Chapter 19
Variations on the McEliece Public Key Cryptoystem

19.1 Introduction and Background

In 1978, the distinguished mathematician Robert McEliece invented a public key encryption system [8] based upon encoding the plaintext as codewords of an error-correcting code from the family of Goppa [6] codes. In this system, the ciphertext, sometimes termed the cryptogram, is formed by adding a randomly chosen error pattern containing up to t bits to each codeword. One or more such corrupted codewords make up the ciphertext. On reception, the associated private key is used to invoke an error-correcting decoder based upon the underlying Goppa code to correct the errored bits in each codeword, prior to retrieval of the plaintext from all of the information bits present in the decoded codewords.

Since the original invention there have been a number of proposed improvements. For example, in US Patent 5054066, Riek and McFarland improved the security of the system by complementing the error patterns so as to increase the number of errors contained in the cryptogram [14] and cited other variations of the original system.

This chapter is concerned with a detailed description of the original system plus some refinements which enhance the bandwidth efficiency and security of the original arrangement. The security strength of the system is discussed and analysed.

19.1.1 Outline of Different Variations of the Encryption System

In the originally proposed system [8] a codeword is generated from plaintext message bits by using a permuted, scrambled generator matrix of a Goppa code [6] of length n symbols, capable of correcting t errors. This matrix is the public key. The digital cryptogram is formed from codewords corrupted by exactly t randomly, or t pseudorandomly, chosen bit errors. The security is provided by the fact that it is

© The Author(s) 2017
M. Tomlinson et al., *Error-Correction Coding and Decoding*,
Signals and Communication Technology,
DOI 10.1007/978-3-319-51103-0_19

impossible to remove the unknown bit errors unless the original unpermuted Goppa code, the private key, is known in which case the errors can be removed by correcting them and then descrambling the information bits in the codeword to recover the original message. Any attempt to descramble the information bits without removing the errors first just results in a scrambled mess. In the original paper by McEliece [8], the Goppa codeword length n is 1024 and t is 50. The number of possible error combinations is 3.19×10^{85} equivalent to a secret key of length 284 bits given a brute force attack. (There are more sophisticated attacks which reduce the equivalent secret key length and these are discussed later in this chapter.)

In a variation of the original theme, after first partitioning the message into message vectors of length k bits each and encoding these message vectors into codewords, the codewords are corrupted by a combination of bit errors and bit deletions to form the cryptogram. The number of bit errors in each corrupted codeword is not fixed, but is an integer s, which is randomly chosen, with the constraint that, $s \leq t$. This increases the number of possible error combinations, thereby increasing the security of the system. As a consequence $2(t - s)$ bits may be deleted from each codeword in random positions adding to the security of the cryptogram as well as reducing its size, without shortening the message. In the case of the original example, above, with $\frac{t}{2} \leq s \leq t$ the number of possible error combinations is increased to 3.36×10^{85} and the average codeword in the cryptogram is reduced to 999 bits from 1024 bits.

Most encryption systems are deterministic in which there is a one-to-one correspondence between the message and the cryptogram with no random variations. Security can be improved through the use of a truly, random integer generator, not a pseudorandom generator to form the cryptogram. Consequently, the cryptogram is not predictable or deterministic. Even with the same message and public key, the cryptogram produced will be different each time and without knowledge of the random errors and bit deletions, which may be determined only by using the structure of the Goppa code, recovery of the original message is practically impossible.

The basic McEliece encryption system has little resistance to chosen-plaintext (message) attacks. For example, if the same message is encrypted twice and the two cryptograms are added modulo 2, the codeword of the permuted Goppa code cancels out and the result is the sum of the two error patterns. Clearly the encryption method does not provide indistinguishability under chosen-plaintext attack (IND-CPA), a quality measure used by the cryptographic community.

However, an additional technique may be used which does provide (IND-CPA) and results in semantically secure cryptograms. The technique is to scramble the message twice by using a second scrambler. With scrambling the message using the fixed non-singular matrix contained in the public key as well, a different scrambler is used to scramble each message in addition. The scrambling function of this second scrambler is derived from the random error vector which is added to the codeword to produce the corrupted codeword after encoding using the permuted, scrambled generator matrix of a Goppa code. As the constructed digital cryptogram is a function of truly randomly chosen vectors, not pseudorandomly chosen vectors, or a fixed vector, the security of this public key encryption system is enhanced compared to the standard system. Even with an identical message and using exactly the same public key, the resulting cryptograms will have no similarity at all to any previously generated cryptograms. This is not true for the standard McEliece public key system as each codeword will only differ in a maximum of $2t$ bit positions. Providing this semantic security eliminates the risk from known plaintext attacks and is useful in several applications such as in RFID, and these are discussed later in the chapter.

An alternative to using a second scrambler is to use a cryptographic hash function such as SHA-256 [11] or SHA-3 [12] to calculate the hash of each t bit error pattern and add, modulo 2, the first k bits of the hash values to the message prior to encoding. Effectively the message is encrypted with a stream cipher prior to encoding.

Having provided additional message scrambling, it now becomes safe to represent the generator matrix in reduced echelon form, i.e. a $k \times k$ identity matrix followed by a $(n - k) \times k$ matrix for the parity bits. Consequently, the public key may be reduced in size from a $n \times k$ matrix to a $(n - k) \times k$ matrix corresponding typically to a reduction in size of around 65%. This is useful because one of the criticisms of the McEliece system is the relatively large size of the public keys.

Most attacks on the McEliece system are blind attacks and rely on the assumption that there are exactly t errors in each corrupted codeword. If there are more than t errors these attacks fail. Consequently, to enhance the security of the system, additional errors known only to intended recipients may be inserted into the digital cryptogram so that each corrupted codeword contains more than t errors. A sophisticated method of introducing the additional errors is not necessary since provided there are sufficient additional errors to defeat decryption based on guessing the positions of the additional errors the message is theoretically unrecoverable from the corrupted digital cryptogram even with knowledge of the private key. This feature may find applications where a message needs to be distributed to several recipients using the same or different public/private keys at the same time, possibly in a commercial, competitive environment. The corrupted digital cryptograms may be sent to each recipient arriving asynchronously, due to variable network delays and only a relatively short secret key containing information of the additional error positions needs to be sent at the same time to all recipients.

In another arrangement designed to enhance the security of the system, additional errors are inserted into each codeword in positions defined by a position vector, which is derived from a cryptographic hash of the previous message vector. Standard hash functions may be used such as SHA-256 [11] or SHA-3 [12]. The first message vector can use a position vector derived from a hash or message already known by the recipient of the cryptogram.

These arrangements may be used in a wide number of different applications such as active and passive RFID, secure barcodes, secure ticketing, magnetic cards, message services, email applications, digital broadcasting, digital communications, video communications and digital storage. Encryption and decryption is amenable to high speed implementation operating at speeds beyond 1 Gbit/s.

19.2 Details of the Encryption System

The security strength of the McEliece public key encryption system stems from the fact that a truly random binary error pattern is added to the encoded message as part of the digital cryptogram. Even with the same message and the same public key a different digital cryptogram is produced each time. Each message is encoded with a scrambled, binary mapped, permuted, version of a $GF(2^m)$ Goppa code. Without the knowledge of the particular Goppa code that is used, the error pattern cannot be corrected and the message cannot be recovered. It is not possible to deduce which particular Goppa code is being used from the public key, which is the matrix used for encoding, because this matrix is a scrambled, permuted version of the original encoding matrix of the Goppa code, plus the fact that for a given m there are an extremely large number of Goppa codes [8].

The message information to be sent, if not in digital form, is digitally encoded into binary form comprising a sequence of information bits. The method of encryption is shown in Fig. 19.1. The message comprising a sequence of information bits is formatted by appending dummy bits as necessary into an integral number m of binary message vectors of length k bits each. This is carried out by *format into message vectors* shown in Fig. 19.1. Each message vector is scrambled and encoded into a codeword, n bits long, defined by an error-correcting code which is derived from a binary Goppa code and a scrambling matrix. The binary Goppa code is derived from a non-binary Goppa code and the procedure is described below for a specific example.

The *encode using public key* shown in Fig. 19.1 carries out the scrambling and codeword encoding for each message vector by selecting rows of the codeword generator matrix according to the message bits contained in the message vector. This operation is described in more detail below for a specific example. The codeword generator matrix to be used for encoding is defined by the public key which is stored in a buffer memory, *public key* shown in Fig. 19.1. As shown in Fig. 19.1, a random number generator generates a number s internally constrained to be less than or equal to t and this is carried out by *generate number of random errors (s)*. The parameter t is the number of bit errors that the Goppa code can correct.

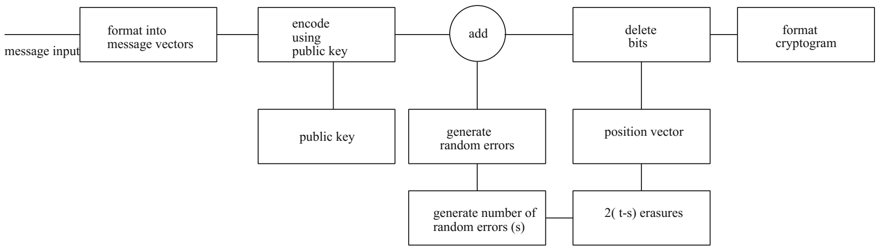


Fig. 19.1 Public key encryption system with s random bit errors and $2(t-s)$ bit deletions

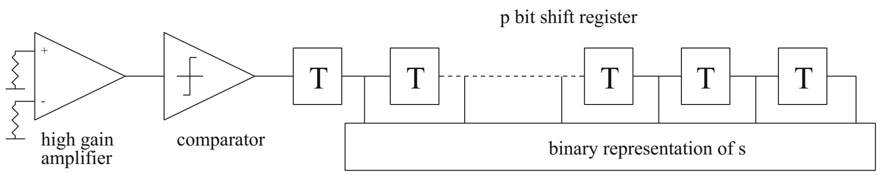


Fig. 19.2 Random integer generator of the number of added, random bit errors, s

The number of random errors s is input to *generate random errors* which for each codeword, initialises an n bit buffer memory with zeros, and uses a random number generator to generate s 1's in s random positions of the buffer memory. The contents of the n bit buffer are added to the codeword of n bits by *add* shown in Fig. 19.1. The 1's are added modulo 2 which inverts the codeword bits in these positions so that these bits are in error. In Fig. 19.1, $t - s$ *erasures* takes the input s, calculates $2(t-s)$ and outputs this value to *position vector* which comprises a buffer memory of n bits containing a sequence of integers corresponding to a position vector described below. The first $2(t - s)$ integers are input to *delete bits* which deletes the bits in the corresponding positions of the codeword so that $2(t - s)$ bits of the codeword are deleted. The procedure is carried out for each codeword so that each codeword is randomly shortened due to deleted bits and corrupted with a random number of bit errors in random positions. In Fig. 19.1, *format cryptogram* has the sequence of shortened corrupted codewords as input and appends these together, together with formatting information to produce the cryptogram.

The highest level of security is provided when the block *generate number of random errors (s)* of Fig. 19.1 is replaced by a truly random number generator and not a pseudorandom generator. An example of a random number generator is shown in Fig. 19.2.

The differential amplifier with high gain amplifies the thermal noise generated by the resistor terminated inputs. The output of the amplifier is the amplified random noise which is input to a comparator which carries out binary quantisation. The comparator output is 1 if the amplifier output is a positive voltage and 0 otherwise. This produces 1's and 0's with equal probability at the output of the comparator. The output of the comparator is clocked into a shift register having p shift register stages, each of delay T. The clock rate is $\frac{1}{T}$. After p clock cycles, the contents of the shift register represent a number in binary which is the random number s having a uniform probability distribution between 0 and $2^p - 1$.

One or more of the bits output from the shift register may be permanently set to 1 to provide a lower limit to the random number of errors s. As an example, if the 4th bit (counting from the least significant bits) is permanently set to 1 then s has a uniform probability distribution between $2^3 = 8$ and $2^p - 1$.

Similarly, the highest level of security is provided if the positions of the errors generated by *generate random errors* of Fig. 19.1 is a truly random number generator and not a pseudorandom generator. An example of an arrangement which generates

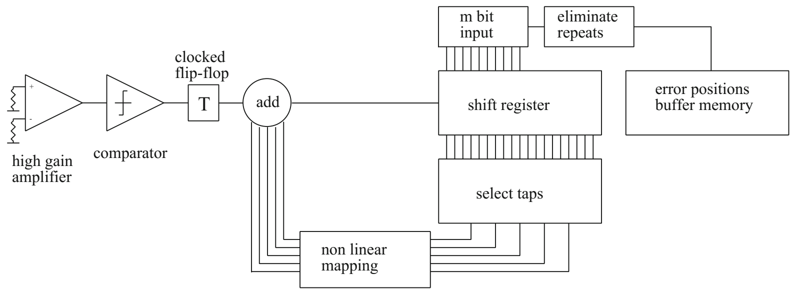


Fig. 19.3 Random integer generator of error positions

truly random positions in the range of 0 to $2^m - 1$ corresponding to the codeword length is shown in Fig. 19.3.

As shown in Fig. 19.3, the differential amplifier, with high gain amplifies the thermal noise generated by the resistor terminated inputs. The output of the amplifier is the amplified random noise which is input to a comparator which outputs a 1 if the amplifier output is a positive voltage and a 0 otherwise. This produces 1's and 0's with equal probability at the output of the comparator. The output of the comparator is clocked into a flip-flop clocked at $\frac{1}{T}$, with the same clock source as the shift register shown in Fig. 19.3, *shift register*. The output of the flip-flop is a clocked output of truly random 1's and 0's which is input to a nonlinear feedback shift register arrangement.

The output of the flip-flop is input to a modulo 2, adder *add* which is added to the outputs of a nonlinear mapping of u selected outputs of the shift register. Which outputs are to be selected correspond to the key which is being used. The parameter u is a design parameter, typically equal to 8.

The nonlinear mapping *nonlinear mapping* shown in Fig. 19.3 has a pseudorandom one-to-one correspondence between each of the 2^u input states to each of the 2^u output states. An example of such a one to one correspondence, for $u = 4$ is given in Table 19.1. For example, the first entry, 0000, value 0 is mapped to 0011, value 3.

The shift register typically has a relatively large number of stages, 64 is a typical number of stages and a number of tapped outputs, typically 8. The relationship between the input of the shift register a_{in} and the tapped outputs is usually represented by the delay operator D. Defining the tap positions as w_i, for $i = 0$ to i_{max}, the input to the nonlinear mapping *nonlinear mapping* shown in Fig. 19.3, defined as x_i for $i = 0$ to i_{max}, is

$$x_i = a_{in}D^{w_i} \tag{19.1}$$

and the output y_j after the mapping function, depicted as M is

$$y_j = M[x_i] = M[a_{in}D^{w_i}] \tag{19.2}$$

The input to the shift register is the output of the adder given by the sum of the random input R_{nd} and the summed output of the mapped outputs. Accordingly,

Table 19.1 Example of nonlinear mapping for $u = 4$

0000	$\rightarrow$	0011
0001	$\rightarrow$	1011
0010	$\rightarrow$	0111
0011	$\rightarrow$	0110
0100	$\rightarrow$	1111
0101	$\rightarrow$	0001
0110	$\rightarrow$	1001
0111	$\rightarrow$	1100
1000	$\rightarrow$	1010
1001	$\rightarrow$	0000
1010	$\rightarrow$	1000
1011	$\rightarrow$	0010
1100	$\rightarrow$	0101
1101	$\rightarrow$	1110
1110	$\rightarrow$	0100
1111	$\rightarrow$	1101

$$a_{in} = R_{nd} + \sum_{j=0}^{i_{max}} y_j = R_{nd} + \sum_{j=0}^{i_{max}} M[x_i] = R_{nd} + \sum_{j=0}^{i_{max}} M[a_{in}D^{w_i}] \tag{19.3}$$

It can be seen that the shift register input a_{in} is a nonlinear function of delayed outputs of itself added to the random input R_{nd}, and so will be a random binary function.

The positions of the errors are given by the output of *m-bit input* shown in Fig. 19.3, an m bit memory register and defined as e_{pos}. Consider that the first m outputs of the shift register are used as the input to *m-bit input*. The output of *m-bit input* is a binary representation of a number given by

$$e_{pos} = \sum_{j=0}^{m-1} 2^j \times a_{in}D^j \tag{19.4}$$

Since a_{in} is a random binary function, e_{pos} will be an integer between 0 and $2^m - 1$ randomly distributed with a uniform distribution. As shown in Fig. 19.3, these randomly generated integers are stored in memory in *error positions buffer memory* after *eliminate repeats* has eliminated any repeated numbers, since repeated integers will occur from time to time in any independently distributed random integer generator.

The random bit errors and bit deletions can only be corrected with the knowledge of the particular non-binary Goppa code, the private key, which is used in deriving the codeword generator matrix. Reviewing the background on Goppa codes: Goppa defined a family of codes [6] where the coordinates of each codeword

$\{c_0, c_1, c_2, \ldots c_{2^m-1}\}$ with $\{c_0 = x_0, c_1 = x_1, c_2 = x_2, \ldots c_{2^m-1} = x_{2^m-1}\}$ satisfy the congruence $p(z)$ modulo $g(z) = 0$ where $g(z)$ is now known as the Goppa polynomial and $p(z)$ is the Lagrange interpolation polynomial.

Goppa codes have coefficients from $GF(2^m)$ and provided $g(z)$ has no roots which are elements of $GF(2^m)$ (which is straightforward to achieve) the Goppa codes have parameters $(2^m, k, 2^m - k + 1)$. Goppa codes can be converted into binary codes. Provided that $g(z)$ has no roots which are elements of $GF(2^m)$ and has no repeated roots, the binary code parameters are $(2^m, 2^m - mt, d_{min})$ where $d_{min} \geq 2t + 1$, the Goppa code bound on minimum Hamming distance. Most binary Goppa codes have equality for the bound and t is the number of correctable errors.

For a Goppa polynomial of degree r, there are r parity check equations defined from the congruence. Denoting $g(z)$ by

$$g(z) = g_r z^r + g_{r-1} z^{r-1} + g_{r-2} z^{r-2} + \cdots + g_1 z + g_0 \tag{19.5}$$

$$\sum_{i=0}^{2^m-1} \frac{c_i}{z - \alpha_i} = 0 \quad \text{modulo}\, g(z) \tag{19.6}$$

Since Eq. (19.6) is modulo $g(z)$ then $g(z)$ is equivalent to 0, and we can add $g(z)$ to the numerator. Dividing each term $z - \alpha_i$ into $1 + g(z)$ produces the following

$$\frac{g(z) + 1}{z - \alpha_i} = q_i(z) + \frac{r_m + 1}{z - \alpha_i} \tag{19.7}$$

where r_m is the remainder, an element of $GF(2^m)$ after dividing $g(z)$ by $z - \alpha_i$.

As r_m is a scalar, $g(z)$ may simply be pre-multiplied by $\frac{1}{r_m}$ so that the remainder cancels with the other numerator term which is 1.

$$\frac{\frac{g(z)}{r_m} + 1}{z - \alpha_i} = \frac{q_i(z)}{r_m} + \frac{\frac{r_m}{r_m} + 1}{z - \alpha_i} = \frac{q(z)}{r_m} \tag{19.8}$$

As

$$g(z) = (z - \alpha_i) q_i(z) + r_m \tag{19.9}$$

When $z = \alpha_i$, $r_m = g(\alpha_i)$.

Substituting for r_m in Eq. (19.8) produces

$$\frac{\frac{g(z)}{g(\alpha_i)} + 1}{z - \alpha_i} = \frac{q_i(z)}{g(\alpha_i)} \tag{19.10}$$

Since $\frac{g(z)}{g(\alpha_i)}$ modulo $g(z) = 0$

$$\frac{1}{z - \alpha_i} = \frac{q_i(z)}{g(\alpha_i)} \tag{19.11}$$

The quotient polynomial $q_i(z)$ is a polynomial of degree $r-1$ with coefficients which are a function of α_i and the Goppa polynomial coefficients. Denoting $q_i(z)$ as

$$q_i(z) = q_{i,0} + q_{i,1}z + q_{i,2}z^2 + q_{i,3}z^3 + \cdots + q_{i,(r-1)}z^{r-1} \tag{19.12}$$

Since the coefficients of each power of z sum to zero the r parity check equations are given by

$$\sum_{i=0}^{2^m-1} \frac{c_i q_{i,j}}{g(\alpha_i)} = 0 \quad \text{for} \quad j = 0 \quad \text{to} \quad r-1 \tag{19.13}$$

If the Goppa polynomial has any roots which are elements of $GF(2^m)$, say α_j, then the codeword coordinate c_j has to be permanently set to zero in order to satisfy the parity check equations. Effectively the codelength is shortened by the number of roots of $g(z)$ which are elements of $GF(2^m)$. Usually the Goppa polynomial is chosen to have distinct roots which are not in $GF(2^m)$.

The security depends upon the number of bit errors added and in practical examples to provide sufficient security, it is necessary to use long Goppa codes of length 2048 bits, 4096 bits or longer. For brevity, the procedure will be described using an example of a binary Goppa code of length 32 bits capable of correcting 4 bit errors. It is important to note that all binary Goppa codes are derived from non-binary Goppa codes which are designed first.

In this example, the non-binary Goppa code consists of 32 symbols from the Galois field $GF(2^5)$ and each symbol takes on 32 possible values with the code capable of correcting two errors. There are 28 information symbols and 4 parity check symbols. (It should be noted that when the Goppa code is used with information symbols restricted to binary values as in a binary Goppa code, twice as many errors can be corrected). The non-binary Goppa code has parameters of a (32, 28, 5) code. There are 4 parity check symbols defined by the 4 parity check equations and the Goppa polynomial has degree 4. Choosing arbitrarily as the Goppa polynomial, the polynomial $1 + z + z^4$ which has roots only in $GF(16)$ and none in $GF(32)$, we determine $q_i(z)$ by dividing by $z - \alpha_i$.

$$q_i(z) = z^3 + \alpha_i z^2 + \alpha_i^2 z + (1 + \alpha_i^3) \tag{19.14}$$

The 4 parity check equations are

$$\sum_{i=0}^{31} \frac{c_i}{g(\alpha_i)} = 0 \tag{19.15}$$

$$\sum_{i=0}^{31} \frac{c_i \alpha_i}{g(\alpha_i)} = 0 \tag{19.16}$$

$$\sum_{i=0}^{31} \frac{c_i \alpha_i^2}{g(\alpha_i)} = 0 \tag{19.17}$$

$$\sum_{i=0}^{31} \frac{c_i (1 + \alpha_i^3)}{g(\alpha_i)} = 0 \tag{19.18}$$

Using the $GF(2^5)$ Table 19.2 to evaluate the different terms for $GF(2^5)$, the parity check matrix is

$$\mathbf{H}_{(\mathbf{32},\,\mathbf{28},\,\mathbf{5})} = \begin{bmatrix} 1 & 1 & \alpha^{14} & \alpha^{28} & \alpha^{20} & \alpha^{25} & \ldots & \alpha^{10} \\ 0 & 1 & \alpha^{15} & \alpha^{30} & \alpha^{23} & \alpha^{29} & \ldots & \alpha^{9} \\ 0 & 1 & \alpha^{16} & \alpha^{1} & \alpha^{26} & \alpha^{2} & \ldots & \alpha^{8} \\ 1 & 0 & \alpha^{12} & \alpha^{24} & \alpha^{5} & \alpha^{17} & \ldots & \alpha^{5} \end{bmatrix} \tag{19.19}$$

To implement the Goppa code as a binary code, the symbols in the parity check matrix are replaced with their m-bit binary column representations of each respective $GF(2^m)$ symbol. For the (32, 28, 5) Goppa code above, each of the 4 parity symbols will be represented as a 5-bit symbol from Table 19.2. The parity check matrix will now have 20 rows for the binary code. The minimum Hamming distance of the binary Goppa code is improved from $r + 1$ to $2r + 1$. Correspondingly, the example binary Goppa code becomes a (32, 12, 9) code with parity check matrix:

$$\mathbf{H}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})} = \begin{bmatrix} 1\,1\,1\,0\,0\,1 \ldots 1 \\ 0\,0\,0\,1\,0\,0 \ldots 0 \\ 0\,0\,1\,1\,1\,0 \ldots 0 \\ 0\,0\,1\,0\,1\,1 \ldots 0 \\ 0\,0\,1\,1\,0\,1 \ldots 1 \\ 0\,1\,1\,0\,1\,1 \ldots 0 \\ 0\,0\,1\,1\,1\,0 \ldots 1 \\ 0\,0\,1\,0\,1\,0 \ldots 0 \\ 0\,0\,1\,0\,1\,1 \ldots 1 \\ 0\,0\,1\,1\,0\,0 \ldots 1 \\ 0\,1\,1\,1\,1\,0 \ldots 1 \\ 0\,0\,1\,0\,1\,0 \ldots 0 \\ 0\,0\,0\,0\,1\,1 \ldots 1 \\ 0\,0\,1\,0\,0\,0 \ldots 1 \\ 0\,0\,1\,0\,1\,0 \ldots 0 \\ 1\,0\,0\,0\,1\,1 \ldots 1 \\ 0\,0\,1\,1\,0\,1 \ldots 0 \\ 0\,0\,1\,1\,1\,0 \ldots 1 \\ 0\,0\,1\,1\,0\,0 \ldots 0 \\ 0\,0\,0\,1\,0\,1 \ldots 0 \end{bmatrix} \tag{19.20}$$

Table 19.2 $GF(32)$ non-zero extension field elements defined by $1+\alpha^2+\alpha^5=0$

$\alpha^0 = 1$
$\alpha^1 = \alpha$
$\alpha^2 = \alpha^2$
$\alpha^3 = \alpha^3$
$\alpha^4 = \alpha^4$
$\alpha^5 = 1+\alpha^2$
$\alpha^6 = \alpha+\alpha^3$
$\alpha^7 = \alpha^2+\alpha^4$
$\alpha^8 = 1+\alpha^2+\alpha^3$
$\alpha^9 = \alpha+\alpha^3+\alpha^4$
$\alpha^{10} = 1+\alpha^4$
$\alpha^{11} = 1+\alpha+\alpha^2$
$\alpha^{12} = \alpha+\alpha^2+\alpha^3$
$\alpha^{13} = \alpha^2+\alpha^3+\alpha^4$
$\alpha^{14} = 1+\alpha^2+\alpha^3+\alpha^4$
$\alpha^{15} = 1+\alpha+\alpha^2+\alpha^3+\alpha^4$
$\alpha^{16} = 1+\alpha+\alpha^3+\alpha^4$
$\alpha^{17} = 1+\alpha+\alpha^4$
$\alpha^{18} = 1+\alpha$
$\alpha^{19} = \alpha+\alpha^2$
$\alpha^{20} = \alpha^2+\alpha^3$
$\alpha^{21} = \alpha^3+\alpha^4$
$\alpha^{22} = 1+\alpha^2+\alpha^4$
$\alpha^{23} = 1+\alpha+\alpha^2+\alpha^3$
$\alpha^{24} = \alpha+\alpha^2+\alpha^3+\alpha^4$
$\alpha^{25} = 1+\alpha^3+\alpha^4$
$\alpha^{26} = 1+\alpha+\alpha^2+\alpha^4$
$\alpha^{27} = 1+\alpha+\alpha^3$
$\alpha^{28} = \alpha+\alpha^2+\alpha^4$
$\alpha^{29} = 1+\alpha^3$
$\alpha^{30} = \alpha+\alpha^4$

The next step is to turn the parity check matrix into reduced echelon form by using elementary matrix row and column operations so that there are 20 rows representing 20 independent parity check equations for each parity bit. From the reduced echelon parity check matrix, the generator matrix can be obtained straightforwardly as it is the transpose of the reduced echelon parity check matrix. The resulting generator matrix is:

$$\mathbf{G_{(32,\,12,\,9)}} = \left[\begin{array}{cccccccccccccccccccccccccccccccc}
1&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&0&1&1&1&1&0&1&0&0&1&1&1&0&1&1&0\\
0&1&0&0&0&0&0&0&0&0&0&0&1&1&0&1&0&1&1&0&0&0&0&1&1&0&1&0&1&1&0&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&1&0&0&0&0&1&1&0&1&0&0&1&0\\
0&0&0&1&0&0&0&0&0&0&0&0&1&1&1&1&1&1&0&1&0&1&0&1&1&0&0&0&1&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&0&1&1&0&0&0&0&1&1&0&1&0&0&1&1&1&0&0&0&0&1\\
0&0&0&0&0&1&0&0&0&0&0&0&1&1&1&1&1&0&0&1&1&0&0&0&1&0&1&0&1&0&0&1\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&1&0&1&1&1&1&1&1&0&0&0&1&1&1&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&1&1&1&0&0&1&1&0&0&1&1&1&0&0&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&1&1&1&1&0&1&1&1&1&1&1&1&0&1&1&1&1&1&0&1\\
0&0&0&0&0&0&0&0&0&1&0&0&1&1&1&0&1&1&0&1&1&1&1&0&0&1&0&0&1&0&1&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&1&0&0&1&1&1&0&1&1&0&0&1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&1&1&1&0&1&1&0&0&1&0&0&1&1&0&1&1&0&1&1&1
\end{array}\right] \tag{19.21}$$

It will be noticed that the generator matrix is in reduced echelon form and has 12 rows, one row for each information bit. Each row is the codeword resulting from that information bit equal to a 1, all other information bits equal to 0.

The next step is to scramble the information bits by multiplying by a $k \times k$ non-singular matrix, that is one that is invertible. As a simple example, the following 12×12 matrix is invertible.

$$\mathbf{NS_{12\times 12}} = \left[\begin{array}{cccccccccccc}
0&1&1&1&0&1&0&0&1&1&1&0\\
0&0&1&1&1&0&1&0&0&1&1&1\\
1&0&0&1&1&1&0&1&0&0&1&1\\
1&1&0&0&1&1&1&0&1&0&0&1\\
1&1&1&0&0&1&1&1&0&1&0&0\\
0&1&1&1&0&0&1&1&1&0&1&0\\
0&0&1&1&1&0&0&1&1&1&0&1\\
1&0&0&1&1&1&0&0&1&1&1&0\\
0&1&0&0&1&1&1&0&0&1&1&1\\
1&0&1&0&0&1&1&1&0&0&1&1\\
1&1&0&1&0&0&1&1&1&0&0&1\\
1&1&1&0&1&0&0&1&1&1&0&0
\end{array}\right] \tag{19.22}$$

The above is invertible using the following matrix:

$$\mathbf{NS^{-1}_{12\times 12}} = \left[\begin{array}{cccccccccccc}
1&1&0&1&0&0&0&0&0&0&0&0\\
0&1&1&0&1&0&0&0&0&0&0&0\\
0&0&1&1&0&1&0&0&0&0&0&0\\
0&0&0&1&1&0&1&0&0&0&0&0\\
0&0&0&0&1&1&0&1&0&0&0&0\\
0&0&0&0&0&1&1&0&1&0&0&0\\
0&0&0&0&0&0&1&1&0&1&0&0\\
0&0&0&0&0&0&0&1&1&0&1&0\\
0&0&0&0&0&0&0&0&1&1&0&1\\
1&0&0&0&0&0&0&0&0&1&1&0\\
0&1&0&0&0&0&0&0&0&0&1&1\\
1&0&1&0&0&0&0&0&0&0&0&1
\end{array}\right] \tag{19.23}$$

The next step is to scramble the generator matrix with the non-singular matrix to produce the scrambled generator matrix given below. The code produced with this generator matrix has the same codewords as the generator matrix given by matrix (19.21) and can correct the same number of errors but there is a different mapping to codewords from a given information bit pattern.

$$\mathbf{SG}_{(32,\,12,\,9)} = \begin{bmatrix}
0&1&1&1&0&1&0&0&1&1&1&0&1&1&1&0&0&1&0&1&1&1&1&0&0&0&0&0&0&1&1&1\\
0&0&1&1&1&0&1&0&0&1&1&1&0&0&0&1&1&0&0&1&1&0&1&0&0&0&1&1&0&1&0&1\\
1&0&0&1&1&1&0&1&0&0&1&1&1&1&0&0&1&0&0&1&1&1&0&1&0&0&0&0&1&0&1&1\\
1&1&0&0&1&1&1&0&1&0&0&1&0&0&1&1&1&0&1&1&1&1&1&0&0&1&0&0&0&1&0&0\\
1&1&1&0&0&1&1&1&0&1&0&0&0&0&0&0&0&1&1&0&0&1&1&0&1&0&0&1&0&0&1&0\\
0&1&1&1&0&0&1&1&1&0&1&0&1&1&0&0&0&0&1&1&1&1&1&0&0&0&1&1&1&1&0&1\\
0&0&1&1&1&0&0&1&1&1&0&1&1&1&0&0&0&0&0&0&1&1&0&1&0&1&0&1&1&1&0&0\\
1&0&0&1&1&1&0&0&1&1&1&0&0&0&0&0&0&1&1&0&1&1&0&0&0&1&0&0&1&1&1&0\\
0&1&0&0&1&1&1&0&0&1&1&1&1&1&1&1&1&0&1&0&1&0&1&1&1&1&0&0&1&0&0&0\\
1&0&1&0&0&1&1&1&0&0&1&1&1&1&0&0&1&1&0&1&1&1&1&0&0&0&0&0&1&1&1&0\\
1&1&0&1&0&0&1&1&1&0&0&1&1&1&0&0&0&1&0&1&1&1&1&0&0&1&0&0&0&0&1&0\\
1&1&1&0&1&0&0&1&1&1&0&0&1&1&0&0&0&0&0&0&1&0&1&0&1&0&1&1&1&0&1&1
\end{bmatrix} \tag{19.24}$$

It may be seen that, for example, the first row of this matrix is the modulo 2 sum of rows 1, 2, 3, 5, 8, 9 and 10 of matrix (19.21) in accordance with the non-singular matrix (19.22).

The final step in producing the public key generator matrix for the codewords from the message vectors is to permute the columns of the matrix above. Any permutation may be randomly chosen. For example we may use the following permutation:

$$\begin{array}{cccccccccccccccccccccccccccccccc}
27&15&4&2&19&21&17&14&7&16&20&1&29&8&11&12&25&5&30&24&6&18&13&3&0&26&23&28&22&31&9&10\\
0&1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31
\end{array} \tag{19.25}$$

so that for example column 0 of matrix (19.24) becomes column 24 of the permuted generator matrix and column 31 of matrix (19.24) becomes column 29 of the permuted generator matrix. The resulting, permuted generator matrix is given below.

$$\mathbf{PSG}_{(32,\,12,\,9)} = \begin{bmatrix}
0&0&0&1&1&1&1&1&0&0&1&1&1&1&0&1&0&1&1&0&0&0&1&1&0&0&0&0&1&1&1&1\\
1&1&1&1&1&0&0&0&0&1&1&0&1&0&1&0&0&0&0&0&1&0&0&1&0&1&0&0&1&1&1&1\\
0&0&1&0&1&1&0&0&1&1&1&0&0&0&1&1&0&1&1&0&0&0&1&1&1&0&1&1&0&1&0&1\\
0&1&1&0&1&1&0&1&0&1&1&1&1&1&1&0&1&1&0&0&1&1&0&0&1&0&0&0&1&0&0&0\\
1&0&0&1&0&1&1&0&1&0&0&1&0&0&0&0&0&1&1&1&1&1&0&0&1&0&0&0&1&0&1&0\\
1&0&0&1&1&1&0&0&1&0&1&1&1&1&0&1&0&0&0&0&1&1&1&1&0&1&0&1&1&1&0&1\\
1&0&1&1&0&1&0&0&1&0&1&0&1&1&1&1&1&0&0&0&0&0&1&1&0&0&1&1&0&0&1&0\\
0&0&1&0&0&1&1&0&0&0&1&0&1&1&0&0&1&1&1&0&0&1&0&1&1&0&0&1&0&0&1&1\\
0&1&1&0&0&0&0&1&0&1&1&1&0&0&1&1&1&1&0&1&1&1&1&0&0&0&1&1&1&0&1&1\\
0&0&0&1&1&1&1&0&1&1&1&0&1&0&1&1&0&1&1&0&1&0&1&0&1&0&0&1&1&0&0&1\\
0&0&0&0&1&1&1&0&1&0&1&1&0&1&1&1&1&0&1&0&1&0&1&1&1&0&0&0&1&0&0&0\\
1&0&1&1&0&0&0&0&1&0&1&1&0&1&0&1&0&0&1&1&0&0&1&0&1&1&0&1&1&1&1&0
\end{bmatrix} \tag{19.26}$$

With this particular example of the Goppa code, the message needs to be split into message vectors of length 12 bits, adding padding bits as necessary so that there is an integral number of message vectors. As a simple example of a plaintext message, consider that the message consists of a single message vector with the information bit pattern:

$$\{0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1\}$$

Starting with an all 0's vector, where the information bit pattern is 1, the corresponding row from the permuted, scrambled matrix, matrix (19.26) with the same position is added modulo 2 to the result so far to produce the codeword which will form the digital cryptogram plus added random errors. In this example, this codeword is generated from adding modulo 2, rows 2, 4, 5, 6 and 12 from the permuted, scrambled matrix, matrix (19.26) to produce:

$$\begin{array}{cccccccccccccccccccccccccccccccc}
0&0&0&1&1&1&1&1&0&0&1&1&1&1&0&1&0&1&1&0&0&0&1&1&0&0&0&0&1&1&1&1\\
+&+\\
0&0&1&0&1&1&0&0&1&1&1&0&0&0&1&1&0&1&1&0&0&0&1&1&1&0&1&1&0&1&0&1\\
+&+\\
0&1&1&0&1&1&0&1&0&1&1&1&1&1&1&0&1&1&0&0&1&1&0&0&1&0&0&0&1&0&0&0\\
+&+\\
1&0&0&1&0&1&1&0&1&0&0&1&0&0&0&0&0&1&1&1&1&1&0&0&1&0&0&0&1&0&1&0\\
+&+\\
1&0&1&1&0&0&0&0&1&0&1&1&0&1&0&1&0&0&1&1&0&0&1&0&1&1&0&1&1&1&1&0\\
\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel&\shortparallel\\
0&1&1&1&1&0&0&0&1&0&0&0&0&1&0&1&1&0&0&0&0&0&1&0&0&1&1&0&0&1&1&0
\end{array} \tag{19.27}$$

The resulting codeword is:

$$\{0\,1\,1\,1\,1\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\}$$

This Goppa code can correct up to 4 errors, ($t = 4$), so a random number is chosen for the number of bits to be in error (s) and $2(t-s)$ bits are deleted from the codeword in pre-determined positions. The pre-determined positions may be given by a secret key, a position vector, known only to the originator and intended recipient of the cryptogram. It may be included as part of the public key, or may be contained in a previous cryptogram sent to the recipient. An example of a position vector, which defines the bit positions to be deleted is:

$$\{19,\ 3,\ 27,\ 17,\ 8,\ 30,\ 11,\ 15,\ 2,\ 5,\ 19,\ \ldots,\ 25\}.$$

The notation being, for example, that if there are 2 bits to be deleted, the bit positions to be deleted are the first 2 bit positions in the position vector, 19 and 3. As well as the secret key, the position vector, the recipient needs to know the number of bits deleted, preferably with the information provided in a secure way. One method is for the message vector to contain, as part of the message, a number indicating the number of errors to be deleted in the next codeword, the following codeword (not the current codeword); the first codeword having a known, fixed number of deleted bits.

The number of bit errors and the bit error positions are randomly chosen to be in error. A truly random source such as a thermal noise source as described above produces the most secure results, but a pseudorandom generator can be used instead, particularly, if seeded from the time of day with a fine time resolution such as 1 ms. If the number of random errors chosen is too few, the security of the digital cryptogram will be compromised. Correspondingly, the minimum number of errors chosen is a design parameter depending upon the length of the Goppa code and t, the number of correctable errors. A suitable choice for the minimum number of errors chosen in practice lies between $\frac{t}{2}$ and $\frac{3t}{4}$.

For the example above, consider that the number of bit errors is 2 and these are randomly chosen to be in positions 7 and 23 (starting the position index from 0). The bits in these positions in the codeword are inverted to produce the result:

$$\{0\,1\,1\,1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,1\,1\,0\}.$$

As there are 2 bits in error, 4 bits $(2(t-s) = 2(4-2))$ may be deleted. Using the position vector example above, the deleted bits are in positions $\{19,\ 3,\ 27$ and $17\}$ resulting in 28 bits,

$$\{0\,1\,1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\}.$$

This vector forms the digital cryptogram which is transmitted or stored depending upon the application.

The intended recipient of this cryptogram retrieves the message in a series of steps. Figure 19.4 shows the decryption system. The retrieved cryptogram is formatted into corrupted codewords by *format into corrupted codewords* shown in Fig. 19.4. In the formatting process, the number of deleted bits in each codeword is determined from the retrieved length of each codeword. The next step is to insert 0's in the deleted bit positions so that each corrupted codeword is of the correct length. This is carried out using *fill erased positions with 0's* as input, the position vector stored in a buffer memory as *position vector* in Fig. 19.4 and the number of deleted (erased) bits from *format into corrupted codewords*. For the example above, the recipient first receives or otherwise retrieves the cryptogram $\{0\,1\,1\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,1\,1\,0\,1\,1\,0\,1\,1\,0\}$. Knowing the number of deleted bits and their positions, the recipient inserts 0's in positions $\{19,\ 3,\ 27$ and $17\}$ to produce:

$$\{0\,1\,1\,0\,1\,0\,0\,1\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,1\,1\,0\}n$$

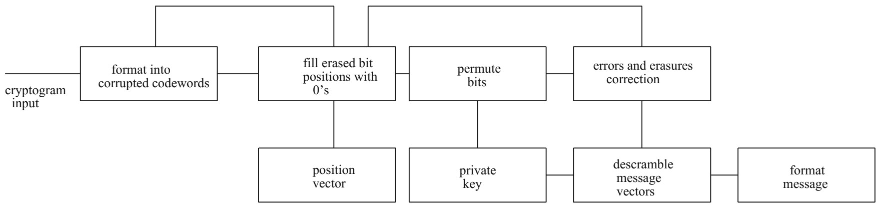

cryptogram input
format into corrupted codewords
fill erased bit positions with 0's
permute bits
errors and erasures correction
position vector
private key
descramble message vectors
format message

Fig. 19.4 Private key decryption system with s random bit errors and $2(t-s)$ bit deletions

The private key contains the information of which Goppa code was used, the inverse of the non-singular matrix used to scramble the data and the permutation applied to codeword symbols in constructing the public key generator matrix. This information is stored in *private key* in Fig. 19.4.

For the example, the private key is used to undo the permutation applied to codeword symbols by applying the following permutation:

$$\begin{array}{cccccccccccccccccccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 27 & 15 & 4 & 2 & 19 & 21 & 17 & 14 & 7 & 16 & 20 & 1 & 29 & 8 & 11 & 12 & 25 & 5 & 30 & 24 & 6 & 18 & 13 & 3 & 0 & 26 & 23 & 28 & 22 & 31 & 9 & 10 \end{array} \tag{19.28}$$

so that, for example, bit 24 becomes bit 0 after permutation and bit 27 becomes bit 31 after permutation. The resulting, corrupted codeword is:

$$\{0\,0\,0\,1\,1\,0\,0\,1\,1\,1\,0\,0\,1\,1\,1\,1\,0\,0\,0\,1\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,1\}$$

The permutation is carried out by *permute bits* shown in Fig. 19.4.

The next step is to treat the bits in the corrupted codeword as $GF(2^5)$ symbols and use the parity check matrix, matrix (19.19), from the private key to calculate the syndrome value for each row of the parity check matrix to produce α^{28}, α^7, α^{13}, and α^{19}. This is carried out by an errors and erasures decoder as a first step in correcting the errors and erasures. The errors and erasures are corrected by *errors and erasures correction*, which knows the positions of the erased bits from *fill erased positions with 0's* shown in Fig. 19.4.

In the example, the errors and erasures are corrected using the syndrome values to produce the uncorrupted codeword. There are several published algorithms for errors and erasures decoding [1, 13, 16]. Using, for example, the method described by Sugiyama [16], the uncorrupted codeword is obtained:

$$\{1\,0\,0\,0\,1\,0\,0\,1\,1\,1\,0\,0\,0\,0\,1\,0\,0\,1\,1\,1\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,0\,1\}$$

The scrambled information data is the first 12 bits of this codeword:

$$\{1\,0\,0\,0\,1\,0\,0\,1\,1\,1\,0\,0\}$$

The last step is to unscramble the scrambled data using matrix (19.23) to produce the original message after formatting the unscrambled data:

$$\{0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1\}$$

In Fig. 19.4, *descramble message vectors* take as input the matrix which is the inverse of the non-singular matrix stored in *private key* and output the descramble message vectors to *format message*.

In practice, much longer codes of length n would be used than described above. Typically n is set equal to 1024, 2048, 4096 bits or longer. Longer codes are more secure but the public key is larger and encryption and decryption take longer time.

Consider an example with $n = 1024$, correcting $t = 60$ bit errors with a randomly chosen irreducible Goppa polynomial of degree 60, say, $g(z) = 1+z+z^2+z^{23}+z^{60}$.

Setting the number of inserted bit errors s as a randomly chosen number from 40 to 60, the number of deleted bits correspondingly, is $2(t-s)$, ranging from 40 to 0 and the average codeword length is 994 bits. There are 9.12×10^{96} different bit error combinations providing security, against naive brute force decoding, equivalent to a random key of length 325 bits. The message vector length is 424 bits per codeword of which 6 bits may be assigned to indicate the number of deleted bits in the following codeword. It should be noted that there are more effective attacks than brute force decoding as discussed in Sect. 19.5.

As another example with $n = 2048$ and correcting $t = 80$ bit errors with a randomly chosen irreducible Goppa polynomial of degree 80, an example being $g(z) = 1 + z + z^3 + z^{17} + z^{80}$.

Setting the number of inserted bit errors s as a randomly chosen number from 40 to 80, the number of deleted bits correspondingly, is $2(t-s)$, ranging from 80 to 0 and the average codeword length is 2008 bits. There are 2.45×10^{144} different bit error combinations providing security, against naive brute force decoding, equivalent to a random key of length 482 bits. The message vector length is 1168 bits per codeword of which 7 bits may be assigned to indicate the number of deleted bits in the following codeword.

In a hybrid arrangement where the sender and recipient share secret information, additional bits in error may be deliberately added to the cryptogram using a secret key, the position vector to determine the positions of the additional error bits. The number of additional bits in error is randomly chosen between 0 and $n-1$. The recipient needs to know the number of additional bits in error (as well as the position vector), preferably with this information provided in a secure way. One method is for the message vector to contain, as part of the message, the number of additional bits in error in the next codeword that is the following codeword (not the current codeword). It is arranged that the first codeword has a known, fixed number of additional bits in error.

As each corrupted codeword contains more than t bits in error, it is theoretically impossible, even with the knowledge of the private key to recover the original codewords free from errors and to determine the unknown bits in the deleted bit positions. It should be noted that this arrangement defeats attacks based on information set decoding, which is discussed later. The system is depicted in Fig. 19.5.

This encryption arrangement is as shown in Fig. 19.1 except that the system accommodates additional errors added by *generate additional errors* shown in Fig. 19.5 using a random integer generator between 0 and $n-1$ generated by *generate random number of additional errors*. Any suitable random integer generator may be used. For example, the random integer generator design shown in Fig. 19.2 may be used with the number of shift register stages p now set equal to m, where $n = 2^m$.

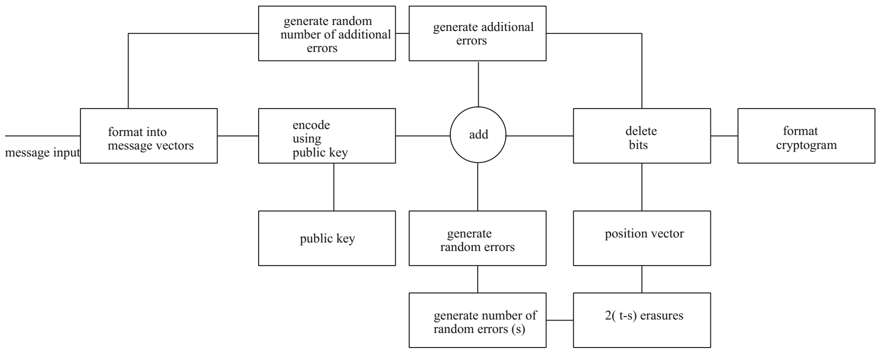

generate random number of additional errors
generate additional errors
message input
format into message vectors
encode using public key
add
delete bits
format cryptogram
public key
generate random errors
position vector
generate number of random errors (s)
2(t-s) erasures

Fig. 19.5 Public key encryption system with s random bit errors, $2(t-s)$ bit deletions and a random number of additional errors

Additional errors may be added in the same positions as random errors, as this provides for a simpler implementation or may take account of the positions of the random errors. However, there is no point in adding additional bit errors to bits which will be subsequently deleted.

As shown in Fig. 19.5, the number of additional errors is communicated to the recipient as part of the message vector in the preceding codeword with the information included with the message. This is carried out by *format into message vectors* shown in Fig. 19.5. In this case, usually 1 or 2 more message vectors in total will be required to convey the information regarding numbers of additional errors and the position vector (if this has not been already communicated to the recipient). Clearly, there are alternative arrangements to communicate the numbers of additional errors to the recipient such as using a previously agreed sequence of numbers or substituting a pseudorandom number generator for the truly random number generator (*generate random number of additional errors* shown in Fig. 19.5) with a known seed.

Using the previous example above, with the position vector:

$$\{19,\ 3,\ 27,\ 17,\ 8,\ 30,\ 11,\ 15,\ 2,\ 5,\ 19,\ \ldots,\ 25\}$$

The errored bits are in positions 7 and 23 (starting the position index from 0) and the deleted bits are in positions $\{19,\ 3,\ 27$ and $17\}$. The encoded codeword prior to corruption is:

$$\{0\,1\,1\,1\,1\,0\,0\,0\,1\,0\,0\,0\,0\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\}$$

The number of additional bits in error is randomly chosen to be 5, say. As the first 4 positions (index 0–3) in the position vector are to be deleted bits, starting from index 4, the bits in codeword positions $\{8,\ 30,\ 11,\ 15,$ and $2\}$ are inverted in addition to the errored bits in positions 7 and 23. The 32 bit corrupted codeword is produced: $\{0\,1\,0\,1\,1\,0\,0\,1\,0\,0\,0\,1\,0\,1\,0\,0\,1\,0\,0\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,1\,0\,0\}$.

The bits in positions {19, 3, 27 and 17} are deleted to produce the 28 bit corrupted codeword:

{0 1 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 1 1 0 1 1 0 1 0 0}

The additional bits in error are removed by the recipient of the cryptogram prior to errors and erasures correction as shown in Fig. 19.6. The number of additional bits in error in the following codewords is retrieved from the descrambled message vectors by *format message* shown in Fig. 19.6 and input to *number of additional errors* which outputs this number to *generate additional errors* which is the same as in Fig. 19.5. The position vector is stored in a buffer memory in *position vector* and outputs this to *generate additional errors*. Each additional error is corrected by the adder *add*, shown in Fig. 19.6, which adds, modulo 2, a 1 which is output from *generate additional errors* in the same position of each additional error. Retrieval of the message from this point follows correction of the errors and erasures, descrambling and formatting as described for Fig. 19.5.

Using the number of deleted bits and the position vector, 0's are inserted in the deleted bit positions to form the 32 bit corrupted codeword:

{0 1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 1 0 1 1 0 0 1 0 0}

After the addition of the output from *generate additional errors* the bits in positions {8, 30, 11, 15, and 2} are inverted, thereby correcting the 5 additional errors to form the less corrupted codeword:

{0 1 1 0 1 0 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0 0 1 1 0 1 1 0 0 1 1 0}

As in the first approach, this corrupted codeword is permuted, the syndromes calculated and the errors plus erasures corrected to retrieve the original message:

{0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1}

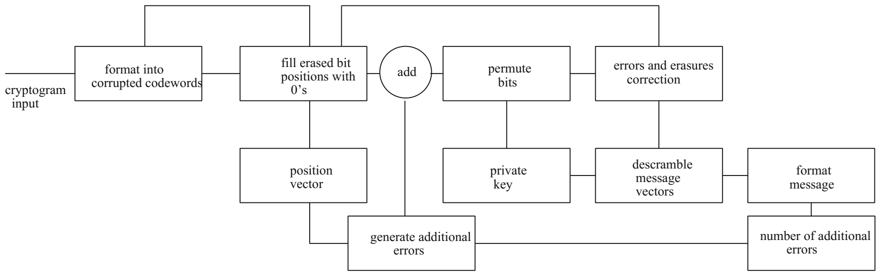

cryptogram input
format into corrupted codewords
fill erased bit positions with 0's
add
permute bits
errors and erasures correction
position vector
private key
descramble message vectors
format message
generate additional errors
number of additional errors

Fig. 19.6 Private key decryption system with s random bit errors, $2(t - s)$ bit deletions and a random number of additional errors

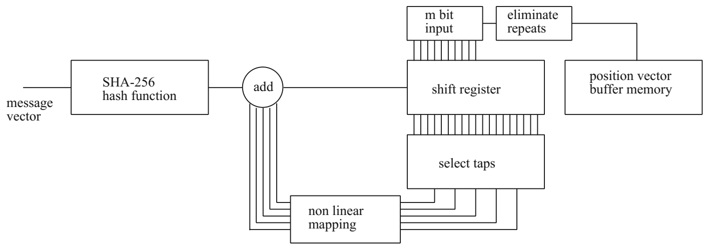


Fig. 19.7 Position vector generated by hash of message vector and nonlinear feedback shift register

In a further option, the position vector, instead of being a static vector, may be derived from a cryptographic hash of a previous message vector. Any standard cryptographic hash function may be used such as SHA-256 [11] or SHA-3 [12] as shown in Fig. 19.7. The message vector of length k bits is hashed using SHA-256 or SHA-3 to produce a binary hash vector of length 256 bits.

For example, the binary hash vector may be input to a nonlinear feedback shift register consisting of *shift register* having p stages, typically 64 stages with outputs determined by *select taps* enabling different scrambling keys to be used by selecting different outputs. The nonlinear feedback shift register arrangement to produce a position vector in *error positions buffer memory* is the same as that of Fig. 19.3 whose operation is described above.

As the hash vector is clocked into the nonlinear feedback shift register of Fig. 19.7, a derived position vector is stored in *error positions buffer memory*, and used for encrypting the message vector as described above. The current message vector is encrypted using a position vector derived from the hash of the previous message vector. As the recipient of the cryptogram has decrypted the previous message vector, the recipient of the cryptogram can use the same hash function and nonlinear feedback shift register to derive the position vector in order to decrypt the current corrupted codeword. There are a number of arrangements that may be used for the first codeword. For example, a static position vector, known only to the sender and recipient of the cryptogram could be used or alternatively a position vector derived from a fixed hash vector known only to the sender and recipient of the cryptogram or the hash of a fixed message known only to the sender and recipient of the cryptogram. A simpler arrangement may be used where the shift register has no feedback so that the position vector is derived directly from the hash vector. In this case the hash function needs to produce a hash vector $\geq n$, the length of the codeword.

As discussed earlier, the original McEliece system is vulnerable to chosen-plaintext attacks. If the same message is encrypted twice, the difference between the two cryptograms is just $2t$ bits or less, the sum of the two error patterns. This vulnerability is completely solved by encrypting or scrambling the plaintext prior to the McEliece system, using the error pattern as the key. To do this, the random error pattern needs to be generated first before the codeword is constructed by encoding with the scrambled generator matrix.

This scrambler which is derived from the error vector, for each message vector, may be implemented in a number of ways. The message vector may be scrambled by multiplying by a $k \times k$ non-singular matrix derived from the error vector.

Alternatively, the message vector may be scrambled by treating the message vector as a polynomial $m_1(x)$ of degree $k - 1$ and multiplying it by a circulant polynomial $p_1(x)$ modulo $1 + x^k$ which has an inverse [7]. The circulant polynomial $p_1(x)$ is derived from the error vector. Denoting the inverse of the circulant polynomial $p_1(x)$ as $q_1(x)$ then

$$p_1(x)q_1(x) = 1 \text{ modulo } 1 + x^k \tag{19.29}$$

Accordingly the scrambled message vector is $m_1(x)p_1(x)$ which is encoded into a codeword using the scrambled generator matrix. Each message vector is scrambled in a different way as the error patterns are random and different from corrupted codeword to corrupted codeword. The corrupted codewords form the cryptogram.

On decoding of each codeword, the corresponding error vector is obtained with retrieval of the scrambled message vector. Considering the above example, the circulant polynomial $p_1(x)$ is derived from the error vector and the inverse $q_1(x)$ is calculated using Euclid's method [7] from $p_1(x)$. The original message vector is obtained by multiplying the retrieved scrambled message vector $m_1(x)p_1(x)$ by $p_1(x)$ because

$$m_1(x)p_1(x)q_1(x) = m_1(x) \text{ modulo } 1 + x^k \tag{19.30}$$

Another method of scrambling each message vector using a scrambler derived from the error vector is to use two nonlinear feedback shift registers as shown in Fig. 19.8. The first operation is for the error vector, which is represented as a s-bit sequence is input to a modulo 2 adder *add* whose output is input to *shift register A* as shown in Fig. 19.8. The nonlinear feedback shift registers are the same as in Fig. 19.3 with operation as described above but *select taps* will usually have a different setting and *nonlinear mapping* also will usually have a different mapping, but this is not essential. After clocking the s-bit error sequence into the nonlinear feedback shift register, *shift register A* shown in Fig. 19.8 will essentially contain a random binary vector. This vector is used by *define taps* to define which outputs of *shift register B* are to be input to *nonlinear mapping B* whose outputs are added modulo 2 to the message vector input to form the input to *shift register B* shown in Fig. 19.8. The scrambling of the message vector is carried out by a nonlinear feedback shift register whose feedback connections are determined by a random binary vector derived from the error vector, the s-bit error sequence.

The corresponding descrambler is shown in Fig. 19.9. Following decoding of each corrupted codeword, having correcting the random errors and bit erasures, the scrambled message vector is obtained and the error vector is in the form of the s-bit error sequence. As in the scrambler, the s bit error sequence is input to a modulo 2 adder *add* whose output is input to *shift register A* as shown in Fig. 19.9. After clocking the s-bit error sequence into the nonlinear feedback shift register, *shift register A* shown in Fig. 19.9 will contain exactly the same binary vector as *shift register A* of Fig. 19.8. Consequently, exactly the same outputs of *shift register B* to

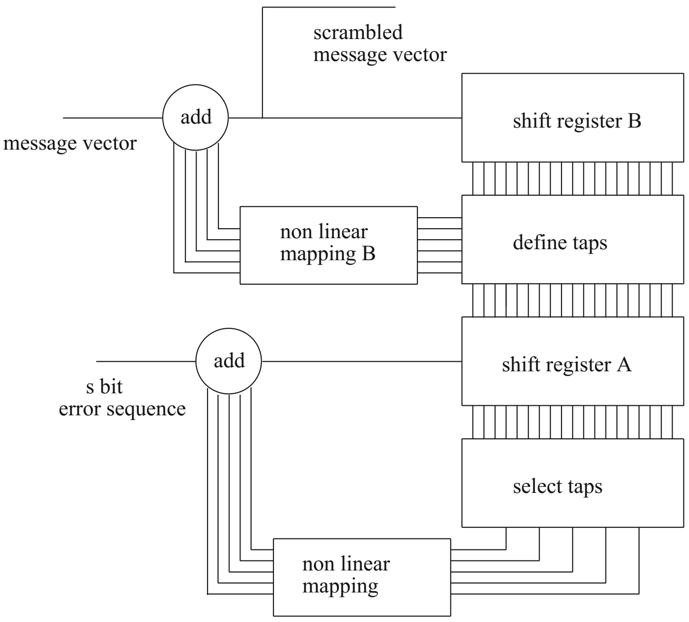


Fig. 19.8 Message vector scrambling by nonlinear feedback shift register with taps defined by s-bit error pattern

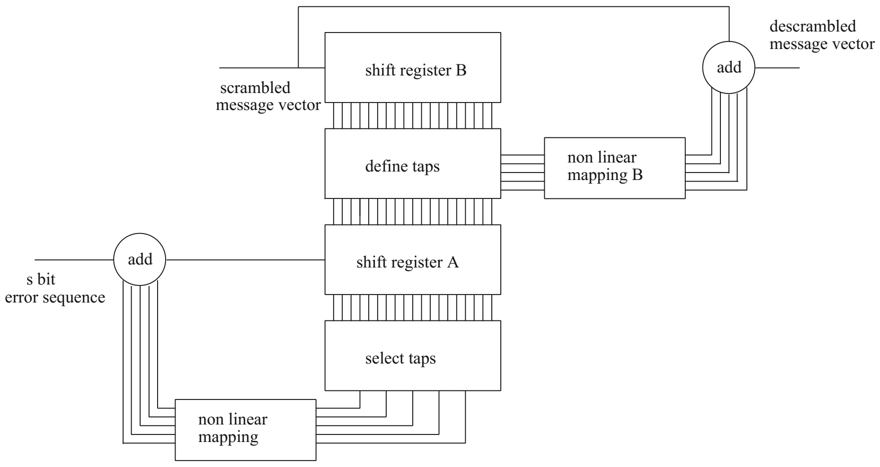


Fig. 19.9 Descrambling independently each scrambled message vector by nonlinear feedback shift register with taps defined by s-bit error pattern

be input to *non linear mapping B* will be defined by *define taps*. Moreover, comparing the input of *shift register B* of the scrambler Fig. 19.8 to the input of *shift register B* of the descrambler Fig. 19.9 it will be seen that the contents are identical and equal to the scrambled message vector.

Consequently, the same selected shift register outputs will be identical and with the same nonlinear mapping *nonlinear mapping B* the outputs of *nonlinear mapping B* in Fig. 19.9 will be identical to those that were the outputs of *nonlinear mapping B* in Fig. 19.8. The result of the addition of these outputs modulo 2 with the scrambled message vector is to produce the original message vector at the output of *add* in Fig. 19.9.

This is carried out for each scrambled message vector and associated error vector to recover the original message.

In some applications, a reduced size cryptogram is essential perhaps due to limited communications or storage capacity. For these applications, a simplification may be used in which the cryptogram consists of only one corrupted codeword containing random errors, the first codeword. The following codewords are corrupted by only deleting bits. The number of deleted bits is $2t$ bits per codeword using a position vector as described above.

For example, with $n = 1024$, and the Goppa code correcting $t = 60$ bit errors, there are $2t$ bits deleted per codeword so that apart from the first corrupted codeword, each corrupted codeword is only 904 bits long and conveys 624 message vector bits per corrupted codeword.

A similar approach is to hash the error vector of the first corrupted codeword and use this hash value as the key of a symmetric encryption system such as the Advanced Encryption Standard (AES) [10] and encrypt any following information this way. Effectively, this is AES encryption operating with a random session key since the error pattern is chosen randomly as in the classic, hybrid encryption system.

19.3 Reducing the Public Key Size

In the original McEliece system, the public key is the $k \times n$ generator matrix which can be quite large. For example with $n = 2048$ and $k = 1148$, the generator matrix needs to be represented by $1148 \times 2048 = 2.35 \times 10^6$ bits. Representing the generator matrix in reduced echelon form reduces the generator matrix to $k \times (n-k) = 1.14 \times 10^6$ bits. In the $n = 32$ example above, the generator matrix is the 12×32 matrix, $\mathbf{PSG_{(32,\,12,\,9)}}$, given by Eq. (19.26). Rows of this matrix may be added together using modulo 2 arithmetic so as to produce a matrix with k independent columns. This matrix is a reduced echelon matrix, possibly permuted to obtain k independent columns, and may be straightforwardly derived by using the Gauss–Jordan variable elimination procedure. With permutations, there are a large number of possible solutions which may be derived and candidate column positions may be selected, either initially in consecutive order to determine a solution, or optionally, selected in random order to arrive at other solutions.

Consider as an example the first option of selecting candidate column positions in consecutive order. For the $\mathbf{PSG_{(32,\,12,\,9)}}$ matrix (19.26), the following permuted reduced echelon generator matrix is produced:

$$\mathbf{PSGR}_{(32,12,9)} = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&1&0&1&0&1&0&1&0&0&0&1&1&1&1&1\\
0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&1&0&0&1&0&0&0&1&0&0&1&0&1&1&1&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&1&0&1&0&0&1&0&1&1&1&1&0&0&1&1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&1&1&0&1&0&0&1&0&1&0&0&0&0&1&0&1&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&1&0&1&0&0&1&1&1&1&0&1&0&0&1&0&0&0&1&0&0&1\\
0&0&0&0&0&1&0&0&0&0&0&1&0&0&0&0&0&1&0&0&1&1&0&1&1&1&1&1&1&1&0&1\\
0&0&0&0&0&0&1&0&0&0&1&1&0&0&0&0&0&0&1&0&1&1&0&0&1&0&0&1&1&0&0&1\\
0&0&0&0&0&0&0&1&0&0&1&1&0&1&0&0&1&1&1&1&1&1&0&0&0&1&1&0&0&1&1&1\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&1&0&0&1&1&0&1&1&0&0&1&1\\
0&0&0&0&0&0&0&0&0&1&1&0&0&0&0&1&1&1&0&0&1&0&1&1&1&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&1&0&1&0&1&0&1&1&1&1&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&1&0&1&0&1&0&1&1&1&1&1&0&1&0&1&1&0
\end{bmatrix} \tag{19.31}$$

The permutation defined by the following input and output bit position sequences is used to rearrange the columns of the permuted, reduced echelon generator matrix.

$$\begin{array}{cccccccccccccccccccccccccccccccc}
0&1&2&3&4&5&6&7&8&9&12&14&10&11&13&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31\\
0&1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31
\end{array} \tag{19.32}$$

This permutation produces a classical reduced echelon generator matrix [7], denoted as $\mathbf{Q}_{(32,12,9)}$:

$$\mathbf{Q}_{(32,12,9)} = \begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&1&1&0&1&0&1&0&1&0&0&0&1&1&1&1&1\\
0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&0&0&0&1&0&0&1&0&1&1&1&0\\
0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&1&0&0&1&0&1&1&1&1&0&0&1&1&0&0&0&0\\
0&0&0&1&0&0&0&0&0&0&0&0&1&1&1&0&1&0&1&0&0&0&0&1&0&1&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&1&1&1&1&0&1&0&0&1&0&0&0&1&0&0&1\\
0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&1&0&0&1&1&0&1&1&1&1&1&1&1&0&1\\
0&0&0&0&0&0&1&0&0&0&0&0&1&1&0&0&0&0&1&0&1&1&0&0&1&0&0&1&1&0&0&1\\
0&0&0&0&0&0&0&1&0&0&0&0&1&1&1&0&1&1&1&1&1&1&0&0&0&1&1&0&0&1&1&1\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&1&0&1&0&0&1&1&0&1&1&0&0&1&1\\
0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&1&1&1&0&0&1&0&1&1&1&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&1&1&0&1&0&1&0&1&1&1&1&0&0&0&1&1&1\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&1&0&1&0&1&1&1&1&1&0&1&0&1&1&0
\end{bmatrix} \tag{19.33}$$

Codewords generated by this matrix are from a systematic code [7] with the first 12 bits being information bits and the last 20 bits being parity bits. Correspondingly, the matrix above, $\mathbf{Q}_{(32,12,9)}$ consists of an identity matrix followed by a matrix denoted as $\mathbf{QT}_{(32,12,9)}$ which defines the parity bits part of the generator matrix. The transpose of this matrix is the parity check matrix of the code [7]. As shown in Fig. 19.10, the public key consists of the parity check matrix, less the identity submatrix, and a sequence of n numbers representing a permutation of the codeword bits after encoding. By permuting the codewords with the inverse permutation, the resulting permuted codewords will be identical to codewords produced by $\mathbf{PSG}_{(32,12,9)}$, the public key of the original McEliece public key system [8]. However, whilst the codewords are identical, the information bits will not correspond.

The permutation is defined by the following input and output bit position sequences.

$$\begin{array}{cccccccccccccccccccccccccccccccc}
0&1&2&3&4&5&6&7&8&9&12&14&10&11&13&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31\\
0&1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31
\end{array} \tag{19.34}$$

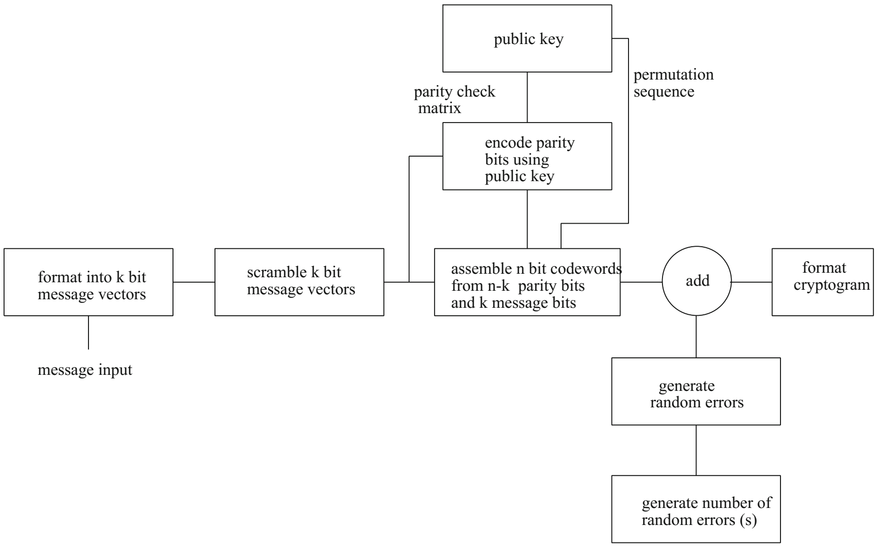

public key
parity check matrix
permutation sequence
encode parity bits using public key
format into k bit message vectors
scramble k bit message vectors
assemble n bit codewords from n-k parity bits and k message bits
add
format cryptogram
message input
generate random errors
generate number of random errors (s)

Fig. 19.10 Reduced size public key encryption system

As the output bit position sequence is just a sequence of bits in natural order, the permutation may be defined only by the input bit position sequence.

In this case, the public key consists of an n position permutation sequence and in this example the sequence chosen is:

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 12\ 14\ 10\ 11\ 13\ 15\ 16\ 17\ 18\ 19\ 20\ 21\ 22\ 23\ 24\ 25\ 26\ 27\ 28\ 29\ 30\ 31 \tag{19.35}$$

and the $k \times (n-k)$ matrix, $\mathbf{QT}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})}$, which in this example is the 12×20 matrix:

$$\mathbf{QT}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})} = \begin{bmatrix}
0&0&1&0&1&1&0&1&0&1&0&1&0&0&0&1&1&1&1&1\\
1&0&0&0&1&0&0&1&0&0&0&1&0&0&1&0&1&1&1&0\\
0&0&1&1&0&0&1&0&1&1&1&1&0&0&1&1&0&0&0&0\\
1&1&1&0&1&0&1&0&0&0&0&1&0&1&0&0&0&0&1&0\\
0&1&1&0&1&1&1&1&0&1&0&0&1&0&0&0&1&0&0&1\\
0&1&0&0&0&1&0&0&1&1&0&1&1&1&1&1&1&1&0&1\\
1&1&0&0&0&0&1&0&1&1&0&0&1&0&0&1&1&0&0&1\\
1&1&1&0&1&1&1&1&1&1&0&0&0&1&1&0&0&1&1&1\\
0&0&0&0&0&1&1&0&1&0&0&1&1&0&1&1&0&0&1&1\\
1&0&0&1&1&1&0&0&1&0&1&1&1&0&0&1&0&1&0&0\\
0&0&0&1&1&0&1&0&1&0&1&1&1&1&0&0&0&1&1&1\\
0&0&0&1&0&1&0&1&0&1&1&1&1&1&0&1&0&1&1&0
\end{bmatrix} \tag{19.36}$$

The public key of this system is much smaller than the public key of the original McEliece public key system, since as discussed below, there is no need to include the permutation sequence in the public key.

The message is split into message vectors of length 12 bits adding padding bits as necessary so that there is an integral number of message vectors. Each message vector, after scrambling, is encoded as a systematic codeword using $\mathbf{QT}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})}$, part of the public key. Each systematic codeword that is obtained is permuted using the permutation (19.35), the other part of the public key. The resulting codewords are identical to codewords generated using the generator matrix $\mathbf{PSG}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})}$ (19.26), the corresponding public key of the original McEliece public key system, but generated by different messages.

It should be noted that it is not necessary to use the exact permutation sequence that produces codewords identical to that produced by the original McEliece public key system for the same Goppa code and input parameters. As every permutation sequence has an inverse permutation sequence, any arbitrary permutation sequence, randomly generated or otherwise, may be used for the permutation sequence part of the public key. The permutation sequence that is the inverse of this arbitrary permutation sequence is absorbed into the permutation sequence used in decryption and forms part of the private key. The security of the system is enhanced by allowing arbitrary permutation sequences to be used and permutation sequences do not need to be part of the public key.

The purpose of scrambling each message vector using the fixed scrambler shown in Fig. 19.10 is to provide a one-to-one mapping between the 2^k possible message vectors and the 2^k scrambled message vectors such that the reverse mapping, which is provided by the descrambler, used in decryption, produces error multiplication if there are any errors present. For many messages, some information can be gained even if the message contains errors. The scrambler and corresponding descrambler prevents information being gained this way from the cryptogram itself or by means of some error guessing strategy for decryption by an attacker. The descrambler is designed to have the property that it produces descrambled message vectors that are likely to have a large Hamming distance between vectors for input scrambled message vectors which differ in a small number of bit positions.

There are a number of different techniques of realising such a scrambler and descrambler. One method is to use symmetric key encryption such as the Advanced Encryption Standard (AES) [10] with a fixed key.

An alternative means is provided by the scrambler arrangement shown in Fig. 19.11. The same arrangement may be used for descrambling but with different shift register taps and is shown in Fig. 19.12. Denoting each k bit message vector as a polynomial $m(x)$ of degree $k-1$:

$$m(x) = m_0 + m_1x + m_2x^2 + m_3x^3 \cdots + m_{k-1}x^{k-1} \tag{19.37}$$

and denoting the tap positions determined by *define taps* of Fig. 19.11 by $\mu(x)$ where

$$\mu(x) = \mu_0 + \mu_1x + \mu_2x^2 + \mu_3x^3 \cdots + \mu_{k-1}x^{k-1} \tag{19.38}$$

where the coefficients μ_0 through to μ_{k-1} have binary values of 1 or 0.

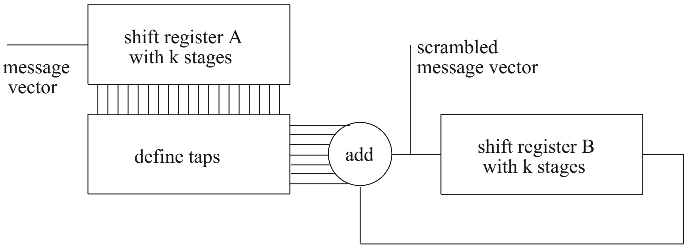

shift register A with k stages
message vector
scrambled message vector
define taps
add
shift register B with k stages

Fig. 19.11 Scrambler arrangement

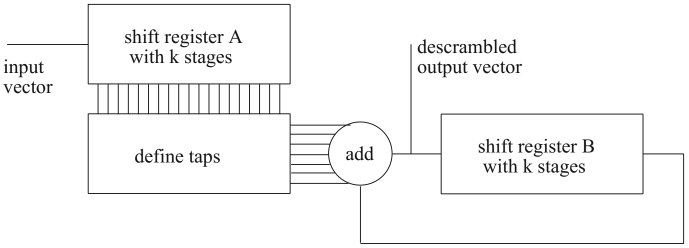

shift register A with k stages
input vector
descrambled output vector
define taps
add
shift register B with k stages

Fig. 19.12 Descrambler arrangement

The output of the scrambler, denoted by the polynomial, $scram(x)$, is the scrambled message vector given by the polynomial multiplication

$$scram(x) = m(x).\mu(x) \text{ modulo } (1 + x^k) \tag{19.39}$$

The scrambled message vector is produced by the arrangement shown in Fig. 19.11 after *shift register A with k stages* and *shift register B with k stages* have been clocked $2k$ times and is present at the input of *shift register B with k stages* whose last stage output is connected to the adder, *adder* input. The input of *shift register B with k stages* corresponds to the scrambled message vector for the next additional k clock cycles, with these bits defining the binary coefficients of $scram(x)$. The descrambler arrangement is shown in Fig. 19.12 and is an identical circuit to that of the scrambler but with different tap settings. The descrambler is used in decryption.

For $k = 12$ an example of a good scrambler polynomial, $\mu(x)$ is

$$\mu(x) = 1 + x + x^4 + x^5 + x^8 + x^9 + x^{11} \tag{19.40}$$

For brevity, the binary coefficients may be represented as a binary vector. In this example, $\mu(x)$ is represented as $\{1\,1\,0\,0\,1\,1\,0\,0\,1\,1\,0\,1\}$. This is a good scrambler polynomial because it has a relatively large number of taps (seven taps) and its inverse, the descrambler polynomial also has a relatively large number of taps (seven taps). The corresponding descrambler polynomial, $\theta(x)$ is

$$\theta(x) = 1 + x + x^3 + x^4 + x^7 + x^8 + x^{11} \tag{19.41}$$

which may be represented by the binary vector {1 1 0 1 1 0 0 1 1 0 0 1}. It is straightforward to verify that

$$\begin{aligned}\mu(x) \times \theta(x) = &\quad 1 + x^2 + x^3 + x^4 + x^5 + x^6 + x^8 + x^{10} \\ &+ x^{14} + x^{15} + x^{16} + x^{17} + x^{18} + x^{20} + x^{22} \\ = &\quad 1 \text{ modulo } (1 + x^k)\end{aligned} \tag{19.42}$$

and so

$$scram(x) \times \theta(x) = m(x) \text{ modulo } (1 + x^k) \tag{19.43}$$

As a simple example of a message, consider that the message consists of a single message vector with the information bit pattern {0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1} and so:

$$m(x) = x + x^3 + x^{11} \tag{19.44}$$

This is input to the scrambling arrangement shown in Fig. 19.11. The scrambled message output is $scram(x) = m(x) \times \mu(x)$ given by

$$\begin{aligned}scram(x) = &\ (1 + x + x^4 + x^5 + x^8 + x^9 + x^{11}).(x + x^3 + x^5 + x^{11}) \\ = &\quad x + x^2 + x^5 + x^6 + x^9 + x^{10} + x^{12} \\ &+ x^3 + x^4 + x^7 + x^8 + x^{11} + x^{12} + x^{14} \\ &+ x^{11} + x^{12} + x^{15} + x^{16} + x^{19} + x^{20} + x^{22} \\ &\quad \text{modulo } (1 + x^{12}) \\ = &\quad 1 + x + x^5 + x^6 + x^9\end{aligned} \tag{19.45}$$

and the scrambling arrangement shown in Fig. 19.11 produces the scrambled message bit pattern {1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0}.

Referring to Fig. 19.10, the next stage is to use the parity check matrix part of the *public key* to calculate the parity bits from the information bits. Starting with an all 0's vector, where the information bit pattern is a 1, the corresponding row from $\mathbf{QT}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})}$ (19.36) with the same position is added modulo 2 to the result so far to produce the parity bits which with the information bits will form the digital cryptogram plus added random errors after permuting the order of the bits. In this example, this codeword is generated from adding modulo 2, rows 1, 2, 6, 7 and 10 of $\mathbf{QT}_{(\mathbf{32},\,\mathbf{12},\,\mathbf{9})}$ to produce:

$$
\begin{array}{cccccccccccccccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\
+ & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
+ & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
+ & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + & + \\
1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel & \shortparallel \\
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\tag{19.46}
$$

The resulting systematic code, codeword is:

$$\{1\,1\,0\,0\,0\,1\,1\,0\,0\,1\,0\,0\,1\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\}$$

The last step in constructing the final codeword which will be used to construct the cryptogram is to apply an arbitrary preset permutation sequence. Referring to Fig. 19.10, the operation *assemble n bit codewords from n-k parity bits and k message bits* simply takes each codeword encoded as a systematic codeword and applies the preset permutation sequence.

In this example, the permutation sequence that is used is not chosen arbitrarily but is the permutation sequence that will produce the same codewords as the original McEliece public key system for the same Goppa code and input parameters. The permutation sequence is:

$$0\;1\;2\;3\;4\;5\;6\;7\;8\;9\;12\;14\;10\;11\;13\;15\;16\;17\;18\;19\;20\;21\;22\;23\;24\;25\;26\;27\;28\;29\;30\;31 \tag{19.47}$$

The notation is that the 10th bit should move to the 12th position, the 11th bit should move to the 14th position, the 12th bit should move to the 10th position, the 13th bit should move to the 11th position, the 14th bit should move to the 13th position and all other bits remain in their same positions.

Accordingly, the permuted codeword becomes:

$$\{1\,1\,0\,0\,0\,1\,1\,0\,0\,1\,1\,0\,0\,1\,0\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,0\,0\,0\,0\,0\,1\}$$

and this will be the input to the adder, *add* of Fig. 19.1.

The Goppa code used in this example can correct up to 4 errors, ($t = 4$), and a random number is chosen for the number of bits to be in error, (s) with $s \leq 4$.

A truly random source such as a thermal noise source as described above produces the most secure results, but a pseudorandom generator can be used instead, particularly if seeded from the time of day with fine time resolution such as 1mS. If the number of random errors chosen is too few, the security of the digital cryptogram will be compromised. Correspondingly, the minimum number of errors chosen is a design parameter depending upon the length of the Goppa code and t, the number of correctable errors. A suitable choice for the minimum number of errors chosen in practice lies between $\frac{t}{2}$ and t. If the cryptogram is likely to be subject to additional errors due to transmission over a noisy or interference prone medium such as wireless, or stored and read using an imperfect reader such as in barcode applications, then these additional errors can be corrected as well as the deliberately introduced errors provided the total number of errors is no more than t errors.

For such applications typically the number of deliberate errors is constrained to be between $\frac{t}{3}$ and $\frac{2t}{3}$.

For the example above, consider that the number of bit errors is 3 and these are randomly chosen to be in positions 4, 11 and 27 (starting the position index from 0). The bits in these positions in the codeword are inverted to produce the result

$$\{1\,1\,0\,0\,1\,1\,1\,0\,0\,1\,1\,1\,0\,1\,0\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,0\,0\,1\}$$

The dcryptogram is this corrupted codeword, which is transmitted or stored depending upon the application.

The intended recipient of this cryptogram retrieves the message in a series of steps. Figure 19.13 shows the system used for decryption. The retrieved cryptogram is formatted into corrupted codewords by *format into corrupted codewords* shown in Fig. 19.4. For the example above, the recipient first receives or otherwise retrieves the cryptogram, which may contain additional errors.

$$\{1\,1\,0\,0\,1\,1\,1\,0\,0\,1\,1\,1\,0\,1\,0\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,0\,1\,0\,0\,0\,1\}.$$

It is assumed in this example that no additional errors have occurred although with this particular example one additional error can be accommodated.

The private key contains the information of which Goppa code was used and a first permutation sequence which when applied to the retrieved, corrupted codewords which make up the cryptogram produces corrupted codewords of the Goppa code with the bits in the correct order. Usually, the private key also contains a second

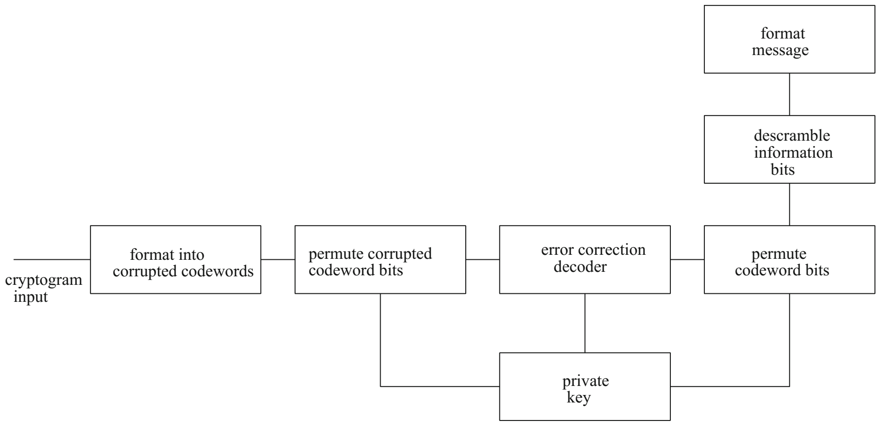

format message
descramble information bits
cryptogram input
format into corrupted codewords
permute corrupted codeword bits
error correction decoder
permute codeword bits
private key

Fig. 19.13 Private key decryption system

permutation sequence which when applied to the error-corrected Goppa codewords puts the scrambled information bits in natural order. Sometimes the private key also contains a third permutation sequence which when applied to the error vectors found in decoding the corrupted corrected Goppa codewords puts the bit errors in the same order that they were when inserted during encryption. All of this information is stored in *private key* in Fig. 19.13. Other information necessary to decrypt the cryptogram, such as the descrambler required may also be stored in the private key or be implicit.

There are two permutation sequences stored as part of the private key and the decryption arrangement is shown in Fig. 19.13. The corrupted codewords retrieved from the received or read cryptogram are permuted with a first permutation sequence which will put the bits in each corrupted codeword in the same order as the Goppa codewords. In this example, the first permutation sequence stored as part of the private key is:

$$24\ 11\ 3\ 23\ 2\ 17\ 20\ 8\ 13\ 30\ 31\ 14\ 15\ 22\ 7\ 1\ 9\ 6\ 21\ 4\ 10\ 5\ 28\ 26\ 19\ 16\ 25\ 0\ 27\ 12\ 18\ 29 \tag{19.48}$$

This defines the following permutation input and output sequences:

$$\begin{array}{cccccccccccccccccccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\
27 & 15 & 4 & 2 & 19 & 21 & 17 & 14 & 7 & 16 & 20 & 1 & 29 & 8 & 11 & 12 & 25 & 5 & 30 & 24 & 6 & 18 & 13 & 3 & 0 & 26 & 23 & 28 & 22 & 31 & 9 & 10
\end{array} \tag{19.49}$$

so that for example bit 23 becomes bit 3 after permutation and bit 30 becomes bit 9 after permutation. The resulting, permuted corrupted codeword is:

$$\{1\,1\,0\,0\,0\,1\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,1\,1\,1\,1\,1\,1\,0\,0\,0\,1\,1\,1\,1\,0\,1\,0\}$$

The permutation is carried out by *permute corrupted codeword bits* shown in Fig. 19.13 with the first permutation sequence input from *private key*.

Following the permutation of each corrupted codeword, the codeword bits are in the correct order to satisfy the parity check matrix, matrix (19.19) if there were no codeword bit errors. (In this case all of the syndrome values would be equal to 0). The next step is to treat each bit in each permuted corrupted codeword as a $GF(2^5)$ symbol with a 1 equal to α^0 and a 0 equal to 0 and use the parity check matrix, matrix (19.19), stored as part of *private key* to calculate the syndrome values for each row of the parity check matrix. The syndrome values produced in this example, are respectively α^{30}, α^{27}, α^4, and α^2. In Fig. 19.13 *error-correction decoder* calculates the syndromes as a first step in correcting the bit errors.

The bit errors are corrected using the syndrome values to produce an error free codeword from the Goppa code for each permuted corrupted codeword. There are many published algorithms for correcting bit errors for Goppa codes, but the most straightforward is to use a BCH decoder as described by Retter [13] because Berlekamp–Massey may then be used to solve the key equation. After decoding, the error free permuted codeword is obtained:

$$\{1\,0\,0\,0\,0\,1\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,1\,1\,1\,0\,1\,1\,0\,0\,0\,1\,1\,1\,0\,0\,1\,0\}$$

and the error pattern, defined as a 1 in each error position is

$$\{0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,0\}.$$

As shown in Fig. 19.13 *permute codeword bits* takes the output of *error-correction decoder* and applies the second permutation sequence stored as part of the private key to each corrected codeword.

Working through the example, consider that the following permutation input and output sequences is applied to the error free permuted codeword (the decoded codeword of the Goppa code).

$$\begin{array}{cccccccccccccccccccccccccccccccc}27&15&4&2&19&21&17&14&7&16&20&1&29&8&11&12&25&5&30&24&6&18&13&3&0&26&23&28&22&31&9&10\\0&1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31\end{array} \tag{19.50}$$

The result is that the scrambled message bits correspond to bit positions:

$$\{0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,12\,14\}$$

from the encryption procedure described above. The scrambled message bits may be repositioned in bit positions:

$$\{0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,10\,11\}$$

by absorbing the required additional permutations into a permutation sequence defined by the following permutation input and output sequences:

$$\begin{array}{cccccccccccccccccccccccccccccccc}27&15&4&2&19&21&17&14&7&16&29&11&20&8&1&12&25&5&30&24&6&18&13&3&0&26&23&28&22&31&9&10\\0&1&2&3&4&5&6&7&8&9&10&11&12&13&14&15&16&17&18&19&20&21&22&23&24&25&26&27&28&29&30&31\end{array} \tag{19.51}$$

The second permutation sequence which corresponds to this net permutation and which is stored as part of the private key, *private key* shown in Fig. 19.13 is:

$$24\ 14\ 3\ 23\ 2\ 17\ 20\ 8\ 13\ 30\ 31\ 11\ 15\ 22\ 7\ 1\ 9\ 6\ 21\ 4\ 12\ 5\ 28\ 26\ 19\ 16\ 25\ 0\ 27\ 10\ 18\ 29 \tag{19.52}$$

The second permutation sequence is applied by *permute codeword bits*. Since the encryption and decryption permutation sequences are all derived at the same time in forming the public key and private key from the chosen Goppa code, it is straightforward to calculate and store the net relevant permutation sequences as part of the private key.

Continuing working through the example, applying the second permutation sequence to the error free permuted codeword produces the output of *permute codeword bits*. The first 12 bits of the result will be the binary vector, {1 1 0 0 0 1 1 0 0 1 0 0} and it can be seen that this is identical to the scrambled message vector produced from the encryption operation. Represented as a polynomial the binary vector is $1 + x + x^5 + x^6 + x^9$.

As shown in Fig. 19.13, the next step is for the k information bits of each permuted error free codeword to be descrambled by *descramble information bits*. In this example, *descramble information bits* is carried out by the descrambler arrangement shown in Fig. 19.12 with *define taps* corresponding to polynomial $1 + x + x^3 + x^4 + x^7 + x^8 + x^{11}$.

The output of the descrambler in polynomial form is $(1 + x + x^5 + x^6 + x^9).(1 + x + x^3 + x^4 + x^7 + x^8 + x^{11})$ modulo $1 + x^{12}$. After polynomial multiplication, the result is $(x + x^3 + x^{11})$ corresponding to the message

{0 1 0 1 0 0 0 0 0 0 0 1}

It is apparent that this is the same as the original plaintext message prior to encryption.

With each cryptogram restricted to contain s errors, the cryptosystem as well as providing security, is able automatically to correct $t - s$ errors occurring in the communication of the cryptogram as shown in Fig. 19.14. It makes no difference to the decryption arrangement of Fig. 19.13, whether the bit errors were introduced deliberately during encryption or were introduced due to errors in transmitting the cryptogram. A correct message is output after decryption provided the total number of bit errors is less than or equal to t, the error-correcting capability of the Goppa code used to construct the public and private keys.

As an illustration, a (512, 287, 51) Goppa code of length 512 bits with message vectors of length 287 bits can correct up to 25 bit errors, ($t = 25$). With $s = 15$, 15 bit errors are added to each codeword during encryption. Up to 10 additional bit errors can occur in transmission of each corrupted codeword and the message will be still recovered correctly from the received cryptogram.

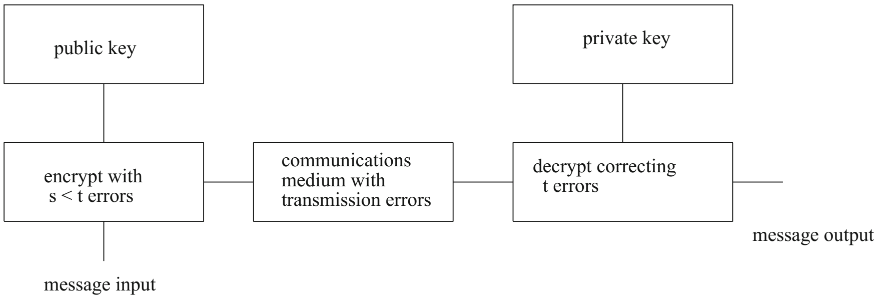


Fig. 19.14 Public key encryption system correcting communication transmission errors

The system will also correct errors in the reading of cryptograms stored in data media. As an example a medium to long range ISO 18000 6B RFID system operating in the 860–930 MHz with 2048 bits of user data can be read back from a tag. A (2048, 1388, 121) Goppa code of length 2048 bits with message vectors of length 1388 bits can correct 60 errors, ($t = 60$). With $s = 25$, 25 bit errors are added to the codeword during encryption and this is written to each passive tag as a cryptogram, stored in non-volatile memory. As well as providing confidentiality of the tag contents, up to 35 additional bit errors can be tolerated in reading each passive tag, thereby extending the operational range. The plaintext message, the encrypted tag payload information of 1388 bits will be recovered more reliably with each scanning of the tag.

19.4 Reducing the Cryptogram Length Without Loss of Security

In many applications a key encapsulation system is used. This is a hybrid encryption system in which a public key cryptosystem is used to send a random session key to the recipient and a symmetric key encryption system, such as AES [10] is used to encrypt the following data. Typically a session key is 256 bits long. To provide the same 256 bit security level a code length of 8192 bits needs to be used, with a code rate of 0.86. Security analysis of the McEliece system is provided in Sect. 19.5. The Goppa code is the (8192, 7048, 177) code. There are 7048 information bits available in each codeword, but only 256 bits are needed to communicate the session key. The code could be shortened in the traditional manner by truncating the generator matrix but this will leave less room to insert the t errors thereby reducing the security. The obvious question is can the codeword be shortened without reducing the security?

Niederreiter [9] solved this problem by transmitting only the $n-k$ bits of the syndrome calculated from the error pattern. Niederreiter originally proposed in his paper a system using Generalised Reed–Solomon codes but this scheme was subsequently broken with an attack by Sidelnikov and Shestakov [15]. However their attack fails if binary Goppa codes are used instead of Generalised Reed–Solomon codes and the Niederreiter system is now associated with the transmission of the $n - k$ parity bits as syndromes of the McEliece system.

It is unfortunate that only a tiny fraction of the 2^{n-k} syndromes correspond to correctable error patterns. For the (8192, 7048, 177) Goppa code, it turns out that there are 2^{697} correctable syndromes out of the total of 2^{1144} syndromes. The probability of an arbitrary syndrome being decoded is 2^{-447}, around 3×10^{-135}. This is the limitation of the Niederreiter system. The plaintext message has to be mapped into an error pattern consisting of t bits uniformly distributed over n bits. Any deterministic method of doing this will be vulnerable to a chosen-plaintext attack. Of course the Niederreiter system can be used to send random messages, first generated as a random error pattern, as in a random session key. However, additional information really needs to be sent as well, such as a MAC, timestamp, sender ID, digital signature or other supplementary information.

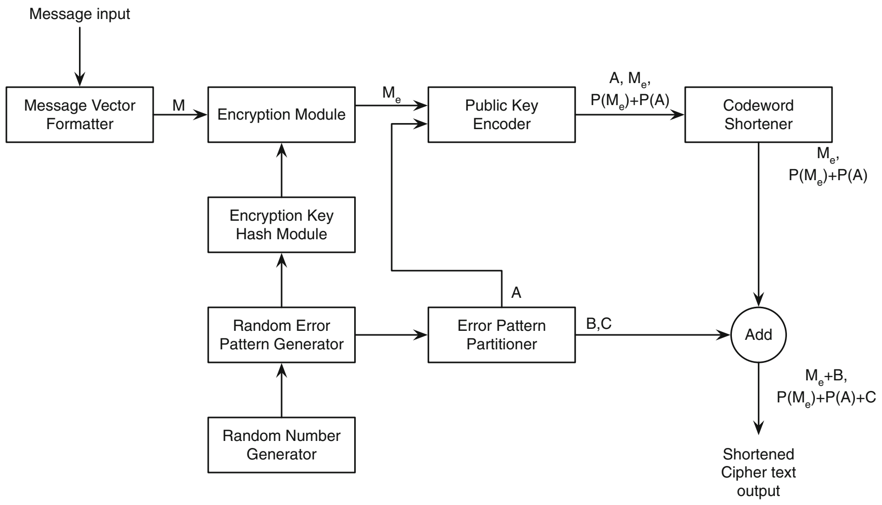


Fig. 19.15 Public key encryption system with shortened codewords

One solution is to use the system shown in Fig. 19.15. The plaintext message, M, consisting of a 256 bit random session key concatenated with 512 bits of supplementary information such as a MAC, time stamp and sender ID is encrypted in the encryption module with a key that is a cryptographic hash, such as SHA-3 [12], of the error pattern. The encrypted message is M_e. The error pattern consists of t bit errors randomly distributed over n bits. This bit pattern is partitioned into three parts, A, B and C as shown in Fig. 19.16. Using the (8192, 7048, 177) Goppa code, part A covers the first 6280 bits, part B covers the next 768 bits and part C covers the last 1144 bits, the parity bits.

The public key encoder consists of the public key generator matrix in reduced echelon form which is used to encode information bits consisting of part A, concatenated with M_e as shown in Fig. 19.15. After encoding, the $n - k$ parity bits of the codeword are $P(A) + P(M_e)$. The codeword is then shortened by removing the first 6280 bits of the codeword. The error pattern parts B and C are added to the shortened codeword of length 1912 bits to form the ciphertext of length 1912 bits. The format of the various parts of the error pattern, the hash derivation, encryption and codeword are shown in Fig. 19.16.

The principle that this system uses, is that the syndrome of any codeword is zero and that the cryptosystem is linear. The codeword resulting from the encoding of A is $\{A\ 0\ \ldots\ 0\ P(A)\}$, where $P(A)$ are the parity bits. The sum of syndromes from sections, of this codeword must be zero. Hence:

$$\text{Syndrome}(A) + \ 0\ \ldots\ 0\ + P(A) = 0$$

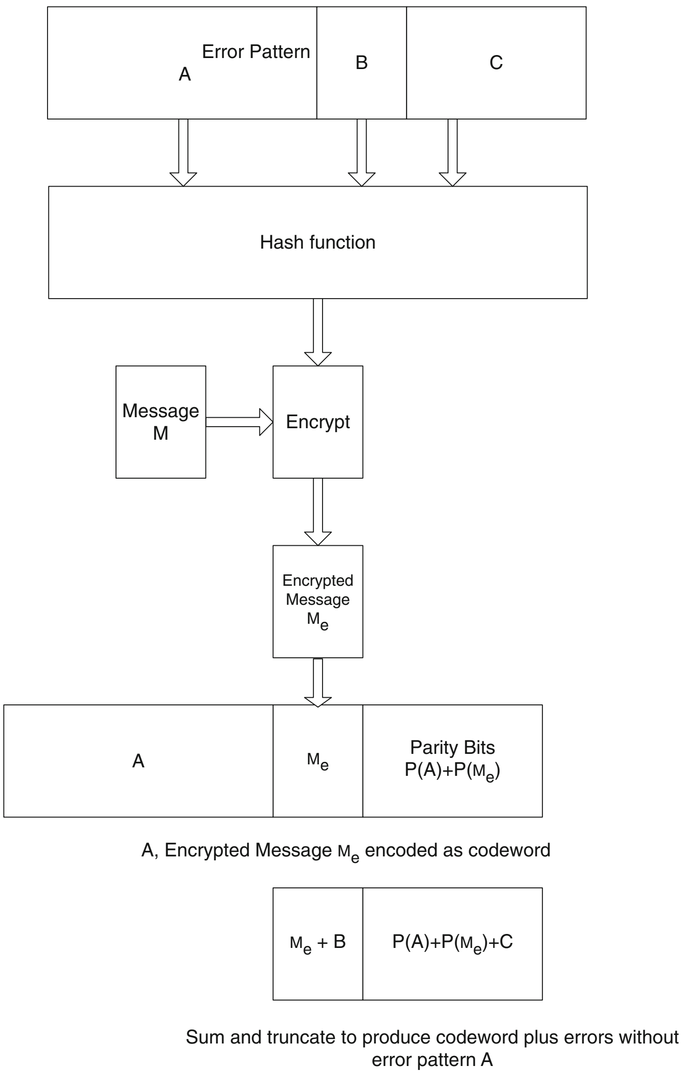

Error Pattern
A
B
C
Hash function
Message
M
Encrypt
Encrypted
Message
M_e
A
M_e
Parity Bits
P(A)+P(M_e)
A, Encrypted Message M_e encoded as codeword
M_e + B
P(A)+P(M_e)+C
Sum and truncate to produce codeword plus errors without error pattern A

Fig. 19.16 Format of the shortened codeword and error pattern

As the base field is 2,

$$\text{Syndrome}(A) = P(A)$$

Consequently, by including $P(A)$ instead of the error bits, part A in the ciphertext results in the same syndrome being calculated in the decoder, namely $P(A) + P(B) +$

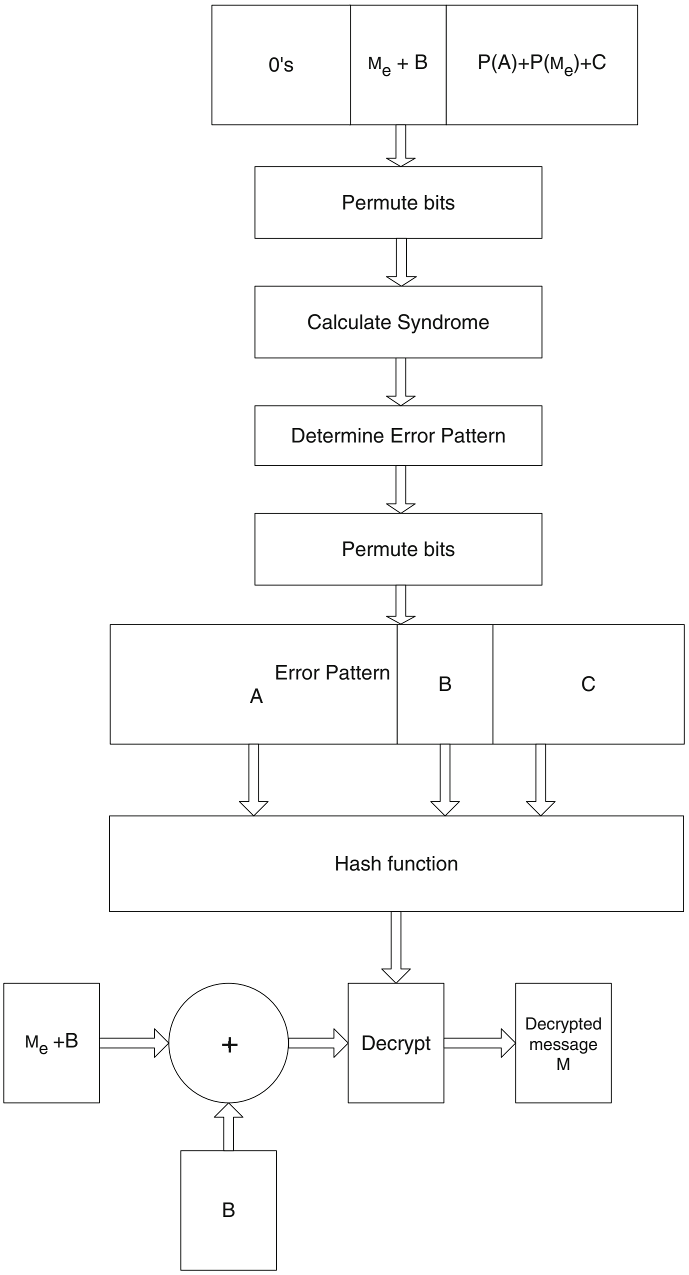


Fig. 19.17 Decryption method for the shortened ciphertext

$P(C)$. Removing error pattern, part A shortens the ciphertext whilst including $P(A)$ instead requires no additional bits in the ciphertext. Since it is necessary to derive the complete error pattern of length n bits in order to decrypt the ciphertext, there is no loss of security from shortening the ciphertext.

The method used to decrypt the ciphertext by using the private key is shown in Fig. 19.17. The received ciphertext is padded with leading zeros to restore its length to 8192 bits. This is then permuted to be in the same order as the Goppa code and the parity check matrix of the Goppa code is used to calculate the syndrome. A Goppa code error-correcting decoder is then used to find the permuted error pattern A, B and C from this syndrome. The most straightforward error-correcting decoder to use is based on Retter's decoding method [13]. This involves calculating a syndrome having $2(n-k)$ parity bits from the parity check matrix of $g^2(z)$ where $g(z)$ is the Goppa polynomial of the code, then using the Berlekamp–Massey method to solve the key equation as in a standard BCH decoder to find the error bit positions. It is because the codewords are binary codewords and the base field is 2, that the Goppa code codewords satisfy the parity checks of $g^2(z)$ as well as the parity checks of $g(z)$, since $1^2 = 1$.

As shown in Fig. 19.17, once the error pattern is determined, it is inverse permuted to produce A, B and C which is hashed to produce the decryption key needed to decrypt M_e back into the plaintext message M. Part B of the derived error pattern is added to the $M_e + B$ contained in the received ciphertext to produce M_e as shown in Fig. 19.17.

19.5 Security of the Cryptosystem

If we consider the parameters that Professor McEliece originally chose, a code of length 1024 bits correcting 50 errors and 524 information bits, then a brute force attack may be based on guessing the error pattern, adding this to the cryptogram and checking if the result is a valid codeword. Checking if the result is a codeword is easy. By using elementary matrix operations on the public key, the generator matrix, we can turn it into a reduced echelon matrix whose transpose is the parity check matrix. We simply use this parity check matrix to calculate the syndrome of the n bit input vector. If the syndrome is equal to zero then the input vector is a codeword.

The maximum number of syndromes that need to be calculated is equal to the number of different error patterns which is $\binom{n}{t} = \binom{1024}{50} = 3.19^{85} \approx 2^{284}$. This may be described as being equivalent to a symmetric key encryption system with a key length of 284 bits.

However, there are much more efficient ways of determining the error pattern in the cryptogram. An attack called information set decoding [2] as described by Professor McEliece in his original paper [8], may be used. For any (n, k, d) code, k columns of the generator matrix may be randomly selected and using Gauss–Jordan elimination of the rows, there is a probability that a permuted, reduced echelon generator matrix will be obtained which generates the same codeword as the original code. The $k \times k$ sub-matrix resulting from the k selected columns needs to be full rank and the probability of this depends on the particular code. For Goppa codes the probability turns out to be the same as the probability of a randomly chosen $k \times k$ binary matrix being full rank. This probability is 0.2887 as described below in Sect. 19.5.1.

Given a cryptogram containing t errors an attacker can select k bits randomly, construct the corresponding permuted, reduced echelon generator matrix with a chance of 0.29. The attacker then uses the matrix to generate a codeword and finds the Hamming distance between this codeword and the cryptogram. If the Hamming distance is exactly t then the cryptogram has been cracked.

For this to happen all of the k selected bits from the cryptogram need to be error free. The probability of this is:

$$\prod_{i=0}^{k-1} \frac{n-t-i}{n-i} = \frac{(n-t)!(n-k)!}{(n-t-k)!n!}$$

Including the chance of 0.29 that the selected matrix has rank k, the average number of selections of k bits from the cryptogram before the cryptogram is cracked, N_{ck} is given by

$$N_{ck} = \frac{(n-t-k)!n!}{0.29(n-t)!(n-k)!} \tag{19.53}$$

For the original code parameters (1024, 524, 101), $N_{ck} = 4.78 \times 10^{16} \approx 2^{55}$.

This is equivalent to a symmetric key encryption system with a key length of 55 bits, a lot less than 284 bits, the base 2 logarithm of the number of error combinations.

Using a longer code offers much more security. For example using code parameters (2048, 1300, 137), $N_{ck} = 1.45 \times 10^{31} \approx 2^{103}$, equivalent to a symmetric key length of 103 bits.

For code parameters (8192, 5124, 473), with a Goppa code which corrects 236 errors, it turns out that $N_{ck} = 5.60 \times 10^{103} \approx 2^{344}$, equivalent to a symmetric key length of 344 bits.

The success of this attack depends upon the code rate. The effect of the code rate, R and the security as expressed in the equivalent symmetric key length in bits is shown in Fig. 19.18 for a code length of 2048 bits. The code rate, R that maximises N_{ck} for a given n is tabulated in Table 19.3, together with t the number of correctable errors and the equivalent symmetric key length in bits.

19.5.1 Probability of a $k \times k$ Random Matrix Being Full Rank

The probability of a randomly chosen $k \times k$ binary matrix being full rank is a classical problem related to the erasure correcting capability of random binary codes [4, 5].

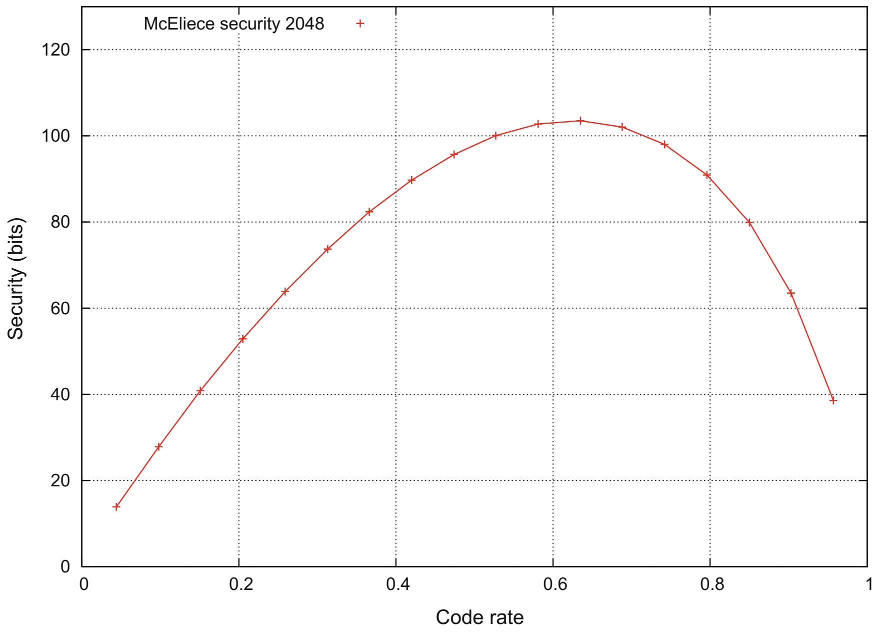


Fig. 19.18 Effect of code rate on security for a code length of 2048 bits

Table 19.3 Optimum code rate giving maximum security as a function of code length

n	R	t	Security (bits)
512	0.6309	21	33.0
1024	0.6289	38	57.9
2048	0.6294	69	103.5
4096	0.6279	127	187.9
8192	0.6287	234	344.6
16384	0.6292	434	637.4

For the binary case it is straightforward to derive the probability, P_k of a $k \times k$ randomly chosen matrix being full rank by considering the process of Gauss–Jordan elimination. Starting with the first column of the matrix, the probability of finding a 1 in at least one of the rows is $(1 - \frac{1}{2})^k$.

Selecting one of these non-zero bit rows, the bit in the second column will be arbitrary and considering the first two bits there are 2^1 linear combinations. As there are 2^2 combinations of two bits, the chances of not finding an independent 2-bit combination in the remaining $k - 1$ rows are $\frac{1}{2^{k-1}}$. Assuming an independent row is found, we next consider the third column and the first three bits. There are 2^2 linear combinations of 3 bits from the two previously found independent rows and there are a total possible 2^3 combinations of 3 bits. The probability of not finding an independent 3-bit pattern in any of the remaining $k - 2$ rows is $(\frac{2^2}{2^3})^{k-2} = \frac{1}{2^{k-2}}$.

Table 19.4 Probability of a random binary $k \times k$ matrix having full rank

k	P_k
5	0.298004
10	0.289070
15	0.288797
20	0.288788
50	0.288788

Proceeding in this way to k rows, it is apparent that P_k is given by

$$P_k = \prod_{i=0}^{k-1} 1 - \left(1 - \frac{1}{2}\right)^{k-i} = \prod_{i=0}^{k-1} 1 - \frac{1}{2^{k-i}} \tag{19.54}$$

The probability of P_k as a function of k is tabulated in Table 19.4. The asymptote of 0.288788 is reached for k exceeding 18.

19.5.2 Practical Attack Algorithms

Practical attack algorithms of course need to factor in the processing cost of Gauss–Jordan elimination compared to the problem of constructing different generator matrices. A completely different set of k coordinates does not need to be selected each time to generate a different generator matrix as discussed in [2]. Also, even if the k selected columns of the generator matrix do not have full rank, usually discarding and adding one or two columns will produce a full rank matrix. It can be shown that on average only 1.6 additional columns are necessary to achieve full rank. Canteaut and Chabaud [3] showed that by including the cryptogram as an additional row in the generator matrix of the $(n, k, 2t + 1)$ code, a code is produced with parameters $(n, k+1, t)$ for which there is only a single codeword with weight t, the original error pattern in the cryptogram. In this case algorithms for finding low-weight codewords may be deployed to break the cryptogram. However these low-weight codeword search algorithms are all very similar to the original algorithm aimed at searching for a codeword of the $(n, k, 2t+1)$ code with Hamming distance t from the cryptogram. The conclusions of the literature are that information set decoding is an efficient method of attacking the McEliece system but that from a practical viewpoint the system is unbreakable provided the code is long enough. Bernstein et al. [2] give recommended code lengths and their corresponding security, providing similar results to that of Table 19.3.

The standard McEliece system is vulnerable to chosen-plaintext attacks. The encoder is the public key, usually publically available, and the attacker can simply guess the plaintext, construct the corresponding ciphertext and compare this to the target ciphertext. In addition, if the same plaintext message is encrypted twice the sum of the two ciphertexts is a n bit vector of $2t$ bits or less.

The standard McEliece system is also vulnerable to chosen-ciphertext attack. Assuming a decryption oracle is available, the attacker inverts two bits randomly in the ciphertext and sends the result to the decryption oracle. With probability $\frac{t(n-t)}{n(n-1)}$, a different ciphertext will be produced containing exactly t errors and the decryption oracle will output the plaintext, breaking the system.

Encrypting the plaintext using a key derived from the error pattern, as described above, defeats all of these attacks.

19.6 Applications

Public key encryption is attractive in a wide range of different applications, particularly those involving communications because the public keys may be exchanged initially using clear text messages followed by information encrypted using the public keys. The private keys remain private because they do not need to be communicated and the public keys are of no help to an eavesdropper.

An example of an application for the iPhone and iPad using the McEliece public key encryption system is the S2S app pictured in Fig. 19.19. In this app, files are encrypted with users' public keys and stored in the cloud so that they may be shared. Sharing is by means of links that index the encrypted files on the cloud and each user uses their private key to decrypt the shared files.

If the same type of application was implemented using symmetric key encryption, it would be necessary for users to share passwords with all of the associated risks that entails. Using public key encryption avoids these risks. Another application example is the secure Instant Messaging (IM) system, PQChat for the iPhone, iPad and Android devices. There is an option button which shows messages in their received encrypted format, as shown in Fig. 19.20. The application is called PQChat and the name stands for Post-Quantum Chat as the McEliece cryptosystem is relatively immune to attack by a quantum computer, unlike the public key encryption systems in common use today, such as Rivest Shamir Adleman, (RSA) and Elgamal.

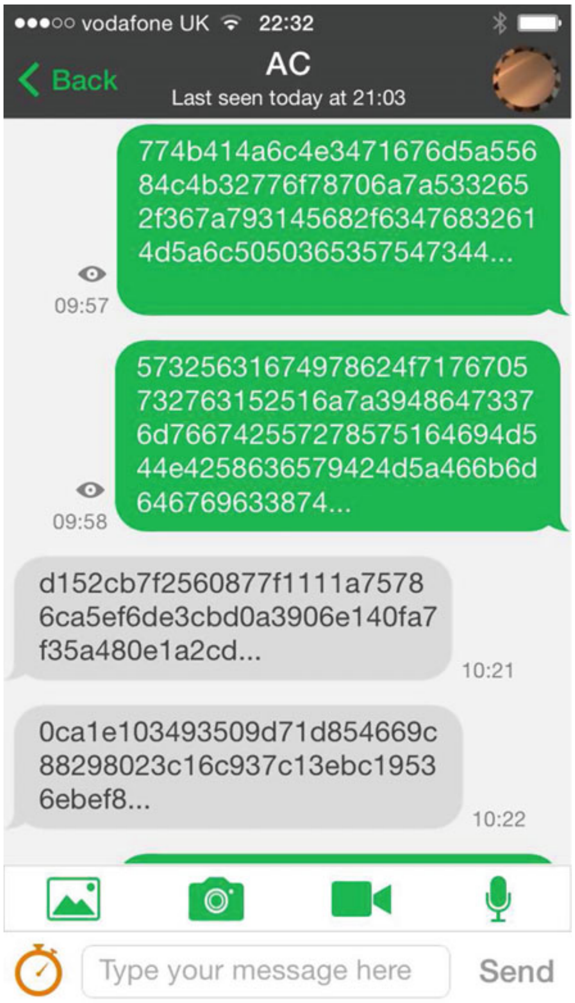
As with other public key methods, the system may be used for mutual authentication. Party X sends a randomly chosen nonce x_1, together with a timestamp to Party Y using Party Y's public key. Party Y returns a randomly chosen nonce y_1, timestamp and $hash(hash(x_1, y_1))$ to Party X using Party X's public key. Party X replies with an encrypted timestamp and acknowlegement, using symmetric key cryptography with encryption key $hash(x_1, y_1)$, the preimage of $hash(hash(x_1, y_1))$, to Party Y. The session key $hash(x_1, y_1)$ is used for further exchanges of information, for the duration of the session. A cryptographic hash function is used such as SHA-3 [12] which also has good, second preimage resistance.

It is assumed that the private keys have been kept secret and the association of IDs with public keys has been independently verified. In this case, Party X knows Party Y holds the private key of Y and is the only one able to learn x_1. Party Y knows Party

Fig. 19.19 S2S application for sharing encrypted files

X holds the private key of X and is the only one able to learn y_1. Consequently Party X and Party Y are the only ones with knowledge of x_1 and y_1. Using the preimage of $hash(hash(x_1, y_1))$ as the session key provides added assurance as both x_1 and y_1 need to be known in order to generate the key, $hash(x_1, y_1)$. The timestamps prevent replay attacks being used.

Fig. 19.20 PQChat secure instant messaging app featuring McEliece cryptosystem (with view as received, enabled)


●●●○○ vodafone UK 22:32

Back AC

Last seen today at 21:03

774b414a6c4e3471676d5a556
84c4b32776f78706a7a533265
2f367a793145682f6347683261
4d5a6c5050365357547344...

09:57

57325631674978624f7176705
732763152516a7a3948647337
6d766742557278575164694d5
44e4258636579424d5a466b6d
646769633874...

09:58

d152cb7f2560877f1111a7578
6ca5ef6de3cbd0a3906e140fa7
f35a480e1a2cd...

10:21

0ca1e103493509d71d854669c
88298023c16c937c13ebc1953
6ebef8...

10:22

Type your message here

Send

19.7 Summary

A completely novel type of public key cryptosystem was invented by Professor Robert McEliece in 1978 and this is based on error-correcting codes using Goppa codes. Other well established public key cryptosystems are based on the difficulty of determining logarithms in finite fields which, in theory, can be broken by quantum computers. Despite numerous attempts by the crypto community, the McEliece system remains unbroken to this day and is one of the few systems predicted to survive attacks by powerful computers in the future. In this chapter, some variations to the McEliece system have been described including a method which destroys the deterministic link between plaintext messages and ciphertexts, thereby providing semantic security. Consequently, this method nullifies the chosen-plaintext attack, of which the classic McEliece is vulnerable. It is shown that the public key size can be reduced and by encrypting the plaintext with a key derived from the ciphertext random error pattern, the security of the system is improved since an attacker has to determine the exact same error pattern used to produce the ciphertext. This defeats a chosen-ciphertext attack in which two random bits of the ciphertext are inverted. The standard McEliece system is vulnerable to this attack. The security of the McEliece system has been analysed and a shortened ciphertext system has been proposed which does not suffer from any consequent loss of security due to shortening. This is important because to achieve 256 bits of security, the security analysis has shown that the system needs to be based on Goppa codes of length 8192 bits. Key encapsulation and short plaintext applications need short ciphertexts in order to be efficient. It is shown that the ciphertext may be shortened to 1912 bits, provide 256 bits of security and an information payload of 768 bits. Some examples of interesting applications that have been implemented on a smartphone in commercial products, such as a secure messaging app and secure cloud storage app, have been described in this chapter.

References

1. Berlekamp, E.R.: Algebraic Coding Theory, Revised edn. Aegean Park Press, Laguna Hills (1984). ISBN 0 894 12063 8
2. Bernstein, D., Lange, T., Peters, C.: Attacking and defending the McEliece cryptosystem. In: Buchmann, J., Ding, J. (eds.) PQCrypto, pp. 31–46 (2008)
3. Canteaut, A., Chabaud, F.: A new algorithm for finding minimum weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. IEEE Trans. Inf. Theory **44**(1), 367–378 (1998)
4. Cooper, C.: On the distribution of rank of a random matrix over a finite field. Random Struct. Algorithms **17**, 197–212 (2000)
5. Dumer, I., Farrell, P.: Erasure correction performance of linear block codes. In: Cohen, G., Litsyn, S., Lobstein, A., Zemor, G. (eds.) Lecture Notes in Computer Science, vol. 781, pp. 316–326. Springer, Berlin (1993)
6. Goppa, V.D.: A new class of linear error-correcting codes. Probl. Inf. Transm. **6**, 24–30 (1970)
7. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland, Amsterdam (1977)
8. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. DSN Prog. Rep. **42–44**, 114–116 (1978)
9. Niederreiter, H.: Knapsack-type cryptosystems and algebraic coding theory. Probl. Control Inf. Theory **15**, 159–166 (1986)
10. Publications FIPS. Advanced Encryption Standard (AES). FIPS PUB 197 (2001)
11. Publications FIPS. Secure Hash Standard (SHS). FIPS PUB 180-3 (2008)
12. Publications FIPS. SHA-3 Standard Permutation Based Hash and Extendable Output Functions. FIPS PUB 202 (2015)
13. Retter, C.T.: Decoding Goppa codes with a BCH decoder. IEEE Trans. Inf. Theory IT **21**, 112–112 (1975)
14. Riek, J., McFarland, G.: Error correcting public key cryptographic method and program. US Patent 5054066 (1988)

15. Sidelnikov, V., Shestakov, S.: On insecurity of cryptosystems based on generalized Reed-Solomon codes. Discrete Math. Appl. **2**(4), 439–444 (1992)
16. Sugiyama, Y., Kasahara, M., Namekawa, T.: An erasures-and-errors decoding algorithm for Goppa codes. IEEE Trans. Inf. Theory IT **22**, 238–241 (1976)