RLCE Key Encapsulation Mechanism (RLCE-KEM) Specification

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

February 7, 2019

Abstract

This document specifies the key encapsulation mechanism RLCE-KEM, where RLCE is a random linear code based McEliece encryption scheme. Practical RLCE parameters for the security levels of 128, 192, and 256 bits are recommended. It is noted that RLCE schemes with these parameters have the corresponding quantum security levels of 80, 110, and 144 respectively.

This document also contains four **informative appendices**. The first appendix contains the security analysis of the RLCE scheme. The second appendix contains the performance analysis of the RLCE scheme. The third appendix contains analysis and comparison of different algorithms for decoding Reed-Solomon codes. The fourth appendix discusses semantics of global variables used in the submitted optimized implementation.

Contents

1 The RLCE scheme

Since McEliece encryption scheme [22] was introduced more than thirty years ago, it has withstood many attacks and still remains unbroken for general cases. It has been considered as one of the candidates for post-quantum cryptography since it is immune to existing quantum computer algorithm attacks. The original McEliece cryptography system is based on binary Goppa codes. Several variants have been introduced to replace Goppa codes in the McEliece encryption scheme though most of them have been broken. Up to the writing of this paper, secure McEliece encryption schemes include MDPC/LDPC code based McEliece encryption schemes [1, 23], Wang's RLCE [32], and the original binary Goppa code based McEliece encryption scheme. Though no essential attacks have been identified for Goppa code and MDPC/LDPC based McEliece encryption schemes yet, the security of these schemes depends on certain structures of the underlying linear codes. The advantage of RLCE encryption scheme is that its security does not depend on any specific structure of underlying linear codes. It is believed that RLCE security depends on the NP-hardness of decoding random linear codes.

Unless specified otherwise, we will use $q = 2^m$ and our discussion will be based on the field $GF(q)$ throughout this paper. Bold face letters such as $\mathbf{a}, \mathbf{b}, \mathbf{e}, \mathbf{f}, \mathbf{g}$ are used to denote row or column vectors over $GF(q)$. It should be clear from the context whether a specific bold face letter represents a row vector or a column vector. RLCE scheme is defined over any efficiently decodable linear code. In this specification, we recommend generalized Reed-Solomon code based RLCE scheme. Let $k < n < q$. The generalized Reed-Solomon code $\mathrm{GRS}_k(\mathbf{x}, \mathbf{y})$ of dimension k is defined as

$$\mathrm{GRS}_k(\mathbf{x}, \mathbf{y}) = \{(y_0 p(x_0), \cdots, y_{n-1} p(x_{n-1})) : p(x) \in GF(q)[x], \deg(p) < k\}$$

where $\mathbf{x} = (x_0, \cdots, x_{n-1}) \in GF(q)^n$ is an n-tuple of distinct elements and $\mathbf{y} = (y_0, \cdots, y_{n-1}) \in GF(q)^n$ is an n-tuple of nonzero (not necessarily distinct) elements.

The RLCE scheme consists of three components: **RLCE.KeySetup**(n, k, d, t, w), **RLCE.Enc**$(G, \mathbf{m}, \mathbf{e})$, and **RLCE.Dec**$(S, G_s, P_1, P_2, A, \mathbf{c})$.

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 $A_0 = \begin{pmatrix} a_{0,00} & a_{0,01} \\ a_{0,10} & a_{0,11} \end{pmatrix}, \cdots, A_{w-1} = \begin{pmatrix} a_{w-1,00} & a_{w-1,01} \\ a_{w-1,10} & a_{w-1,11} \end{pmatrix} \in GF(q)^{2\times 2}$ be non-singular 2×2 matrices chosen uniformly at random such that $a_{i,00}a_{i,01}a_{i,10}a_{i,11} \neq 0$ for all $i = 0, \cdots, w - 1$, and let

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

   be an $(n + w) \times (n + w)$ non-singular matrix.

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

4. The public key is the $k \times (n+w)$ matrix $G = SG_1AP_2$ 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 row vector $\mathbf{m} \in GF(q)^k$ and a row vector $\mathbf{e} \in GF(q)^{n+w}$ with Hamming weight $\mathbf{wt}(\mathbf{e}) \le t$, let the cipher text $\mathbf{c} = \mathbf{m}G + \mathbf{e}$.

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

$$\mathbf{c}P_2^{-1}A^{-1} = \mathbf{m}SG_1 + \mathbf{e}P_2^{-1}A^{-1} = [c'_0, \ldots, c'_{n+w-1}].$$

Let $\mathbf{c}' = [c'_0, c'_1, \cdots, c'_{n-w}, c'_{n-w+2}, c'_{n-w+4}, \cdots, c'_{n+w-2}]$ be the row vector of length n selected from the length $n+w$ row vector $\mathbf{c}P_2^{-1}A^{-1}$. Then $\mathbf{c}'P_1^{-1} = \mathbf{m}SG_s + \mathbf{e}'$ 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 recover $\mathbf{m}SG_s$ from $\mathbf{c}'P_1^{-1}$. Let D be a $k \times k$ inverse matrix of SG'_s where G'_s is the first k columns of G_s. Then $\mathbf{m} = \mathbf{c}_1 D$ where $\mathbf{c}_1$ is the first k elements of $\mathbf{m}SG_s$. Let $\mathbf{e} = \mathbf{c} - \mathbf{m}G$. If $\mathbf{wt}(\mathbf{e}) \le t$, then output $(\mathbf{m}, \mathbf{e})$. 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 $k(n-k)$ field elements for a systematic public key matrix instead of nk field elements 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 first recovers $\mathbf{m}SG_s$ from $\mathbf{c}'P_1^{-1} = \mathbf{m}SG_s + \mathbf{e}'$ using an efficient decoding algorithm for the underlying linear code. 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 $m_i = d_j$ where $\mathbf{m} = (m_0, \cdots, m_{k-1})$. 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 $i \in \bar{I}_R$.

Let $i_0, \cdots, i_{u-1} \ge k$ 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

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

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 the recovered message symbols m_i ($i \in I_R$) are the first $k-u$ elements in $\mathbf{m}P$. That is,

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

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

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

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 remaining message symbols

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

Private key for a systematic RLCE: With the decoding algorithm in the preceding paragraphs, we can include the matrices V and W^{-1} in the private key instead of including the matrix S. Thus, the private key for a systematic RLCE scheme is the tuple (X, G_s, P_1, P_2, A) where $X = (V, W^{-1})$.

Defeating side-channel attacks. For the decoding algorithm in the preceding paragraph, the value u is dependent on the choice of the private permutation P_2. Though the leakage of value u (e.g., by estimating the decryption time) 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 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/optimized 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
u_0	200	303	482

Other decoding approaches. There are other decoding approaches for systematic RLCE schemes. The reader is referred to Wang [33] for more details.

3 RLCE parameters

In the Appendix Section 9, we will carry out security analysis on the RLCE schemes. Based on these analysis, RLCE parameters for various security strength are recommended in Table 2. In particular, the recommendation takes into account of the conditions for avoiding improved classical and quantum information set decoding, the conditions for avoiding Sidelnikov-Shestakov attacks, the conditions for filtration attacks (with or without brute force), the cost of recovering McEliece encryption scheme secret keys from the public keys, and the cost of recovering plaintext messages from ciphertexts. In Table 2, κ_c denotes the conventional security strength and κ_q denotes the quantum security strength. For example, $\kappa_c = 128$ means an equivalent security of AES-128. For each security strength (κ_c, κ_q), the even-ID uses the value $w = n-k$ and the odd-ID uses the minimum value w required for the corresponding security strength. The recommended parameters is based on GRS codes over $GF(q)$ where $q = 2^{\lceil \log_2 n\rceil}$. For GRS codes, the BCH-style construction requires $n = q-1$. However, GRS codes could be shortened to length $n < q-1$ codes by interpreting the unused $q-1-n$ information symbols as zeros.

The following is a comparison of the parameters in Table 2 against binary Goppa code based McEliece encryption scheme parameters from [6].

1. For the security strength 128, binary Goppa code uses $n = 2960, k = 2288, t = 57$ and the public key size is 188KB while RLCE has a public key size of 188001 bytes (that is, 183KB).

Table 2: RLCE Parameters

ID	κ_c	κ_q	n	k	t	w	m	sk	cipher	pk	mLen	RLCEpad	
												k_1	$k_2(k_3)$
0	**128**	**80**	630	470	80	160	10	310116	988	188001	5500	624	32
1	**192**	**110**	1000	764	118	236	10	747393	1545	450761	8820	1007	48
2	**256**	**144**	1360	800	280	560	11	1773271	2640	1232001	11880	1365	60
3	22	22	40	20	10	5	10	1059	57	626	300	30	4

2. For the security strength 192, binary Goppa code uses $n = 4624, k = 3468, t = 97$ and the public key size is 490KB while RLCE has a public key size of 450761 bytes (that is, 440KB).

3. For the security strength 256, binary Goppa code uses $n = 6624, k = 5129, t = 117$ and the public key size is 900KB while RLCE has a public key size of 1232001 bytes (that is, 1203KB).

4 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 G is a $k \times (n + w)$ matrix over $GF(2^m)$ and the ciphertext is $\mathbf{c} = \mathbf{m}G + \mathbf{e}$. There are various approaches to encode messages within the ciphertext. This document specifies the **mediumEncoding** methods. For other encoding methods, the reader is referred to Wang [33] for more details.

- **mediumEncoding**: For a ciphertext $\mathbf{c}$, the encoded message is $m_0, \cdots, m_{k-1}, e_{i_1}, \cdots, e_{i_t} \in GF(q)^{k+t}$ where $\mathbf{m} = (m_0, \cdots, m_{k-1})$ and $e_{i_1}, \cdots, e_{i_t} \in GF(q) \setminus \{0\}$ are the non-zero elements within $\mathbf{e}$.

With this message encoding approach, 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.

We next 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+. Specifically, the RLCEpad converts a $8k_1$-bits input message $\mathbf{m}$ to a pair $(\mathbf{y}_1, \mathbf{e})$ where $\mathbf{y}_1 \in GF(q)^k$ and $\mathbf{e} \in GF(q)^{n+k}$ with $\mathbf{wt}(\mathbf{e}) = t$.

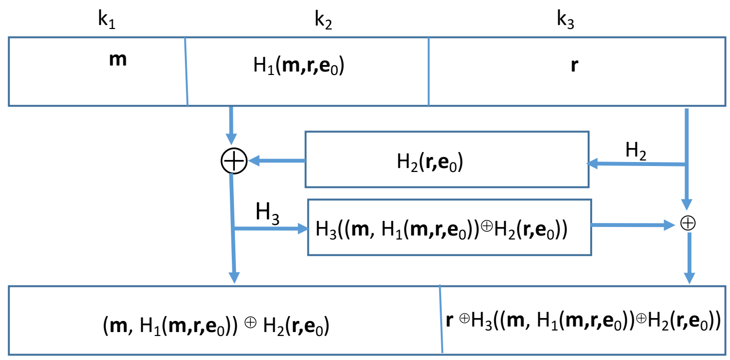
$\texttt{RLCEpad}(\texttt{mLen}, k_1, k_2, k_3, t, \mathbf{m}, \mathbf{r})$: Let k_1, k_2, k_3 be parameters such that $k_1 + k_2 + k_3 = \left\lceil \frac{\texttt{mLen}}{8} \right\rceil$ and $\nu = 8(k_1 + k_2 + k_3) - \texttt{mLen}$. Let H_1, H_2, and H_3 be random oracles that take arbitrary-length binary input strings and output k_2-bytes, $(k_1 + k_2)$-bytes, and k_3-bytes strings respectively. Let $\mathbf{m} \in \{0, 1\}^{8k_1}$ be a message to be padded and $\mathbf{r} = \mathbf{r}_0 \| 0^\nu$ where $\mathbf{r}_0 \in \{0, 1\}^{8k_3-\nu}$ is a randomly selected binary string. Then the padding process proceeds as follows:

- Select random $0 \le l_0 < l_1 < \cdots < l_{t-1} \le n + w - 1$ and let $\mathbf{e}_0 = l_0 \| l_1 \cdots \| l_{t-1} \in \{0, 1\}^{16t}$. Note that one may use $\mathbf{r}$ to compute $\mathbf{e}_0$. Set

$$\mathbf{y} = ((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e}_0)) \oplus H_2(\mathbf{r}, \mathbf{e}_0)) \| (\mathbf{r} \oplus H_3((\mathbf{m} \| H_1(\mathbf{m}, \mathbf{r}, \mathbf{e}_0)) \oplus H_2(\mathbf{r}, \mathbf{e}_0))) \tag{3}$$

- 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 $0 \le i < t$ and $\mathbf{e}[j] = 0$ for $j \ne l_i$. Outputs $(\mathbf{y}_1, \mathbf{e})$.

Figure 1: mediumEncoding based RLCEpad



The RLCEpad process is shown graphically in Figure 1.

Shoup [28, Theorem 3] showed the following result for OAEP+: "*If the underlying trapdoor permutation scheme is one way, then OAEP+ is secure against adaptive chosen ciphertext attack in the random oracle model*". Our padding scheme RLCEpad is identical to OAEP+ with the following exceptions: In OAEP+, the function H_2 outputs a string of k_1-bytes which is $\oplus$-ed with $\mathbf{m}$. In RLCEpad, the function H_2 outputs a string of $(k_1 + k_2)$-bytes which is $\oplus$-ed with $\mathbf{m}\|H_2(\mathbf{r}, \mathbf{e_0})$. Since H_1, H_2, H_3 are random oracles, this revision requires no change in the security proof of [28, Theorem 3]. Thus, assuming the hardness of decoding RLCE ciphertexts, the proof in [28, Theorem 3] could be used to show that RLCE-RLCEpad is secure against IND-CCA2 attacks. The proof in [28] shows that the adversary A's advantage is bounded by

$$\texttt{InvAdv}(A') + \frac{(q_{H_1} + q_D)}{2^{8k_3}} + \frac{(q_D + 1)q_{H_2}}{2^{8k_2}} \tag{4}$$

where q_D is the maximum number of decryption oracle queries, q_{H_1}, q_{H_2}, and q_{H_3} are the maximum number of queries made by A to the oracles H_1, H_2 and H_3 respectively, and $\texttt{InvAdv}(A')$ is the success probability that a particular adversary A' has in breaking the one-way trapdoor permutation scheme. Furthermore, the time and space requirements of A' are related to those of A as follows:

$$\begin{aligned} T(A') &= O\left(T(A) + q_{H_2}q_{H_3}T_f + (q_{H_1} + q_{H_2} + q_{H_3} + q_D)\texttt{mLen}\right) \\ S(A') &= O\left(S(A) + (q_{H_1} + q_{H_2} + q_{H_3})\texttt{mLen}\right) \end{aligned}$$

where T_f is the time required to compute the one-way permutation f and space is measured in bits of storage.

The selection of RLCEpad parameters k_1, k_2, k_3 in Table 2 is based on the above reduction proof and bounds. As an example, for 128-bit secure RLCE scheme $(532, 376, 78)$, we use $k_2 = k_3 = 32$-bytes. Thus, we can encrypt $k_1 = 504$-bytes of information.

Remark: In RLCE encryption scheme, the error positions $\mathbf{e}_0 = l_0\|\cdots\|l_{t-1}$ and the error vector $\mathbf{e}$ are used in the RLCEpad process. The message recipient needs to recover the values of $\mathbf{e}_0$ and $\mathbf{e}$ for RLCEpad decoding. In case that $\mathbf{e}_1[i] = 0$ for some $0 \le i < t$, the corresponding error position cannot be recovered by the recipient. To avoid this from happening, each time when $\mathbf{e}_1[i] = 0$ for some $0 \le i < t$, one restarts the padding approach with a new random value $\mathbf{r}$. This process continues until all error values are non-zero.

Table 2 lists the message bandwidth and RLCEpad scheme parameters for the recommended schemes. In case that $v = 8(k_1 + k_2 + k_3) - \texttt{mLen} > 0$, the last v-bits of the k_3-bytes random seed $\mathbf{r}$ should be set zero and the last v-bit of the encoded string $\mathbf{y}$ is discarded. For RLCEpad with $v > 0$, the decoding process produces an encoded string **y** with last v-bits missing. After apply H_3 to the first part of **y**, the recipient obtains k_3-bytes. The recipient then discards the last v-bits and $\oplus$ the remaining $(8k_3 - v)$-bits with the second half of **y** to obtain the $(8k_3 - v)$-bits of **r** without the v-bits zero trailer.

5 Cryptographic Elements

This section describes the basic cryptographic elements that support the RLCE-KEM scheme specified in this proposal.

5.1 Cryptographic Hash Functions

In RLCE-KEM schemes, cryptographic hash functions may be used in key generation and message padding processes. In the reference/optimized implementation, SHA-256 and SHA-512 from FIPS 180-4 [25] are used.

5.2 Random Bit Generators

Whenever RLCE-KEM 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 [3], that provides an appropriate security strength. In the reference/optimized implementation, SHA-512 based Hash_DRBG and AES based CTR_DRBG from NIST SP 800-90Ar1 are used.

5.3 Nonces

RLCE-KEM 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.

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 $(n, k, d, t, w; X, G_s, P_1, P_2, A, G)$ and an RLCE public key $(n + w, k, t, G)$ as specified in Section 2.

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 $\texttt{paraB} = b_0b_1b_2b_3b_4b_5b_6b_7$ indicating the RLCE parameters supported. The first four bits $\texttt{paraB}[0, \cdots, 3] = b_0b_1b_2b_3$ specify the padding and message encoding schemes. In the following specification, we have $\texttt{paraB}[0, \cdots, 3]$ =0001. The other values are reserved for future use. For example, the optimized implementations use the following values:

$$b_0b_1b_2b_3 = \begin{cases} 0001 & \text{RLCEpad-mediumEncoding (schemes specified in this document)} \\ 0011 & \text{reserved for RLCEpad-basicEncoding specified in Wang [33]} \\ 0000 & \text{reserved for RLCEspad-mediumEncoding specified in Wang [33]} \\ 0010 & \text{reserved for RLCEspad-basicEncoding specified in Wang [33]} \end{cases}$$

The last 4 bits $\texttt{paraB}[4, \cdots, 7] = b_4b_5b_6b_7$ represent the RLCE parameter ID (= $0, 1, 2$) in Table 2. For example, $b_4b_5b_6b_7 = 0001$ represents the RLCE scheme with ID=1 ($\kappa_c = 192$ and $\kappa_q = 110$).

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} which 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} which 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 $A^{-1} = \text{diag}(A_0^{-1}, \cdots, A_{w-1}^{-1})$ which 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 $\texttt{invA} = a_1\|\cdots\|a_{2w}$.
- The $k \times (u_0 + 1)$ matrix $X = \begin{pmatrix} W^{-1} & U^T & 0 \\ V & 0 & 0 \end{pmatrix}$ where the $u \times u$ matrix W^{-1}, the $(k - u) \times u$ matrix V and the value u_0 are defined in Section 2. The $u \times 1$ matrix U^T is defined in Section 7.11. For the step by step generation of X, it is referred to Section 7.11. In the binary representation of the private key, X is represented by the list of $k(u_0 + 1)$ field elements $\texttt{matX} = x_1\|\cdots\|x_{k(u_0+1)}$.
- The n field elements corresponding to the GRS inverse coefficients $\texttt{grsInv} = v_0\|\cdots\|v_{n-1}$.
- The public key G_E which consists of $k(n + w - k)$ field elements $\texttt{pk} = z_1\|\cdots\|z_{k\times(n+w-k)}$. The public key is included to speed up the decryption process.

In a summary, 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

$$\texttt{paraB}\|\texttt{perm}_1\|\texttt{perm}_2\|\texttt{invA}\|\texttt{matX}\|\texttt{grsInv}\|\texttt{pk}\|0^\nu$$

of

$$4n + 2w + 1 + \lceil m(n + k + 2w + kn + kw + ku_0 - k^2)/8\rceil$$

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

As an example for the RLCE parameter ID=0, we have $n = 630, k = 470, w = 160, m = 10$. Thus the private key is 310116 bytes.

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 [3], 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`$[0,\cdots,3] = b_0b_1b_2b_3$ specifies the padding and message encoding schemes used.
   (b) `paraB`$[4,\cdots,7] = b_4b_5b_6b_7$ 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. Check that the value `paraB` satisfies the conditions:
   (a) The integer defined by the first four bits `paraB`$[0,\cdots,3] = 1$.
   (b) The integer defined by the last four bits `paraB`$[4,\cdots,7] = b_4b_5b_6b_7$ lies in the interval $[0,6]$.
2. 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.
3. Check 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 $=\lceil(m \times \texttt{nRE})/8\rceil + 4n + 2w$.

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

14. Let randE = B2FE(randBytes$[0, \cdots, \lceil(m \times \texttt{nRE})/8\rceil - 1]$, 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$[\lceil(m \times \texttt{nRE})/8\rceil, \cdots, \lceil(m \times \texttt{nRE})/8\rceil + 2n - 3]$, $n, n-1$) 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 ≥ 0 do

    (a) Let P_2 = getPermutation(randBytes$[\lceil(m \times \texttt{nRE})/8\rceil + 2n - 2 + \texttt{pctr}, \cdots, \lceil(m \times \texttt{nRE})/8\rceil + 4n + 2w - 5 + \texttt{pctr}]$, $n+w, n+w-1$) 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. Let $G = [I_k, G_E]$ be the echelon form of G_3.

27. Let X = precompute(P_2, G, u_0) where precompute is defined in Section 7.11.

28. 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.

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: $\texttt{RLCE\_encrypt}(\texttt{pk}, \texttt{entropy}, \texttt{nonce}, \texttt{msg})$
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. Recover `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) $\texttt{paraB}[0, \cdots, 3] = b_0b_1b_2b_3$ specifies the padding and message encoding schemes used.
   (b) $\texttt{paraB}[4, \cdots, 7] = b_4b_5b_6b_7$ specifies the RLCE scheme ID defined in Table 2.
2. Check that the value `paraB` satisfies the conditions:
   (a) The integer defined by the first four bits $\texttt{paraB}[0, \cdots, 3] = b_0b_1b_2b_3$ equals to 1.
   (b) The integer defined by the last four bits $\texttt{paraB}[4, \cdots, 7] = b_4b_5b_6b_7$ lies in the interval $[0, 6]$.
3. Retrieve 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. Check that `entropy` is at least $128 + \kappa_c$ bits.
5. Check that `msg` is k_1 bytes.
6. Let `pers` = "`PQENCRYPTIONRLCEver1`".
7. Let `addS` = "`GRSbasedPQEncryption`"||`0x00`.
8. If $\kappa_c = 256$, then let `nonce` = `nonce`||"`RLCEencNonceVer1`" 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. Let $\texttt{ctr} = 0$ and $\texttt{repeat} = 1$.

12. While $\texttt{repeat} > 0$ do

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

    (b) If $\kappa_c \le 192$, then run CTR_DRBG_Generate_algorithm(DRBG_state, 8 ∗ nRB0, addS) which is specified in Section 10.2.1.5.1 of SP 800-90A rev 1. This process will return an array randBytes of nRB0-bytes and return a new DRBG state DRBG_state.

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

    (d) Let $\texttt{padrand} = \texttt{randBytes}[0, \cdots, k_3 - 1]$.

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

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

    (g) Let $\texttt{e0} = l_0||l_1||\cdots||l_{t-1}$ 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 $0 \le i < t$.

    (h) Let paddedMSG=RLCEpad(msg,pk,padrand,e0) be a $(k_1+k_2+k_3)$-bytes string which RLCEspad is defined in Section 7.12.

    (i) 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 ν-bits information of paddedMSG is lost during this conversion.

    (j) Let $\texttt{errValue}[0, \cdots, t - 1] = \texttt{FEMSG}[k, \cdots, k + t - 1]$.

    (k) If $\texttt{errValue}[i] = 0$ for some $0 \le i < t$, then let $\texttt{repeat} = 1$. Otherwise, let $\texttt{repeat} = 0$.

13. Let $\texttt{cipher}[0, \cdots, n + w - 1] = \texttt{FEMSG}[0, \cdots, k - 1] \times G$.

14. Let $\texttt{cipher}[l_i] = \texttt{cipher}[l_i] + \texttt{errValue}[i]$ for all $0 \le i < t$.

15. 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. Recover paraB and $(X, \texttt{grs}, P_1^{-1}, P_2^{-1}, A^{-1}, G)$ from sk.

2. Check that the value $\texttt{paraB}$ satisfies the conditions:

   (a) The integer defined by the first four bits $\texttt{paraB}[0, \cdots, 3] = b_0 b_1 b_2 b_3$ equals to 1.

   (b) The integer defined by the last four bits $\texttt{paraB}[4, \cdots, 7] = b_4 b_5 b_6 b_7$ lies in the interval $[0, 6]$.

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

4. Let $\texttt{cipher} = \texttt{B2FE}(\texttt{c}, \texttt{m})$ where $\texttt{B2FE}$ is defined in Section 7.10 and $\texttt{cipher}$ is a list of field elements.

5. Let $\texttt{cipher}' = \texttt{matrix\_col\_permutation}(\texttt{cipher}, P_2^{-1})$ where $\texttt{matrix\_col\_permutation}$ is defined in Section 7.9 and $\texttt{cipher}$ is considered as an $1 \times (n + w)$ matrix.

6. Let $C_1 = \texttt{matrix\_mul\_A}(\texttt{cipher}', A^{-1})$ where $\texttt{matrix\_mul\_A}$ is defined in Section 7.9.

7. 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.

8. Let $C_3 = \texttt{matrix\_col\_permutation}(C_2, P_1^{-1})$ be a $1 \times n$ matrix.

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

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

11. Let $\texttt{code}' = \texttt{rs\_decode}(\texttt{TBD}, G_s, m)$ where $\texttt{rs\_decode}$ is defined in Section 7.13. It is noted that $\texttt{code}'$ is a list of $2^m - 1$ field elements that contains a prefix of $2^m - 1 - n$ zero elements.

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

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

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

15. 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 $0 \leq i \leq u - 1$.

16. 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 $0 \leq i \leq k - u - 1$.

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

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

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

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

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

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

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

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

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

25. Let `message=RLCEpadDecode(paddedMSG,sk,e0)` which is a k_1 bytes string and `RLCEpadDecode` is defined in Section 7.12.

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 $\texttt{msg} = \texttt{ss}||\texttt{0x00}||\cdots||\texttt{0x00}$ be a k_1 byte string obtained by adding $k_1 - \texttt{sslen}$ 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 $\texttt{ss} = \texttt{msg}[0]||\cdots\texttt{msg}[\texttt{sslen}-1]$

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 $GF(2^m)$:

$$\begin{array}{ll} GF(2^{10}): & \pi(x) = x^{10} + x^3 + 1 \\ GF(2^{11}): & \pi(x) = x^{11} + x^2 + 1 \end{array}$$

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 $2 \times$ `numI` bytes.
2. `numI` is a positive integer number.

Process:

1. Let `numBytes` $= 2 \times$ `numI`;
2. Let `shortIntegers`$[i]$ = `randBytes`$[2i] \times 2^8$ + `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 $S[0, \cdots, n-1]$ such that $X = S[0] \cdot 2^{8(n-1)} + \cdots + S[n-1] \cdot 2^0$.

I2BS Output: $S[0, \cdots, n-1]$.

BS2I Input:

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

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 $D = \texttt{I2BS}(\texttt{counter}, 4)$ 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,`κ_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 [3].
Function call: `hash_DRBG(entropy, nonce, pers_string, add_string, numBytes, `κ_c`)`
Input:

1. `entropy, nonce, pers_string, add_string` are binary strings.
2. `numBytes` is a positive integer.
3. κ_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,`κ_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 2×2 matrix defined by

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

   (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 $0 \leq i \leq k - 1$ and $0 \leq j \leq w - 1$, let $R[i][j] =$ `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`$[0, \cdots, 2v-1], u, v$)

Input:

1. `randBytes`$[0, \cdots, 2v-1]$ is an array of $2v$ bytes.
2. u is a positive integer number.
3. $v \leq u-1$ 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) $j =$ `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 $0 \le i < k, 0 \le j < u$.

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 $0 \le i < k$ and $0 \le j < d$, let $G_1[i][j] = G[i][j]$.
3. For $0 \le i < k$ and $0 \le j < w$, 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. $A = \mathtt{diag}(A_0, \cdots, A_{w-1})$ is a $2w \times 2w$ matrix where A_i is a 2×2 matrix for $i = 0, \cdots, w - 1$.

Process:

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

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

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

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

7.11 Pre-computation for private key

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

7.12 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 $v = 8 * (k_1 + k_2 + k_3) - \mathtt{mLen}$ and let $\mathtt{mask} = 1^{8-v}0^v$, 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 v-bits of `padrand` to zero.
5. Set $\mathtt{re0} = \mathtt{padrand}||\mathtt{e0}$ and $\mathtt{mre0} = \mathtt{msg}||\mathtt{re0}$.
6. Let $\mathtt{h1mre0} = \mathtt{RLCE\_MGF}(\mathtt{mre0}, k_2)$ where `RLCE_MGF` is defined in Section 7.4.
7. Let $\mathtt{h2re0} = \mathtt{RLCE\_MGF}(\mathtt{re0}, \mathtt{k}_1 + \mathtt{k}_2)$.
8. Set $\mathtt{paddedMSG1} = (\mathtt{msg}||\mathtt{h1mre0}) \oplus \mathtt{h2re0}$.
9. Let $\mathtt{h3mh1} = \mathtt{RLCE\_MGF}(\mathtt{paddedMSG1}, \mathtt{k}_3)$.
10. Set $\mathtt{paddedMSG} = \mathtt{paddedMSG1}||(\mathtt{padrand} \oplus \mathtt{h3mh1})$.

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 $\mathtt{paddedMSG1} = \mathtt{paddedMSG}[0, \cdots, k_1 + k_2 - 1]$.
4. Let $\mathtt{h3mh1} = \mathtt{RLCE\_MGF}(\mathtt{paddedMSG1}, k_3)$ 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 ν-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 $\texttt{paddedMSG}[k_1, \cdots, k_1 + k_2 - 1] \neq \texttt{h1mre0}$ return error.

Output: the k_1 bytes string msg.

7.13 Reed-Solomon decoding

The function rs_decode removes errors within a received Reed-Solomon code that contains errors.
Function call: rs_decode(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 Appendix A: RLCE Security Analysis (Informative)

8.1 The dual RLCE scheme

For McEliece-style encryption schemes, one may analyze its security by mounting attacks on the dual of McEliece schemes. It is shown that McEliece encryption scheme is equivalent to Niederreiter encryption scheme (see Wang [33] for details). That is, for each McEliece encryption scheme public key, one can derive a Niederreiter encryption scheme public key and, for each Niederreiter encryption scheme public key, one can derive a McEliece encryption scheme public key. One can break the McEliece encryption scheme (respectively the Niederreiter encryption scheme) if and only if one can break the corresponding Niederreiter encryption scheme (respectively, the McEliece encryption scheme). In this section, we show that a similar equivalent result may not hold for RLCE schemes. We first try to give a natural candidate construction of Niederreiter RLCE scheme and show it is challenging (or infeasible) to design an efficient decryption algorithm. Thus it is not clear whether there exists an efficient equivalent Niederreiter RLCE encryption scheme corresponding to the McEliece RLCE encryption scheme.

`RLCEdual.KeySetup`(n, k, d, t, r). For an $(n, k, 2t+1)$ linear code C, let $H_s = [\mathbf{h}_0, \cdots, \mathbf{h}_{n-1}]$ be an $(n-k)\times n$ parity check matrix of C. The keys are generated using the following steps.

1. Let $C_0, C_1, \cdots, C_{n-1} \in GF(q)^{(n-k)\times r}$ be $(n-k)\times r$ matrices drawn uniformly at random and let

$$H_1 = [\mathbf{h}_0, C_0, \mathbf{g}_1, C_1 \cdots, \mathbf{h}_{n-1}, C_{n-1}] \tag{5}$$

   be the $(n-k)\times n(r+1)$ matrix obtained by inserting the random matrices C_i into H_s.

2. Let $A_0, \cdots, A_{n-1} \in GF(q)^{(r+1)\times(r+1)}$ be non-singular $(r+1)\times(r+1)$ matrices chosen uniformly at random and let $A = \mathrm{diag}[A_0, \cdots, A_{n-1}]$ be an $n(r+1)\times n(r+1)$ non-singular matrix.

3. Let S be a random dense $(n-k)\times(n-k)$ non-singular matrix and P be an $n(r+1)\times n(r+1)$ permutation matrix.

4. The public key is the $(n-k)\times n(r+1)$ matrix $H = SH_1AP$ and the private key is (S, H_s, P, A).

`RLCEdual.Enc`$(H, \mathbf{m})$. For a row message $\mathbf{m} \in GF(q)^{n(r+1)}$ of weight t, compute the ciphertext $\mathbf{c} = \mathbf{m}H^T$.

Candidate decryption algorithms? For a received ciphertext $\mathbf{c} = \mathbf{m}H^T$, we have $\mathbf{c}(S^T)^{-1} = \mathbf{m}P^TA^TH_1^T$. Since each non-zero element in $\mathbf{m}$ can be converted to at most $(t+1)$-nonzero elements in $\mathbf{m}P^TA^T$, the weight of $\mathbf{m}P^TA^T$ is at most $(r+1)t$. Thus we can decrypt the ciphertext $\mathbf{c}$ only if we had an efficient $(r+1)t$-error-correcting algorithm for the code defined by the parity check matrix H_1. Since the matrices $C_0, C_1, \cdots, C_{n-1}$ are selected at random, it is unknown whether there is an efficient error correcting algorithm for the code defined by the parity check matrix H_1. In the following, we describe a natural candidate algorithm for decrypting the ciphertext and show that this algorithm will not work. Let $G_s = [\mathbf{g}_0, \cdots, \mathbf{g}_{n-1}]$ be the $k\times n$ generator matrix for the linear code C such that $G_sH_s^T = 0$. Furthermore, let $D_0, D_1, \cdots, D_{n-1}$ be $k\times r$ matrices, such that $D_0C_0^T + D_1C_1^T + \cdots + D_{n-1}C_{n-1}^T = 0$ (for example, one may take $D_0 = D_1 = \cdots = D_{n-1} = 0$). Let $G_1 = [\mathbf{g}_0, D_0, \cdots, \mathbf{g}_{n-1}, D_{n-1}]$, and $G = G_1(A^T)^{-1}(P^T)^{-1}$. Then

$$GH^T = G_1(A^T)^{-1}(P^T)^{-1}P^TA^TH_1^TS^T = G_1H_1^T = 0.$$

For a received ciphertext $\mathbf{c}$ with $\mathbf{c}(S^T)^{-1} = \mathbf{m}P^TA^TH_1^T$, one can find a vector $\mathbf{a} \in GF(q)^{n(r+1)}$ such that $\mathbf{c}(S^T)^{-1} = \mathbf{a}H^T$. Then we have $(\mathbf{a} - \mathbf{m}P^TA^T)H^T = 0$. Since the space spanned by the rows of H is of dimension $n-k$, the orthogonal space to the space spanned by the rows of H is of dimension $nt+k$.

However, the space spanned by the rows of G only has dimension k. Thus only with a negligible probability, the vector $\mathbf{a} - \mathbf{m}P^T A^T$ is in the code space generated by the rows of G. In other words, the above candidate decryption algorithm will succeed only with a negligible probability.

The arguments in the preceding paragraph show that it is hard to design an equivalent Niederreiter-type encryption scheme for RLCE scheme. This provides certain evidence for the robustness of RLCE scheme.

8.2 Weak keys and algebraic attacks

Loidreau and Sendrier [20] pointed out some weak keys for binary Goppa code based McEliece schemes and similar weak keys for RLCE schemes should not be used. For an RLCE scheme ciphertext $\mathbf{c}$ of a message $\mathbf{m}$, one can obtain a valid ciphertext for a message $\mathbf{m}+\mathbf{m}'$ by letting $\mathbf{c}' = \mathbf{c}+\mathbf{m}'G$ without knowing the message $\mathbf{m}$. This kind of attacks could be defeated by using IND-CCA2-secure message padding schemes. Faugere, Otmani, Perret, and Tillich [12] developed an algebraic attack against quasi-cyclic and dyadic structure based compact variants of McEliece encryption scheme. Wang [32] showed that the algebraic attacks will not work against the RLCE encryption scheme. A straightforward modification of the analysis in [32] can be used to show that the algebraic attacks will not work against the revised RLCE scheme either.

8.3 Classical and quantum Information-Set Decoding

Information-set decoding (ISD) is one of the most important message recovery attacks on McEliece encryption schemes. The state-of-the-art ISD attack for non-binary McEliece scheme is the one presented in Peters [27], which is an improved version of Stern's algorithm [30]. Peters's attack [27] also integrated analysis techniques for ISD attacks on binary McEliece scheme discussed in [6]. For the RLCE encryption scheme, the ISD attack is based on the number of columns in the public key G instead of the number of columns in the private key G_s. The cost of ISD attack on an $[n, k, t; w]$-RLCE scheme is equivalent to the cost of ISD attack on an $[n + w, k; t]$-McEliece scheme.

For the naive ISD, one first uniformly selects k columns from the public key and checks whether it is invertible. If it is invertible, one multiplies the inverse with the corresponding ciphertext values in these coordinates that correspond to the k columns of the public key. If these coordinates contain no errors in the ciphertext, one recovers the plain text. To be conservative, we may assume that randomly selected k columns from the public key is invertible. For each $k \times k$ matrix inversion, Strassen algorithm takes $O(k^{2.807})$ field operations (though Coppersmith-Winograd algorithm takes $O(k^{2.376})$ field operations in theory, it may not be practical for the matrices involved in RLCE encryption schemes). In a summary, the naive information-set decoding algorithm takes approximately $2^{\kappa'_c}$ steps to find k-error free coordinates where, by Sterling's approximation,

$$\kappa'_c = \log_2\left(\frac{\binom{n+w}{k}\left(k^{2.807}+k^2\right)}{\binom{n+w-t}{k}}\right) \simeq (n+w)I\left(\frac{k}{n+w}\right) - (n+w-t)I\left(\frac{k}{n+w-t}\right) + \log_2\left(k^{2.807}+k^2\right) \quad (6)$$

and $I(x) = -x\log_2(x) - (1-x)\log_2(1-x)$ is the binary entropy of x. There are several improved ISD algorithms in the literature. These improved ISD algorithms allow a small number of error positions within the selected k ciphertext values or select $k + \delta$ columns of the public key matrix for a small number $\delta > 0$ or both. Peters provided a script [27][1] to calculate the security strength of a McEliece encryption scheme using the improved ISD algorithms. For the security strength $128 \le \kappa_c \le 256$, our experiment shows that generally we have $\kappa'_c - 10 \le \kappa_c \le \kappa'_c - 4$.

[1]available from https://christianepeters.wordpress.com/publications/tools/.

An RLCE scheme is said to have quantum security level κ_q if the expected running time (or circuit depth) to decrypt an RLCE ciphertext using Grover's algorithm based ISD is 2^{κ_q}. For a function $f : \{0,1\}^l \to \{0,1\}$ with the property that there is an $x_0 \in \{0,1\}^l$ such that $f(x_0) = 1$ and $f(x) = 0$ for all $x \neq x_0$, Grover's algorithm finds the value x_0 using $\frac{\pi}{4}\sqrt{2^l}$ Grover iterations and $O(l)$ qubits. Specifically, Grover's algorithm converts the function f to a reversible circuit C_f and calculates

$$|x\rangle \xrightarrow{C_f} (-1)^{f(x)}|x\rangle$$

in each of the Grover iterations, where $|x\rangle$ is an l-qubit register. Thus the total steps for Grover's algorithm is bounded by $\frac{\pi|C_f|}{4}\sqrt{2^l}$.

For the RLCE scheme, quantum ISD could be carried out similarly as in Bernstein's [5]. One first uniformly selects k columns from the public key and checks whether it is invertible. If it is invertible, one multiplies the inverse with the ciphertext. If these coordinates contain no errors in the ciphertext, one recovers the plain text. Though Grover's algorithm requires that the function f evaluate to 1 on only one of the inputs, there are several approaches (see, e.g., Grassl et al [14]) to cope with cases that f evaluates to 1 on multiple inputs.

For randomly selected k columns from a RLCE encryption scheme public key, the probability that the ciphertext contains no errors in these positions is $\frac{\binom{n+w-t}{k}}{\binom{n+w}{k}}$. Thus the quantum ISD algorithm requires $\sqrt{\binom{n+w}{k}/\binom{n+w-t}{k}}$ Grover iterations. For each Grover iteration, the function f needs to carry out the following computations:

1. Compute the inverse of a $k \times k$ sub-matrix G_{sub} of the public key and multiply it with the corresponding entries within the ciphertext. This takes $O\left(k^{2.807} + k^2\right)$ field operations if Strassen algorithm is used.
2. Check that the selected k positions contain no errors in the ciphertext. This can be done with one of the following methods:
   (a) Multiply the recovered message with the public key and compare the differences from the ciphertext. This takes $O((n+w)k)$ field operations.
   (b) Use the redundancy within message padding scheme to determine whether the recovered message has the correct padding information. The cost for this operation depends on the padding scheme.

It is expensive for circuits to use look-up tables for field multiplications. Using Karatsuba algorithm, Kepley and Steinwandt [19] constructed a field element multiplication circuit with gate counts of $7 \cdot (\log_2 q)^{1.585}$. In a summary, the above function f for the RLCE quantum ISD algorithm could be evaluated using a reversible circuit C_f with $O\left(7\left((n+w)k + k^{2.807} + k^2\right)(\log_2 q)^{1.585}\right)$ gates. To be conservative, we may assume that a randomly selected k-columns sub-matrix from the public key is invertible. Thus Grover's quantum algorithm requires approximately

$$7\left((n+w)k + k^{2.807} + k^2\right)(\log_2 q)^{1.585}\sqrt{\frac{\binom{n+w}{k}}{\binom{n+w-t}{k}}} \tag{7}$$

steps for the simple ISD algorithm against RLCE encryption scheme. Advanced quantum ISD techniques may be developed based on improved ISD algorithms. However our analysis shows that the reduction on the quantum security is marginal. The reader is also referred to a recent report [18] for an analysis of quantum ISD based on improved ISD algorithms. For each of the recommended schemes in Table 2, the row (κ_c', κ_q) in

Table 3 shows the security strength under the classical ISD and classical quantum ISD attacks. For example, the RLCE scheme with ID = 1 in Table 2 has 139-bits security strength under classical ISD attacks and 89-bits security strength under quantum ISD attacks.

Table 3: Security strength for RLCE schemes in Table 2

Scheme ID (κ_c, κ_q)	**0 (128,80)**	**1 (192,110)**	**2 (256,144)**
(κ_c', κ_q)	(139, 90)	(205, 124)	(269, 156)
(κ_c^s, κ_q^s)	(135, 86)	(202,120)	(266,154)
$(\kappa_c^{Stern}, \kappa_q^{Stern})$	(130, 80)	(195, 113)	(257, 145)
insecure cipher prob.	$(7, 2^{-76})$	$(11, 2^{-117})$	$(14, 2^{-167})$
known non-rand. pos.	459	741	772

8.4 Improved Information Set Decoding

In this section, we briefly review Stern's algorithm [30]. Let the $k \times (n+w)$ matrix G be the public key and $\mathbf{c}$ be an RLCE scheme ciphertext. Let $G_e = \begin{pmatrix} \mathbf{c} \\ G \end{pmatrix}$ be a $(k+1) \times (n+w)$ matrix. Stern's algorithm will find the minimal weight code $\mathbf{e}$ that is generated by G_e. It is straightforward to show that $\mathbf{e}$ is the error vector for the ciphertext $\mathbf{c}$. Stern's information set decoding algorithm for finding the vector $\mathbf{e}$ is as follows.

1. Select two small numbers $p < k/2$ and $l < n+w-k$.
2. Select k columns $\mathbf{g}_{i_1}, \cdots, \mathbf{g}_{i_k}$ from G_e and l columns $\mathbf{g}_{j_1}, \cdots, \mathbf{g}_{j_l}$ from the remaining $n+w-k$ columns of G_e where $0 \le i_1, \cdots, i_k, j_1, \cdots, j_l \le n+w-1$ are distinct numbers. It is expected that the ciphertext $\mathbf{c}$ contains $2p$ errors within the locations $i_1, \cdots, i_k$ and no errors within the positions $j_1, \cdots, j_l$.
3. Let $P_{i_1,\cdots,i_k,j_1,\cdots,j_l}$ be a $(n+w) \times (n+w)$ permutation matrix so that
$$G_e P_{i_1,\cdots,i_k,j_1,\cdots,j_l} = (\mathbf{g}_{i_1}, \cdots, \mathbf{g}_{i_k}, \mathbf{g}_{j_1}, \cdots, \mathbf{g}_{j_l}, G_r),$$
where G_r is a $(k+1) \times (n+w-k-l)$ matrix.
4. Compute the echelon format
$$G_E = E(G_e P_{i_1,\cdots,i_k,j_1,\cdots,j_l}) = S G_e P_{i_1,\cdots,i_k,j_1,\cdots,j_l} = (I, L, G_r)$$
where S is a $(k+1) \times (k+1)$ matrix.
5. Find random vectors $\mathbf{u}, \mathbf{v} \in GF(q)^{(k+1)/2}$ of weight p such that $(\mathbf{u}, \mathbf{v})L = \mathbf{0}$. If no such $\mathbf{u}, \mathbf{v}$ found, go to Step 2.
6. If $(\mathbf{u}, \mathbf{v})L = \mathbf{0}$, then check whether $(\mathbf{u}, \mathbf{v})G_r$ has weight $t-2p$. If it does not have weight $t-2p$, go to Step 2.
7. If $(\mathbf{u}, \mathbf{v})G_r$ has weight $t-2p$, then $\mathbf{e} = (\mathbf{u}, \mathbf{v})G_E P^{-1}_{i_1,\cdots,i_k,j_1,\cdots,j_l}$ is the error vector for the ciphertext $\mathbf{c}$.

It is noted that if we take $p = l = 0$, then Stern's algorithm is the naive ISD algorithm that we have discussed in the preceding section. For the convenience of analysis, we assume that $pl > 0$ in the following discussion. The algorithm takes approximately

$$S_I = \frac{\binom{n+w}{\lfloor k/2 \rfloor}\binom{n+w-\lfloor k/2 \rfloor}{k-\lfloor k/2 \rfloor}\binom{n+w-k}{l}}{\binom{n+w-t}{\lfloor k/2 \rfloor - p}\binom{t}{p}\binom{n+w-t-\lfloor k/2 \rfloor - p}{k-\lfloor k/2 \rfloor - p}\binom{t-p}{p}\binom{n+w-t-k+2p}{l}} \tag{8}$$

iterations. For each iteration, Step 4 takes $(2n+2w-k)k^2$ field operations, and Step 5 takes $2\binom{k/2}{p}(q-1)^p l(k+1)$ field operations. For each iteration, Step 6 runs $\binom{k/2}{p}^2(q-1)^{2p-l}$ times approximately and each runs takes $(n-k-l)(k+1)$ field operations. In a summary, Stern's ISD takes approximately 2^{κ_c} steps to find the error vector $\mathbf{e}$ where,

$$\kappa_c = \min_{p,l}\left\{\log_2\left(S_I\left((2n+2w-k)k^2 + 2\binom{k/2}{p}(q-1)^p l(k+1) + \binom{k/2}{p}^2(q-1)^{2p-l}(n-k-l)(k+1)\right)\right)\right\}. \tag{9}$$

Our experiments show that for RLCE schemes that we have interest in, the equation (9) is always achieved with $p = 1$ and $l = 3$. For quantum version of Stern's ISD algorithm, the Grover's algorithm could be used to reduce the iteration steps to $\sqrt{S_I}$. Thus the quantum security level under Stern attacks is approximately

$$\kappa_q = \min_{p,l}\left\{\log_2\left(\sqrt{S_I}\left((2n+2w-k)k^2 + 2\binom{k/2}{p}(q-1)^p l(k+1) + \binom{k/2}{p}^2(q-1)^{2p-l}(n-k-l)(k+1)\right)\right)\right\}. \tag{10}$$

In order to speed up Stern's algorithm, Peters [27] considers the following improvement:

1. For each iteration, one does not randomly selects k columns from G_e in Step 2. Instead, one reuses $k-c$ columns from the previous iteration where c is a fixed constant.

2. For a small finite field, fix a parameter $r > 1$ for certain pre-computation of row sums. This will not provide any benefit for a large field size such as those used in RLCE schemes.

3. For a small finite field, fix a parameter $m > 1$ such that one can use m error-free sets of size l. This will not provide any benefit for a large field size such as those used in RLCE schemes.

Our experiments show that for $\kappa_c \le 200$, Peters's improved version in [27] is at most 8 times fast than Stern's algorithm discussed in this section. That is, we generally have $\kappa_c - 3 \le \kappa_c^{Peter} \le \kappa_c$ where κ_c^{Peter} is the κ_c obtained from Peter's improved algorithm. For $\kappa_c \ge 250$, our experiments show that Peter's improved version has the same performance as Stern's algorithm discussed in this section. Furthermore, our experiments show that the optimal values for p, l in Peter's improved algorithm on all RLCE schemes are $p = 1$ and $l = 3$ also.

8.5 Information Set Decoding for systematic RLCE schemes

Canteaut and Sendrier [9] discussed a known-partial-plaintext-attack against McEliece encryption scheme where $\mathbf{c} = \mathbf{m}G + \mathbf{e}$. Let l, r be two positive integers such that $k = l + r$. Assume that $\mathbf{m} = [\mathbf{m}_l, \mathbf{m}_r]$ and $G = \begin{bmatrix} G_l \\ G_r \end{bmatrix}$. Then we have

$$\mathbf{c} = \mathbf{m}G + \mathbf{e} = [\mathbf{m}_l, \mathbf{m}_r]\begin{bmatrix} G_l \\ G_r \end{bmatrix} + \mathbf{e} = \mathbf{m}_l G_l + \mathbf{m}_r G_r + \mathbf{e}. \tag{11}$$

Thus if one knows the value of $\mathbf{m}_l$, the identity (11) becomes $\mathbf{c} - \mathbf{m}_l G_l = \mathbf{m}_r G_r + \mathbf{e}$ which could be much easy to decode than the original code-word $\mathbf{c}$ since $r < k$. Though this attack against RLCE could be defeated by using appropriate message padding for IND-CCA2-security, this attack can be integrated into information set decoding to design more efficient attacks against systematic RLCE schemes.

For the ISD against a systematic RLCE scheme, one uniformly selects $k = k_1 + k_2$ columns from the public key where k_1 columns are from the first k columns of the public key. Instead of multiplying the inverse of the selected k columns with the corresponding ciphertext values in these coordinates, one uses the corresponding ciphertext values for the selected k_1 columns within the first k columns of the public key to determine k_1 entries of the plaintext. Using these "recovered" k_1 entries of the plaintext, one calculates a new ciphertext $\mathbf{c}'$ with k_2 unknown plaintext entries as in the known-partial-plaintext-attack. Next one uses the inverse of the $k_2 \times k_2$ matrix to recover the remaining k_2 entries of the plaintext. In a summary, for each guessed k columns, one needs k_1k_2 field multiplications to compute the new ciphertext $\mathbf{c}'$, needs $k_2^{2.807}$ field multiplications to compute the matrix inverse, and additional k_2^2 steps to compute the remaining k_2 entries of the plaintext. If one selects the k columns uniformly at random, then the expected values for k_1, k_2 are $k_1 = \frac{k^2}{n+w}$ and $k_2 = \frac{k(n+w-k)}{n+w}$ respectively. Thus the above information-set decoding algorithm against systematic RLCE scheme takes approximately $2^{\kappa_c^s}$ steps to find k-error free coordinates where,

$$\kappa_c^s = \log_2\left(\frac{\binom{n+w}{k}\left(\frac{k^3(n+w-k)}{(n+w)^2}+\left(\frac{k(n+w-k)}{n+w}\right)^{2.807}+\left(\frac{k(n+w-k)}{n+w}\right)^2\right)}{\binom{n+w-t}{k}}\right). \tag{12}$$

Similarly, Grover's quantum algorithm based on the above ISD against systematic RLCE requires approximately

$$7\left(\frac{k^3(n+w-k)}{(n+w)^2}+\left(\frac{k(n+w-k)}{n+w}\right)^{2.807}+\left(\frac{k(n+w-k)}{n+w}\right)^2\right)(\log_2 q)^{1.585}\sqrt{\frac{\binom{n+w}{k}}{\binom{n+w-t}{k}}} \tag{13}$$

steps for the simple ISD algorithm against RLCE encryption scheme. For each of the recommended schemes in Table 2, the row (κ_c^s, κ_q^s) in Table 3 shows the security strength under the ISD and quantum ISD attacks against systematic RLCE schemes. For example, the RLCE scheme with ID = 1 in Table 2 has 135-bits security strength under ISD attacks and 85-bits security strength under quantum ISD attacks.

8.6 Insecure ciphertexts for systematic RLCE schemes

For a systematic RLCE encryption scheme, if a small number of errors were added to the first k components of the ciphertext, one may be able to exhaustively search these errors and recover the message. Given a ciphertext $\mathbf{c}$ with l errors within the first k components (note that the adversary does not know this value l), the adversary starts from $i = 1$, randomly select $k - i$ positions within the ciphertext, take these values as the uncorrupted message values, guess the remaining i values for the message. If these $k - i$ positions contain no errors, the adversary can use the redundant information within the padding scheme to check whether the guessed message is correct. Under the condition that there are l errors within the first k components of the ciphertext, the probability for this attack to be successful is bounded by

$$\gamma_l = \max_{l\le i\le t}\left\{\frac{\binom{k-l}{k-i}}{q^i\binom{k}{i}}\right\}$$

For each $i \le l$, the probability that there are at most l errors in the first k components of the ciphertext is bounded by

$$E_l = \frac{\sum_{i\le l}\binom{k}{i}\binom{n+w-k}{t-i}}{\binom{n+w}{t}}.$$

The RLCE scheme encryption process produces an insecure ciphertext in case that the ciphertext contains at most l errors within the first k components of the ciphertext and $\gamma_l > 2^{-\kappa_c}$ where κ_c is the security parameter.

In order to avoid producing insecure ciphertexts, RLCE encryption process should repeatedly encrypt the message until it produces a ciphertext with at least l errors in the first k components such that $\gamma_l \leq 2^{-\kappa_c}$. If the error locations are chosen uniformly at random, then the RLCE scheme encryption process produces an insecure ciphertext with the probability of at most

$$\max\{E_l : l \leq t, \gamma_l > 2^{-\kappa_c}\} \tag{14}$$

This probability is negligible for security parameters that we are interested in. Thus the RLCE scheme needs to repeat the encryption process for a second time only with a negligible probability. For each of the recommended schemes in Table 2, the row "insecure cipher prob." in Table 3 shows the number of errors that should be contained in the first k components of a secure ciphertext and the probability that a ciphertext is insecure. An an example, for the RLCE scheme with ID = 1 in Table 2, the first k components of an insecure ciphertext contain 7 or less errors and the probability for this to happen is smaller than 2^{-76}.

8.7 Sidelnikov-Shestakov's attack

Niederreiter's scheme [24] replaces the binary Goppa codes in McEliece scheme by GRS codes. Sidelnikov and Shestakov [29] broke Niederreiter's scheme by recovering an equivalent private key $(\mathbf{x}', \mathbf{y}')$ from a public key G for the code $\mathrm{GRS}_k(\mathbf{x}, \mathbf{y})$. For the given public key G, one computes the echelon form $E(G) = [I|G']$ using Gaussian elimination.

$$E(G) = \begin{bmatrix} 1 & 0 & \cdots & 0 & b_{0,k} & \cdots & b_{0,n-1} \\ 0 & 1 & \cdots & 0 & b_{1,k} & \cdots & b_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & b_{k-1,k} & \cdots & b_{k-1,n-1} \end{bmatrix} \tag{15}$$

Assume the ith row code-word $\mathbf{b}_i$ of $E(G)$ encodes a message $p_i(x) = a_0 + a_1 x + \cdots + a_{k-1}x^{k-1}$. Then

$$y_0 p_i(x_0) = 0, \cdots, y_i p_i(x_i) = 1, \cdots, y_{n-1} p_i(x_{n-1}) = b_{i,n-1} \tag{16}$$

Since the only non-zero elements are $b_{i,i}, b_{i,k}, \cdots, b_{i,n-1}$, p_i can be written as

$$p_i(x) = c_i \cdot \prod_{j=1, j\neq i}^{k} (x - x_j) \tag{17}$$

for some $c_i \neq 0$. By the fact that $\mathrm{GRS}_k(\mathbf{x}, \mathbf{y}) = \mathrm{GRS}_k(a\mathbf{x} + b, c\mathbf{y})$ for all $a, b, c \in GF(q)$ with $ab \neq 0$, we may assume that $x_0 = 0$ and $x_1 = 1$. In the following, we try to recover $x_2, \cdots, x_{n-1}$. Using equation (17), one can divide the row entries in (15) by the corresponding nonzero entries in another row to get several equations. For example, if we divide entries in row i_0 by corresponding nonzero entries in row i_1, we get

$$\frac{b_{i_0,j}}{b_{i_1,j}} = \frac{y_j p_{i_0}(x_j)}{y_j p_{i_1}(x_j)} = \frac{c_{i_0}(x_j - x_{i_1})}{c_{i_1}(x_j - x_{i_0})} \tag{18}$$

for $j = k, \cdots, n-1$. First, by taking $i_0 = 0$ and $i_1 = 1$, equation (18) could be used to recover $x_k, \cdots, x_{n-1}$ by guessing the value of $\frac{c_0}{c_1}$ which is possible when q is small. By letting $i_0 = 0$ and $i_1 = 2, \cdots, k-1$ respectively, equation (18) could be used to recover x_{i_1}. Sidelnikov and Shestakov [29] showed that the values of $\mathbf{y}$ can then be recovered by solving a linear equation system based on $x_0, \cdots, x_{n-1}$.

In the RLCE scheme, $2w$ columns of the public key matrix G are randomized. In case that the filtration attack in the next Section can identify the $n - w$ non-randomized columns, one can permute the columns of

G to obtain a new matrix G_N such that the first $n-w$ columns are the non-randomized columns. Then one can compute an echelon form $E(G_N)$ for G_N. Since the last $2w$ columns are randomized, they could not be used to establish any of the equations in Sidelnikov and Shestakov attack. We distinguish the following two cases:

1. If $w \geq n-k$, then one cannot establish enough equations within (16) to obtain the equation (17). Thus no equations in (18) could be established and Sidelnikov and Shestakov attack could not continue.
2. If $n-k > w$, equations (18) may only be used to recover the values of $x_0, \cdots, x_{n-w-1}$. If it has a negligible probability for one to guess the remaining values $x_{n-w}, \cdots, x_{n-1}$, then Sidelnikov and Shestakov attack will not be successful. The probability for one to guess the remaining values $x_{n-w}, \cdots, x_{n-1}$ correctly is bounded by $1/\binom{q-n+w+1}{w}w!$.

Thus for a security parameter κ_c, the RLCE parameters should be chosen in such a way that

$$w \geq n-k \text{ or } \binom{q-n+w+1}{w}w! \geq 2^{\kappa_c}. \tag{19}$$

For RLCE schemes that we are interested in, we generally have $w \geq n-k$ or $\binom{q-n+w+1}{w}w! > \sqrt{2^{\kappa_c}}$. For each of the recommended schemes in Table 2, the row κ_{SS} in Table 3 shows the security strength under the Sidelnikov-Shestakov attack. For example, the RLCE scheme with ID = 1 in Table 2 has 4429-bits security strength under the above Sidelnikov-Shestakov attack.

8.8 Known non-randomized column attack

In this section, we consider the security of RLCE schemes when the positions of non-randomized $n-w$ GRS columns are known to the adversary. In this scenario, the adversary has two ways to attack the RLCE scheme. In the first approach, the adversary may guess the remaining w columns of the GRS generator matrix. The probability for this attack to be successful is shown in (19) which is very small compared against the security parameters. Alternatively, the adversary may use Sidelnikov-Shestakov attack to calculate a private key for the punctured $[n-w, k]$ GRS_k code consisting of the non-randomized GRS columns and then list-decode the punctured $[n-w, k]$ GRS_k code. We first review some results for GRS list-decoding. The error distance of a received word $\mathbf{y} \in GF(q)^n$ to a code C is defined as $\Delta(\mathbf{y}, C) = \min\{\mathtt{wt}(\mathbf{y}-\mathbf{x}) : \mathbf{x} \in C\}$. For a vector $\mathbf{y} \in GF(q)^n$, $\mathbf{y}$'s Hamming ball of radius r is $B(\mathbf{y}; r) = \{\mathbf{y}' : \mathtt{wt}(\mathbf{y}-\mathbf{y}') \leq r\}$. For an MDS $[n, k, d]$ code C and a vector $\mathbf{y} \in GF(q)^n$, $B(\mathbf{y}; r)$ contains at most one code-word from C if $r \leq d/2$. If $d/2 < r \leq n - \sqrt{n(k-1)}$, $B(\mathbf{y}; r) \cap C$ contains at most polynomial many elements and the list-decoding algorithm by Guruswami and Sudan [15] can be used to efficiently output all elements in $B(\mathbf{y}; r) \cap C$. If the radius is stretched further, $B(\mathbf{y}; r) \cap C$ may contain exponentially many code-words.

For an RLCE ciphertext $\mathbf{c}$, let $\mathbf{c}'$ be the punctured ciphertext of length $n-w$ by restricting $\mathbf{c}$ to the punctured $[n-w, k]$ GRS_k code. In case that there are at most $n-w-\sqrt{(n-w)(k-1)}$ errors in $\mathbf{c}'$, one can decode the shortened $[n-w, k]$ GRS_k code using the list-decoding algorithm by Guruswami and Sudan [15]. Note that the probability for $\mathbf{c}'$ to contain at most $n-w-\sqrt{(n-w)(k-1)}$ errors is bounded by the hyper-geometric cumulative distribution function

$$PK_{n,w,t} = \frac{\sum_{i=0}^{n-w-\sqrt{(n-w)(k-1)}} \binom{n-w}{i}\binom{2w}{t-i}}{\binom{n+w}{t}} \tag{20}$$

That is, with probability $PK_{n,w,t}$ the encryption process produces a ciphertext that could be list-decoded using the $[n-w, k]$ GRS_k code. Thus the parameters should be chosen in such a way that $P_{n,w,t}$ is negligible (e.g., $P_{n,w,t} \leq 2^{\kappa_c}$) or the encryption process should repeatedly encrypt the message until it produces a ciphertext with at least $n - w - \sqrt{(n-w)(k-1)} + 1$ errors corresponding to the known non-randomized columns. Justesen and Hoholdt [17] showed the following theorem.

Theorem 8.1 *(Justesen and Hoholdt [17]) For an $[n, k]$ Reed-Solomon code C and an integer $\delta < n - k$, the expected size of $B(\mathbf{u}; \delta) \cap C$ is $\binom{n}{n-\delta}/q^{n-\delta-k}$ for randomly chosen $\mathbf{u} \in GF(q)^n$.*

By theorem 8.1, we may further require that the RLCE scheme repeatedly encrypt the message until it produces a ciphertext such that the size of $B(\mathbf{u}; \delta) \cap C$ is large than 2^{κ_c}, where δ is the number of errors that the ciphertext $\mathbf{c}'$ contains.

In order to avoid the attacks that we mentioned in this section, it is recommended that the encryption process should produce a ciphertext that avoids these attacks if the positions of non-randomized columns are publicly known. Alternatively, we may recommend that one select RLCE parameters in such a way that it is computationally infeasible to identify non-randomized columns from the public key.

8.9 Filtration attacks

Couvreur et al. [11] designed a filtration technique to attack GRS code based McEliece scheme. For two codes C_1 and C_2 of length n, the star product code $C_1 * C_2$ is the vector space spanned by $\mathbf{a} * \mathbf{b}$ for all pairs $(\mathbf{a}, \mathbf{b}) \in C_1 \times C_2$ where

$$\mathbf{a} * \mathbf{b} = [a_0 b_0, a_1 b_1, \cdots, a_{n-1} b_{n-1}].$$

For the square code $C^2 = C * C$ of C, we have $\dim C^2 \leq \min\left\{n, \binom{\dim C + 1}{2}\right\}$. For an $[n, k]$ GRS code C, let $\mathbf{a}, \mathbf{b} \in \mathrm{GRS}_k(\mathbf{x}, \mathbf{y})$ where $\mathbf{a} = (y_0 p_1(x_0), \cdots, y_{n-1} p_1(x_{n-1}))$ and $\mathbf{b} = (y_0 p_2(x_0), \cdots, y_{n-1} p_2(x_{n-1}))$. Then

$$\mathbf{a} * \mathbf{b} = (y_0^2 p_1(x_0) p_2(x_0), \cdots, y_{n-1}^2 p_1(x_{n-1}) p_2(x_{n-1})).$$

Thus $\mathrm{GRS}_k(\mathbf{x}, \mathbf{y})^2 \subseteq \mathrm{GRS}_{2k-1}(\mathbf{x}, \mathbf{y} * \mathbf{y})$ where we assume $2k - 1 \leq n$. This property has been used in [11] to recover non-random columns in a Wieschebrink scheme's public key [34].

Let G be the public key for an (n, k, d, t, w) RLCE encryption scheme based on a GRS code. Let C be the code generated by the rows of G. Let $\mathcal{D}_1$ be the code with a generator matrix D_1 obtained from G by replacing the randomized $2w$ columns with all-zero columns and let $\mathcal{D}_2$ be the code with a generator matrix D_2 obtained from G by replacing the $n - w$ non-randomized columns with zero columns. Since $C \subset \mathcal{D}_1 + \mathcal{D}_2$ and the pair $(\mathcal{D}_1, \mathcal{D}_2)$ is an orthogonal pair, we have $C^2 \subset \mathcal{D}_1^2 + \mathcal{D}_2^2$. It follows that

$$2k - 1 \leq \dim C^2 \leq \min\{2k - 1, n - w\} + 2w \tag{21}$$

where we assume that $2w \leq k^2$. In the following discussion, we assume that *the $2w$ randomized columns in $\mathcal{D}_2$ behave like random columns in the filtration attacks*. For the parameters that we use, we have $k \geq n - w$. In this case, we have $\dim C^2 = \mathcal{D}_1^2 + \mathcal{D}_2^2 = n - w + \mathcal{D}_2^2 = n + w$. Furthermore, for any code C' of length n' that is obtained from C using code puncturing and code shortening, we have $\dim C'^2 = n'$. Thus filtration techniques could not be used to recover any non-randomized columns in D_1.

8.10 Filtration with brute-force attack

In addition to the filtration attacks that we have discussed in the preceding section, the adversary may carry out a filtration attack by exhaustively searching some GRS columns. That is, the adversary randomly selects $u \leq w$ pairs of columns from the public key G with the hope that u columns of the underlying GRS code generator matrix could be reconstructed using exhaustive search. The probability that the u pairs are correctly selected so that u columns of the underlying GRS code generator matrix could be exhaustively searched from these u pairs is bounded by $\frac{\binom{w}{u}}{\binom{n+w}{2u}}$. For each pair $(\mathbf{x}_i, \mathbf{y}_i)$ of columns, one randomly selects two elements $a_i, b_i \in GF(q)$ and computes a column vector $a_i\mathbf{x}_i + b_i\mathbf{y}_i$. In case that the u pairs of column selection is correct, then the probability that these calculated u column vectors are correct GRS code generator matrix columns is bounded by $\frac{1}{q^{2u}}$. In a summary, one can obtain u columns of GRS code generator matrix from the public key with a probability $\frac{\binom{w}{u}}{q^{2u}\binom{n+w}{2u}}$.

Assume that one has correctly guessed u columns of the GRS code generator matrix and $k < n - w + u$. Similar to the discussion in the preceding section, we can distinguish two cases: $n - w + u \geq 2k$ and $n - w + u < 2k$. In case that $n - w + u \geq 2k$, the filtration attack could be carried out straightforwardly. Thus it is recommended to have $n < 2k$ so that $n - w + u \leq n < 2k$. In the following, we consider the case that $n - w + u < 2k$. Let G' be the $k \times (n + w - u)$ matrix consisting of the guessed u columns and the remaining $n - 2u$ columns of the public key. Randomly select $l < k - 1$ columns from the non-guessed columns of G' and shorten G' from these locations. Among the $l = l_1 + l_2$ columns, l_1 columns are non-randomized columns and l_2 columns are from randomized columns. Then the shortened code has dimension

$$d'_{l,l_1} = \min\left\{(k-l)^2, \min\left\{2(k-l_1) - 1, n - w + u - l_1, (k-l)^2\right\} + \min\left\{2(w-u) - l_2, (k-l)^2\right\}\right\}. \tag{22}$$

A necessary condition for the filtration attack to be observable is that, after the shortening of the l_1 columns in G', the following conditions are satisfied

$$d'_{l,l_1} = 2(k - l_1) - 1 + \min\left\{2(w-u) - l_2, (k-l)^2\right\}. \tag{23}$$

Thus for a given l, the probability for the filtration attack to be successful is bounded by the probability

$$\frac{\sum_{l_1=\max\{0,l-2(w-u)\}}^{l} \lambda(d'_{l,l_1})\binom{n-w}{l_1}\binom{2w-2u}{l-l_1}}{\binom{n+w-2u}{l}}$$

where $\lambda(d'_{l,l_1}) = 1$ if (23) holds and $\lambda(d'_{l,l_1}) = 0$ otherwise. A similar discussion as in the preceding section shows that the expected time for one to carry out the filtration attack for a given l and u is

$$PF_{n,k,w,l,u} = \frac{q^{2u}\binom{n+w}{2u}\binom{n+w-2u}{l}\left(O(kl(n+w-2u)) + O((k-l)^4(n+w-2u-l))\right)}{\binom{w}{u}\sum_{l_1=\max\{0,l-2(w-u)\}}^{l} \lambda(d_{l,l_1})\binom{n-w}{l_1}\binom{2w-2u}{l-l_1}}$$

Let

$$\kappa^{fb}_{n,k,w} = \log_2 \min\left\{PF_{n,k,w,l,u} : 2k - n + w - u \leq l \leq k - u - 2, 0 \leq u \leq w\right\}. \tag{24}$$

Then in order to guarantee that the RLCE scheme is secure against filtration attacks, the parameters should be chosen in such a way that $\kappa_c \leq \kappa^{fb}_{n,k,w}$. Similarly, filtration attacks with brute-force could be combined with Grover's quantum search algorithm and, under quantum filtration attacks with brute-force, the RLCE scheme has quantum security level κ^{fb}_q as

$$\log_2 \min_{l,u}\left\{\frac{7\cdot(\log_2 q)^{1.585}\cdot q^{2u}\cdot\left(O(kl(n+w-2u)) + O((k-l)^4(n+w-2u-l))\right)\sqrt{\binom{n+w}{2u}\binom{n+w-2u}{l}}}{\sqrt{\binom{w}{u}\sum_{l_1=\max\{0,l-2(w-u)\}}^{l}\lambda(d_{l,l_1})\binom{n-w}{l_1}\binom{2w-2u}{l-l_1}}}\right\}. \tag{25}$$

Our experiments show that the values $(\kappa^{fb}_{n,k,w}, \kappa^{fb}_q)$ always equal $(\kappa^{f}_{n,k,w}, \kappa^{f}_q)$ with $u = 0$. That is, there is no improvement by using the exhaustive search for filtration attacks.

8.11 Related message attack, reaction attack, and side channel attacks

Berson [7] discussed the following related message attack. Assume that $\mathbf{c}_1 = \mathbf{m}_1 G + \mathbf{e}_1$, $\mathbf{c}_2 = \mathbf{m}_2 G + \mathbf{e}_2$, and that the adversary knows the relation between $\mathbf{m}_1$ and $\mathbf{m}_2$. For example, assume that $\mathbf{m} = \mathbf{m}_1 + \mathbf{m}_2$ and that the adversary knows the value of $\mathbf{m}$. Then we have $\mathbf{c}_1 + \mathbf{c}_2 - \mathbf{m}G = \mathbf{e}_1 + \mathbf{e}_2$. Since $\mathbf{e}_1$ and $\mathbf{e}_1$ are different and both of them have low weight t, it could be easy for the adversary to recover both $\mathbf{e}_1$ and $\mathbf{e}_1$ by trying all combinations. Even if one cannot enumerate all combinations to recover either $\mathbf{e}_1$ or $\mathbf{e}_1$, one can use the 0 entries within $\mathbf{e}_1 + \mathbf{e}_2$ as a hint to speed up the information set decoding algorithm for recovering $\mathbf{m}_1$ from $\mathbf{c}_1 = \mathbf{m}_1 G + \mathbf{e}_1$. A special case of this attack is the attack on two ciphertexts of the identical message encrypted using different error vectors. The related-message-attack could be defeated using appropriate message padding for IND-CCA2 security.

Hall et al [16] discussed the following reaction attack. Assume that an McEliece decryption oracle outputs an error message each time when the given ciphertext contains too many errors to decrypt. For a given ciphertext $\mathbf{c}$, the adversary first randomly selects positions to add errors until the decryption oracle complains. That is, the adversary first obtains a ciphertext $\mathbf{c}'$ that contains maximum errors that the decryption oracle could handle. Then the adversary selects a random position i and add errors to this position. If the decryption oracle could decrypt the resulting ciphertext, it means that $\mathbf{c}'$ contains error at this position. Otherwise, this position is error-free. The adversary continues this process until she obtains k error-free positions for the ciphertext $\mathbf{c}$. These error-free positions could be used to recover the plaintext message for the ciphertext $\mathbf{c}$. The reaction-attack could be defeated using appropriate message padding for IND-CCA2 security.

Message padding schemes for IND-CCA2 security could be used to defeat the reaction attack. However, for a ciphertext that contains too many errors to decrypt and for a ciphertext with padding errors that decrypt successfully, the decryption oracle normally uses different amount of times. Thus an adversary may introduce errors in some positions of the ciphertext and observe the amount of time used for the decryption oracle to report errors. This will allow the adversary to distinguish whether the original ciphertext contains errors in these positions or not. The observed results could be used as in the reaction attack to recover the plaintext. In order to defeat such kind of reaction-attack based side-channel attacks, appropriate delays should be introduced in a decryption process of padded RLCE schemes so that the decryption process takes the same amount of times to report errors for padding errors and for decoding errors.

9 Appendix B: RLCE Performance evaluation (Informative)

9.1 Time cost

Table 4 lists the performance results for RLCE encryption scheme on a MacBook Pro with 2.9 GHz Intel Core i7 and MacOS Sierra. The first column contains the encryption scheme ID from Table 2. The second column contains the time needed for a public/private key pair generation. The third two-column contains the time needed for one plaintext encryption. The fourth two-column contains the time needed for one ciphertext decryption.

Table 4: Running times for RLCE (in milliseconds)

ID	key	encryption	decryption
0	340.616	0.538	1.509
1	1253.926	1.166	2.937
2	3215.791	2.796	12.925

9.2 CPU cycles

Table 5 lists the CPU cycles for RLCE encryption scheme. It was tested with MacOS Sierra on a MacBook Pro with 2.9 GHz Intel Core i7. The first column contains the encryption scheme ID from Table 2. The second column contains CPU cycles for a public/private key pair generation. The third column contains CPU cycles for encrypting a plaintext. The fourth column contains CPU cycles for decrypting a ciphertext.

Table 5: RLCE CPU cycles

ID	key generation	encryption	decryption
0	1011071617	1805010	4646941
1	3829675407	3331234	8668186
2	9612380645	8184051	36705481

9.3 Memory requirements

Table 6 lists the memory requirements for RLCE encryption scheme with decoding algorithms 0, 1, and 2 respectively. It was tested on a Amazon AWS cloud computer running Ubuntu 16.10 with Intel(R) Xeon(R) CPU E5-2630L v2 @ 2.40GHz. The first column is the RLCE scheme ID. The second column shows whether a finite field multiplication table is generated or not. These data shows that for schemes over $GF(2^{10})$ (that is, schemes 0, 1, 2, 3 for 128-bit and 192-bit security), there is around 2MB difference for the RAM requirement with a multiplication table and without a multiplication table. For schemes over $GF(2^{11})$ (that is, schemes 4, 5 for 256-bit security), there is around 7MB difference for the RAM requirement with multiplication table and without multiplication table. In practice, it is convenient to deploy hardware based multiplication tables.

Table 6: RLCE peak memory usage (bytes)

ID	Mul. Table	key generation	encryption	decryption
0	N	2,536,704	798,288	1,335,280
0	Y	4,648,656	2,437,320	2,856,584
1	N	6,178,744	1,906,576	3,178,688
1	Y	8,287,312	2,865,400	3,825,112
2	N	11,561,352	4,829,968	7,010,368
2	Y	19,975,040	10,258,112	12,227,384

9.4 Performance comparison with OpenSSL RSA

Table 7 shows the comparison of the RLCE performance against OpenSSL RSA performance. Both RSA and RLCE were tested with a MacOS Sierra on a MacBook Pro with 2.9 GHz Intel Core i7.

10 Appendix C: Optimized implementation and decoding generalized Reed-Solomon codes (Informative)

This section investigates efficient algorithms for implementing the scheme RLCE. Specifically, we will compare various decoding algorithms for generalized Reed-Solomon (GRS) codes: Berlekamp-Massey decod-

Table 7: Comparison of RLCE and RSA performance (milliseconds)

κ_c	RSA modulus	key setup		encryption		decryption	
		RSA	RLCE	RSA	RLCE	RSA	RLCE
128	3072	433.607	151.834	0.135540	0.360	6.576281	1.345
192	7680	9346.846	637.988	0.672769	0.776	75.075443	2.676
256	15360	80790.751	1587.330	2.498523	1.745	560.225740	9.383

ing algorithms; Berlekamp-Welch decoding algorithms; and Euclidean decoding algorithms. This section also compares various efficient algorithms for polynomial and matrix operations over finite fields. For example, this section will cover Chien's search algorithm; Berlekamp trace algorithm; Forney's algorithm, Strassen algorithm, and many others. The focus of this section is to identify the optimized algorithms for implementing the RLCE encryption scheme on 64-bit CPUs.

10.1 Representation of elements in finite fields

Let p be a prime and $\pi(x)$ be an irreducible polynomial of degree m over $GF(p)$. Then the set of all polynomials in x of degree $\leq m-1$ and coefficients from $GF(p)$ form the finite field $GF(p^m)$ where field elements addition and multiplication are defined as polynomial addition and multiplication modulo $\pi(x)$

For an irreducible polynomial $f(x) \in GF(p)[x]$ of degree m, $f(x)$ has a root α in $GF(p^m)$. Furthermore, all roots of $f(x)$ are given by the m distinct elements $\alpha, \alpha^p, \cdots, \alpha^{p^{m-1}} \in GF(p^m)$. A primitive polynomial $\pi(x)$ of degree m over $GF(p)$ is an irreducible polynomial that has a root α in $GF(p^m)$ so that $GF(p^m) = \{0\} \cup \{\alpha^i : i = 0, \cdots, p^m - 1\}$ where α is called a generator. As an example for $GF(2^3)$, x^3+x+1 is a primitive polynomial with root $\alpha = 010$. That is,

$\alpha^0 = 001$	$\alpha^1 = 010$	$\alpha^2 = 100$	$\alpha^3 = 011$
$\alpha^4 = 110$	$\alpha^5 = 111$	$\alpha^6 = 101$	$\alpha^7 = 001$

Note that not all irreducible polynomials are primitive. For example $1 + x + x^2 + x^3 + x^4$ is irreducible over $GF(2)$ but not primitive. The root of a primitive polynomial is called a primitive element.

10.2 FFT over $GF(2^m)$ and Cantor's algorithm

The Fast Fourier transform maps a polynomial $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$ to its values

$$\text{FFT}(f(x)) = (f(\alpha^0), \cdots, f(\alpha^{n-1})).$$

Fast Fourier Transforms (FFT) are useful for improving RLCE decryption performance. For finite fields with characteristics 2 such as $GF(2^m)$, one may use Cantor's algorithm [10] and its variants [31, 13] for efficient FFT computation. These techniques are also called additive FFT algorithms and could be used to compute FFT($f(x)$) over $GF(2^m)$ in $O(m^2 2^m)$ steps.

Let $\beta_0, \cdots, \beta_{d-1} \in GF(2^m)$ be linearly independent over $GF(2)$ and let B be a subspace spanned by β_i's over $GF(2)$. That is,

$$B = \text{span}(\beta_0, \cdots, \beta_{d-1}) = \left\{ \sum_{j=0}^{d-1} a_j \beta_j : a_j \in GF(2) \right\}.$$

For $0 \leq i < 2^d$ with the binary representation $i = a_{d-1}a_{d-1} \cdots a_0$, the i-th element in B is $B[i] = \sum_{j=0}^{d-1} a_j \beta_j$. For $0 \leq i \leq d-1$, let $W_i = \text{span}(\beta_0, \cdots, \beta_i)$. Then we have

$$\{0\} = W_{-1} \subsetneq W_0 \subsetneq W_1 \subsetneq \cdots \subsetneq W_{d-1}$$

and $W_i = (\beta_i + W_{i-1}) \cup W_i$ for $i = 0, \cdots, d-1$. This can be further generalized to

$$\beta + W_i = (\beta + \beta_i + W_{i-1}) \cup (\beta + W_i)$$

for $i = 0, \cdots, d-1$ and all $\beta \in GF(2^m)$. Next define the minimal polynomial $s_i(x) \in GF(2^m)[x]$ of W_i as

$$s_i(x) = \prod_{\alpha \in W_i} (x - \alpha)$$

for $i = 0, \cdots, d-1$. It is shown in [31] that $s_i(x)$ is a $GF(2)$-linearized polynomial where the concept of linearized polynomial is given in Section 10.6.3. Furthermore, by the fact that

$$s_i(x) = \prod_{\alpha \in W_i} (x - \alpha) = \left(\prod_{\alpha \in W_{i-1}} (x - \alpha) \right) \left(\prod_{\alpha \in \beta_i + W_{i-1}} (x - \alpha) \right) = s_{i-1}(x) \cdot s_{i-1}(x - \beta_i)$$

and by the fact that $s_i(x)$ is a linearized polynomial, we have

$$s_i(x) = s_{i-1}(x) \cdot s_{i-1}(x - \beta_i) = s_{i-1}(x)\,(s_{i-1}(x) - s_{i-1}(\beta_i))$$

for $i = 0, \cdots, d-1$. Table 8 lists the polynomials $s_i(x)$ over $GF(2^{10})$ for the base $\beta_i = b_9 b_8 \cdots b_0$ where $b_j = 0$ for $j \neq i$ and $b_i = 1$.

Table 8: Linearized polynomials $s_i(x)$ over $GF(2^{10})$

$$\begin{aligned}
s_0(x) &= x^2 + x \\
s_1(x) &= x^4 + \mathtt{0x007}x^2 + \mathtt{0x006}x \\
s_2(x) &= x^8 + \mathtt{0x17d}x^4 + +\mathtt{0x205}x^2 + \mathtt{0x379}x \\
s_3(x) &= x^{16} + \mathtt{0x2b5}x^8 + \mathtt{0x3f4}x^4 + \mathtt{0x177}x^2 + \mathtt{0x037}x \\
s_4(x) &= x^{32} + \mathtt{0x18a}x^{16} + \mathtt{0x139}x^8 + \mathtt{0x353}x^4 + \mathtt{0x3f4}x^2 + \mathtt{0x015}x \\
s_5(x) &= x^{64} + \mathtt{0x179}x^{32} + \mathtt{0x0b3}x^{16} + \mathtt{0x303}x^8 + \mathtt{0x09f}x^4 + \mathtt{0x0b2}x^2 + \mathtt{0x2e5}x \\
s_6(x) &= x^{128} + \mathtt{0x394}x^{64} + \mathtt{0x35f}x^{32} + \mathtt{0x28f}x^{16} + \mathtt{0x3ef}x^8 + \mathtt{0x041}x^4 + \mathtt{0x0de}x^2 \\
&\quad +\mathtt{0x135}x \\
s_7(x) &= x^{256} + \mathtt{0x2bd}x^{128} + \mathtt{0x2cf}x^{64} + \mathtt{0x2e1}x^{32} + \mathtt{0x1a5}x^{16} + \mathtt{0x3f4}x^8 + \mathtt{0x279}x^4 \\
&\quad +\mathtt{0x3a8}x^2 + \mathtt{0x112}x \\
s_8(x) &= x^{512} + \mathtt{0x214}x^{256} + \mathtt{0x043}x^{128} + \mathtt{0x292}x^{64} + \mathtt{0x070}x^{32} + \mathtt{0x0ce}x^{16} + \mathtt{0x0b3}x^8 \\
&\quad +\mathtt{0x24c}x^4 + \mathtt{0x081}x^2 + \mathtt{0x204}x
\end{aligned}$$

Table 9 lists the polynomials $s_i(x)$ over $GF(2^{10})$ for the base $\beta_i = b_{10} b_9 \cdots b_0$ where $b_j = 0$ for $j \neq i$ and $b_i = 1$.

With these preliminary definition, we first review von zur Gathen and Gerhard's additive FFT algorithm. Let $\beta_0, \cdots, \beta_{d-1} \in GF(2^m)$ be linearly independent over $GF(2)$ and let $B = \text{span}(\beta_0, \cdots, \beta_{d-1})$. For a given polynomial $f(x)$ of degree less than 2^d, we evaluate $f(x)$ over all points in B using the following algorithm $\mathtt{GGFFT}(f(x), d, B) = \langle f(B[0]), \cdots, f(B[2^d - 1]) \rangle$. The algorithm assumes that the polynomials $s_i(x)$, the values $s_i(\beta)$ and $s_i(\beta_{i+1})^{-1}$ for $-1 \le i < j \le d-1$ are pre-computed.

Gathen-Gerhard's $\mathtt{GGFFT}(f(x), i, d, B, b_{i+1}, \cdots, b_{d-1})$:
`Input:` $i \in [-1, d-1]$, $f \in GF(2^m)[x]$, $\deg(f(x)) < 2^{i+1}$, and $b_{i+1}, \cdots, b_{d-1} \in GF(2)$.
`Output:` $\langle f(\alpha + \beta) : \alpha \in W_i \rangle$ where $\beta = b_{i+1}\beta_{i+1} + \cdots + b_{d-1}\beta_{d-1}$.
`Algorithm:`

Table 9: Linearized polynomials $s_i(x)$ over $GF(2^{11})$

$$\begin{aligned}
s_0(x) &= x^2 + x\\
s_1(x) &= x^4 + \texttt{0x007}x^2 + \texttt{0x006}x\\
s_2(x) &= x^8 + \texttt{0x17d}x^4 + +\texttt{0x60c}x^2 + \texttt{0x770}x\\
s_3(x) &= x^{16} + \texttt{0x4c3}x^8 + \texttt{0x6c0}x^4 + +\texttt{0x390}x^2 + \texttt{0x192}x\\
s_4(x) &= x^{32} + \texttt{0x48a}x^{16} + \texttt{0x278}x^8 + \texttt{0x528}x^4 + \texttt{0x274}x^2 + \texttt{0x1af}x\\
s_5(x) &= x^{64} + \texttt{0x69e}x^{32} + \texttt{0x4ec}x^{16} + \texttt{0x619}x^8 + \texttt{0x4fd}x^4 + \texttt{0x05b}x^2\\
&\quad +\texttt{0x0cc}x\\
s_6(x) &= x^{128} + \texttt{0x734}x^{64} + \texttt{0x294}x^{32} + \texttt{0x357}x^{16} + \texttt{0x4a0}x^8 + \texttt{0x1f8}x^4\\
&\quad +\texttt{0x211}x^2 + \texttt{0x1bf}x\\
s_7(x) &= x^{256} + \texttt{0x50b}x^{128} + \texttt{0x52b}x^{64} + \texttt{0x31b}x^{32} + \texttt{0x0da}x^{16} + \texttt{0x56e}x^8\\
&\quad +\texttt{0x0c0}x^4 + \texttt{0x230}x^2 + \texttt{0x47e}x\\
s_8(x) &= x^{512} + \texttt{0x385}x^{256} + \texttt{0x584}x^{128} + \texttt{0x4b0}x^{64} + \texttt{0x11f}x^{32} + \texttt{0x2ef}x^{16}\\
&\quad +\texttt{0x261}x^8 + \texttt{0x429}x^4 + \texttt{0x68d}x^2 + \texttt{0x185}x\\
s_9(x) &= x^{1024} + \texttt{0x703}x^{512} + \texttt{0x781}x^{256} + \texttt{0x7c9}x^{128} + \texttt{0x7da}x^{64} + \texttt{0x4d2}x^{32}\\
&\quad +\texttt{0x444}x^{16} + \texttt{0x60c}x^8 + \texttt{0x69f}x^4 + \texttt{0x5d7}x^2 + \texttt{0x542}x
\end{aligned}$$

1. If $i = -1$, return f.
2. Compute $g(x), r_0(x) \in GF(2^m)[x]$ such that

$$f(x) = g(x)\,(s_{i-1}(x) + s_{i-1}(\beta)) + r_0(x) \text{ and } \deg(r_0(x)) < 2^{i-1}.$$

   Let $r_1(x) = r_0(x) + s_{i-1}(\beta_i) \cdot g(x)$.
3. Return $\texttt{GGFFT}(r_0(x), i-1, d, B, 0, b_{i+1}, \cdots, b_{d-1}) \cup \texttt{GGFFT}(r_1(x), i-1, d, B, 1, b_{i+1}, \cdots, b_{d-1})$.

It is shown in [31] that the algorithm $\texttt{GGFFT}(f(x), d, B)$ runs with $O(2^d d^2)$ multiplications and additions. We next review Gao-Mateer's FFT algorithm [13] which runs with $O(2^d d)$ multiplications and $O(2^d d^2)$ additions.

Gao-Mateer's $\texttt{GMFFT}(f(x), d, B)$):
`Input:` $f \in GF(2^m)[x]$, $\deg(f(x)) < 2^d$, $B = \text{span}(\beta_0, \cdots, \beta_{d-1})$
`Output:` $\langle f(B[0]), \cdots, f(B[2^d - 1])\rangle$.
`Algorithm:`

1. If $\deg(f(x)) = 0$, return $\langle f(0), f(0)\rangle$.
2. If $d = 1$, return $\langle f(0), f(\beta_1)\rangle$.
3. Let $g(x) = f(\beta_d x)$.
4. Use the algorithm in the next paragraph to compute $\texttt{Taylor}(g(x))$ as in (27) and let

$$g_0(x) = \sum_{i=0}^{l-1} g_{i,0}x^i \quad \text{and} \quad g_1(x) = \sum_{i=0}^{l-1} g_{i,1}x^i. \tag{26}$$

5. Let $\gamma_i = \beta_i\beta_d^{-1}$ and $\delta_i = \gamma_i^2 - \gamma_i$ for $0 \le i \le d-2$.

6. Let $G = \text{span}(\gamma_0, \cdots, \gamma_{d-2})$ and $D = \text{span}(\delta_0, \cdots, \delta_{d-2})$

7. Let
$$\texttt{FFT}(g_0(x), d-1, D) = \langle u_0, \cdots, u_{2^{d-1}-1}\rangle$$
$$\texttt{FFT}(g_1(x), d-1, D) = \langle v_0, \cdots, v_{2^{d-1}-1}\rangle$$

8. Let $w_i = u_i + G[i] \cdot v_i$ and $w_{2^{d-1}+i} = w_i + v_i$ for $0 \le i < 2^{d-1}$.

9. Return $\langle w_0, \cdots, w_{2^d-1}\rangle$.

For a polynomial $g(x)$ of degree $2l-1$ over $GF(2^m)$, the Taylor expansion of $g(x)$ at $x^2 - x$ is a list $\langle g_{0,0} + g_{0,1}x, \cdots, g_{l-1,0} + g_{l-1,1}x\rangle$ where

$$g(x) = (g_{0,0} + g_{0,1}x) + (g_{1,0} + g_{1,1}x)(x^2 - x) + \cdots + (g_{l-1,0} + g_{l-1,1}x)(x^2 - x)^{l-1} \tag{27}$$

and $g_{i,j} \in GF(2^m)$. The Taylor expansion of $g(x)$ could be computed using the following algorithm $\texttt{Taylor}(g(x))$:

1. If $\deg(g(x)) < 2$, return $g(x)$.

2. Find l such that $2^{l+1} < 1 + \deg(g(x)) \le 2^{l+2}$.

3. Let $g(x) = h_0(x) + x^{2^{l+1}}\left(h_1(x) + x^{2^l} h_2(x)\right)$ where $\deg(h_0) < 2^{l+1}, \deg(h_1) < 2^l, \deg(h_2) < 2^l$.

4. Return $\langle \texttt{Taylor}(h_0(x) + x^{2^l}(h_1(x) + h_2(x))), \texttt{Taylor}(h_1(x) + h_2(x) + x^{2^l} h_2(x))\rangle$.

It is shown in [13] that the algorithm $\texttt{GMFFT}$ uses at most $2^{d-1}\log^2(2^d)$ additions and $2^{d+1}\log(2^d)$ multiplications.

10.3 Inverse FFT

For a polynomial $f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1}$, the Inverse FFT is defined as

$$\text{IFFT}(\text{FFT}(f(x))) = \text{IFFT}(f(\alpha^0), \cdots, f(\alpha^{n-1})) = (f_0, \cdots, f_{n-1}).$$

Assume that $n = p^m - 1$ and $\alpha^n = 1$. The Mattson-Solomon polynomial of f is defined as

$$F(x) = \sum_{i=0}^{n-1} f(\alpha^i)x^{n-i}. \tag{28}$$

By the fact that

$$x^n - 1 = (x-1)(1 + x + \cdots + x^{n-1}),$$

we have $\sum_{i=0}^{n-1} a^i = 0$ for all $a \in GF(q)$ with $a \ne 1$. Then

$$\begin{aligned} F(\alpha^j) &= \sum_{i=0}^{n-1} f(\alpha^i)\alpha^{j(n-i)} \\ &= \sum_{i=0}^{n-1}\sum_{u=0}^{n-1} f_u \alpha^{ui}\alpha^{j(n-i)} \\ &= \sum_{u=0}^{n-1} f_u \sum_{i=0}^{n-1} \alpha^{(u-j)i} \\ &= n f_j \end{aligned} \tag{29}$$

It follows that IFFT(FFT($f(x)$)) = FFT$\left(\frac{F(x)}{n}\right)$.

For FFT over $GF(2^m)$ in Section 10.2, the output is in the order $f(B[0]), \cdots, f(B[2^m-1])$ instead of the order $f(\alpha^0), \cdots, f(\alpha^{2^m-1})$. Thus in order to calculate $F(x)$, we need to find a list of indices $j_0, \cdots, j_{2^{m-1}-1}$ such that $B[j_i] = \alpha^i$ for $0 \le i \le 2^{m-1}-1$ (this could be done in $O(2^m)$ steps). Then we can let

$$F(x) = \sum_{i=0}^{n-1} f(B[j_i])x^{n-i}.$$

Similarly, after IFFT($F(x)$) = $(F(B[0]), \cdots, F(B[2^m-1]))$ is obtained, we will have $f_i = F(B[j_i])$ for $0 \le i \le 2^{m-1}-1$.

However, in order to interpolate a polynomial, one essentially needs a base $\{\beta_0, \cdots, \beta_{m-1}\}$ to generate the entire field $GF(2^m)$ and to compute FFT over the entire field $GF(2^m)$. This is inefficient for polynomials whose degrees are much smaller than 2^{m-1}.

In the following, we describe the Chinese Remainder Theorem based IFFT algorithm from von zur Gathen and Gerhard [31] that takes advantage of the additive FFT property. Let $\beta_0, \cdots, \beta_{d-1} \in GF(2^m)$ be linearly independent over $GF(2)$ and let $B = \text{span}(\beta_0, \cdots, \beta_{d-1})$.

Gathen-Gerhard's `GGIFFT`$(i, B, \beta, f(\beta + W_i))$:

`Input:` $i \in [0, d-1]$, β, and $\langle f(\beta + W_i[0]), \cdots, f(\beta + W_i[2^{i+1}-1])\rangle$ where $\beta = \sum_{j=i+1}^{d-1} b_j\beta_j$ for some $b_{i+1}, \cdots, b_{d-1} \in GF(2)$.
`Output:` $f(x) \in GF(2^m)[x]$ with $\deg(f(x)) < 2^{i+1}$.
`Algorithm:`

1. If $i = 0$, then return $f(x) = \beta_0^{-1}(f(\beta) + f(\beta+\beta_0))x + f(\beta) + \beta_0^{-1}\beta(f(\beta) + f(\beta+\beta_0))$.
2. Let $\beta' = \beta + \beta_i$ and
$$\begin{array}{l} f_0(x) = \texttt{GGIFFT}(i-1, B, \beta, f(\beta + W_{i-1})) \\ f_1(x) = \texttt{GGIFFT}(i-1, B, \beta', f(\beta' + W_{i-1})) \end{array}$$
where $\deg(f_0(x)) < 2^i$ and $\deg(f_1(x)) < 2^i$.
3. Return $f(x) = (s_{i-1}(x) + s_{i-1}(\beta)) \cdot (f_0(x) + f_1(x)) \cdot s_{i-1}(\beta_i)^{-1} + f_0(x)$.

10.4 Polynomial multiplication I: Karatsuba algorithm

For two polynomials $f(x)$ and $g(x)$, we can rewrite them as

$$f(x) = f_1(x)x^{n_1} + f_2(x) \qquad \text{and} \qquad g(x) = g_1(x)x^{n_1} + g_2(x)$$

where f_1, f_2, g_1, g_2 has degree less than n_1. Then

$$f(x)g(x) = h_1(x)x^{2n_1} + h_2(x)x^{n_1} + h_3(x)$$

where

$$\begin{array}{l} h_1(x) = f_1(x)g_1(x) \\ h_2(x) = (f_1(x) + f_2(x))(g_1(x) + g_2(x)) - h_1(x) - h_3(x) \\ h_3(x) = f_2(x)g_2(x) \end{array}$$

Karatsuba's algorithm could be recursively called and the time complexity is $O(n^{1.59})$. Our experiments show that Karatsuba's algorithm could improve the efficiency of RLCE scheme for most security parameters.

10.5 Polynomial multiplication II: FFT

For RLCE over $GF(p^m)$, one can use FFT to speed up the polynomial multiplication and division. For two polynomials $f(x)$ and $g(x)$, we first compute FFT$(f(x))$ and FFT$(g(x))$ in at most $O(n\log^2 n)$ steps. With n more multiplications, we obtain FFT$(f(x)g(x))$. From FFT$(f(x)g(x))$, the interpolation can be computed using the inverse FFT as $f(x)g(x) = \text{FFT}^{-1}(f(x)g(x))$. This can be done in $O(n\log^2 n)$ steps. Thus polynomial multiplication can be done in $O(n\log^2 n)$ steps. Our experiments show that FFT based polynomial multiplication helps none of the RLCE encryption schemes.

10.6 Factoring polynomials and roots-finding

10.6.1 Exhaustive search algorithms

The problem of finding roots of a polynomial $\Lambda(x) = 1 + \lambda_1 x + \cdots + \lambda_t x^t$ could be solved by an exhaustive search in time $O(tp^m)$. Alternatively, one may use Fast Fourier Transform that we have discussed in the preceding sections to find roots of $\Lambda(x)$ using at most $m^2 p^m \log^2(p)$ steps. Furthermore, one may also use Chien's search to find roots of $\Lambda(x)$. Chien's search is based on the following observation.

$$\begin{array}{rcl}
\Lambda(\alpha^i) & = & 1 + \lambda_1\alpha^i + \cdots + \lambda_t(\alpha^i)^t \\
 & = & 1 + \lambda_{1,i} + \cdots + \lambda_{t,i} \\
\Lambda(\alpha^{i+1}) & = & 1 + \lambda_1\alpha^{i+1} + \cdots + \lambda_t(\alpha^{i+1})^t \\
 & = & 1 + \lambda_{1,i}\alpha + \cdots + \lambda_{t,i}\alpha^t \\
 & = & 1 + \lambda_{1,i+1} + \cdots + \lambda_{t,i+1}
\end{array}$$

Thus, it is sufficient to compute the set $\{\lambda_{j,i} : i = 1, \cdots, q-1; j = 1, \cdots, t\}$ with $\lambda_{j,i+1} = \lambda_{j,i}\alpha^j$. Chien's algorithm can be used to improve the performance of RLCE encryption schemes when 64-bits $\oplus$ is used for parallel field additions. For non-64 bits CPUs, Chien's search does not provide advantage over exhaustive search algorithms. Our experiments show that for all security levels, Chien's search has better performance than FFT-based search and exhaustive search.

10.6.2 Berlekamp Trace Algorithm

Berlekamp Trace Algorithm (BTA) can find the roots of a degree t polynomial in time $O(mt^2)$. A polynomial $f(x) = f_0 + f_1 x + \cdots + f_t x^t$ has no repeated roots if $\gcd(f(x), f'(x)) = 1$. Without loss of generality, we may assume that $f(x)$ has no repeated roots. For each $x \in GF(p^m)$, the trace of x is defined as

$$\text{Tr}(x) = \sum_{i=0}^{m-1} x^{p^i}.$$

We recall that if we consider $GF(p^m)$ as a m-dimensional vector space over $GF(p)$, then a trace function is linear. That is, $\text{Tr}(ax + by) = \text{Tr}(ax) + \text{Tr}(bx)$ for $a, b \in GF(p)$ and $x, y \in GF(p^m)$. Furthermore, we have $\text{Tr}(x^p) = \text{Tr}(x)$ for $x \in GF(p^m)$ and $\text{Tr}(a) = ma$ for $a \in GF(p)$. It is known that in $GF(p^m)$, we have

$$x^{p^m} - x = \prod_{s\in GF(p)} (\text{Tr}(x) - s)\,. \tag{30}$$

Let α be the root of a primitive polynomial of degree m over $GF(p)$. Then $(1, \alpha, \cdots, \alpha^{m-1})$ is a polynomial basis for $GF(p^m)$ over $GF(p)$ and $(\alpha, \cdots, \alpha^{p^{m-1}})$ is a normal basis for $GF(p^m)$ over $GF(p)$. Substituting $\alpha^i x$ for x in equation (30), we get

$$(\alpha^i)^{p^m} x^{p^m} - \alpha^i x = \prod_{s\in GF(p)} \left(\text{Tr}(\alpha^i x) - s\right).$$

This implies

$$x^{p^m} - x = \alpha^{-i} \prod_{s \in GF(p)} \left(\mathrm{Tr}(\alpha^i x) - s \right).$$

If $f(x)$ is a nonlinear polynomial that splits in $GF(p^m)$, then $f(x) | (x^{p^m} - x)$. Thus we have

$$f(x) = \prod_{s \in GF(p)} \gcd\left(f(x), \mathrm{Tr}(\alpha^i x) - s \right). \tag{31}$$

By applying equation (31) with $i = 0, 1, \cdots, m-1$ or $i = 1, p, \cdots, p^{m-1}$, we can factor $f(x)$. In order to speed up the computation of $\mathrm{Tr}(\alpha^i x)$ modulo $f(x)$, one pre-computes the residues of $x, x^2, \cdots, x^{p^m}$ modulo $f(x)$. By adding these residues, one gets the residue of $\mathrm{Tr}(x)$. Furthermore, by multiplying these residues with $\alpha^i, \alpha^{2i}, \cdots, \alpha^{ip^m}$ respectively, one obtains the residue of $\mathrm{Tr}(\alpha^i x)$.

For RLCE implementation over $GF(2^m)$, the BTA algorithm can be described as follows.

Input: A polynomial $f(x)$ and pre-compute $\mathrm{Tr}_i(x) = x^{2^i} \mod f(x)$ for $i = 1, \cdots, m$.
Output: A list of roots $(r_0, \cdots, r_{n_f}) = \mathrm{BTA}(f(x))$.
Algorithm:

1. Let $j = 0$.
2. If $f(x) = x + \alpha$, return α.
3. Use $\mathrm{Tr}_i(x)$ to compute $\mathrm{Tr}(\alpha^j x) \mod f(x)$.
4. If $j > m$, return $\emptyset$.
5. Let $p(x) = \gcd(\mathrm{Tr}(\alpha^j x), f(x))$ and $q(x) = \frac{f(x)}{p(x)}$.
6. Let $j = j + 1$ and return $\mathrm{BTA}(p(x)) \cup \mathrm{BTA}(q(x))$.

BTA algorithm converts one multiplication into several additions. In RLCE scheme, field multiplication is done via table look up. Our experiments show that BTA algorithm is slower than Chien's search or exhaustive search algorithms for RLCE encryption scheme.

10.6.3 Linearized and affine polynomials

In the preceding section, we showed how to compute the roots of polynomials using BTA algorithm. In practice, one factors a polynomial using BTA algorithm until degree four or less. For polynomials of lower degrees (e.g., lower than 4), one can use affine multiple of polynomials to find the roots of the polynomial more efficiently (see., e.g., Berlekamp [4, Chapter 11]). We first note that a linearized polynomial over $GF(p^m)$ is a polynomial of the form

$$g(x) = \sum_{i=0}^{n} g_i x^{p^i}$$

with $g_i \in GF(p^m)$. Note that for a linearized polynomial g, we have $g(ax + by) = g(ax) + g(bx)$ for $a, b \in GF(p)$ and $x, y \in GF(p^m)$. An affine polynomial is a polynomial in the form $a(x) = g(x) + a$ where $g(x)$ is a linearized polynomial and $a \in GF(p^m)$. For small degree polynomials, one can convert it to an affine polynomial which is a multiple of the given polynomial. The root of the affine polynomial could be found by solving a linear equation system of m equations.

The roots of a degree t polynomial $f(x)$ are calculated as follows. At step $i \geq 0$, one computes a degree $2^{\lceil \log_2 t \rceil + i}$ affine multiple of $f(x)$. The roots of the affine polynomial could be found by solving the following linear equation system of order m over $GF(2)$. If the system has no solution, one moves to step $i + 1$.

Let $A(x) = g(x) + c = \sum_{i=0}^{n} g_i x^{p^i} + c$ be an affine polynomial and $\alpha^0, \alpha, \cdots, \alpha^{m-1}$ be a polynomial basis for $GF(2^m)$ over $GF(2)$. Let $c = c_0\alpha^0 + \cdots + c_{m-1}\alpha^{m-1}$ and $x = x_0\alpha^0 + \cdots + x_{m-1}\alpha^{m-1} \in GF(2^m)$ be a root for $A(x)$. Then we have the following linear equation system:

$$\begin{aligned}
A(x) = 0 \quad &\Longleftrightarrow \quad g(x) = c \\
&\Longleftrightarrow \quad g\left(\sum_{i=0}^{m-1} x_i\alpha^i\right) = \sum_{i=0}^{m-1} x_i \cdot g(\alpha^i) = \sum_{i=0}^{m-1} c_i\alpha^i = c \\
&\Longleftrightarrow \quad \sum_{i=0}^{m-1}\left(x_i \sum_{j=0}^{n} g_j \alpha^{ip^j}\right) = \sum_{i=0}^{m-1} c_i\alpha^i \\
&\Longleftrightarrow \quad \sum_{i=0}^{m-1}\left(x_i \sum_{j=0}^{m-1} e_{i,j} \alpha^{j}\right) = \sum_{i=0}^{m-1} c_i\alpha^i \\
&\Longleftrightarrow \quad \sum_{i=0}^{m-1}\left(\alpha^i \sum_{j=0}^{m-1} x_j e_{j,i}\right) = \sum_{i=0}^{m-1} c_i\alpha^i
\end{aligned}$$

That is, $c_i = \sum_{j=0}^{m-1} x_j e_{j,i}$ for $i = 0, \cdots, m$ where $e_j = (e_{j,0}, \cdots, e_{j,m-1}) = \sum_{i=0}^{n} g_i \alpha^{jp^i}$. The linear system could also be written as:

$$\begin{pmatrix} e_{0,0} & e_{1,0} & \cdots & e_{m-1,0} \\ e_{0,1} & e_{1,1} & \cdots & e_{m-1,1} \\ \vdots & \vdots & \ddots & \ldots \\ e_{0,m-1} & e_{1,1} & \cdots & e_{m-1,m-1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{m-1} \end{pmatrix} = \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} \tag{32}$$

For the affine polynomial $x^2 + ax + c$. We consider two cases. For $a = 0$, the square root of c could be calculated directly as $c^{p^{m-1}}$. For $a \neq 0$, we substitute x with $x = ay$ and obtain a new polynomial $y^2 + y + \frac{c}{a^2}$. Thus we have $e_j = \alpha^j + \alpha^{2j}$ which could be pre-computed. For a polynomial $p(x) = x^3 + ax^2 + bx + c$, it has a degree 4 affine multiple polynomial $p_1(x) = (x+a)(x^3 + ax^2 + bx + c) = x^4 + (a^2+b)x^2 + (ab_1 + c)x + ac$. For a degree 4 polynomial $p(x) = x^4 + ax^3 + bx^2 + cx + d$, let $x = y + \sqrt{\frac{c}{a}}$. We obtain $p(y) = y^4 + ay^3 + (a\sqrt{\frac{c}{a}} + b)y^2 + (\frac{cb}{a} + d)$. Next let $z = \frac{1}{y}$. Then we have the affine polynomial $p(z) = z^4 + \frac{a\sqrt{\frac{c}{a}}+b)}{\frac{bc}{a}+d} z^2 + \frac{a}{\frac{cb}{a}+d} z + \frac{1}{\frac{cb}{a}+d}$. For the affine polynomial $x^4 + ax^2 + bx + c$, we have $e_j = b\alpha^j + a\alpha^{2j} + \alpha^{4j}$. For the affine polynomial $x^8 + ax^4 + bx^2 + dx + c$, we have $e_j = d\alpha^j + b\alpha^{2j} + a\alpha^{4j} + \alpha^{8j}$.

As a special case, we consider the roots for quadratic polynomials over the finite fields $GF(2^{10})$ and $GF(2^{11})$. For $p(x) = x^2 + x + c$ over $GF(2^m)$ with $c \neq 0$, $p(x)$ has a root if and only if $\mathrm{Tr}(x) = 0$. Let $c = c_0 + c_1\alpha + \cdots + c_{m-1}\alpha^{m-1}$ and $\mathrm{Tr}(x) = 0$. Then the roots for $p(x)$ are $x = x_0 + x_1\alpha + \cdots + x_{m-1}\alpha^{m-1}$ and $x + 1$ where

1. If $m = 10$, then

$$
\begin{aligned}
x_9 &= c_3 + c_5 + c_6 + c_9 \\
x_8 &= c_3 + c_5 + c_6 \\
x_7 &= c_0 + c_1 + c_2 + c_4 + c_5 + c_8 + c_9 \\
x_6 &= c_0 + c_5 \\
x_5 &= c_0 \\
x_4 &= c_8 + c_9 \\
x_3 &= c_0 + c_3 \\
x_2 &= c_0 + c_1 + c_2 + c_3 + c_6 + c_9 \\
x_1 &= c_1 + c_3 + c_5 + c_6 + c_9 \\
x_0 &= 0
\end{aligned}
$$

2. If $m = 11$, then

$$
\begin{aligned}
x_{10} &= c_5 + c_7 + c_9 + c_{10} \\
x_9 &= c_3 + c_5 + c_6 + c_9 + c_{10} \\
x_8 &= c_3 + c_6 \\
x_7 &= c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_8 + c_{10} \\
x_6 &= c_9 + c_{10} \\
x_5 &= c_3 + c_5 + c_6 + c_8 + c_9 + c_{10} \\
x_4 &= c_1 + c_2 + c_3 + c_4 + c_5 + c_8 + c_{10} \\
x_3 &= c_3 + c_4 + c_5 + c_6 + c_8 + c_9 + c_{10} \\
x_2 &= c_2 + c_3 + c_4 + c_5 + c_6 + c_8 + c_{10} \\
x_1 &= c_0 \\
x_0 &= 0
\end{aligned}
$$

10.7 Matrix multiplication and inverse: Strassen algorithm

Strassen algorithm is more efficient than the standard matrix multiplication algorithm. Assume that A is a $n_1 \times n_2$ matrix, B is a $n_2 \times n_3$ matrix, and all n_1, n_2, n_3 are even numbers. Then $C = AB$ could be computed by first partition A, B, C as follows

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

where $A_{i,j}$ are $\frac{n_1}{2} \times \frac{n_2}{2}$ matrices, $B_{i,j}$ are $\frac{n_2}{2} \times \frac{n_3}{2}$ matrices, and $B_{i,j}$ are $\frac{n_1}{2} \times \frac{n_3}{2}$ matrices. Then we compute the following 7 matrices of appropriate dimensions:

$$
\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}
$$

Next the $C_{i,j}$ can be computed as follows:

$$
\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_6
\end{aligned}
$$

The process can be carried out recursively until A and B are small enough (e.g., of dimension around 30) to use standard matrix multiplication algorithms. Note that if the numbers of rows or columns are odd, we can add zero rows or columns to the matrix to make these numbers even. Please note that in Strassen's original paper, the performance is analyzed for square matrices of dimension $u2^v$ where v is the recursive steps and u is the matrix dimension to stop the recursive process. For a matrix of dimension n, Strassen recommend $n \leq u2^v$. Our experiments show that Strassen matrix multiplication could be used to speed up RLCE encryption scheme for several security parameters.

For matrix inversion, let

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, A^{-1} = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Then we compute

$$\begin{aligned} M_1 &= A_{1,1}^{-1} \\ M_2 &= A_{2,1}M_1 \\ M_3 &= M_1A_{1,2} \\ M_4 &= A_{2,1}M_3 \\ M_5 &= M_4 - A_{2,2} \\ M_6 &= M_5^{-1} \\ C_{1,2} &= M_3M_6 \\ C_{2,1} &= M_6M_2 \\ M_7 &= M_3C_{2,1} \\ C_{1,1} &= M_1 - M_7 \\ C_{2,2} &= -M_6 \end{aligned}$$

Similarly, for matrices with odd dimensions, we can add zero rows/columns and identity matrices in the lower right corner to carry out the computation recursively.

Strassen inversion algorithm generally has better performance than Gauss elimination based algorithm. However, it has high incorrect abortion rate. Thus it is not useful for RLCE encryption schemes. For example, Strassen inversion algorithm will abort on the following matrix over $GF(2^{10})$ though its inverse does exist. The following matrix is a common matrix for which the matrix inverse is needed in RLCE implementation.

$$\begin{pmatrix} 0 & 313 & 0 & 626 & 252 & 266 & 62 & 841 & 0 & 506 & 0 \\ 0 & 0 & 0 & 636 & 389 & 357 & 852 & 638 & 0 & 869 & 0 \\ 0 & 0 & 701 & 656 & 635 & 143 & 130 & 392 & 0 & 278 & 0 \\ 0 & 0 & 711 & 433 & 1020 & 841 & 46 & 185 & 1000 & 369 & 0 \\ 0 & 0 & 813 & 692 & 219 & 657 & 579 & 0 & 13 & 777 & 0 \\ 0 & 0 & 350 & 923 & 632 & 270 & 950 & 0 & 228 & 105 & 0 \\ 0 & 0 & 105 & 445 & 0 & 954 & 916 & 0 & 809 & 268 & 0 \\ 0 & 0 & 963 & 217 & 0 & 619 & 903 & 0 & 566 & 442 & 0 \\ 0 & 0 & 0 & 455 & 0 & 815 & 219 & 0 & 708 & 242 & 0 \\ 129 & 0 & 0 & 334 & 0 & 702 & 481 & 0 & 0 & 614 & 0 \\ 769 & 0 & 0 & 4 & 0 & 729 & 955 & 0 & 0 & 545 & 433 \end{pmatrix}$$

Note that in order to avoid the incorrect abortion in Strassen inversion algorithm, one may use the Bunch-Hopcroft [8] triangular factorization approach LUP combined with Strassen inversion algorithm. Since the LUP factorization requires additional steps for factorization, it will not improve the performance for RLCE encryption schemes and we did not implement it. Alternatively, one may use the Method of Four

Russians for Inversion (M4RI) [2] to speed up the matrix inversion process. Our analysis shows that the M4RI performance gain for RLCE encryption scheme is marginal. Thus we did not implement it either.

10.8 Vector matrix multiplication: Winograd algorithm

Winograd's algorithm can be used to reduce the number of multiplication operations in vector matrix multiplication by 50%. Note that this approach could also be used for matrix multiplication. The algorithm is based on the following algorithm for inner product computation of two vectors $x = (x_0, \cdots, x_{n-1})$ and $y = (y_0, \cdots, y_{n-1})$. We first compute

$$\bar{x} = \sum_{j=0}^{\lfloor \frac{n}{2}-1 \rfloor} x_{2j}x_{2j+1} \qquad \text{and} \qquad \bar{y} = \sum_{j=0}^{\lfloor \frac{n}{2}-1 \rfloor} y_{2j}y_{2j+1}$$

Then the inner product $x \cdot y$ is given by

$$x \cdot y = \begin{cases} \sum_{j=0}^{\lfloor \frac{n}{2}-1 \rfloor} (x_{2j} + y_{2j+1})(x_{2j+1} + y_{2j}) - \bar{x} - \bar{y} & n \text{ is even} \\ \sum_{j=0}^{\lfloor \frac{n}{2}-1 \rfloor} (x_{2j} + y_{2j+1})(x_{2j+1} + y_{2j}) - \bar{x} - \bar{y} + x_{n-1}y_{n-1} & n \text{ is odd} \end{cases}$$

The Winograd algorithm converts each field multiplication into several field additions. Our experiments show that Winograd algorithm is extremely slow for RLCE encryption implementations when table look up is used for field multiplication.

10.9 Experimental results

We have implemented these algorithms that we have discussed in the preceding sections. Table 10 gives experimental results on finding roots of error locator polynomials in RLCE schemes. The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor. The reported time is the required milliseconds for finding roots of a degree t polynomial over $GF(2^{10})$ (an average of 10,000 trials). These results show that generally Chien's search is the best choice.

Table 10: Milliseconds for finding roots of a degree t error locator polynomial over $GF(2^{10})$

t	FFT	Chien Search	Exhaustive search	BTA
78	.4781572	.2871678	.7360182	1.1814685
80	.5021798	.2864403	.7506306	1.2784691
114	.6632026	.4155929	1.0445943	1.9991356
118	.6892365	.4280331	1.0773125	2.1493591
230	1.3742336	.8323220	2.0717924	5.7388549
280	1.7690640	1.0194170	2.4806118	8.3730290

On the other hand, for very small degree polynomials (that is, degree less than 4), linearized and affine polynomial based approach is the best choice. Table 11 gives experimental results on finding roots of small degree polynomials. These polynomial degrees are the common degrees for polynomials in list-decoding based RLCE schemes. The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor. The reported time is the required milliseconds for finding roots of a degree t polynomial over $GF(2^{10})$ (an average of 10,000 trials). These results show that for degree 4 or less, the linearized and affine polynomial based BTA is the best choice. For degrees above 4, Chien's search is the best choice.

Table 11: Milliseconds for finding roots of a small degree t polynomial over $GF(2^{10})$

t	Chien Search	BTA	FFT	Exhaustive search
4	.0197496	.0009202	.1117984	.1175816
6	.0261202	.0537054	.1174620	.1252327
8	.0330730	.1215397	.1402607	.1419983
10	.0418521	.1288605	.1417330	.1605130
14	.0537797	.1780427	.1481447	.1908748
18	.0669920	.2288600	.1805597	.2228205

Table 12 gives experimental results for RLCE polynomial multiplications. The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor. The reported time is the required milliseconds for multiplying a degree t polynomial with a degree $2t$ polynomial over $GF(2^{10})$ (an average of 10,000 trials). From the experiment, it shows that Karatsuba's polynomial algorithm only outperforms standard polynomial algorithm for polynomial degrees above degree 115. It is noted that in standard test, Karatsuba's polynomial algorithm outperforms standard polynomial algorithm for polynomial degrees above degree 35 already.

Table 12: Milliseconds for multiplying a pair of degree t and $2t$ polynomials over $GF(2^{10})$

t	Karatsuba	Standard Algorithm	FFT
78	.0470269	.0374369	1.4651561
80	.0546122	.0423766	1.4891211
114	.0794242	.0775524	2/4723263
118	.0811117	.0833309	2.5360034
230	.2371405	.3117507	6.3380415
280	.3444224	.4547458	7.8866734

Table 13 gives experimental results for RLCE related matrix multiplications. The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor. The reported time is the required seconds for multiplying two $n \times n$ matrices (or invert an $n \times n$ matrix) over $GF(2^{10})$ (an average of 100 trials).

10.10 Reed-Solomon codes

10.10.1 The original approach

Let $k < n < q$ and $a_0, \cdots, a_{n-1}$ be distinct elements from $GF(q)$. The Reed-Solomon code is defined as

$$C = \{(m(a_0), \cdots, m(a_{n-1})) : m(x) \text{ is a polynomial over } GF(q) \text{ of degree } < k\}.$$

Table 13: Seconds for multiplying a pairs of (inverting a) $n \times n$ matrices over $GF(2^{10})$

n	Strassen Mul.	Standard Mul.	Winograd Mul.	Gauss Elimination Inv	Strassen Inv.
376	.17881616	.15684892	.57614453	.23071715	.22307581
470	.42498317	.30317405	1.12305698	.44601063	.53218560
618	.77971244	.65356388	2.68176523	.97155253	.98632941
700	1.01458090	.94067030	3.77942598	1.41453963	1.30181261
764	1.20244299	1.21845951	4.88860081	1.82576160	1.55965069
800	1.36761960	1.605249880	6.27596202	2.14227823	1.80930063

There are two ways to encode k-element messages within Reed-Solomon codes. In the original approach, the coefficients of the polynomial $m(x) = m_0 + m_1x + \cdots + m_{k-1}x^{k-1}$ is considered as the message symbols. That is, the generator matrix G is defined as

$$G = \begin{pmatrix} 1 & \cdots & 1 \\ a_0 & \cdots & a_{n-1} \\ \vdots & \ddots & \vdots \\ a_0^{k-1} & \cdots & a_{n-1}^{k-1} \end{pmatrix}$$

and the the codeword for the message symbols $(m_0, \cdots, m_{k-1})$ is $(m_0, \cdots, m_{k-1})G$.

Let α be a primitive element of $GF(q)$ and $a_i = \alpha^i$. Then it is observed that Reed-Solomon code is cyclic when $n = q - 1$. For each $j > 0$, let $\mathbf{m} = (m_0, \cdots, m_{k-1})$ and $\mathbf{m}' = (m_0\alpha^0, m_1\alpha^1, \cdots, m_{k-1}\alpha^{k-1})$. Then $m'(\alpha^i) = m_0\alpha^0 + m_1\alpha^1\alpha^i + \cdots + m_{k-1}\alpha^{k-1}\alpha^{i(k-1)} = m(\alpha^{i+1})$. That is, $\mathbf{m}'$ is encoded as

$$\left(m'(\alpha^0), \cdots, m'(\alpha^{n-1})\right) = \left(m(\alpha), \cdots, m(\alpha^{n-1}), m(\alpha^0)\right)$$

which is a cyclic shift of the codeword for $\mathbf{m}$.

Instead of using coefficients to encode messages, one may use $m(a_0), \cdots, m(a_{k-1})$ to encode the message symbols. This is a systematic encoding approach and one can encode a message vector using Lagrange interpolation.

10.10.2 The BCH approach

We first give a definition for the t-error-correcting BCH codes of distance δ. Let $1 \le \delta < n = q - 1$ and let $g(x)$ be a polynomial over $GF(q)$ such that $g(\alpha^b) = g(\alpha^{b+1}) = \cdots = g(\alpha^{b+\delta-2}) = 0$ where α is a primitive n-th root of unity (note that it is not required to have $\alpha \in GF(q)$). It is straightforward to check that $g(x)$ is a factor of $x^n - 1$. For $w = n - \deg(g) - 1$, a message polynomial $m(x) = m_0 + m_1x + \cdots + m_wx^w$ over $GF(q)$ is encoded as a degree $n - 1$ polynomial $c(x) = m(x)g(x)$. A BCH codes with $b = 1$ is called a narrow-sense BCH code. A BCH code with $n = q^m - 1$ is called a primitive BCH code where m is the multiplicative order of q modulo n. That is, m is the least integer so that $\alpha \in GF(q^m)$.

A BCH code with $n = q - 1$ and $\alpha \in GF(q)$ is called a Reed-Solomon code. Specifically, let $1 \le k < n = q - 1$ and let $g(x) = (x - \alpha^b)(x - \alpha^{b+1}) \cdots (x - \alpha^{b+n-k-1}) = g_0 + g_1x + \cdots + g_{n-k}x^{n-k}$ be a polynomial over $GF(q)$. Then a message polynomial $m(x) = m_0 + m_1x + \cdots + m_{k-1}x^{k-1}$ is encoded as a degree $n - 1$ polynomial $c(x) = m(x)g(x)$. In other words, the Reed-Solomon code is the cyclic code generated by the polynomial $g(x)$. The generator matrix for this definition is as follows:

$$G = \begin{pmatrix} g_0 & g_1 & \cdots & g_{n-k} & 0 & \cdots & 0 \\ 0 & g_0 & \cdots & g_{n-k-1} & g_{n-k} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & g_{n-2k+1} & g_{n-2k+2} & \cdots & g_{n-k} \end{pmatrix} = \begin{pmatrix} g(x) \\ xg(x) \\ \vdots \\ x^{k-1}g(x) \end{pmatrix}$$

For BCH systematic encoding, we first choose the coefficients of the k largest monomials of $c(x)$ as the message symbols. Then we set the remaining coefficients of $c(x)$ in such a way that $g(x)$ divides $c(x)$. Specifically, let $c_r(x) = m(x) \cdot x^{n-k} \mod g(x)$ which has degree $n-k-1$. Then $c(x) = m(x) \cdot x^{n-k} - c_r(x)$ is a systematic encoding of $m(x)$. The code polynomial $c(x)$ can be computed by simulating a LFSR with degree $n-k$ where the feedback tape contains the coefficients of $g(x)$.

10.10.3 The equivalence

The equivalence of the two definitions for Reed-Solomon code could be established using the relationship between FFT and IFFT. For each Reed-Solomon codeword $f(x)$ in the BCH approach, it is a multiple of the generating polynomial $g(x) = \prod_{j=1}^{n-k} \left(x - \alpha^j\right)$. Let $F(x)$ be defined as in (28). Since $f(\alpha^j) = 0$ for $1 \le j \le n-k$, $F(x)$ has degree at most $k-1$. By the identity (29), we have

$$\text{FFT}(F(x)) = \left(F(\alpha^0), \cdots, F(\alpha^{n-1})\right) = n \cdot f(x).$$

Thus $f(x)$ is also a Reed-Solomon codeword in the original approach.

For each Reed-Solomon codeword $(a_0, \cdots, a_{n-1})$ in the original approach, it is an evaluation of a polynomials $F(x)$ of degree at most $k-1$ on $\alpha^0, \cdots, \alpha^{n-1}$. Let $f(x)$ be the function satisfying the identity (28) obtained by interpolation. Then $f(x) = \text{FFT}\left(\frac{F(x)}{n}\right)$, $(a_0, \cdots, a_{n-1})$ is the coefficients of $n \cdot f(x)$, and $f(\alpha^j) = 0$ for $j = 1, \cdots, n-k$. Thus $f(x)$ is a multiple of the generating polynomial $g(x)$.

10.10.4 Generalized Reed-Solomon codes

For an $[n, k]$ generator matrix G for a Reed-Solomon code, we can select n random elements $v_0, \cdots, v_{n-1} \in GF(q)$ and define a new generator matrix

$$G(v_0, \cdots, v_{n-1}) = G \begin{pmatrix} v_0 & 0 & \cdots & 0 \\ 0 & v_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & v_{n-1} \end{pmatrix} = G \cdot \text{diag}(v_0, \cdots, v_{n-1}).$$

The code generated by $G(v_0, \cdots, v_{n-1})$ is called a generalized Reed-Solomon code. For a generalized Reed-Solomon codeword $\mathbf{c}$, it is straightforward that $\mathbf{c} \cdot \text{diag}\left(v_0^{-1}, \cdots, v_{n-1}^{-1}\right)$ is a Reed-Solomon codeword. Thus the problem of decoding generalized Reed-Solomon codes could be easily reduced to the problem of decoding Reed-Solomon codes.

10.11 Decoding Reed-Solomon code

10.11.1 Peterson-Gorenstein-Zierler decoder

This sections describes Peterson-Gorenstein-Zierler decoder which has computational complexity $O(n^3)$. Assume that Reed-Solomon code is based on BCH approach and the received polynomial is

$$r(x) = c(x) + e(x) = r_0 + r_1 x + \cdots + r_{n-1} x^{n-1}.$$

We first calculate the syndromes $S_j = r(\alpha^j)$ for $j = 1, \cdots, n-k$.

$$\begin{aligned} S_j &= r_0 + r_1\alpha^j + \cdots + r_{n-1}(\alpha^j)^{n-1} \\ &= r_0 + r_{1,j} + \cdots + r_{n-1,j} \\ S_{j+1} &= r_0 + r_1\alpha^{j+1} + \cdots + r_{n-1}(\alpha^{j+1})^{n-1} \\ &= r_0 + r_{1,j}\alpha + \cdots + r_{n-1,j}\alpha^{n-1} \\ &= r_0 + r_{1,j+1} + \cdots + r_{n-1,j+1} \end{aligned}$$

From the above equations, it is sufficient to compute the set $\{r_{i,j} : i = 1, \cdots, n-1; j = 1, \cdots, n-k\}$ with $r_{i,j+1} = r_{i,j}\alpha^i$ and then add them together to get the syndromes.

Let the numbers $0 \le p_1, \cdots, p_t \le n-1$ be error positions and e_{p_i} be error magnitudes (values). Then

$$e(x) = \sum_{i=1}^{t} e_{p_i} x^{p_i}.$$

For convenience, we will use $X_i = \alpha^{p_i}$ to denote error locations and $Y_i = e_{p_i}$ to denote error magnitudes. It should be noted that for the syndromes S_j for $j = 1, \cdots, n-k$, we have

$$S_j = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) = e(\alpha^j) = \sum_{i=1}^{t} e_{p_i}(\alpha^j)^{p_i} = \sum_{i=1}^{t} Y_i X_i^j.$$

That is, we have

$$\begin{pmatrix} X_1^1 & X_2^1 & \cdots & X_t^1 \\ X_1^2 & X_2^2 & \cdots & X_t^2 \\ \vdots & \vdots & \ddots & \vdots \\ X_1^{n-k} & X_2^{n-k} & \cdots & X_t^{n-k} \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_t \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{n-k} \end{pmatrix} \tag{33}$$

Thus we obtained $n-k$ equations with $n-k$ unknowns: $X_1, \cdots, X_t, Y_1, \cdots, Y_t$. The error locator polynomial is defined as

$$\Lambda(x) = \prod_{i=1}^{t}(1 - X_i x) = 1 + \lambda_1 x + \cdots + \lambda_t x^t. \tag{34}$$

Then we have

$$\Lambda(X_i^{-1}) = 1 + \lambda_1 X_i^{-1} + \cdots + \lambda_t X_i^{-t} = 0 \qquad (i = 1, \cdots, t) \tag{35}$$

Multiply both sides of (35) by $Y_i X_i^{j+t}$, we get

$$Y_i X_i^{j+t}\Lambda(X_i^{-1}) = Y_i X_i^{j+t} + \lambda_1 Y_i X_i^{j+t-1} + \cdots + \lambda_t Y_i X_i^{j} = 0 \tag{36}$$

For $i = 1, \cdots, t$, add equations (36) together, we obtain

$$\sum_{i=1}^{t}(Y_i X_i^{j+t}) + \lambda_1 \sum_{i=1}^{t}(Y_i X_i^{j+t-1}) + \cdots + \lambda_t \sum_{i=1}^{t}(Y_i X_i^{j}) = 0 \tag{37}$$

Combing (33) and (37), we obtain

$$S_j\lambda_t + S_{j+1}\lambda_{t-1} + \cdots + S_{j+t-1}\lambda_1 + S_{j+t} = 0 \qquad (j = 1, \cdots, t) \tag{38}$$

which yields the following linear equation system:

$$\begin{pmatrix} S_1 & S_2 & \cdots & S_t \\ S_2 & S_3 & \cdots & S_{t+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_t & S_{t+1} & \cdots & S_{2t-1} \end{pmatrix} \begin{pmatrix} \lambda_t \\ \lambda_{t-1} \\ \vdots \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t} \end{pmatrix} \tag{39}$$

Since the number of errors is unknown, Peterson-Gorenstein-Zierler tries various t from the maximum $\frac{n-k}{2}$ to solve the equation system (39). After the error locator polynomial $\Lambda(x)$ is identified, one can use exhaustive search algorithm, Chien's search algorithm, BTA algorithms, or other root-finding algorithms to find the roots of $\Lambda(x)$. After the error locations are identified, one can use Forney's algorithm to determined the error values. With $e(x)$ in hand, one subtracts $e(x)$ from $r(x)$ to obtain $c(x)$.
Computational complexity: Assume that $(\alpha^j)^i$ for $i = 0, \cdots, n-1$ and $j = 0, \cdots, n-k$ have been pre-computed in a table. Then it takes $2(n-1)(n-k)$ field operations to compute the values of $S_1, \cdots, S_{n-k}$. After S_i are computed, it takes $O(t^3)$ field operations (for Gaussian eliminations) to solve the equation (39) for each chosen t.

10.11.2 Forney's algorithm

For Forney's algorithm, we define the error evaluator polynomial (note that $n - k \geq 2t$)

$$\Omega(x) = \Lambda(x) + \sum_{i=1}^{t} X_i Y_i x \prod_{j=1, j\neq i}^{t} (1 - X_j x) \tag{40}$$

and the syndrome polynomial

$$S(x) = S_1 x + S_2 x^2 + \cdots S_{2t} x^{2t}.$$

Note that

$$\begin{aligned} S(x)\Lambda(x) &= \left(\sum_{l=1}^{2t} \sum_{i=1}^{t} Y_i X_i^l x^l\right) \prod_{j=1}^{t} (1 - X_j x) \qquad \bmod x^{2t+1} \\ &= \sum_{i=1}^{t} Y_i \sum_{l=1}^{2t} (X_i x)^l \prod_{j=1}^{t} (1 - X_j x) \qquad \bmod x^{2t+1} \\ &= \sum_{i=1}^{t} Y_i (1 - X_i x) \sum_{l=1}^{2t} (X_l x)^i \prod_{j=1, j\neq i}^{t} (1 - X_j x) \qquad \bmod x^{2t+1} \end{aligned} \tag{41}$$

Using the fact that $(1 - x^{2t+1}) = (1 - x)(1 + x + \cdots + x^{2t})$, we have

$$(1 - X_i x) \sum_{l=1}^{2t} (X_i x)^l = X_i x - (X_i x)^{2t+1} = X_i x \quad \bmod x^{2t+1}.$$

Thus

$$S(x)\Lambda(x) = \sum_{i=1}^{t} Y_i X_i x \prod_{j=1, j\neq i}^{t} (1 - X_j x) \qquad \bmod x^{2t+1}.$$

This gives us the key equation

$$\Omega(x) = (1 + S(x))\Lambda(x) \qquad \bmod x^{2t+1}. \tag{42}$$

Note: In some literature, syndrome polynomial is defined as $S(x) = S_1 + S_2 x + + S_{2t}x^{2t-1}$. In this case, the key equation becomes

$$\Omega(x) = S(x)\Lambda(x) \qquad \mod x^{2t}. \tag{43}$$

Let $\Lambda'(x) = -\sum_{i=1}^{t} X_i \prod_{j\neq i}(1 - X_j x) = \sum_{i=1}^{t} i\lambda_i x^{i-1}$. Then we have $\Lambda'(X_l^{-1}) = -X_l \prod_{j\neq l}(1 - X_j X_l^{-1})$. By substituting X_l^{-1} into $\Omega(x)$, we get

$$\Omega(X_l^{-1}) = \sum_{i=1}^{t} X_i Y_i X_l^{-1} \prod_{j=1, j\neq i}^{t} (1 - X_j X_l^{-1}) = Y_l \prod_{j=1, j\neq l}^{t} (1 - X_j X_l^{-1}) = -Y_l X_l^{-1} \Lambda'(X_l^{-1})$$

This shows that

$$e_{p_l} = Y_l = -\frac{X_l \cdot \Omega(X_l^{-1})}{\Lambda'(X_l^{-1})}.$$

Computational complexity: Assume that $(\alpha^j)^i$ for $i = 0, \cdots, n-1$ and $j = 0, \cdots, n-k$ have been pre-computed in a table. Furthermore, assume that both $\Lambda(x)$ and $S(x)$ have been calculated already. Then it takes $O(n^2)$ field operations to calculate $\Omega(x)$. After both $\Omega(x)$ and $\Lambda(x)$ are calculated, it takes $O(n)$ field operations to calculate each e_{p_l}. As a summary, assuming that $S(x)$ and $\Lambda(x)$ are known, it takes $O(n^2)$ field operations to calculate all error values.

10.11.3 Berlekamp-Massey decoder

In this section we discuss Berlekamp-Massey decoder [21] which has computational complexity $O(n^2)$. Note that there exists an implementation using Fast Fourier Transform that runs in time $O(n \log n)$. Berlekamp-Massey algorithm is an alternative approach to find the minimal degree t and the error locator polynomial $\Lambda(x) = 1 + \lambda_1 x \cdots + \lambda_t x^t$ such that all equations in (38) hold. The equations in (38) define a general linear feedback shift register (LFSR) with initial state $S_1, \cdots, S_t$. Thus the problem of finding the error locator polynomial $\Lambda(x)$ is equivalent to calculating the linear complexity (alternatively, the connection polynomial of the minimal length LFSR) of the sequence $S_1, \cdots, S_{2t}$. The Berlekamp-Massey algorithm constructs an LFSR that produces the entire sequence $S_1, \cdots, S_{2t}$ by successively modifying an existing LFSR to produce increasingly longer sequences. The algorithm starts with an LFSR that produces S_1 and then checks whether this LFSR can produce $S_1 S_2$. If the answer is yes, then no modification is necessary. Otherwise, the algorithm revises the LFSR in such a way that it can produce $S_1 S_2$. The algorithm runs in $2t$ iterations where the ith iteration computes the linear complexity and connection polynomial for the sequence $S_1, \cdots, S_i$. The following is the original LFSR Synthesis Algorithm from Massey [21].

1. $\Lambda(x) = 1, B(x) = 1, u = 1, L = 0, b = 1, i = 0$.
2. If $i = 2t$, stop. Otherwise, compute
$$d = S_i + \sum_{j=1}^{L} \lambda_j S_{i-j} \tag{44}$$
3. If $d = 0$, then $u = u + 1$, and go to (6).
4. If $d \neq 0$ and $i < 2L$, then
$$\begin{aligned} \Lambda(x) &= \Lambda(x) - db^{-1}x^u B(x) \\ u &= u + 1 \end{aligned}$$
and go to (6).
5. If $d \neq 0$ and $i \geq 2L$, then
$$\begin{aligned} T(x) &= \Lambda(x) \\ \Lambda(x) &= \Lambda(x) - db^{-1}x^u B(x) \\ L &= i + 1 - L \\ B(x) &= T(x) \\ b &= d \\ u &= 1 \end{aligned} \tag{45}$$
6. $i = i + 1$ and go to step (2).

Discussion: For the sequence $S_1, \cdots, S_i$, we use $L_i = L(S_1, \cdots, S_i)$ to denote its linear complexity. We use $\Lambda^{(i)}(x) = 1 + \lambda_1^{(i)} x + \lambda_2^{(i)} x^2 + \cdots + \lambda_{L_i}^{(i)} x^{L_i}$ to denote the connection polynomial for the sequence $S_1 \cdots S_i$ that we have obtained at iteration i. At iteration i, the constructed LFSR can produce the sequence $S_1 S_2 \cdots S_i$. That is,
$$S_j = -\sum_{l=1}^{L_i} \lambda_j^{(i)} S_{j-l}, \qquad j = L_i + 1, \cdots, i$$
Let i_0 denote the last position where the linear complexity changes during the iteration and let d_i denote the discrepancy obtained at iteration i using the equation (44). That is,
$$d_i = S_i + \sum_{j=1}^{L_{i-1}} \lambda_j^{(i-1)} S_{i-j}.$$
We show that $\Lambda^{(i)}(x) = \Lambda^{(i-1)}(x) - d_i b^{-1} x^u B(x)$ is the connection polynomial for the sequence $S_1, \cdots, S_i$. The case for $d_i = 0$ is trivial. Assume that $d_i \neq 0$. Then $B(x) = \Lambda^{(i_0)}(x)$ and $b = d_{i_0+1}$. By the construction in Step 4 and Step 5, we have $\Lambda^{(i)}(x) = \Lambda^{(i-1)}(x) - d_i d_{i_0+1}^{-1} x^u \Lambda^{(i_0)}(x)$. For $v = L_i, L_i + 1, \cdots, i - 1$, we have
$$\begin{aligned} S_v + \textstyle\sum_{j=1}^{L_i} \lambda_j^{(i)} S_{v-j} &= S_v + \textstyle\sum_{j=1}^{L_{i-1}} \lambda_j^{(i-1)} S_{v-j} + d_i d_{i_0+1}^{-1} \left(S_{v-i+i_0+1} + \textstyle\sum_{j=1}^{L_{i_0}} \lambda_j^{(i_0)} S_{v-i+i_0+1-j} \right) \\ &= \begin{cases} 0 & L_i \leq u \leq i - 1 \\ d_i - d_i d_{i_0+1}^{-1} d_{i_0+1} & u = i \end{cases} \end{aligned}$$

Computational complexity: As we have mentioned in Section 10.11, it takes $2(n-1)(n-k)$ field operations to calculates the sequence $S_1, \cdots, S_{n-k}$. In the Berlekamp-Massey decoding process, iteration i requires at most $2(i-1)$ field operations to calculate d_i and at most $2(i-1)$ operations to calculate the polynomial $\Lambda^{(i)}(x)$. Thus it takes at most $4t(2t - 1)$ operations to finish the iteration process. In a summary, Berlekamp-Massey decoding process requires at most $2(n - 1)(n - k) + 4t(2t - 1)$ field operations.

10.11.4 Euclidean decoder

Assume that the polynomial $S(x)$ is known already. By the key equation (42), we have

$$\Omega(x) = (1 + S(x))\Lambda(x) \mod x^{2t+1}$$

with $\deg(\Omega(x)) \leq \deg(\Lambda(x)) \leq t$. The generalized Euclidean algorithm could be used to find a sequence of polynomials $R_1(x), \cdots, R_u(x)$, $Q_1(x), \cdots, Q_u(x)$ such that

$$\begin{array}{l} x^{2t+1} - Q_1(x)(1 + S(x)) = R_1(x) \\ 1 + S(x) - Q_2(x)R_1(x) = R_2(x) \\ \cdots \\ R_{u-2}(x) - Q_u(x)R_{u-1}(x) = R_u(x) \end{array}$$

where $\deg(1 + S(x)) > \deg(R_1(x))$, $\deg(R_i(x)) > \deg(R_{i+1}(x))$ $(i = 1, \cdots, u - 1)$, $\deg(R_{u-1}(x)) \geq t$, and $\deg(R_u(x)) < t$. By substituting first $u - 1$ identities into the last identity, we obtain the key equation

$$\Lambda(x)(1 + S(x)) - \Gamma(x)x^{2t+1} = \Omega(x)$$

where $R_u(x) = \Omega(x)$.

In case that the syndrome polynomial is defined as $S(x) = S_1 + S_2x + + S_{2t}x^{2t-1}$, the Euclidean decoder will calculate the key equation

$$\Lambda(x)S(x) - \Gamma(x)x^{2t} = \Omega(x)$$

Computational complexity: As we mentioned in the previous sections, it takes $2(n - 1)(n - k)$ field operations to calculate the polynomial $S(x)$. After $S(x)$ is obtained, the above process stops in u steps where $u \leq t + 1$. For each identity, it requires at most $O(t)$ steps to obtain the pair of polynomials (R_i, Q_i). Thus the total steps required by the Euclidean decoder is bounded by $O(t^2)$.

10.11.5 Berlekamp-Welch decoder

In previous sections, we discussed syndrome-based decoding algorithms for Reed-Solomon codes. In this and next sections we will discuss syndrome-less decoding algorithms that do not compute syndromes and do not use the Chien search and Forneys formula. We first introduce Berlekamp-Welch decoding algorithm which has computational complexity $O(n^3)$. Berlekamp-Welch decoding algorithm first appeared in the US Patent 4,633,470 (1983). The algorithm is based on the classical definition of Reed-Solomon codes and can be easily adapted to the BCH definition of Reed-Solomon codes. The decoding problem for the classical Reed-Solomon codes is described as follows: We have a polynomial $m(x)$ of degree at most $k - 1$ and we received a polynomial $c(x)$ which is given by its evaluations $(r_0, \cdots, r_{n-1})$ on n distinct field elements. We know that $m(x) = r(x)$ for at least $n - t$ points. We want to recover $m(x)$ from $r(x)$ efficiently.

Berlekamp-Welch decoding algorithm is based on the fundamental vanishing lemma for polynomials: If $m(x)$ is a polynomial of degree at most d and $m(x)$ vanishes at $d + 1$ distinct points, then m is the zero polynomial. Let the graph of $r(x)$ be the set of q points:

$$\{(x, y) \in GF(q) : y = r(x)\}.$$

Let $R(x, y) = Q(x) - E(x)y$ be a non-zero lowest-degree polynomial that vanishes on the graph of $r(x)$. That is, $Q(x) - E(x)r(x)$ is the zero polynomial. In the following, we first show that $E(x)$ has degree at most t and $Q(x)$ has degree at most $k + t - 1$.

Let $x_1, \cdots, x_{t'}$ be the list of all positions that $r(x_i) \neq m(x_i)$ for $i = 1, \cdots, t'$ where $t' \leq t$. Let

$$E_0(x) = (x - x_1)(x - x_2) \cdots (x - x_{t'}) \text{ and } Q_0(x) = m(x)E_0(x).$$

By definition, we have $\deg(E_0(x)) = t' \leq t$ and $\deg(Q_0(x)) = t' + k - 1 \leq t + k - 1$. Next we show that $Q_0(x) - E_0(x)r(x)$ is the zero polynomial. For each $x \in GF(q)$, we distinguish two cases. For the first case, assume that $m(x) = r(x)$. Then $Q_0(x) = m(x)E_0(x) = r(x)E_0(x)$. For the second case, assume that $m(x) \neq r(x)$. Then $E_0(x) = 0$. Thus we have $Q_0(x) = m(x)E_0(x) = 0 = r(x)E_0(x)$. This shows that there is a polynomial $E(x)$ of degree at most t and a polynomial $Q(x)$ of degree at most $k + t - 1$ such that $R(x, y) = Q(x) - E(x)y$ vanishes on the graph of $r(x)$.

The arguments in the preceding paragraph show that, for the minimal degree polynomial $R(x, y) = Q(x) - E(x)y$, both $Q(x)$ and $m(x)E(x)$ are polynomials of degree at most $k+t-1$. Thus $Q(x) - m(x)E(x)$ has degree at most $k+t-1$. For each x such that $m(x) - r(x) = 0$, we have $Q(x) - m(x)E(x) = 0$. Since $m(x) - r(x)$ vanishes on at least $n - t$ positions and $n - t > k + t - 1$, the polynomial $R(x, m(x)) = Q(x) - m(x)E(x)$ must be the zero polynomial.

The equation $Q(x) - E(x)r(x) = 0$ is called the key equation for the decoding algorithm. The arguments in the preceding paragraphs show that for any solutions $Q(x)$ of degree at most $k + t - 1$ and $E(x)$ of degree at most t, $Q(x) - m(x)E(x)$ is the zero polynomial. That is, $m(x) = \frac{Q(x)}{E(x)}$. This implies that, after solving the key equation, we can calculate the message polynomial $m(x)$. Let $(m(a_0), \cdots, m(a_{n-1}))$ be the transmitted code and $(r_0, \cdots, r_{n-1})$ be the received vector. Define two polynomials with unknown coefficients:

$$\begin{aligned} Q(x) &= u_0 + u_1 x + \cdots + u_{k+t-1} x^{k+t-1} \\ E(x) &= v_0 + v_1 x + \cdots + v_t x^t \end{aligned}$$

Using the identities

$$Q(a_i) = r_i \cdot E(a_i) \qquad (i = 0, \cdots, n-1)$$

to build a linear equation system of n equations in $n + 1$ unknowns $u_0, \cdots, u_{k+t-1}, v_0, \cdots, v_t$. Find a non-zero solution of this equation system and obtain the polynomial $Q(x)$ and $E(x)$. Then $m(x) = \frac{Q(x)}{E(x)}$.

Computational complexity: The Berlekamp-Welch decoding process solves an equation system of n equations in $n + 1$ unknowns. Thus the computational complexity is $O(n^3)$.

10.11.6 Experimental results

Table 14 gives experimental results on decoding Reed-Solomon codes for various parameters corresponding RLCE schemes. The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor. The reported time is the required milliseconds for decoding a received codeword over $GF(2^m)$ (an average of 10,000 trials).

Table 14: Milliseconds for decoding Reed-Solomon codes over $GF(2^m)$

(n, k, t, m)	BM-decoder	Euclidean decoder
$(630, 470, 80, 10)$	1.9261904	2.6511796
$(1000, 764, 118, 10)$	3.1226213	4.0247824
$(1360, 800, 280, 11)$	12.4488992	16.3140049

10.12 Efficient random bits generation

For RLCE scheme key set up, RLCE scheme encryption, and RLCE scheme decryption, deterministic random bits need to be generated using entropy string. We compared hash function based NIST DRBG schemes and AES counter mode based NIST DRBG schemes. Table 15 gives experimental results on generating a 10000-bytes sequence (an average of 10,000 trials). The implementation was run on a MacBook Pro with MacOS Sierra version 10.12.5 with 2.9GHz Intel Core i7 Processor.

Table 15: Milliseconds for generating a 10000-bytes sequence

SHA-256	.4004342
SHA-512	.2394226
AES-128	.3526720
AES-256	.3840904
AES-512	.4645755

10.13 Conclusion

This section compares different algorithms for implementing the RLCE encryption scheme. The experiments show that for all of the RLCE encryption scheme parameters (corresponding to AES-128, AES-192, and AES-256), Chien's search algorithm should be used in the root-finding process of the error locator polynomials. For polynomial multiplications, one should use optimized classical polynomial multiplication algorithm for polynomials of degree 115 and less. For polynomials of degree 115 and above, one should use Karatsuba algorithm. For matrix multiplications, one should use optimized classical matrix multiplication algorithm for matrices of dimension 750 or less. For matrices of dimension 750 or above, one should use Strassen's algorithm. For the underlying Reed-Solomon decoding process, Berlekamp-Massey outperforms Euclidean decoding process. For large amount of pseudo-random bit generations, SHA-512 based DRBG should be used for RLCE schemes.

11 Appendix D: Optimized implementation (Informative)

This section discusses the optimized implementation submitted to NIST. The following global variables in the file config.h are used to control various aspects of the optimized implementation.

```
config.h

#define DECODINGMETHOD  1 /* 0: include S; 1: W^{-1}; 2: no help matrix    */
#define GFMULTAB 1        /* 0: No Mul-TABLE; 1: GF multiplication-table   */
#define DRBG 0            /* 0: SHA512_DRBG; 1: CTR-AES DRBG; 2:MGF512     */
#define DECODER 0         /* 0: BM-decoder, 1: Euclidean Decoder           */
#define ROOTFINDING 0     /* 0: Chien; 1: Exhaustive; 2: BAT; 3: FFT       */
#define KARATSUBA 1       /* 0: standard poly-mul; 1: karatsuba; 2: FFT    */
#define WINOGRADVEC 0     /* 0: standard; 1: winograd vec*matrix multi     */
#define MATRIXMUL 0       /* 0: standard mat*mat; 1: strassen; 2: Winograd */
#define MATINV 0          /* 0: standard mat-inv; 1: strassen              */
```

```
#define STRASSENCONST 750 /* mat-dim below this use standard multi.        */
#define STRAINVCONST 500  /* mat-dim below this use standard inverse       */
```

These variables have the following semantics:

- `GFMULTAB`= 1 means that we use finite field multiplication tables and `GFMULTAB`= 0 means that no multiplication table is used.
- `DRBG`= 0 means that the `SHA512_DRBG` is used; `DRBG`= 1 means that the `CTR_AES_DRBG` is used; and `DRBG`= 1 means that the `MGF512` is used as the DRBG.
- `DECODER`= 0 means that Berlekamp-Massey decoder is used for the Reed-Solomon decoding process and `DECODER`= 1 means that Euclidean decoder is used for the Reed-Solomon decoding process.
- `ROOTFINDING`= 0 means that Chien's algorithm is used for finding roots of a polynomial; `ROOTFINDING` = 1 means that exhaustive search is used for finding roots of a polynomial; `ROOTFINDING`= 2 means that Berlekamp Trace Algorithm is used for finding roots of a polynomial; and `ROOTFINDING`= 3 means that FFT is used for finding roots of a polynomial.
- `KARATSUBA`= 0 means that the standard polynomial multiplication algorithm is used; `KARATSUBA`= 1 means that Karatsuba polynomial multiplication algorithm is used; and `KARATSUBA`= 2 means that FFT polynomial multiplication algorithm is used.
- `WINOGRADVEC`= 0 means that standard vector-matrix multiplication algorithm is used and `WINOGRADVEC` = 1 means that Winograd vector-matrix multiplication algorithm is used.
- `MATRIXMUL`= 0 means that the standard matrix-matrix multiplication algorithm is used and `MATRIXMUL`= 1 means that Strassen matrix-matrix multiplication algorithm is used.
- `MATINV`= 0 means that the standard matrix-inverse algorithm is used and `MATINV`= 1 means that the Strassen matrix-inverse algorithm is used.
- `STRASSENCONST`= 750 means that the standard matrix-matrix multiplication algorithm is used if the matrix dimension is smaller than 750.
- `STRAINVCONST`= 500 means that the standard matrix-inverse algorithm is used if the matrix dimension is smaller than 500.

11.1 Reducing private key size and reducing key generation time: DECODINGMETHOD

The default value for the variable DECODINGMETHOD is 1. This means that the RLCE-KEM private key contains a $k \times (u_0 + 1)$ matrix X. Since the matrix X (or a variant of X) could be computed on the fly, one does not need to include X in the private key. This is achieved by setting DECODINGMETHOD to 2. The reduced private key sizes are shown in Table 16. In this case, the variable `CRYPTO_SECRETKEYBYTES` in api.h should be set to the corresponding private key size as shown in Table 16.

On the other hand, one may reduce the key generation time by including certain columns of S^{-1} instead of W^{-1} in the X component of the private key. In Section 2, we mentioned that it suffices to recover the remaining u message symbols m_i with $i \in \bar{I}_R$. That is, one can recover $\mathbf{m}' = \mathbf{m}S$ from $\mathbf{m}SG_s$ first and then multiply $\mathbf{m}'$ with the corresponding u columns within the matrix S^{-1} to get m_i for $i \in \bar{I}_R$. That is, one only needs to include these u columns of the matrix S^{-1} within X. Since S^{-1} is free to obtain, this decoding approach improves the efficiency of RLCE key set-up process. However, it is generally slow to recover $\mathbf{m}' = \mathbf{m}S$ from $\mathbf{m}SG_s$. Thus the decryption process is less efficient. This decoding approach is achieved by setting DECODINGMETHOD to 0.

Table 16: Reduced private key sizes

ID	RLCE sk size	reduced RLCE sk size
0	310116	192029
1	747393	457073
2	1773271	1241971

11.2 Scheme ID and message padding approaches

In the file `api.h`, two global variables `CRYPTO_SCHEME` and `CRYPTO_PADDING` are declared. In the proposed RLCE-KEM as discussed in Section 1, `CRYPTO_SCHEME` takes the values from $0, \cdots, 6$ as shown in Table 2 and `CRYPTO_PADDING` takes the value 1. In the optimized implementation, `CRYPTO_SCHEME` may also take the values from $7, \cdots, 14$ for list-decoding based RLCE-KEM schemes as defined in Wang [33]. Furthermore, `CRYPTO_SCHEME` may also take the values $0, 2, 3$ which are based on basic message encoding and RLCEspad padding scheme. The details for these message encoding and message padding schemes could be found in Wang [33].

References

[1] M. Baldi, M. Bodrato, and F. Chiaraluce. A new analysis of the mceliece cryptosystem based on QC-LDPC codes. In *Security and Cryptography for Networks*, pages 246–262. Springer, 2008.

[2] G.V. Bard. Accelerating cryptanalysis with the method of four russians. *IACR Cryptology EPrint Archive*, 2006:251, 2006.

[3] 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.

[4] E.R. Berlekamp. *Algebraic coding theory*. McGraw-Hill, 1968.

[5] D.J. Bernstein. Grover vs. McEliece. In *Proc. Int. Workshop PQC*, pages 73–80. Springer, 2010.

[6] D.J. Bernstein, T. Lange, and C. Peters. Attacking and defending the McEliece cryptosystem. In *Proc. Int. Workshop PQC*, pages 31–46. Springer, 2008.

[7] T.A Berson. Failure of the mceliece public-key cryptosystem under message-resend and related-message attack. In *Proc Crypto*, pages 213–220. Springer, 1997.

[8] J.R. Bunch and J.E. Hopcroft. Triangular factorization and inversion by fast matrix multiplication. *Mathematics of Computation*, 28(125):231–236, 1974.

[9] A. Canteaut and N. Sendrier. Cryptanalysis of the original McEliece cryptosystem. In *Proc. Asiacrypt*, pages 187–199. Springer, 1998.

[10] D.G. Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2):285–300, 1989.

[11] A. Couvreur, P. Gaborit, V. Gauthier-Umaña, A. Otmani, and J.-P. Tillich. Distinguisher-based attacks on public-key cryptosystems using Reed–Solomon codes. *Designs, Codes and Cryptography*, pages 1–26, 2013.

[12] J.-C. Faugere, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In *Eurocrypt 2010*, pages 279–298. Springer, 2010.

[13] S. Gao and T. Mateer. Additive fast fourier transforms over finite fields. *IEEE Tran. Information Theory*, 56(12):6265–6272, 2010.

[14] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt. Applying Grover's algorithm to AES: quantum resource estimates. In *Proc. Int. Workshop PQC*, pages 29–43. Springer, 2016.

[15] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. *IEEE Tran. Information Theory*, 45:1757–1767, 1999.

[16] C. Hall, I. Goldberg, and B. Schneier. Reaction attacks against several public-key cryptosystem. In *Proc. ICICS*, pages 2–12. Springer, 1999.

[17] J. Justesen and T. Hoholdt. Bounds on list decoding of mds codes. *Information Theory, IEEE Tran.*, 47(4):1604–1609, 2001.

[18] G. Kachigar and J.-P. Tillich. Quantum information set decoding algorithms. Cryptology ePrint Archive, Report 2017/213, 2017. http://eprint.iacr.org/2017/213.

[19] S. Kepley and R. Steinwandt. Quantum circuits for F_{2^m}-multiplication with subquadratic gate count. *Quantum Information Processing*, 14(7):2373–2386, 2015.

[20] P. Loidreau and N. Sendrier. Some weak keys in mceliece public-key cryptosystem. In *Proc. IEEE ISIT*, pages 382–382, 1998.

[21] J. Massey. Shift-register synthesis and bch decoding. *IEEE Trans. Information Theory*, 15(1):122–127, 1969.

[22] R.J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN progress report*, 42(44):114–116, 1978.

[23] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. Barreto. MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In *Proc. IEEE ISIT 2013*, pages 2069–2073, 2013.

[24] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Prob. Control and Information Theory*, 15(2):159–166, 1986.

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

[26] 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.

[27] C. Peters. Information-set decoding for linear codes over F_q. In *Proc. Int. Workshop PQC*, pages 81–94. Springer, 2010.

[28] V. Shoup. OAEP reconsidered. In *CRYPTO 2001*, pages 239–259. Springer, 2001.

[29] V. M Sidelnikov and S. Shestakov. On insecurity of cryptosystems based on generalized Reed-Solomon codes. *Discrete Mathematics and Applications*, 2(4):439–444, 1992.

[30] J. Stern. A method for finding codewords of small weight. In *Coding theory and applications*, pages 106–113. Springer, 1989.

[31] J. Von zur Gathen and J. Gerhard. Arithmetic and factorization of polynomial over f 2. In *Proc. ISSAC*, pages 1–9. ACM, 1996.

[32] Y. Wang. Quantum resistant random linear code based public key encryption scheme RLCE. In *Proc. IEEE ISIT*, pages 2519–2523, July 2016.

[33] Y. Wang. Revised quantum resistant public key encryption scheme RLCE and IND-CCA2 security for McEliece schemes. In *IACR ePrint https://eprint.iacr.org/2017/206.pdf*, July 2017.

[34] C. Wieschebrink. Two NP-complete problems in coding theory with an application in code based cryptography. In *Proc. IEEE ISIT*, pages 1733–1737. IEEE Press, 2006.