cryptography

MDPI

Article

A Scalable Symmetric Cryptographic Scheme Based on Latin Square, Permutations, and Reed-Muller Codes for Resilient Encryption

Hussain Ahmad * and Carolin Hannusch

Department of Computer Science, Faculty of Informatics, University of Debrecen, 4028 Debrecen, Hungary; hannusch.carolin@inf.unideb.hu

* Correspondence: hussain.ahmad@inf.unideb.hu

Abstract

Symmetric cryptography is essential for secure communication as it ensures confidentiality by using shared secret keys. This paper proposes a novel substitution-permutation network (SPN) that integrates Latin squares, permutations, and Reed-Muller (RM) codes to achieve robust security and resilience. As an adaptive design using binary representation with base-n Latin square mappings for non-linear substitutions, it supports any n (Codeword length and Latin square order), k (RM code dimension), d (RM code minimum distance) parameters aligned with the Latin square and RM(n, k, d) codes. The scheme employs $2\log_2 n$-round transformations using $\log_2 n$ permutations ρ_z, where in the additional $\log_2 n$ rounds, row and column pairs are swapped for each pair of rounds, with key-dependent π_z permutations for round outputs and fixed ρ_z permutations for codeword shuffling, ensuring strong diffusion. The scheme leverages dynamic Latin square substitutions for confusion and a vast key space, with permutations ensuring strong diffusion and RM(n, k, d) codes correcting transmission errors and enhancing robustness against fault-based attacks. Precomputed components optimize deployment efficiency. The paper presents mathematical foundations, security primitives, and experimental results, including avalanche effect analysis, demonstrating flexibility and balancing enhanced security with computational and storage overhead.

Keywords: symmetric cryptography; substitution-permutation network; Latin square; permutation; Reed-Muller code; error correction

check for updates

Academic Editor: Carlo Blundo

Received: 17 September 2025
Revised: 26 October 2025
Accepted: 29 October 2025
Published: 31 October 2025

Citation: Ahmad, H.; Hannusch, C. A Scalable Symmetric Cryptographic Scheme Based on Latin Square, Permutations, and Reed-Muller Codes for Resilient Encryption. *Cryptography* **2025**, *9*, 70. https://doi.org/10.3390/cryptography9040070

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Symmetric cryptography, especially block ciphers, plays a fundamental role in secure communication by ensuring confidentiality through the use of shared secret keys. Block ciphers encrypt fixed-size blocks of plaintext into ciphertext using a series of iterative transformations. They are highly valued for their efficiency and security [1]. Traditional block ciphers are strong but often do not include built-in error correction, which limits their effectiveness in various environments. As a result, modern systems encounter challenges like transmission errors in noisy channels, fault-based attacks that exploit hardware vulnerabilities, and the need for adaptable, scalable designs that strike a balance between security and resource constraints [2]. This aspect has gained importance in addressing these attacks [3–5].

Recent studies have employed Latin squares for cryptographic substitutions, utilizing their nonlinear properties and combinatorial diversity. The Zefreh and Abdali Latin

Square Image Encryption (LSIE) scheme uses Latin square S-boxes with chaotic systems for fast and noise-resilient image encryption [6]. Walid El-Shafai, et al. developed a compression-encryption scheme using Latin squares for IoT multimedia security, achieving low computational overhead to guarantee security and high speed without compromising the complexity [7]. The Latin square image cipher (LSIC) of Wu et al. uses Latin square whitening, Latin square S-box, and Latin square P-box for confusion and diffusion [8]. A hybrid model was presented by Nada Ali et al. that uses the Latin square matrix (LSM) and subtractive random number generator (SRNG) algorithms to generate random keys that raise the degree of diffusion and strengthen the cipher key's defense against various attacks [9]. These schemes, often designed for image coding, lack general applicability and resistance to noise or bit-flipping attacks, challenges addressed in our scheme through Reed-Muller codes.

Error-correcting codes (ECCs) play a vital role in ensuring data integrity and improving the reliability and security of cryptographic systems [10]. Repka et al. discussed the use of ECCs in secure protocols and their effectiveness in mitigating fault attacks [10]. Alabady's low complexity product code (LCPC) offers an efficient method of error correction that is particularly suitable for Internet of Things (IoT), which often have limited resources [11]. Boneh et al. highlighted the importance of ECCs in eliminating computational errors, which is crucial for protecting against fault attacks in block ciphers [12]. Carlet investigated the use of RM codes in Boolean functions to enhance resistance against fault attacks [13]. Therefore, integrating ECCs with cryptographic systems is highly recommended to prevent errors, as even a single-bit change in an encrypted message can cause significant damage in the decrypted output [14].

We propose a novel block cipher based on an SPN that integrates Latin squares, permutations, and Reed-Muller codes to deliver robust security and resilience in noisy and adversarial environments. Our adaptive design supports any parameters n, k, d and utilizes binary representation along with base-n Latin square mappings. The cipher incorporates Latin square S-boxes to enhance confusion, applies diffusion permutations, and employs RM codes, which add an extra layer of security, enable error correction, and provide resistance to fault attacks. It features a scalable key space and parameter flexibility, optimizing the balance between security and resource usage. The scheme operates over $2\log_2 n$ rounds. This framework is versatile for symmetric cryptography, supporting independent encoding and decoding processes that allow for parallelization in multi-core or distributed systems. Precomputed components are included to ensure deployment efficiency across various applications. The paper concludes with an evaluation of both security and performance.

The structure of our paper is organized as follows: Section 2 introduces a background for primitives. Section 3 describes the cryptographic system. Section 4 presents experimental results. Section 5 evaluates security properties. Section 6 discusses limitations and outlines future directions.

2. Background

2.1. Latin Squares

Let n (order) be a non-negative integer. A Latin square $L = (l_{i,j})_{1 \le i,j \le n}$ of order n is an $n \times n$ array with entries $l_{i,j} \in \{0, \ldots, n-1\}$, such that each element of the set $\{0, \ldots, n-1\}$, is contained exactly once in each row and in each column.

A key defined over key space $K = \{0, \ldots, n-1\}$ is used and distributed uniformly over the message space $M = \{0, \ldots, n-1\}$ [15].

A Latin square of order n (on the symbol set $1, 2, \ldots, n$ or on the symbol set $0, 1, \ldots, n-1$) is reduced or in standard form if in the first row and column the symbols occur in natural order as shown in Table 1.

Table 1. A reduced Latin square of order 3 [16].

1	2	3
2	3	1
3	1	2

A cipher is defined over the message space and the key space, and the function

$$y = C_k(m) = l_{k,m} \tag{1}$$

describes the encryption of a plaintext $m \in M$ under a key $k \in K$ [17].

2.1.1. Relation Between Cayley Tables, Quasigroups, and Latin Squares

A Cayley table [18,19], which resembles an addition or multiplication table in square form, represents the structure of a finite group [20], by arranging all possible products of all of the group's components. An order-n Latin square, with each row (or column) encoding a permutation of the group's components, is what an unbordered Cayley table of an order-n group essentially is, according to a Cayley discovery [18,19,21].

Theorem 1. *If $(G, \bigoplus)$ is a finite group of order n and L^G is the unbordered Cayley table of G, then L^G is a Latin square of order n.*

Definition 1. *A set G equipped with a binary operation $*$ is said to be a quasigroup if the following conditions are satisfied:*

- *G is closed under the operation: $q_i * q_j \in G, \forall q_i, q_j \in G$.*
- *The Latin square property holds: $\forall q_i, q_j \in G, \exists! x, y \in G, q_i * x = q_j$ and $y * q_i = q_j$.*

G is said to be a finite quasigroup if it contains a finite number of elements.

Theorem 2. *If $G = (G, *)$ is a finite quasigroup and L^G is the unbordered Cayley table of G, then L^G is a Latin square.*

From Definition 1 and Theorem 2, we guarantee that every element of G appears exactly once in each row and once in each column of the Cayley table of G. It is an elementary property of all groups and the defining property of all quasigroups. In Section 3.2.4, we will illustrate how to construct a Latin square by creating a quasi-group from a finite group using permutations.

2.1.2. Conjugacy Class

Each cell of a Latin square can be represented by a triple (row, column, symbol). One method to represent L is to conceive of a set of ordered triples instead of a Latin square [22]. If $L = l_{i,j}$ is a Latin square of order n, the corresponding n^2 triples are:

$$T_L = \{(r_i, c_j, l_{i,j}) : 0 \leq i, j \leq n-1\} \tag{2}$$

The entry that appears at a particular position inside a triple is referred to as a coordinate.

A Latin square's isotopism L is a permutation of L that permutes its symbols, rows, and columns. Another Latin square that is stated to be isotopic to L is the result [23].

The operation of permuting the coordinates themselves is referred to as conjugacy or parastrophy. For two Latin squares L_1 and L_2, we say that L_2 is conjugate equivalent of L_1 where they are in the same conjugacy class [23].

Every conjugate may be identified by the permutation used to generate it. Since we have three coordinates, for each Latin square, it has at most six distinct conjugates, including the Latin square itself [16].

We require the conjugate of a Latin square to decrypt the ciphertext c produced by Latin square $L = (r_i, c_j, l_{i,j})$ using key k and plaintext m, which we can construct as follows:

- $L^{(12)} = (c_j, r_i, l_{i,j})$, where the row and column numbers of the Latin square are switched (transpose).
- $L^{(13)} = (l_{i,j}, c_j, r_i)$, where the entries and row numbers of the Latin square are switched.
- $L^{(23)} = (r_i, l_{i,j}, c_j)$, where the entries and column numbers of the Latin square are switched [22,24].

We will use a Latin square L as a substitution square for encryption, and apply $L^{(23)}$ as the substitution square for decryption in our scheme.

Latin squares are used in many modern applications, such as experimental design in statistics [25], programming language compiler testing [26], telecommunications [27], and cryptography [7,28–30].

Latin squares are important in cryptography because they can be used to construct efficient and secure cryptographic algorithms. They have been used in cryptography for various purposes [31], including as a way to generate cryptographic keys [7], and as a tool for designing symmetric cryptosystems [32]. In addition to these applications, Latin squares have also been used in other areas of cryptography, for example, in the design of cryptographic hash functions [33,34], in the analysis of cryptographic protocols [35], and in cryptographic key management systems [36].

2.2. Permutation Groups

Let Ω be an arbitrary non-empty set of non-negative integers. A permutation is a one-to-one map of Ω to itself. The set of all permutations on Ω forms a group under the composition of permutations (that is, the composition of permutations is associative, the identity permutation is the unit element, and for each permutation there exists an inverse element).

The group of all permutations on a set with n elements is called the symmetric group, which is usually denoted by S_n. Every permutation group is a subgroup of the symmetric group for some positive integer n [37]. Let $\Omega = \{1, 2, \ldots, n\}$, then a permutation ρ on Ω is defined by:

$$\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ \rho(1) & \rho(2) & \rho(3) & \cdots & \rho(n) \end{pmatrix} \tag{3}$$

Example 1. *A particular permutation of the set* $\{1, 2, 3, 4, 5\}$ *can be written as:*

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 3 & 1 \end{pmatrix}$$

This means $\rho(1) = 2$, $\rho(2) = 5$, $\rho(3) = 4$, $\rho(4) = 3$, *and* $\rho(5) = 1$.

The composition of permutations involves combining two or more permutations to create a new permutation. For example, if we have two permutations, f and g, we denote their composition as fg. To compute the composition fg, we first apply permutation g to the elements and then apply permutation f to the resulting set of elements.

Example 2. *Let* $f = (123)$, $g = (231)$. *We first apply* g *to the set of elements:* $g(1) = 2$, $g(2) = 3$, $g(3) = 1$. *Then, we apply* f *to the resulting set of elements:* $f(g(1)) = 3$, $f(g(2)) = 1$, $f(g(3)) = 2$. *Therefore,* $fg = (132)$.

The most common applications of symmetric groups in cryptography occur during the key generation phase [38]. Permutations play a vital role in this process by enhancing randomness through the shuffling of bits or bytes within the key. This ensures that each bit position in

the key is influenced by all parts of the original key. Additionally, permutations contribute to security by introducing confusion and diffusion, making it more difficult for attackers to compromise the generated keys. They are employed to mix, distribute, and derive key material in a manner that ensures cryptographic keys meet the necessary security standards.

Moreover, permutations play a crucial role in both the encryption and decryption processes as part of the multi-step approach used in symmetric encryption algorithms, such as the Advanced Encryption Standard (AES) [39]. These permutations contribute to two essential properties of cryptographic algorithms: confusion and diffusion.

Confusion ensures that the relationship between the ciphertext and the key is complex and non-linear, making it difficult for an attacker to infer the key. On the other hand, diffusion ensures that a change in a single bit of the plaintext affects many bits of the ciphertext. By rearranging the data in a controlled manner, permutations help achieve both confusion and diffusion. This sensitivity to input changes enhances the overall security of the encryption process.

Permutations are often combined with other operations, such as substitution (like the S-box in AES) and bitwise operations (such as XOR), to create a mixing effect that further strengthens the security and cryptographic robustness of the algorithm.

2.3. Error-Correcting Codes

Error-correcting codes are used in digital communication systems to detect and correct errors that may happen during data transmission. These codes introduce redundancy into the data, allowing for the detection and correction of errors even if some of the transmitted information becomes corrupted.

One crucial parameter of an error-correcting code is its minimum distance, which is defined as the smallest number of bit or symbol changes needed to convert one valid codeword into another. This minimum distance directly influences the code's ability to detect and correct errors. Specifically, a code with a minimum distance d can detect up to $(d-1)$ errors and can correct up to $\lfloor \frac{d-1}{2} \rfloor$ errors [40].

Error-correcting codes are a vital component of cryptography, as they help ensure that data transmitted between two parties remains accurate and secure [10]. In the realm of cryptography, these codes are commonly employed to protect messages or data from corruption or alteration during transmission or storage. They are essential for maintaining the reliability, integrity, and security of encrypted communications in cryptographic systems. By detecting and correcting errors, error-correcting codes help preserve the confidentiality and authenticity of encrypted data [10].

3. The Proposed Cryptographic Scheme

In this section, we present our proposed scheme, which encompasses the process of secret key generation and the method for generating security parameters. Next, we will explain the structure of the encoding/encryption and decoding/decryption algorithms used in this scheme. Lastly, we will illustrate how to generate a Latin square.

3.1. Scheme Description

Our encryption scheme operates on a message M and a key K, each of length $\alpha = \log_2 n \cdot k$ $(\alpha \geq n)$ bits, where n denotes both the order of the Latin square L and the codeword length of the Reed–Muller code, and k represents its dimension. The scheme employs a multi-round encryption process consisting of $2\log_2 n$ rounds, which seamlessly integrates multiple parallel encoding operations. Both communicating parties precompute all key-related components to enable efficient real-world deployment. The scheme utilizes two types of permutations: π_z, which act on α-bit blocks during the Latin square transfor-

mations, and ρ_z, which are n-bit permutations used in the generation of π_z and applied to the RM codewords. All indexing conventions (for permutations, rounds, and arrays) begin at zero. The process proceeds as follows:

1. **Multi-Round Latin Square Encryption**: We divide M and K into k segments, each of length $\log_2 n$ bits. Using L, we perform $2\log_2 n$ rounds. For each pair of rounds, the first uses the original row and column subsets, and the second swaps them. This process includes converting the message and the key to n-base, mapping the message with the Latin square using the generated scheduled keys, and permuting the intermediate messages using π_z permutations for each round.
2. **Portioning**: We split Y into $\log_2 n$ portions, each of length k bits.
3. **Reed-Muller Encoding**: Each portion is independently encoded using the RM code, transforming it into an n-bit codeword.
4. **Permutations and Error Addition**: Each codeword is permuted using the corresponding n-bit permutation ρ_z and modified by adding a generated error vector e_z.
5. **Final Ciphertext**: The modified codewords are concatenated to form the ciphertext C, of length $n \cdot \log_2 n$ bits.

The decryption process reverses the encryption steps by utilizing the conjugate Latin square $L^{(23)}$ together with the inverse permutations π_z^{-1} for the rounds and ρ_z^{-1} for the codewords. It employs RM decoding, which can correct up to t errors per segment. By executing $\log_2 n$ RM encodings in parallel within a single encryption cycle, the proposed scheme enhances computational efficiency while preserving interdependence through the Latin square transformations and the associated permutations. Furthermore, the scheme is flexible and can be adapted to different parameter settings of n and k.

3.2. Proposed Scheme

Let L be a Latin square of order n, and denote its row permutations by $\sigma_0, \ldots, \sigma_{n-1}$ and its column permutations by $\tau_0, \ldots, \tau_{n-1}$, respectively. In our cryptographic scheme, we employ the generator matrix of an RM code. For comprehensive details on RM codes, the reader is referred to [14,41]. Let $RM(n, k, d)$ denote an RM code of length $n = 2^m$, dimension k, and minimum distance d, which can correct up to t errors and possesses a generator matrix G.

3.2.1. Key Generation

The key generation process is formalized in (Algorithm 1), which produces the secret key and parameters for the cryptographic scheme. It relies on subroutines for key scheduling (Algorithm 2) and permutation extension (Algorithm 3). The process involves generating a random key, computing round-specific permutations, deriving key schedules, and extending permutations for transformations. The secret key consists of the generator matrix G, the binary key K, the Latin square L, and subsets $\{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1}$.

The key generation process proceeds as follows:

1. **Key Initialization**:
   - Generate a random binary key K of length $\alpha = k \cdot \log_2 n$.
   - Convert K to a sequence of k base-n digits, $K_{\text{digits}} = \{k_0, k_1, \ldots, k_{k-1}\}$, where each k_i is a $\log_2 n$-bit digit.
2. **Permutation Setup**: For $z = 0, \ldots, \log_2 n - 1$, select random subsets $I_{r_z}, I_{c_z} \subseteq \{0, \ldots, n-1\}$ to define permutations for the first $\log_2 n$ rounds. For each round $z = 0, \ldots, 2\log_2 n - 1$:
   - Compute the round index $k = \lfloor z/2 \rfloor$.
   - If z is even, compute the permutation $\rho_z = \prod_{i \in I_{r_k}} \sigma_i \prod_{j \in I_{c_k}} \tau_j$, where σ_i and τ_j are permutations derived from the row $L[i]$ and column $L[:, j]$ of the Latin square L.

- If z is odd, compute $\rho_z = \prod_{j \in I_{c_k}} \tau_j \prod_{i \in I_{r_k}} \sigma_i$, swapping the order of row and column permutations.

3. **Key Schedule Computation**: Compute the key schedule $D^{(z)}_{\text{digits}}$ for each round $z = 0, \ldots, 2\log_2 n - 1$ using Algorithm 2:
   - For $z = 0$: Set $D^{(0)}_{\text{digits}} = K_{\text{digits}}$.
   - For $z = 1$: Compute $D^{(1)}_{\text{digits}} = \{L[k_i][k_{(i+1) \mod k}] \mid i = 0, \ldots, k-1\}$.
   - For $z = 2, \ldots, 2\log_2 n - 1$: Compute $D^{(z)}_{\text{digits}} = \{L[d_i^{(z-1)}][d_i^{(z-2)}] \mid i = 0, \ldots, k-1\}$, where $d_i^{(z-1)}$ and $d_i^{(z-2)}$ are digits from the previous two rounds.
4. **Permutation Extension**: Extend each permutation ρ_z to an α-bit permutation π_z using Algorithm 3. This approach is inspired by key-dependent permutation techniques [42,43], but incorporates modifications to enhance security and adaptability:
   - For each index $i = 0, \ldots, n-1$, compute:
   $$v_i = \left(\rho_z[i] + d_{i \mod k} \cdot (i^2 + 1) + z \cdot (i+1)\right) \mod \alpha,$$
   where $d_{i \mod k}$ is the corresponding key digit from $D^{(z)}_{\text{digits}}$.
   - If $\alpha > n$, compute additional values for $i = 0, \ldots, \alpha - n - 1$:
   $$v_{n+i} = (d_{i \mod k} \cdot (z + i + 1) \cdot (i+2)) \mod \alpha.$$
   - Form tuples $(v_j, d_{j \mod k}, j)$ for $j = 0, \ldots, \alpha - 1$. Sort them by v_j, then $d_{j \mod k}$, and finally by $\left(j + \sum_{i=0}^{k-1} d_i + z\right) \mod \alpha$. Use the sorted indices j to construct the permutation.
   - Apply a circular shift to the permutation by $\left(\sum_{i=0}^{k-1} d_i + z\right) \mod \alpha$.
5. **Precomputations**: Compute the conjugate Latin square $L^{(23)} = (r_i, l_{i,j}, c_j)$.

The resulting secret key is $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, with key schedules $\{D^{(z)}_{\text{digits}}\}_{z=0,\ldots,2\log_2 n-1}$ and permutations $\{\rho_z, \pi_z\}_{z=0,\ldots,2\log_2 n-1}$.

Algorithm 1 Key and Parameters Setup Algorithm

Require: Latin square L of order n, $RM(n,k,d)$ generator matrix G, parameters n, k, $\alpha = k\log_2 n$

Ensure: Secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, key schedules $\{D^{(z)}_{\text{digits}}\}_{z=0,\ldots,2\log_2 n-1}$, permutations $\{\rho_z, \pi_z\}_{z=0,\ldots,2\log_2 n-1}$

$K \leftarrow \text{RandomBinary}(\alpha)$
$K_{\text{digits}} \leftarrow \text{ConvertToBaseN}(K, n)$
for $z = 0$ to $2\log_2 n - 1$ **do**
  $k \leftarrow \lfloor z/2 \rfloor$
  if $z < \log_2 n$ **then**
    Choose random subsets $I_{r_k}, I_{c_k} \subseteq \{0, \ldots, n-1\}$
  if $z \mod 2 = 0$ **then**
    $\rho_z \leftarrow \prod_{i \in I_{r_k}} \sigma_i \prod_{j \in I_{c_k}} \tau_j$
  else
    $\rho_z \leftarrow \prod_{j \in I_{c_k}} \tau_j \prod_{i \in I_{r_k}} \sigma_i$
  $D^{(z)}_{\text{digits}} \leftarrow \text{KeySchedule}(L, K_{\text{digits}}, z)$
  $\pi_z \leftarrow \text{ExtendPermutation}(\rho_z, D^{(z)}_{\text{digits}}, z, \alpha, k, n)$
return $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1}), \{D^{(z)}_{\text{digits}}\}_{z=0,\ldots,2\log_2 n-1}, \{\rho_z, \pi_z\}_{z=0,\ldots,2\log_2 n-1}$

Algorithm 2 Key Schedule Subroutine

procedure KEYSCHEDULE(L, K_{digits}, z)
  if $z = 0$ **then**
    return K_{digits}
  else if $z = 1$ **then**
    return $\{L[k_i][k_{(i+1) \bmod k}] \mid i = 0, \ldots, k-1\}$
  else
    $D^{(z-1)} \leftarrow \text{KEYSCHEDULE}(L, K_{\text{digits}}, z-1)$
    $D^{(z-2)} \leftarrow \text{KEYSCHEDULE}(L, K_{\text{digits}}, z-2)$
    return $\{L[d_i^{(z-1)}][d_i^{(z-2)}] \mid i = 0, \ldots, k-1\}$

Algorithm 3 Permutation Extension Subroutine

procedure EXTENDPERMUTATION($\rho_z, D^{(z)}_{\text{digits}}, z, \alpha, k, n$)
  for $i = 0$ to $n-1$ **do**
    $v_i \leftarrow (\rho_z[i] + d_{i \bmod k} \cdot (i^2+1) + z \cdot (i+1)) \mod \alpha$
  for $i = 0$ to $\alpha - n - 1$ **do**
    $v_{n+i} \leftarrow (d_{i \bmod k} \cdot (z+i+1) \cdot (i+2)) \mod \alpha$
  Sort tuples $(v_j, d_{j \bmod k}, j)$ for $j = 0, \ldots, \alpha-1$ by v_j, $d_{j \bmod k}$, $\left(j + \sum_{i=0}^{k-1} d_i + z\right) \mod \alpha$
  Initialize permutation as empty list
  for $j = 0$ to $\alpha - 1$ **do**
    Append j to permutation
  Apply circular shift by $\left(\sum_{i=0}^{k-1} d_i + z\right) \mod \alpha$
  return resulting permutation

3.2.2. Encryption

The encryption process, formalized in Algorithm 4, transforms a message M into a ciphertext C using the secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, permutations $\{\pi_z, \rho_z\}$, and key schedules $\{D^{(z)}_{\text{digits}}\}$. It relies on subroutines for multi-round mapping (Algorithm 5) and RM encoding (Algorithm 6). The process involves converting the message and key to base-n digits, applying multi-round transformations with Latin square lookups and permutations, encoding with an RM code, and adding error vectors.

1. The encryption process proceeds as follows:
   - **Message and Key Conversion**: Convert the input message M of length $\alpha = k \log_2 n$ and the key K into base-n digits, where n is the order of the Latin square L. This yields k digits for each: $M_{\text{digits}} = \{m_0, \ldots, m_{k-1}\}$ and $K_{\text{digits}} = \{k_0, \ldots, k_{k-1}\}$, with each digit in $\{0, \ldots, n-1\}$. This conversion ensures uniform digit distribution [44].
   - **Multi-Round Mapping**: Perform $2 \log_2 n$ rounds of mapping using Algorithm 5 to produce an intermediate output Y:
     - For round $z = 0$:
       - Compute $y'^{(0)}_i = L[k_i][m_i]$ for $i = 0, \ldots, k-1$.
       - Convert $\{y'^{(0)}_0, \ldots, y'^{(0)}_{k-1}\}$ to an α-bit binary vector, apply the permutation π_0 to obtain $y^{(0)}_{\text{binary}}$, and convert back to base-n digits $\{y^{(0)}_0, \ldots, y^{(0)}_{k-1}\}$.
     - For rounds $z = 1, \ldots, 2 \log_2 n - 1$:

* Compute $y_i'^{(z)} = L[d_i^{(z)}][y_i^{(z-1)}]$ for $i = 0, \ldots, k-1$, where $d_i^{(z)}$ is from $D_{\text{digits}}^{(z)}$.
* Convert $\{y_0'^{(z)}, \ldots, y_{k-1}'^{(z)}\}$ to an α-bit binary vector, and apply π_z to obtain $y_{\text{binary}}^{(z)}$.

– For $z < 2\log_2 n - 1$, convert $y_{\text{binary}}^{(z)}$ back to base-n digits $\{y_0^{(z)}, \ldots, y_{k-1}^{(z)}\}$.
– Output $Y = y_0^{(2\log_2 n-1)} \| \cdots \| y_{k-1}^{(2\log_2 n-1)}$, an α-bit binary vector.

- **Splitting the Intermediate Output**: Divide Y into $\log_2 n$ portions $\{Y_0, \ldots, Y_{\log_2 n-1}\}$, each of length k bits.
- **RM Encoding and Permutation**: For each portion Y_i, $i = 0, \ldots, \log_2 n - 1$, apply Algorithm 6:
  – Compute $c_i' = Y_i \cdot G$, where G is the generator matrix of the $RM(n, k, d)$ code, producing an n-bit vector.
  – Apply the n-bit permutation ρ_i to c_i', resulting in $c_i = (Y_i \cdot G)^{\rho_i}$.
- **Error Vector Addition**: For each $i = 0, \ldots, \log_2 n - 1$, generate an error vector $e_i \in \{0,1\}^n$ with weight at most t. Compute the encrypted portion:

$$C_i = c_i + e_i = (Y_i \cdot G)^{\rho_i} + e_i.$$

- **Ciphertext Construction**: Concatenate the encrypted portions to form the ciphertext:

$$C = C_0 \| C_1 \| \cdots \| C_{\log_2 n-1},$$

where $\text{length}(C) = n \cdot \log_2 n$.

The resulting ciphertext C has length $n \cdot \log_2 n$.

Algorithm 4 Encryption Algorithm

Require: Message M of length α, secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, permutations $\{\pi_z, \rho_z\}$, key digits $\{D_{\text{digits}}^{(z)}\}$
Ensure: Ciphertext C of length $\log_2 n \times n$

$Y \leftarrow \text{MULTIROUNDMAPPING}(M, K, L, \{D_{\text{digits}}^{(z)}\}, \{\pi_z\}, n, k, \alpha)$
$\{Y_0, \ldots, Y_{\log_2 n-1}\} \leftarrow \text{Split}(Y, \log_2 n, k)$
for $i = 0$ to $\log_2 n - 1$ **do**
    $c_i \leftarrow \text{RMENCODE}(Y_i, G, \rho_i)$
    $e_i \leftarrow \text{GenerateErrorVector}(n, t)$
    $C_i \leftarrow c_i + e_i$
$C \leftarrow C_0 \| \cdots \| C_{\log_2 n-1}$
return C

Algorithm 5 Multi-Round Mapping Subroutine

procedure MULTIROUNDMAPPING($M, K, L, \{D^{(z)}_{\text{digits}}\}, \{\pi_z\}, n, k, \alpha$)
  $M_{\text{digits}} \leftarrow \text{ConvertToBaseN}(M, n)$
  $K_{\text{digits}} \leftarrow \text{ConvertToBaseN}(K, n)$
  for $z = 0$ to $2\log_2 n - 1$ **do**
    if $z = 0$ **then**
      for $i = 0$ to $k - 1$ **do**
        $y'^{(0)}_i \leftarrow L[k_i][m_i]$
    else
      for $i = 0$ to $k - 1$ **do**
        $y'^{(z)}_i \leftarrow L[d^{(z)}_i][y^{(z-1)}_i]$
    $y'^{(z)}_{\text{binary}} \leftarrow \text{ConvertToBinary}(y'^{(z)}_0, \ldots, y'^{(z)}_{k-1}, \alpha)$
    $y^{(z)}_{\text{binary}} \leftarrow \text{ApplyPermutation}(y'^{(z)}_{\text{binary}}, \pi_z)$
    if $z \neq 2\log_2 n - 1$ **then**
      $y^{(z)}_0, \ldots, y^{(z)}_{k-1} \leftarrow \text{ConvertToBaseN}(y^{(z)}_{\text{binary}}, n)$
  return $y^{(2\log_2 n-1)}_0 \| \cdots \| y^{(2\log_2 n-1)}_{k-1}$

Algorithm 6 RM Encoding Subroutine

procedure RMENCODE(Y_i, G, ρ_i)
  $c_i \leftarrow Y_i \cdot G$
  $c_i \leftarrow \text{ApplyPermutation}(c_i, \rho_i)$
  return c_i

3.2.3. Decryption

The decryption process, formalized in Algorithm 7, recovers the original message M from the ciphertext C using the secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, inverse permutations $\{\pi_z^{-1}, \rho_z^{-1}\}$, key schedules $\{D^{(z)}_{\text{digits}}\}$, and a conjugate Latin square $L^{(23)}$.

It relies on subroutines for multi-round mapping (Algorithm 8) and RM decoding (Algorithm 9).

The process involves splitting the ciphertext, decoding RM codewords, correcting errors, and reversing the multi-round mapping to retrieve the message.

1. The decryption process proceeds as follows:
   - **Ciphertext Splitting**: Divide the ciphertext C of length $\log_2 n \cdot n$ into $\log_2 n$ portions $\{C_0, \ldots, C_{\log_2 n-1}\}$, where each C_i is an n-bit vector representing an encoded, permuted, and error-corrupted portion of the intermediate message.
   - **RM Decoding**: For each portion C_i, $i = 0, \ldots, \log_2 n - 1$, apply Algorithm 9:
     - Compute the inversely permuted vector $r_i = \text{ApplyPermutation}(C_i, \rho_i^{-1})$, where $C_i = (Y_i \cdot G)^{\rho_i} + e_i$, Y_i is a k-bit portion of the intermediate message, G is the RM generator matrix, and e_i is an error vector with weight at most t.
     - Decode r_i using the $RM(n, k, d)$ decoding algorithm to correct up to $t = \lfloor (d-1)/2 \rfloor$ errors, yielding the k-bit portion Y_i.
   - **Reconstructing Intermediate Message**: Concatenate the decoded portions to form the intermediate message:

$$Y = Y_0 \| Y_1 \| \cdots \| Y_{\log_2 n-1},$$

     where Y is an $\alpha = k \log_2 n$-bit binary vector.
   - **Inverse Multi-Round Mapping**: Apply Algorithm 8 to reverse the $2\log_2 n$ rounds of mapping:

- Initialize $y_{\text{binary}}^{(2\log_2 n)} = Y$.
- For rounds $z = 2\log_2 n - 1$ down to 0:
  - If $z \neq 2\log_2 n - 1$, convert $\{y_0^{(z+1)}, \ldots, y_{k-1}^{(z+1)}\}$ to an α-bit binary vector $y_{\text{binary}}^{(z+1)}$.
  - Apply the inverse permutation π_z^{-1} to $y_{\text{binary}}^{(z+1)}$ to obtain $y_{\text{binary}}^{\prime(z)}$.
  - Convert $y_{\text{binary}}^{\prime(z)}$ to base-n digits $\{y_0^{\prime(z)}, \ldots, y_{k-1}^{\prime(z)}\}$.
  - For $i = 0, \ldots, k-1$, compute $y_i^{(z)} = L^{(23)}[y_i^{\prime(z)}][d_i^{(z)}]$, where $L^{(23)}$ is the conjugate Latin square, and $d_i^{(z)}$ is from the key schedule $D_{\text{digits}}^{(z)}$.
  - For $z = 0$, use $d_i^{(0)} = k_i$ from K_{digits}.
- Output $M = \text{ConvertToBinary}(y_0^{(0)}, \ldots, y_{k-1}^{(0)}, \alpha)$, the original α-bit message.

The resulting message M has length $\alpha = k\log_2 n$.

Algorithm 7 Decryption Algorithm

Require: Ciphertext C of length $\log_2 n \times n$, secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$, permutations $\{\pi_z^{-1}, \rho_z^{-1}\}$, key digits $\{D_{\text{digits}}^{(z)}\}$, conjugate $L^{(23)}$

Ensure: Message M of length α

1: $\{C_0, \ldots, C_{\log_2 n-1}\} \leftarrow \text{Split}(C, \log_2 n, n)$
2: **for** $i = 0$ to $\log_2 n - 1$ **do**
3: $\quad Y_i \leftarrow \text{RMDecode}(C_i, G, \rho_i^{-1})$
4: $Y \leftarrow Y_0 \| \cdots \| Y_{\log_2 n-1}$
5: $M \leftarrow \text{InverseMultiRoundMapping}(Y, K, L^{(23)}, \{D_{\text{digits}}^{(z)}\}, \{\pi_z^{-1}\}, n, k, \alpha)$
6: **return** M

Algorithm 8 Inverse Multi-Round Mapping Subroutine

1: **procedure** InverseMultiRoundMapping($Y, K, L^{(23)}, \{D_{\text{digits}}^{(z)}\}, \{\pi_z^{-1}\}, n, k, \alpha$)
2: $\quad y_{\text{binary}}^{(2\log_2 n)} \leftarrow Y$
3: $\quad$ **for** $z = 2\log_2 n - 1$ **downto** 0 **do**
4: $\quad\quad$ **if** $z \neq 2\log_2 n - 1$ **then**
5: $\quad\quad\quad y_{\text{binary}}^{(z+1)} \leftarrow \text{ConvertToBinary}(y_0^{(z+1)}, \ldots, y_{k-1}^{(z+1)}, \alpha)$
6: $\quad\quad y_{\text{binary}}^{\prime(z)} \leftarrow \text{ApplyPermutation}(y_{\text{binary}}^{(z+1)}, \pi_z^{-1})$
7: $\quad\quad y_0^{\prime(z)}, \ldots, y_{k-1}^{\prime(z)} \leftarrow \text{ConvertToBaseN}(y_{\text{binary}}^{\prime(z)}, n)$
8: $\quad\quad$ **for** $i = 0$ to $k - 1$ **do**
9: $\quad\quad\quad y_i^{(z)} \leftarrow L^{(23)}[y_i^{\prime(z)}][d_i^{(z)}]$
10: $\quad$ **return** $\text{ConvertToBinary}(y_0^{(0)}, \ldots, y_{k-1}^{(0)}, \alpha)$

Algorithm 9 RM Decoding Subroutine

1: **procedure** RMDecode(C_i, G, ρ_i^{-1})
2: $\quad r_i \leftarrow \text{ApplyPermutation}(C_i, \rho_i^{-1})$
3: $\quad Y_i \leftarrow \text{RMDecode}(r_i, G)$
4: $\quad$ **return** Y_i

3.2.4. Generation of Latin Squares

There are numerous ways to generate Latin squares in the literature which aim to rapidly generate in an efficient and clear way the entire set of Latin squares of order n or a proper set of Latin squares of order n for practical use [31]. These methods rely on several

concepts such as simple product of quasigroups, generation using linear mapping, and generation using keyed permutation, etc. [45]. Since we want to generate a single Latin square for both sides, we will use a simple method to achieve that by means of a non-linear mapping [45].

Both parties to the connection produce two random permutations, f and g, and each permutation is then stored in a one-dimensional array with a size equal to the permutation. To generate the (i, j)-th element of the Latin square, add the i-th element of the first permutation to the j-th element of the second permutation. Then, apply the modulus operation with respect to the size of the Latin square to be generated to the addition of the i-th element of the first permutation and the j-th element of the second permutation.

Example 3. *Let $A = \mathbb{Z}_n = \{0, 1, \ldots, n-1\}$ and let the group operation be addition modulo n. Then, we can create a quasigroup $(Q, +)$ from $(\mathbb{Z}_n, +)$ by supposing $f(x)$ and $g(x)$ (Table 2) and defining $h(x, y) = (f(x) + g(y)) \mod n$.*

To generate a Latin square of order 4, first, we will generate two random permutations f and g and then apply $h(x, y) = (f(x) + g(y)) \mod 4$ for x and y ranging from 0 to 3.

Table 2. Defining two permutations f and g.

x	0	1	2	3
f(x)	1	3	0	2
g(x)	2	3	1	0

Then, the quasigroup $(Q, +)$ created from f and g by supposing $h(x, y) = (f(x) + g(y)) \mod n$ is shown in Table 3.

Table 3. The quasigroup $(Q, +)$ created from two permutations f and g.

+	0	1	2	3
0	3	0	2	1
1	1	2	0	3
2	2	3	1	0
3	0	1	3	2

The main advantage of using this method is that $(Q, +)$ is not associative because $h(0, h(2, 3)) = h(0, 0) = 3$ and $h(h(0, 2), 3) = h(2, 3) = 0$, and not commutative because $h(2, 3) = 0$ and $h(3, 2) = 3$. Because there are no inverses or identity elements, the rows are independent, meaning that no one can guess the entire Latin square by figuring out any one row [45].

4. Experimental Results

To validate the proposed symmetric cryptographic scheme, we conducted a numeric experiment using a Latin square of order $n = 8$ and the Reed-Muller code $RM(8, 4, 4)$, with $\log_2 n = 3$, resulting in $2 \log_2 n = 6$ rounds. The goal is to demonstrate secure encryption using $2 \log_2 n$ rounds of Latin square transformations with row-column swapping in the additional $\log_2 n$ rounds and α-bit permutations π_z, and reliable message recovery leveraging the error-correcting properties of $RM(8, 4, 4)$ despite transmission errors. This section details the environment, setup, processes, and results.

4.1. Experimental Environment

The experiment was performed on:

- System Model: HP ProBook 450 15.6-inch G10 Notebook PC (HP Inc., Palo Alto, CA, USA);
- Processor: 13th Gen Intel(R) Core(TM) i5-1335U (12 CPUs), up to 1.3 GHz;
- Memory: 16,384 MB RAM (16 GB).

Execution times were measured on this hardware, implemented in Python 3.12, and reflect performance under these conditions. Variations in system specifications may affect timings.

The experiment used a `CipherSystem` class in Python, handling Latin square mappings, permutation generation, Reed-Muller encoding/decoding, and error correction. Key functions include `multi_round_mapping`, `inverse_multi_round_mapping`, `rm_encode`, and `rm_decode`. Source code is available in the Supplementary Material.

4.2. Experimental Setup

The experiment uses:

- Parameters: $n = 8$, $\log_2 8 = 3$, $k = 4$, $\alpha = 12$ bits.
- Latin Square L:

$$L = \begin{bmatrix} 2 & 4 & 7 & 0 & 6 & 1 & 5 & 3 \\ 3 & 2 & 4 & 7 & 0 & 6 & 1 & 5 \\ 5 & 3 & 2 & 4 & 7 & 0 & 6 & 1 \\ 1 & 5 & 3 & 2 & 4 & 7 & 0 & 6 \\ 6 & 1 & 5 & 3 & 2 & 4 & 7 & 0 \\ 0 & 6 & 1 & 5 & 3 & 2 & 4 & 7 \\ 7 & 0 & 6 & 1 & 5 & 3 & 2 & 4 \\ 4 & 7 & 0 & 6 & 1 & 5 & 3 & 2 \end{bmatrix}$$

- Generator Matrix G: For $RM(8, 4, 4)$:

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

- Message: $M = 101100110010$ ($[5, 4, 6, 2]$).
- Key: $K = 011010110000$ ($[3, 2, 6, 0]$).
- Subsets: For $z = 0, 1, 2$:
  - $I_{r_0} = \{0, 1, 3, 4, 7\}$, $I_{c_0} = \{2, 3, 5\}$
  - $I_{r_1} = \{1, 2, 5, 6, 7\}$, $I_{c_1} = \{0, 2, 4, 6\}$
  - $I_{r_2} = \{2, 3, 4\}$, $I_{c_2} = \{1, 5\}$
- Error Vectors:

$$e_0 = [0, 0, 0, 0, 1, 0, 0, 0], \quad e_1 = [0, 0, 0, 0, 0, 0, 0, 1], \quad e_2 = [0, 0, 0, 1, 0, 0, 0, 0]$$

4.3. Key Generation

- Select a 12-bit key $K = 011010110000$.
- Compute 8-bit permutations ρ_z, where for even $z = 0, 2, 4$, $\rho_z = \prod_{i \in I_{r_k}} \sigma_i \prod_{j \in I_{c_k}} \tau_j$, and for odd $z = 1, 3, 5$, $\rho_z = \prod_{j \in I_{c_k}} \tau_j \prod_{i \in I_{r_k}} \sigma_i$, with $k = \lfloor z/2 \rfloor$, and σ_i, τ_j are row and column permutations of L:
  - $\rho_0 = [6, 2, 5, 0, 4, 1, 3, 7]$ (from $I_{r_0} = \{0, 1, 3, 4, 7\}$, $I_{c_0} = \{2, 3, 5\}$)
  - $\rho_1 = [7, 6, 5, 0, 4, 1, 3, 2]$ (from swapped I_{c_0}, I_{r_0})
  - $\rho_2 = [0, 4, 6, 5, 7, 3, 1, 2]$ (from $I_{r_1} = \{1, 2, 5, 6, 7\}$, $I_{c_1} = \{0, 2, 4, 6\}$)

- $\rho_3 = [4,2,1,5,6,0,7,3]$ (from swapped I_{c_1}, I_{r_1})
- $\rho_4 = [4,0,5,3,2,1,7,6]$ (from $I_{r_2} = \{2,3,4\}, I_{c_2} = \{1,5\}$)
- $\rho_5 = [2,6,5,7,4,1,3,0]$ (from swapped I_{c_2}, I_{r_2})

- Extend ρ_z to 12-bit round permutations π_z, using a key-dependent transformation with non-linear operations:
  - $\pi_0 = [0,2,3,4,5,6,7,11,1,8,9,10]$
  - $\pi_1 = [0,1,3,5,8,9,10,2,4,6,7,11]$
  - $\pi_2 = [0,2,4,6,9,10,11,1,5,3,7,8]$
  - $\pi_3 = [0,1,4,5,6,7,8,10,11,2,3,9]$
  - $\pi_4 = [2,6,7,8,10,11,5,0,1,3,4,9]$
  - $\pi_5 = [0,3,7,9,11,2,5,6,1,4,8,10]$
- Compute key schedule:
  - Round 0: $D^{(0)}_{\text{digits}} = K_{\text{digits}} = [3,2,6,0]$.
  - Round 1: $D^{(1)}_{\text{digits}} = [L[k_i][k_{(i+1) \mod 4}] \mid i = 0,\ldots,3] = [3,6,7,0]$.
  - Round 2: $D^{(2)}_{\text{digits}} = [L[d_i^{(1)}][d_i^{(0)}] \mid i = 0,\ldots,3] = [2,6,3,2]$.
  - Round 3: $D^{(3)}_{\text{digits}} = [L[d_i^{(2)}][d_i^{(1)}] \mid i = 0,\ldots,3] = [4,2,6,5]$.
  - Round 4: $D^{(4)}_{\text{digits}} = [L[d_i^{(3)}][d_i^{(2)}] \mid i = 0,\ldots,3] = [5,6,1,1]$.
  - Round 5: $D^{(5)}_{\text{digits}} = [L[d_i^{(4)}][d_i^{(3)}] \mid i = 0,\ldots,3] = [3,6,1,6]$.

  Secret key: $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,1,2})$.

4.4. Encryption Process

For $M = 101100110010$:

1. **Base-8 Conversion**:
$$M = [5,4,6,2], \quad K = [3,2,6,0]$$
2. **Round 0 Encryption**:
   - $D^{(0)}_{\text{digits}} = [3,2,6,0]$.
   - $y_i'^{(0)} = L[k_i][m_i]$:
$$y_0'^{(0)} = L[3][5] = 7, \quad y_1'^{(0)} = L[2][4] = 7, \quad y_2'^{(0)} = L[6][6] = 2, \quad y_3'^{(0)} = L[0][2] = 7$$
   - $y'^{(0)} = [7,7,2,7] \rightarrow 111111010111$.
   - Apply $\pi_0 = [0,2,3,4,5,6,7,11,1,8,9,10]$: 111110111011.
   - Base-8: $y^{(0)} = [7,6,7,3]$.
3. **Round 1 Encryption**:
   - $D^{(1)}_{\text{digits}} = [3,6,7,0]$.
   - $y_i'^{(1)} = L[d_i^{(1)}][y_i^{(0)}]$:
$$y_0'^{(1)} = L[3][7] = 6, \quad y_1'^{(1)} = L[6][6] = 2, \quad y_2'^{(1)} = L[7][7] = 2, \quad y_3'^{(1)} = L[0][3] = 0$$
   - $y'^{(1)} = [6,2,2,0] \rightarrow 110010010000$.
   - Apply $\pi_1 = [0,1,3,5,8,9,10,2,4,6,7,11]$: 110000001010.
   - Base-8: $y^{(1)} = [6,0,1,2]$.
4. **Round 2 Encryption**:
   - $D^{(2)}_{\text{digits}} = [2,6,3,2]$.
   - $y_i'^{(2)} = L[d_i^{(2)}][y_i^{(1)}]$:

$$y_0'^{(2)} = L[2][6] = 6, \quad y_1'^{(2)} = L[6][0] = 7, \quad y_2'^{(2)} = L[3][1] = 5, \quad y_3'^{(2)} = L[2][2] = 2$$

- $y'^{(2)} = [6, 7, 5, 2] \rightarrow 110111101010$.
- Apply $\pi_2 = [0, 2, 4, 6, 9, 10, 11, 1, 5, 3, 7, 8]$: 101101101011.
- Base-8: $y^{(2)} = [5, 5, 3, 5]$.

5. **Round 3 Encryption**:
   - $D_{\text{digits}}^{(3)} = [4, 2, 6, 5]$.
   - $y_i'^{(3)} = L[d_i^{(3)}][y_i^{(2)}]$:

$$y_0'^{(3)} = L[4][5] = 4, \quad y_1'^{(3)} = L[2][5] = 0, \quad y_2'^{(3)} = L[6][3] = 1, \quad y_3'^{(3)} = L[5][5] = 2$$

   - $y'^{(3)} = [4, 0, 1, 2] \rightarrow 100000001010$.
   - Apply $\pi_3 = [0, 1, 4, 5, 6, 7, 8, 10, 11, 2, 3, 9]$: 100000110000.
   - Base-8: $y^{(3)} = [4, 0, 6, 0]$.
6. **Round 4 Encryption**:
   - $D_{\text{digits}}^{(4)} = [5, 6, 1, 1]$.
   - $y_i'^{(4)} = L[d_i^{(4)}][y_i^{(3)}]$:

$$y_0'^{(4)} = L[5][4] = 3, \quad y_1'^{(4)} = L[6][0] = 7, \quad y_2'^{(4)} = L[1][6] = 1, \quad y_3'^{(4)} = L[1][0] = 3$$

   - $y'^{(4)} = [3, 7, 1, 3] \rightarrow 011111001011$.
   - Apply $\pi_4 = [2, 6, 7, 8, 10, 11, 5, 0, 1, 3, 4, 9]$: 100111110110.
   - Base-8: $y^{(4)} = [4, 7, 5, 6]$.
7. **Round 5 Encryption**:
   - $D_{\text{digits}}^{(5)} = [3, 6, 1, 6]$.
   - $y_i'^{(5)} = L[d_i^{(5)}][y_i^{(4)}]$:

$$y_0'^{(5)} = L[3][4] = 4, \quad y_1'^{(5)} = L[6][7] = 4, \quad y_2'^{(5)} = L[1][5] = 6, \quad y_3'^{(5)} = L[6][6] = 2$$

   - $y'^{(5)} = [4, 4, 6, 2] \rightarrow 100100110010$.
   - Apply $\pi_5 = [0, 3, 7, 9, 11, 2, 5, 6, 1, 4, 8, 10]$: 111000010001.
   - Base-8: $y^{(5)} = [7, 0, 2, 1]$.
8. **Divide into Portions**: $Y = 111000010001 \rightarrow Y_0 = 1110, Y_1 = 0001, Y_2 = 0001$.
9. **Encode with** G:

$$c_0 = [1, 1, 0, 0, 0, 0, 1, 1], \quad c_1 = [0, 1, 0, 1, 0, 1, 0, 1], \quad c_2 = [0, 1, 0, 1, 0, 1, 0, 1]$$

10. **Apply Permutations**:

$$c_0^{\rho_0} = [1, 0, 0, 1, 0, 1, 0, 1], \quad c_1^{\rho_1} = [1, 0, 1, 0, 0, 1, 1, 0], \quad c_2^{\rho_2} = [0, 0, 0, 1, 1, 1, 1, 0]$$

11. **Add Errors**:

$$C_0 = [1, 0, 0, 1, 1, 1, 0, 1], \quad C_1 = [1, 0, 1, 0, 0, 1, 1, 1], \quad C_2 = [0, 0, 0, 0, 1, 1, 1, 0]$$

12. **Ciphertext**: $C = 100111011010011100001110$.

4.5. Decryption Process

For $C = 100111011010011100001110$:

1. **Reconstruct Permutations**: Using the secret key $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,1,2})$, compute the 12-bit round permutations π_z, their inverses π_z^{-1}, and the 8-bit codeword permutations ρ_z, their inverses ρ_z^{-1}:
   - $\rho_0 = [6,2,5,0,4,1,3,7]$, $\rho_0^{-1} = [3,5,1,6,4,2,0,7]$
   - $\rho_1 = [7,6,5,0,4,1,3,2]$, $\rho_1^{-1} = [3,5,7,6,4,2,1,0]$
   - $\rho_2 = [0,4,6,5,7,3,1,2]$, $\rho_2^{-1} = [0,6,7,5,1,3,2,4]$
   - $\rho_3 = [4,2,1,5,6,0,7,3]$, $\rho_3^{-1} = [5,2,1,7,0,3,4,6]$
   - $\rho_4 = [4,0,5,3,2,1,7,6]$, $\rho_4^{-1} = [1,5,4,3,0,2,7,6]$
   - $\rho_5 = [2,6,5,7,4,1,3,0]$, $\rho_5^{-1} = [7,5,0,6,4,2,1,3]$
   - $\pi_0 = [0,2,3,4,5,6,7,11,1,8,9,10]$, $\pi_0^{-1} = [0,8,1,2,3,4,5,6,9,10,11,7]$
   - $\pi_1 = [0,1,3,5,8,9,10,2,4,6,7,11]$, $\pi_1^{-1} = [0,1,7,2,8,3,9,10,4,5,6,11]$
   - $\pi_2 = [0,2,4,6,9,10,11,1,5,3,7,8]$, $\pi_2^{-1} = [0,7,1,9,2,8,3,10,11,4,5,6]$
   - $\pi_3 = [0,1,4,5,6,7,8,10,11,2,3,9]$, $\pi_3^{-1} = [0,1,9,10,2,3,4,5,6,11,7,8]$
   - $\pi_4 = [2,6,7,8,10,11,5,0,1,3,4,9]$, $\pi_4^{-1} = [7,8,0,9,10,6,1,2,3,11,4,5]$
   - $\pi_5 = [0,3,7,9,11,2,5,6,1,4,8,10]$, $\pi_5^{-1} = [0,8,5,1,9,6,7,2,10,3,11,4]$
2. **Split Ciphertext**: Divide into C_0, C_1, C_2, each 8 bits:

$$C_0 = [1,0,0,1,1,1,0,1], \quad C_1 = [1,0,1,0,0,1,1,1], \quad C_2 = [0,0,0,0,1,1,1,0]$$

3. **Apply Inverse Permutations and Decode**:
   - $r_0 = C_0^{\rho_0^{-1}} = [1,1,0,0,1,0,1,1] \rightarrow Y_0 = [1,1,1,0]$.
   - $r_1 = C_1^{\rho_1^{-1}} = [0,1,1,1,0,1,0,1] \rightarrow Y_1 = [0,0,0,1]$.
   - $r_2 = C_2^{\rho_2^{-1}} = [0,1,0,1,0,0,0,1] \rightarrow Y_2 = [0,0,0,1]$.
4. **Reconstruct** Y: $Y = 111000010001$.
5. **Round 5 Decryption**:
   - Apply π_5^{-1}: $111000010001 \rightarrow 100100110010 \rightarrow [4,4,6,2]$.
   - Inverse mapping: $y_i^{(5)} = L^{(23)}[y_i'^{(5)}][d_i^{(5)}]$:

$$y_0^{(5)} = L^{(23)}[4][3] = 4, \quad y_1^{(5)} = L^{(23)}[4][6] = 7$$

$$y_2^{(5)} = L^{(23)}[6][1] = 5, \quad y_3^{(5)} = L^{(23)}[2][6] = 6$$

   - $y^{(5)} = [4,7,5,6]$.
6. **Round 4 Decryption**:
   - $y_{\text{binary}}^{(5)} = 100111110110$.
   - Apply π_4^{-1}: $011111001011 \rightarrow [3,7,1,3]$.
   - Inverse mapping: $y_i^{(4)} = L^{(23)}[y_i'^{(4)}][d_i^{(4)}]$:

$$y_0^{(4)} = L^{(23)}[3][5] = 4, \quad y_1^{(4)} = L^{(23)}[7][6] = 0$$

$$y_2^{(4)} = L^{(23)}[1][1] = 6, \quad y_3^{(4)} = L^{(23)}[3][1] = 0$$

   - $y^{(4)} = [4,0,6,0]$.
7. **Round 3 Decryption**:
   - $y_{\text{binary}}^{(4)} = 100000110000$.

- Apply π_3^{-1}: $100000001010 \rightarrow [4, 0, 1, 2]$.
- Inverse mapping: $y_i^{(3)} = L^{(23)}[y_i'^{(3)}][d_i^{(3)}]$:

$$y_0^{(3)} = L^{(23)}[4][4] = 5, \quad y_1^{(3)} = L^{(23)}[0][2] = 5$$

$$y_2^{(3)} = L^{(23)}[1][6] = 3, \quad y_3^{(3)} = L^{(23)}[2][5] = 5$$

- $y^{(3)} = [5, 5, 3, 5]$.

8. **Round 2 Decryption**:
   - $y_{\text{binary}}^{(3)} = 101101101011$.
   - Apply π_2^{-1}: $110111101010 \rightarrow [6, 7, 5, 2]$.
   - Inverse mapping: $y_i^{(2)} = L^{(23)}[y_i'^{(2)}][d_i^{(2)}]$:

$$y_0^{(2)} = L^{(23)}[6][2] = 6, \quad y_1^{(2)} = L^{(23)}[7][6] = 0$$

$$y_2^{(2)} = L^{(23)}[5][3] = 1, \quad y_3^{(2)} = L^{(23)}[2][2] = 2$$

   - $y^{(2)} = [6, 0, 1, 2]$.
9. **Round 1 Decryption**:
   - $y_{\text{binary}}^{(2)} = 110000001010$.
   - Apply π_1^{-1}: $110010010000 \rightarrow [6, 2, 2, 0]$.
   - Inverse mapping: $y_i^{(1)} = L^{(23)}[y_i'^{(1)}][d_i^{(1)}]$:

$$y_0^{(1)} = L^{(23)}[6][3] = 7, \quad y_1^{(1)} = L^{(23)}[2][6] = 6$$

$$y_2^{(1)} = L^{(23)}[2][7] = 7, \quad y_3^{(1)} = L^{(23)}[0][0] = 3$$

   - $y^{(1)} = [7, 6, 7, 3]$.
10. **Round 0 Decryption**:
    - $y_{\text{binary}}^{(1)} = 111110111011$.
    - Apply π_0^{-1}: $111111010111 \rightarrow [7, 7, 2, 7]$.
    - Inverse mapping: $y_i^{(0)} = L^{(23)}[y_i'^{(0)}][d_i^{(0)}]$:

$$y_0^{(0)} = L^{(23)}[7][3] = 5, \quad y_1^{(0)} = L^{(23)}[7][2] = 4$$

$$y_2^{(0)} = L^{(23)}[2][6] = 6, \quad y_3^{(0)} = L^{(23)}[7][0] = 2$$

    - $y^{(0)} = [5, 4, 6, 2] \rightarrow 101100110010$.

4.6. Results

- Original Message: $M = 101100110010$;
- Ciphertext: $C = 100111011010011100001110$;
- Decrypted Message: 101100110010, and it matches the original;
- Outcome: Successfully encrypted and decrypted, correcting single-bit errors using $RM(8, 4, 4)$.

4.7. Avalanche Effect and Differential Analysis

To evaluate the diffusion properties and resistance against differential cryptanalysis for practical parameter sets, we conducted avalanche effect and differential pattern consistency tests. These tests align with configurations used in the performance analysis:

- $RM(16, 15, 2)$: $n = 16, k = 15, d = 2$ (Input: 60-bit, Output: 64-bit)
- $RM(32, 16, 8)$: $n = 32, k = 16, d = 8$ (Input: 80-bit, Output: 160-bit)
- $RM(32, 26, 4)$: $n = 32, k = 26, d = 4$ (Input: 130-bit, Output: 160-bit)

For each configuration, 100 random messages were used, and every possible single input bit flip was performed using our implementation.

4.7.1. Avalanche Effect Analysis

The avalanche effect measures how a single input bit flip impacts the output bits. Ideally, approximately 50% of the output bits should change. We measured the average Hamming Distance (HD) between the original and modified ciphertexts for each input bit flip, as well as the distribution of these changes across the $\log_2 n$ output portions corresponding to the RM codewords. The results are summarized in Table 4.

The results demonstrate consistently strong diffusion properties across all tested parameter sets incorporating different $RM(n, k, d)$ codes.

- **Overall Avalanche:** The average change across all input bits is remarkably close to the ideal 50% for all configurations ($RM(16, 15, 2)$, $RM(32, 16, 8)$, and $RM(32, 26, 4)$). This indicates excellent overall diffusion regardless of the specific code parameters.
- **Consistency:** The range between the minimum and maximum average avalanche percentages observed for individual input bit flips remains reasonably tight across all sets, suggesting good uniformity in diffusion behavior across different input bit positions.
- **Distribution:** The analysis of flip distribution across the $\log_2 n$ output sections shows near-perfect uniformity for all configurations. The average percentage of flips landing in each section consistently matches the ideal ($100/\log_2 n$%), indicating that the diffusion mechanism effectively spreads changes evenly throughout the entire ciphertext block, irrespective of the specific $RM(n, k, d)$ used.
- **SAC:** The Strict Avalanche Criterion is met in 14–24% of individual flips. While not approaching 100%, the excellent average avalanche and uniform distribution are more indicative of strong practical diffusion.

Table 4. Avalanche Effect Summary for Different Parameter Sets (Avg. over 100 messages).

Metric	$\boldsymbol{RM(16, 15, 2)}$ **(In: 60b, Out: 64b)**	$\boldsymbol{RM(32, 16, 8)}$ **(In: 80b, Out: 160b)**	$\boldsymbol{RM(32, 26, 4)}$ **(In: 130b, Out: 160b)**
Overall Avg. Avalanche *(Ideal: 50%)*	49.75% *(Avg. HD: 31.84)*	50.00% *(Avg. HD: 80.00)*	50.06% *(Avg. HD: 80.10)*
Min Avg. Avalanche *(per input bit)*	42.81% *(Bit 21)*	45.75% *(Bit 71)*	47.13% *(Bit 67)*
Max Avg. Avalanche *(per input bit)*	53.44% *(Bit 0, 36)*	52.75% *(Bit 31, 36)*	54.12% *(Bit 27)*
SAC Met (%) *	21.17% (127/600 flips)	23.62% (189/800 flips)	14.15% (184/1300 flips)
Output Sections ($\log_2 n$) Ideal Flip Distr. (%) Observed Avg. Distr. (%)	4 25.0% [24.9, 25.2, 24.8, 25.1]	5 20.0% [20.1, 20.0, 20.0, 19.8, 20.0]	5 20.0% [19.9, 20.1, 20.1, 20.0, 19.9]

* SAC: Strict Avalanche Criterion (% of flips with exactly 50% output change).

4.7.2. Differential Pattern Consistency Analysis

We tested for consistent input/output differential characteristics by flipping each input bit for 100 random messages and observing the resulting output XOR patterns (ΔC).

For all three parameter sets tested ($RM(16, 15, 2)$; $RM(32, 16, 8)$; and $RM(32, 26, 4)$), the results were optimal:

Found common differential patterns for 0 / α input bit flips across 100 messages.

This indicates that for every single input bit flip tested, the specific pattern of output bit changes varied depending on the message being encrypted. No consistent input/output differential characteristic was detected in these empirical tests for any configuration. This provides strong evidence suggesting resistance against simple first-order differential attacks across different scales and configurations of the cipher.

4.7.3. Implications

These comprehensive results, obtained using practical parameters and NumPy optimizations, significantly strengthen the security claims. The cipher consistently exhibits excellent average diffusion, near-perfect distribution of changes across the output block, and empirical resistance to consistent differential patterns across different $RM(n, k, d)$ configurations. This suggests the core design principles—including $2 \log_2 n$ SPN rounds with row/column swapping, key-dependent permutations, and the overlapping structure—effectively provide robust confusion and diffusion properties that scale well with the chosen parameters.

4.8. Performance and Comparison

To quantify the computational overhead and scalability of our scheme, we conducted a comprehensive performance analysis. This analysis addresses the reviewer's request by benchmarking our scheme against its own "SPN-only" baseline (with RM codes removed), the Advanced Encryption Standard (AES-128), and the lightweight cipher PRESENT-128.

We benchmarked two parameter sets for our scheme on the hardware specified in Section 4.1. The results, averaged over 100 runs, are presented in Table 5. We measured three metrics for our scheme:

- **SPN-Only (Baseline):** The core SPN operation, representing the performance without the error-correction layer.
- **Full Scheme (Single Node):** The total time for the entire serial process, including the SPN and all RM encoding/decoding operations.
- **Full Scheme (Parallel Est.):** The theoretical time in a distributed environment, calculated as the SPN time plus the time of the slowest parallel RM-encoding/decoding node observed during the runs.

Overhead of RM Codes: The "SPN-Only" baseline clearly quantifies the cost of the resilience feature. For the 130-bit scheme, adding RM codes introduces a substantial 9.5× overhead for encryption (1.383 ms vs 0.146 ms) and a very significant 69.2× overhead for decryption (10.859 ms vs 0.157 ms). This stark difference underscores that the RM decoding, likely the majority-logic algorithm used, is the primary performance bottleneck, representing the explicit trade-off for error correction and fault-attack resilience absent in AES and PRESENT.

Comparison to PRESENT: The SPN baseline demonstrates excellent performance. Both the 60-bit (0.094 ms) and 130-bit (0.146 ms) baseline encryptions are faster than PRESENT encryption (0.233 ms), showcasing the efficiency of the core SPN structure when implemented with NumPy.

Parallelization Advantage: The "Parallel Est." results confirm the critical importance of the distributed design. For the 130-bit scheme, parallelization dramatically reduces the decryption time from 10.859 ms to 2.412 ms, achieving a 77.8% speedup. For the 60-bit scheme, the decryption speedup is 70.2% (from 2.048 ms to 0.610 ms). Even encryption benefits significantly, with the 130-bit parallel estimate (0.409 ms) being 70.4% faster than the single-node full scheme. This proves that the parallel architecture effectively mitigates the RM processing overhead, particularly for decryption, making the scheme viable for multi-core or distributed systems.

Comparison to AES: AES remains orders of magnitude faster due to optimized libraries and hardware support, serving as a performance ceiling. Our scheme focuses on resilience, a feature AES lacks natively.

Table 5. Comprehensive Performance Comparison (Average time in ms).

Cipher/Scheme	Block Size/Key Size (bits)	Encryption (ms)	Decryption (ms)
Proposed $RM(32, 26, 4)$ *	130/130		
SPN-Only (Baseline)		0.146	0.157
Full Scheme (Single Node)		1.383	10.859
Full Scheme (Parallel Est.)		0.409	2.412
Proposed $RM(16, 15, 2)$ *	60/60		
SPN-Only (Baseline)		0.094	0.105
Full Scheme (Single Node)		0.432	2.048
Full Scheme (Parallel Est.)		0.177	0.610
Standard Ciphers			
PRESENT-128	64/128	0.233	0.224
AES-128 (ECB)	128/128	0.008	0.002
AES-128 (CBC)	128/128	0.003	0.002
AES-128 (CTR)	128/128	0.003	0.002

* Benchmarks for the proposed scheme are from a Python prototype using NumPy 2.3. Standard cipher results may reflect library optimizations or hardware acceleration.

5. Security Analysis

The proposed cryptographic scheme achieves robust security by integrating Latin square transformations, permutations, and Reed-Muller (RM) codes. This section analyzes the scheme's security primitives using general parameters: Latin square order $n = 2^m$, RM code length n, dimension k, minimum distance d, and error-correcting capability $t = \lfloor (d-1)/2 \rfloor$. The total message/key length is $\alpha = k \cdot \log_2 n$. We evaluate the key space, resistance to common cryptanalytic attacks, and the impact of parameter variation, supported by the empirical results presented in Section 4.

5.1. Key Space and Components

The security of the proposed cryptographic scheme relies on the size and complexity of its secret key, which directly influences the final ciphertext C. The secret key consists of the binary key K, the Latin square L, the generator matrix G of the Reed-Muller code $RM(r, m)$, and distinct subsets $\{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1}$ used to generate permutations ρ_z. The ciphertext C is produced through multi-round Latin square transformations over $2\log_2 n$ rounds with row-column swapping, π_z permutations derived from ρ_z and the key schedule, RM encoding with G, application of ρ_z, and potential addition of e_z. Each key component contributes to the key space, making brute-force attacks computationally infeasible. Below, we analyze each component's contribution to the key space.

- **Binary Key** K: The key K is a binary string of length $\alpha = k \cdot \log_2 n$, where $n = 2^m$, $k = \sum_{i=0}^{r} \binom{m}{i}$, and $m = \log_2 n$. The key space for K is 2^α. For the tested parameters:

$RM(16, 15, 2)$ gives $\alpha = 15 \cdot 4 = 60$, $RM(32, 16, 8)$ gives $\alpha = 16 \cdot 5 = 80$, and $RM(32, 26, 4)$ gives $\alpha = 26 \cdot 5 = 130$. The key space ranges from 2^{60} to 2^{130} for these examples. K is converted to base-n digits, used in the first round and to derive the key schedule. The key schedule ($D^{(z)}_{\text{digits}}$) depends deterministically on K and L, ensuring non-linear transformations across rounds and enhancing K's diffusion.

- **Latin Square** L: The number of Latin squares of order n, denoted $L(n)$, grows factorially with n. For large values of n, the exact number of Latin squares $L(n)$ is not known, but there are bounds on the number of Latin squares $L(n)$. One such bound is: $L(n) \geq (n!)^{2n}/n^{n^2}$ [46]. This bound implies that the number of Latin squares of order n grows very rapidly as n increases. The number of Latin squares is known for $n \leq 11$ [47]. This lower bound assures that against the development of hardware, n can always be chosen in a way such that it is computationally infeasible to find the Latin square L. For $n = 16$, $\log_2 L(16) \approx 379$ bits, or $L(16) \approx 2^{379}$. For $n = 32$, $\log_2 L(32) \approx 2407$ bits, or $L(32) \approx 2^{2407}$. The Latin square L is used for substitution in each of the $2\log_2 n$ rounds, mapping message digits to new values based on key digits, and its conjugate $L^{(23)}$ is used for decryption. The vast number of possible Latin squares ensures that guessing L is impractical, even if an attacker knows the scheme's structure. Since L directly affects the intermediate value Y and, through the key schedule, the permutations π_z, it significantly contributes to the randomness of C.

- **Generator Matrix** G: The RM code $RM(r, m)$ has a generator matrix G of size $k \times n$, where $n = 2^m$, $k = \sum_{i=0}^{r} \binom{m}{i}$, and the minimum distance $d = 2^{m-r}$. For $n = 8$, $m = 3$, and $r = 1$, $k = 4$, the specific G used in the experiment is fixed. In general, the choice of G for $RM(r, m)$ can vary by selecting different bases for the code's k-dimensional vector space. The number of distinct generator matrices is the number of ordered bases, which is:

$$\prod_{i=0}^{k-1} (2^k - 2^i)$$

For $n = 16$, $m = 4$, $r = 1$, $k = 5$, this is approximately 2^{25}, and for $n = 32$, $m = 5$, $r = 1$, $k = 6$, approximately 2^{35}. We assume a fixed G, but including it as a variable key component would increase the key space by these factors. G affects C by encoding portions of Y into codewords, ensuring error correction and adding a security layer [14,48].

- **Subsets for Permutations**: For each of the first $\log_2 n$ rounds, a distinct pair of subsets $(I_{r_z}, I_{c_z}) \subseteq \{0, \ldots, n-1\}$ is chosen to generate the permutation ρ_z for rounds $z = 0, 2, \ldots, 2\log_2 n - 2$, and swapped for rounds $z = 1, 3, \ldots, 2\log_2 n - 1$, which is extended to the α-bit permutation π_z using key-dependent transformations. The number of possible subsets for each I_{r_z} and I_{c_z} is 2^n, so each pair contributes $2^n \times 2^n = 2^{2n}$ possibilities. With $\log_2 n$ subset pairs (used across $2\log_2 n$ rounds with swapping), the total contribution from subsets is $(2^{2n})^{\log_2 n} = 2^{2n \log_2 n}$. For $n = 16$, $\log_2 16 = 4$, so $2^{2\times 16\times 4} = 2^{128}$. For $n = 32$, $\log_2 32 = 5$, so $2^{2\times 32\times 5} = 2^{320}$. Each ρ_z is computed as $\rho_z = \prod_{i \in I_{r_k}} \sigma_i \prod_{j \in I_{c_k}} \tau_j$ for even z, and $\rho_z = \prod_{j \in I_{c_k}} \tau_j \prod_{i \in I_{r_k}} \sigma_i$ for odd z, where $k = \lfloor z/2 \rfloor$, and σ_i and τ_j are row and column permutations of L. These permutations affect C by permuting RM codewords and, through π_z, shuffling intermediate values during the $2\log_2 n$ round transformations, significantly increasing the key space.

- **Combined Key Space**: The total key space is the product of independent components, adjusted for dependencies. The secret key is $(G, K, L, \{I_{r_z}, I_{c_z}\}_{z=0,\ldots,\log_2 n-1})$. Assuming a fixed G, the size of the key space is:

$$|K| \times |L| \times \prod_{z=0}^{\log_2 n - 1} |I_{r_z} \times I_{c_z}| = 2^{\alpha} \times L(n) \times 2^{2n \log_2 n}$$

For $n = 16$, this is approximately $2^{20} \times 2^{379} \times 2^{128} = 2^{527}$. For $n = 32$, it is $2^{30} \times 2^{2407} \times 2^{320} = 2^{2757}$. These values far exceed recommended key sizes for symmetric ciphers [1,2]. Dependencies, such as the key schedule deriving $D_{\text{digits}}^{(z)}$ from K and L over $2\log_2 n$ rounds with swapping, may slightly reduce the effective key space, but the factorial growth of $L(n)$ and the exponential contribution of subsets dominate, ensuring robustness.

- **Impact on Ciphertext** C: The key components collectively determine C. The key K and Latin square L govern the multi-round transformations over $2\log_2 n$ rounds with row-column swapping, producing Y. The subsets $\{I_{r_z}, I_{c_z}\}$ generate ρ_z, which, with K, derives π_z, affecting Y, and directly permutes RM codewords. The matrix G encodes Y's portions. An attacker attempting to guess C without the key faces the full key space, as each component is essential for decryption. Even partial knowledge (e.g., G) leaves an infeasible number of possibilities for K, L, and $\{I_{r_z}, I_{c_z}\}$.

5.2. Resistance to Linear Cryptanalysis

Linear cryptanalysis seeks linear relationships between plaintext, ciphertext, and key bits to approximate the cipher's behavior. The scheme's non-linear Latin squares, scrambling permutations, and increased $2\log_2 n$ rounds with row-column switching provide strong resistance.

Each Latin square substitution introduces a bias of approximately $\frac{1}{n}$ in any linear approximation. Over $2m$ rounds (where $m = \log_2 n$), the total bias becomes:

$$\left(\frac{1}{n}\right)^{2m}$$

with required data $D \approx \text{bias}^{-2}$: - For $n = 16$, $m = 4$, $2m = 8$, the bias is $\left(\frac{1}{16}\right)^8 = 2^{-32}$, requiring 2^{64} known plaintext-ciphertext pairs to exploit, a significant computational burden. For $n = 32$, $m = 5$, $2m = 10$, the bias is $\left(\frac{1}{32}\right)^{10} = 2^{-50}$, needing 2^{100} pairs, rendering the attack infeasible.

Although RM codes are linear, the permutations ρ_z with swapping disrupt linear patterns by shuffling bits unpredictably, ensuring that linear cryptanalysis is ineffective for practical n.

5.3. Resistance to Differential Cryptanalysis

Differential cryptanalysis exploits non-random propagation of differences through the cipher. The strength against this relies on the non-linearity of L, the diffusion properties of π_z, and the number of rounds. The maximum differential probability (DP) of the Latin square substitution is expected to be low (e.g., $\approx 2/n$ for random S-boxes). Over $2\log_2 n$ rounds, the probability of a differential characteristic drops exponentially. Furthermore, the empirical tests presented in Section 4.7.2 show excellent avalanche characteristics (average close to 50% change) and, crucially, found no evidence of consistent input/output differential patterns for any tested parameter set ($RM(16, 15, 2)$, $RM(32, 16, 8)$, $RM(32, 26, 4)$). This suggests strong resistance against first-order differential attacks for these configurations.

5.4. Resistance to Algebraic Attacks

Algebraic attacks model the cipher as a system of multivariate polynomial equations over a finite field (GF(2)) to solve for the secret key. The scheme's resistance relies on its vast key space and high algebraic complexity across the $2\log_2 n$ rounds.

1. **Vast Key Space Complexity.** The full secret key includes the binary key K (length α), the Latin square L (order n), the generator matrix G of RM(r, m), and the permutation

subsets $\{I_r^z, I_c^z\}$. The combined size of the key components (excluding G for a fixed RM code) grows as:

$$\alpha \ + \ \log_2 L(n) \ + \ 2n \log_2 n.$$

This metric results in hundreds to thousands of bits (e.g., 527 bits for $n = 16$ and 2757 bits for $n = 32$), making exhaustively solving the system infeasible due to the massive number of variables.

2. **High Structural Degree.** The core algebraic strength stems from the composition of highly complex, non-linear round functions:
   - Non-linear Substitution: Each round's Latin square lookup, $y_i' = L[k_i][x_i]$, is inherently non-linear, causing the overall polynomial degree to grow exponentially with the number of rounds.
   - Key-Dependent Permutations: The α-bit round permutations π_z are derived through non-linear functions of ρ_z and the round key digits. These mix all state bits and prevent the isolation or linearization of individual S-box equations.
   - Tangled Constraints: The final stage involves linear encoding by G and permutation by ρ_z of the intermediate state Y. This adds tangled linear constraints that cannot be separated from the prior non-linear SPN equations, complicating the inversion process.

Together with the vast key space and the effect of $2 \log_2 n$ rounds, these design features ensure that deriving and solving a unified algebraic system for the key remains beyond current computational resources.

5.5. Resistance to Fault Attacks

Fault attacks inject errors to gain information. The integrated RM codes provide inherent resilience. The ciphertext consists of $\log_2 n$ independently encoded portions using $RM(n, k, d)$ with error-correcting capability t. Based on the tested parameters:

- For $RM(16, 15, 2)$: $n = 16, k = 15, d = 2 \implies t = \lfloor (2-1)/2 \rfloor = 0$. Each of the $\log_2 16 = 4$ portions can *detect* 1 bit-fault but corrects 0. An injected fault leads to a decoding failure (detectable error).
- For $RM(32, 16, 8)$: $n = 32, k = 16, d = 8 \implies t = \lfloor (8-1)/2 \rfloor = 3$. Each of the $\log_2 32 = 5$ portions corrects up to 3 bit-faults. More faults cause decoding failure.
- For $RM(32, 26, 4)$: $n = 32, k = 26, d = 4 \implies t = \lfloor (4-1)/2 \rfloor = 1$. Each of the $\log_2 32 = 5$ portions corrects up to 1 bit-fault. More faults cause decoding failure.

In all cases, faults up to t are corrected silently. Faults exceeding t result in a decoding failure (producing an incorrect, likely all-zero or garbage, block segment) rather than revealing intermediate state information related to the key. This inherent detection/correction mechanism significantly hinders fault attacks aiming to extract secrets through differential fault analysis.

5.6. Resistance to Chosen-Ciphertext Attacks

Chosen-Ciphertext Attacks (CCAs) involve an adversary querying a decryption oracle with chosen ciphertexts to gain information about a target ciphertext or the secret key. The proposed scheme's architecture provides resistance through the combined action of the Reed-Muller codes and the core SPN structure.

- **RM Codes as a Malleability Barrier:** A key feature against CCA is the integrated RM codes. When an adversary submits a modified ciphertext C_i to the oracle, the RM decoding behavior acts as a crucial barrier:

  - If modifications constitute less than $t = \lfloor (d-1)/2 \rfloor$ errors, the decoder corrects them, returning the original intermediate segment Y_i. The adversary gains no information from the modification.
  - If modifications exceed t errors, the decoder fails, producing an invalid output (e.g., all zeros or garbage). This prevents the adversary from observing a meaningful plaintext related to their manipulated ciphertext.

  This mechanism directly prevents attacks relying on ciphertext malleability, as predictable plaintext changes cannot be induced from controlled ciphertext alterations.
- **Core SPN Security:** The fundamental cryptographic strength against CCA resides in the complex, multi-round SPN. Operating over $2\log_2 n$ rounds, it employs:
  - Non-linear Latin square substitutions (L).
  - Key-dependent α-bit permutations (π_z) derived from a non-linear process.
  - Unique round keys ($D^{(z)}_{\text{digits}}$) from a non-linear key schedule.
  - Round structure variations via row-column swapping.

  These components ensure that the relationship between the intermediate state Y (the SPN output before RM encoding) and the final plaintext M is computationally infeasible to invert without the secret key. Even if an adversary could somehow bypass the RM layer, recovering M or key information from Y remains intractable.
- **Codeword Permutation Obscurity:** The application of the secret n-bit permutations ρ_z to the RM codewords further obscures the link between the SPN output Y and the transmitted ciphertext C, adding another layer of difficulty for the adversary analyzing oracle responses.

In concert, the RM layer's ability to detect or correct modifications, combined with the cryptographic strength and complexity of the multi-round SPN, renders the scheme resistant to Chosen-Ciphertext Attacks.

5.7. Security Enhancement via Overlapping Portions

The scheme enhances diffusion by splitting the α-bit intermediate state Y (output of the SPN) into $\log_2 n$ overlapping k-bit portions $Y^{(i)}$ for RM encoding. Since Y is formed by concatenating k base-n digits (each $\log_2 n$ bits wide) after $2\log_2 n$ rounds of substitution and permutation, a single input bit flip in M typically affects multiple digits in the final SPN state. When this state Y is re-segmented into k-bit chunks $Y^{(i)}$, the changes are distributed across the inputs to multiple independent RM encoders. This redistribution effect is supported by the experimental results in Table 4, which demonstrate a near-uniform distribution of output bit changes across all RM portions following a single input bit flip. This mechanism ensures that local changes are rapidly spread across the entire ciphertext block, complementing the round function's diffusion.

5.8. Parameter Flexibility and Limitations

The proposed cryptographic scheme offers flexible adaptation to diverse applications. By varying the Latin square order n, the number of key/message segments k, and the RM code's minimum distance d, the scheme balances security, error correction, and efficiency. The independent encoding of $\log_2 n$ k-bit portions supports parallelization, enhancing performance in multi-core systems.

Storage Overhead. Parameter growth escalates resource demands. The primary storage cost comes from precomputed components required for efficient operation. To accurately quantify the memory requirement per device, we empirically measured the total process memory increment (Resident Set Size, RSS) during the sequential initialization

of two identical cipher instances (Alice and Bob) using the `psutil` module. The total measured increment was then divided by two to estimate the footprint for a single user.

As shown, the measured practical storage requirements range from an estimated 6.0 KiB up to 238.0 KiB for the largest tested configuration ($RM(32, 26, 4)$). This footprint, dominated by the allocation of permutation tables and the RM decoding structure, could be challenging for highly constrained IoT devices (often under 20 KiB RAM).

Computational Overhead and Mitigation. Computationally, the SPN complexity is approximately $O(\log_2 n \cdot \alpha \log \alpha)$, dominated by permutation generation. Reed-Muller encoding/decoding complexity typically scales with n, often as $O(n \log n)$ or $O(\log^2 n)$. The RM decoding latency, identified in the performance analysis (Table 5), is the most significant cost but is mitigated by parallelization.

Mitigation in Distributed Systems. The memory footprint represents a fixed, one-time investment per device. When the scheme is utilized in a distributed or cloud environment, this cost is less restrictive because the large, precomputed structures (Latin squares and permutation tables) are loaded only once, and the investment is justified by the parallel processing benefits and high throughput achieved by processing multiple $\log_2 n$ portions concurrently.

The scheme's flexibility with n, k, and d supports different applications, with robust key spaces and error correction. However, the calculated storage and computational overheads highlight the need for careful parameter selection and potential optimizations like partial precomputation or hardware acceleration for deployment in resource-constrained environments.

6. Conclusions

This paper presents a novel symmetric block cipher based on an SPN that integrates Latin squares, permutations, and Reed-Muller (RM) codes to provide robust security and resilience. The scheme processes a message and key of length $\alpha = k \cdot \log_2 n$ bits over $2 \log_2 n$ rounds, featuring row-column swapping for enhanced diffusion. The intermediate state is split into $\log_2 n$ portions, each encoded using $RM(n, k, d)$, permuted by ρ_z, yielding a ciphertext of length $\log_2 n \cdot n$ bits. The design supports parallel processing for efficiency.

The scheme offers a large key space, ensuring resistance to brute-force attacks. The RM code corrects up to $t = \lfloor (d-1)/2 \rfloor$ errors per portion (or detects errors if $t = 0$), enhancing resilience against channel noise and fault attacks (Section 5). Experimental results (Section 4.7) demonstrate excellent diffusion properties, with average avalanche effects near 50% and uniform distribution of changes across output blocks for tested parameters ($RM(16, 15, 2)$, $RM(32, 16, 8)$, $RM(32, 26, 4)$). Empirical tests also showed no consistent differential patterns, suggesting resistance to first-order differential attacks. The multi-round SPN structure, overlapping portions, and key-dependent elements contribute to resistance against linear, algebraic, and chosen-ciphertext attacks (Section 5).

Limitations primarily involve resource usage. While empirical differential tests were positive for the tested parameters and messages, the theoretical possibility of weak keys or higher-order characteristics warrants further study. Significant measured storage demands, ranging from approximately 6.0 KiB to 238.0 KiB per user for the practical parameter sets tested (Table 6), pose a challenge for highly memory-constrained devices. Computational complexity, particularly the $O(n \log^2 n)$ scaling and observed latency of RM decoding, impacts low-power devices, though parallelization provides substantial mitigation (Table 5).

Table 6. Measured Memory Footprint for Precomputed Components (Single User Estimate).

Parameter Set	Total Bits (α)/Code	Estimated Single User Footprint (KiB)
RM(32, 26, 4)	130/RM(3, 5)	238.0 KiB
RM(32, 16, 8)	80/RM(2, 5)	20.0 KiB
RM(16, 15, 2)	60/RM(3, 4)	6.0 KiB
RM(8, 4, 4)	12/RM(1, 3)	<1.0 KiB

The single-user footprint is estimated by dividing the total process memory increment (RSS) during sequential initialization (Alice + Bob) by two.

Future research should focus on optimizing storage, potentially through dynamic Latin square generation or partial precomputation. Deeper cryptanalysis, including formal proofs and simulations against side-channel attacks, is needed. Investigating the use of other error-correcting codes, such as Reed-Solomon codes, to handle different error models (e.g., burst errors) could broaden applicability. Exploring secret sharing for key/component distribution and hardware acceleration for permutations and RM decoding could further enhance security and performance.

In conclusion, the proposed scheme offers a flexible, resilient framework for symmetric cryptography. It balances a vast key space and error correction with computational and storage overheads. The demonstrated diffusion properties and potential for parallelization make it a promising candidate for diverse applications, particularly where resilience is paramount, with future optimizations poised to improve practicality for resource-constrained environments.

Supplementary Materials: The following supporting information can be downloaded at: https://www.mdpi.com/article/10.3390/cryptography9040070/s1.

Author Contributions: Conceptualization, H.A. and C.H.; methodology, H.A. and C.H.; software, H.A.; validation, H.A. and C.H.; formal analysis, H.A.; investigation, H.A.; resources, H.A.; data curation, H.A.; writing—original draft preparation, H.A.; writing—review and editing, C.H.; visualization, H.A.; supervision, C.H.; project administration, C.H. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The code and data are in the process of being prepared for open-source publication.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Daemen, J.; Rijmen, V. *The Design of Rijndael: AES—The Advanced Encryption Standard*; Springer: Berlin, Germany, 2020.
2. Bogdanov, A.; Knudsen, L.R.; Leander, G.; Paar, C.; Poschmann, A.; Robshaw, M.J.; Seurin, Y.; Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems—CHES 2007*; Springer: Berlin, Germany, 2007; pp. 450–466.
3. Prouff, E.; Renault, G.; Rivain, M.; O'Flynn, C.; Mukhopadhyay, C.; Saha, D. Fault Attacks on Symmetric Cryptography. In *Embedded Cryptography 1*; Prouff, E., Renault, G., Rivain, M., O'Flynn, C., Eds.; Wiley: Hoboken, NJ, USA, 2025; pp. 209–230.
4. Baksi, A.; Bhasin, S.; Breier, J.; Jap, D.; Saha, D. A survey on fault attacks on symmetric key cryptosystems. *ACM Comput. Surv.* **2022**, *55*, 86. [CrossRef]
5. Challa, R.; Gunta, V. Towards the construction of reed-muller code based symmetric key FHE. *Ing. Syst. Inf.* **2021**, *26*, 585–590. [CrossRef]
6. Zarei Zefreh, E.; Abdali, M. LSIE: A fast and secure Latin square-based image encryption scheme. *Multimed. Tools Appl.* **2024**, *23*, 7939–7979. [CrossRef]
7. El-Shafai, W.; Mesrega, A.K.; Ahmed, H.E.H.; El-Bahnasawy, N.A.; Abd El-Samie, F.E. An efficient multimedia compression-encryption scheme using latin squares for securing Internet-of-things networks. *J. Inf. Secur. Appl.* **2022**, *63*, 103039. [CrossRef]
8. Wu, Y.; Zhou, Y.; Noonan, J.P.; Agaian, S. Design of image cipher using latin squares. *Inf. Sci.* **2014**, *264*, 317–339. [CrossRef]

9. Ali, N.H.M.; Hoobi, M.M.; Saffo, D.F. Development of Robust and Efficient Symmetric Random Keys Model Based on the Latin Square Matrix. *Mesopotamian J. Cybersecur.* **2024**, *4*, 203–215. [CrossRef]
10. Repka, M.; Cayrel, P.L. Cryptography based on error correcting codes: A survey. In *Multidisciplinary Perspectives in Cryptology and Information Security*; IGI Global: Hershey, PA, USA, 2014; pp. 133–156.
11. Alabady, S.A.; Salleh, M.F.M.; Al-Turjman, F. LCPC error correction code for IoT applications. *Sustain. Cities Soc.* **2018**, *42*, 663–673. [CrossRef]
12. Boneh, D.; DeMillo, R.A.; Lipton, R.J. On the importance of eliminating errors in cryptographic computations. *J. Cryptol.* **2001**, *14*, 101–119. [CrossRef]
13. Carlet, C. Boolean functions for cryptography and error correcting codes. In *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*; Cambridge University Press: Cambridge, UK, 2010; pp. 257–397.
14. MacWilliams, F.J.; Sloane, N.J.A. *The Theory of Error-Correcting Codes*; Elsevier: Amsterdam, The Netherlands, 1977.
15. Vaudenay, S. *A Classical Introduction to Cryptography: Applications for Communications Security*, 1st ed.; Springer: New York, NY, USA, 2005; p. 336.
16. Colbourn, C.J.; Dinitz, J.H. *Handbook of Combinatorial Designs*, 2nd ed.; Chapman and Hall/CRC: Boca Raton, FL, USA, 2006; pp. 135–151.
17. Thomas, B.; Yi, L.; Vaudenay, S.; Junod, P.; Monnerat, J. *A Classical Introduction to Cryptography Exercise Book*, 1st ed.; Springer: New York, NY, USA, 2007; p. 254.
18. Cayley, A. Desiderata and suggestions: No. 1. The theory of groups. *Am. J. Math.* **1878**, *1*, 50–52. [CrossRef]
19. Cayley, A. Desiderata and suggestions: No. 2. The theory of groups: Graphical representation. *Am. J. Math.* **1878**, *1*, 174–176. [CrossRef]
20. Humphreys, J.F. *A Course in Group Theory*, 1st ed.; Oxford University Press: Oxford, UK, 1996; p. 296.
21. Cayley, A. On Latin squares. In *Messenger of Math*; Glaisher, J.W.L., Ed.; Macmillan and Co.: London, UK, 1890; pp. 135–137.
22. Cameron, P.J. Notes on Cryptography. University of London 2003. Available online: https://cameroncounts.wordpress.com/wp-content/uploads/2013/11/crypt.pdf (accessed on 17 October 2025).
23. Keedwell, A.D.; Dénes, J. Elementary properties. In *Latin Squares and Their Applications*, 2nd ed.; North-Holland: Boston, MA, USA, 2015; pp. 1–36. [CrossRef]
24. Singh, B.; Athithan, G.; Pillai, R. On extensions of the one-time-pad. *AIP Conf. Proc.* **2021**, *298*. Available online: https://eprint.iacr.org/2021/298 (accessed on 19 October 2025).
25. Zhang, J.; Zhu, Y.; Abdelraheem, A.; Elkins-Arce, H.D.; Dever, J.; Wheeler, T.; Isakeit, T.; Hake, K.; Wedegaertner, T. Use of a Latin square design to assess experimental errors in field evaluation of cotton for resistance to Fusarium wilt race 4. *Crop Sci.* **2022**, *62*, 575–591. [CrossRef]
26. Chen, J.; Patra, J.; Pradel, M.; Xiong, Y.; Zhang, H.; Hao, D.; Zhang, L. A survey of compiler testing. *ACM Comput. Surv.* **2020**, *53*, 4. [CrossRef]
27. Luo, Y.; Lutsenko, V.I.; Shulga, S.N. New method for designing non-equidistant plane antenna arrays with full coverage of spatial frequencies based on Latin squares and their triangular matrix. *Telecommun. Radio Eng.* **2021**, *80*, 15–28. [CrossRef]
28. Zolfaghari, B.; Bibak, K. Combinatorial cryptography and Latin squares. In *Perfect Secrecy in IoT: A Hybrid Combinatorial-Boolean Approach*; Springer: Cham, Switzerland, 2022; pp. 37–55.
29. Chauhan, D.; Gupta, I.; Verma, R. Quasigroups and their applications in cryptography. *Cryptologia* **2021**, *45*, 227–265. [CrossRef]
30. Mohammed, S.D.; Hasan, T.M. Cryptosystems using an improving hiding technique based on Latin square and magic square. *Indones. J. Electr. Eng. Comput. Sci.* **2020**, *20*, 510–520. [CrossRef]
31. Schmidt, N.O. Latin Squares and Their Applications in Cryptography. Master's Thesis, Boise State University, Boise, ID, USA, September 2016.
32. Hua, Z.; Li, J.; Chen, Y.; Yi, S. Design and application of an S-box using complete Latin square. *Nonlinear Dyn.* **2021**, *104*, 807–825. [CrossRef]
33. Kumar, U.; Venkaiah, V.C. A new modified MD5-224 bits hash function and an efficient message authentication code based on quasigroups. In *Cyber Security, Privacy and Networking: Proceedings of ICSPN 2021*; Springer Nature Singapore: Singapore, 2022; pp. 1–12.
34. Ahmad, H.; Hannusch, C. A new keyed hash function based on Latin squares and error-correcting codes to authenticate users in smart home environments. In Proceedings of the Codes, Cryptology and Information Security, Rabat, Morocco, 29–31 May 2023; pp. 129–135.
35. Wu, W.; Wang, Q. Cryptanalysis and improvement of an image encryption algorithm based on chaotic and Latin square. *Nonlinear Dyn.* **2023**, *111*, 3831–3850. [CrossRef]
36. Shen, J.; Zhang, T.; Jiang, Y.; Zhou, T.; Miao, T. A novel key agreement protocol applying Latin square for cloud data sharing. *IEEE Trans. Sustain. Comput.* **2022**, *8*, 639–651. [CrossRef]
37. Dixon, J.D.; Mortimer, B. *Permutation Groups*, 1st ed.; Springer: New York, NY, USA, 1996.

38. Doliskani, J.N.; Malekian, E.; Zakerolhosseini, A. A cryptosystem based on the symmetric group Sn. *Int. J. Comput. Sci. Netw. Secur.* **2008**, *8*, 226–234.
39. Stinson, D.R.; Paterson, M.B. Block ciphers and stream ciphers. In *Cryptography: Theory and Practice*, 4th ed.; Chapman and Hall/CRC: Boca Raton, FL, USA, 2019; pp. 83–136.
40. Roth, R. *Introduction to Coding Theory*, 1st ed.; Cambridge University Press: Cambridge, UK, 2006; p. 580.
41. Abbe, E.; Shpilka, A.; Ye, M. Reed–Muller codes: Theory and algorithms. *IEEE Trans. Inf. Theory* **2021**, *67*, 3251–3277. [CrossRef]
42. Kuppusamy, A.; Pitchai Iyer, S.; Krithivasan, K. Two-key dependent permutation for use in symmetric cryptographic system. *Math. Probl. Eng.* **2014**, *2014*, 795292. [CrossRef]
43. Scharinger, J. An excellent permutation operator for cryptographic applications. In Proceedings of the Computer Aided Systems Theory—EUROCAST 2005, Las Palmas de Gran Canaria, Spain, 7–11 February 2005; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany; Volume 3643, pp. 317–326.
44. Chou, Y.C.; Lin, C.H.; Li, P.C.; Li, Y.C. A (2,3) threshold secret sharing scheme using Sudoku. In Proceedings of the 6th International Conference on Intelligent Information Hiding and Multimedia Signal Processing, Darmstadt, Germany, 15–17 October 2010; pp. 43–46.
45. Pal, S.K.; Kapoor, S.; Arora, A.; Chaudhary, R.; Khurana, J. Design of strong cryptographic schemes based on Latin squares. *J. Discret. Math. Sci. Cryptogr.* **2010**, *13*, 233–256. [CrossRef]
46. Van Lint, J.H.; Wilson, R.M. *A Course in Combinatorics*, 2nd ed.; Cambridge University Press: Cambridge, UK, 2001.
47. Sequence A002860 in the On-Line Encyclopedia of Integer Sequences. Available online: https://oeis.org/A002860 (accessed on 27 May 2025).
48. Huffman, W.C.; Pless, V. *Fundamentals of Error-Correcting Codes*, 1st ed.; Cambridge University Press: Cambridge, UK, 2003.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.