Decoding Generalized Reed-Solomon Codes and Its Application to RLCE Encryption Schemes

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

February 24, 2017

Abstract

This paper presents a survey on generalized Reed-Solomon codes and various decoding algorithms: Berlekamp-Massey decoding algorithms; Berlekamp-Welch decoding algorithms; Euclidean decoding algorithms; discrete Fourier decoding algorithms, Chien's search algorithm, and Forney's algorithm.

Key words: Reed-Solomon code; generalized Reed-Solomon code.

1 Finite fields

1.1 Representation of elements in finite fields

In this section, we present a Layman's guide to several representations of elements in a finite field $GF(q)$. We assume that the reader is familiar with the finite field $GF(p) = Z_p$ for a prime number p and we concentrate on the construction of finite fields $GF(p^m)$.

Polynomials: Let $\pi(x)$ 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 $\alpha, \alpha^p, \cdots, \alpha^{p^{m-1}} \in GF(p^m)$.

Generator and primitive polynomial: 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 $GF(p^m) = \{0\} \cup \{\alpha^i : i = 0, \cdots, p^m - 1\}$. 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 generator polynomial is called a primitive element.

Matrix approach: The companion matrix of a polynomial $\pi(x) = a_0 + a_1 x + \cdots + a_{m-1}x^{m-1} + x^m$ is defined to be the $m \times m$ matrix

$$M = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{m-1} \end{pmatrix}$$

The set of matrices $0, M, \cdots, M^{p^m-1}$ with matrix addition and multiplication over $GF(p)$ forms the finite field $GF(p^m)$.

Splitting field: Let $\pi(x) \in GF(p)[x]$ be a degree m irreducible polynomial. Then $GF(p^m)$ can be considered as a splitting field of $\pi(x)$ over $GF(p)$. That is, assume that $\pi(x) = (x-\alpha_1)\cdots(x-\alpha_m)$ in $GF(p^m)$. Then $GF(p^m)$ is obtained by adjoining these algebraic elements $\alpha_1, \cdots, \alpha_m$ to $GF(p)$.

1.2 Finite field arithmetics

Let α be a primitive element in $GF(q)$. Then for each non-zero $x \in GF(q)$, there exists a $0 \le y \le q-2$ such that $x = \alpha^y$ where y is called the discrete logarithm of x. When field elements are represented using their discrete logarithms, multiplication and division are efficient since they are reduced to integer addition and subtraction modulo $q-1$. For additions, one may use Zech's logarithm which is defined as

$$Z(y) : y \mapsto \log_\alpha(1+\alpha^y). \tag{1}$$

That is, for a field element α^y, we have $\alpha^{Z(y)} = 1 + \alpha^y$. If one stores Zech's logarithm in a table as pairs $(y, Z(y))$, then the addition could be calculated as

$$\alpha^{y_1} + \alpha^{y_2} = \alpha^{y_1}(1+\alpha^{y_2-y_1}) = \alpha^{y_1}\alpha^{Z(y_2-y_1)} = \alpha^{y_1+Z(y_2-y_1)}.$$

For the finite field $GF(2^m)$, the addition is the efficient XOR operation. Thus it is better to store two tables to speed up the multiplication: discrete logarithm table and exponentiation tables. For the discrete logarithm table, one obtains y on input x such that $x = \alpha^y$. For the exponentiation table, one obtains y on input x such that $y = \alpha^x$. In order to multiply two field elements x_1, x_2, one first gets their discrete logarithms y_1, y_2 respectively. Then one calculates $y = y_1 + y_2$. Next one looks up the exponentiation table to find out the value of α^y. Note that we have $x_1x_2 = \alpha^{y_1}\alpha^{y_2} = \alpha^{y_1+y_2}$.

2 Reed-Solomon codes

2.1 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

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

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.

2.2 The BCH approach

We first give a definition for the t-error-correcting BCH codes of distance δ. Let $1 \leq \delta < n = q - 1$ 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 $m(x) = m_0 + m_1 x + \cdots + m_w x^w$ 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 $1 \leq k < n = q - 1$ and let $g(x) = (x - \alpha^b)(x - \alpha^{b+1}) \cdots (x - \alpha^{b+n-k-1}) = g_0 + g_1 x + \cdots + g_{n-k} x^{n-k}$ be a polynomial over $GF(q)$. Then a message polynomial $m(x) = m_0 + m_1 x + \cdots + m_{k-1} x^{k-1}$ 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 $g(x)$.

2.3 The equivalence

The equivalence of the two definitions for Reed-Solomon code could be established using discrete Fourier transform (Mattson-Solomon polynomial). In the following, we assume that $n = q - 1$ and α is a primitive element of $GF(q)$. For a polynomial $f \in GF(q)[x]$ with $\deg(f) \leq n$, the Mattson-Solomon polynomial of f is defined as

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

It is straightforward that Φ is a linear map over $GF(q)[x]$. By the fact that

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

we have $x(x^n - 1) = (x - 1)(x + x^2 + \cdots + x^n)$. Thus $\sum_{i=1}^{n} a^i = 0$ for all $a \in GF(q)$ with $a \neq 1$. Let $f(x) = \sum_{j=0}^{n-1} a_j x^j$. Then

$$\begin{aligned} F(\alpha^j) &= \textstyle\sum_{i=1}^{n} f(\alpha^i) \alpha^{j(n-i)} \\ &= \textstyle\sum_{i=1}^{n} \sum_{u=0}^{n-1} a_u \alpha^{ui} \alpha^{j(n-i)} \\ &= \textstyle\sum_{u=0}^{n-1} a_u \sum_{i=1}^{n} \alpha^{(u-j)i} \\ &= n a_j \end{aligned} \tag{2}$$

If $f(x)$ is a multiple of the generating polynomial $g(x) = \prod_{j=1}^{n-k}\left(x - \alpha^j\right)$, then $f(\alpha^j) = 0$ for $1 \le j \le n-k$. Thus $F(x) = \Phi(f)$ has degree at most $k-1$. The identity (2) gives us an inverse of Φ as follows:

$$\Phi^{-1}(F)(x) = n^{-1}\Phi(F)(x^{-1}) \mod x^n - 1.$$

Φ^{-1} maps polynomial of degree at most $k-1$ to multiples of the generating polynomial $g(x)$.

For each Reed-Solomon codeword $f(x)$ in the BCH approach, it is a multiple of the generating polynomial. The above arguments show that $F(x)$ has degree at most $k-1$ and $(F(\alpha^0)/n, \cdots, F(\alpha^{n-1})/n) = \frac{1}{n}f(x)$. 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 $F(x) = \Phi(f)$ of degree at most $k-1$ on $\alpha^0, \cdots, \alpha^{n-1}$. By the identity (2), $(a_0, \cdots, a_{n-1})$ is the coefficients of $\Phi^{-1}(F)$ which is a multiple of the generating polynomial $g(x)$.

2.4 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

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

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 $\mathbf{c} \cdot \mathrm{diag}\left(v_0^{-1}, \cdots, v_{n-1}^{-1}\right)$ 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.

3 Decoding Reed-Solomon code

3.1 Peterson-Gorenstein-Zierler decoder

This sections describes Peterson-Gorenstein-Zierler decoder which has computational complexity $O(n^3)$. We assume that the Reed-Solomon encoding process uses the BCH's systematic encoding process. We first calculate syndromes. Let

$$r(x) = c(x) + e(x)$$

where

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

has $t \le \frac{n-k}{2}$ non-zero coefficients. The numbers $0 \le p_1, \cdots, p_t \le n-1$ are error positions and e_{p_i} are error magnitudes (values). For convenience, we will use $X_i = \alpha^{p_i}$ to denote error locations and $Y_i = e_{p_i}$ to denote error magnitudes. The syndromes S_j for $j = 1, \cdots, n-k$ are defined as

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

That is, we have

$$\begin{pmatrix} X_1^1 & X_2^1 & \cdots & X_t^1 \\ X_1^2 & X_2^2 & \cdots & X_t^2 \\ \vdots & \vdots & \ddots & \vdots \\ X_1^{n-k} & X_2^{n-k} & \cdots & X_t^{n-k} \end{pmatrix} \begin{pmatrix} Y_1 \\ Y_2 \\ \vdots \\ Y_t \end{pmatrix} = \begin{pmatrix} S_1 \\ S_2 \\ \vdots \\ S_{n-k} \end{pmatrix} \tag{3}$$

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

$$\Lambda(x) = \prod_{i=1}^{t}(1 - X_i x) = 1 + \lambda_1 x + \cdots + \lambda_t x^t. \tag{4}$$

Then we have

$$\Lambda(X_i^{-1}) = 1 + \lambda_1 X_i^{-1} + \cdots + \lambda_t X_i^{-t} = 0 \qquad (i = 1, \cdots, t) \tag{5}$$

Multiply both sides of (5) by $Y_i X_i^{j+t}$, we get

$$Y_i X_i^{j+t} \Lambda(X_i^{-1}) = Y_i X_i^{j+t} + \lambda_1 Y_i X_i^{j+t-1} + \cdots + \lambda_t Y_i X_i^j = 0 \tag{6}$$

For $i = 1, \cdots, t$, add equations (6) together, we obtain

$$\sum_{i=1}^{t}(Y_i X_i^{j+t}) + \lambda_1 \sum_{i=1}^{t}(Y_i X_i^{j+t-1}) + \cdots + \lambda_t \sum_{i=1}^{t}(Y_i X_i^j) = 0 \tag{7}$$

Combing (3) and (7), we obtain

$$S_j \lambda_t + S_{j+1}\lambda_{t-1} + \cdots + S_{j+t-1}\lambda_1 + S_{j+t} = 0 \qquad (j = 1, \cdots, t) \tag{8}$$

which yields the following linear equation system:

$$\begin{pmatrix} S_1 & S_2 & \cdots & S_t \\ S_2 & S_3 & \cdots & S_{t+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_t & S_{t+1} & \cdots & S_{2t-1} \end{pmatrix} \begin{pmatrix} \lambda_t \\ \lambda_{t-1} \\ \vdots \\ \lambda_1 \end{pmatrix} = \begin{pmatrix} -S_{t+1} \\ -S_{t+2} \\ \vdots \\ -S_{2t} \end{pmatrix} \tag{9}$$

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 (9). After the error locator polynomial $\Lambda(x)$ is identified, one can use exhaustive search algorithm (or Chien's search algorithm for hardware implementations) 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 substracts $e(x)$ 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 $S_1, \cdots, S_{n-k}$. Affter S_i are computed, it takes $O(t^3)$ field operations (for Gaussian eliminations) to solve the equation (9) for each chosen t.

3.1.1 Chien's search algorithm

The problem of find roots for the error locator polynomial in (4) could be solved by an exhaustive search. Alternatively, one may use Chien's search which is based on the following observation.

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

Thus, it is sufficient for us 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 performance for hardware implemenation though it has limited advantage for software implementation with logarithm and exponentation table look-up for field multiplications.

3.1.2 Forney's algorithm

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

$$\Omega(x) = \Lambda(x) + \sum_{i=1}^{t} X_i Y_i x \prod_{j=1, j\neq i}^{t} (1 - X_j x) \tag{10}$$

and the syndrome polynomial

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

Note that

$$\begin{aligned} S(x)\Lambda(x) &= \left(\sum_{l=1}^{2t}\sum_{i=1}^{t} Y_i X_i^l x^l\right)\prod_{j=1}^{t}(1 - X_j x) \quad \mod x^{2t+1} \\ &= \sum_{i=1}^{t} Y_i \sum_{l=1}^{2t}(X_i x)^l \prod_{j=1}^{t}(1 - X_j x) \quad \mod x^{2t+1} \\ &= \sum_{i=1}^{t} Y_i(1 - X_i x)\sum_{l=1}^{2t}(X_l x)^i \prod_{j=1, j\neq i}^{t}(1 - X_j x) \quad \mod x^{2t+1} \end{aligned} \tag{11}$$

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

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

Thus

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

This gives us the key equation

$$\Omega(x) = (1 + S(x))\Lambda(x) \quad \mod x^{2t+1}. \tag{12}$$

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

$$\Omega(x) = S(x)\Lambda(x) \quad \mod x^{2t}. \tag{13}$$

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.

3.2 Berlekamp-Massey decoder

In this section we discuss Berlekamp-Massey decoder [1] 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 (8) hold. The equations in (8) 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 [1].

1. $\Lambda(x) = 1, B(x) = 1, u = 1, L = 0, b = 1, i = 0$.
2. If $i = 2t$, stop. Otherwise, compute
$$d = S_i + \sum_{j=1}^{L} \lambda_j S_{i-j} \tag{14}$$
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
$$\begin{aligned} &T(x) = \Lambda(x) \\ &\Lambda(x) = \Lambda(x) - db^{-1} x^u B(x) \\ &L = i + 1 - L \\ &B(x) = T(x) \\ &b = d \\ &u = 1 \end{aligned} \tag{15}$$
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 (14). 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

$$\begin{aligned} S_v + \sum_{j=1}^{L_i} \lambda_j^{(i)} S_{v-j} &= S_v + \sum_{j=1}^{L_{i-1}} \lambda_j^{(i-1)} S_{v-j} + d_i d_{i_0+1}^{-1} \left(S_{v-i+i_0+1} + \sum_{j=1}^{L_{i_0}} \lambda_j^{(i_0)} S_{v-i+i_0+1-j} \right) \\ &= \begin{cases} 0 & L_i \leq u \leq i-1 \\ d_i - d_i d_{i_0+1}^{-1} d_{i_0+1} & u = i \end{cases} \end{aligned}$$

Computational complexity: As we have mentioned in Section 3, 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.

3.3 Berlekamp-Welch decoder

This section discusses Berlekamp-Welch decoding algorithm which has computational complexity $O(n^3)$. It is noted that 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 shows 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 $O(n^3)$.

3.4 Euclidean decoder

Assume that the polynomial $S(x)$ is known already. By the key equation (12), 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

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

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 equaton

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

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

In case that the syndrome polynomial is defined as $S(x) = S_1 + S_2 x + + S_{2t} x^{2t-1}$, the Euclidean decoder will calculte 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 $O(t^2)$.

3.5 Discrete Fourier transform decoder

The discrete Fourier transform maps a polynomial $v(x) = v_0 + v_1 x + \cdots + v_{n-1} x^{n-1}$ to its values

$$\mathrm{DFT}(v_0, v_1, \cdots, v_{n-1}) = (v(a_1), \cdots, v(a_n))$$

where $a_1, \cdots, a_n$ are fixed distinct points from $GF(q)$. DFT is a bijection on $GF(q)^n$ with the interpolation as the inverse transform, which is often called the inverse discrete Fourier transform DFT^{-1}.

Let $c(x)$ be the code word and $r(x) = c(x) + e(x)$ be the received word. The discrete Fourier transforms of $c(x)$, $e(x)$, and $r(x)$ could be defined as $C(x)$, $E(x)$, and $R(x)$ respectively. By the fact that $r(x) = c(x) + e(x)$, we have $R(x) = C(x) + E(x)$. Note that

$$R_j = C_j + E_j = r(\alpha^j).$$

Thus $R_1, \cdots, R_{2t}$ are the syndromes that could be used to generate the error locator polynomial. The values $E_1, \cdots, E_t$ satisfies the following equations.

$$E_0 = -\frac{1}{\lambda_t}(E_t + \lambda E_{t-1} + \cdots + \lambda_{t-1} E_1)$$
$$E_j = -(\lambda_1 E_{j-1} + \lambda_2 E_{j-1} + \cdots + \lambda_t E_{j-t})$$

After $E(x)$ is obtained, one can obtain $C(x) = R(x) - E(x)$ and use DFT^{-1} to produce $c(x)$.

4 RLCE performance comparison

Wang [2, 3, 4, 5] proposed the RLCE parameters in Table 1. For the parameters in Table 1, the decoding

Table 1: n, k, t, q, key size, where "$360, 200, 80, 101$KB"' represents $n = 360, k = 200, t = 80$.

	RLCE ($w = n$)	RLCE $\left(w = n - \frac{k}{2}\right)$	RLCE ($w = n - k$)	RLCE $\left(w = \frac{n-k}{2}\right)$
60	360,200, 80, 101KB	300,140,80,60KB	255,155,50,30KB	200,120,40,12.89KB
80	560, 380, 90, 267KB	440,280,80,141.5KB	360,200, 80, 101KB	300,140,80,36.91KB
128	1020, 660, 180, 0.98MB	800,440,180,504.88KB	600,464,68,154KB	511,381,65,81.62KB
192	1560, 954, 203, 2.46MB	1300,880,210,1.47MB	1000,790,105,405KB	800,600,100,219.7KB
256	2184, 1260, 462, 4.88MB	1720,1000,360,2.54MB	1300,800,250,1MB	1023,663,180,437KB

performance is listed in Table 2. For each entry, it contains the number of field operations required for Peterson-Gorenstein-Zieler decoder, Berlekamp-Massey decoder, and Berlekamp-Welch decoder. For the performance evaluation of $[n, k, t]$ RLCE scheme, it is done via the extended $[q-1, k+q-1-n, t]$ Reed-Solomon code.

Table 2: Performance comparisons: number of field operations

scheme	**decoder**	**60**	**80**	**128**	**192**	**256**
RLCE ($w = n$)	PGZ	$4 \cdot 10^7$	$7 \cdot 10^7$	10^9	$2 \cdot 10^9$	$5 \cdot 10^{10}$
	BM	$2 \cdot 10^5$	$4 \cdot 10^5$	10^6	$3 \cdot 10^6$	$9 \cdot 10^6$
	BW	10^8	10^9	10^9	$9 \cdot 10^9$	$7 \cdot 10^{10}$
RLCE $\left(w = n - \frac{k}{2}\right)$	PGZ	$4 \cdot 10^7$	$4 \cdot 10^7$	10^9	$2 \cdot 10^9$	$2 \cdot 10^{10}$
	BM	$2 \cdot 10^5$	$2 \cdot 10^5$	10^6	$2 \cdot 10^6$	$4 \cdot 10^6$
	BW	10^8	10^8	10^9	$9 \cdot 10^9$	$9 \cdot 10^9$
RLCE ($w = n - k$)	PGZ	$6 \cdot 10^6$	$4 \cdot 10^7$	$2 \cdot 10^7$	10^8	$4 \cdot 10^9$
	BM	$7 \cdot 10^4$	$2 \cdot 10^5$	$2 \cdot 10^5$	$5 \cdot 10^5$	$3 \cdot 10^6$
	BW	$2 \cdot 10^7$	10^8	10^8	10^9	$9 \cdot 10^9$
RLCE $\left(w = \frac{n-k}{2}\right)$	PGZ	$3 \cdot 10^6$	$4 \cdot 10^7$	$2 \cdot 10^7$	10^8	10^9
	BM	$5 \cdot 10^4$	$2 \cdot 10^5$	$2 \cdot 10^5$	$5 \cdot 10^5$	10^6
	BW	$2 \cdot 10^7$	10^8	10^8	10^9	10^9

References

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

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

[3] Y. Wang. Quantum resistant encryption scheme RLCE and IND-CCA2 security for McEliece schemes. In *Sumbitted*, pages ??–??, July 2017.

[4] Y. Wang. Revised quantum resistant public key encryption scheme RLCE. In *Sumbitted*, pages ??–??, July 2017.

[5] Y. Wang. Revised quantum resistant public key encryption scheme RLCE and IND-CCA2 security for McEliece schemes. In *Sumbitted*, pages ??–??, July 2017.