0.365877
encrypted_data_127.bin	0.426659|0.213329	0.080330|0.080330	0.628529|0.628529	0.905010|0.905010	0.788751|0.788751	0.908157|0.908157	0.621571|0.621571	0.391326|0.391326
success count of the 128	125	126	128	128	126	128	126	126
Data Algoithm(XCR/Little OaldresPuzzle	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0.637119	0.779788	0.588437	0.463631	0.241043	0.145069	0.463631	0.261012
encrypted_data_127.bin	0.040098|0.020049	0.882765|0.882765	0.647887|0.647887	0.430932|0.430932	0.200837|0.200837	0.456368|0.456368	0.250174|0.250174	0.355805|0.355805
success count of the 128	126	128	127	126	128	127	128	127

Figure 3: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 1) Statistical Test



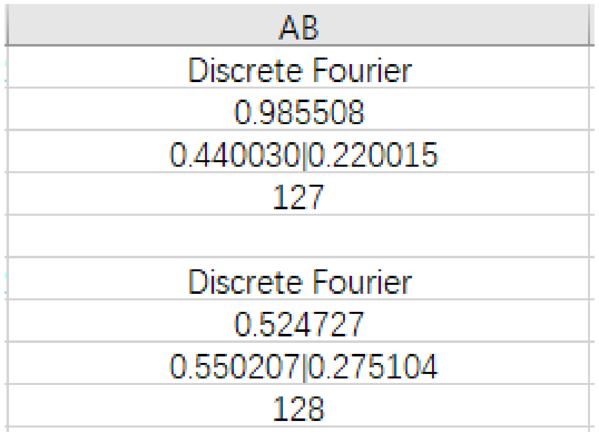
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0.875539	0.180322	0.222262	0.353021	0.823278	0.701879	0.875539	0.096217	0.49378
0.346180|0.173090	0.703419|0.703419	0.180247|0.180247	0.568201|0.568201	0.482586|0.758707	0.918329|0.540836	0.346997|0.173498	0.664583|0.332292	0.269801|0.134901
126	126	127	128	128	128	126	127	127
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0.945993	0.717841	0.316241	0.733647	0.353021	0.733647	0.966721	0.653383	0.701879
0.654531|0.327266	0.133614|0.133614	0.741892|0.741892	0.003010|0.003010	0.132857|0.933572	0.838274|0.580863	0.658212|0.329106	0.253213|0.126607	0.579107|0.710447
128	126	127	125	127	128	128	126	125

Figure 4: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 1) Statistical Test



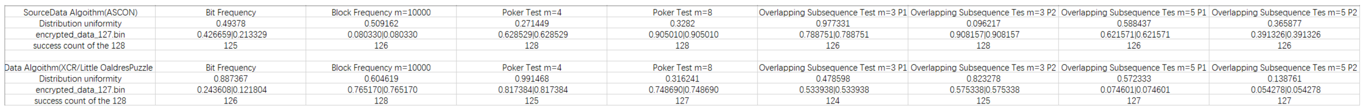
Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0.15841	0.271449	0.293235	0.293235	0.991468	0.669618	0.075865	0.49378	0.056583
0.094546|0.047273	0.566020|0.566020	0.273852|0.273852	0.735775|0.735775	0.788349|0.788349	0.711070|0.711070	0.621920|0.621920	0.567503|0.567503	0.792955|0.603523
126	125	126	125	127	128	127	127	126
Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0.937783	0.749263	0.764655	0.701879	0.250878	0.733647	0.271449	0.837024	0.937783
0.571115|0.285557	0.314310|0.314310	0.070162|0.070162	0.068820|0.068820	0.200137|0.200137	0.166470|0.166470	0.830293|0.830293	0.947413|0.947413	0.786403|0.606799
124	125	126	125	128	127	125	127	126

Figure 5: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 1) Statistical Test



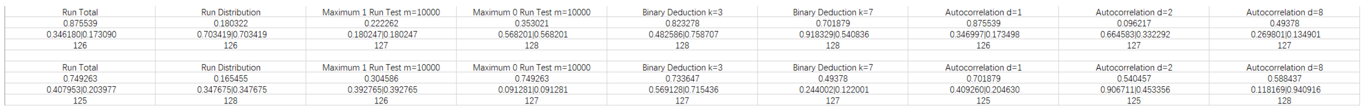
AB
Discrete Fourier
0.985508
0.440030|0.220015
127
Discrete Fourier
0.524727
0.550207|0.275104
128

Figure 6: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 1) Statistical Test



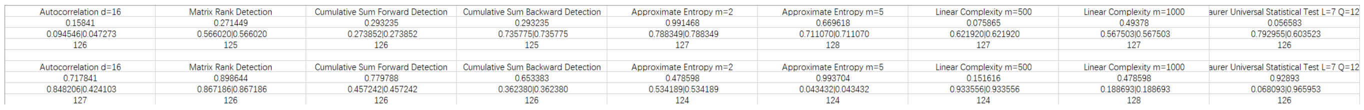
SourceData Algoithm(ASCON)	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0.49378	0.509162	0.271449	0.3282	0.977331	0.096217	0.588437	0.365877
encrypted_data_127.bin	0.426659|0.213329	0.080330|0.080330	0.628529|0.628529	0.905010|0.905010	0.788751|0.788751	0.908157|0.908157	0.621571|0.621571	0.391326|0.391326
success count of the 128	125	126	128	128	126	128	126	126
Data Algoithm(XCR/Little OaldresPuzzle	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0.887367	0.604619	0.991468	0.316241	0.478598	0.823278	0.572333	0.138761
encrypted_data_127.bin	0.243608|0.121804	0.765170|0.765170	0.817384|0.817384	0.748690|0.748690	0.533938|0.533938	0.575338|0.575338	0.074601|0.074601	0.054278|0.054278
success count of the 128	126	128	125	127	124	125	127	127

Figure 7: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 2) Statistical Test



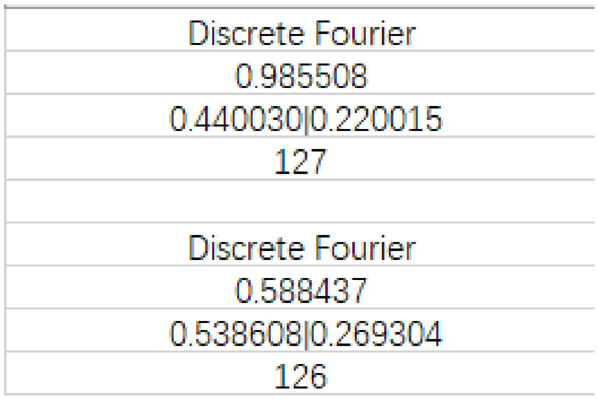
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0.875539	0.180322	0.222262	0.353021	0.823278	0.701879	0.875539	0.096217	0.49378
0.346180|0.173090	0.703419|0.703419	0.180247|0.180247	0.568201|0.568201	0.482586|0.758707	0.918329|0.540836	0.346997|0.173498	0.664583|0.332292	0.269801|0.134901
126	126	127	128	128	128	126	127	127
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0.749263	0.165455	0.304586	0.749263	0.733647	0.49378	0.701879	0.540457	0.588437
0.407953|0.203977	0.347675|0.347675	0.392765|0.392765	0.091281|0.091281	0.569128|0.715436	0.244002|0.122001	0.409260|0.204630	0.906711|0.453356	0.118169|0.940916
125	128	126	127	127	127	125	125	128

Figure 8: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 2) Statistical Test



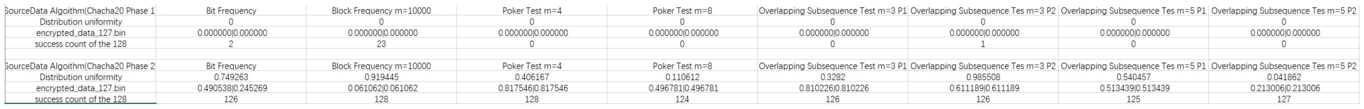
Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0.15841	0.271449	0.293235	0.293235	0.991468	0.669618	0.075865	0.49378	0.056583
0.094546|0.047273	0.566020|0.566020	0.273852|0.273852	0.735775|0.735775	0.788349|0.788349	0.711070|0.711070	0.621920|0.621920	0.567503|0.567503	0.792955|0.603523
126	125	126	125	127	128	127	127	126
Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0.717841	0.898644	0.779788	0.653383	0.478598	0.993704	0.151616	0.478598	0.92893
0.848206|0.424103	0.867186|0.867186	0.457242|0.457242	0.362380|0.362380	0.534189|0.534189	0.043432|0.043432	0.933556|0.933556	0.188693|0.188693	0.068093|0.965953
127	126	126	126	124	124	124	128	126

Figure 9: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 2) Statistical Test



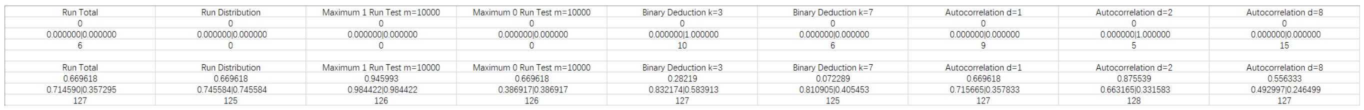
Discrete Fourier
0.985508
0.440030|0.220015
127
Discrete Fourier
0.588437
0.538608|0.269304
126

Figure 10: ASCON vs XCR/Little_OaldresPuzzle_Cryptic(Phase 2) Statistical Test



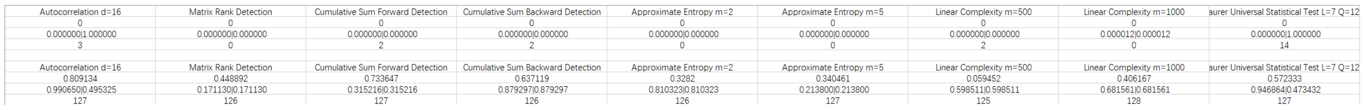
SourceData Algoithm(Chacha20 Phase 1	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0	0	0	0	0	0	0	0
encrypted_data_127.bin	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000
success count of the 128	2	23	0	0	0	1	0	0
SourceData Algoithm(Chacha20 Phase 2	Bit Frequency	Block Frequency m=10000	Poker Test m=4	Poker Test m=8	Overlapping Subsequence Test m=3 P1	Overlapping Subsequence Tes m=3 P2	Overlapping Subsequence Tes m=5 P1	Overlapping Subsequence Tes m=5 P2
Distribution uniformity	0.749263	0.919445	0.406167	0.110612	0.3282	0.985508	0.540457	0.041862
encrypted_data_127.bin	0.490538|0.245269	0.061062|0.061062	0.817546|0.817546	0.496781|0.496781	0.810226|0.810226	0.611189|0.611189	0.513439|0.513439	0.213006|0.213006
success count of the 128	126	128	128	124	126	126	125	127

Figure 11: Chacha20((Phase 1 vs Phase 2) Statistical Test



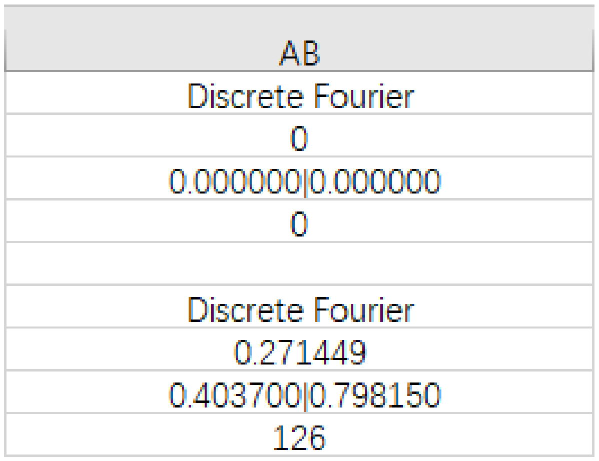
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0	0	0	0	0	0	0	0	0
0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|1.000000	0.000000|0.000000	0.000000|0.000000	0.000000|1.000000	0.000000|0.000000
6	0	0	0	10	6	9	5	15
Run Total	Run Distribution	Maximum 1 Run Test m=10000	Maximum 0 Run Test m=10000	Binary Deduction k=3	Binary Deduction k=7	Autocorrelation d=1	Autocorrelation d=2	Autocorrelation d=8
0.669618	0.669618	0.945993	0.669618	0.28219	0.072289	0.669618	0.875539	0.556333
0.714590|0.357295	0.745584|0.745584	0.984422|0.984422	0.386917|0.386917	0.832174|0.583913	0.810905|0.405453	0.715665|0.357833	0.663165|0.331583	0.492997|0.246499
127	125	126	126	127	125	127	128	127

Figure 12: Chacha20((Phase 1 vs Phase 2) Statistical Test



Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0	0	0	0	0	0	0	0	0
0.000000|1.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000000|0.000000	0.000012|0.000012	0.000000|1.000000
3	0	2	2	0	0	2	0	14
Autocorrelation d=16	Matrix Rank Detection	Cumulative Sum Forward Detection	Cumulative Sum Backward Detection	Approximate Entropy m=2	Approximate Entropy m=5	Linear Complexity m=500	Linear Complexity m=1000	aurer Universal Statistical Test L=7 Q=12
0.809134	0.448892	0.733647	0.637119	0.3282	0.340461	0.059452	0.406167	0.572333
0.990650|0.495325	0.171130|0.171130	0.315216|0.315216	0.879297|0.879297	0.810323|0.810323	0.213800|0.213800	0.598511|0.598511	0.681561|0.681561	0.946864|0.473432
127	126	127	126	126	127	125	128	127

Figure 13: Chacha20((Phase 1 vs Phase 2) Statistical Test



AB
Discrete Fourier
0
0.000000|0.000000
0
Discrete Fourier
0.271449
0.403700|0.798150
126

Figure 14: Chacha20((Phase 1 vs Phase 2) Statistical Test

Algorithm	Type	Key size (bits)	Block size (bits)	AEAD mode
Ascon	Block cipher	80 or 128	64	Yes
LED	Block cipher	64 or 128	64	No
PHOTON	Hash function	N/A	N/A	No
SPONGENT	Hash function	N/A	N/A	No
PRESENT	Block cipher	80 or 128	64	No
CLEFIA	Block cipher	128, 192, or 256	128	No
LEA	Block cipher	128, 192, or 256	128	No
Grain-128a	Stream cipher	128	N/A	No
Enocoro-128v2	Stream cipher	128	N/A	No
Lesamnta-LW	Hash function	N/A	N/A	No
ChaCha	Stream cipher	128 or 256	N/A	Yes
LBlock	Block cipher	80	64	No
SIMECK	Block cipher	64 or 128	32, 48, or 64	No
SIMON	Block cipher	64, 72, 96, 128, 144, or 192	32, 48, 64, 96, or 128	No
PRIDE	Block cipher	128	64	No
TWINE	Block cipher	80 or 128	64	No
ESF	Block cipher	128	128	No

Table 1: Cryptography Algorithm Specifications (No Internal State or Initial Vector)

Algorithm	Rounds of use round function
Ascon	12 or 8
LED	48 or 32
PRESENT	31
CLEFIA	18, 22, or 26
LEA	24, 28, or 32
SIMECK	32, 36, or 44
SIMON	32, 36, 42, 44, 52, 54, 68, 69, 72, or 84
PRIDE	20
TWINE	36

Table 2: Rounds of Use Round Function for Cryptography Algorithms

Algorithm	Internal state size (bits)	Initial vector size (bits)
Ascon	320	128
Grain-128a	256	96
Enocoro-128v2	128	128
Lesamnta-LW	256	N/A
ChaCha	512	64 or 96

Table 3: Cryptography Algorithm Specifications (With Internal State and Initial Vector)

Algorithm	Type	Key size (bits)	Block size (bits)	AEAD mode
Ascon	Stream cipher	80 or 128	64	Yes
ChaCha20	Stream cipher	128 or 256	32	Yes
XCR/Little_OaldresPuzzle_Cryptic	Stream cipher	128 or Custom	128 or or Mutable	Yes(eg. Poly1305)

Table 4: The opposite algorithm we chose to compare with (Excluding Internal State and Initialization Vector Sizes, and Rounds of Use Round Function)

Algorithm	Internal State Size (bits)	Initialization Vector Size (bits)
Ascon	64	128
ChaCha20	512	64 or 96
XCR/Little_OaldresPuzzle_Cryptic	192	Not specified

Table 5: Internal State and Initialization Vector Sizes for The opposite algorithm we chose to compare with

Algorithm	**Rounds of use round function**
Ascon	12 or 8
ChaCha20	8, 12, or 20
XCR/Little_OaldresPuzzle_Cryptic	Custom

Table 6: Rounds of Use Round Function for The opposite algorithm we chose to compare with