# RLCE Key Encapsulation Mechanism (RLCE-KEM) Specification

Yongge Wang
UNC Charlotte, USA.
yongge.wang@uncc.edu

September 14, 2017

**Abstract**

This document gives the specification for the RLCE key encapsulation mechanism (KEM).

# Contents

# 1 Executive summary

This section presents an executive summary of the RLCE encryption scheme. The scheme consists of three components: **RLCE.KeySetup**$(n, k, d, t, w)$, **RLCE.Enc**$(G, \mathbf{m}, \mathbf{e})$, and **RLCE.Dec**$(G_s, P_1, P_2, A, \mathbf{c})$.

**RLCE.KeySetup**$(n, k, d, t, w)$. Let $n$, $k$, $d$, $t > 0$, and $w \in \{1, \cdots, n\}$ be given parameters such that $n - k + 1 \geq d$. Generally we have $d \geq 2t + 1$, though it is allowed to have $d < 2t + 1$ in case that efficient list-decoding algorithms exist. Let $G_s$ be a $k \times n$ generator matrix for an $[n, k, d]$ generalized Reed-Solomon code $C$. Let $P_1$ be a randomly chosen $n \times n$ permutation matrix and $G_s P_1 = [\mathbf{g}_0, \cdots, \mathbf{g}_{n-1}]$.

1. Let $\mathbf{r}_0, \mathbf{r}_1, \cdots, \mathbf{r}_{w-1} \in GF(q)^k$ be column vectors drawn uniformly at random and let

$$G_1 = [\mathbf{g}_0, \cdots, \mathbf{g}_{n-w}, \mathbf{r}_0, \cdots, \mathbf{g}_{n-1}, \mathbf{r}_{w-1}, ] \tag{1}$$

   be the $k \times (n + w)$ matrix obtained by inserting column vectors $\mathbf{r}_i$ into $G_s$.

2. Let $A_0, \cdots, A_{w-1} \in GF(q)^{2\times 2}$ be randomly chosen $2 \times 2$ matrices that do not contain zero elements and let

$$A = \text{diag}[I_{n-w}, A_0, \cdots, A_{w-1}] = \begin{bmatrix} I_{n-w} & & & \\ & A_0 & & \\ & & \ddots & \\ & & & A_{w-1} \end{bmatrix} \tag{2}$$

   be an $(n + w) \times (n + w)$ nonsingular matrix.

3. Let $S$ be a randomly chosen dense $k \times k$ nonsingular matrix and $P_2$ be an $(n+w)\times(n+w)$ permutation matrix.

4. The public key is the $k \times (n + w)$ matrix $G = S G_1 A P_2$ and the private key is $(S, G_s, P_1, P_2, A)$. For a systematic RLCE scheme, one chooses $S$ in such a way that $G$ is in echelon form.

**RLCE.Enc**$(G, \mathbf{m}, \mathbf{e})$. For a message to be encrypted, one first pads it using an appropriate padding scheme (which will be described in the next section) and converts it to a row vector $\mathbf{m} \in GF(q)^k$, choose a random row vector $\mathbf{e} = [e_0, \ldots, e_{n+w-1}] \in GF(q)^{n+w}$ with Hamming weight at most $t$. The cipher text is $\mathbf{c} = \mathbf{m}G + \mathbf{e}$.

**RLCE.Dec**$(S, G_s, P_1, P_2, A, \mathbf{c})$. For a received cipher text $\mathbf{c} = [c_0, \ldots, c_{n+w-1}]$, compute

$$\mathbf{c} P_2^{-1} A^{-1} = \mathbf{m} S G_1 + \mathbf{e} P_2^{-1} A^{-1} = [c'_0, \ldots, c'_{n+w-1}].$$

Let $\mathbf{c}' = [c'_0, c'_1, \cdots, c'_{n-w}, c'_{n-w+2}, \cdots, c'_{n+w-2}]$ be the row vector of length $n$ selected from the length $n + w$ row vector $\mathbf{c} P_2^{-1} A^{-1}$. Then $\mathbf{c}' P_1^{-1} = \mathbf{m} S G_s + \mathbf{e}'$ for some error vector $\mathbf{e}' \in GF(q)^n$ where the Hamming weight of $\mathbf{e}' \in GF(q)^n$ is at most $t$. Using an efficient decoding algorithm, one can compute $\mathbf{m}' = \mathbf{m} S$ and $\mathbf{m} = \mathbf{m}' S^{-1}$. Finally, calculate the Hamming weight $wt = \textbf{weight}(\mathbf{c} - \mathbf{m}G)$. If $wt \leq t$ then output $\mathbf{m}$ as the decrypted padded-message. Otherwise, output error.

# 2 Systematic RLCE encryption scheme

To reduce RLCE scheme public key sizes, one can use semantic secure message encoding approach (e.g., an IND-CCA2 padding scheme) so that the public key can be stored in a systematic matrix. For a McEliece encryption scheme over $GF(q)$, one needs to store $k(n-k)$ elements from $GF(q)$ for a systematic public key matrix instead of $nk$ elements from $GF(q)$ for a non-systematic generator matrix public key.

In a systematic RLCE encryption scheme, the decryption could be done more efficiently. In the RLCE decryption process, one recovers $\mathbf{m}SG_s$ from $\mathbf{c}'P_1^{-1} = \mathbf{m}SG_s + \mathbf{e}'$ first. Let $\mathbf{m}SG_sP_1 = (d_0, \cdots, d_{n-1})$ and $\mathbf{c}_d = (d'_0, \cdots, d'_{n+w}) = (d_0, d_1, \cdots, d_{n-w}, \perp, d_{n-w+1}, \perp, \cdots, d_{n-1}, \perp)P_2$ be a length $n+w$ vector. For each $i < k$ such that $d'_i = d_j$ for some $j < n-w$, we have $m_i = d_i$. Let

$$I_R = \{i : m_i \text{ is recovered via } \mathbf{m}SG_s\} \text{ and } \bar{I}_R = \{0, \cdots, k-1\} \setminus I_R.$$

Assume that $|\bar{I}_R| = u$. It suffices to recover the remaining $u$ message symbols $m_i$ with $i \in \bar{I}_R$. In the following paragraphs, we present three approaches to recover these message symbols.

## 2.1 Decoding algorithm 0 for systematic RLCE encryption scheme

In the first approach, one recovers $\mathbf{m}S$ from $\mathbf{m}SG_s$ first. Then one multiplies $\mathbf{m}'$ with the corresponding $u$ columns $S_{\bar{I}_R}$ of the matrix $S^{-1}$ to get $m_i$ with $i \in \bar{I}_R$.

## 2.2 Decoding algorithm 1 for systematic RLCE encryption scheme

Instead of recovering $\mathbf{m}S$ first, one may use public key to recover the remaining message symbols using $\mathbf{m}SG_s$. Let $i_0, \cdots, i_{u-1} \geq k$ be indices such that for each $i_j$, we have $d'_{i_j} = d_i$ for some $i < n-w$. The remaining message symbols with indices in $\bar{I}_R$ could be recovered by solving the linear equation system

$$\mathbf{m}\left[\mathbf{g}_{i_0}, \cdots, \mathbf{g}_{i_{u-1}}\right] = [d'_{i_0}, \cdots, d'_{i_{u-1}}]$$

where $\mathbf{g}_{i_0}, \cdots, \mathbf{g}_{i_{u-1}}$ are the corresponding columns in the public key. Let $P$ be a permutation matrix so that $m_i$ ($i \in I_R$) are the first $k-u$ elements in $\mathbf{m}P$. That is,

$$\mathbf{m}PP^{-1}\left[\mathbf{g}_{i_0}, \cdots, \mathbf{g}_{i_{u-1}}\right] = (\mathbf{m}_{I_r}, \mathbf{m}_{\bar{I}_r})P^{-1}\left[\mathbf{g}_{i_0}, \cdots, \mathbf{g}_{i_{u-1}}\right] = [d'_{i_0}, \cdots, d'_{i_{u-1}}]$$

where $\mathbf{m}_{I_r}$ is the message symbols with indices in $I_R$. Let

$$P^{-1}\left[\mathbf{g}_{i_0}, \cdots, \mathbf{g}_{i_{u-1}}\right] = \begin{pmatrix} V \\ W \end{pmatrix}$$

where $V$ is a $(k-u) \times u$ matrix and $W$ is a $u \times u$ matrix. Then we have

$$\mathbf{m}_{\bar{I}_r}W = [d'_{i_0}, \cdots, d'_{i_{u-1}}] - \mathbf{m}_{I_r}V.$$

Furthermore, one may pre-compute the inverse of $W$ and include $W^{-1}$ in the private key. Then one can recover the message symbols

$$\mathbf{m}_{\bar{I}_r} = \left([d'_{i_0}, \cdots, d'_{i_{u-1}}] - \mathbf{m}_{I_r}V\right)W^{-1}.$$

## 2.3 Decoding algorithm 2 for systematic RLCE encryption scheme

In practice, one may use a larger $I_R$. Recall that in the RLCE decryption process, one recovers $\mathbf{m}SG_s$ from $\mathbf{c}'P_1^{-1} = \mathbf{m}SG_s + \mathbf{e}'$ first. Let $\mathbf{e}'P_1 = (e_0, \cdots, e_{n-1})$ and

$$\mathbf{e}_c = (e'_0, \cdots, e'_{n+w}) = (e_0, e_1, \cdots, e_{n-w}, \bar{e}_{n-w}, e_{n-w+1}, \bar{e}_{n-w+1}, \cdots, e_{n-1}, \bar{e}_{n-1})P_2$$

be a length $n+w$ vector. For each $e_{n-w+i_0} = 0$ ($0 \leq i_0 < w$), if $e'_i = e_{n-w+i_0}$ or $e'_i = \bar{e}_{n-w+i_0}$ for some $i < k$, then with high probability, we have $m_i = c_i$ since matrices $A_i$ do not contain zero elements. Thus $m_i$ might be recovered as $c_i$. Let

$$I_R^a = I_R \cup \{i < k : e'_i = e_{n-w+i_0} \text{ or } e'_i = \bar{e}_{n-w+i_0} \text{ for some } i_0 < w \text{ with } e_{n-w+i_0} = 0\}$$

and $\bar{I}_R^a = \{0, \cdots, k-1\} \setminus I_R^a$. Using the same algorithm as in Section 2.2 with $(I_R, \bar{I}_R)$ replaced by $(I_R^a, \bar{I}_R^a)$, one can then recover message symbols with indices in $\bar{I}_R^a$. With a small probability, the message recovered via $(I_R^a, \bar{I}_R^a)$ might be incorrect. If this happens, one restarts the decoding process using the pair $(I_R, \bar{I}_R)$.

### 2.4 Defeating side-channel attacks

The decoding algorithm 2 described in Section 2.3 might be vulnerable to side-channel attacks. The attacker may generate ciphertexts with appropriately chosen error locations and watch whether the decoding time is significantly long (which means that the message recovered via $(I_R^a, \bar{I}_R^a)$ might be incorrect). This information may be used to recover part of the private permutation $P_2$. If such kind of attacks needs to be defeated, then one should not use the decoding algorithm 2 described in Section 2.3.

For the decoding algorithms 1 and 2, the value $u$ is dependent on the choice of the private permutation $P_2$. Though the leakage of the size of $u$ is not sufficient for the adversary to recover $P_2$ or to carry out other attacks against RLCE scheme, this kind of side-channel information leakage could be easily defeated. Table 1 lists the values of $u_0$ such that, for each scheme, the value of $u$ is smaller than $u_0$ for 90% of the choices of $P_2$ where the RLCE ID is the scheme ID described in Table 2. In this protocol specification and in the reference implementation, $P_2$ should be selected in such a way that $u$ is smaller than the given $u_0$ of Table 1. Furthermore, during the decoding process, one can use dummy computations so that the decoding time is the same as the decoding time for $u = u_0$.

Table 1: The value $u_0$ for RLCE schemes

| RLCE ID | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $u_0$ | 200 | 123 | 303 | 190 | 482 | 309 | 7 |

## 3 Message bandwidth and padding schemes

We first analyze the amount of information that could be encoded within each ciphertext. Let $(n, k, t, w)$ be the system parameters and let $GF(2^m)$ be the underlying finite field. It is noted that the public key is a $k \times (n + w)$ matrix over $GF(2^m)$. There are various approaches to encode messages within the ciphertext. In this protocol specification, we are mainly interested in the following two approaches:

1. **basicEncoding**: Encode information within the vector $\mathbf{m} \in GF(q)^k$ and the ciphertext is $\mathbf{c} = \mathbf{m}G + \mathbf{e}$. In this case, we can encode $\texttt{mLen} = mk$ bits information within each ciphertext.

2. **mediumEncoding**: In addition to **basicEncoding**, further information is encoded in the non-zero entries of $\mathbf{e}$. That is, let $e_{i_1}, \cdots, e_{i_t} \in GF(q) \setminus \{0\}$ be the non-zero elements within $\mathbf{e}$. Then the encoded message within each ciphertext is $m_0, \cdots, m_{k-1}, e_{i_1}, \cdots, e_{i_t} \in GF(q)^{k+t}$. In this case, we can encode $\texttt{mLen} = m(k + t)$ bits information within each ciphertext. Strictly speaking, the encoded information is less than $m(k + t)$ bits since $e_{i_j}$ cannot be zeros. For the mediumEncoding, after one recovers the vector $\mathbf{m}$, one needs to compute $\mathbf{m}G - \mathbf{c}$ to obtain the values of $e_{i_1}, \cdots, e_{i_t}$.

We assume that the message bandwidth is mLen-bits for each ciphertext which is $mt$ or $m(k + t)$. We present a padding scheme for the RLCE encryption scheme. Our padding scheme is adapted from the well analyzed Optimal Asymmetric Encryption Padding (OAEP) for RSA/Rabin encryption schemes and its variants OAEP+. The padding scheme RLCEpad is a two-round Feistel network that is similar to OAEP+. In the following discussions, we assume that messages are binary strings. After padding, it will be converted

to field elements. For a RLCE setup process RLCE.KeySetup$(n, k, d, t, w)$, let $k \times (n+w)$ matrix $G$ be a public key and $(G_s, P_1, P_2, A)$ be a corresponding private key. Assume that the RLCE encryption scheme is over finite field $GF(2^m)$. The padding scheme RLCEpad is based on OAEP+ and proceeds as follows.

RLCEpad(mLen, $k_1, k_2, k_3, t$): Let $k_1, k_2, k_3$ be parameters such that $k_1 + k_2 + k_3 = \left\lceil \frac{\text{mLen}}{8} \right\rceil$, $\min\{k_2, k_3\} \geq \kappa_c$ where $\kappa_c$ is the security parameter. Let $H_1$ be a random oracle that takes any-length inputs and outputs $k_2$ bytes, $H_2$ be a random oracle that takes any-length inputs and outputs $k_1 + k_2$ bytes, and $H_3$ be a random oracle that takes any-length inputs and outputs $k_3$ bytes. Let $\mathbf{m} \in \{0,1\}^{8k_1}$ be a message to be encrypted, $\mathbf{r}_0 \in \{0,1\}^{8k_3 - \nu}$ be a randomly selected sequence, and $\mathbf{r} = \mathbf{r}_0 \| 0^\nu$. We distinguish the following two cases:

- basicEncoding: Select a random $\mathbf{e} \in GF(q)^{n+w,t}$ of weight $t$ and set

$$\mathbf{y} = ((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e})) \oplus H_2(\mathbf{r}, \mathbf{e})) \| \mathbf{r} \oplus H_3(((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e})) \oplus H_2(\mathbf{r}, \mathbf{e}))) \tag{3}$$

  Convert $\mathbf{y}$ to an element $\mathbf{y}_1 \in GF(q)^k$. Let the ciphertext be $\mathbf{c} = \mathbf{y}_1 G + \mathbf{e}$.

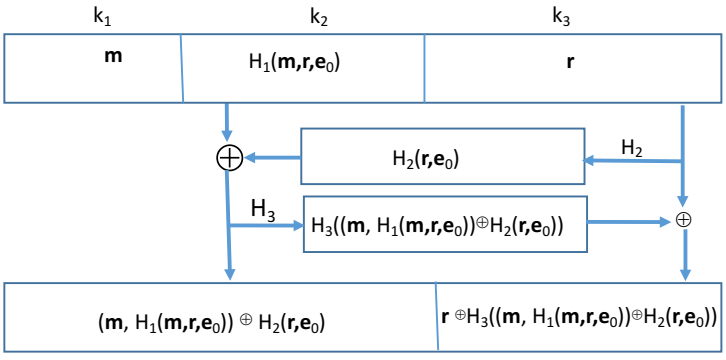
- mediumEncoding: Select random $0 \leq l_0 < l_1 < \cdots < l_{t-1} \leq n + w - 1$ and let $\mathbf{e}_0 = l_0 \| l_1 \cdots \| l_{t-1} \in \{0,1\}^{16t}$. Set

$$\mathbf{y} = ((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e}_0)) \oplus H_2(\mathbf{r}, \mathbf{e}_0)) \| \mathbf{r} \oplus H_3(((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e}_0)) \oplus H_2(\mathbf{r}, \mathbf{e}_0))) \tag{4}$$

  Convert $\mathbf{y}$ to an element $(\mathbf{y}_1, \mathbf{e}_1) \in GF(q)^{k+t}$ where $\mathbf{y}_1 \in GF(q)^k$ and $\mathbf{e}_1 \in GF(q)^t$. Let $\mathbf{e} \in GF(q)^{n+w}$ such that $\mathbf{e}[l_i] = \mathbf{e}_1[i]$ for $0 \leq i < t$ and $\mathbf{e}[j] = 0$ for $j \neq l_i$. Let the ciphertext be $\mathbf{c} = \mathbf{y}_1 G + \mathbf{e}$.

As an example with $\kappa_c = 128$ bits security RLCE scheme $(532, 376, 78)$ over $GF(2^{10})$ in Table 2, we use $k_2 = k_3 = 32$-bytes for both mediumEncoding and basicEncoding. Thus, we can encrypt $k_1 = 504$-bytes of information for mediumEncoding and $k_1 = 406$-bytes of information for basicEncoding per RLCEpad ciphertext. The mediumEncoding based RLCEpad is shown graphically in Figure 1.

Figure 1: mediumEncoding based RLCEpad




**Remark 1:** In RLCE encryption scheme, error position $\mathbf{e}_0 = l_0 \| \cdots \| l_{t-1}$ and error vector $\mathbf{e}$ are used in the RLCEpad process and the message recipient needs to have the exact $\mathbf{e}_0$ and $\mathbf{e}$ for message decoding. In case that the randomly generated error values contain zero field elements, the corresponding error positions will be unavailable for the recipient. To avoid this potential issue, the message encryption process needs to guarantee that error values should never be zero. A simple approach to address this challenge is that, when

calculated error values (using the given random value **r**) contain zero field elements, one revises the random value **r** to a new value and tries the padding approach again. This process continues until all error values are non-zero.

**Remark 2:** In our scheme, we use $k_1 + k_2 + k_3 = \left\lceil \frac{\texttt{mLen}}{8} \right\rceil$. Alternatively, one may use $k_1 + k_2 + k_3 = \left\lfloor \frac{\texttt{mLen}}{8} \right\rfloor$ and adjust the schemes correspondingly.

# 4 RLCE parameters

Table 2 contains the recommended parameters for the RLCE encryption scheme where $\kappa_c$ denotes the corresponding traditional security level and $\kappa_q$ denotes the quantum security level. The boldface security levels $(\kappa_c, \kappa_q)$: $(128, 80), (192, 110), (256, 144)$ are the parameters required by the NIST call for proposals. Schemes 6 is for testing purpose only. In the column `sk`, the top row is the private key size when decoding algorithm 0 or 1 is used (see Section 2) and the bottom row is the private key size when decoding algorithm 2 is used (see Section 2).

Table 2 lists the message bandwidth and message padding scheme parameters for the recommended schemes. For each security strength $(\kappa_c, \kappa_q)$, the even-ID uses the value $w = n - k$ and the odd-ID uses the minimum value $w$ that is required for the corresponding security strength. Thus RLCE schemes with odd-ID are more efficient. In case that $\nu = 8(k_1 + k_2 + k_3) - \texttt{mLen} > 0$, the last $\nu$-bits of the $k_3$-bytes random seed **r** should be set zero and the last $\nu$-bit of the encoded string **y** is discarded. For RLCEpad with $\nu > 0$, the decoding process produces an encoded string **y** with last $\nu$-bits missing. After using $H_3$ to hash the first part of **y** resulting in $k_3$-bytes hash output, one discards the last $\nu$-bits from the hash output and $\oplus$ the remaining $(8k_3 - \nu)$-bits with the second half of **y** to obtain the $(8k_3 - \nu)$-bits of **r** without the $\nu$-bits zero trailer.

Table 2 also lists the parameters for the padding scheme RLCSspad which is discussed in the Appendix if someone has interests in it. The reference implementation implements both RLCEpad and RLCEspad if someone wants to test them.

Table 2: Padding parameters: bE for basicEncoding, mE for mediumEncoding

| ID | $\kappa_c$ | $\kappa_q$ | $n$ | $k$ | $t$ | $w$ | $m$ | sk | cipher | pk | | mLen | RLCEspad | | RLCEpad | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | $k_1(k_2)$ | $k_3$ | $k_1$ | $k_2(k_3)$ |
| 0 | **128** | **80** | 630 | 470 | 80 | 160 | **10** | 310116<br>192029 | 988 | 188001 | bE | 4700 | 146 | 296 | 524 | 32 |
| | | | | | | | | | | | mE | 5500 | 171 | 346 | 624 | 32 |
| 1 | **128** | **80** | 532 | 376 | 78 | 96 | **10** | 179946<br>121666 | 785 | 118441 | bE | 3760 | 117 | 236 | 406 | 32 |
| | | | | | | | | | | | mE | 4540 | 141 | 286 | 504 | 32 |
| 2 | **192** | **110** | 1000 | 764 | 118 | 236 | **10** | 747393<br>457073 | 1545 | 450761 | bE | 7640 | 238 | 479 | 859 | 48 |
| | | | | | | | | | | | mE | 8820 | 275 | 553 | 1007 | 48 |
| 3 | **192** | **110** | 846 | 618 | 114 | 144 | **10** | 440008<br>292461 | 1238 | 287371 | bE | 6180 | 193 | 387 | 677 | 48 |
| | | | | | | | | | | | mE | 7320 | 228 | 459 | 819 | 48 |
| 4 | **256** | **144** | 1360 | 800 | 280 | 560 | **11** | 1773271<br>1241971 | 2640 | 1232001 | bE | 8800 | 275 | 550 | 980 | 60 |
| | | | | | | | | | | | mE | 11880 | 371 | 743 | 1365 | 60 |
| 5 | **256** | **144** | 1160 | 700 | 230 | 311 | **11** | 1048176<br>749801 | 2023 | 742089 | bE | 7700 | 240 | 483 | 843 | 60 |
| | | | | | | | | | | | mE | 10230 | 319 | 641 | 1159 | 60 |
| 6 | 22 | 22 | 40 | 20 | 10 | 5 | **10** | 1059<br>859 | 57 | 626 | bE | 200 | 6 | 13 | 17 | 4 |
| | | | | | | | | | | | mE | 300 | 9 | 20 | 30 | 4 |

# 5 Cryptographic Elements

This section describes the basic cryptographic elements that support RLCE encryption schemes specified in this proposal.

### 5.1 Cryptographic Hash Functions

In RLCE encryption schemes, cryptographic hash functions may be used in key generation and message padding processes. NIST approved hash functions such as FIPS 180-4 [2] or FIPS 202 [3] can be used when a hash function is required. In the reference implementation, SHA-256 and SHA-512 from FIPS 180-4 are used.

### 5.2 Random Bit Generators

Whenever RLCE encryption scheme requires the use of a randomly generated value (for example, for obtaining random matrices), the values shall be generated using an NIST approved random bit generator (RBG), as specified in SP 800-90A rev 1 [1], that provides an appropriate security strength. In the reference implementation, SHA-512 based Hash_DRBG and AES based CTR_DRBG from NIST SP 800-90Ar1 are used.

### 5.3 Nonces

RLCE encryption scheme requires nonce values for private key generation and for each encryption. The nonce values should be generated using a NIST approved random bit generator.

## 6 RLCE protocol specification

Throughout the section, we assume that RLCE scheme is over the finite field $GF(2^m)$ where $m$ is included as one of the key parameters. In the protocol specification, we will use a global variable DECODINGMETHOD to denote the decoding algorithms. DECODINGMETHOD=0 means that the decoding algorithm 0 in Section 2.1 is used. DECODINGMETHOD=1 means that the decoding algorithm 1 in Section Section 2.2 is used. DECODINGMETHOD=2 means that the decoding algorithm 2 in Section Section 2.3 is used.

### 6.1 RLCE Key Pairs

#### 6.1.1 Definition of an RLCE Key Pair

A valid RLCE key pair **shall** consist of an RLCE private key $(n, k, d, t, w; G_s, P_1, P_2, A)$ and an RLCE public key $(n + w, k, t, G)$ as specified in the Executive Summary (Section 1).

#### 6.1.2 RLCE Key Pair Formats

Both the public key and private key for an RLCE scheme are represented as binary strings in the protocol. The following paragraphs describe the format of the binary strings for public and private keys.

Both private key and public keys start with a 1-byte string `paraB` = $b_0b_1b_2b_3b_4b_5b_6b_7$ indicating the RLCE parameters supported. The first four bits `paraB`[0..3] = $b_0b_1b_2b_3$ specify the padding and message encoding schemes used. Specifically, the first four bits are interpreted as follows:

$$b_0b_1b_2b_3 = \begin{cases} 0001 & \text{RLCEpad-mediumEncoding} \\ 0011 & \text{RLCEpad-basicEncoding} \\ 0000 & \text{for RLCEspad-mediumEncoding (discussed in Appendix)} \\ 0010 & \text{for RLCEspad-basicEncoding (discussed in Appendix)} \end{cases}$$

The reference implementation supports the four defined values $b_0b_1b_2b_3$ as above. The last 4 bits `paraB`[4..7] = $b_4b_5b_6b_7$ specifies the RLCE parameter ID (= $0, \cdots, 14$) in Table 2. For example, $b_4b_5b_6b_7 = 0101$ represents the RLCE scheme with ID=5 ($\kappa_c = 256$ and $\kappa_q = 144$).

**Public key**. Let $G = [I_k, G_E]$ be the public key in echelon format where $I_k$ is the $k \times k$ identity matrix and $G_E$ is a $k \times (n + w - k)$ matrix. Let $z_1, \cdots, z_{k\times(n+w-k)} \in GF(2^m)$ be a list of elements of $G_E$, where the first $n + w - k$ elements consist of the first row of $G_E$, the second $n + w - k$ elements consist of the second row of $G_E$, and so on. Then the public key is the following binary string of $1 + \lceil mk(n + w - k)/8 \rceil$ bytes:

$$\texttt{paraB}||z_1||\cdots||z_{k\times(n+w-k)}||0^\nu$$

where $\nu = 8 * \lceil mk(n + w - k)/8 \rceil - mk(n + w - k)$. As an example for the RLCE parameter ID=0, we have $n = 630, k = 470, w = 160, m = 10$. Thus the public key size is 188001 bytes.

**Private key**. The private key consists of one byte paraB and the following fields:

- The inverse permutation matrix $P_1^{-1}$ is represented by a list of 2-byte integers: $p_{1,1}, \cdots, p_{1,n}$. Let $\texttt{perm}_1 = p_{1,1}||\cdots||p_{1,n}$ be a $2n$-bytes binary string.
- The inverse permutation matrix $P_2^{-1}$ is represented by a list of 2-byte integers: $p_{2,1}, \cdots, p_{2,n+w}$. Let $\texttt{perm}_2 = p_{2,1}||\cdots||p_{2,n+w}$ be a $2(n + w)$-bytes binary string.
- The inverse matrix $A^{-1} = \text{diag}(A_0^{-1}, \cdots, A_{w-1}^{-1})$ consists of $4w$ field elements. For the decryption process, only the first column of each $A_i^{-1}$ is required. Thus $A^{-1}$ is represented by $2w$ field elements $a_1, \cdots, a_{2w} \in GF(2^m)$.
- If DECODINGMETHOD=0, then include the $u$ columns $S_{\bar{I}_R}$ from $S^{-1}$. To ensure that the private key has a constant size, we extend $S_{\bar{I}_R}$ to a $k \times (u_0 + 1)$ matrix where $u_0$ is defined in Table 1. Specifically, let the key include the elements $x_1, \cdots, x_{k(u_0+1)}$ of the extended $S_{\bar{I}_R}$.
- If DECODINGMETHOD=1, then construct a $k\times(u+1)$ matrix $X$ where the upper left $u\times u$ sub-matrix is the inverse matrix $W^{-1}$, the lower left $(k - u) \times u$ sub-matrix is the matrix $V$, and the last column of $X$ is used to denote the corresponding column indices $i_0, \cdots, i_{u-1}$. In particular, for $j < u$ we have $X[j][u] = i$ if $e_i = e'_{i_j}$ where $V, W, e_i, e'_{i_j}$ are as defined in Section 2.2. To ensure that the private key has a constant size, we extend $X$ to a $k \times (u_0 + 1)$ matrix where $u_0$ is defined in Table 1. Specifically, let the key include the elements $x_1, \cdots, x_{k(u_0+1)}$ of the extended $X$.
- The $n$ GRS inverse coefficients: $v_0, \cdots, v_{n-1} \in GF(2^m)$.
- The public key $G_E$ consists of $k(n + w - k)$ field elements: $z_1, \cdots, z_{k\times(n+w-k)} \in GF(2^m)$. The public key is included to speed up the decryption process.

In a summary, we have

- If DECODINGMETHOD=0 or 1, the private key consists of a $4n+2w+1$-byte string $\texttt{paraB}||\texttt{perm}_1||\texttt{perm}_2$ and $2w + k(u_0 + 1) + n + k(n + w - k) = n + k + 2w + kn + kw + ku_0 - k^2$ field elements. That is, a private key is represented by the following binary string of $4n + 2w + 1 + \lceil m(n + k + 2w + kn + kw + ku_0 - k^2)/8 \rceil$ bytes:

  $$\texttt{paraB}||\texttt{perm}_1||\texttt{perm}_2||a_1||\cdots||a_{2w}||x_1||\cdots||x_{k(u_0+1)}||v_1||\cdots||v_n||z_1||\cdots||z_{k\times(n+w-k)}||0^\nu$$

  where $\nu = 8\lceil m(n + k + 2w + kn + kw + ku_0 - k^2)/8 \rceil - m(n + k + 2w + kn + kw + ku_0 - k^2)$.

- If DECODINGMETHOD=2, the private key consists of a $4n+2w+1$-byte string $\texttt{paraB}||\texttt{perm}_1||\texttt{perm}_2$ and $2w + n + k(n + w - k) = 2w + n + kn + kw - k^2$ field elements. That is, a private key is represented by the following binary string of $4n + 2w + 1 + \lceil m(2w + n + kn + kw - k^2)/8 \rceil$ bytes:

  $$\texttt{paraB}||\texttt{perm}_1||\texttt{perm}_2||a_1||\cdots||a_{2w}||v_1||\cdots||v_n||z_1||\cdots||z_{k\times(n+w-k)}||0^u$$

  where $u = 8\lceil m(2w + n + kn + kw - k^2)/8 \rceil - m(2w + n + kn + kw - k^2)$.

As an example for the RLCE parameter ID=0, we have $n = 630, k = 470, w = 160, m = 10$. Thus the private key is 192029 bytes for DECODINGMETHOD= 2.

### 6.1.3 RLCE Key Pair Generators

The key pairs for RLCE encryption schemes specified in this Recommendation **shall** be generated using an NIST approved random bit generator (RBG), as specified in SP 800-90A rev 1 [1], that provides an appropriate security strength. In this reference implementation, Hash_DRBG from NIST SP 800-90Ar1 is used.
`RLCE_key_setup` is the key pair generator that produces a valid RLCE key pair.
**Function call**: `RLCE_key_setup(paraB, entropy, nonce)`
**Input**:

1. `paraB` = $b_0b_1b_2b_3b_4b_5b_6b_7$ is a byte string defined in Section 6.1.2, where
   (a) `paraB`[0..3] = $b_0b_1b_2b_3$ specifies the padding and message encoding schemes used.
   (b) `paraB`[4..7] = $b_4b_5b_6b_7$ specifies the RLCE scheme ID defined in Table 2.
2. `entropy` is a binary string of at least $128 + \kappa_c$ bits that provides the entropy for the key generation process.
3. `nonce` is an optional binary string. If present, it is recommended to be at least $\kappa_c/2$ bits.

**Process**:

1. Checks the value `paraB`:
   (a) The integer defined by the first four bits `paraB`[0..3] = $b_0b_1b_2b_3$ equals to 1 or 3 (or equals to 0 or 2 as specified in Appendix).
   (b) The integer defined by the last four bits `paraB`[4..7] = $b_4b_5b_6b_7$ lies in the interval $[0, 14]$.
2. Retrieves parameters $(\kappa_c, \kappa_q, n, k, t, w, m, \texttt{mLen}, k_1, k_2, k_3)$ corresponding to `paraB` from Table 2.
3. Checks that `entropy` is at least $128 + \kappa_c$ bits.
4. If `nonce` = `NULL`, then let `nonce` = `0x5e7d69e187577b0433eee8eab9f77731`.
5. Let $\pi(x)$ be the primitive polynomial of degree $m$ in section 7.1.
6. Let $\alpha = 0^{m-2}10$ be a root of $\pi(x)$ that generates $GF(2^m)$.
7. Let $g(x) = \prod_{i=1}^{n-k}(x - \alpha^i) = g_0 + g_1x + \cdots + g_{n-k}x^{n-k} \in GF(2^m)[x]$ where $g_0, \cdots, g_{n-k} \in GF(2^m)$.
8. Let $G_0$ be the $k \times n$ matrix with $G_0[i, i + j] = g_j$ for $0 \le i \le k - 1$ and $0 \le j \le n - k$. Let $G_0[i, i + j] = 0$ for all other $i, j$. That is,

$$G_0 = \begin{pmatrix} g_0 & g_1 & \cdots & g_{n-k} & 0 & \cdots & 0 \\ 0 & g_0 & \cdots & g_{n-k-1} & g_{n-k} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{n-2k+1} & g_{n-2k+2} & \cdots & g_{n-k} \end{pmatrix}$$

9. Let `pers="PostQuantumCryptoRLCEversion2017"`.

10. Let addS="GRSbasedPostQuantumENCSchemeRLCE".

11. Let nRE $=n + (4 + k)w + 25$.

12. Let nRB $=\lceil (m \times \text{nRE})/8 \rceil + 4n + 2w$.

13. Let randBytes = hash_DRBG(entropy, nonce, pers, addS, nRB,$\kappa_c$) where hash_DRBG is defined in Section 7.5 and randBytes is an array of nRB bytes.

14. Let randE = B2FE(randBytes$[0..\lceil (m \times \text{nRE})/8 \rceil - 1], m)$ where B2FE is defined in Section 7.10 and randE is a list of nRE field elements from $GF(2^m)$.

15. Let $\langle v_0, v_1, \cdots, v_{n-1} \rangle$ be the first $n$ non-zero elements from randE$[0]$, randE$[n + 4]$ and let the $k \times n$ matrix $G'_0 = G_0 \begin{pmatrix} v_0 & 0 & \cdots & 0 \\ 0 & v_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_{n-1} \end{pmatrix}$ be the generator matrix for a generalized Reed-Solomon code. For the matrix $G'_0$, it is sufficient to store the tuple grs $= \langle v_0^{-1}, \cdots, v_{n-1}^{-1} \rangle$ in the private key.

16. Let $A$ = getMatrixA(randE$[n + 5, \ldots, n + 4w + 24], w)$ be an $2w \times 2w$ matrix where getMatrixA() is defined in Section 7.6 and let $A^{-1}$ be the matrix inverse of $A$.

17. Let $R$ = getRandomMatrix(randE$[n + 4w + 25, \ldots, n + (4 + k)w + 24], k, w)$ be a $k \times w$ matrix where getRandomMatrix() is defined in Section 7.7.

18. Let $P_1$ = getPermutation(randBytes$[\lceil (m \times \text{nRE})/8 \rceil..\lceil (m \times \text{nRE})/8 \rceil + 2n - 3], n, n - 1)$ be a permutation of the numbers $0, \cdots, n - 1$ where getPermutation is defined in Section 7.8.

19. Let $P_1^{-1}$ = permu_inv$(P_1)$ be the inverse permutation of $P_1$ where permu_inv() is defined in Section 7.8.

20. Let $G'$ = matrix_col_permutation$(G'_0, P_1)$ where matrix_col_permutation is defined in Section 7.9.

21. Let $G_1$ = matrix_join$(G', R)$ where matrix_join is defined in Section 7.9.

22. Let $G_2$ = matrix_mul_A$(G_1, A)$ where matrix_mul_A is defined in Section 7.9.

23. Let pctr $= 0$. While pctr $\geq 0$ do

    (a) Let $P_2$ = getPermutation(randBytes$[\lceil (m \times \text{nRE})/8 \rceil + 2n - 2 + \text{pctr}..\lceil (m \times \text{nRE})/8 \rceil + 4n + 2w - 5 + \text{pctr}], n + w, n + w - 1)$ be a permutation of the numbers $0, \cdots, n + w - 1$ where getPermutation is defined in Section 7.8.

    (b) Let $0 \leq I_0 < I_1 < \cdots < I_{u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[I_i] \geq n-w$ for all $0 \leq i \leq u - 1$.

    (c) If $u \leq u_0$ then let pctr $= -1$, otherwise, let pctr = pctr + 1, where $u_0$ is defined in Table 1.

24. Let $P_2^{-1}$ = permu_inv$(P_2)$ be the inverse permutation of $P_2$ where permu_inv is defined in Section 7.8.

25. Let $G_3$ = matrix_col_permutation$(G_2, P_2)$ where matrix_col_permutation is defined in Section 7.9.

26. If DECODINGMETHOD=0, then let $S$ be a $k \times k$ matrix such that $G = SG_3$ is in the echelon format and let $X = \texttt{selectScolumn}(S, P_2, u_0)$ which is defined in Section 7.11.

27. If DECODINGMETHOD= 1, let $X = \texttt{precompute}(P_2, G, u_0)$ which is defined in Section 7.12.

28. If DECODINGMETHOD=0 or 1, then convert the private key $(X, \texttt{grs}, P_1^{-1}, P_2^{-1}, A^{-1}, G)$ and the public key $G$ to binary strings `privateKey` and `publicKey` respectively according to the steps in Section 6.1.2.

29. If DECODINGMETHOD=2, then convert the private key $(\texttt{grs}, P_1^{-1}, P_2^{-1}, A^{-1}, G)$ and the public key $G$ to binary strings `privateKey` and `publicKey` respectively according to the steps in Section 6.1.2.

**Output**: The private key `privateKey` and the public key `publicKey`.

## 6.2 RLCE Encryption

RLCE encryption scheme requires random bit generators and mask generation functions. In this reference implementation, Hash_DRBG and CTR_DRBG from NIST SP 800-90Ar1 are used for random bit generation and mask generation function from NIST SP 800-56 r1 is used.
**Function call**: `RLCE_encrypt`(`pk`, `entropy`, `nonce`, `msg`)
**Input**:

1. `pk` is the public key
2. `entropy` is a binary string of at least $128 + \kappa_c$ bits that provides the entropy for the encryption process.
3. `nonce` is an optional binary string. If present, it is recommended to be at least $\kappa_c/2$ bits.
4. `msg` is the message to be encrypted (which is a binary string).

**Process**:

1. Recovers `paraB` and $G$ from `pk` where $G$ is a $k \times (n + w)$ matrix and $\texttt{paraB} = b_0b_1b_2b_3b_4b_5b_6b_7$ is a byte defined in Section 6.1.2 with
   (a) $\texttt{paraB}[0..3] = b_0b_1b_2b_3$ specifies the padding and message encoding schemes used.
   (b) $\texttt{paraB}[4..7] = b_4b_5b_6b_7$ specifies the RLCE scheme ID defined in Table 2.
2. Checks the value `paraB`:
   (a) The integer defined by the first four bits $\texttt{paraB}[0..3] = b_0b_1b_2b_3$ equals to 1 or 3 (or equals to 0 or 2 as specified in Appendix).
   (b) The integer defined by the last four bits $\texttt{paraB}[4..7] = b_4b_5b_6b_7$ lies in the interval $[0, 14]$.
3. Retrieves parameters $(\kappa_c, \kappa_q, n, k, t, w, m, \texttt{mLen}, k_1, k_2, k_3)$ corresponding to `paraB` from Table 2.
4. Checks that `entropy` is at least $128 + \kappa_c$ bits.
5. Checks that `msg` is $k_1$ bytes.
6. Let `pers` = "`PQENCRYPTIONRLCEver1`".
7. Let `addS` = "`GRSbasedPQEncryption`"||`0x00`.

8. If $\kappa_c = 256$, then let nonce = nonce||"RLCEencNonce" and initiate DRBG state by running

$$\texttt{DRBG\_state} = \texttt{hash\_DRBG\_Instantiate\_algorithm}(\texttt{entropy}, \texttt{nonce}, \texttt{pers}, \kappa_c)$$

as specified in Section 10.1.2 of SP 800-90A rev 1 where the underlying hash algorithm is SHA-512.

9. If $\kappa_c \leq 192$, then initiate DRBG state by running

$$\texttt{DRBG\_state} = \texttt{CTR\_DRBG\_Instantiate\_algorithm}(\texttt{entropy}, \texttt{pers}, \kappa_c)$$

as specified in Section 10.2.3.1 of SP 800-90A rev 1.

10. Let $\texttt{nRB0} = k_3 + 2t$.

11. If "paraB[0..3] = 0010 or paraB[0..3] = 0011", let $\texttt{nRB1} = \lceil m(t+10)/8 \rceil$. Otherwise, let nRB1 = 0.

12. Lets ctr = 0 and repeat = 1.

13. While repeat > 0 do

    (a) If $\kappa_c = 256$, then run Hash_DRBG_Generate_algorithm(DRBG_state, $8 * (\texttt{nRB0} + \texttt{nRB1})$, addS) which is specified in Section 10.1.1.4 of SP 800-90A rev 1. This process will return an array of nRB0 + nRB1-byte randBytes and return a new DRBG state DRBG_state.

    (b) If $\kappa_c \leq 192$, then run CTR_DRBG_Generate_algorithm(DRBG_state, $8 * (\texttt{nRB0} + \texttt{nRB1})$, addS) which is specified in Section 10.2.1.5.1 of SP 800-90A rev 1. This process will return an array of nRB0 + nRB1-byte randBytes and return a new DRBG state DRBG_state.

    (c) let ctr = ctr + 1 and addS = "GRSbasedPQEncryption"||ctrB where ctrB is one byte.

    (d) Let padrand = randBytes$[0..k_3 - 1]$.

    (e) Let $P$ = getPermutation(randBytes$[k_3..\texttt{nRB0} - 1], n + w, t)$ be a permutation of the numbers $0, \cdots, n + w - 1$ where getPermutation is defined in Section 7.8.

    (f) Let errLocation = $\{P[0], \cdots, P[t-1]\}$.

    (g) Let e0 = $l_0 || l_1 || \cdots || l_{t-1}$ be a binary string of $2t$-bytes where $l_0 < l_1 < \cdots < l_{t-1}$ is a list of all elements in errLocation and $l_i$ is two-byte for each $0 \leq i < t$.

    (h) If "paraB[0..3] = 0010 or paraB[0..3] = 0011", then

        i. let randE = B2FE(randBytes$[\texttt{nRB0}..\texttt{nRB0} + \texttt{nRBB1} - 1], m)$ where B2FE is defined in Section 7.10 and randE is a list of $t + 10$ field elements from $GF(2^m)$,

        ii. let errValue= $[v_0, \cdots, v_{t-1}]$ be the first $t$ non-zero field elements in randE,

        iii. let eV = $v_0 || \cdots || v_{t-1}$ be a $2t$-bytes string.

        iv. let e0 = e0||eV.

    (i) Distinguish the following two cases:

        i. paraB[0..3] = 0000 or 0010: Let paddedMSG=RLCEspad(msg,pk,padrand,e0) be a $(k_1 + k_2 + k_3)$-bytes string where RLCEspad is defined in the Appendix Section 8.2.

        ii. paraB[0..3] = 0001 or 0011: Let paddedMSG=RLCEpad(msg,pk,padrand,e0) be a $(k_1 + k_2 + k_3)$-bytes string which RLCEspad is defined in Section 7.13.

    (j) Let FEMSG=B2FE(paddedMSG,m) where B2FE is defined in Section 7.10. Note that if $\nu = 8(k_1 + k_2 + k_3) - \texttt{mLen} > 0$, then the last $\nu$-bits information of paddedMSG is lost during this conversion.

(k) If paraB[0..3] = 0000 or paraB[0..3] = 0001, let errValue[0..$t-1$] = FEMSG[$k$..$k+t-1$].

(l) If errValue[$i$] = 0 for some $0 \leq i < t$, then let repeat = 1. Otherwise, let repeat = 0.

14. Let cipher[0..$n+w-1$] = FEMSG[0..$k-1$] $\times G$.

15. Let cipher[$l_i$] = cipher[$l_i$] + errValue[$i$] for all $0 \leq i < t$.

16. Let c = FE2B(cipher, m) where FE2B is defined in Section 7.10.

**Output**: a binary string $c$.

## 6.3 RLCE Decryption

RLCE_decrypt is the function that produces a valid RLCE plaintext from a ciphertext.
**Function call**: RLCE_decrypt(sk, c)
**Input**:

1. sk is the private key

2. c is a binary string.

**Process**:

1. If DECODINGMETHOD=0 or 1, then recover paraB and $(X, \mathtt{grs}, P_1^{-1}, P_2^{-1}, A^{-1}, G)$ from sk.

2. If DECODINGMETHOD=2, then recover paraB and $(\mathtt{grs}, P_1^{-1}, P_2^{-1}, A^{-1}, G)$ from sk.

3. Check the value paraB:

   (a) The integer defined by the first four bits paraB[0..3] = $b_0b_1b_2b_3$ equals to 1 or 3 (or equals to 0 or 2 as specified in Appendix).

   (b) The integer defined by the last four bits paraB[4..7] = $b_4b_5b_6b_7$ lies in the interval $[0, 14]$.

4. Retrieve parameters $(\kappa_c, \kappa_q, n, k, t, w, m, \mathtt{mLen}, k_1, k_2, k_3)$ corresponding to paraB from Table 2.

5. Let cipher = B2FE(c, m) where B2FE is defined in Section 7.10 and cipher is a list of field elements.

6. Let cipher′ = matrix_col_permutation(cipher, $P_2^{-1}$) where matrix_col_permutation is defined in Section 7.9 and cipher is considered as an $1 \times (n + w)$ matrix.

7. Let $C_1$ = matrix_mul_A(cipher′, $A^{-1}$) where matrix_mul_A is defined in Section 7.9.

8. Assume that $C_1 = (c_0, \cdots, c_{n+w-1})$. Let $C_2 = (c_0, \cdots, c_{n-w-1}, c_{n-w}, c_{n-w+2}, \cdots, c_{n+w-2})$ be a length $n$ array.

9. Let $C_3$ = matrix_col_permutation($C_2$, $P_1^{-1}$) be a $1 \times n$ matrix.

10. Assume that grs = $\langle v_0^{-1}, \cdots, v_{n-1}^{-1} \rangle$. Let $C_4[0][i] = v_i^{-1} C_3[0][i]$ for $i = 0, \cdots, n-1$.

11. Let TBD = $(0, \cdots, 0, C_4[0][0], \cdots, C_4[0][n-1]) \in GF(2^m)^{2^m-1}$ be an array of $2^m - 1$ field elements obtained by adding a prefix of $2^m - 1 - n$ zero elements to $C_4$.

12. Let code′ = rs_decode(TBD, $G_s$, $m$) where rs_decode is defined in Section 7.14. It is noted that code′ is a list of $2^m - 1$ field elements that contains a prefix of $2^m - 1 - n$ zero elements.

13. Let $\texttt{code} \in GF(2^m)^n$ be obtained from $\texttt{code}'$ by removing the $(2^m - 1 - n)$-zero prefix.

14. Let $\texttt{code}[i] = \texttt{code}[i] \cdot v_i$.

15. Let $\texttt{cB4A} = \texttt{matrix\_col\_permutation}(\texttt{code}, P_1)$ be a $1 \times n$ matrix.

16. If DECODINGMETHOD=0 or 1:

    (a) Let $0 \leq I_0 < I_1 < \ldots < I_{u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[I_i] \geq n-w$ for all $0 \leq i \leq u-1$.

    (b) Let $0 \leq J_0 < J_1 < \ldots < J_{k-u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[J_i] < n-w$ for all $0 \leq i \leq k-u-1$.

    (c) Let $\texttt{msg}[J_i] = \texttt{cB4A}[P_2[J_i]]$ for all $0 \leq i \leq k-u-1$.

17. If DECODINGMETHOD=0:

    (a) Let $g(x)$ be the degree $n-k$ Reed-Solomon code generator polynomial and let $q(x) = \frac{\texttt{code}'}{g(x)}$ where $\texttt{code}'$ is treated as a polynomial and $q(x)$ is a polynomial with the first $2^m - 1 - n$ coefficients being zero. That is, $q(x) = q_{2^m-1-n}x^{2^m-1-n} + \cdots, q_{2^m-1-n+k-1}x^{2^m-1-n+k-1}$.

    (b) Assume that $X = S_{\bar{I}_R}$.

    (c) Let $(\texttt{msg}[I_0], \cdots, \texttt{msg}[I_{u-1}]) = (q_{2^m-1-n}, \cdots, q_{2^m-1-n+k-1}) \cdot S_{\bar{I}_R}$.

18. If DECODINGMETHOD=1:

    (a) Assume that $X = \begin{pmatrix} W^{-1} & U^T \\ V & 0 \end{pmatrix}$.

    (b) Let $(\texttt{msg}[I_0], \cdots, \texttt{msg}[I_{u-1}])$ be

    $$((\texttt{msg}[J_0], \cdots, \texttt{msg}[J_{k-u-1}])\, V + (\texttt{cB4A}[U[0]], \cdots, \texttt{cB4A}[U[u-1]]))\, W^{-1}.$$

19. If DECODINGMETHOD=2:

    (a) Let $0 \leq J_0 < J_1 < \ldots < J_{k-u-1} < k$ be a list of all integers in the interval $[0, k-1]$ such that $P_2[J_i] < n-w$ or "$P_2[J_i] \geq n-w$ and $\texttt{cB4A}\left[n-w+\left\lfloor \frac{(P_2[J_i]-(n-w))}{2} \right\rfloor\right] = C_2[J_i]$".

    (b) Let $0 \leq I_0 < I_1 < \ldots < I_{u-1} < k$ be a list of integers in $\{0, \cdots, k\} \setminus \{J_i : 0 \leq i \leq k-u-1\}$.

    (c) Let $k \leq T_0 < T_1 < \cdots < T_{u-1} < n+w$ be the first $u$ integers with $P_2[T_i] < n-w$.

    (d) Let $W$ be a $u \times u$ matrix such that $W[i][j] = G[I_i][T_j]$ for $0 \leq i, j \leq u-1$.

    (e) Let $V$ be a $(k-u)\times u$ matrix such that $V[i][j] = G[J_i][T_j]$ for all $0 \leq i \leq k-u-1$ and $0 \leq j \leq u-1$.

    (f) If $P_2[J_i] < n-w$ then let $\texttt{msg}[J_i] = \texttt{cB4A}[P_2[J_i]]$. Otherwise, let $\texttt{msg}[J_i] = \texttt{cipher}[J_i]$.

    (g) Let $(\texttt{msg}[I_0], \cdots, \texttt{msg}[I_{u-1}])$ be

    $$((\texttt{msg}[J_0], \cdots, \texttt{msg}[J_{k-u-1}])\, V + (\texttt{cB4A}[P_2[T_0]], \cdots, \texttt{cB4A}[P_2[T_{u-1}]]))\, W^{-1}.$$

20. Let $\texttt{cipher}'' = \texttt{msg} \times G \in GF(2^m)^{n+w}$.

21. If $\texttt{weight}(\texttt{cipher}'' - \texttt{cipher}) > t$ and DECODINGMETHOD=2, then let $X = \texttt{precompute}(P_2, G)$ which is defined in Section 7.12 and re-calculate $\texttt{msg}$ using DECODINGMETHOD 1 in step 18.

22. Let $l_0 < l_1 < \cdots < l_{t-1} < n+w$ such that $\texttt{cipher}''[l_i] \neq \texttt{cipher}[l_i]$ for $0 \leq i < t$.

23. Let $\texttt{e0} = l_0 \| l_1 \| \cdots \| l_{t-1}$ be a binary string of $2t$-bytes.

24. If $\texttt{paraB}[0..3] = 0000$ or $\texttt{paraB}[0..3] = 0001$, then

    (a) let $\texttt{FE} \in GF(2^m)^{k+t}$ with $\texttt{FE}[i] = \texttt{msg}[i]$ for $0 \leq i < k$ and $\texttt{FE}[k+j] = \texttt{cipher}''[l_j] - \texttt{cipher}[l_j]$ for $0 \leq j < t$.

    (b) let `paddedMSG=FE2B(FE,m)` where `FE2B` is defined in Section 7.10 and `paddedMSG` is a byte string of $\lceil \frac{m(k+t)}{8} \rceil$ bytes. Note that if $\nu = 8(k_1 + k_2 + k_3) - m(k+t) > 0$, then the last $\nu$-bits of `paddedMSG` should be replaced with 0s.

25. If $\texttt{paraB}[0..3] = 0010$ or $\texttt{paraB}[0..3] = 0011$, then

    (a) let $\texttt{FE} \in GF(2^m)^k$ with $\texttt{FE}[i] = \texttt{msg}[i]$ for $0 \leq i < k$.

    (b) let `paddedMSG=FE2B(FE,m)` where `FE2B` is defined in Section 7.10 and `paddedMSG` is a byte string of $\lceil \frac{mk}{8} \rceil$ bytes. Note that if $\nu = 8(k_1 + k_2 + k_3) - mk > 0$, then the last $\nu$-bits of `paddedMSG` should be replaced with 0s.

    (c) let $\texttt{eV} = e_0 \| \cdots \| e_{t-1}$ be a $2t$-bytes string where $e_i = \texttt{cipher}''[l_i] - \texttt{cipher}[l_i]$ for $0 \leq i < t$.

26. Distinguish the following cases:

    (a) $\texttt{paraB}[0..3] = 0001$: Let `message=RLCEpadDecode(paddedMSG,sk,e0)` which is a $k_1$ bytes string and `RLCEpadDecode` is defined in Section 7.13.

    (b) $\texttt{paraB}[0..3] = 0011$: Let `message=RLCEpadDecode(paddedMSG,sk,e0||eV)` which is a $k_1$ bytes string and `RLCEspadDecode` is defined in Section 7.13.

    (c) $\texttt{paraB}[0..3] = 0000$: Let `message=RLCEspadDecode(paddedMSG,sk,e0)` which is a $k_1$ bytes string and `RLCEspadDecode` is defined in the Appendix Section 8.2.

    (d) $\texttt{paraB}[0..3] = 0010$: Let `message=RLCEspadDecode(paddedMSG,sk,e0||eV)` which is a $k_1$ bytes string and `RLCEspadDecode` is defined in the Appendix Section 8.2.

**Output**: The $k_1$-bytes string `message`.

## 6.4 RLCE Key Encapsulation Mechanism (KEM)

The `RLCE_encrypt(pk, entropy, nonce, msg)` and `RLCE_decrypt(sk, cipher)` function calls in Sections 6.2 and 6.3 can encrypt and decrypt $k_1$ bytes information each time. For key encapsulation mechanisms, the encapsulated secret is generally less than $k_1$ bytes. The following function calls are for encapsulating and decapsulating secrets.
**Function call**: `kem_encapsulate(pk, entropy, nonce, ss, sslen)`.
**Input**:

1. `pk` is the public key
2. `entropy` is a binary string of at least $128 + \kappa_c$ bits that provides the entropy for the encryption process.
3. `nonce` is an optional binary string. If present, it is recommended to be at least $\kappa_c/2$ bits.
4. `ss` is a binary string of `sslen` bytes.

**Process**:

1. Let msg = ss||0x00||$\cdots$||0x00 be a $k_1$ byte string obtained by adding $k_1$ − sslen zero bytes at the end of ss.

2. Let c = RLCE_encrypt(pk, entropy, nonce, msg) where RLCE_encrypt is defined in Section 6.2.

**Output**: The byte string c.

The following function call is for decapsulating a secret.
**Function call**: kem_decapsulate(sk, c, sslen).
**Input**:

1. sk is the private key

2. c is cipher text that encapsulates the secret

3. sslen is byte-length of the secret

**Process**:

1. Let msg =RLCE_decrypt(sk, cipher) be a $k_1$ byte string where RLCE_decrypt is defined in Section 6.3.

2. Let ss = msg[0]||$\cdots$msg[sslen − 1]

**Output**: The byte string ss.

# 7 Auxiliary Functions

## 7.1 Primitive polynomials

The following is a list of primitive polynomials that are used for finite fields $GF(2^m)$:

$$GF(2^{10}): \quad \pi(x) = x^{10} + x^3 + 1$$
$$GF(2^{11}): \quad \pi(x) = x^{11} + x^2 + 1$$

## 7.2 Get short integers

The function getShortIntegers returns a list of 16-bit integers.
**Function call**: getShortIntegers(randBytes, numI)
**Input**:

1. randBytes is an array of 2 × numI bytes.

2. numI is a positive integer number.

**Process**:

1. Let numBytes = 2 × numI;

2. Let shortIntegers[$i$] = randBytes[$2i$] × $2^8$ + randBytes[$2i + 1$] for $0 \leq i <$ numI

**Output**: A list of numI unsigned short integers shortIntegers.

### 7.3 I2BS and BS2I

We use the Integer-to-Byte-String (I2BS) and Byte-String-to-Integer (BS2I) conversion from NIST SP800-56 r1 Appendix B.
**I2BS Input**:

1. A non-negative integer $X$.
2. A positive integer $n$.

**I2BS Process**:

1. Find an $n$-byte string $S[0..n-1]$ such that $X = S[0] \cdot 2^{8(n-1)} + \cdots + S[n-1] \cdot 2^0$.

**I2BS Output**: $S[0..n-1]$.

**BS2I Input**:

1. An $n$-byte string $S[0..n-1]$.

**BS2I Process**:

1. Let $X = S[0] \cdot 2^{8(n-1)} + \cdots + S[n-1] \cdot 2^0$.

**BS2I Output**: The integer $X$.

### 7.4 The Mask Generation Function (MGF)

We use the Mask Generation Function (MGF) from NIST SP800-56 r1 Section 7.2.2.2. Our MGF is based on a NIST approved hash function such as SHA-512. The MGF is used in RLCE padding schemes. Let `hash` be an approved hash function, and let `hashLen` denote the length of the hash function output in bytes.
**Function call**: `RLCE_MGF(mgfSeed, maskLen)`
**Input**:

1. `mgfSeed`: a byte string from which the mask is generated.
2. `maskLen`: the intended length of the mask (in bytes).

**Process**:

1. Set $T$ = NULL, the empty string.
2. For `counter` from 0 to $\lceil \texttt{maskLen}/\texttt{hashLen} \rceil - 1$, do the following:
   (a) Let $D$ = `I2BS(counter, 4)` where `I2BS` is defined Section 7.3.
   (b) Let $T = T \| \texttt{hash}(\texttt{mgfSeed} \| D)$.
3. Output the leftmost `maskLen` bytes of $T$ as the byte string `mask`.

**Output**: The byte string `mask` (of length `maskLen` bytes).

### 7.5 NIST SP800-90Ar1 DRBG

The function hash_DRBG(entropy, nonce, pers_string, add_string, numBytes,$\kappa_c$) returns numBytes-bytes and should be defined using the mechanism Hash_DRBG specified in Section 10.1.1 of SP 800-90A rev 1 [1].
**Function call**: hash_DRBG(entropy, nonce, pers_string, add_string, numBytes, $\kappa_c$)
**Input**:

1. entropy, nonce, pers_string, add_string are binary strings.
2. numBytes is a positive integer.
3. $\kappa_c$ is the security strength.

**Process**:

1. Use SHA-512 as the underlying hash function.
2. Let state=Hash_DRBG_Instantiate_algorithm(entropy,nonce,pers_string,$\kappa_c$) as specified in Section 10.1.1.2 of SP 800-90A rev 1.
3. Let returned_bytes=NULL
4. while |returned_bytes|< numBytes do
   (a) Let nBytes = min{$2^{16}$, numBytes − |eturned_bytes|}.
   (b) Let new_bytes = Hash_DRBG_Generate_algorithm(state, nBytes, add_string) as specified in Section 10.1.1.4 of SP 800-90A rev 1.
   (c) returned_bytes = returned_bytes||new_bytes

**Output**: returned_bytes which is numBytes bytes long.

### 7.6 Get a random matrix $A$

The function getMatrixA returns a $2w \times 2w$ matrix $A$.
**Function call**: getMatrixA(randE$[0, \ldots, 4w + 19], w$)
**Input**:

1. randE$[0, \ldots, 4w + 19]$ is an array of $4w + 20$ field elements.
2. $w$ is a positive integer.

**Process**:

1. Let rE$[0, \ldots, d]$ be a list of non-zero elements from randE$[0, \ldots, 4w + 19]$.
2. Let $i = 0$ and $j = 0$.
3. while $i < w$ do
   (a) Let $det = 0$.
   (b) while "det = 0" do
      i. Let det = rE$[j] \times$ rE$[j + 3]$ − rE$[j + 1] \times$ rE$[j + 2]$

ii. If det = 0 then $j = j + 1$.

(c) Let $A_i$ be the $2 \times 2$ matrix defined by

$$A_i[0][0] = \texttt{rE}[j]$$
$$A_i[0][1] = \texttt{rE}[j+1]$$
$$A_i[1][0] = \texttt{rE}[j+2]$$
$$A_i[1][1] = \texttt{rE}[j+3]$$

(d) Let $j = j + 4$ and $i = i + 1$.

**Output**: A $2w \times 2w$ matrix $A = \texttt{diag}(A_0, \cdots, A_{w-1})$.

### 7.7 Get a random matrix

The function `getRandomMatrix` returns a random $k \times w$ matrix.
**Function call**: `getRandomMatrix`(`randE`$[0, \ldots, kw-1], k, w$)
**Input**:

1. `randE`$[0, \ldots, kw-1]$ is an array of $kw$ field elements;
2. $k, w$ are positive integer numbers.

**Process**:

1. For $0 \le i \le k-1$ and $0 \le j \le w-1$, let $R[i][j] = \texttt{randE}[jk+i]$.

**Output**: A $k \times w$ matrix $R$.

### 7.8 Get a permutation and a permutation inverse

A permutation for the numbers $0, \cdots, u-1$ shall be obtained using a randomized algorithm. The following process is based on Fisher-Yates shuffle algorithm that is named "Algorithm P" in in Knuth's "The Art of Computer Programming".
**Function call**: `getPermutation`(`randBytes`$[0..2v-1], u, v$)
**Input**:

1. `randBytes`$[0..2v-1]$ is an array of $2v$ bytes.
2. $u$ is a positive integer number.
3. $v \le u-1$ is a positive integer number.

**Process**:

1. Let `shortIntegers=getShortIntegers`(`randBytes`, $v$) where `getShortIntegers` is defined in Section 7.2 and `shortIntegers` is an array of $v$ 16-bit-integers.
2. For $i = 0, \cdots, u-1$, let $P[i] = i$.
3. For $i$ from 0 to $v-1$ do
   (a) $j = \texttt{shortIntegers}[i] \% (u-i)$.
   (b) $j = j + i$.

(c) `tmp` $=P[i]$.

(d) $P[i] = P[j]$.

(e) $P[j] =$ `tmp`.

**Output**: A permutation $P$ of the numbers $0, \cdots, u-1$. Note that only the first $t$ positions are randomly permuted in $P$.

The function `permu_inv` returns the inverse of a permutation.
**Function call**: `permu_inv`($P$)
**Input**: $P$ is a permutation of the numbers $0, \cdots, u-1$.
**Process**: For $i = 0, \cdots, u-1$, let $P^{-1}[P[i]] = i$.
**Output**: The inverse permutation $P^{-1}$ of $P$.

## 7.9 Matrix operations

The function `matrix_col_permutation` permutes the columns of a matrix.
**Function call**: `matrix_col_permutation`($M, P$).
**Input**:

1. $M$ is a $k \times u$ matrix;
2. $P$ is a permutation of the numbers $0, \cdots, u-1$.

**Process**:

1. Let $M_1 = M$.
2. Let $M[i][j] = M_1[i][P[j]]$ for all $0 \leq i < k, 0 \leq j < u$.

**Output**: The $k \times u$ matrix $M$.

The function `matrix_join` combines two matrices into one matrix.
**Function call**: `matrix_join`($G, R$).
**Input**:

1. $G$ is a $k \times n$ matrix;
2. $R$ is a $k \times w$ matrix;

**Process**:

1. Let $d = n - w$.
2. For $0 \leq i < k$ and $0 \leq j < d$, let $G_1[i][j] = G[i][j]$.
3. For $0 \leq i < k$ and $0 \leq j < w$, let $G_1[i][d + 2j] = G[i][d + j]$ and $G_1[i][d + 2j + 1] = R[i][d + j]$.

**Output**: The $k \times (n + w)$ matrix $G_1$.

The function `matrix_mul_A` multiplies a matrix with a matrix $A$ from the right hand side.
**Function call**: `matrix_mul_A`($G, A$).
**Input**:

1. $G$ is a $k \times (n + w)$ matrix.
2. $A = \texttt{diag}(A_0, \cdots, A_{w-1})$ is a $2w \times 2w$ matrix where $A_i$ is a $2 \times 2$ matrix for $i = 0, \cdots, w - 1$.

**Process**:

1. Let $I_{n-w}$ be the $(n - w) \times (n - w)$ identity matrix.
2. Let $G_1 = G \times \texttt{diag}(I_{n-w}, A_0, \cdots, A_{w-1})$.

**Output**: The $k \times (n + w)$ matrix $G_1$.

## 7.10 Byte array and field element array conversions

The section describes the conversion function from a byte array to a field element array and the conversion function from a field element array to a byte array.
**Function call**: B2FE(BYTES,m).
**Input**:

1. BYTES is a length Blen bytes array.
2. $m$ is a positive integer.

**Process**:

1. Let $\texttt{FElen} = \left\lfloor \frac{8 \times \texttt{Blen}}{m} \right\rfloor$.
2. Let $f_0 || \cdots || f_{\texttt{FElen}-1}$ be a prefix of BYTES where $f_i$ is $m$-bits long for $0 \leq i < \texttt{FElen}$.

**Output**: FElen field elements $f_0, \cdots, f_{\texttt{FElen}-1}$.

**Function call**: FE2B(FE, m).
**Input**:

1. FE is a length FElen array of field elements in $GF(2^m)$.
2. $m$ is a positive integer.

**Process**:

1. Let $\texttt{Blen} = \left\lceil \frac{m \times \texttt{FElen}}{8} \right\rceil$.
2. Let BYTES be a binary string such that $\texttt{FE} = f_0 || \cdots || f_{\texttt{FElen}-1}$ is a prefix of BYTES.

**Output**: The byte array BYTES of Blen bytes.

## 7.11 Select columns from $S^{-1}$

The function $X = \texttt{selectScolumn}(S, P_2, u_0)$ outputs a matrix $X$.
**Function call**: $\texttt{precompute}(S, P_2, u_0)$
**Input**:

1. $P_2$ is a permutation of numbers $0, \cdots, n + w - 1$
2. $S$ is a $k \times k$ matrix

3. $u_0 < k$ is an integer

**Process**:

1. Let $S^{-1}$ be the inverse of $S$.
2. Let $0 \leq I_0 < I_1 < \ldots < I_{u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[I_i] \geq n - w$ for all $0 \leq i \leq u-1$. If $u > u_0$ return an error.
3. Let $k \leq T_0 < T_1 < \cdots < T_{u-1} < n + w$ be the first $u$ integers such that $P_2[T_i] < n - w$ for all $0 \leq i \leq u-1$.
4. Let $X$ be a $k \times (u_0 + 1)$ matrix such that $X[i][j] = G[i][T_j]$ for all $0 \leq i \leq k-1$ and $0 \leq j \leq u-1$.

**Output**: $X$.

## 7.12 Pre-computation for private key

The function $X = \texttt{precompute}(P_2, G, u_0)$ outputs a matrix $X$.
**Function call**: $\texttt{precompute}(P_2, G)$
**Input**:

1. $P_2$ is a permutation of numbers $0, \cdots, n + w - 1$
2. $G$ is a $k \times (n + w)$ matrix
3. $u_0 < k$ is an integer

**Process**:

1. Let $0 \leq I_0 < I_1 < \ldots < I_{u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[I_i] \geq n - w$ for all $0 \leq i \leq u-1$. If $u > u_0$ return an error.
2. Let $0 \leq J_0 < J_1 < \ldots < J_{k-u-1} < k$ be a list of all integers in the interval $[0, k-1]$ with $P_2[J_i] < n - w$ for all $0 \leq i \leq k-u-1$.
3. Let $k \leq T_0 < T_1 < \cdots < T_{u-1} < n + w$ be the first $u$ integers such that $P_2[T_i] < n - w$ for all $0 \leq i \leq u-1$.
4. Let $W$ be a $u \times u$ matrix such that $W[i][j] = G[I_i][T_j]$ for all $0 \leq i, j \leq u-1$.
5. Let $V$ be a $(k-u) \times u$ matrix such that $V[i][j] = G[J_i][T_j]$ for all $0 \leq i \leq k-u-1$ and $0 \leq j \leq u-1$.
6. Let $U = (P_2[T_0], \cdots, P_2[T_{u-1}])$ be a $1 \times u$ matrix.

**Output**: The $k \times (u_0 + 1)$ matrix $X = \begin{pmatrix} W^{-1} & U^T & 0 \\ V & 0 & 0 \end{pmatrix}$ where columns of 0 are added at the right hand side of the matrix in case $u < u_0$.

### 7.13 RLCEpad and RLCEpadDecode

The function RLCEpad outputs a padded byte string.
**Function call**: RLCEpad(msg,pk,padrand,e0)
**Input**:

1. msg is a binary string to be padded.
2. pk is the public key
3. padrand and e0 are binary strings.

**Process**:

1. Recover $k_1, k_2, k_3$ from pk.
2. Check that msg is $k_1$ bytes and padrand is $k_3$ bytes.
3. Calculate $\nu = 8 * (k_1 + k_2 + k_3) - \mathtt{mLen}$ and let $\mathtt{mask} = 1^{8-\nu}0^{\nu}$, where mLen is defined in Table 2 according to the parameters in pk.
4. Set $\mathtt{padrand}[k_3 - 1] = \mathtt{padrand}[k_3 - 1]\&\mathtt{mask}$. This sets last $\nu$-bits of padrand to zero.
5. Set re0 = padrand||e0 and mre0 = msg||re0.
6. Let h1mre0 = RLCE_MGF(mre0,$k_2$) where RLCE_MGF is defined in Section 7.4.
7. Let h2re0 = RLCE_MGF(re0, $\mathtt{k}_1 + \mathtt{k}_2$).
8. Set paddedMSG1 = (msg||h1mre0) $\oplus$ h2re0.
9. Let h3mh1 = RLCE_MGF(paddedMSG1, $\mathtt{k}_3$).
10. Set paddedMSG = paddedMSG1||(padrand $\oplus$ h3mh1).

**Output**: the $k_1 + k_2 + k_3$ bytes string paddedMSG.

The function RLCEpadDecode decodes a padded byte string.
**Function call**: RLCEpadDecode(paddedMSG,sk, e0)
**Input**:

1. paddedMSG is a binary string to be padded.
2. sk is the private key
3. e0 is binary strings.

**Process**:

1. Recover $k_1, k_2, k_3$ from sk.
2. Check that paddedMSG is $k_1 + k_2 + k_3$ bytes.
3. Set paddedMSG1 = paddedMSG$[0..k_1 + k_2 - 1]$.
4. Let h3mh1 = RLCE_MGF(paddedMSG1,$k_3$) where RLCE_MGF is defined in Section 7.4.

5. Calculate $\nu = 8 * (k_1 + k_2 + k_3) - \texttt{mLen}$ and let $\texttt{mask} = 1^{8-\nu}0^{\nu}$, where mLen is defined in Table 2 according to the parameters in pk.

6. Set $\texttt{padrand} = (\texttt{paddedMSG}[k_1 + k_2] \cdots \texttt{paddedMSG}[k_1 + k_2 + k_3 - 1]) \oplus \texttt{h3mh1}$.

7. Set $\texttt{padrand}[k_3 - 1] = \texttt{padrand}[k_3 - 1]\&\texttt{mask}$. This sets last $\nu$-bits of padrand to zero.

8. Set $\texttt{re0} = \texttt{padrand}||\texttt{e0}$.

9. Let $\texttt{h2re0} = \texttt{RLCE\_MGF}(\texttt{re0}, \texttt{k}_1 + \texttt{k}_2)$

10. Set $\texttt{paddedMSG}[0] \cdots \texttt{paddedMSG}[k_1 + k_2 - 1] = (\texttt{paddedMSG}[0] \cdots \texttt{paddedMSG}[k_1 + k_2 - 1]) \oplus \texttt{h2re0}$.

11. Set $\texttt{msg} = \texttt{paddedMSG}[0] \cdots \texttt{paddedMSG}[k_1 - 1]$.

12. Set $\texttt{mre0} = \texttt{msg}||\texttt{re0}$.

13. Let $\texttt{h1mre0} = \texttt{RLCE\_MGF}(\texttt{mre0}, \texttt{k}_2)$

14. If $\texttt{paddedMSG}[k_1..k_1 + k_2 - 1] \neq \texttt{h1mre0}$ return error.

**Output**: the $k_1$ bytes string msg.

### 7.14 Reed-Solomon decoding

The function rs_decode removes errors within a received Reed-Solomon code that contains errors.
**Function call**: $\texttt{rs\_decode}(\texttt{TBD}, G_s, m)$.
**Input**:

1. TBD is a list of $2^m - 1$ elements from the finite field $GF(2^m)$.

2. $G_s$ is a $k \times n$ generator matrix.

**Process**:

1. In case that $h = 2^m - 1 - n > 0$, extend $G_s$ to a $(k + h) \times (h + n)$ generator matrix $G_s$ by adding zero rows on top on $G_s$ and adding zero columns on the left hand side of $G_s$.

2. Using one of the well known Reed-Solomon decoding algorithms such as Berlekamp-Massey decoder, Euclidean decoder, or Berlekamp-Welch decoder to output a codeword msg which is a list of $2^m - 1$ elements from the finite field $GF(2^m)$.

**Output**: the codeword msg.

## 8 Apendix

### 8.1 RLCE message padding scheme RLCEspad

In addition to the RLCE message padding scheme RLCEpad, one mahy also use the padding scheme RLCEspad for RLCE encryption schemes. RLCEspad is a one-round Feistel network that is based on SAEP+. RLCEspad could be used to encrypt short messages (e.g., mLen/4-bits) and is sufficient for applications such as symmetric key transportation using the RLCE public key encryption scheme. Assume that RLCE is over the finite field $GF(2^m)$. The RLCEspad proceeds as follows.

RLCEspad(mLen, $k_1, k_2, k_3$): Let $k_1, k_2, k_3$ be parameters such that $k_1 + k_2 + k_3 = \left\lceil \frac{\texttt{mLen}}{8} \right\rceil$, $k_1 + k_2 < k_3$, and $8k_1 \leq \texttt{mLen}/4$. Let $\nu = 8(k_1 + k_2 + k_3) - \texttt{mLen}$. Let $H_1$ be a random oracle that takes any-length inputs and outputs $k_2$-bytes and let $H_2$ be a random oracle that takes any-length inputs and outputs $(k_1 + k_2)$-bytes. Let $\mathbf{m} \in \{0,1\}^{8k_1}$ be a message to be encrypted, $\mathbf{r}_0 \in \{0,1\}^{8k_3-\nu}$ be a randomly selected sequence, and $\mathbf{r} = \mathbf{r}_0 \| 0^\nu$. We distinguish the following two cases:

- basicEncoding: Select a random $\mathbf{e} \in GF(q)^{n+w}$ of weight $t$ and set

$$\mathbf{y} = ((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e})) \oplus H_2(\mathbf{r}, \mathbf{e})) \| \mathbf{r}. \tag{5}$$

  Convert $\mathbf{y}$ to an element $\mathbf{y}_1 \in GF(q)^k$. Let the ciphertext be $\mathbf{c} = \mathbf{y}_1 G + \mathbf{e}$.

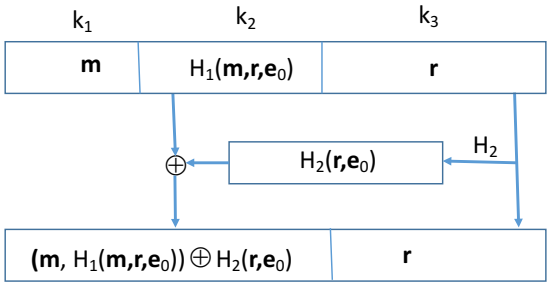
- mediumEncoding: Select random $0 \leq l_0 < l_1 < \cdots < l_{t-1} < n+w$ and let $\mathbf{e}_0 = l_0 \| l_1 \cdots \| l_{t-1} \in \{0,1\}^{16t}$. Set

$$\mathbf{y} = ((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e}_0)) \oplus H_2(\mathbf{r}, \mathbf{e}_0)) \| \mathbf{r}. \tag{6}$$

  Convert $\mathbf{y}$ to an element $(\mathbf{y}_1, \mathbf{e}_1) \in GF(q)^{k+t}$ where $\mathbf{y}_1 \in GF(q)^k$ and $\mathbf{e}_1 \in GF(q)^t$. Let $\mathbf{e} \in GF(q)^{n+w}$ such that $\mathbf{e}[l_i] = \mathbf{e}_1[i]$ for $0 \leq i < t$ and $\mathbf{e}[j] = 0$ for $j \neq l_i$. Let the ciphertext be $\mathbf{c} = \mathbf{y}_1 G + \mathbf{e}$.

The mediumEncoding based RLCEspad is shown graphically in Figure 2.

Figure 2: mediumEncoding based RLCEspad




As an example with $\kappa_c = 128$ bits security RLCE scheme $(630, 470, 80)$ over $GF(2^{10})$ in Table 2, we use $k_1 = k_2 = 171$-bytes for mediumEncoding. Thus, we can encrypt $k_1 = 171$-bytes of information for mediumEncoding per RLCEspad ciphertext.

## 8.2 RLCEspad and RLCEspadDecode functional call specifications

The function RLCEspad outputs a padded byte string.
**Function call**: `RLCEspad(msg,pk,padrand,e0)`
**Input**:

1. `msg` is a binary string to be padded.
2. `pk` is the public key
3. `padrand` and `e0` are binary strings.

**Process**:

1. Recover $k_1, k_2, k_3$ from pk.
2. Check that msg is $k_1$ bytes and padrand is $k_3$ bytes.
3. Calculate $\nu = 8 * (k_1 + k_2 + k_3) - \mathtt{mLen}$ and let $\mathtt{mask} = 1^{8-\nu}0^{\nu}$, where mLen is defined in Table 2 according to the parameters in pk.
4. Set $\mathtt{padrand}[k_3 - 1] = \mathtt{padrand}[k_3 - 1]\&\mathtt{mask}$. This sets last $\nu$-bits of padrand to zero.
5. Set re0 = padrand||e0.
6. Set mre0 = msg||re0.
7. Let h1mre0 = RLCE_MGF(mre0,$k_2$) where RLCE_MGF is defined in Section 7.4.
8. Set $\mathtt{h2re0} = \mathtt{RLCE\_MGF}(\mathtt{re0}, \mathtt{k}_1 + \mathtt{k}_2)$
9. Set paddedMSG = ((msg||h1mre0) $\oplus$ h2re0)||padrand.

**Output**: the $k_1 + k_2 + k_3$ bytes string paddedMSG.

The function RLCEspadDecode decodes a padded byte string.
**Function call**: RLCEspadDecode(paddedMSG,sk, e0)
**Input**:

1. paddedMSG is a binary string to be padded.
2. sk is the private key
3. e0 is binary strings.

**Process**:

1. Recover $k_1, k_2, k_3$ from sk.
2. Check that paddedMSG is $k_1 + k_2 + k_3$ bytes.
3. Calculate $\nu = 8 * (k_1 + k_2 + k_3) - \mathtt{mLen}$ and let $\mathtt{mask} = 1^{8-\nu}0^{\nu}$, where mLen is defined in Table 2 according to the parameters in pk.
4. Set $\mathtt{padrand} = \mathtt{paddedMSG}[k_1 + k_2] \cdots \mathtt{paddedMSG}[k_1 + k_2 + k_3 - 1]$.
5. Set $\mathtt{padrand}[k_3 - 1] = \mathtt{padrand}[k_3 - 1]\&\mathtt{mask}$. This sets last $\nu$-bits of padrand to zero.
6. Set re0 = padrand||e0.
7. Sets h2re0 = RLCE_MGF(re0,$k_1 + k_2$) where RLCE_MGF is defined in Section 7.4.
8. Sets $\mathtt{paddedMSG}[0] \cdots \mathtt{paddedMSG}[k_1 + k_2 - 1] = (\mathtt{paddedMSG}[0] \cdots \mathtt{paddedMSG}[k_1 + k_2 - 1]) \oplus \mathtt{h2re0}$.
9. Sets $\mathtt{msg} = \mathtt{paddedMSG}[0] \cdots \mathtt{paddedMSG}[k_1 - 1]$.
10. Sets mre0 = msg||re0.
11. Sets $\mathtt{h1mre0} = \mathtt{RLCE\_MGF}(\mathtt{mre0}, \mathtt{k}_2)$
12. If $\mathtt{paddedMSG}[k_1..k_1 + k_2 - 1] \neq \mathtt{h1mre0}$ return error.

**Output**: the $k_1$ bytes string msg.

## References


[1] E. Barker and J. Kelsey. *NIST SP 800-90A Revision 1: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. Available at http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf*. NIST, 2015.

[2] NIST. Fips pub 180-4. *Secure hash standard (SHS), Available at: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf*, 2012.

[3] NIST. Fips pub 202. sha-3 standard: Permutation-based hash and extendable-output functions. *Information Technology Laboratory, National Institute of Standards and Technology. Available at: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf*, 2015.