Classic McEliece Algorithm Explained in Detail

CONTENT

---

01. Key Generation Phase

The goal of the key generation phase is to produce a public key T and a private key (δ, c, g, a, s).

- **Public Key**: T
- **Private Key**: (δ, c, g, a, s)
  - $c = (c_{mt-\mu}, \ldots, c_{mt-1})$
  - $\alpha = ({\alpha'}_0, \ldots, \alpha'_{n-1}, \alpha_n, \ldots, \alpha_{q-1})$

---

1. Key Generation Steps

1.1 Generate a uniformly random *l*-bit string δ.

This δ serves as the seed for a Pseudorandom Generator (PRG).

1.2 Run $SeededKeyGen(\delta)$ to generate the public and private keys.

1.2.1 Compute $E = PRG(\delta)$, which is an $n + \sigma_2 q + \sigma_1 t + l$ bit string.

1.2.2 Define δ' as the last l bits of E.

1.2.3 Define s as the first n bits of E.

1.2.4 Use the next $\sigma_2 q$ bits from E to compute $\alpha_0, \ldots, \alpha_{q-1}$ via the $FieldOrdering$ algorithm. If it fails, set $\delta \leftarrow \delta'$ and restart the algorithm.

1.2.5 Use the next $\sigma_1 t$ bits from E to compute g via the $Irreducible$ algorithm. If it fails, set $\delta \leftarrow \delta'$ and restart the algorithm.

> Note: As mentioned in Chapter 9 of the code and documentation, this step also involves calculating the control bits for the Benes network corresponding to the permutation $\pi(i)$ stored in the private key sk. This is handled by the $controlbitsfrompermutation$ function.

1.2.6 Define $\Gamma = (g, \alpha_0, \alpha_1, \ldots, \alpha_{n-1})$. (Note that $\alpha_n, \ldots, \alpha_{q-1}$ are not used in Γ).

1.2.7 Compute $(T, c_{mt-\mu}, \ldots, c_{mt-1}, \Gamma') \leftarrow MatGen(\Gamma)$. If it fails, set $\delta \leftarrow \delta'$ and restart the algorithm.

1.2.8 Write Γ' as $(g, \alpha'_0, \alpha'_1, \ldots, \alpha'_{n-1})$.

1.2.9 Output T as the public key and (δ, c, g, a, s) as the private key, where $c = (c_{mt-\mu}, \ldots, c_{mt-1})$ and $a = (\alpha'_0, \ldots, \alpha'_{n-1}, \alpha_n, \ldots, \alpha_{q-1})$.

---

1. Key Generation Phase --- 1.2.4 FieldOrdering Algorithm

This algorithm generates the support elements for the Goppa code.

1. Take the first σ_2 input bits $b_0, b_1, \ldots, b_{\sigma_2-1}$ and interpret them as an integer a_0 (σ_2 bits): $a_0 = b_0 + 2b_1 + \ldots + 2^(\sigma_2 - 1) * b_{\sigma_2-1}$. Repeat this process to generate $a_1, \ldots, a_{q-1}$.

2. If there are any duplicate values among $a_0, a_1, \ldots, a_{q-1}$, return $\perp$ (failure).

3. Sort the pairs (a_i, i) lexicographically to get $(\alpha_{\pi(i)}, \pi(i))$, where π is a permutation of $0, 1, \ldots, q-1$.

4. Define α_i as a polynomial: $\alpha_i = \Sigma_{j=0}^{m-1} \pi(i)_j \cdot z^(m - 1 - j)$.

> Here, $\pi(i)_j$ represents the j-th least significant bit of $\pi(i)$. The finite field F_q is constructed as $F_2[z]/f(z)$.

---

1. Key Generation Phase --- 1.2.5 Irreducible Algorithm (Computing g)

This algorithm takes a $\sigma_1 t$-bit input string $d_0, d_1, \ldots, d_{\sigma_1 t-1}$ and outputs either $\perp$ (failure) or a monic, irreducible polynomial g of degree t in $F_q[x]$.

1. For each $j \in 0, 1, \ldots, t-1$, define $\beta_j = \Sigma_{i=0}^{m-1} d_{\sigma_1 j+i} z^{\text{i}}$.

> Within each block of σ_1 input bits, only the first m bits are used. The algorithm ignores the remaining bits.

2. Define $\beta = \beta_0 + \beta_1 y + \ldots + \beta_{t-1} y^(t-1) \in F_q[y]/F(y)$. This is used to construct a matrix.
3. Compute the minimal polynomial g of β over F_q.

> By definition, g is monic and irreducible, and $g(\beta) = 0$.

- **Construct a linearly dependent set:** We know $g(x)$ has degree t. Therefore, the $t+1$ elements $1, \beta, \beta^2, \ldots, \beta^t$ must be linearly dependent in $F_(2^m)t$. This means there exist coefficients $g_0, g_1, \ldots, g_t \in F_2 m$, not all zero, such that:
  $g_0 \cdot 1 + g_1 \cdot \beta + g_2 \cdot \beta^2 + \ldots + g_t \cdot \beta^t = 0$
  Since $g(x)$ is monic, we can set $g_t = 1$.

- **Build the matrix:** Express each power β^k (for $k = 0, \ldots, t$) as a polynomial in y of degree less than t:
  $\beta^k = b_{k,0} + b_{k,1} y + \ldots + b_{k,t-1} y^(t-1)$
  By expanding the linear dependency equation and setting the coefficient of each power of y to zero, we obtain a txt system of linear equations for the unknown coefficients $g_0, \ldots, g_{t-1}$.

- **Matrix Form:** The system of equations can be written in matrix form:

```
[ b0,0   b1,0   ...  b_{t-1},0 ] [ g0 ]   [ b_{t,0}  ]
[ b0,1   b1,1   ...  b_{t-1},1 ] [ g1 ]   [ b_{t,1}  ]
[  :      :     ...      :     ] [ :  ] =-[   :      ]
[ b0,t-1 b1,t-1 ...  b_{t-1},t-1] [g_{t-1}] [ b_{t,t-1} ]
```

- **Solve the System:** Use a method like Gaussian elimination over the field F_2m to solve this linear system and find the unique solution $g_0, g_1, \ldots, g_{t-1}$.

- **Construct g(x):** The final Goppa polynomial is:
$g(x) = g_0 + g_1x + \ldots + g_{t-1}x^(t-1) + x^t$

4. If the degree of g is t, return g. Otherwise, return $\perp$.

> This check is equivalent to determining if the matrix after Gaussian elimination has a non-zero pivot in every column.

5. This step is not part of the $Irreducible$ algorithm itself but is mentioned on slide 10. The $FieldOrdering$ algorithm (1.2.4) outputs $(\alpha_0, \alpha_1, \ldots, \alpha_{q-1})$.

---

1. Key Generation Phase --- $controlbitsfrompermutation$

This algorithm is used to compute the control bits for a Benes network to perform a permutation, which is crucial for security and efficiency.

- **Problem Context**: We need to reorder a large set of data (e.g., an array). A naive approach of creating a new array and copying elements can have two major drawbacks:
  - **Security**: In cryptography, data access patterns (e.g., the order of reading memory addresses) can leak secret information. This is known as a **Timing Attack**.
  - **Efficiency**: For hardware implementations, specialized circuits are often much faster than general-purpose memory read/write operations.

- **Permutation Networks**: These are specialized hardware structures designed to solve this problem. They consist of a series of basic "switches" that can realize any permutation of the input data. The **Beneš network** is a classic and efficient type of permutation network. To make the network perform a specific permutation (e.g., transform (a, b, c, d) to (c, a, d, b)), each switch in the network must be set to the correct state (either pass-through or swap inputs). These settings are the **control bits**.
- **Core Problem**: Given a desired permutation π, how do we quickly and correctly compute the set of control bits needed to configure the network?

Beneš Network Decomposition

A key property of a Beneš network for $n = 2^k$ inputs is that its permutation π can be decomposed into a composition of three sub-operations:
$\pi = F \circ M \circ L$
(Function composition is executed from right to left).

- **L (Input Side)**: The first layer of switches, controlled by a set of bits l (lastcontrol). It performs conditional swaps on adjacent input pairs $(x, x+1)$, specifically $(0, 1), (2, 3), (4, 5), \ldots$.
- **M (Middle)**: The core of the network. After the L operation, the data enters two smaller, independent Beneš networks, each of size $n/2$. M represents the permutation performed by these two sub-networks. M has a crucial **parity-preserving property**: even-indexed inputs are only ever sent to even-indexed outputs, and odd-indexed inputs are only ever sent to odd-indexed outputs. Because of this, M can be decomposed into two independent permutations of size $n/2$:
    - M_0: Permutes the even-indexed positions.
    - M_1: Permutes the odd-indexed positions.
- **F (Output Side)**: The final layer of switches, controlled by a set of bits f (firstcontrol). Its function is similar to L, performing conditional swaps on adjacent positions $(x, x+1)$ to complete the final steps of the permutation.

The core task of the algorithm is to compute the correct f and l and deduce the sub-permutations M_0 and M_1. Then, the same algorithm is called recursively on M_0 and M_1 until the network size is reduced to 2.

Algorithm Steps

Step 1: Introduce the $XbackXforth$ Transform (Define π')
The algorithm first applies a clever transformation to the original permutation π to get a new permutation π'.
$\pi' = XbackXforth(\pi)$, defined as $\pi'(x) = \pi(\pi^{-1}(x \oplus 1) \oplus 1)$
where $\oplus$ is the bitwise XOR operation. This gives π' a useful property (Theorem 4.4):
$cyclemin(\pi')(x \oplus 1) = cyclemin(\pi')(x) \oplus 1$
This means that if we calculate the "cycle minimum" for an even number x, we can find the cycle minimum for its odd neighbor $x \oplus 1$ with a single XOR operation, effectively halving the computation.

Step 2: Compute Cycle Minimum ($cyclemin$)
The goal is to compute $c(x) = cyclemin(\pi')(x)$.

- **Definition**: A permutation consists of disjoint cycles. $cyclemin(\pi')(x)$ refers to the smallest element in the cycle that contains x. This smallest value is called the **cycle leader** of x.

- **Computation** ($fastcyclemin$): This is an iterative process suitable for parallelization.

  - Let $c^0(x) = x$
  - $c_1(x) = min(c^0(x), c^0(\pi'(x))) = min(x, \pi'(x))$
  - $c_2(x) = min(c_1(x), c_1(\pi'^2(x)))$ (This finds the minimum among $x, \pi'(x), \pi'^2(x), \pi'^3(x)$)
  - ...
  - $c_i(x) = min(c_{i-1}(x), c_{i-1}(\pi'^(2^(i-1)))(x))$
    For an input size $n = 2^m$, after $m-1$ iterations, $c_{m-1}(x)$ will find the minimum value in the entire cycle containing x.

Step 3: Compute $firstcontrol$ (f)
$f_j = c(2j) mod 2$ (for j from 0 to $n/2 - 1$)
The j-th control bit f_j is simply the parity (0 for even, 1 for odd) of the cycle leader $c(2j)$ of the j-th even number $2j$.

Step 4: Compute $lastcontrol$ (l)
The calculation for l is slightly more complex, depending on both the original permutation π and the F operation (defined by f).
$l_k = F(\pi(2k)) mod 2$ (for k from 0 to $n/2 - 1$)

1. Take the k-th even number $2k$.
2. Find where the original permutation π maps it: $\pi(2k)$.
3. Apply the F operation to this result: $F(y) = y \oplus f[y/2]$.
4. The parity of the final result is l_k.

Step 5: Compute and Decompose the Middle Permutation M
With F and L known, M can be derived from the relation $\pi = F \circ M \circ L$:
$M = F^{-1} \circ \pi \circ L^{-1}$
Since F and L are composed of conditional swaps, they are their own inverses ($F^{-1} = F$, $L^{-1} = L$). So:
$M = F \circ \pi \circ L$
Theorem 5.5 guarantees that this M is parity-preserving. It can therefore be decomposed into two sub-permutations:

- $M_0(j) = M(2j)/2$
- $M_1(j) = (M(2j + 1) - 1)/2$

Recursion and Termination

- **Recursive Call**: The algorithm is now called on the two new permutations M_0 and M_1 (of size $n/2$) to repeat steps 1-5 and find their respective control bits.
- **Termination Condition**: The recursion stops when $n = 2$. The permutation is either $(0, 1)- > (0, 1)$ or $(0, 1)- > (1, 0)$, and the control bit is simply $\pi[0]$.
- **Combined Result**: The final, complete control bit sequence for the n-input network is constructed by concatenating the results: the bits for f, followed by the interleaved bits from the recursive calls on M_0 and M_1, followed by the bits for l.

---

1. Key Generation Phase --- 1.2.5 Computing the Systematic Form

This section describes how the public key matrix T is derived from the Goppa code's parity-check matrix.

Case 1: Systematic Form $(\mu, \nu) = (0, 0)$

1. Compute the txn matrix $M = h_{i,j}$ over F_q, where $h_{i,j} = a_j{}^i / g(a_j)$, for $i = 0, \ldots, t-1$ and $j = 0, \ldots, n-1$.
2. Expand this into a binary $mtxn$ matrix N by replacing each entry $u_0 + u_1 z + \ldots + u_{m-1} z^(m-1)$ of M with an m-bit column vector $(u_0, u_1, \ldots, u_{m-1})^T$.
3. Reduce N to systematic form $(I_{mt} | T)$, where I_{mt} is the $mtxmt$ identity matrix. If this fails, return $\perp$. The right-hand part T is a portion of the public key.
4. Return (T, Γ).

The Goppa Code Parity-Check Matrix (H)

- **Phase 1: Initial Form over $GF(2^m)$**
  The initial parity-check matrix H is constructed as:

  All operations (addition, multiplication, inversion) are performed in the finite field $GF(2^m)$.

- **Phase 2: Conversion to a Binary Matrix**
  The matrix H has elements from $GF(2^m)$, not the bits 0 and 1 ($GF(2)$) that computers handle directly. A "trace construction" is used for conversion.

  - $GF(2^m)$ can be viewed as an m-dimensional vector space over $GF(2)$. This means any element of $GF(2^m)$ can be uniquely represented as an m-bit binary vector.

  - **Conversion Process**:

    1. Expand each row of H into m rows.
    2. Replace each element (from $GF(2^m)$) in the original matrix with its corresponding $mx1$ binary column vector.

  - This results in a binary $mtxn$ parity-check matrix H_bin.

- Using Gaussian elimination, H_bin is converted to systematic form:
  $H_sys = [I|T]$
  where I is an $mtxmt$ identity matrix and T is the $mtx(n-mt)$ public key matrix.

Case 2: Semi-Systematic Form (General μ, ν)

For the general case, the algorithm produces a matrix in semi-systematic form.

1. Steps 1 and 2 (calculating M and N) are the same as in the systematic case.
2. Reduce N to (μ, ν)-semi-systematic form to get matrix H'. If this fails, return $\perp$.

> In this form, for $0 \leq i < mt - \mu$, the i-th row has its leading 1 at column $c_i = i$. For the remaining rows, the leading 1s are at columns c_i where
> $mt - \mu \leq c_{mt-\mu} < \ldots < c_{mt-1} < mt - \mu + \nu$.

3. Set $(\alpha'_0, \ldots, \alpha'_{n-1}) \leftarrow (\alpha_0, \ldots, \alpha_{n-1})$.
4. For i from $mt - \mu$ to $mt - 1$ (in order), swap column i with column c_i in H'. Simultaneously, swap a'_i and a'_{c_i}.

> After this swap, the i-th row has its leading 1 in the i-th column. If $c_i = i$, no swap is performed.

5. The matrix H' is now in the full systematic form $(I_{mt}|T)$. The algorithm returns $(T, c_{mt-\mu}, \ldots, c_{mt-1}, \Gamma')$, where Γ' contains the modified support elements.

---

02. Encapsulation Phase

The randomized $Encap$ algorithm takes the public key T as input and outputs a ciphertext C and a session key K.

Algorithm for non-pc parameter sets:

1. Generate a vector $e \in F_2{}^n$ of weight t using the $FixedWeight$ algorithm.
2. Compute $C = Encode(e, T)$.
3. Compute $K = Hash(1, e, C)$.
4. Output ciphertext C and session key K.

Algorithm for pc parameter sets:

1. Generate a vector $e \in F_2{}^n$ of weight t using the $FixedWeight$ algorithm.
2. Compute $C_0 = Encode(e, T)$.
3. Compute $C_1 = Hash(2, e)$. Let $C = (C_0, C_1)$.
4. Compute $K = Hash(1, e, C)$.
5. Output ciphertext C and session key K.

---

2. Encapsulation Phase --- 2.1 FixedWeight()

This algorithm outputs a vector $e \in F_2{}^n$ with Hamming weight t.

1. Generate $\sigma_1 \tau$ uniformly random bits, where τ is a pre-calculated integer $\tau \geq t$.
2. For each $j \in 0, 1, \ldots, \tau - 1$, define d_j by taking a block of σ_1 bits and interpreting the first m of them as an integer.
3. Define $a_0, a_1, \ldots, a_{t-1}$ as the first t unique entries selected from $d_0, d_1, \ldots, d_{\tau-1}$ in the range $0, 1, \ldots, n-1$. If fewer than t unique entries are found, restart the algorithm.
4. If there are any duplicate elements among $a_0, a_1, \ldots, a_{t-1}$, restart.
5. Define the weight-t vector $e = (e_0, e_1, \ldots, e_{n-1}) \in F_2{}^n$ such that for each i, $e_{a_i} = 1$.
6. Return e.

---

2. Encapsulation Phase --- 2.2 Encode(e, T)

This algorithm takes two inputs: a weight-t column vector $e \in F_2{}^n$ and the public key T, which is an $mtxk$ matrix over F_2. It outputs a vector $C \in F_2{}^{mt}$.

1. Define the public parity-check matrix $H = (I_{mt}|T)$.
2. Compute and return $C = He \in F_2{}^{mt}$.

---

03. Decapsulation Phase

The $Decap$ algorithm takes a ciphertext C and the private key as input and outputs a session key K.

Algorithm for non-pc parameter sets:

1. Set $b \leftarrow 1$.
2. Extract $s \in F_2{}^n$ and $\Gamma' = (g, \alpha'_0, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow Decode(C, \Gamma')$. If $e = \perp$ (decoding failure), set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $K = Hash(b, e, C)$.
5. Output session key K.

Algorithm for pc parameter sets:

1. Split the ciphertext C into (C_0, C_1), where $C_0 \in F_2{}^{mt}$. Set $b \leftarrow 1$.
2. Extract $s \in F_2{}^n$ and $\Gamma' = (g, \alpha'_0, \ldots, \alpha'_{n-1})$ from the private key.
3. Compute $e \leftarrow Decode(C_0, \Gamma')$. If $e = \perp$, set $e \leftarrow s$ and $b \leftarrow 0$.
4. Compute $C'_1 = Hash(2, e)$.
5. If $C'_1 \neq C_1$, set $e \leftarrow s$ and $b \leftarrow 0$.
6. Compute $K = Hash(b, e, C)$.
7. Output session key K.

---

3. Decapsulation Phase --- 3.3 Decode(C, Γ')

The $Decode$ function attempts to decode a syndrome $C \in F_2{}^{mt}$ into an error word e of Hamming weight $wt(e) = t$ such that $C = He$. If it cannot find such a word, it returns failure ($\perp$).

The function uses the private key components:

- Γ' has the form $(g, \alpha'_0, \ldots, \alpha'_{n-1})$.
- g is a monic, irreducible Goppa polynomial of degree t loaded from sk.
- $\alpha'_0, \ldots, \alpha'_{n-1}$ are distinct elements of F_q, which form the support set L. The permutation to generate these is reconstructed from the Benes network control bits stored in sk.

Algorithm:

1. Extend C with k zeros to form $v = (C, 0, \ldots, 0) \in F_2{}^n$.
2. Find the unique codeword $c \in F_2{}^n$ such that $(1) Hc = 0$ and (2) the Hamming distance between c and v is $\leq t$. If no such c exists, return $\perp$.
3. Set $e = v + c$.
4. If $wt(e) = t$ and $C = He$, return e. Otherwise, return $\perp$.

---

3. Decapsulation Phase --- 3.3.1 Finding c (Decoding)

Phase 1: Compute Syndromes and Derive the Key Equation

1. **Syndrome Definition**: Assume the Goppa code is defined by the polynomial $g(x)$ and support set $L = \alpha_0, \ldots, \alpha_{n-1}$. If an error occurs at a set of positions I, the j-th component of the syndrome vector $s = (s_0, \ldots, s_{2t-1})$ is:
   $s_j = \Sigma_{i \in I} \alpha_i{}^j / g(\alpha_i)^2$
   For convenience, this is often expressed as a formal power series called the **syndrome polynomial**:
   $S(z) = \Sigma_{j=0}^{\infty} s_j z^j$

2. **Error-Locator Polynomial** $\sigma(z)$: This polynomial's roots reveal the error locations.
   $\sigma(z) = \Pi_{i \in I}(1 - \alpha_i z)$

   - The reciprocal of its roots, $1/\alpha_i$, are the support elements corresponding to the error locations.
   - Its degree $deg(\sigma(z))$ is the number of errors, $|I|$. Since the system can correct up to t errors, $deg(\sigma(z)) \leq t$.
   - Its constant term is $\sigma(0) = 1$.

3. **Key Equation Derivation**: By multiplying $\sigma(z)$ and $S(z)$, we arrive at the **Key Equation** of decoding theory:
   $\sigma(z)S(z) = \omega(z)$
   where $\omega(z)$ is the **error-evaluator polynomial**. The degree of $\omega(z)$ is less than t.

4. **Meaning (Connection to LFSRs)**: The Key Equation implies that the syndrome sequence s_k can be generated by a Linear Feedback Shift Register (LFSR). For $k \geq L$ (where L is the number of errors), we have:
   $s_k + \sigma_1 s_{k-1} + \ldots + \sigma_L s_{k-L} = 0$
   This means that from term L onwards, each syndrome term can be calculated as a fixed

linear combination of the previous L terms. The coefficients of this linear relationship, $(\sigma_1, \ldots, \sigma_L)$, are precisely the coefficients of the error-locator polynomial $\sigma(z)$. The problem of finding $\sigma(z)$ is equivalent to finding the minimal polynomial of the syndrome sequence.

Phase 2: Solving with the Berlekamp-Massey (BM) Algorithm

The BM algorithm is an efficient method to find the shortest LFSR (and thus the minimal polynomial $\sigma(z)$) for a given sequence $s_0, s_1, \ldots, s_{N-1}$.

Key Variables:

- $C(z)$: The current best guess for $\sigma(z)$.
- L: The length (degree) of the current LFSR/$C(z)$.
- d: The discrepancy (error) when predicting the next sequence element.
- $B(z)$: A "backup" polynomial from the last time L was updated.
- b: The discrepancy associated with $B(z)$.
- m: A counter for steps since the last L update.

Algorithm Iteration (at step N):

1. **Calculate Discrepancy d_N**:
   $d_N = s_N + \Sigma_{i=1}^{L} C_i s_{N-i}$

2. **Check Discrepancy**:

   - **If $d_N = 0$**: The prediction is correct. $C(z)$ is still valid. No changes are needed. Move to the next step $N + 1$.
   - **If $d_N \neq 0$**: Prediction failed. $C(z)$ must be corrected. The core update formula is:
     $C_{new}(z) = C_{old}(z) - d_N \cdot b^{-1} \cdot z \cdot B(z)$
     (Note: the slides use $z^(N - m)$, but a simplified z term is often used in basic descriptions. The core idea is to "patch" the current polynomial using a scaled and shifted version of a previous good polynomial).

3. **Update State Variables (only if $d_N \neq 0$)**:

   - **If $2L \leq N$**: The current length L is "too short" to explain the sequence. We must increase the length.

1. $L_{new} = N + 1 - L_{old}$

2. Update the backup state: The old $C(z)$ and d_N become the new "best snapshot".

   - $B(z) \leftarrow C_{old}(z)$
   - $b \leftarrow d_N$
   - $m \leftarrow N$

- **If** $2L > N$: The length L is still "long enough". We only updated the coefficients of $C(z)$, not its degree. Do not update L, $B(z)$, or b.

Phase 3: Find the Roots of $\sigma(z)$

Once the BM algorithm terminates, we have the error-locator polynomial $\sigma(z)$. The final step is to find the error locations.

1. **Iterate through all possible locations**: For each index j from 0 to $n - 1$:
   a. Get the corresponding support element α_j.
   b. Evaluate $\sigma(z)$ at $z = \alpha_j^{-1}$.

2. **Check the result**:

   - If $\sigma(\alpha_j^{-1}) = 0$, we have found a root. This means an error occurred at position j.
   - If $\sigma(\alpha_j^{-1}) \neq 0$, there is no error at position j.

3. **Record Error Locations**: Create a list of all indices j for which the check in step 2 was true. This list is the set of error positions I, which defines the error vector e.