Implementation of McEliece using quasi-dyadic Goppa codes

Part I

Statement

I hereby declare that the work presented in this thesis is my own work and that to the best of my knowledge it is original except where indicated by references to other authors.

Gerhard Hoffmann, Darmstadt, April 25, 2011

Contents

Part II

Acknowledgements

First of all, I wish to thank my supervisor Dr. Pierre-Louis Cayrel, and Prof. Paulo S. L. M. Barreto for their patience and support during the thesis. Despite not being a member of CASED, especially Mr. Barreto has been very kind and responsive to my many, many questions via private Email communications. Without their guidance, I do not think I would have made it.

Secondly, I would also like to thank my fellow members at CASED for their support and helpful discussions, specially Robert Niebuhr, Richard Lindner and Michael Schneider.

Part III
Introduction

In 1948 Claude Shannon founded communication and coding theory [31] and defined three different kinds of coding mechanisms: coding for encryption purposes, source coding and error control coding.

Source coding means compressing messages before transmission, such that unneeded redundancy is removed and the communication system has to transmit less data.

Error control coding is about the opposite: the sender adds information to messages. The redundant information shall enable the receiver to decode the message, even when some transmission errors have occurred.

The codes used for error control coding are typically so called linear codes, which are just vector spaces over some finite field. At first glance, the rich algebraic structure of these codes seems to prevent an application in cryptography, but in his pioneering paper [21], Robert J. McEliece combined the algebraic approach of linear error-correcting codes and public-key cryptosystems.

The basic idea is to hide the algebraic structure of the linear code and to use its secret error-correcting capabilities, i.e. its decoding mechanism, as trapdoor for the public-key cryptosytem. The hidden structure makes it intractable for an outsider to find such a decoder [6]. Thus, any errors injected into a message by the sender can be corrected only by the receiver, who knows the underlying linear code and therefore knows how to decode in the presence of errors.

Using proper parameter settings, no praticable attack is known against McEliece's original scheme [21], although it has been introduced more than 30 years ago. Despite its computational efficiency it never attracted so much attention like RSA or Diffie-Hellman, mainly due to its relatively big size for the public keys, typically some 100kB for reasonable parameter settings.

Things changed in 1995, when Peter W. Shor [32] showed that prime number factorization would be practical on the quantum computer. On the other hand, there is no quantum algorithm known for the McEliece scheme in order to attack it, which makes it a promising candidate for post-quantum cryptography.

One of the approaches to reduce the scheme's public key size via binary quasi-dyadic Goppa codes will be presented in this thesis [23]. The basic idea is to use separable binary Goppa codes, which can be represented by a highly structured generator matrix without revealing too much information of the underlying Goppa code.

The thesis is intended as exposition of all the necessary facts concerning binary quasi-dyadic Goppa codes and how to use these codes for an implementation of the McEliece scheme. It will also shortly address the issue of a recent structural attack against McEliece based on these codes [12].

The goal of the implementation is not to be as efficient as possible, but to make the main ideas accessible for the reader at source code level. Writing all the necessary details would have gone beyond the scope of a bachelor thesis. Parts of the source code are therefore based on HyMES [29, 30], a recent library[1] for a hybrid McEliece scheme using irreducible binary Goppa codes. It will be always clearly visible when source code stems from HyMES.

The thesis is written in literate programming style using the CWEB system [17]. In traditional programming the source code comes before the documentation. In literate programming the documentation comes first, fleshed out with code later. Each section of this thesis is started with some background information, followed by a real implementation in C.

[1]Released under the GPL license.

Part IV

The Algebraic Setup

1 Finite fields

Linear codes are subspaces of $\mathbb{F}_q^n$, where $\mathbb{F}_q$ is the finite field with order q. Hence, codewords are vectors of length n over the alphabet $\mathbb{F}_q$. We summarize some important facts of finite fields. Further details can be found in the Appendix or in [1, 14, 19].

Let $\mathbb{F}$ denote a finite field. Because it is finite, its characteristic must be finite. By definition, a field does not have zero-divisors. Therefore, the characteristic must be a prime number p. The canonical example of a finite field of order p is

$$\mathbb{F}_p := \{0, \ldots, p-1\},$$

where addition and multiplication is taken modulo p. Each finite field $\mathbb{F}$ with characteristic p contains an isomorphic copy $\mathbb{P}$ of $\mathbb{F}_p$ as subfield. Therefore $\mathbb{F}_p$ is called the *prime field* of characteristic p. A finite field $\mathbb{F}$ with characteristic p is canonically a finite-dimensional vector space over $\mathbb{P}$.

Theorem 1.0.1. *Let $\mathbb{F}$ a finite field.*

(i) *There exists a prime number p and $n \in \mathbb{N}$ such that $|\mathbb{F}| = p^n$.*

(ii) *Every two finite fields with p^n elements are isomorphic.*

Proof. See [1], p. 140, Cor. (3.1.4) and p. 153, Thm. (3.2.10). □

For the classical construction of $\mathbb{F}_q = \mathbb{F}_{p^n}$, let $f \in \mathbb{F}_p[X]$ a monic, irreducible polynomial of degree $\deg n$.[2] Let $I(f) := \{fg \mid g \in \mathbb{F}_p[X]\}$ the principal ideal generated by f and $\mathbb{F}_p[X]/I(f)$ the factor or residue class ring of $\mathbb{F}_p$ modulo $I(f)$. $\mathbb{F}_p[X]/I(f)$ is a field with p^n elements,

$$\mathbb{F}_p[X]/I(f) = \{g + I(f) \mid \deg(g) < n\} = \{\sum_{i=0}^{n-1} a_i X^i + I(f) \mid a_i \in \mathbb{F}_p\},$$

where addition and multiplication are explicitly given as

$$(g + I(f)) + (h + I(f)) := (g + h) + I(f)$$
$$(g + I(f)) \cdot (h + I(f)) := gh + I(f).$$

Let $\alpha := X + I(f), f := \sum_{i=0}^{n} f_i X^i$. For $a_i \in \mathbb{F}_p$, $i \in \{0, \ldots, n-1\}$ we have

$$(\sum_{i=0}^{n-1} a_i X^i) + I(f) = \sum_{i=0}^{n-1} (a_i X^i + I(f)) = \sum_{i=0}^{n-1} a_i (X^i + I(f)) = \sum_{i=0}^{n-1} a_i (X + I(f))^i = \sum_{i=0}^{n-1} a_i \alpha^i,$$

and

$$f(\alpha) = \sum_{i=0}^{n} f_i \alpha^i = \sum_{i=0}^{n} f_i X^i + I(f) = f + I(f) = I(f) = 0 \in \mathbb{F}_p[X]/I(f).$$

[2]Such a polynomial always exists for every degree n.

Hence, $\{1 = \alpha^0, \alpha, \ldots, \alpha^{n-1}\}$ is a generating system of $\mathbb{F}_p[X]/I(f)$ and α is a root of f, which is called the *minimal polynomial* of α over $\mathbb{F}_p$[3]. The generating system $\{1 = \alpha^0, \alpha, \ldots, \alpha^{n-1}\}$ is actually a basis of $\mathbb{F}_p[X]/I(f)$ over $\mathbb{F}_p$. Indeed, if

$$g := \sum_{i=0}^{n-1} b_i \alpha^i \in \mathbb{F}_p[X],\ g(\alpha) = 0,$$

then the minimal polynomial f divides g, hence $g = 0$ for degree reasons. Summarizing, we have

$$\mathbb{F}_p[X]/I(f) = \{ \sum_{i=0}^{n-1} a_i \alpha^i \mid a_i \in \mathbb{F}_p\}.$$

Remark 1.0.2. *Up to isomorphism, fields are uniquely determined. In the sequel, we denote the field with $q = p^m$ elements by $\mathbb{F}_q$ ($m, q \in \mathbb{N}$, p a prime number).*

For computations in software it is often more convenient to use another representation of field elements. Rather than expressing them as linear combinations they are represented as exponentials via the generating system $\{1 = \alpha^0, \alpha, \ldots, \alpha^{n-1}\}$.

Definition and Theorem 1.0.3 (Primitive elements, primitive polynomials)**.**

(i) *The multiplicative group $\mathbb{F}_q^\times$ of a finite field $\mathbb{F}_q$ is cyclic.*

(ii) *A generator α of $\mathbb{F}_q^\times$ is called* primitive element *of* $\mathbb{F}_q$.

(iii) *Minimal polynomials of primitive elements are called* primitive polynomials.

Proof. See [1], pp. 151–152. □

2 Linear codes

Let $q, m, n \in \mathbb{N}$, $q = p^m$, p a prime number.

The set of all n-tuples with components in $\mathbb{F}_q$ will be denoted by $\mathbb{F}_q^n$:

$$\mathbb{F}_q^n := \{\boldsymbol{x} = (x_0, \ldots, x_{n-1}) : x_0, \ldots, x_{n-1} \in \mathbb{F}_q\}$$

For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}_q^n$ and $\alpha \in \mathbb{F}_q$ define

$$\begin{array}{lllll} \boldsymbol{x} + \boldsymbol{y} & := & (x_0, \ldots, x_{n-1}) + (y_0, \ldots, y_{n-1}) & := & (x_0 + y_0, \ldots, x_{n-1} + y_{n-1}) \\ \alpha\boldsymbol{x} & := & \alpha(x_0, \ldots, x_{n-1}) & := & (\alpha x_0, \ldots, \alpha x_{n-1}) \end{array},$$

which gives $\mathbb{F}_q^n$ the structure of an n-dimensional vector space over $\mathbb{F}_q$.

[3]The polynomial f is monic, irreducible over $\mathbb{F}_p$ and has α as root. Using the division theorem it is easily shown that these three conditions determine f uniquely.

2.1 Encoding messages

Messages can be seen as members of $\mathbb{F}_q^k$, where k is the *length* of the message and $\mathbb{F}_q$ is the underlying alphabet. Because these messages are to be transmitted over a noisy channel, some redundancy has to be added, which can then be used by the receiver to recognize or correct errors. Technically, this is done by *embedding* the messages into a bigger vector space $\mathbb{F}_q^n$, $n > k$.[4] This procedure is also known as *encoding* the message.

Definition 2.1.1 (Encoder, linear code). *Let $k, m, n, q \in \mathbb{N}$ with $n > k$ and $q = p^m$ for a prime number p.*

(i) *An* encoder *is an injective linear map g from $\mathbb{F}_q^k$ into $\mathbb{F}_q^n$:*

$$g : \mathbb{F}_q^k \hookrightarrow \mathbb{F}_q^n.$$

(ii) *The image of the encoder g,*

$$\mathcal{C} := g(\mathbb{F}_q^k),$$

is an $\mathbb{F}_q$-subspace of $\mathbb{F}_q^n$, which is isomorphic to $\mathbb{F}_q^k$. $\mathcal{C}$ is called a linear $[n, k]$-code *or briefly an* $[n, k]$-code *over $\mathbb{F}_q$.*

(iii) *The number* k *is the dimension of the code and the number* n *is called the* block length *or just the* length *of the code $\mathcal{C}$.*[5]

(iv) *Vectors in $\mathcal{C}$ are called* codewords *or* code vectors*, whereas vectors in $\mathbb{F}_q^k$ are called* messages*, which is why $\mathbb{F}_q^k$ is also referred to as* message space.

(v) *Codes over the field $\mathbb{F}_2 := \{0, 1\}$ of two elements are called* binary codes.

Following the row-convention (writing vectors as row-vectors), the encoder can be expressed as multiplication by a matrix $\boldsymbol{G}$ of rank k.

$$g : \mathbb{F}_q^k \hookrightarrow \mathbb{F}_q^n : \boldsymbol{x} \mapsto \boldsymbol{xG} \tag{2.1.1}$$

$$\mathcal{C} = g(\mathbb{F}_q^k) = \{\boldsymbol{xG} \mid \boldsymbol{x} \in \mathbb{F}_q^k\} \tag{2.1.2}$$

Definition 2.1.2 (Generator matrix). *The matrix $\boldsymbol{G}$ in* (2.1.1)*, which is in general not uniquely determined, is called a* generator matrix *of the code $\mathcal{C}$. Its rows form a* basis *of $\mathcal{C}$.*

It is easily shown that generator matrices are related by regular matrices:

Theorem 2.1.3. *The set of all generator matrices of a linear code with generator matrix $\boldsymbol{G}$ is*

$$\{\boldsymbol{BG} \mid \boldsymbol{B} \in GL_k(\mathbb{F}_q)\},$$

where $GL_k(\mathbb{F}_q)$ is the set of all regular $k \times k$ matrices.

[4]It would be possible to allow $n \geq k$, but without redundancy there is no error correcting capability.

[5]The number k/n is called the *information rate* of the code $\mathcal{C}$. By design, it shall be as close to 1 as possible.

Proof. Let $\boldsymbol{B} \in \mathbb{F}_q^{k \times k}$ a regular matrix over $\mathbb{F}_q$, i.e. $\{\boldsymbol{xB} \mid \boldsymbol{x} \in \mathbb{F}_q^k\} = \mathbb{F}_q^k$. It follows that $\boldsymbol{BG}$ is also a generator matrix. □

Remark 2.1.4. *An $[n,k]$-code $\mathcal{C}$ can be considered either as the image of an injective linear map $g : \mathbb{F}_q^k \hookrightarrow \mathbb{F}_q^n$ or as the kernel of a surjective linear map $h : \mathbb{F}_q^n \to \mathbb{F}_q^{n-k}$.*

Proof. Indeed, let $\boldsymbol{G}$ a generator matrix of $\mathcal{C}$ of rank k, and let $\{\boldsymbol{x}_0, \dots, \boldsymbol{x}_{n-k-1}\}$ be a basis of the solution space of the homogenous linear system $\boldsymbol{G}\boldsymbol{x}^T = 0$, where $\boldsymbol{x} \in \mathbb{F}_q^n$. Define

$$h : \mathbb{F}_q^n \to \mathbb{F}_q^{n-k} \ : \ \boldsymbol{x} \mapsto \boldsymbol{x}\boldsymbol{H}^T,$$

where

$$\boldsymbol{H} := \begin{pmatrix} \boldsymbol{x}_0 \\ \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}_{n-k-1} \end{pmatrix} \in \mathbb{F}_q^{(n-k) \times n},$$

$$rg(\boldsymbol{H}) = n - k, \tag{2.1.3}$$

which proves that h is surjective. The rows of $\boldsymbol{G}$ form a basis of the code $\mathcal{C}$, therefore $\mathcal{C}$ is by construction a subspace of $\ker(h)$. On the other hand we have $\dim(\mathcal{C}) = k = \dim(\ker(h))$, which gives $\mathcal{C} = \ker(h)$. □

In other words,

$$\boldsymbol{c} \in \mathcal{C} \iff \boldsymbol{c}\boldsymbol{H}^T = \boldsymbol{0}, \tag{2.1.4}$$

which makes it easy to check if a vector $\boldsymbol{v} \in \mathbb{F}_q^n$ is a codeword or not. The matrix $\boldsymbol{H}$ is therefore called a *check matrix.*

Definition 2.1.5 (Check matrices). *Let $k, m, n, q \in \mathbb{N}$ with $n > k$ and $q = p^m$ for a prime number p. Let $\mathcal{C}$ be an $[n,k]$-code over $\mathbb{F}_q$.*

Then there exists an $(n-k) \times n$ matrix $\boldsymbol{H}$ over $\mathbb{F}_q$, which is of rank $n - k$ and satisfies

$$\mathcal{C} = \ker(h) = \{\boldsymbol{w} \in \mathbb{F}_q^n \mid \boldsymbol{w}\boldsymbol{H}^T = \boldsymbol{0}\},$$

where $\boldsymbol{H}^T$ denotes the transpose of the matrix $\boldsymbol{H}$. Any such matrix $\boldsymbol{H}$ is called (parity) check matrix *of $\mathcal{C}$.*

Remark 2.1.6. *Let $\mathcal{C}$ be an $[n,k]$-code over $\mathbb{F}_q$ with generator matrix $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$ and $\boldsymbol{H} \in \mathbb{F}_q^{(n-k) \times n}$ a check matrix of $\mathcal{C}$. Then there are equivalent:*

(i) *$\boldsymbol{H}$ is a check matrix for $\mathcal{C}$.*

(ii) $\boldsymbol{G}\boldsymbol{H}^T = \boldsymbol{0}$.

Proof. Let $\boldsymbol{u} \in \mathbb{F}_q^k$ arbitrary, $\boldsymbol{c} = \boldsymbol{uG}$.

(i) ⇒ (ii): $\boldsymbol{0} = \boldsymbol{c}\boldsymbol{H}^T = (\boldsymbol{uG})\boldsymbol{H}^T = \boldsymbol{u}(\boldsymbol{G}\boldsymbol{H}^T)$. As $\boldsymbol{u}$ is arbitrarily chosen, $\boldsymbol{G}\boldsymbol{H}^T = \boldsymbol{0}$ follows.

(ii) ⇒ (i): $\boldsymbol{c}\boldsymbol{H}^T = (\boldsymbol{uG})\boldsymbol{H}^T = \boldsymbol{u}(\boldsymbol{G}\boldsymbol{H}^T) = \boldsymbol{0}$. □

Just like generator matrices (Theorem (2.1.3)), parity check matrices are in general not uniquely determinded.

Theorem 2.1.7. *The set of all parity check matrices of a linear code with parity check matrix $\boldsymbol{H}$ is*

$$\{\boldsymbol{S}\boldsymbol{H} \mid \boldsymbol{S} \in GL_{n-k}(\mathbb{F}_q)\},$$

where $GL_{n-k}(\mathbb{F}_q)$ is the set of all regular $(n-k) \times (n-k)$ matrices over $\mathbb{F}_q$.

Proof. Let $\boldsymbol{H} \in \mathbb{F}_q^{(n-k)\times n}$ a parity check matrix and $\boldsymbol{G} \in \mathbb{F}_q^{k\times n}$ an associated generator matrix of some linear code $\mathcal{C}$ over $\mathbb{F}_q$, i.e. $\boldsymbol{G}\boldsymbol{H}^T = \boldsymbol{0}$.

Let $\boldsymbol{S} \in \mathbb{F}_q^{(n-k)\times(n-k)}$ a regular matrix and let $\boldsymbol{H}' := \boldsymbol{S}\boldsymbol{H}$. Since

$$\boldsymbol{G}\boldsymbol{H}'^T = (\boldsymbol{G}\boldsymbol{H}^T)\boldsymbol{S}^T = \boldsymbol{0},$$

$\boldsymbol{H}'$ will be another parity check matrix. □

2.2 Decoding messages

As we have shown, messages are encoded into codewords using a generator matrix (2.1.1). The codewords $\boldsymbol{c}$ are sent over an unreliable channel. However, actually received are vectors $\boldsymbol{y} \in \mathbb{F}_q^n$, i.e. we have $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ with some error vector $\boldsymbol{e} \in \mathbb{F}_q^n$. The question that now arises is how to *decode* the correct $\boldsymbol{c}$.

In practice, one relies on the so-called *maximum-likelihood-principle* or in other words, one assumes that with high probability not too many errors have occurred. The decoder tries to find the codeword $\boldsymbol{c}$ which is closest to the received vector $\boldsymbol{y}$, and hands the message $g^{-1}(\boldsymbol{c})$ over to the receiver, where g^{-1} is the inverse of the injective encoding map $g : \mathbb{F}_q^k \to \mathbb{F}_q^n$.

In order to make sense, we will now define a notion of *distance* on $\mathbb{F}_q^n$ and give conditions under which a successful decoding process is possible.

Definition and Theorem 2.2.1 (Hamming metric, Hamming weight)**.** *Let $m, n, q \in \mathbb{N}$ with $q = p^m$ for a prime number p. The function*

$$d : \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{N} : (\boldsymbol{u}, \boldsymbol{v}) \mapsto |\{i \in \{0, \ldots, n-1\} : u_i \neq v_i\}|$$

is a metric on $\mathbb{F}_q^n$, the so-called Hamming metric. *It follows that d satisfies*

$$\begin{aligned} d(\boldsymbol{u},\boldsymbol{v}) &= 0 \iff \boldsymbol{u} = \boldsymbol{v} \\ d(\boldsymbol{u},\boldsymbol{v}) &= d(\boldsymbol{v},\boldsymbol{u}) \\ d(\boldsymbol{u},\boldsymbol{v}) &\leq d(\boldsymbol{u},\boldsymbol{w}) + d(\boldsymbol{w},\boldsymbol{v}) \end{aligned}$$

for all $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{F}_q^n$.

The nonnegative integer $d(\boldsymbol{u},\boldsymbol{v})$ is called the Hamming distance *between the vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}_q^n$. Hence, the pair $(\mathbb{F}_q^n, d)$ is a metric space, the* Hamming space *of dimension n over $\mathbb{F}_q$.*

The Hamming distance is invariant under translation and multiplication by nonzero scalars: For $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in (H, \mathbb{F}_q)$ and $\lambda \in \mathbb{F}_q,\ \lambda \neq 0$,

$$\begin{aligned} d(\boldsymbol{u},\boldsymbol{v}) &= d(\boldsymbol{u}+\boldsymbol{w}, \boldsymbol{v}+\boldsymbol{w}) \\ d(\boldsymbol{u},\boldsymbol{v}) &= d(\lambda\boldsymbol{u}, \lambda\boldsymbol{v}) \end{aligned}$$

For a vector $\boldsymbol{v} \in \mathbb{F}_q^n$, its Hamming weight *is defined as*

$$w(\boldsymbol{v}) := d(\boldsymbol{v}, \boldsymbol{0}). \tag{2.2.1}$$

Proof. The equivalence $d(\boldsymbol{u},\boldsymbol{v}) = 0 \iff \boldsymbol{u} = \boldsymbol{v}$ and the symmetry $d(\boldsymbol{u},\boldsymbol{v}) = d(\boldsymbol{v},\boldsymbol{u})$ are trivial. It remains to show the triangle equality.

Let $\boldsymbol{u}$,$\boldsymbol{v}$,$\boldsymbol{w} \in \mathbb{F}_q^n$. Suppose $u_i \neq v_i$ for the ith component. Then $u_i \neq w_i$ or $v_i \neq w_i$, which establishes the triangle equality. Using the linear structure of $\mathbb{F}_q^n$ we have $d(\boldsymbol{u},\boldsymbol{v}) = d(\boldsymbol{u}-\boldsymbol{v},\boldsymbol{0}) = d(\boldsymbol{u}+\boldsymbol{w}-\boldsymbol{w}-\boldsymbol{v},\boldsymbol{0}) = d(\boldsymbol{u}+\boldsymbol{w},\boldsymbol{v}+\boldsymbol{w})$. Similarly, for $\lambda \neq 0$, $d(\boldsymbol{u},\boldsymbol{v}) = d(\boldsymbol{u}-\boldsymbol{v},\boldsymbol{0}) = d(\lambda(\boldsymbol{u}-\boldsymbol{v}),\boldsymbol{0}) = d(\lambda\boldsymbol{u}-\lambda\boldsymbol{v},\boldsymbol{0}) = d(\lambda\boldsymbol{u},\lambda\boldsymbol{v})$, which completes the proof. □

Definition 2.2.2 (Packing radius, minimum distance). *Let $n,t \in \mathbb{N}$, $\mathcal{C}$ a linear code over $\mathbb{F}_q$ and $\boldsymbol{x} \in \mathbb{F}_q^n$.*

$$\mathbb{B}_n(\boldsymbol{x},t) := \{\boldsymbol{y} \in \mathbb{F}_q^n \mid d(\boldsymbol{x},\boldsymbol{y}) \leq t\} \tag{2.2.2}$$

denotes the ball of radius t around $\boldsymbol{x}$.

(i) *The* packing radius *of $\mathcal{C}$ is the largest integer t, such that balls of radius t around codewords do not intersect:*

$$pr(\mathcal{C}) := \max_{\boldsymbol{c},\boldsymbol{c}' \in C, \boldsymbol{c} \neq \boldsymbol{c}'} \{t \in \mathbb{N} \mid \mathbb{B}_n(\boldsymbol{c},t) \cap \mathbb{B}_n(\boldsymbol{c}',t) = \emptyset\} \tag{2.2.3}$$

(ii) *The* minimum distance *of $\mathcal{C}$ is defined as*

$$d := dist(\mathcal{C}) := \min\{d(\boldsymbol{c},\boldsymbol{c}') \mid \boldsymbol{c},\boldsymbol{c}' \in \mathcal{C},\ \boldsymbol{c} \neq \boldsymbol{c}'\}. \tag{2.2.4}$$

Remark 2.2.3. *Using the linear structure of $\mathbb{F}_q^n$, i.e. $d(\boldsymbol{u},\boldsymbol{v}) = d(\boldsymbol{u}-\boldsymbol{v},\boldsymbol{0})$ for $\boldsymbol{u},\boldsymbol{v} \in \mathbb{F}_q^n$, we see that the minimal distance of a linear code $\mathcal{C}$ is equivalent with its minimal weight:*

$$\min\{w(\boldsymbol{c}) \mid \boldsymbol{c} \in \mathcal{C}, \boldsymbol{c} \neq \boldsymbol{0}\} = dist(\mathcal{C}). \tag{2.2.5}$$

Note that a $[n,k]$-code $\mathcal{C}$ with minimal weight d is also denoted as $[n,k,d]$-code. Such a code is also known as of type $[n,k,d]$.

Theorem 2.2.4. *The check matrix $\boldsymbol{H}$ of an $[n,k,d]$-code over $\mathbb{F}_q$ with $0 < k < n$ has the following properties:*

(i) *$\boldsymbol{H}$ is an $(n-k) \times n$ matrix over $\mathbb{F}_q$ of rank $n-k$.*

(ii) *Any $d-1$ columns are linearly independent.*

(iii) *There exist d columns that are linearly dependent.*

Conversely, any matrix $\boldsymbol{H}$ satisfying these properties is a check matrix of an $[n,k,d]$-code over $\mathbb{F}_q$.

Proof. (i): See (2.1).

(ii): Assume $s < d$ columns of $\boldsymbol{H}$ linear dependent. In matrix form this means that there is a $\boldsymbol{c} \in \mathbb{F}_q^n$ such that $\boldsymbol{c}\boldsymbol{H}^T = \boldsymbol{0}$ and $w(\boldsymbol{c}) = s < d$. By definition of $\boldsymbol{H}$ it follows that $\boldsymbol{c} \in \mathcal{C}$. Contradiction.

(iii) By assumption, there is a codeword $\boldsymbol{c} \in \mathcal{C}$, $\boldsymbol{c} \neq \boldsymbol{0}$, $w(\boldsymbol{c}) = d$. Because $\boldsymbol{c} \in \mathcal{C}$, it follows that $\boldsymbol{c}\boldsymbol{H}^T = \boldsymbol{0}$, and therefore exist d linear dependent columns.

Conversely, let $\boldsymbol{H} \in \mathbb{F}_q^{(n-k)\times n}$ with rank $n-k$. This means, that the set $\mathcal{C}' := \{\boldsymbol{x} \in \mathbb{F}_q^n \mid \boldsymbol{x}\boldsymbol{H}^T = \boldsymbol{0}\}$ is a subspace of $\mathbb{F}_q^n$ with dimension k. As before, we conclude that d is the minimum distance of $\mathcal{C}'$. □

As an immediate consequence of Theorem (2.2.4) we get:

Corollary 2.2.5. *Every $(n-k) \times k$ matrix over $\mathbb{F}_q$, in which any $d-1$ columns are independent, is a check matrix of some $[n,k]$-code $\mathcal{C}$ over $\mathbb{F}_q$ with minimal distance $dist(\mathcal{C}) \geq d$.*

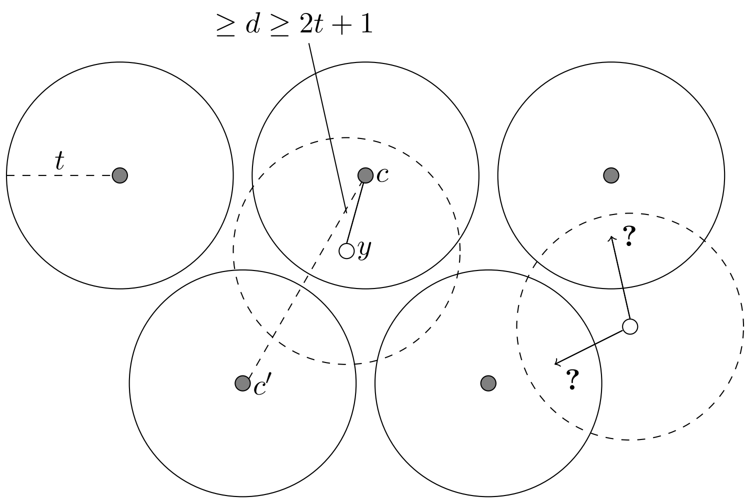

Figure 1: The maximum-likelihood-decoding method

Maximum likelihood-decoding Note that the minimal distance d of a linear code $\mathcal{C}$ is a measure for the quality of the code, i.e. for its error-correcting capabilities. If $d \geq 2t+1$, a decoder using the maximum-likelihood principle can correct up to t errors.

It is possible to correct up to

$$t = \lfloor (dist(\mathcal{C}) - 1)/2 \rfloor \tag{2.2.6}$$

errors in the following way [1]:

(i) Using *maximum-likelihood-decoding*, a vector $\boldsymbol{y} \in \mathbb{F}_q^n$ is decoded in a codeword $\boldsymbol{c} \in C$, which is closest to $\boldsymbol{y}$ with respect to the Hamming metric. In formal terms: $\boldsymbol{y}$ is decoded into a codeword $\boldsymbol{c} \in \mathbb{F}_q^n$, such that

$$d(\boldsymbol{c}, \boldsymbol{y}) \leq d(\boldsymbol{c}', \boldsymbol{y}), \ \forall \boldsymbol{c}' \in C.$$

If there are several $\boldsymbol{c} \in C$ with this property, one of them is chosen at random.

(ii) If the codeword $\boldsymbol{c} \in C$ was sent and no more than t errors occured during transmission, the received vector is

$$\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e} \in \mathbb{F}_q^n,$$

where e denotes the *error vector*. It satisfies

$$d(\boldsymbol{c}, \boldsymbol{y}) = d(\boldsymbol{e}, \boldsymbol{0}) \leq t$$

and hence $\boldsymbol{c}$ is the $unique$ element of $\mathcal{C}$ which lies in a ball of radius t around $\boldsymbol{y}$. A maximum likelihood decoder yields this element $\boldsymbol{c}$, and so we obtain the correct codeword.

Remark 2.2.6. *The packing radius t of a linear code $\mathcal{C}$ is $\lfloor (dist(\mathcal{C}) - 1)/2 \rfloor$.* □

2.3 Equivalence of codes and information sets

Definition 2.3.1 (Isometry, equivalence). *Let $(\mathbb{F}_q^n, d)$ the Hamming space with Hamming metric d ($n \in \mathbb{N}$).*

(i) *A bijective linear map $\iota : \mathbb{F}_q^n \to \mathbb{F}_q^n$ with $d(\boldsymbol{u}, \boldsymbol{v}) = d(\iota(\boldsymbol{u}), \iota(\boldsymbol{v}))\ \forall \boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}_q^n$ is called* isometry.[6]

(ii) *Let ι be an isometry. Two $[n, k]$-codes $\mathcal{C}, \mathcal{C}'$ over $\mathbb{F}_q$ are called* equivalent *if $\iota(\mathcal{C}) = \mathcal{C}'$.*

Remark 2.3.2. *The Hamming weight of a vector $\boldsymbol{v} \in \mathbb{F}_q^n$ is invariant under an isometry ι.*
Indeed, let $\boldsymbol{v} \in \mathbb{F}_q^n$. Then $w(\boldsymbol{v}) = d(\boldsymbol{v}, \boldsymbol{0}) = d(\iota(\boldsymbol{v}), \iota(\boldsymbol{0})) = d(\iota(\boldsymbol{v}), \boldsymbol{0}) = w(\iota(\boldsymbol{v}))$.

Let $\iota : \mathbb{F}_q^n \to \mathbb{F}_q^n$ be an isometry. Like any other linear map on $\mathbb{F}_q^n$, it is uniquely determined by the images of the unit vectors. By remark (2.3.2), isometries do not change the Hamming weight. Hence, units vectors are mapped to multiples of unit vectors. Conversely, each linear map with this property is clearly an isometry.

Remark 2.3.3. *Isometries on $\mathbb{F}_q^n$ are expressed by those invertible $\mathbb{F}_q^{n\times n}$ matrices, which contain in each row and column precisely one element of $\mathbb{F}_q$.*

Let $\boldsymbol{J} \in \mathbb{F}_q^{n\times n}$ an isometry in matrix form, $\boldsymbol{G} \in \mathbb{F}_q^{k\times n}$ a generator matrix of a $[n, k]$-code $\mathcal{C}$ over $\mathbb{F}_q$. Hence, $\boldsymbol{GJ}$ is just $\boldsymbol{G}$ with some columns permutated and/or multiplied by some non-zero field element. On the other, hand we can also multiply $\boldsymbol{G}$ from the left by some invertible $\boldsymbol{B} \in \mathbb{F}_q^k$ without leaving the code $\mathcal{C}$.

It follows that we can apply to $\boldsymbol{G}$ the *Gaussian algorithm*. Multiplication with some $\boldsymbol{B}$ will generate unit vectors in certain colums, which can be shifted afterwards by multiplication from the right with isometries.

Definition and Theorem 2.3.4 (Systematic encoding, information sets). *For each $[n, k]$-code $\mathcal{C}$ with generator matrix $\boldsymbol{G}$ there exists an equivalent $[n, k]$-code $\mathcal{C}'$ with generator matrix $\boldsymbol{G}'$ of the form*

$$\boldsymbol{G}' = (\boldsymbol{I}_k | \boldsymbol{A}), \tag{2.3.1}$$

where $I_k \in \mathbb{F}_q^{k\times k}$ denotes the $k \times k$ identity matrix and $A \in \mathbb{F}_q^{k\times(n-k)}$. The corresponding encoding $\boldsymbol{v} \mapsto \boldsymbol{v}\boldsymbol{G}'$ ($\boldsymbol{v} \in \mathbb{F}_q^k$) is called systematic encoding *and $\boldsymbol{G}'$ a* systematic generator matrix *of $\mathcal{C}'$.*

The first k coordinates of codewords $\boldsymbol{c} \in \mathcal{C}'$ are called its information set, *the remaining $n - k$ places are known as* redundancy set[7].

Remark 2.3.5. *When using systematic encoding $\boldsymbol{v} \mapsto \boldsymbol{v}\boldsymbol{G}' = \boldsymbol{v}(\boldsymbol{I}_k|\boldsymbol{A}) = \boldsymbol{w}$, the first k coordinates simply repeat the k components of the message $\boldsymbol{v}$. However, errors may also have occured in the first k coordinates, so decoding by simply reading out the first k coordinates values does not work. But if we are given the values of a codeword on these first k coordinates, then the remaining $n - k$ coordinates are* uniquely *determined. For let $\boldsymbol{H}'$ a corresponding check matrix, i.e. $\boldsymbol{G}'\boldsymbol{H}'^T = \boldsymbol{0}$. It follows by inspection that $\boldsymbol{H}' = (-\boldsymbol{A}^T|\boldsymbol{I}_{n-k})$. Two codewords $\boldsymbol{c}, \boldsymbol{c}'$ which coincide in the first k coordinates must therefore be equal in the last $n - k$ coordinates as well.*

[6]Obvious isometries are the permutations of the coordinates.
[7]They are also called *check bits*, since they may be used for error correction and error detection.

2.4 Generalized Reed Solomon codes

Notation 2.4.1. [8] *For $1 \leq k \leq n \in \mathbb{N}$ denote the subspaces of all polynomials over $\mathbb{F}_q$ of degree strict less than k by*

$$\mathbb{F}_q[X]_{<k} := \bigoplus_{i=0}^{k-1} \mathbb{F}_q X^i.$$

Let $n \leq q \in \mathbb{N}$, $\boldsymbol{L} = (L_0, \dots, L_{n-1})$ an n-tuple of pairwise distinct elements of $\mathbb{F}_q$ and $\boldsymbol{\beta} = (\beta_0, \dots, \beta_{n-1})$ an n-tuple on nonzero elements of $\mathbb{F}_q$.

Definition 2.4.2 (Generalized Reed-Solomon code). *For $1 \leq k \leq n \in \mathbb{N}$ we define the* Generalized Reed-Solomon-Code *$GRS_k(\boldsymbol{L}, \boldsymbol{\beta})$ as*

$$GRS_k(\boldsymbol{L}, \boldsymbol{\beta}) := \{(f(L_0)\beta_0, \dots, f(L_{n-1})\beta_{n-1}) \mid f(X) \in \mathbb{F}_q[X]_{<k}\}. \tag{2.4.1}$$

We add an equivalent, more explicit formulation of (2.4.1). Consider the following notations:

$$\begin{aligned}
\Phi_{(\boldsymbol{L})} &:= (L_j^i)_{0 \leq i < k,\ 0 \leq j < n} \\
&:= \begin{bmatrix} 1 & 1 & \dots & 1 \\ L_0 & L_1 & \dots & L_{n-1} \\ L_0^2 & L_1^2 & \dots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{k-1} & L_1^{k-1} & \dots & L_{n-1}^{k-1} \end{bmatrix} \in \mathbb{F}_q^{k \times n}, \\
\boldsymbol{\Delta}(\boldsymbol{\beta}) &:= \begin{bmatrix} \beta_0 & 0 & \dots & 0 \\ 0 & \beta_1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \beta_{n-1} \end{bmatrix} \in \mathbb{F}_q^{n \times n}, \qquad (2.4.2) \\
f(X)\Phi_{(\boldsymbol{L})} &:= (f_0, \dots, f_{k-1})\Phi_{(\boldsymbol{L})} = (f(L_0), \dots, f(L_{n-1})), \\
\boldsymbol{\Gamma} &:= \begin{bmatrix} \beta_0 & \beta_1 & \dots & \beta_{n-1} \\ L_0\beta_0 & L_1\beta_1 & \dots & L_{n-1}\beta_{n-1} \\ L_0^2\beta_0 & L_1^2\beta_1 & \dots & L_{n-1}^2\beta_{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{k-1}\beta_0 & L_1^{k-1}\beta_1 & \dots & L_{n-1}^{k-1}\beta_{n-1} \end{bmatrix} = \Phi_{(\boldsymbol{L})}\boldsymbol{\Delta}(\boldsymbol{\beta}) \in \mathbb{F}_q^{k \times n}.
\end{aligned}$$

Using these notations we see that $GRS_k(\boldsymbol{L}, \boldsymbol{\beta})$ is formally generated in the following way:

$$GRS_k(\boldsymbol{L}, \boldsymbol{\beta}) = \{f(X)\boldsymbol{\Gamma} \mid f(X) \in \mathbb{F}_q[X]_{<k}\}. \tag{2.4.3}$$

Remark 2.4.3. *As $\Phi_{(\boldsymbol{L})}$ is a submatrix of the Vandermonde matrix and $\boldsymbol{\Delta}(\boldsymbol{\beta})$ is invertible, $\Phi_{(\boldsymbol{L})}$ and $\boldsymbol{\Gamma}$ do have rank k.*

An important point to note here is that the sequence $\boldsymbol{\beta}$ can be replaced by a polynomial $g(X) \in \mathbb{F}_q[X]$, which later will be used to define so-called Goppa codes. To this end, we first recall some definitions and well-known facts.

Let R and S commutative rings with 1.

[8] For the following explanations, see [1].

Definition 2.4.4 (Ideal, principal ideal, relatively prime ideals).

(i) *A subgroup (I,+) of (R,+) is called* ideal *iff* $ax \in I$ *for all* $a \in R$ *and* $x \in I$.

(ii) *For* $a \in R$ *the* principal ideal ⟨a⟩ *generated by* a *is defined as*

$$\langle a \rangle := (a) := \{ar \,|\, r \in R\}.$$

(iii) *Let* I_1, s, I_n *ideals in R* $(n \geq 2)$. *They are called* relatively prime *(or* coprime*) iff* $I_k + I_l = R$ *for* $k \neq l$.

Theorem 2.4.5. *Let* ϕ: $R \to S$ *a surjective ring homomorphism. The induced homomorphism*

$$h : R/\ker\phi \to S \;\; (a + \ker\phi \mapsto \phi(a))$$

is a ring isomorphism.

Proof. [18], (6.11). □

Theorem 2.4.6. *Let* $I_1, \ldots, I_n$ *relatively prime ideals in R* $(n \geq 2)$. *The canonical ring homomorphism*

$$\alpha \colon R \longrightarrow R/I_1 \; \times \; \cdots \; \times \; R/I_n$$

$$r \mapsto (r + I_1, \ldots, r + I_n)$$

is an epimorphism with kernel

$$ker \; \alpha = \bigcap_{i=1}^{n} I_i.$$

Proof. [18], (6.24). □

Combining (2.4.6) and (2.4.5) yields:

Theorem 2.4.7 (Chinese Remainder Theorem (CRT)).

$$R/\bigcap_{k=1}^{n} I_k \simeq R/I_1 \; \times \; \cdots \; \times \; R/I_n, \tag{2.4.4}$$

$$(r + \bigcap_{k=1}^{n} I_k \mapsto (r + I_1, \ldots, r + I_n)).$$

In other words, theorem (2.4.7) states that for pairwise relatively prime ideals $I_1, \ldots, I_n$, $n \geq 2$ and arbitrary elements $r_1, \ldots, r_n \in R$, there is always a solution for the system

$$\begin{aligned} X &\equiv r_1 \mod I_1 \\ &\vdots \\ X &\equiv r_n \mod I_n, \end{aligned} \tag{2.4.5}$$

and that for a particular solution r the set of all solutions is $r + \bigcap_{k=1}^{n} I_k$.

We finally note:

Lemma 2.4.8. *A coset $r + I \in R/I$ is a unit of R/I if and only if $I(r) = \langle r\rangle$ and I are relatively prime.*

Proof. [18], (6.32). $\square$

Consider now

$$h(X) := \prod_{i=0}^{n-1}(X - L_i), \tag{2.4.6}$$

and denote by $\langle h\rangle$ the principal ideal in $\mathbb{F}_q[X]$ generated by $h(X)$:

$$\langle h\rangle = \{hg \mid g \in \mathbb{F}_q[X]\} = \bigcap_{i=0}^{n-1}\langle X - L_i\rangle,$$

As the L_i $(0 \le i < n)$ are pairwise distinct, the $X - L_i$ are relatively prime in $\mathbb{F}_q[X]$ and theorem (2.4.7) states:

$$\mathbb{F}_q[X]/\langle h\rangle \cong \mathbb{F}_q[X]/\langle X - L_0\rangle \ \times\ \dots\ \times\ \mathbb{F}_q[X]/\langle X - L_{n-1}\rangle.$$

Polynomial division gives for $i \in \{0, \dots, n-1\}$ and $f(X) \in \mathbb{F}_q[X]$ a unique representation as:

$$f(X) = q(X)(X - L_i) + f(L_i),$$

where $q(X) \in \mathbb{F}_q[X]$. Thus we have an $\mathbb{F}_q$- algebra isomorphism

$$\Phi\colon \mathbb{F}_q[X]/\langle h\rangle \xrightarrow{\simeq} \mathbb{F}_q^n$$

$$f + \langle h\rangle \mapsto (f(L_0), \dots, f(L_{n-1}))$$

This map restricts to an isomorphism between the two groups of units, which are the polynomials $g(X) \in \mathbb{F}_q[X]/\langle h\rangle$ with $g(L_i) \neq 0$ for $0 \le i < n$ on one side (due to lemma (2.4.8)) and $(\mathbb{F}_q^\times)^n$, the sequences of length n over $\mathbb{F}_q$ whose entries are all non-zero, on the other side (where we have the Hadamard product resp. componentwise multiplication):

$$\Phi\colon (\mathbb{F}_q[X]/\langle h\rangle)^\times \longrightarrow (\mathbb{F}_q^\times)^n \tag{2.4.7}$$

$$g + \langle h\rangle \mapsto (g(L_0), \dots, g(L_{n-1})). \tag{2.4.8}$$

In the other direction, given a sequence $\boldsymbol{c} = (c_0, \dots, c_{n-1})$ with $c_i \neq 0$ for $0 \le i < n$, we obtain the inverse image under the map Φ via Lagrange's interpolation formula. Namely, we have

$$\Phi^{-1}\colon (\mathbb{F}_q^\times)^n \longrightarrow \mathbb{F}_q[X]/\langle h\rangle \tag{2.4.9}$$

$$\boldsymbol{c} = (c_0, \dots, c_{n-1}) \mapsto \sum_{i=0}^{n-1}\frac{c_i}{l(L_i)}\prod_{\substack{j=0\\ j\neq i}}^{n-1}(X - L_j) + \langle h\rangle, \tag{2.4.10}$$

where $l(X)$ is the unique polynomial of degree less than n with

$$l(L_i) := \prod_{\substack{j=0\\ j\neq i}}^{n-1}(L_i - L_j), \tag{2.4.11}$$

and is given by Lagrange in explicit form as

$$l(X) = \sum_{i=0}^{n-1} l(L_i) \frac{\prod_{\substack{j=0\\j\neq i}}^{n-1}(X-L_j)}{\prod_{\substack{j=0\\j\neq i}}^{n-1}(L_i-L_j)} = \sum_{i=0}^{n-1}\prod_{\substack{j=0\\j\neq i}}^{n-1}(X-L_j). \tag{2.4.12}$$

By construction, the residue class $l + \langle h\rangle$ is a unit in $\mathbb{F}_q[X]/\langle h\rangle$. It serves in the following as a normalizing factor. For each unit $g + \langle h\rangle$ we build the following linear code (which is isometric to $GRS_k(\boldsymbol{L}, \mathbb{1}_n)$, where $\mathbb{1} := (1, 1, \ldots, 1) \in \mathbb{F}_q^n$).

Definition 2.4.9.

$$GRS_k(\boldsymbol{L}, g) := GRS_k(\boldsymbol{L}, \boldsymbol{\beta}) \tag{2.4.13}$$

where $\boldsymbol{\beta} = (\beta_0, \ldots, \beta_{n-1})$ *is the sequence whose entries are* $\beta_i = \dfrac{g(L_i)}{l(L_i)}$.

Theorem 2.4.10.

$$GRS_k(\boldsymbol{L}, g) = \left\{ \left(\frac{c_1 g(L_0)}{l(L_0)}, \ldots, \frac{c_{n-1} g(L_{n-1})}{l(L_{n-1})}\right) \mid \boldsymbol{c} \in GRS_k(\boldsymbol{L}, \mathbb{1}_n)\right\} \tag{2.4.14}$$

Proof. Let $\boldsymbol{c} \in GRS_k(\boldsymbol{L}, g)$ and $\boldsymbol{\beta} = \left(\frac{g(L_0)}{l(L_0)}, \ldots, \frac{g(L_{n-1})}{l(L_{n-1})}\right)$.

Using the notations (2.4.2), there exists $f(X) \in \mathbb{F}_q[X]_{<k}$ such that

$$\begin{aligned} \boldsymbol{c} &= f(X)\boldsymbol{\Gamma} \\ &= f(X)\phi_{(\boldsymbol{L})}\boldsymbol{\Delta}(\boldsymbol{\beta}) \\ &= \boldsymbol{c}'\boldsymbol{\Delta}(\boldsymbol{\beta}), \end{aligned}$$

where $f(X)\phi_{(\boldsymbol{L})} =: \boldsymbol{c}' \in GRS_k(\boldsymbol{L}, \mathbb{1}_n)$. This shows (2.4.14). □

The following theorem will show extremely useful characterizations of $GRS_k(\boldsymbol{L}, g)$ codes. They will be used below for the definition of alternant and Goppa codes.

Theorem 2.4.11. *Let* $1 \leq k \leq n \leq q \in \mathbb{N}$.
Let $\boldsymbol{L} = (L_0, \ldots, L_{n-1})$ *be a sequence of pairwise distinct elements of* $\mathbb{F}_q$ *and*

$$h(X) := \prod_{i=0}^{n-1}(X - L_i).$$

Let $g(X) \in \mathbb{F}_q[X]_{<n}$ *with* $g(L_i) \neq 0$ *for all* $0 \leq i < n$.

$$GRS_k(\boldsymbol{L}, g) = \left\{ \boldsymbol{c} \in \mathbb{F}_q^n \mid \exists f(X) \in \mathbb{F}_q[X]_{<k} \colon \sum_{i=0}^{n-1} c_i \prod_{\substack{j=0\\j\neq i}}^{n-1}(X - L_j) \equiv fg \mod \langle h\rangle \right\} \tag{2.4.15}$$

If $\deg g(X) = n - k$, *then*

$$GRS_k(\boldsymbol{L}, g) = \left\{ \boldsymbol{c} \in \mathbb{F}_q^n \mid \exists f(X) \in \mathbb{F}_q[X]_{<k} \colon \sum_{i=0}^{n-1} c_i \prod_{\substack{j=0\\j\neq i}}^{n-1}(X - L_j) \equiv 0 \mod \langle g\rangle \right\}. \tag{2.4.16}$$

$GRS_k(\boldsymbol{L}, g)$ can also be characterized as the set of all $\boldsymbol{c} \in \mathbb{F}_q^n$ which satisfies

$$\sum_{i=0}^{n-1} c_i(X - L_i)^{-1} = 0 \tag{2.4.17}$$

in $\mathbb{F}_q[X]/\langle g \rangle$.

Proof. Let $\boldsymbol{c} \in \mathbb{F}_q^n$.

As in the proof of (2.4.14) we see that $\boldsymbol{c} \in GRS_k(\boldsymbol{L}, g)$ if and only if there exists a polynomial $f(X) \in \mathbb{F}_q[X]_{<k}$ such that

$$\boldsymbol{c} = \left(\frac{f(L_0)g(L_0)}{l(L_0)}, \dots, \frac{f(L_{n-1})g(L_{n-1})}{l(L_{n-1})} \right)$$

Recall that on $(\mathbb{F}_q^\times)^n$ the multiplication is defined componentwise, see (2.4.7). Therefore, we get

$$\begin{aligned} (c_0 l(L_0), \dots, c_{n-1} l(L_{n-1})) &= (fg(L_0), \dots, fg(L_{n-1})) = \phi(fg + \langle h \rangle) \\ \overset{(2.4.9)}{\Rightarrow} \quad fg + \langle h \rangle &= \phi^{-1}(c_0 l(L_0), \dots, c_{n-1} l(L_{n-1})) \\ \overset{(2.4.10)}{\Rightarrow} \qquad fg &= \textstyle\sum_{i=0}^{n-1} c_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - L_j) \mod \langle h \rangle \end{aligned}$$

Let $\deg g(X) = n - k$ and $\boldsymbol{c} \in GRS_k(\boldsymbol{L}, g)$ a codeword. According to the last shown equation, there exist polynomials $f(X) \in \mathbb{F}_q[X]_{<k}$ and $s(X) \in \mathbb{F}_q[X]$ with

$$\sum_{i=0}^{n-1} c_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - L_j) = f(X)g(X) + s(X)h(X).$$

Because $\deg f(X) < k$, it follows that $\deg f(X)g(X) < n = \deg h(X)$, which implies

$$\sum_{i=0}^{n-1} c_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - L_j) = f(X)g(X),$$

and which is finally equivalent to

$$\sum_{i=0}^{n-1} c_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} (X - L_j) \equiv 0 \mod \langle g \rangle.$$

Using (2.4.8), it follows that $h + \langle g \rangle$ is a unit in $\mathbb{F}_q[X]/\langle g \rangle$. Multiplication by its inverse yields therefore in $\mathbb{F}_q[X]/\langle g \rangle$

$$\sum_{i=0}^{n-1} c_i(X - L_i)^{-1} = 0.$$

Since these arguments can be reversed, we obtain the assertion. □

2.4.1 Alternant codes

The class of alternant codes is obtained by restricting GRS-codes to subfields which means we now start with an extension field $\mathbb{F}_{q^m}$.

Definition 2.4.12 (Subfield subcode). *Consider the field extension $\mathbb{F}_{q^m}/\mathbb{F}_q$. Let $\mathcal{C} \subseteq (\mathbb{F}_{q^m})^n$ be a code over $\mathbb{F}_{q^m}$.*

$$\mathcal{C}|_{\mathbb{F}_q} := \mathcal{C} \cap \mathbb{F}_q^n \tag{2.4.18}$$

is called the subfield subcode *of $\mathcal{C}$ (or the* restriction *of $\mathcal{C}$ to $\mathbb{F}_q$).*

Remark 2.4.13. *Note that the dimension of $\mathcal{C}$ is its dimension as a vector space over $\mathbb{F}_{q^m}$, whereas the dimension of $\mathcal{C}|_{\mathbb{F}_q}$ is the dimension as a vector space over $\mathbb{F}_q$.*[9]

Proposition 2.4.14. *Restricting codes defined over an extension field $\mathbb{F}_{q^m}$ reduces in general the code dimension:*

$$\dim \mathcal{C}|_{\mathbb{F}_q} \leq \dim \mathcal{C}.$$

Proof. The inequality follows from the fact that a basis of $\mathcal{C}|_{\mathbb{F}_q}$ over $\mathbb{F}_q$ is also linearly independent over $\mathbb{F}_{q^m}$ [35]. Indeed, let $(\alpha_i)_{i=1,\dots,n}$ be a $\mathbb{F}_q$-basis of $\mathcal{C}|_{\mathbb{F}_q}$ and $\sum_{i=1}^n a_i\alpha_i = 0$, where $a_i \in \mathbb{F}_{q^m}$. To show is that $a_i = 0$ for all $i = 1, \dots, n$.

Let $(\beta_i)_{j=1,\dots,m}$ be a $\mathbb{F}_q$-basis of $\mathbb{F}_{q^m}$ and $a_i = \sum_{j=1}^m b_j\beta_j$, with $b_j \in \mathbb{F}_q$.

$$0 = \sum_{i=1}^n a_i\alpha_i = \sum_{i=1}^n (\sum_{j=1}^m b_j\beta_j)\alpha_i = \sum_{j=1}^m (\sum_{i=1}^n b_j\alpha_i)\beta_j.$$

Because $(\beta_i)_{j=1,\dots,m}$ is a basis, it follows that $\sum_{i=1}^n b_j\alpha_i = 0$ for all $j = 1, \dots, m$. As the b_j are in $\mathbb{F}_q$ and $(\alpha_i)_{i=1,\dots,n}$ is an $\mathbb{F}_q$-basis, this means $b_j = 0$ for all $j = 1, \dots, m$. Hence, $a_i = 0$ for all $i = 1, \dots, n$. $\square$

Definition 2.4.15 (Alternant code). *The restriction of $GRS_k(\boldsymbol{L}, g)$ over $\mathbb{F}_{q^m}$ to the subfield $\mathbb{F}_q$ is called alternant code over $\mathbb{F}_q$, denoted as*

$$Alt_{k,q}(\boldsymbol{L}, g) := GRS_k(\boldsymbol{L}, g) \cap \mathbb{F}_q^n. \tag{2.4.19}$$

2.4.2 Goppa codes

Definition 2.4.16 (Goppa code). *The restriction of a $GRS_k(\boldsymbol{L}, g)$ code over $\mathbb{F}_{q^m}$ with $\deg g(X) = n - k$ to $\mathbb{F}_q$ is called a q-ary Goppa code. It is a special alternant code and indicated by $GO_q(\boldsymbol{L}, g)$:*

$$GO_q(\boldsymbol{L}, g) := GRS_k(\boldsymbol{L}, g) \cap \mathbb{F}_q^n, \tag{2.4.20}$$

where $\deg g(X) = n - k$.

Remark 2.4.17. *According to Theorem (2.4.11), the Goppa code $GO_q(\boldsymbol{L}, g)$ has the form*

$$\begin{aligned} GO_q(\boldsymbol{L}, g) &= \Big\{ \boldsymbol{c} \in \mathbb{F}_q^n \mid \sum_{i=0}^{n-1} c_i \prod_{\substack{j=0 \\ j\neq i}}^{n-1} (X - L_j) \equiv 0 \mod \langle g \rangle \Big\} \\ &= \Big\{ \boldsymbol{c} \in \mathbb{F}_q^n \mid \sum_{i=0}^{n-1} \frac{c_i}{X - L_i} \equiv 0 \mod \langle g \rangle \Big\}, \end{aligned}$$

where $\deg g(X) = n - k$.

[9]See [35].

Lemma 2.4.18. *Suppose $\mathbb{F}$ an arbitrary field, $L \in \mathbb{F}$, $g(X) \in \mathbb{F}[X]$, $\deg g(X) = t > 0$ and $g(L) \neq 0$. Then there exists a uniquely determinded $h(X) \in \mathbb{F}[X]$, $\deg h(X) < t$, such that*

$$(X - L)h(X) \equiv 1 \mod g(X). \tag{2.4.21}$$

Proof. Indeed, using division by remainder, it follows that $g(L) - g(X) = \alpha(X)(X - L) + r(X)$, where $\alpha(X), r(X) \in \mathbb{F}[X]$. Inserting L on both sides gives $r(X) = 0$. Hence, $X - L$ divides $g(L) - g(X)$ and we can define $h(X) \in \mathbb{F}[X]$ as

$$h(X) := \frac{g(L) - g(X)}{g(L)(X - L)}.$$

Since $\deg g(X) = t$, it is immediate that $\deg h(X) < t$. We apply division with remainder again to the polynomial $(X - L)h(X) - 1$. For degree reasons $\alpha(X)$ has to be a constant:

$$(X - L)h(X) - 1 = \alpha(X)g(X) + r(X) = \alpha g(X) + r(X).$$

Using the definition of $h(X)$ gives $-r(X)g(L) = (\alpha g(L) + 1)g(X)$. Because $\deg g(X) = t$, but $\deg r(X) < t$, we conclude $\alpha g(L) + 1 = 0$ and finally because $g(L) \neq 0$, that $r(X) = 0$. This proves (2.4.21).

For the uniqueness, assume that $h(X), h'(X)$ solve (2.4.21). Thus $g(X)$ divides $(X - L)h(X) - 1$ and $(X - L)h'(X) - 1$ and consequently also their difference $(X - L)(h(X) - h'(X))$. But $g(X)$ and $(X - L)$ are relatively prime, which means that $g(X)$ divides $(h(X) - h'(X))$. Because $\deg(h(X) - h'(X)) < t$ and $\deg g(X) = t$, it follows that $h(X) = h'(X)$. □

Remark 2.4.19. *Under the assumptions of Lemma* (2.4.18) *we write*

$$\frac{1}{X - L} = \frac{g(L) - g(X)}{g(L)(X - L)}, \tag{2.4.22}$$

but this notation is a bit sloppy. $1/(X - L)$ is not a polynomial. Rather, we identify a polynomial $h(X) \in \mathbb{F}[X]$ and its residue class in $\mathbb{F}[X]/\langle g \rangle$. In this sense, $1/(X - L)$ can be considered as polynomial, as long as we work modulo a polynomial $g(X)$, which does not have L as root.

2.4.2.1 Parity check matrix of a Goppa code.

The check matrix of a code that restricts to a Goppa code $GO_q(\boldsymbol{L}, g)$ can be obtained in the following way: [10]

According to remark (2.4.17) and lemma (2.4.18) an $\boldsymbol{c} \in \mathbb{F}_q^n$ is contained in $GO_q(\boldsymbol{L}, g)$ if and only if

$$\sum_{i=0}^{n-1} c_i \frac{g(X) - g(L_i)}{X - L_i} g(L_i)^{-1} = 0 \tag{2.4.23}$$

in $\mathbb{F}_{q^m}[X]/\langle g \rangle$. As

$$\deg \frac{(g(X) - g(L_i))}{(X - L_i)} < \deg g(X),$$

equation (2.4.23) can be considered as an equation in $\mathbb{F}_{q^m}[X]$:

$$\sum_{i=0}^{n-1} c_i \frac{g(X) - g(L_i)}{X - L_i} g(L_i)^{-1} = 0. \tag{2.4.24}$$

[10] [26], pp. 390 – 393, with minor corrections by the author.

Let $g(X)=\sum_{i=0}^{t} g_i X^i$ with $g_t \neq 0$. Then [11]

$$\frac{g(X)-g(L_i)}{X-L_i} = \sum_{j=0}^{t} g_j \frac{X^j - L_i^j}{X-L_i} = \sum_{i=0}^{t} g_j \sum_{u=0}^{j-1} L_i^{j-1-u} X^u.$$

The left hand side of (2.4.24) is therefore, by changing the order of summation,

$$\begin{aligned}\sum_{i=1}^{n} c_i \Big(\sum_{i=0}^{t} g_j \sum_{u=0}^{j-1} L_i^{j-1-u} X^u \Big) g(L_i)^{-1} &= \sum_{i=1}^{n} c_i g(L_i)^{-1} \sum_{u=0}^{t-1} \Big(\sum_{j=u+1}^{t} g_j L_i^{j-1-u} \Big) X^u \\ &= \sum_{u=0}^{t-1} \Big(\sum_{i=1}^{n} c_i g(L_i)^{-1} \sum_{j=u+1}^{t} g_j L_i^{j-1-u} \Big) X^u.\end{aligned}$$

Hence, $\boldsymbol{c} \in GO_q(\boldsymbol{L}, g)$ if and only if for all $0 \le u \le t-1$

$$\begin{aligned} 0 &= \sum_{i=1}^{n} c_i g(L_i)^{-1} \Big(\sum_{j=u+1}^{t} g_j L_i^{j-1-u} \Big) \\ &= \Big[\sum_{j=u+1}^{t} g_j L_1^{j-1-u}, \dots, \sum_{j=u+1}^{t} g_j L_n^{j-1-u} \Big] \begin{bmatrix} c_1/g(L_1) \\ c_2/g(L_2) \\ \vdots \\ c_n/g(L_n) \end{bmatrix} \\ &= \Big[\sum_{j=u+1}^{t} g_j L_1^{j-1-u}, \dots, \sum_{j=u+1}^{t} g_j L_n^{j-1-u} \Big] \begin{bmatrix} 1/g(L_1) & 0 & \cdots & 0 \\ 0 & 1/g(L_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 1/g(L_n) \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \\ &= \Big[\sum_{j=u+1}^{t} g_j L_1^{j-1-u}, \dots, \sum_{j=u+1}^{t} g_j L_n^{j-1-u} \Big] \cdot \boldsymbol{D} \cdot \boldsymbol{c}, \end{aligned}$$

where

$$\boldsymbol{D} = \begin{bmatrix} 1/g(L_1) & 0 & \cdots & 0 \\ 0 & 1/g(L_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 1/g(L_n) \end{bmatrix}. \tag{2.4.25}$$

Furthermore,

$$\begin{aligned} \Big[\sum_{j=u+1}^{t} g_j L_1^{j-1-u}, \dots, \sum_{j=u+1}^{t} g_j L_n^{j-1-u} \Big] &= \Big[g_{u+1} \cdots g_t \; 0 \cdots 0 \Big] \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_1 & L_2 & \cdots & L_n \\ L_1^2 & L_2^2 & \cdots & L_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ L_1^{t-1} & L_2^{t-1} & \cdots & L_n^{t-1} \end{bmatrix} \\ &= \Big[g_{u+1} \cdots g_t \; 0 \cdots 0 \Big] \cdot \boldsymbol{V}, \end{aligned}$$

[11] $(X^j - L_i^j)/(X-L_i) = X^{j-1} + L_i X^{j-2} + \dots + L_i^{j-2} X + L_i^{j-1}$.

where $\boldsymbol{V}$ resembles in structure the Vandermonde matrix,[12]

$$\boldsymbol{V} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_1 & L_2 & \cdots & L_n \\ L_1^2 & L_2^2 & \cdots & L_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ L_1^{t-1} & L_2^{t-1} & \cdots & L_n^{t-1} \end{bmatrix}.$$

Summarizing, this means that (2.4.24) holds if and only if for all $0 \le u \le t-1$

$$\begin{bmatrix} g_{u+1} \cdots g_t \; 0 \cdots 0 \end{bmatrix} \cdot \boldsymbol{V} \cdot \boldsymbol{D} \cdot \boldsymbol{c} = 0,$$

that is, if and only if

$$\begin{bmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ g_{t-2} & g_{t-1} & g_t & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{bmatrix} \cdot \boldsymbol{V} \cdot \boldsymbol{D} \cdot \boldsymbol{c} = \boldsymbol{0}.$$

The canonical parity check matrix $\boldsymbol{H}$ for a GRS code, which restricts to a Goppa code, has therefore the following form:

$$\boldsymbol{H} = \begin{bmatrix} g_t & 0 & 0 & \cdots & 0 \\ g_{t-1} & g_t & 0 & \cdots & 0 \\ g_{t-2} & g_{t-1} & g_t & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_t \end{bmatrix} \begin{bmatrix} 1 & 1 & \dots & 1 \\ L_0 & L_1 & \dots & L_{n-1} \\ L_0^2 & L_1^2 & \dots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & & \dots L_{n-1}^{t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-1} & 0 & \dots & 0 \\ 0 & g(L_1)^{-1} & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-1} \end{bmatrix}. \tag{2.4.26}$$

As the Toeplitz matrix on the left is invertible ($g_t \neq 0$), this is equivalent to

$$\boldsymbol{V}\boldsymbol{D}\boldsymbol{c} = \boldsymbol{0}.$$

Hence, a parity check matrix $\boldsymbol{H}$ takes also the following form:

$$\boldsymbol{H} = \boldsymbol{V}\boldsymbol{D} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ L_0 & L_1 & \dots & L_{n-1} \\ L_0^2 & L_1^2 & \dots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & & \dots L_{n-1}^{t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-1} & 0 & \dots & 0 \\ 0 & g(L_1)^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-1} \end{bmatrix} \tag{2.4.27}$$

With proposition (2.4.14) in mind, the following theorem is not surprising:

Theorem 2.4.20. *The Goppa code $GO_2(\boldsymbol{L}, g)$, where $\deg g(X) = t < n$, has length $n = |\boldsymbol{L}|$, dimension k satisfying*

$$n - mt \le k \le n - t \tag{2.4.28}$$

and minimum distance $d \ge t + 1$.

[12]The Vandermonde matrix is quadratic, and transposed to $\boldsymbol{V}$. Note also the similarity with (2.4.2).

Proof. See [26], Thm. (8.3.2), p. 393. □

We consider now a special class of Goppa codes, so-called *binary Goppa codes*, i.e. Goppa codes with $q = 2$. As it will turn out, they have a fundamental property, which is of crucial importance for the security of cryptosystems based on quasi-dyadic Goppa codes.

2.4.3 Binary Goppa codes

Definition 2.4.21 (Binary Goppa code). *The restriction of a $GRS_k(\boldsymbol{L}, g)$ code over $\mathbb{F}_{2^m}$ with $\deg g(X)$ = $n - k$ = t to $\mathbb{F}_2$ is called a binary Goppa code. It is a special alternant code and indicated by $GO_2(\boldsymbol{L}, g)$:*

$$GO_2(\boldsymbol{L}, g) := GRS_k(\boldsymbol{L}, g) \cap \mathbb{F}_2^n.$$

Because of its importance, let's write down the above definition (2.4.21) explicitly. The ingredients of a binary Goppa code are the following:

(i) A monic polynomial $g(X)$ of degree t = $n - k$,

$$g(X) = \sum_{i=0}^{t} g_i X^i \in \mathbb{F}_{2^m}[X]. \tag{2.4.29}$$

(ii) A tuple $\boldsymbol{L}$ of n pairwise distinct elements

$$\boldsymbol{L} = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_{2^m}^n, \tag{2.4.30}$$

such that

$$g(L_i) \neq 0, \quad i \in \{0, \ldots, n-1\}. \tag{2.4.31}$$

Then the binary Goppa code $GO_2(\boldsymbol{L}, g)$ is

$$GO_2(L, g) = \left\{ \boldsymbol{c} = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_2^n \mid \sum_{i=0}^{n-1} \frac{c_i}{X - L_i} \equiv 0 \mod g(X) \right\}. \tag{2.4.32}$$

Remark 2.4.22. *The elements $L_0, \ldots, L_{n-1} \in \mathbb{F}_{2^m}$ are also called* code support, *whereas $g(X) \in \mathbb{F}_{2^m}[X]$ is referred to as* Goppa polynomial.

$$S_{\boldsymbol{c}}(X) := -\sum_{i=0}^{n-1} \frac{c_i}{g(L_i)} \frac{g(X) - g(L_i)}{X - L_i} \mod g(X) \tag{2.4.33}$$

is known as the syndrome polynomial *of* $\boldsymbol{c}$. *The binary Goppa code $GO_2(L, g)$ consists of all $\boldsymbol{c}$ = $(c_0, \ldots, c_{n-1}) \in \mathbb{F}_2^n$ such that*

$$S_{\boldsymbol{c}}(X) \equiv \sum_{i=0}^{n-1} \frac{c_i}{X - L_i} \equiv 0 \mod g(X). \tag{2.4.34}$$

Consider now a codeword $\boldsymbol{c} = (c_0, \dots, c_{n-1}) \in GO_2(\boldsymbol{L}, g)$ with some Hamming weight $w(\boldsymbol{c}) = \omega$, i.e. $c_{i_1} = \dots = c_{i_\omega} = 1$, whereas the other coordinates are 0 (see [26], with some additions by the author). If

$$f_{\boldsymbol{c}}(X) := \prod_{j=1}^{\omega} (X - L_{i_j}), \tag{2.4.35}$$

then the formal derivative of $f_{\boldsymbol{c}}(X)$ is

$$f_{\boldsymbol{c}}(X)' = \sum_{k=1}^{\omega} \prod_{j \neq k} (X - L_{i_j}), \tag{2.4.36}$$

and further

$$S_{\boldsymbol{c}}(X) f_{\boldsymbol{c}}(X) = f'_{\boldsymbol{c}}(X). \tag{2.4.37}$$

Note that $f_{\boldsymbol{c}}(X)$ and $g(X)$ are relatively prime: because of $g(L_i) \neq 0$, they have no common roots in any extension. Hence,

$$\boldsymbol{c} \in GO_2(\boldsymbol{L}, g) \iff g(X) | S_{\boldsymbol{c}}(X) \iff g(X) | f_{\boldsymbol{c}}(X)'. \tag{2.4.38}$$

Remark 2.4.23. *Because we are in characteristic 2, $f_{\boldsymbol{c}}(X)' = \sum_{i=1}^{n} i f_i X^{i-1}$ contains only even powers of X. Furthermore, it is a perfect square, i.e. $f_{\boldsymbol{c}}(X)' = h(X^2) = k(X)^2$ for some polynomials $h(X), k(X) \in \mathbb{F}_{2^m}[X]$.*

Proof. The map $\mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$, $a \mapsto a^2$ is the Frobenius automorphism on $\mathbb{F}_{2^m}$. Therefore, each element $a \in \mathbb{F}_{2^m}$ has a unique square root. Given $h(X) = \sum a_k X^{2k} \in \mathbb{F}_{2^m}[X]$, define $k(X) = \sum \sqrt{a_k} X^k$. Then $k(X)^2 = h(X)$. □

Lemma 2.4.24. *Let $\hat{g}(X)$ the perfect square with smallest degree, which is divisible by $g(X)$, i.e. $\hat{g}(X) = \alpha(X) g(X)$ for some $\alpha(X) \in \mathbb{F}_{2^m}[X]$. Then*

$$\hat{g}(X) | f_{\boldsymbol{c}}(X)' \iff g(X) | f_{\boldsymbol{c}}(X)'. \tag{2.4.39}$$

Proof. $\Rightarrow$: $f_{\boldsymbol{c}}(X)' = \gamma(X)\hat{g}(X) = \gamma(X)\alpha(X)g(X)$ for some $\gamma(X) \in \mathbb{F}_{2^m}[X]$. $\Leftarrow$: Let $\hat{g}(X) = \alpha(X)g(X)$, $\hat{g}(X)$ a perfect square. Because $g(X)|f_{\boldsymbol{c}}(X)'$, it follows that $\deg \hat{g}(X) \leq \deg f_{\boldsymbol{c}}(X)'$ (because $f_{\boldsymbol{c}}(X)'$ is a perfect square according to (2.4.23) and the minimality of $\hat{g}(X)$). Therefore $f_{\boldsymbol{c}}(X)' = \beta(X)\hat{g}(X) + r(X) = \beta(X)\alpha(X)g(X) + r(X)$. By assumption, we have $r(X) = 0$, which finally gives $\hat{g}(X)|f_{\boldsymbol{c}}(X)'$. □

The next theorem will summarize the latest results:

Theorem 2.4.25. *Let $GO_2(\boldsymbol{L}, g)$ be a binary Goppa code. If $\hat{g}(X)$ is the polynomial of smallest degree that is a perfect square and is divisible by $g(X)$, then*

$$GO_2(\boldsymbol{L}, g) = GO_2(\boldsymbol{L}, \hat{g}). \tag{2.4.40}$$

In particular, $GO_2(\boldsymbol{L}, g)$ has a minimum distance at least $\deg \hat{g}(X) + 1$.

Proof. Let $\boldsymbol{c} \in GO_2(\boldsymbol{L}, g)$.

$$\begin{aligned} \boldsymbol{c} \in GO_2(\boldsymbol{L}, g) &\overset{\text{Def.}}{\iff} S_{\boldsymbol{c}}(X) \equiv 0 \mod g(X) \\ &\overset{(2.4.38)}{\iff} g(X) | f_{\boldsymbol{c}}(X)' \\ &\overset{(2.4.39)}{\iff} \hat{g}(X) | f_{\boldsymbol{c}}(X)' \\ &\overset{(2.4.38)}{\iff} S_{\boldsymbol{c}}(X) \equiv 0 \mod \hat{g}(X) \\ &\overset{\text{Def.}}{\iff} \boldsymbol{c} \in GO_2(\boldsymbol{L}, \hat{g}) \end{aligned}$$

For the statement about the minimum distance, see (2.4.20). □

Corollary 2.4.26. [13] *Let $GO_2(\boldsymbol{L}, g)$ be a binary Goppa code and suppose that the Goppa polynomial $g(X)$ has no multiple roots in any extension field. Then*

$$GO_2(\boldsymbol{L}, g) = GO_2(\boldsymbol{L}, g^2). \tag{2.4.41}$$

In particular, the minimum distance of $GO_2(\boldsymbol{L}, g)$ is at least $2 \deg g(X) + 1$. Hence, $GO_2(\boldsymbol{L}, g)$ can correct at least $\deg g(X)$ errors.

Definition 2.4.27 (Irreducible Goppa code, separable Goppa code).

(i) *A binary Goppa code with a Goppa polynomial irreducible over $\mathbb{F}_q$ is called a* irreducible *Goppa code.*

(ii) *A binary Goppa code whose Goppa polynomial has no multiple roots in any extension field is called a* separable *Goppa code.*

Remark 2.4.28. *The crucial meaning of Cor.* (2.4.26) *is the fact that separable binary Goppa codes have two different representations with* different *error correcting capabilities. In other words, an alternant decoder based on $g(X)^2$ can correct twice as much errors as a decoder based on $g(X)$.*

2.4.4 Examples for binary Goppa codes

2.4.4.1 Example Let [14] $g(X) = x^2 + x + 1$ and the support $\boldsymbol{L} = \{0, 1, \omega, \dots, \omega^6\} = \mathbb{F}_8$. Thus $m = 3, t = 2, n = 8$ and ω is a primitive element of $\mathbb{F}_8$. The zeros of $g(X)$ are not in $\mathbb{F}_8$, but in $\mathbb{F}_4$. If we write $\mathbb{F}_4$ as $\mathbb{F}_4 = \{0, 1, z, z^2\}$, then the zeros are z and z^2, as is easily calculated.

The parity check matrix $\boldsymbol{H}'$ of the Goppa code $GO_2(\boldsymbol{L}, g)$ is obtained from the matrix

$$\begin{aligned} \boldsymbol{H} &= \begin{bmatrix} 1/g(0) & 1/g(1) & 1/g(\omega) & \dots & 1/g(\omega^6) \\ 0/g(0) & 1/g(1) & \omega/g(\omega) & \dots & \omega^6/g(\omega^6) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 1 & \omega^2 & \omega^4 & \omega^2 & \omega & \omega & \omega^4 \\ 0 & 1 & \omega^3 & \omega^6 & \omega^5 & \omega^5 & \omega^6 & \omega^3 \end{bmatrix}, \end{aligned}$$

[13] Nicola Sendrier calls it the binary miracle. It will be the key of the quasi-dyadic cryptosystem, as will be shown below.
[14] [26], p. 394.

which can be seen by using a field table of $\mathbb{F}_8$. As the Goppa code is a subfield subcode, we need to express the entries of $\boldsymbol{H}$ as vectors $\boldsymbol{a} = (a_0, a_1, a_2)$ with $a_i \in \mathbb{F}_2$ ([20], Ch. 7, p. 207).

$$\boldsymbol{H}' = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}.$$

Since the rank of $\boldsymbol{H}'$ is 6, $\dim GO_2(\boldsymbol{L}, g) = 2$ and with some calculation it follows that

$$GO_2(\boldsymbol{L}, g) = \{00000000,\ 11110100,\ 11001011,\ 00111111\},$$

hence the Goppa code has a minimum distance of 5.

2.4.4.2 Example Let [15] $g(X) = x^2 + 1$ and the support $\boldsymbol{L} = \{0, \omega, \omega^2, \omega^3, \omega^5, \omega^6\} \subset \mathbb{F}_8$. Thus $m = 3, t = 2, n = 6$ and ω is again a primitive element of $\mathbb{F}_8$. As char $\mathbb{F}_8 = 2$, it holds that $g(X) = (X+1)^2$. Using corollary (2.4.26) we see that $GO_2(\boldsymbol{L}, g) = GO_2(\boldsymbol{L}, X+1)$. By theorem (2.4.20), we conclude the dimension k to be greater than $n - mt = 6 - 3 \cdot 1 = 3$. Furthermore,

$$\begin{aligned} \boldsymbol{H} &= \begin{bmatrix} \frac{1}{0+1} & \frac{1}{\omega+1} & \frac{1}{\omega^2+1} & \frac{1}{\omega^3+1} & \frac{1}{\omega^4+1} & \frac{1}{\omega^5+1} \end{bmatrix} \\ &= \begin{bmatrix} 1 & \omega^4 & \omega & \omega^6 & \omega^3 & \omega^5 \end{bmatrix}, \end{aligned}$$

and thus a parity check matrix for $GO_2(\boldsymbol{L}, g)$ is

$$\boldsymbol{H}' = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix},$$

and we see that $GO_2(\boldsymbol{L}, g)$ has minimum distance 3. □

2.4.5 Parity check matrix generated by $g(X)^2$ in case of a binary, separable Goppa codes

Let $\boldsymbol{Z} = \{z_0, \ldots, z_{t-1}\} \subset \mathbb{F}_{2^m}$ and $\boldsymbol{L} = \{L_0, \ldots, L_{n-1}\} \subset \mathbb{F}_{2^m}$, where all the members of $\boldsymbol{L}$ are distinct. Let $\boldsymbol{Z} \cap \boldsymbol{L} = \emptyset$ and define the Goppa polynomial $g(X)$ without multiple roots as

$$g(X) = (X - z_0) \cdots (X - z_{t-1}) \in \mathbb{F}_{2^m}[X],$$

where $t < n$.

Using (2.4.27), we see that a parity check matrix $\boldsymbol{H}'$ for the Goppa code $GO_2(\boldsymbol{L}, g^2)$ is given as:

[15] [26], p. 395.

$$\boldsymbol{H}' = \boldsymbol{V}\boldsymbol{D} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_0 & L_1 & \cdots & L_{n-1} \\ L_0^2 & L_1^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & \cdots & L_{n-1}^{t-1} \\ L_0^{t} & L_1^{t} & \cdots & L_{n-1}^{t} \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{2t-1} & L_1^{2t-1} & \cdots & L_{n-1}^{2t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-2} & 0 & \cdots & 0 \\ 0 & g(L_1)^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-2} \end{bmatrix}. \tag{2.4.42}$$

As in (2.4.26), it is also possible to add the corresponding Toeplitz matrix on the left, of course[16].

2.4.6 Parity check matrix generated by $g(X)^2$ in Tzeng-Zimmermann form

To derive another form (see [36]) for the parity check matrix $\boldsymbol{H}'$ for the Goppa code $GO_2(\boldsymbol{L}, g^2)$ generated by

$$g(X)^2 = (X - z_0)^2 \cdots (X - z_{t-1})^2,$$

apply (2.4.27) to $g_l(X) = (X - z_l)^2$, where $0 \le l \le t-1$. Because it is allowed to add a row of a parity check matrix, multiplied by a common factor, to any other row of the matrix, we have for $g_l(X)$ as a parity check matrix $\boldsymbol{H}_l$:

$$\begin{aligned} \boldsymbol{H}_l \;&= \begin{bmatrix} (L_0 - z_l)^{-2} & (L_1 - z_l)^{-2} & \cdots & (L_{n-1} - z_l)^{-2} \\ L_0(L_0 - z_l)^{-2} & L_1(L_1 - z_l)^{-2} & \cdots & L_{n-1}(L_{n-1} - z_l)^{-2} \end{bmatrix} \\ &\rightsquigarrow \begin{bmatrix} (L_0 - z_l)^{-2} & (L_1 - z_l)^{-2} & \cdots & (L_{n-1} - z_l)^{-2} \\ (L_0 - z_l)(L_0 - z_l)^{-2} & (L_1 - z_l)(L_1 - z_l)^{-2} & \cdots & (L_{n-1} - z_l)(L_{n-1} - z_l)^{-2} \end{bmatrix} \\ &\rightsquigarrow \begin{bmatrix} (L_0 - z_l)^{-2} & (L_1 - z_l)^{-2} & \cdots & (L_{n-1} - z_l)^{-2} \\ (L_0 - z_l)^{-1} & (L_1 - z_l)^{-1} & \cdots & (L_{n-1} - z_l)^{-1} \end{bmatrix} \\ &\rightsquigarrow \begin{bmatrix} (L_0 - z_l)^{-1} & (L_1 - z_l)^{-1} & \cdots & (L_{n-1} - z_l)^{-1} \\ (L_0 - z_l)^{-2} & (L_1 - z_l)^{-2} & \cdots & (L_{n-1} - z_l)^{-2} \end{bmatrix} \end{aligned}$$

It is easily seen that the Goppa code $GO_2(\boldsymbol{L}, g^2)$ is the intersection of the Goppa codes $GO_2(\boldsymbol{L}, g_l)$,

$$GO_2(\boldsymbol{L}, g^2) = \bigcap_{l=0}^{t-1} GO_2(\boldsymbol{L}, g_l).$$

Hence, we have

[16]These two matrices $\boldsymbol{H}$ and $\boldsymbol{H}'$ are leading to slightly different key equations, however. For the decoder, we will use the Toeplitz-form $\boldsymbol{H}$.

$$\boldsymbol{H}' = \begin{bmatrix} \boldsymbol{H}_0 \\ \boldsymbol{H}_1 \\ \vdots \\ \boldsymbol{H}_{t-1} \end{bmatrix} = \begin{bmatrix} (L_0 - z_0)^{-1} & (L_1 - z_0)^{-1} & \cdots & (L_{n-1} - z_0)^{-1} \\ (L_0 - z_0)^{-2} & (L_1 - z_0)^{-2} & \cdots & (L_{n-1} - z_0)^{-2} \\ (L_0 - z_1)^{-1} & (L_1 - z_1)^{-1} & \cdots & (L_{n-1} - z_1)^{-1} \\ (L_0 - z_1)^{-2} & (L_1 - z_1)^{-2} & \cdots & (L_{n-1} - z_1)^{-2} \\ \vdots & \vdots & \vdots & \\ (L_0 - z_{t-1})^{-1} & (L_1 - z_{t-1})^{-1} & \cdots & (L_{n-1} - z_{t-1})^{-1} \\ (L_0 - z_{t-1})^{-2} & (L_1 - z_{t-1})^{-2} & \cdots & (L_{n-1} - z_{t-1})^{-2} \end{bmatrix}. \tag{2.4.43}$$

More general, we have:

Theorem 2.4.29. *The Goppa code generated by a monic polynomial $g(X)$ without multiple zeros $g(X) = (X - z_0) \dots (X - z_{t-1})$ admits a parity-check matrix of the form $\boldsymbol{H} = C(\boldsymbol{z}, \boldsymbol{L})$, i.e. $H_{ij} = 1/z_i - L_j$, $0 \le i < t, 0 \le j < n$.*

Proof. See [20], Ch. 12, p. 345, Problem 5. or [36], p. 713. □

Remark 2.4.30. *Goppa codes in Tzeng-Zimmermann form turn out to be special cases of so-called* Srivastava codes. *See [20] (Chap. 12, §6) or [25].*

2.4.7 Goppa codes in Cauchy and dyadic form

Definition 2.4.31 (Dyadic matrix, quasi-dyadic matrix, signature [3]**).** *Let $r = 2^k$ for some $k \in \mathbb{N}$.*

(i) *Given a ring $\mathcal{R}$ and a vector $\boldsymbol{h} = (h_0, \dots, h_{r-1}) \in \mathcal{R}$, the dyadic matrix $\boldsymbol{\Delta(h)} \in \mathcal{R}^{r \times r}$ is the symmetric matrix with components $\Delta_{ij} = h_{i \oplus j}$, where $\oplus$ stands for the bitwise exclusive-or. Such a matrix is said to have order k. The sequence $\boldsymbol{h}$ is called its* signature. *The set of dyadic $r \times r$ matrices over $\mathcal{R}$ is denoted $\boldsymbol{\Delta}(\mathcal{R}^{r \times r})$.*

(ii) *A* quasi-dyadic *matrix is a (possibly non-dyadic) matrix, whose elements are dyadic submatrices, i.e. an element of $\boldsymbol{H}(\mathcal{R}^r)$.*

Remark 2.4.32. *It is easy to check that the signature of a dyadic matrix is nothing more than its first row (column). By the definition of dyadic matrices, the signature is enough to rebuild the whole matrix. Hence, such a matrix allows a very compact represenation.*

We give a visualization of such a matrix for $r = 3$:

$$H(\boldsymbol{h}) := (h_{i \oplus j}) := \begin{bmatrix} A & B & C & D & E & F & G & H \\ B & A & D & C & F & E & H & G \\ C & D & A & B & G & H & E & F \\ D & C & B & A & H & G & F & E \\ E & F & G & H & A & B & C & D \\ F & E & H & G & B & A & D & C \\ G & H & E & F & C & D & A & B \\ H & G & F & E & D & C & B & A \end{bmatrix}$$

In the sequel, we will give the definition of a so-called *Cauchy matrix* and a theorem about Goppa codes having a Cauchy matrix as generating matrix. Afterwards, we will see how to connect Cauchy matrices and dyadic matrices.

Definition 2.4.33 (Cauchy matrix). *Let $\boldsymbol{z} = (z_0, \dots, z_{t-1}) \in \mathbb{F}_q^t$ and $\boldsymbol{L} = (L_0, \dots, L_{n-1}) \in \mathbb{F}_q^n$ two disjoint sequences of distinct elements.*

The Cauchy matrix *$\boldsymbol{C}(\boldsymbol{z}, \boldsymbol{L})$ is the matrix with elements $C_{ij} = \dfrac{1}{z_i - L_j} \in \mathbb{F}_q$:*

$$\boldsymbol{C}(\boldsymbol{z}, \boldsymbol{L}) = (C_{ij}) = \begin{bmatrix} \frac{1}{z_0 - L_0} & \frac{1}{z_0 - L_1} & \cdots & \frac{1}{z_0 - L_{n-1}} \\ \frac{1}{z_1 - L_0} & \frac{1}{z_1 - L_1} & \cdots & \frac{1}{z_1 - L_{n-1}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{z_{t-1} - L_0} & \frac{1}{z_{t-1} - L_1} & \cdots & \frac{1}{z_{t-1} - L_{n-1}} \end{bmatrix} \in \mathbb{F}_q^{t \times n} \tag{2.4.44}$$

The question now is if there are Cauchy matrices in dyadic form. The answer is yes, thereby having the additional property of a very compact representation.

Theorem 2.4.34 (Cauchy matrices in dyadic form [23]). *Let $\boldsymbol{H} \in \mathbb{F}_q^{n \times n}$ with $n > 1$ be simultaneously a dyadic matrix $\boldsymbol{H} = \boldsymbol{\Delta}(\boldsymbol{h})$ for some $\boldsymbol{h} \in \mathbb{F}_q^n$ and a Cauchy matrix $\boldsymbol{H} = \boldsymbol{C}(\boldsymbol{z}, \boldsymbol{L})$ for two disjoint sequences $\boldsymbol{z} \in \mathbb{F}_q^n$ and $\boldsymbol{L} \in \mathbb{F}_q^n$ of distinct elements. Then $\mathbb{F}_q$ is a binary field, $\boldsymbol{h}$ satisfies*

$$\frac{1}{h_{i \oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}, \tag{2.4.45}$$

and $z_i = 1/h_i + \omega$, $L_j = 1/h_j + 1/h_0 + \omega$ for some $\omega \in \mathbb{F}_q$.

Remark 2.4.35. *Dyadic matrices allow a very compact representation. Parity check matrices in Cauchy form can be generated by Goppa polynomials of the form*

$$g(X) = (X - z_0)(X - z_1) \cdots (X - z_{t-1}).$$

Due to theorem (2.4.34)*, the intersection between dyadic matrices and Cauchy matrices is not empty: we have to use values z_i and L_j satisfying equation* (2.4.45).

As noted in [23], a cryptosystem based on a parity-check matrix in Cauchy form would immediately reveal the Goppa polynomial as there would be an overdefined linear system $z_i - L_j = 1/H_{ij}$ consisting of tn equations in $t + n$ unknowns. Hence, additional techniques will have to be applied, in particular the use of quasi-dyadic subfield subcodes as the most important. We will address some of these points in the implementation section.

2.4.8 The fast Walsh-Hadamard transform and the dyadic convolution

By saving only the necessary signatures, dyadic and quasi-dyadic matrices allow a substantial reduction of the public key size. Instead of keeping a whole generator matrix $\boldsymbol{G} \in \mathbb{F}_q^{k \times n}$, for a purely dyadic code we would only need its first row. The encryption of a message $\boldsymbol{u} \in \mathbb{F}_q^k$ into the codeword $\boldsymbol{c} \in \mathbb{F}_q^n$ is done by the vector-matrix product $\boldsymbol{c} = \boldsymbol{u}\boldsymbol{G}$.

This raises the question of how to perform $\boldsymbol{uG}$ efficiently in terms of time and space. If it would be necessary to expand the signature $\boldsymbol{h}$ into the matrix $\boldsymbol{G}$, then the scheme would be less useful. As it turns out, $\boldsymbol{uG}$ can be done without expanding the signature $\boldsymbol{h}$ using the Walsh-Hadamard transform [3, 15]

The following is taken from [3]. We make the same assumptions as for (2.4.31), in particular we always assume $r = 2^k$.

Definition 2.4.36 (Dyadic convolution). *The dyadic convolution of two vectors $\boldsymbol{u}, \boldsymbol{v} \in \mathcal{R}$ is the unique vector denoted by $\boldsymbol{u} \vartriangle \boldsymbol{v} \in \mathcal{R}$ such that $\boldsymbol{\Delta}(\boldsymbol{u} \vartriangle \boldsymbol{v}) = \boldsymbol{\Delta}(\boldsymbol{u})\boldsymbol{\Delta}(\boldsymbol{v})$. The dyadic inverse of a vector $\boldsymbol{u} \in \mathcal{R}$, which exists whenever* $\det \boldsymbol{\Delta}(\boldsymbol{u}) \neq 0$, *is the vector $\boldsymbol{u}^{\vartriangle_1}$ such that $\boldsymbol{\Delta}(\boldsymbol{u}^{\vartriangle_1}) = \boldsymbol{\Delta}(\boldsymbol{u})^{-1}$.*

Definition 2.4.37 (Sylvester-Hadamard matrix). *Let $\mathbb{F}$ be a field with characteristic* $\operatorname{char}(\mathbb{F}) \neq 2$. *The Sylvester-Hadamard matrix $\boldsymbol{H}_k \in \mathbb{F}^r$ is recursively defined as*

$$\begin{aligned} \boldsymbol{H}_0 &= \begin{bmatrix} 1 \end{bmatrix} \\ \boldsymbol{H}_k &= \begin{bmatrix} \boldsymbol{H}_{k-1} & \boldsymbol{H}_{k-1} \\ \boldsymbol{H}_{k-1} & -\boldsymbol{H}_{k-1} \end{bmatrix}, \quad k > 0. \end{aligned} \tag{2.4.46}$$

Remark 2.4.38. *It is easily seen that*

$$\begin{aligned} \boldsymbol{H}_0^{-1} &= \begin{bmatrix} 1 \end{bmatrix} \\ \boldsymbol{H}_k^{-1} &= \frac{1}{2} \begin{bmatrix} \boldsymbol{H}_{k-1}^{-1} & \boldsymbol{H}_{k-1}^{-1} \\ \boldsymbol{H}_{k-1}^{-1} & -\boldsymbol{H}_{k-1}^{-1} \end{bmatrix}, \quad k > 0. \end{aligned} \tag{2.4.47}$$

Lemma 2.4.39. *Let $\mathbb{F}$ be a field with characteristic* $\operatorname{char}(\mathbb{F}) \neq 2$. *If $\boldsymbol{M} \in \mathbb{F}^{r \times r}$ is dyadic, then $\boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k$ is diagonal.*

Proof. We describe the proof as given in [3] for ease of reference. The assertion is obviously true for $k = 0$. Let $k > 0$, and write

$$\boldsymbol{M} = \begin{bmatrix} \boldsymbol{AB} \\ \boldsymbol{BA} \end{bmatrix},$$

where $\boldsymbol{A}$ and $\boldsymbol{B}$ are dyadic. It follows that

$$\begin{aligned} \boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k &= \frac{1}{2} \begin{bmatrix} \boldsymbol{H}_{k-1}^{-1} & \boldsymbol{H}_{k-1}^{-1} \\ \boldsymbol{H}_{k-1}^{-1} & -\boldsymbol{H}_{k-1}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{AB} \\ \boldsymbol{BA} \end{bmatrix} \begin{bmatrix} \boldsymbol{H}_{k-1} & \boldsymbol{H}_{k-1} \\ \boldsymbol{H}_{k-1} & -\boldsymbol{H}_{k-1} \end{bmatrix} \\ &= \begin{bmatrix} \boldsymbol{H}_{k-1}\boldsymbol{M}_+\boldsymbol{H}_k & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_-\boldsymbol{H}_{k-1} \end{bmatrix}, \end{aligned} \tag{2.4.48}$$

and because both $\boldsymbol{M}_+ = \boldsymbol{A} + \boldsymbol{B}$ and $\boldsymbol{M}_- = \boldsymbol{A} - \boldsymbol{B}$ are dyadic, $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_+\boldsymbol{H}_{k-1}$ and $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_-\boldsymbol{H}_{k-1}$ are diagonal by induction, so is $\boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k$. □

Remark 2.4.40. *Lemma* (2.4.39) *suggests an efficient way to multiply two dyadic matrices $\boldsymbol{M}$ and $\boldsymbol{N}$ using the diagonal forms $\boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k$ and $\boldsymbol{H}_k^{-1}\boldsymbol{N}\boldsymbol{H}_k$ as*

$$(\boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k)(\boldsymbol{H}_k^{-1}\boldsymbol{N}\boldsymbol{H}_k) = \boldsymbol{H}_k^{-1}(\boldsymbol{M}\boldsymbol{N})\boldsymbol{H}_k.$$

However, it is not necessary to carry out $\boldsymbol{H}_k^{-1}\boldsymbol{M}\boldsymbol{H}_k$ completely in order to diagonalize $\boldsymbol{M}$, as the following lemma (2.4.41) *shows.*

Lemma 2.4.41. *Let* $\mathbb{F}$ *be a field with characteristic* $\operatorname{char}(\mathbb{F}) \neq 2$. *The diagonal form of a dyadic matrix* $\boldsymbol{M} \in \mathbb{F}^{r\times r}$ *is the first line of* $\boldsymbol{M}\boldsymbol{H}_k$. *In other words,* $\boldsymbol{H}_k^{-1}\boldsymbol{\Delta}(\boldsymbol{h})\boldsymbol{H}_k = \operatorname{diag}(\boldsymbol{h}\boldsymbol{H}_k)$.

Proof. The statement is true for $k = 0$. Therefore, let $k > 0$, $\boldsymbol{M} = \left[\begin{smallmatrix}\boldsymbol{A}\boldsymbol{B}\\ \boldsymbol{B}\boldsymbol{A}\end{smallmatrix}\right]$, $\boldsymbol{M}_+ = \boldsymbol{A} + \boldsymbol{B}$ and $\boldsymbol{M}_- = \boldsymbol{A} - \boldsymbol{B}$ as before. By (2.4.48) the diagonal of $\boldsymbol{H}_k^{-1}(\boldsymbol{M})\boldsymbol{H}_k$ is the concatenation of the diagonals of $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_+\boldsymbol{H}_{k-1}$ and $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_-\boldsymbol{H}_{k-1}$. Similarly, since

$$\boldsymbol{M}\boldsymbol{H}_k = \begin{bmatrix}\boldsymbol{A} & \boldsymbol{B}\\ \boldsymbol{B} & \boldsymbol{A}\end{bmatrix}\begin{bmatrix}\boldsymbol{H}_{k-1} & \boldsymbol{H}_{k-1}\\ \boldsymbol{H}_{k-1} & -\boldsymbol{H}_{k-1}\end{bmatrix} = \begin{bmatrix}\boldsymbol{M}_+\boldsymbol{H}_{k-1} & \boldsymbol{M}_-\boldsymbol{H}_{k-1}\\ \boldsymbol{M}_+\boldsymbol{H}_{k-1} & -\boldsymbol{M}_-\boldsymbol{H}_{k-1}\end{bmatrix},$$

the first line of $\boldsymbol{M}\boldsymbol{H}_k$ is the concatenation of the first lines of $\boldsymbol{M}_+\boldsymbol{H}_{k-1}$ and $\boldsymbol{M}_-\boldsymbol{H}_{k-1}$, which by induction are the diagonals of $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_+\boldsymbol{H}_{k-1}$ and $\boldsymbol{H}_{k-1}^{-1}\boldsymbol{M}_-\boldsymbol{H}_{k-1}$, respectively. □

Corollary 2.4.42. *Computing* $\boldsymbol{w}$ *such that* $\boldsymbol{\Delta}(\boldsymbol{u})\boldsymbol{\Delta}(\boldsymbol{v}) = \boldsymbol{\Delta}\boldsymbol{w}$ *involves only three multiplications of vectors by Sylvester-Hadamard matrices.*

Proof. By Lemma (2.4.41),

$$\begin{aligned}\operatorname{diag}(\boldsymbol{u}\boldsymbol{H}_k)\operatorname{diag}(\boldsymbol{v}\boldsymbol{H}_k) &= (\boldsymbol{H}_k^{-1}\boldsymbol{\Delta}(\boldsymbol{u})\boldsymbol{H}_k)(\boldsymbol{H}_k^{-1}\boldsymbol{\Delta}(\boldsymbol{v})\boldsymbol{H}_k)\\ &= \boldsymbol{H}_k^{-1}(\boldsymbol{\Delta}(\boldsymbol{u})\boldsymbol{\Delta}(\boldsymbol{v}))\boldsymbol{H}_k\\ &= \boldsymbol{H}_k^{-1}\boldsymbol{\Delta}(\boldsymbol{w})\boldsymbol{H}_k\\ &= \operatorname{diag}(\boldsymbol{w}\boldsymbol{H}_k).\end{aligned}$$

Now retrieve $\boldsymbol{w}$ from the vector $\boldsymbol{z} = \boldsymbol{w}\boldsymbol{H}_k$ as $\boldsymbol{w} = \boldsymbol{z}\boldsymbol{H}_k^{-1} = 2^{-k}\boldsymbol{z}\boldsymbol{H}_k$. □

Remark 2.4.43. *The structure of Sylvester-Hadamard matrices leads to an efficient algorithm to compute* $\boldsymbol{u}\boldsymbol{H}_k$ *for* $\boldsymbol{u} \in \mathbb{F}^r$, *which is known as the fast Walsh-Hadamard transform.*

Let $[\boldsymbol{u}_0, \boldsymbol{u}_1]$ *be the two halves of* $\boldsymbol{u}$. *Then*

$$\boldsymbol{u}\boldsymbol{H}_k = [\boldsymbol{u}_{\boldsymbol{0}}, \boldsymbol{u}_1]\begin{bmatrix}\boldsymbol{H}_{k-1} & \boldsymbol{H}_{k-1}\\ \boldsymbol{H}_{k-1} & -\boldsymbol{H}_{k-1}\end{bmatrix} = [(\boldsymbol{u}_0 + \boldsymbol{u}_1)\boldsymbol{H}_{k-1}, (\boldsymbol{u}_0 - \boldsymbol{u}_1)\boldsymbol{H}_{k-1}], \tag{2.4.49}$$

which is a recursive algorithm of complexity $\Theta(r \log r)$.

The reader may have noticed that the general assumption for the conclusions above was always a field not having characteristic 2. At first glance, this fact seems to exclude the Walsh-Hadamard transform as an efficient tool in the context of binary Goppa codes, but there is a solution for the problem. It consists of lifting the algorithm to characteristic 0, namely from $\mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$ to $\mathbb{Z}$. What actually happens is that all the bits involved in the calculations are seens as integers. By the end of the computation, the result will be reduced modulo 2, see section (8.1) for details.

Another approach may be to compute the product $\boldsymbol{u}\boldsymbol{H}_k$ directly using only the signature of $\boldsymbol{H}_k$ for proper parameter settings. That means that during the multiplication process, the current row of $\boldsymbol{H}_k$ is computed on the fly. It might be good choice due to its small memory footprint, see section (8.1) as well.

Part V
Code-based cryptosystems

3 The classical McEliece cryptosystem

The original McEliece cryptosystem was introduced by R. McEliece in 1978 using irreducible binary Goppa codes [21].

3.1 Setup

The system parameters are $k, m, n, t \in \mathbb{N}$, where $k < n$, $t \ll n$. For the public and private keys, generate the following matrices:

- ▷ A secret generator matrix $\boldsymbol{G'} \in \mathbb{F}_{2^m}^{k\times n}$ of an irreducible binary $[n, k, d]$ Goppa code $GO_2(\boldsymbol{L}, g)$ with minimum distance $d \geq 2t+1$ and a fixed, public support $\boldsymbol{L}$.
- ▷ A secret random binary non-singular matrix $\boldsymbol{S} \in \mathbb{F}_{2^m}^{k\times k}$.
- ▷ A secret random permutation matrix $\boldsymbol{P} \in \mathbb{F}_{2^m}^{n\times n}$.
- ▷ A public generator matrix $\boldsymbol{SG'P} = \boldsymbol{G} \in \mathbb{F}_{2^m}^{k\times n}$ for a code equivalent to $GO_2(\boldsymbol{L}, g)$.

The pair $(\boldsymbol{G}, t)$ is the *public key*, whereas the *private key* consists of the triple $(\boldsymbol{S}, \mathcal{D}_{GO_2(\boldsymbol{L},g)}, \boldsymbol{P})$. Here $\mathcal{D}_{GO_2(\boldsymbol{L},g)}$ is an efficient decoding algorithm for $GO_2(\boldsymbol{L}, g)$ [17].

3.2 Encryption

To encrypt a plaintext $\boldsymbol{m} \in \mathbb{F}_2^k$, choose a vector $\boldsymbol{z} \in \mathbb{F}_2^n$ of weight t randomly and compute the ciphertext as follows:

$$\boldsymbol{c} = \boldsymbol{mG} \oplus \boldsymbol{z}.$$

3.3 Decryption

To decrypt a ciphertext $\boldsymbol{c} \in \mathbb{F}_2^n$ calculate

$$\boldsymbol{cP^{-1}} = \boldsymbol{mSG'} \oplus \boldsymbol{zP^{-1}}.$$

Apply the decoding algorithm $\mathcal{D}_{GO_2(\boldsymbol{L},g)}$ for $GO_2(\boldsymbol{L}, g)$. Since $\boldsymbol{cP^{-1}}$ has a Hamming distance of t to $GO_2(\boldsymbol{L}, g)$, we obtain the codeword

$$\boldsymbol{mSG'} = \mathcal{D}_{GO_2(\boldsymbol{L},g)}(\boldsymbol{cP^{-1}})$$

Let $J \subseteq \{1, \ldots, n\}$ be a set such that $\boldsymbol{G}_J$ is invertible. Then we compute the plaintext $\boldsymbol{m} = \boldsymbol{mSG'}_J(\boldsymbol{G'}_J)^{-1}\boldsymbol{S}^{-1}$.

[17] Typical choices for the parameters are $2 \leq t \leq (2^m-1)/m$, $m \in \{10, 11, 12\}$ and $mt+1 \leq n \leq 2^m$ [8].

4 A modern McEliece cryptosystem

The private key for the McEliece scheme as described above consists of the Goppa polynomial $g(X)$, $\boldsymbol{S}$ and $\boldsymbol{P}$. The support $\boldsymbol{L}$ is fixed and public. From an implementation point of view, it is more suitable to work with a permuted, secret support and to transform the matrix $\boldsymbol{G'}$ to systematic form. As mentioned in [11], Section (3.1), $\boldsymbol{S}$ has no cryptographic function in hiding the secret polynomial $g(X)$. The net effect is therefore that we can get rid of the matrices $\boldsymbol{S}$ and $\boldsymbol{P}$ [2].

4.1 Setup

The system parameters are as in section (3), except the support $\boldsymbol{L}$ is secret and permuted.

For the public and private keys, generate the following matrices:

- ▷ A secret generator matrix $\boldsymbol{G'} \in \mathbb{F}_{2^m}^{k \times n}$ of an irreducible binary $[n, k, d]$ Goppa code $GO_2(\boldsymbol{L}, g)$ with minimum distance $d \geq 2t + 1$ and a secret, permuted support $\boldsymbol{L}$.
- ▷ A public generator matrix $\boldsymbol{G} \in \mathbb{F}_{2^m}^{k \times n}$ for a code equivalent to $GO_2(\boldsymbol{L}, g)$ in systematic form.

Set the *private key* $\mathrm{K}_{\mathrm{priv}}$ = $(\boldsymbol{L}, \boldsymbol{g})$ and the *public key* $\mathrm{K}_{\mathrm{pub}}$ = $(\boldsymbol{G}, t)$.

4.2 Encryption

To encrypt a plaintext $\boldsymbol{m} \in \mathbb{F}_2^k$, we proceed as in the classical case, choosing a vector $\boldsymbol{z} \in \mathbb{F}_2^n$ of weight t randomly and computing the ciphertext:

$$\boldsymbol{c} = \boldsymbol{mG} \oplus \boldsymbol{z}.$$

4.3 Decryption

The decryption of a ciphertext $\boldsymbol{c} \in \mathbb{F}_2^n$ is considerable simpler than in the classical case: we just apply the decoding algorithm $\mathcal{D}_{GO_2(\boldsymbol{L},g)}$ for $GO_2(\boldsymbol{L}, g)$. Since $\boldsymbol{c}$ has a Hamming distance of t to $GO_2(\boldsymbol{L}, g)$, we obtain the codeword

$$\boldsymbol{mG'} = \mathcal{D}_{GO_2(\boldsymbol{L},g)}(\boldsymbol{c})$$

No permutation is necessary, since it is hidden in the secret support $\boldsymbol{L}$, which is only used for the key generation process. Since $\boldsymbol{G}$ is in systematic form, no multiplication with an invers matrix $\boldsymbol{S}^{-1}$ is necessary as well. All what remains to be done is to read the infobits of $\boldsymbol{mG}$.

5 McEliece based on binary quasi-dyadic codes

As already mentioned above, the original McEliece scheme [21] suffers from a major drawback: the size of the public keys, which is typically several 100kB or even above the megabyte limit.

The canonical way in solving this dilemma is to use generator matrices with a high algebraic, even crystalline structure for the underlying Goppa codes. This has to be done with great care, because a potential adversary might use exactly that additional structure for launching attacks against the modified scheme, just what the algebraic attack [12] aims to do.

An advantage of the McEliece scheme compared to other PKC-schemes like e.g. RSA is its computational simplicity. The necessary operations for the encoding step are just byte level operations like *xor* and the like. Consequently, McEliece is quite fast, encryption has time complexity of $O(n^2)$ over a code of length n, where RSA with n-bit keys has time complexity $O(n^3)$ [23].

Saving space might have the drawback of loss in speed. So even when the compactification of the code representation would be successful, this point should be addressed, too.

In the sequel we show how to design generator matrices for Goppa codes allowing a compact representation. We follow the construction in [4, 23].

5.1 System parameters

The (effective) system parameters $k, m, n, t \in \mathbb{N}$, where $k < n$, $t \ll n$, are again as in section (4), but the generation process of the public and private keys as given in [23] is more complicated due to techniques used to hide the code structure. For instance, the generated code would have initially a code length of N, where $N \gg n$. It would be shortened afterwards to a code of length n.

For simplicity however, we assume here that a permuted support $\boldsymbol{L}$ suffices to hide the quasi-dyadic structure.[18]

5.1.1 Generating the public and private keys

The generation process of the public and private keys requires to solve equation (2.4.45) of theorem (2.4.34),

$$\frac{1}{h_{i\oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}.$$

It will yield the necessary z_i and L_j to construct a dyadic Cauchy matrix as

$$z_i = 1/h_i + \omega$$
$$L_j = 1/h_j + 1/h_0 + \omega$$

for some $\omega \in \mathbb{F}_q$.

[18]See also section (12).

5.1.2 Generating the public generator matrix

The first step generates a binary Goppa code. This is done according the construction given in [4], Algorithm 2. The results of this algorithm (see section (7.1)) are two sets $\boldsymbol{z} = \{z_0, \dots, z_{t-1}\}$ and $\boldsymbol{L} = \{L_0, \dots, L_{n-1}\}$. Using these two sets, set up the a Cauchy matrix (2.4.44):

$$\boldsymbol{H} = \begin{bmatrix} \frac{1}{z_0 - L_0} & \frac{1}{z_0 - L_1} & \cdots & \frac{1}{z_0 - L_{n-1}} \\ \frac{1}{z_1 - L_0} & \frac{1}{z_1 - L_1} & \cdots & \frac{1}{z_1 - L_{n-1}} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{1}{z_{t-1} - L_0} & \frac{1}{z_{t-1} - L_1} & \cdots & \frac{1}{z_{t-1} - L_{n-1}} \end{bmatrix} \in \mathbb{F}_q^{t \times n}.$$

By the construction of Algorithm 2, $\boldsymbol{H}$ is also in quasi-dyadic form. The next step consists in co-tracing the matrix $\boldsymbol{H} \in \mathbb{F}_q^{t \times n}$. Co-tracing describes a process to transform $\boldsymbol{H} \in \mathbb{F}_q^{t \times n}$ into a matrix $\boldsymbol{H'} \in \mathbb{F}_2^{mt \times n}$, while keeping its dyadic structure. We give a visual example for the co-trace construction.

Let $u_0 = (1, 1, 0, 1), u_1 = (0, 1, 0, 1) \in \mathbb{F}_{2^4}$ and

$$T = \begin{bmatrix} u_0 & u_1 \\ u_1 & u_0 \end{bmatrix} \in \mathbb{F}_{2^4}^{2 \times 2},$$

a dyadic matrix. Then the usual trace construction, i.e. writing the individual bits just below each other, would destroy the dyadic structure:

$$T = \begin{bmatrix} u_0 & u_1 \\ u_1 & u_0 \end{bmatrix} = \begin{bmatrix} (1,1,0,1) & (0,1,0,1) \\ (0,1,0,1) & (1,1,0,1) \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 1 \end{bmatrix}.$$

To keep the structure, the individual bits have to be used in an interleaved fashion, which is what the co-trace construction does (see Fig. (2)).

Finally, compute $\boldsymbol{H} \in \mathbb{F}_2^{mt \times n}$, the systematic form of $\boldsymbol{H'} \in \mathbb{F}_2^{mt \times n}$. Note that this step might fail as $\boldsymbol{H}'$ might possibly not have full rank, in which case the procedure has to be restarted to generate a new code. If $\boldsymbol{H}'$ does have full rank, i.e. its of the form $\boldsymbol{H}' = [\boldsymbol{R}^T \mid \boldsymbol{I}_{mt}] \in \mathbb{F}_2^{mt \times n}$, a respective generator matrix in systematic form is $\boldsymbol{G} = [\boldsymbol{I}_{n-mt} \mid \boldsymbol{R}]$ (see (11.1.1)).

5.1.3 Generating a private parity check matrix

Since binary Goppa codes are subfield subcodes, a procedure able to decode GRS-codes is able to decode binary Goppa codes as well. It is therefore possible to use parity check matrices $\boldsymbol{H}$ over $\mathbb{F}_{2^m}$ to build a decoder for binary Goppa codes.

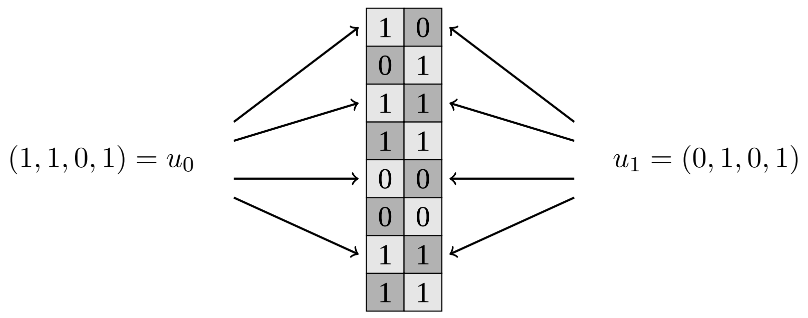

Figure 2: The co-trace construction.

Because of remark (2.4.26), it is possible to choose as private parity check matrix an $\boldsymbol{H'} \in \mathbb{F}_{2^m}^{2t\times n} = \boldsymbol{VD}$ as given in (2.4.42):

$$\boldsymbol{H'} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_0 & L_1 & \cdots & L_{n-1} \\ L_0^2 & L_1^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & \cdots & L_{n-1}^{t-1} \\ L_0^t & L_1^t & \cdots & L_{n-1}^t \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{2t-1} & L_1^{2t-1} & \cdots & L_{n-1}^{2t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-2} & 0 & \cdots & 0 \\ 0 & g(L_1)^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-2} \end{bmatrix}$$

Note that the Goppa polynomial $g(X)$ was gained as part of building the public generator matrix (5.1.2),

$$g(X) = (X - z_0)\cdots(X - z_{t-1}),$$

and that the support $\boldsymbol{L}$ is secret. Hence, $\boldsymbol{H'}$ is actually a secret matrix. Note that $\boldsymbol{H}$ in (5.1.1) is also a parity check matrix. Due to the key equation (5.3.7) is actually the parity check matrix used in the decoder.

$$\boldsymbol{H} = \begin{bmatrix} g_{2t} & 0 & 0 & \cdots & 0 \\ g_{2t-1} & g_{2t} & 0 & \cdots & 0 \\ g_{2t-2} & g_{2t-1} & g_{2t} & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_{2t} \end{bmatrix} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_0 & L_1 & \cdots & L_{n-1} \\ L_0^2 & L_1^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & \cdots & L_{n-1}^{t-1} \\ L_0^t & L_1^t & \cdots & L_{n-1}^t \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{2t-1} & L_1^{2t-1} & \cdots & L_{n-1}^{2t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-2} & 0 & \cdots & 0 \\ 0 & g(L_1)^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-2} \end{bmatrix}. \tag{5.1.1}$$

5.2 The encryption step

To encrypt a plaintext $\boldsymbol{m} \in \mathbb{F}_2^k$, we proceed again as in the classical case, choosing a vector $\boldsymbol{z} \in \mathbb{F}_2^n$ of weight t randomly and computing the ciphertext:

$$\boldsymbol{c} = \boldsymbol{mG} \oplus \boldsymbol{z}.$$

Note that because the redundant part of $\boldsymbol{G}$ is a quasi-dyadic matrix, the vector-matrix product $\boldsymbol{mG}$ can be done using the fast Walsh-Hadamard transform and the dyadic convolution (see section (2.4.8)).

5.3 The decryption step

The decryption step is again as outlined in section (4.3). To be more specific, in particular for the implementation, we give more details about the construction of the syndrome polynomial.

5.3.1 The setup

Suppose we are given a separable binary Goppa code with Goppa polynomial $g(X) \in \mathbb{F}_{2^m}[X]$ and support $\boldsymbol{L} = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_{2^m}^n$, where the degree r of $g(X)$ is even, $\deg g(X) = r = 2t$. Suppose further the codeword $\boldsymbol{u} \in \mathbb{F}_2^n$ has been received. It can be written as $\boldsymbol{u} = \boldsymbol{c} + \boldsymbol{e}$, where $\boldsymbol{c} \in \mathbb{F}_2^n$ is the correct codeword and $\boldsymbol{e} \in \mathbb{F}_2^n$ is an error vector. If $e_i \neq 0$, we say that an error has occurred in position L_i [22].

The decoding problem now consists in computing the error vector $\boldsymbol{e}$. Substracting it from the received vector u results in finding the correct code word c. The syndrome polynomial of the received vector $\boldsymbol{u}$ is given as

$$\begin{aligned} S(X) &\equiv \sum_{i=0}^{n-1} \frac{u_i}{X - L_i} \mod g(X) \\ &\equiv \sum_{i=0}^{n-1} \frac{e_i}{X - L_i} \mod g(X) \end{aligned} \tag{5.3.1}$$

and the set of all error locations will be denoted by $\mathbb{B} := \{L_i : e_i \neq 0\}$. As we are in the binary case, having $\beta \in \mathbb{B}$ as error location is equivalent to $e_\beta = 1$. Using this notation, the syndrome polynomial $S(X)$ can be defined as

Definition 5.3.1 (Syndrome polynomial).

$$S(X) \equiv \sum_{\beta \in \mathbb{B}} \frac{e_\beta}{X - \beta} \mod g(X). \tag{5.3.2}$$

We further define the error locator polynomial $\sigma(X)$ and the error evaluator polynomial $\omega(X)$. Once they are computed, they will solve the decoding problem.

Definition 5.3.2 (Error locator polynomial). [19]

$$\sigma(X) := \prod_{\beta\in\mathbb{B}} (X-\beta) \tag{5.3.3}$$

Definition 5.3.3 (Error evaluator polynomial). [20]

$$\omega(X) := \sum_{\beta\in\mathbb{B}} e_\beta \cdot \prod_{\substack{\gamma\in\mathbb{B}\\ \gamma\neq\beta}} (X-\gamma) \tag{5.3.4}$$

The zeros of $\sigma(X)$ are clearly the error positions, whereas the values of $\omega(X)$, when evaluated at those positions, are the actual error values. Again, because we are in the binary case, those values are 1.[21]

Theorem 5.3.4. *The following equations hold [22]:*

$$\deg\sigma(X) = |\mathbb{B}|, \quad \deg\omega(X) < |\mathbb{B}|, \tag{5.3.5}$$
$$\gcd(\sigma(X), \omega(X)) = 1, \tag{5.3.6}$$
$$\omega(X) \equiv \sigma(X)S(X) \mod g(X), \tag{5.3.7}$$

Proof. Equation (5.3.5) is an immediate consequence of (5.3.3) and (5.3.4). For $\beta\in\mathbb{B}$ it holds that

$$\omega(\beta) = \prod_{\beta\neq\gamma} (\beta-\gamma) \neq 0, \tag{5.3.8}$$

and therefore (5.3.6) follows. Finally, we get

$$\begin{aligned} S(X)\sigma(X) &\equiv \left(\sum_{\beta\in\mathbb{B}} \frac{e_\beta}{X-\beta}\right) \prod_{\beta\in\mathbb{B}} (X-\beta) \\ &\equiv \sum_{\beta\in\mathbb{B}} e_\beta \cdot \prod_{\substack{\gamma\in\mathbb{B}\\ \gamma\neq\beta}} (X-\gamma) \\ &\equiv \omega(X) \mod g(X). \end{aligned}$$

□

Remark 5.3.5 (Key equation.). *The equation* (5.3.7)

$$\omega(X) \equiv \sigma(X)S(X) \mod g(X)$$

is also known as key equation. Solving this equation gives $\sigma(X)$ and $\omega(X)$ and thus in turn solves the decoding problem as well.

[19] Note the similarity to (2.4.6)
[20] Not the similarity to (2.4.12). In the binary case $\omega(X) = \sum_{\beta\in\mathbb{B}} \prod_{\substack{\gamma\in\mathbb{B}\\ \gamma\neq\beta}} (X-\gamma)$.
[21] The definition of $\omega(X)$ can also be used in non-binary cases, and here the error values are not a-prior known.

5.3.2 Solving the key equation using the Euclidean algorithm

The Euclidean algorithm is typically used for computing greatest common divisors, e.g. of two integers or polynomials. We give a quick reminder [26].

Let $\mathbb{F}$ an arbitrary field, $f(X), h(X) \in \mathbb{F}[X]$ with $\deg h(X) \leq \deg f(X)$. Furthermore, let $r_{-1}(X) := f(X)$ and $r_0 := h(X)$. The steps of the algorithm consist of a division by remainder. As the degrees of the remainders strictly decrease, the algorithm continues until the last remainder eventually becomes 0 and the greatest common divisor of $f(X)$ and $h(X)$ becomes $r_s(X)$, see (5.3.9):

$$\begin{array}{lcll} r_{-1}(X) &=& q_1(X)r_0(X) + r_1(X), & deg(r_1) < deg(r_0) \\ & & \vdots & \vdots \\ r_{k-2}(X) &=& q_k(X)r_{k-1}(X) + r_k(X), & deg(r_k) < deg(r_{k-1}) \\ & & \vdots & \\ r_{s-1}(X) &=& q_{s+1}(X)r_s(X) & \\ & & & \\ r_s(X) &=& \gcd(f(X), g(X)). & \end{array} \tag{5.3.9}$$

In each step of the process it is possible to write the current remainder $r_k(X)$ in terms of the two previous remainders. Moreover, it can be shown that it is possible to write all remainders, including $r_s(X)$, in terms of $f(X)$ and $h(X)$. The fact is stated in the following theorem, see [26], Thm. (8.3.5).

Theorem 5.3.6. *The remainders $r_k(X)$, $k \geq -1$, in the Euclidean algorithm satisfy*

$$r_k(X) = a_k(X)f(X) + b_k(X)h(X), \text{ where}$$

$$\begin{aligned} a_k(X) &= -q_k(X)a_{k-1}(X) + a_{k-2}(X), \quad k \geq 1 \\ b_k(X) &= -q_k(X)b_{k-1}(X) + b_{k-2}(X), \quad k \geq 1 \\ a_0(X) &= 0 \\ b_0(X) &= 0 \\ a_{-1}(X) &= 1 \\ b_{-1}(X) &= 1 \end{aligned} \tag{5.3.10}$$

□

To solve the key equation (5.3.7), run the Euclidean algorithm with $f(X)$ replaced by the Goppa polynomial $g(X)$ and $h(X)$ replaced by the syndrome polynomial. As is also shown in [26], the algorithm has to be applied until[22]

$$deg(r_k) < t/2, \;\; deg(r_{k-1}) \geq t/2, \tag{5.3.11}$$

where $t = \deg g(X)$ is the degree of the Goppa polynomial. Then the error locator polynomial $\sigma(X)$ and the error evaluator polynomial $\omega(X)$ can be computed as

$$\begin{aligned} \sigma(X) &= b_k(0)^{-1}b_k(X) \\ \omega(X) &= b_k(0)^{-1}r_k(X). \end{aligned} \tag{5.3.12}$$

□

[22]One might speak of running the extended Euclidean algorithm partially [25].

5.3.3 Finding the roots of the error locator polynomial

In the binary case, finding the actual roots of the error locator polynomial is the last remaining step. Typically, it is done using Chien search [9] or using the Berlekamp trace algorithm [7].

However, for the test example below, we can evaluate the error locator polynomial using brute force since the chosen parameter values are quite small $(n = 32, m = 6, t = 4)$.

Part VI
Algebraic attacks against quasi-dyadic McEliece

In 2010, new algebraic attacks against the quasi-cyclic and quasi-dyadic variant of McEliece have been proposed [12]. The attack aims to use the highly symmetric structure of the underlying matrices, and in almost all cases succeeds. In case of quasi-dyadic binary Goppa codes, however, the attack fails.

The reason for this failure lies in the fact that separable, binary Goppa codes allow two different representations based on $g(X)$ or $g(X)^2$, respectively (see (2.4.26) and (2.4.41)):

$$GO_2(\boldsymbol{L}, g) = GO_2(\boldsymbol{L}, g^2).$$

The encoding step is done via the weak representation based on $g(X)$, i.e. on a generator matrix $\boldsymbol{G}$ gained via a parity check matrix $\boldsymbol{H} \in \mathbb{F}_{2^m}^{t \times n}$ like (see (2.4.27))

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & \dots & 1 \\ L_0 & L_1 & \dots & L_{n-1} \\ L_0^2 & L_1^2 & \dots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & & \dots L_{n-1}^{t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-1} & 0 & \dots & 0 \\ 0 & g(L_1)^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-1} \end{bmatrix},$$

but for the decoding step a version based on $g(X)^2$ is used (see (2.4.42)):

$$\boldsymbol{H}' = \boldsymbol{V}\boldsymbol{D} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ L_0 & L_1 & \cdots & L_{n-1} \\ L_0^2 & L_1^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{t-1} & L_1^{t-1} & \cdots & L_{n-1}^{t-1} \\ L_0^{t} & L_1^{t} & \cdots & L_{n-1}^{t} \\ \vdots & \vdots & \vdots & \vdots \\ L_0^{2t-1} & L_1^{2t-1} & \cdots & L_{n-1}^{2t-1} \end{bmatrix} \begin{bmatrix} g(L_0)^{-2} & 0 & \cdots & 0 \\ 0 & g(L_1)^{-2} & \cdots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & g(L_{n-1})^{-2} \end{bmatrix}$$

The algebraic attack [12] is not against the Goppa polynomial $g(X)$ itself. Rather it exploits the quasi-dyadic structure of the Goppa code and is able to come up with a general alternate decoder, which is able to correct $\lfloor t/2 \rfloor$ errors. But the decoder based on the private trapdoor is able to correct twice as much errors. The attack is therefore able to find a general alternant decoder, but not a Goppa decoder.

One approach for an attacker might be to simple guess the remaining errors. However, the workload is as high as $\binom{n}{t/2}/\binom{t}{t/2}$, which is prohibitive for proper parameter settings [5].

Another approach would be to launch an attack to find the Goppa polynomial itself, but to the author's knowledge, there is no source available describing such an attack.

Part VII

Implementation

The implementation is written with CWEB [17], a tool which contains two main programs:

- ▷ *cweave* for generating the TEX documentation.
- ▷ *ctangle* for generating the corresponding C source.

Thus, running *ctangle* with the *.w file of this document will produce C sources which can be compiled by a usual C compiler, yielding an executable program. On the other hand, running *cweave* with *.w will produce a *.tex file, which can then be compiled into a *.dvi or *.pdf file.

The following implementation is in parts based on HyMES [29].

6 Finite field implementation

The field implementation is taken form HyMES [29]. Some comments have been removed and some variables renamed. The implementation deals with field extensions $[\mathbb{F}_{p^m} : \mathbb{F}_p]$ of degree m for $p = 2$.

35 ⟨ gf.h 35 ⟩ ≡

```
#ifndef GF_H_INCLUDED
#define GF_H_INCLUDED
  ⟨ Exported field declarations 36 ⟩
  ⟨ Exported field macros 39 ⟩
  ⟨ Exported field functions 37 ⟩
  ⟨ Exported field variables 38 ⟩
#endif
```

For the current implementation we chose a maximal extension degree of 16. Each field element of $\mathbb{F}_{2^m}$ can be therefore represented by maximally 16 bits. The necessary datatype *uint16_t* is contained in *inttypes.h*, renamed to *gf_t* and added to the external interface.

36 ⟨ Exported *field* declarations 36 ⟩ ≡

```
#include <inttypes.h>
#define MAX_EXT_DEG 16
  typedef uint16_t gf_t;
```

This code is used in section 35.

For initializing and releasing the internally used memory we add two functions, *gf_init* and *gf_free*, to the external interface of the field implementation.

37 ⟨ Exported *field* functions 37 ⟩ ≡
```
   extern void gf_init(int extdeg);
   extern void gf_free( );
   extern gf_t gf_pow(gf_t x, int i);
   extern gf_t gf_rand(int(*u8rnd)( ));
```
This code is used in section 35.

38 ⟨ Exported *field* variables 38 ⟩ ≡
```
   extern int gf_extd;      /* extension degree */
   extern int gf_card;      /* cardinality */
   extern int gf_ord;      /* multiplicative order */
   extern gf_t *gf_log;
   extern gf_t *gf_exp;
```
This code is used in section 35.

39 ⟨ Exported *field* macros 39 ⟩ ≡
```
#define gf_unit( )  1
#define gf_zero( )  0
#define gf_add(x, y)  ((x) ⊕ (y))
#define _gf_modq_1(d)  (((d) & gf_ord) + ((d) ≫ gf_extd))
#define gf_mul_fast(x, y)  ((y) ? gf_exp[_gf_modq_1(gf_log[x] + gf_log[y])] : 0)
#define gf_mul(x, y)  ((x) ? gf_mul_fast(x, y) : 0)
#define gf_square(x)  ((x) ? gf_exp[_gf_modq_1(gf_log[x] ≪ 1)] : 0)
#define gf_sqrt(x)  ((x) ? gf_exp[_gf_modq_1(gf_log[x] ≪ (gf_extd − 1))] : 0)
#define gf_div(x, y)  ((x) ? gf_exp[_gf_modq_1(gf_log[x] − gf_log[y])] : 0)
#define gf_inv(x)gf_exp  [gf_ord − gf_log[x]]
```
This code is used in section 35.

40 ⟨ `gf.c` 40 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#include "gf.h"
   ⟨ Define field variables 41 ⟩
   ⟨ Static field variables 43 ⟩
   ⟨ Static field functions 50 ⟩
   ⟨ field functions 46 ⟩
```

41 ⟨ Define *field* variables 41 ⟩ ≡
```
   int gf_extd = 0;      /* extension degree */
   int gf_card = 0;      /* cardinality */
   int gf_ord = 0;      /* multiplicative order */
```

```
gf_t ∗ gf_log = Λ;
gf_t ∗ gf_exp = Λ;
```
This code is used in section 40.

Static field variables are meant for internal use only. The array *prim_poly* containes the `MAX_EXT_DEG` primitive polynomials which can be used in the implementation. They are decoded in binary. For instance, $07 = 111_2$ means the polynomial $X^2 + X + 1$, used for an extension degree of $m = 2$ and $013 = 1011_2$ stands for $X^3 + X + 1$ for an extension degree of $m = 3$. Extension degrees $m = 0$ and $m = 1$ are not used.

43 ⟨ Static *field* variables 43 ⟩ ≡
```
static unsigned prim_poly[MAX_EXT_DEG + 1] = {°1 , °3 ,        /∗ not used ∗/
°7 , °13 , °23 , °45 , °103 , °203 , °435 , °1041 , °2011 , °4005 , °10123 , °20033 , °42103 , °100003 , °210013 };
```
See also section 44.

This code is used in section 40.

44 ⟨ Static *field* variables 43 ⟩ +≡
```
static int init_done = 0;        /∗ flag for intialization check ∗/
```

6.1 Initialize the field

As outlined above, in software we work with the multiplicative representation of the field. Initializing therefore involves creating the exponential and logarithmic tables. The single parameter of *gf_init* is the chosen extension degree for the field.

46 ⟨ *field* functions 46 ⟩ ≡
```
void gf_init(int extdeg)
{
  if (extdeg > MAX_EXT_DEG) {
    fprintf (stderr, "Extension␣degree␣%d␣not␣implemented␣!\n", extdeg);
    exit(EXIT_FAILURE);
  }
  if (init_done ≠ extdeg) {        /∗ check for a previous field usage ∗/
    if (init_done)
      gf_free( );        /∗ release the field tables ∗/
    init_done = gf_extd = extdeg;        /∗ initialize the field parameters ∗/
    gf_card = 1 ≪ extdeg;
    gf_ord = gf_card − 1;
    gf_init_exp( );        /∗ initialize the exponential table ∗/
    gf_init_log( );        /∗ initialize the logarithmic table ∗/
```

```
      }
   }
```

See also sections 48 and 54.

This code is used in section 40.

6.2 Release the field tables.

Releasing the field tables involves only releasing the internal memory.

```
48  ⟨ field functions 46 ⟩ +≡
    void gf_free( )
    {
       if (gf_exp)
          free(gf_exp);
       if (gf_log)
          free(gf_log);
    }
```

6.3 Initialize the exponential table

α is chosen as root of a primitive polynomial f. As outlined above, α does have the explicit form $X + I(f)$. Multiplying by α means therefore multiplying by X modulo $I(f)$, which in turn in binary means a right shift by one bit position. By successively multiplying the previous field element in the exponential table by α, we generate the whole field: *gf_exp*$[i-1]\cdot\alpha = \alpha^i$.

```
50  ⟨ Static field functions 50 ⟩ ≡
    static void gf_init_exp( )
    { int i;
       gf_exp = malloc((1 ≪ gf_extd) ∗ sizeof (∗gf_exp));      /∗ fetch some memory ∗/
       gf_exp[0] = 1;
       for (i = 1; i < gf_ord; ++i) {
          gf_exp[i] = gf_exp[i − 1] ≪ 1;      /∗ multiply by α ∗/
          if (gf_exp[i − 1] & (1 ≪ (gf_extd − 1)))      /∗ modulo I(f),i.e. ∗/
             gf_exp[i] ⊕= prim_poly[gf_extd];      /∗ substract f for powers too high ∗/
       }
       gf_exp[gf_ord] = 1;      /∗ should be 0: hack for the multiplication ∗/
    }
```

See also section 52.

This code is used in section 40.

6.4 Initialize the logarithm table

Initializing the logarithmic table *gf_log* is done using *gf_exp* and applying the formula *gf_log*$[\alpha^i] = i$.

```
52  ⟨ Static field functions 50 ⟩ +≡
    static void gf_init_log( )
    {  int i;
       gf_log = malloc((1 << gf_extd) * sizeof (*gf_log));      /* fetch some memory */
       gf_log[0] = gf_ord;      /* log of 0 is gf_ord by convention */
       for (i = 0; i < gf_ord; ++i)
          gf_log[gf_exp[i]] = i;
    }
```

6.5 Powers of field elements

For the '*gf_pow*' procedure we assume $i >= 0$. By convention $0^0 = 1$

```
54  ⟨ field functions 46 ⟩ +≡
    gf_t gf_pow(gf_t x, int i)
    {
       if (i ≡ 0)
          return 1;
       else if (x ≡ 0)
          return 0;
       else {
          while (i >> gf_extd)
             i = (i & gf_ord) + (i >> gf_extd);
          i *= gf_log[x];
          while (i >> gf_extd)
             i = (i & gf_ord) + (i >> gf_extd);
          return gf_exp[i];
       }
    }
```

7 Building a binary Goppa code in quasi-dyadic form

The first step in generating a quasi-dyadic binary Goppa code is to generate the dyadic submatrices.

7.1 Constructing a purely dyadic Goppa code

57 ⟨ `binary-quasi-dyadic-goppa-code.h` 57 ⟩ ≡

```
#ifndef BINARY_QD_GOPPA_CODE_H_INC
#define BINARY_QD_GOPPA_CODE_H_INC
  ⟨ Exported dyadic functions 58 ⟩
#endif
```

58 ⟨ Exported *dyadic* functions 58 ⟩ ≡

```
  void binary_quasi_dyadic_goppa_code(uint32_t m, uint32_t n, uint32_t t, int *b, gf_t *h,
      gf_t *omega, int *bc, int debug);
```

This code is used in section 57.

59 ⟨ `binary-quasi-dyadic-goppa-code.c` 59 ⟩ ≡

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "gf.h"
#include "utils.h"
  ⟨ Static dyadic macros 60 ⟩
  ⟨ dyadic functions 62 ⟩
```

60 ⟨ Static *dyadic* macros 60 ⟩ ≡

```
#define REMOVE_FROM_U(elt)
  for (int l = k; l < q; ++l) {
    if (U[l] ≡ elt) {
      U[l] = 0;
      break;
    }
  }
```

See also section 61.

This code is used in section 59.

61 ⟨ Static *dyadic* macros 60 ⟩ +≡

```
#define INIT_U_RANDOM()
  U[0] = 0;
  for (int u = 1; u < q; ++u) {
    U[u] = gf_exp[u];
  }
  for (int u = 1; u < q − 1; ++u) {
    register int v = (rand() % (u + 1));
    gf_ttmp = U[u];
    U[u] = U[v + 1];
    U[v + 1] = tmp;
  }
```

To construct a binary dyadic code, we follow the description as given in [4], Algorithm 2. The corresponding C code is given in *binary_quasi_dyadic_goppa_code*. Inputs are the extension degree m, code length n, error capability t. Output is a dyadic signature h, ω and the sequence bc of all consistent blocks of columns. For simplicity, we use a codelength $n \leq q/2$ and a designed error number t which is a power of 2.

62 ⟨ *dyadic* functions 62 ⟩ ≡

```
  void binary_quasi_dyadic_goppa_code(uint32_t m, uint32_t n, uint32_t t, int *b, gf_t *h,
          gf_t *omega, int *bc, int debug)
  { gf_t *U;
    unsigned int c, k;
    int const q = 1 ≪ m;
    int const C = q/t;
    ⟨ Check if t is a power of 2; exit in case it is not or in case t > (2^m − 1)/m 63 ⟩;
    ⟨ q ← 2^m; exit in case code length n > q/2; initialize underlying field buffer U: U ← F_q \ {0} 64 ⟩;
    ⟨ Generate consistent root and support sets. 65 ⟩;
    ⟨ return h, bc and ω. Free used buffer. 74 ⟩;
  }
```

This code is used in section 59.

63 ⟨ Check if t is a power of 2; *exit* in case it is not or in case $t > (2^m - 1)/m$ 63 ⟩ ≡

```
  c = t;      /* count the bits contained in t */
  c = (c & #55555555U) + ((c ≫ 1) & #55555555U);
  c = (c & #33333333U) + ((c ≫ 2) & #33333333U);
  c = (c & #0F0F0F0FU) + ((c ≫ 4) & #0F0F0F0FU);
  c = (c & #00FF00FFU) + ((c ≫ 8) & #00FF00FFU);
  c = (c & #0000FFFFU) + ((c ≫ 16) & #0000FFFFU);
  if ((c ≠ 1) ∨ (t ≡ 1) ∨ (t > 128) ∨ (t > ((double)(q − 1)/m))) {
    fprintf(stderr,
    "ERROR␣t(%d)␣is␣not␣a␣power␣of␣2␣or␣bigger␣than␣%d\n", t, (q − 1)/m);
    exit(−1);
  }
```

This code is used in section 62.

To implement Algorithm 2, we first initialize the field cardinality: $q := 2^m$. In case $n \geq q$, we print an error message and exit. Otherwise we allocate memory for the underlying buffer U of the finite field $\mathbb{F}_q$.

64 ⟨ $q \leftarrow 2^m$; *exit* in case code length $n > q/2$; initialize underlying field buffer U: $U \leftarrow \mathbb{F}_q \setminus \{0\}$ 64 ⟩ ≡

```
    if $(n > q/2)$ {
        fprintf(stderr, "ERROR␣n(%d)␣>␣q/2(%d)\n", $n, q/2$);
        exit$(-1)$;
    }
    $U = malloc(q * \textbf{sizeof}\ (*U))$;
```
This code is used in section 62.

In order to construct consistent root and support sets for the Goppa polynomial, the next step is to initialize U with the elements of $\mathbb{F}_q \setminus \{0\}$ and to pick randomly an $h_0 \in U$. The remaining entries of the signature h are determind in m steps. Per step, we generate n new entries h_n, where $2^s \leq n < 2^{s+1}$ and $0 \leq s < m$. It is possible that not all of the h_n will receive a value different from zero. Depending on the number of consistent support blocks and the value of t, it might be necessary to re-initialize the field and to repeat those m steps. However, the probability for this case is quite low.

After the signature h has been calculated, we need to check its consistency. We also need to find a proper ω to use in equation (2.4.45), theorem (2.4.34). To prevent spurious intersections between the root set and the support set, ω has to be chosen with care. Firstly, reset U to $\mathbb{F}_q$. While checking for the root and support set consistency, continuously remove elements out of those sets from U. ω can be chosen from the remaining U.

65 ⟨ Generate consistent root and support sets. 65 ⟩ ≡
```
    do {
        ⟨ Set $U \leftarrow \mathbb{F}_q \setminus \{0\}$ 66 ⟩;
        ⟨ Set $h_0 \xleftarrow{\$} U,\ U \leftarrow U \setminus \{h_0\}$ 67 ⟩;
        ⟨ Determine the remaining signature entries $h_n$ $(1 \leq n \leq q-1)$ 68 ⟩;
        ⟨ Reset $U$ : $U \leftarrow \mathbb{F}_q$ for finding $\omega$ 71 ⟩;
        ⟨ Check for consistent root and support set 72 ⟩;
    } while $(c * t < n)$;
```
This code is used in section 62.

Initialize U with the elements of $\mathbb{F}_q \setminus \{0\}$. By construction, the exponential table *gf_exp* of the field does not contain zero, therefore we just copy *gf_exp* into U. As the algorithm proceeds, used elements of U will be reset to zero. After the copy, both the first and the last position of U contain 1, so $U[0]$ is not needed. By setting $U[0]$ to zero, it is marked as already used. Finally, entries of U except $U[0]$ are permuted randomly.

66 ⟨ Set $U \leftarrow \mathbb{F}_q \setminus \{0\}$ 66 ⟩ ≡
```
    srand((unsigned) rdtsc());
    ⟨ Reset $U$ : $U \leftarrow \mathbb{F}_q$ for finding $\omega$ 71 ⟩;
```
This code is used in section 65.

Since the entries above $U[1]$ have been permuted randomly in the previous step, we just take $h_0 := U[1]$. Removing h_0 from U is done by setting $U[1]$ to zero.

67 ⟨ Set $h_0 \xleftarrow{\$} U,\ U \leftarrow U \setminus \{h_0\}$ 67 ⟩ ≡
```
    $h[0] = U[1]$;
    $U[1] = 0$;
```
This code is used in section 65.

In m steps, we determine the h_n for $1 \le n < q$. For each step, we update the read position k, which means that $U[n] = 0$ for $n < k$. The value at the read position is taken to be h_i, where $i = 2^s$ and $0 \le s < m$. The h_{i+j}, where $1 \le j \le i - 1$, are computed according to equation (2.4.45) in theorem (2.4.34).

68 $\langle$ Determine the remaining signature entries h_n $(1 \le n \le q-1)$ 68 $\rangle \equiv$

```
$k = 1;$
**for** (**int** $s = 0;\ s < m;\ {+}{+}s)$ {
  **while** $(U[k] \equiv 0 \land k < q - 1)$        /* move to the next read position */
    $++k;$
  $\langle\, i \leftarrow 2^s,\ h_i \overset{\$}{\leftarrow} U,\ U \leftarrow U \setminus \{h_i\}$ 69 $\rangle$
  $\langle$ Generate next $h_{i+j}$ for $1 \le j \le i-1;\ U \leftarrow U \setminus \{h_{i+j}\}$ 70 $\rangle$
}
```

This code is used in section 65.

Use for h_i, $i = 2^s$, the value at current read position k, and delete the value of h_i from U.

69 $\langle\, i \leftarrow 2^s,\ h_i \overset{\$}{\leftarrow} U,\ U \leftarrow U \setminus \{h_i\}$ 69 $\rangle \equiv$

```
**int const** $i = 1 \ll s;$
$h[i] = U[k];$
REMOVE_FROM_U($h[i]$);
```

This code is used in section 68.

Once h_i is found, we calculate the value for h_{i+j}, where $1 \le j \le i - 1$. To apply equation (2.4.45), theorem (2.4.34), we need to check for $h_i \ne 0$, $h_j \ne 0$ and $1/h_i + 1/h_j + 1/h_0 \ne 0$. In case all three checks succeed, we set $h_{i+j} = 1/(1/h_i + 1/h_j + 1/h_0)$ and remove its value from U. Otherwise, h_{i+j} is an undefined entry and set to zero.

70 $\langle$ Generate next h_{i+j} for $1 \le j \le i-1;\ U \leftarrow U \setminus \{h_{i+j}\}$ 70 $\rangle \equiv$

```
**for** (**int** $j = 1;\ j < i;\ {+}{+}j)$ {
  $h[i + j] = 0;$
  **if** $(h[i] \land h[j])$ {
    $h[i + j] = gf\_add(gf\_inv(h[i]), gf\_inv(h[j]));$
    $h[i + j] = gf\_add(h[i + j], gf\_inv(h[0]));$
  }
  **if** $(h[i + j])$ {
    $h[i + j] = gf\_inv(h[i + j]);$
    REMOVE_FROM_U($h[i + j]$);
  } **else if** ($debug \ge 5$)
    fprintf(stderr, "INFO:␣undefined␣entry␣at␣%d\n", $i + j$);
}
```

This code is used in section 68.

71 $\langle$ Reset $U:\ U \leftarrow \mathbb{F}_q$ for finding ω 71 $\rangle \equiv$

```
INIT_U_RANDOM();
```

This code is used in sections 65, 66, and 72.

Theorem (2.4.34) shows that the root set is consistent, if $0 \notin \{h_0, \ldots, h_{t-1}\}$, and that elements z_i of the root set are defined as $z_i = 1/h_i$, $0 \leq i \leq t-1$. If the root set is consistent, all z_i will be removed from U. To prevent spurios intersections between the root set $\boldsymbol{z}$ and the support set $\boldsymbol{L}$, we also remove the elements of the form $1/h_i + 1/h_0$. Finally, check for consistent support blocks.

```
72 ⟨ Check for consistent root and support set 72 ⟩ ≡
    $c = 0;$
    int consistent_root_set = 1;
    for (int $i = 0;\ i < t;\ {+}{+}i$)
        consistent_root_set &= $(h[i] \neq 0);$
    if (consistent_root_set) {
        $k = 1;$
        memset($b$, #00, $C *$ sizeof $(*b)$);
        ⟨ Reset $U :\ U \leftarrow \mathbb{F}_q$ for finding $\omega$ 71 ⟩
        $b[0] = 0;$
        $c = 1;$
        REMOVE_FROM_U(gf_inv($h[0]$));
        for (int $i = 1;\ i < t;\ {+}{+}i$) {
            REMOVE_FROM_U(gf_add(gf_inv($h[i]$), gf_inv($h[0]$)));
            REMOVE_FROM_U(gf_inv($h[i]$));
        }
        ⟨ Determine consistent support blocks $\{h_{jt}, \ldots, h_{(j+1)t-1}\}$ 73 ⟩
    }
```
This code is used in section 65.

As with the root set, a support block is consistent if $0 \notin \{h_{jt}, \ldots, h_{(j+1)t-1}\}$ for $1 \leq j \leq \lfloor q/t \rfloor$, in which case we remove the corresponding elements from U. According to theorem (2.4.34), those element are characterized by $\{1/h_i + 1/h_0 \,|\, i = jt, \ldots, (j+1)t-1\}$, where $1 \leq j \leq \lfloor q/t \rfloor$. The positions of the consistent blocks is saved for later use.

```
73 ⟨ Determine consistent support blocks $\{h_{jt}, \ldots, h_{(j+1)t-1}\}$ 73 ⟩ ≡
    for (int $j = 1;\ j < C;\ {+}{+}j$) {
        while $(U[k] \equiv 0 \land k < q-1)$ $++k;$
        int consistent_support_block = 1;
        for (int $i = j * t;\ i < (j+1) * t;\ {+}{+}i$) consistent_support_block &= $(h[i] \neq 0);$
        if (consistent_support_block) {
            $b[c] = j;$
            $++c;$
            for (int $i = j * t;\ i < (j+1) * t;\ {+}{+}i$) {
                REMOVE_FROM_U(gf_add(gf_inv($h[i]$), gf_inv($h[0]$)));
            }
        }
    }
```
This code is used in section 72.

Return a signature h, an array bc desribing the positions of consistent blocks (with the root set block on position 0) and the field element $omega$, which is needed for building the final root set and support set values (see again theorem (2.4.34)). Finally, free the buffer used.

74 $\langle$ **return** h, bc and ω. Free used buffer. 74 $\rangle \equiv$

```
$*bc = c$;
**for** (**int** $i = 0$; $i < q$; $++i$) {
    **if** ($U[i]$)
        $*omega = U[i]$;
}
$free(U)$;
```

This code is used in section 62.

7.2 Constructing the binary quasi-dyadic Goppa code

76 $\langle$ *main* build quasi-dyadic Goppa code 76 $\rangle \equiv$

```
**int** done_building_goppa_code;
**do** {
    done_building_goppa_code = 0;
    ⟨ Call binary_quasi_dyadic_goppa_code 77 ⟩
    ⟨ Compute $\boldsymbol{z}$ 78 ⟩
    ⟨ Compute the support 83 ⟩
    ⟨ Check $\boldsymbol{z}$ for consistency 79 ⟩
    ⟨ Check $\boldsymbol{L}$ for consistency 80 ⟩
    ⟨ Check $\boldsymbol{z} \cap \boldsymbol{L} = \emptyset$ 81 ⟩
    ⟨ Compute $\hat{H} \in \mathbb{F}_q^{t \times n}$ 84 ⟩
    ⟨ Compute the co-trace matrix 85 ⟩
    ⟨ Use $H'$ to build parity check matrix $H \in \mathbb{F}_2^{mt \times n}$ in systematic form 87 ⟩
    ⟨ Use $H$ to build generator matrix $G$ in systematic form 92 ⟩
    done_building_goppa_code = 1;
} **while** ($\neg$done_building_goppa_code);
```

This code is used in section 174.

Calling *binary_quasi_dyadic_goppa_code* delivers a consistent root set and consistent support blocks to construct the matrix $\Delta(t, h) =: \hat{H} \in \mathbb{F}_q^{t \times N}$, but[23] it will be necessary to hide the purely dyadic code structure, which is done in a next step.

77 $\langle$ Call *binary_quasi_dyadic_goppa_code* 77 $\rangle \equiv$

```
binary_quasi_dyadic_goppa_code($m, n, t, b, h$, &omega, &bc, debug);
```

This code is used in section 76.

[23]To stay consistent with [23], denote by $N = n$ the code length, and assume for simplicity that t is a power of 2.

As given in theorem (2.4.34), we have $z_i := 1/h_i + \omega$.

78 $\langle$ Compute $\boldsymbol{z}$ 78 $\rangle \equiv$

```
for (int $i = 0$; $i < t$; $++i$) {
   $z[i] = gf\_add(gf\_inv(h[i]), omega)$;
   if ($debug \geq 3$)
      printf("z[%d]␣=␣%04x\n", $i, z[i]$);
}
```

This code is used in section 76.

79 $\langle$ Check $\boldsymbol{z}$ for consistency 79 $\rangle \equiv$

```
for (int $i = 0$; $i < t$; $++i$)
   for (int $j = 0$; $j < t$; $++j$)
      if ($(i \neq j) \wedge (z[i] \equiv z[j])$) {
         fprintf(stderr, "ERROR:␣z[%d]=%04x␣==␣z[%d]=%04x\n", $i, z[i], j, L[j]$);
         continue;
      }
```

This code is used in section 76.

80 $\langle$ Check $\boldsymbol{L}$ for consistency 80 $\rangle \equiv$

```
for (int $i = 0$; $i < n$; $++i$)
   for (int $j = 0$; $j < n$; $++j$)
      if ($(i \neq j) \wedge (L[i] \equiv L[j])$) {
         fprintf(stderr, "ERROR:␣L[%d]=%04x␣==␣L[%d]=%04x\n", $i, L[i], j, L[j]$);
         continue;
      }
```

This code is used in section 76.

81 $\langle$ Check $\boldsymbol{z} \cap \boldsymbol{L} = \emptyset$ 81 $\rangle \equiv$

```
for (int $i = 0$; $i < t$; $++i$)
   for (int $j = 0$; $j < n$; $++j$)
      if ($z[i] \equiv L[j]$) {
         fprintf(stderr, "ERROR:␣z[%d]=%04x␣==␣L[%d]=%04x\n", $i, z[i], j, L[j]$);
         continue;
      }
```

This code is used in section 76.

$\Delta(t, h) = \hat{H} \in \mathbb{F}_q^{t\times N}$ can be seen as an array of N/t dyadic blocks of size $t\times t$: $\hat{H} = \begin{bmatrix} B_0 & \dots & B_{N/t-1}\end{bmatrix}$, where $B_0 = \Delta(h_0, \dots, h_{t-1})$ is the root block. The calculation of the roots z_i of the Goppa polynomial $g(X) \in \mathbb{F}_q[X]$ is done as prescribed by theorem (2.4.34):

$$z_i := 1/h_i + \omega,$$
$$g(X) := \prod_{i=0}^{t-1}(X - z_i).$$

82 ⟨ *main* assemble the Goppa polynomial 82 ⟩ ≡

```
$g = poly\_alloc(1)$;
$poly\_set\_coeff(g, 0, z[0])$;
$poly\_set\_coeff(g, 1, 1)$;
$poly\_calcule\_deg(g)$;
$poly\_t\ p = poly\_copy(g)$;
$poly\_calcule\_deg(p)$;
for (int $i = 1$; $i < t$; $++i$) {
   $poly\_set\_coeff(p, 0, z[i])$;
   $poly\_t\ p\_old = g$;
   $g = poly\_mul(g, p)$;
   $poly\_free(p\_old)$;
}
$poly\_free(p)$;
$poly\_calcule\_deg(g)$;
$g2 = poly\_mul(g, g)$;
```

This code is used in section 174.

Using the consistent root set and consistent support blocks, the support L is again computed using theorem (2.4.34):

$$L_j = 1/h_j + 1/h_0 + \omega.$$

83 ⟨ Compute the support 83 ⟩ ≡

```
for (int $j = 0, k = 0$; $k < bc$; $++k$) {
   for (int $i = 0$; $i < t$; $++i$) {
      if ($j < n$) {
         $gf\_t\ a0 = gf\_inv(h[0])$;
         $a0 = gf\_add(a0, omega)$;
         $gf\_t\ a1 = gf\_inv(h[b[k] * t + i])$;
         $L[j] = gf\_add(a0, a1)$;
         if ($debug \geq 3$)
            printf("L[%d]␣=␣%04x\n", $j, L[j]$);
         $++j$;
      }
   }
}
```

This code is used in section 76.

84 ⟨ Compute $\hat{H} \in \mathbb{F}_q^{t\times n}$ 84 ⟩ ≡

```
for (int $i = 0$; $i < t$; $++i$)
   for (int $j = 0$; $j < n$; $++j$)
      $H[i * n + j] = gf\_inv(gf\_add(z[i], L[j]))$;
```

This code is used in section 76.

```
85  ⟨ Compute the co-trace matrix 85 ⟩ ≡
    if (Hbin ≠ Λ) mat_free(Hbin);
    Hbin = mat_ini($m * t, n$);
    if (Hbin ≡ Λ) {
      fprintf(stderr, "INFO:␣mat_ini␣failed\n");
      continue;
    }
    mat_set_to_zero(Hbin);
    for (int $i = 0$; $i < n$; $i$++)
      for (int $j = 0$; $j < t$; $j$++) {
        const gf_t $y = H[j * n + i]$;
        for (int $k = 0$; $k < m$; $k$++) {
          if ($y$ & $(1 \ll k)$) {
            const int idx = $(t * k + j) *$ Hbin→rwdcnt $+ i/$BITS_PER_LONG;
            Hbin→elem[idx] ⊕= ($1_{\mathrm{UL}} \ll (i$ % BITS_PER_LONG));
          }
        }
      }
    if (debug ≥ 5)
      ⟨ Print co-traced matrix 86 ⟩
  This code is used in section 76.
```

```
86  ⟨ Print co-traced matrix 86 ⟩ ≡
    print_bin_matrix($m * t, n$, Hbin, "Hbin␣AFTER␣CO-TRACING:");
  This code is used in section 85.
```

To transform *Hbin* into systematic form, the *HyMES* function *mat_rref* is used. Because it is possible that a co-traced matrix does not have the full rank $n - mt$, an explicit check is necessary.

```
87  ⟨ Use $H'$ to build parity check matrix $H \in \mathbb{F}_2^{mt \times n}$ in systematic form 87 ⟩ ≡
    int *perm = mat_rref(Hbin);
    if (perm ≡ Λ) {
      if (debug ≥ 5)
        fprintf(stderr, "INFO:␣mat_rref␣FAILED␣FOR␣Hbin\n");
      continue;
    }
    free(perm);      /* permutation not used */
    int Hbin_in_systematic_form = 1;
      ⟨ Check Hbin for systematic form 88 ⟩
    if (Hbin_in_systematic_form)
      ⟨ Change row order 89 ⟩
    else {
      if (debug ≥ 5)
        fprintf(stderr, "INFO:␣co-traced␣matrix␣not␣of␣full␣rank.\n");
      continue;
    }
```
This code is used in section 76.

Despite returning a valid pointer *perm*, the *HyMES* function *mat_rref* spuriously does not convert *Hbin* into systematic form. Using such an *Hbin* leads to segfaults afterwards, such that an explicit check is necessary.

88 ⟨ Check Hbin for systematic form 88 ⟩ ≡

```
for (int $j = 0$; $j < m * t$; $++j$)
  for (int $i = 0$; $i < n$; $++i$)
    if $((i + j) \equiv (n - 1))$
      Hbin_in_systematic_form &= (mat_coeff(Hbin, $j, i$) $\equiv 1$);
```
This code is used in section 87.

When returning *Hbin* in systematic form, the corresponding matrix $\boldsymbol{J}$ in $[-\boldsymbol{R}^T|\boldsymbol{J}]$ has the form

$$\boldsymbol{J} = \begin{bmatrix} 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & \cdots & 1 & 0 \\ \vdots & & \vdots & & \vdots \\ 1 & 0 & \cdots & 0 & 0 \end{bmatrix},$$

so we permute the rows to get $[-\boldsymbol{R}^T|\boldsymbol{I}]$.

89 ⟨ Change row order 89 ⟩ ≡

```
{
  for (int $i = 0$; $i < (Hbin\to rown + 1)/2$; $++i$)
    for (int $j = 0$; $j < Hbin\to rwdcnt$; $++j$) {
      unsigned long $e = Hbin\to elem[i * Hbin\to rwdcnt + j]$;
      $Hbin\to elem[i * Hbin\to rwdcnt + j] = Hbin\to elem[(Hbin\to rown - 1 - i) * Hbin\to rwdcnt + j]$;
      $Hbin\to elem[(Hbin\to rown - 1 - i) * Hbin\to rwdcnt + j] = e$;
    }
  if (debug $\geq 3$)
    ⟨ Print parity check matrix H 90 ⟩
}
```
This code is used in section 87.

90 ⟨ Print parity check matrix H 90 ⟩ ≡

```
print_bin_matrix($m * t, n$, Hbin, "Hbin␣IN␣SYSTEMATIC␣FORM:");
```
This code is used in sections 89 and 91.

91 ⟨ Check $GH^T = \mathbf{0}$ 91 ⟩ ≡

```
for (int $i = 0$; $i < n - m * t$; ++$i$)
  for (int $k = 0$; $k < m * t$; ++$k$) {
    int $x = 0$;
    for (int $j = 0$; $j < n$; ++$j$)
      $x$ += mat_coeff(Gbin, $i, j$) $*$ mat_coeff(Hbin, $k, j$);
    if ($x$ & 1) {
      fprintf(stderr, "ERROR:␣GH^T␣!=␣0␣(%d,%d)\n", $i, k$);
      ⟨ Print parity check matrix H 90 ⟩⟨ Print generator matrix G 93 ⟩$i = n - m * t$;
      $k = m * t$;
      continue;
    }
  }
```

This code is used in section 92.

As the parity check matrix has form $\boldsymbol{H} = [\boldsymbol{R}^T | \boldsymbol{I}_{n-mt}] \in \mathbb{F}_2^{(n-mt)\times n}$, the public generator matrix in systematic form has form $\boldsymbol{G} = [\boldsymbol{I}_{mt} | \boldsymbol{R}] \in \mathbb{F}_2^{mt\times n}$. The redundant part $\boldsymbol{R} \in \mathbb{F}_2^{mt\times(n-mt)}$ is a quasi-dyadic matrix.

92 ⟨ Use H to build generator matrix G in systematic form 92 ⟩ ≡

```
if (Gbin ≠ Λ) mat_free(Gbin);
Gbin = mat_ini($n - m * t, n$);        /* n-mt = k */
if (Gbin ≠ Λ) {
  mat_set_to_zero(Gbin);
  for (int $i = 0$; $i < n - m * t$; ++$i$)
    for (int $j = 0$; $j < n$; ++$j$)
      if ($i \equiv j$)
        mat_set_coeff_to_one(Gbin, $i, j$);
      else
      if ($j \geq n - m * t$)
        if (mat_coeff(Hbin, $j - n + m * t, i$))
          mat_set_coeff_to_one(Gbin, $i, j$);
  ⟨ Check $GH^T = \mathbf{0}$ 91 ⟩;
  if (debug ≥ 3)
    ⟨ Print generator matrix G 93 ⟩;
}
else {
  fprintf(stderr, "INFO:␣mat_ini␣FAILED␣FOR␣Gbin\n");
  continue;
}
```

This code is used in section 76.

93 ⟨ Print generator matrix G 93 ⟩ ≡

```
print_bin_matrix($n - m * t, n$, Gbin, "Gbin␣IN␣SYSTEMATIC␣FORM:");
```

This code is used in sections 91, 92, and 183.

8 The encryption step

8.1 The fast Walsh-Hadamard transform (FWHT) and the dyadic convolution

As noted in section (2.4.8), remark (2.4.43) provides an efficient way to perform a vector-matrix product in the form of $\boldsymbol{u}\boldsymbol{H}_k$ for $\boldsymbol{u} \in \mathbb{F}^r$. The algorithm is shown in *fwht*. For the purposes of this thesis, a datatype of **int** provides enough space for the lifting to $\mathbb{Z}$. In case the datatype has to be extended, *fwht* will be more complicated as well.

Besides *fwht*, a more direct way to compute $\boldsymbol{u}\boldsymbol{H}_k$ is given with the *vm*-functions (see section (8.1.2)). Although only *vm4* is used in the actual implementation, other versions for other datatypes are given as well to clarify the pattern underlying the *vm*-functions.

96 ⟨`fwht.h` 96⟩ ≡

```
#ifndef FWHT_H_INCLUDED
#define FWHT_H_INCLUDED
#include <inttypes.h>
   typedef int fwht_t;
   extern fwht_t *fwht(unsigned k, fwht_t *u);
   extern uint8_t vm4(uint8_t v, uint8_t m);
   extern uint8_t vm2x4(uint8_t v, uint8_t m);
   extern uint8_t vm8(uint8_t v, uint8_t m);
   extern uint16_t vm16(uint16_t v, uint16_t m);
   extern uint32_t vm32(uint32_t v, uint32_t m);
#endif
```

8.1.1 $\boldsymbol{u}\boldsymbol{H}_k$ via the fast Walsh-Hadamard transform

98 ⟨`fwht.c` 98⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include "fwht.h"
   ⟨fwht functions 99⟩
```

The fast *Walsh − Hadamard transform* takes as input $k \in \mathbb{N}$, $\boldsymbol{u} \in \mathbb{F}^r$ with $r = 2^k$ and is done in characteristic $\neq 2$. *fwht* is a straight-forward implementation of (2.4.49).

99 ⟨*fwht* functions 99⟩ ≡

```
**fwht_t** $*fwht$(**unsigned** $k$, **fwht_t** $*u$)
{
  **register unsigned const** $r = (1 \ll k)$;
  **register unsigned** $i,\ j,\ h,\ s = k + 1$;
  **register unsigned** $d = 1$;
  **while** $(--s)$ {
    $h = d$;
    $d \ll= 1$;
    **for** $(i = 0;\ i < r;\ i += d)$ {
      **for** $(j = 0;\ j < h;\ ++j)$ {
        **register fwht_t** $*p = u + i + j$;
        **register fwht_t** $*q = u + i + j + h$;
        **register fwht_t const** $v = *p$;
        **register fwht_t const** $w = *q$;
        $*p = v + w$;
        $*q = v - w$;
      }
    }
  }
  **return** $u$;
}
```

See also sections 101, 102, 103, and 104.

This code is used in section 98.

8.1.2 $\boldsymbol{u H}_k$ directly via *vm*-functions

To illustrate the idea of the *vm*-functions, consider the dyadic matrix $\boldsymbol{H}_k$ with $k = 8$:

$$\boldsymbol{H}_k = \begin{bmatrix} A & B & C & D & E & F & G & H \\ B & A & D & C & F & E & H & G \\ C & D & A & B & G & H & E & F \\ D & C & B & A & H & G & F & E \\ E & F & G & H & A & B & C & D \\ F & E & H & G & B & A & D & C \\ G & H & E & F & C & D & A & B \\ H & G & F & E & D & C & B & A \end{bmatrix}.$$

If we interpret the characters as bits, we see that the second row is the first one, where the bits have been swapped. The third row is like the first one, where nyps[24] have been swapped. The fourth one is like the third one, where the bits have been swapped. Finally, the fifth row is again like the first one, where nybbles[25] have been swapped, and all the previous steps are repeated.

[24]Groups of two bits.

[25]Groups of four bits.

To compute the product $\boldsymbol{u}\boldsymbol{H}_k$ we store the current row. In case the corresponding bit is set in $\boldsymbol{u}$, the current row is added to the end result. The same pattern applies for the other *vm*-functions. The only difference is how much must be swapped when crossing the $(n/2)$.th row.

```
101 ⟨fwht functions 99⟩ +≡
  uint8_t vm8(uint8_t v, uint8_t m)
  {
    int i;
    uint8_t res = 0;
    for (i = 0; i < 8; ++i) {
      int n = i;
      uint8_t r = m;      /* save signature in r */
      if (n ≥ 4) {
        r = (((r & #0F) ≪ 4) | ((r & #F0) ≫ 4));      /* swap nybbles */
        n = n − 4;
      }
      if (n ≥ 2) {
        r = (((r & #33) ≪ 2) | ((r & #CC) ≫ 2));      /* swap nyps */
        n = n − 2;
      }
      if (n ≡ 1) {
        r = (((r & #55) ≪ 1) | ((r & #AA) ≫ 1));      /* swap bits */
      }
      if ((v ≫ (7 − i)) & 1) {      /* bitsetinv, addrow */
        res = res ⊕ r;
      }
    }
    return res;
  }

102 ⟨fwht functions 99⟩ +≡
  uint8_t vm4(uint8_t v, uint8_t m)
  {
    int i;
    uint8_t res = 0;
    for (i = 0; i < 4; ++i) {
      int n = i;
      uint8_t row = m;
      if (n ≥ 2) {
        row = (((row & #33) ≪ 2) | ((row & #CC) ≫ 2));      /* swap nyps */
        n = n − 2;
      }
      if (n ≡ 1) {
        row = (((row & #55) ≪ 1) | ((row & #AA) ≫ 1));      /* swap bits */
        }
        if ((v ≫ (3 − i)) & 1) {
            res = res ⊕ row;
        }
    }
    return res;
}
```

103 ⟨ *fwht* functions 99 ⟩ +≡

```
uint16_t vm16(uint16_t v, uint16_t m)
{
    int i;
    uint16_t res = 0;
    for (i = 0; i < 16; ++i) {
        int n = i;
        uint16_t r = m;
        if (n ≥ 8) {
            r = (((r & #00FF) ≪ 8) | ((r & #FF00) ≫ 8));      /* swap bytes */
            n = n − 8;
        }
        if (n ≥ 4) {
            r = (((r & #0F0F) ≪ 4) | ((r & #F0F0) ≫ 4));      /* swap nybbles */
            n = n − 4;
        }
        if (n ≥ 2) {
            r = (((r & #3333) ≪ 2) | ((r & #CCCC) ≫ 2));      /* swap nyps */
            n = n − 2;
        }
        if (n ≡ 1) {
            r = (((r & #5555) ≪ 1) | ((r & #AAAA) ≫ 1));      /* swap bits */
        }
        if ((v ≫ (15 − i)) & 1) {
            res = res ⊕ r;
        }
    }
    return res;
}
```

104 ⟨ *fwht* functions 99 ⟩ +≡

```
uint32_t vm32(uint32_t v, uint32_t m)
{
    int i;
    uint32_t res = 0;
    for (i = 0; i < 32; ++i) {
      int n = i;
      uint32_t r = m;
      if (n ≥ 16) {
        r = (((r & #0000FFFF) ≪ 16) | ((r & #FFFF0000) ≫ 16));      /* swap half words */
        n = n − 16;
      }
      if (n ≥ 8) {
        r = (((r & #00FF00FF) ≪ 8) | ((r & #FF00FF00) ≫ 8));      /* swap bytes */
        n = n − 8;
      }
      if (n ≥ 4) {
        r = (((r & #0F0F0F0F) ≪ 4) | ((r & #F0F0F0F0) ≫ 4));      /* swap nybbles */
        n = n − 4;
      }
      if (n ≥ 2) {
        r = (((r & #33333333) ≪ 2) | ((r & #CCCCCCCC) ≫ 2));      /* swap nyps */
        n = n − 2;
      }
      if (n ≡ 1) {
        r = (((r & #55555555) ≪ 1) | ((r & #AAAAAAAA) ≫ 1));      /* swap bits */
      }
      if ((v ≫ (31 − i)) & 1) {
        res = res ⊕ r;
      }
    }
    return res;
  }
```

8.1.3 The dyadic convolution

Let $\Delta(\boldsymbol{u})$ and $\Delta(\boldsymbol{v})$ dyadic matrices. Then the dyadic convolution computes $\Delta(\boldsymbol{w}) = \Delta(\boldsymbol{u})\Delta(\boldsymbol{v})$ using the signatures $\boldsymbol{u}$ and $\boldsymbol{v}$ only. There is no need to unfold the dyadic matrices in memory.

106 ⟨dyadic-convolution.h 106⟩ ≡

```
#ifndef DYADIC_CONVOLUTION_H_INCLUDED
#define DYADIC_CONVOLUTION_H_INCLUDED
#include "fwht.h"
  extern fwht_t *dyadic_conv(unsigned k, fwht_t *u, fwht_t *v);
#endif
```

107 ⟨ `dyadic-convolution.c` 107 ⟩ ≡

```
#include "dyadic-convolution.h"
  ⟨ dyadic convolution functions 108 ⟩
```

The input for the dyadic convolution is $k \in \mathbb{N}$, $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^r$ with $r = 2^k$ and $\mathrm{char}(\mathbb{F}) \neq 2$. The output is $\boldsymbol{w} \in \mathbb{F}_r$ such that $\Delta(\boldsymbol{w}) = \Delta(\boldsymbol{u})\Delta(\boldsymbol{v})$.

108 ⟨ *dyadic convolution* functions 108 ⟩ ≡

```
  fwht_t *dyadic_conv(unsigned k, fwht_t *u, fwht_t *v)
  {
    register fwht_t const *const r = &u[1 << k];
    register fwht_t *p = fwht(k, u);      /* compute $\hat{\boldsymbol{u}} = \boldsymbol{u}\boldsymbol{H}_k$ via FWHT */
    register fwht_t *q = fwht(k, v);      /* compute $\hat{\boldsymbol{v}} = \boldsymbol{v}\boldsymbol{H}_k$ via FWHT */
    while (p < r) {      /* $\hat{w}_i = \hat{u}_i\hat{v}_i$ */
      *p *= *q;
      ++p;
      ++q;
    }
    p = fwht(k, u);      /* compute $\boldsymbol{w} = \hat{\boldsymbol{w}}\boldsymbol{H}_k$ via FWHT */
    while (p < r) {      /* $\boldsymbol{w} = 2^{-k}\boldsymbol{w}$ */
      *p >>= k;
      ++p;
    }
    return u;      /* return $\boldsymbol{w}$ */
  }
```

This code is used in section 107.

9 The decryption step

The original McEliece scheme was based on binary irreducible Goppa codes. An efficient decoder for codes of this kind is based on Patterson's algorithm [24]. However, for separable binary Goppa codes Patterson's algorithm is not directly usable, and only recently a generalization of Patterson's algorithm has been published [5] [26]. Because Goppa codes are special alternant codes, alternant decoders can be used to decode them. The classical decoder presented now turns out to be slower slower than Patterson's, but it is still in widespread use. As MacWilliams and Sloane quote ([20], p. 369):

"Nevertheless, decoding using the Euclidean algorithm is by far the simplest to understand, and is certainly at least comparable in speed with the other methods (for $n < 10^6$) and so it is the method we prefer."

The following exposition is according to [20, 26].

[26] Which is beyond the scope of this thesis.

110 ⟨ alternant-decode.h 110 ⟩ ≡

```
#ifndef ALTERNANT_DECODE_H_INCLUDED
#define ALTERNANT_DECODE_H_INCLUDED
#include "gf.h"
#include "poly.h"
  extern gf_t*cons_parity_check(int n, int t, poly_t g2, const gf_t *L, int debug);
  extern poly_t cons_syndrome_polynomial(int n, int tt, const gf_t *cw, const gf_t *H2T);
  extern void solve_key_equation(poly_t *sigma, poly_t *omega, poly_t S, poly_t g2, int t);
#endif
```

111 ⟨ alternant-decode.c 111 ⟩ ≡

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "alternant-decode.h"
#include "utils.h"
  ⟨ decode functions 113 ⟩
```

9.1 The setup

Construct the parity check matrix as desribed in (5.1.3). Inputs are the dimensions n and $2t$, the square of the Goppa polynomial $g(X)$ and the support $\boldsymbol{L}$. Output is $\boldsymbol{H}' \equiv \boldsymbol{H2T}$.

113 ⟨ *decode* functions 113 ⟩ ≡

```
  gf_t *cons_parity_check(int n, int tt, poly_t g2, const gf_t *L, int debug)
  {
    gf_t e[n], c[n];
    gf_t *H2 = calloc(tt * n, sizeof (*H2));
    gf_t *H2T = calloc(tt * n, sizeof (*H2T));
    if (H2 ∧ H2T) {
      for (int j = 0; j < n; ++j) {
        e[j] = poly_eval(g2, L[j]);
        e[j] = gf_inv(e[j]);
        c[j] = 0;
      }
      for (int i = 0; i < tt; ++i) {
        for (int j = 0; j < n; ++j) {
          c[j] = gf_mul(c[j], L[j]);
          c[j] = gf_add(c[j], poly_coeff(g2, tt − i));
        }
        for (int j = 0; j < n; ++j)
```

```
            H2[$i * n + j$] = gf_mul($e[j], c[j]$);
        }
        for (int $i = 0$; $i < tt$; ++$i$)
            for (int $j = 0$; $j < n$; ++$j$)
                H2T[$j * tt + i$] = H2[$i * n + j$];
        free(H2);
        if (debug $\geq 3$) {
            ⟨ Print parity check H2T 114 ⟩;
        }
    }
    return H2T;
}
```
See also sections 116 and 118.
This code is used in section 111.

114 ⟨ Print parity check H2T 114 ⟩ ≡
```
print_matrix(tt, $n$, H2T, "H2T");
```
This code is used in section 113.

Call *cons_parity_check* in the *main* function.

115 ⟨ *main* construct parity check matrix H2T for the private decoder 115 ⟩ ≡
```
H2T = cons_parity_check($n, 2 * t$, g2, $L$, debug);
```
This code is used in section 174.

9.2 Construct the syndrome polynomial

The syndrome polynomial of a received vector $\boldsymbol{u} \in \mathbb{F}_2^n$ can be computed either directly using the definition,

$$S(X) \equiv \sum_{i=0}^{n-1} \frac{u_i}{X - L_i} \equiv \sum_{i=0}^{n-1} \frac{e_i}{X - L_i} \mod g(X)$$

or, as we do, using the canonical parity check matrix $\boldsymbol{H}$, see (2.4.26). The received vector $\boldsymbol{u}$ has to be multiplied from the left onto $\boldsymbol{H}^T \in \mathbb{F}_{2^m}^{n \times 2t}$,

$$\boldsymbol{u}\boldsymbol{H}^T = [S_{2t-1}, \ldots, S_0] \tag{9.2.1}$$

where the S_i with $0 \leq i \leq 2t - 1$ represent the coefficients of the syndrome polynomial.

116 ⟨ *decode* functions 113 ⟩ +≡
```
poly_t cons_syndrome_polynomial(int $n$, int tt, const gf_t *$u$, const gf_t *H2T)
{ gf_t $s$[tt];
    poly_t $S$ = poly_alloc(tt $- 1$);
    for (int $i = 0;\ i < tt;\ ++i$)
      $s[i] = 0$;
    for (int $i = 0;\ i < tt;\ ++i$)
      for (int $j = 0;\ j < n;\ ++j$) {
        gf_t $c$ = gf_mul($u[j]$, H2T[$j * tt + i$]);
        $s[i]$ = gf_add($s[i], c$);
      }
    for (int $i = 0;\ i < tt;\ ++i$)
      poly_set_coeff($S, tt - 1 - i, s[i]$);
    poly_calcule_deg($S$);
    return $S$;
  }
```

117 ⟨ *main* compute the syndrome polynomial 117 ⟩ ≡

```
  SyM = cons_syndrome_polynomial($n, 2 * t$, cw, H2T);
```

This code is used in section 185.

9.3 Solve the key equation

$$\omega(X) \equiv \sigma(X)S(X) \mod g(X)$$

118 ⟨ *decode* functions 113 ⟩ +≡

```
  void solve_key_equation(poly_t *poly_sigma, poly_t *poly_omega, poly_t $S$, poly_t g2, int $t$)
  {
    poly_eeaux(poly_sigma, poly_omega, $S$, g2, $t$);
  }
```

119 ⟨ *main* solve the key equation $\omega(X) = \sigma(X)S(X) \mod g(X)$ 119 ⟩ ≡

```
  solve_key_equation(&poly_sigma, &poly_omega, SyM, g2, $t$);
```

This code is used in section 185.

9.4 Find the error positions and correct codeword

Finding and correcting the error positions as done in the *main* function using the support $\boldsymbol{L}$. Note that in general much more sophisticated methods are deployed like Chien search [9] or Berlekamp's trace algorithm [7].

121 ⟨ *main* correct errors 121 ⟩ ≡

```
poly_calcule_deg(poly_sigma);
poly_calcule_deg(poly_omega);
for (int i = 0; i < n; ++i)
  if (poly_eval(poly_sigma, L[i]) ≡ 0)
    cw[i] = (cw[i] ≡ 1) ? 0 : 1;
```

This code is used in section 185.

10 Additional source code

Sections (10.1) and (10.2) are taken from HyMES [29] with minor corrections.

10.1 Polynomials

124 ⟨ poly.h 124 ⟩ ≡

```
#ifndef POLY_H_INCLUDED
#define POLY_H_INCLUDED
  typedef struct polynome {
    int deg;
    int size;
    gf_t * coeff;
  } *poly_t;
#define poly_deg(p) ((p)→deg)
#define poly_size(p) ((p)→size)
#define poly_set_deg(p, d) ((p)→deg = (d))
#define poly_coeff(p, i) ((p)→coeff[i])
#define poly_set_coeff(p, i, a) ((p)→coeff[i] = (a))
#define poly_addto_coeff(p, i, a) ((p)→coeff[i] = gf_add((p)→coeff[i], (a)))
#define poly_multo_coeff(p, i, a) ((p)→coeff[i] = gf_mul((p)→coeff[i], (a)))
#define poly_tete(p) ((p)→coeff[(p)→deg])
  extern int poly_calcule_deg(poly_t p);
  extern void poly_set(poly_t p, poly_t q);
  extern void poly_set_to_zero(poly_t p);
  extern poly_t poly_alloc(int d);
  extern poly_t poly_copy(poly_t p);
  extern void poly_free(poly_t p);
  extern poly_t poly_mul(poly_t p, poly_t q);
  extern gf_t poly_eval(poly_t p, gf_t a);
  extern void poly_eeaux(poly_t *u, poly_t *v, poly_t p, poly_t g, int t);
#endif
```

125 ⟨poly.c 125⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gf.h"
#include "poly.h"
   ⟨ Static poly functions 133 ⟩
   ⟨ Exported poly functions 126 ⟩
```

126 ⟨ Exported *poly* functions 126 ⟩ ≡

```
   poly_t poly_alloc(int d) { poly_t p;
         p = (poly_t) malloc(sizeof(struct polynome));
         p→deg = −1;
         p→size = d + 1; p→coeff = ( gf_t ∗ ) calloc(p→size, sizeof (gf_t));
         return p; }
```

See also sections 127, 128, 129, 130, 131, 132, 134, and 135.

This code is used in section 125.

127 ⟨ Exported *poly* functions 126 ⟩ +≡

```
   void poly_free(poly_t p)
   {
      free(p→coeff);
      free(p);
   }
```

128 ⟨ Exported *poly* functions 126 ⟩ +≡

```
   void poly_set(poly_t p, poly_t q)
   {      /∗ copy q in p ∗/
      int d = p→size − q→size;
      if (d < 0) {
         memcpy(p→coeff, q→coeff, p→size ∗ sizeof (gf_t));
         poly_calcule_deg(p);
      }
      else {
         memcpy(p→coeff, q→coeff, q→size ∗ sizeof (gf_t));
         memset(p→coeff + q→size, 0, d ∗ sizeof (gf_t));
         p→deg = q→deg;
      }
   }
```

129 ⟨Exported *poly* functions 126⟩ +≡

```
void poly_set_to_zero(poly_t p)
{
   memset(p→coeff, 0, p→size ∗ sizeof (gf_t));
   p→deg = −1;
}
```

130 ⟨Exported *poly* functions 126⟩ +≡

```
poly_t poly_copy(poly_t p) { poly_t q;
      q = (poly_t) malloc(sizeof(struct polynome));
      q→deg = p→deg;
      q→size = p→size; q→coeff = ( gf_t ∗ ) calloc(q→size, sizeof (gf_t));
      memcpy(q→coeff, p→coeff, p→size ∗ sizeof (gf_t));
      return q; }
```

131 ⟨Exported *poly* functions 126⟩ +≡

```
int poly_calcule_deg(poly_t p)
{
   int d = p→size − 1;
   while ((d ≥ 0) ∧ (p→coeff[d] ≡ gf_zero( ))) −−d;
   p→deg = d;
   return d;
}
```

132 ⟨Exported *poly* functions 126⟩ +≡

```
poly_t poly_mul(poly_t p, poly_t q)
{
   int i, j, dp, dq;
   poly_t r;
   poly_calcule_deg(p);
   poly_calcule_deg(q);
   dp = poly_deg(p);
   dq = poly_deg(q);
   r = poly_alloc(dp + dq);
   for (i = 0; i ≤ dp; ++i)
      for (j = 0; j ≤ dq; ++j) poly_addto_coeff(r, i + j, gf_mul(poly_coeff(p, i), poly_coeff(q, j)));
   poly_calcule_deg(r);
   return (r);
}
```

133 ⟨ Static *poly* functions 133 ⟩ ≡

```
gf_t poly_eval_aux(gf_t * coeff, gf_t a, int d)
{
   gf_t b;
   b = coeff[d--];
   for ( ; d ≥ 0; --d)
      if (b ≠ gf_zero( ))  b = gf_add(gf_mul(b, a), coeff[d]);
      else  b = coeff[d];
   return b;
}
```

This code is used in section 125.

134 ⟨ Exported *poly* functions 126 ⟩ +≡

```
gf_t poly_eval(poly_t p, gf_t a)
{
   return poly_eval_aux(p→coeff, a, poly_deg(p));
}
```

The extended Euclidean algorithm. General assumption: $\deg g \geq \deg p$.

135 ⟨ Exported *poly* functions 126 ⟩ +≡

```
void poly_eeaux(poly_t *u, poly_t *v, poly_t p, poly_t g, int t)
{
   int i, j, dr, du, delta;
   gf_t a;
   poly_t aux, r0, r1, u0, u1;
   dr = poly_deg(g);       /* $r_0 := g, r_1 := p, u_0 := 0, u_1 := 1$ */
   r0 = poly_alloc(dr);
   r1 = poly_alloc(dr - 1);
   u0 = poly_alloc(dr - 1);
   u1 = poly_alloc(dr - 1);
   poly_set(r0, g);
   poly_set(r1, p);
   poly_set_to_zero(u0);
   poly_set_to_zero(u1);
   poly_set_coeff(u1, 0, gf_unit( ));
   poly_set_deg(u1, 0);
      /* invariants: $r_1 = u_1 * p + v_1 * g r_0 = u_0 * p + v_0 * g$ and $\deg u_1 = \deg g - \deg r_0$. It stops
        when $\deg r_1 < t$ $(\deg r_0 \geq t)$. And therefore $\deg u_1 = \deg g - \deg r_0 < \deg g - t$ */
   du = 0;
   dr = poly_deg(r1);
   delta = poly_deg(r0) - dr;
   while (dr ≥ t) {
      for (j = delta; j ≥ 0; --j) {
        a = gf_div(poly_coeff(r0, dr + j), poly_coeff(r1, dr));
        if (a ≠ gf_zero()) {      /* u0(z) <- u0(z) + a * u1(z) * z^j */
          for (i = 0; i ≤ du; ++i) {
            poly_addto_coeff(u0, i + j, gf_mul_fast(a, poly_coeff(u1, i)));
          }      /* r0(z) <- r0(z) + a * r1(z) * z^j */
          for (i = 0; i ≤ dr; ++i) poly_addto_coeff(r0, i + j, gf_mul_fast(a, poly_coeff(r1, i)));
        }
      }      /* exchange */
      aux = r0;
      r0 = r1;
      r1 = aux;
      aux = u0;
      u0 = u1;
      u1 = aux;
      du = du + delta;
      delta = 1;
      while (poly_coeff(r1, dr − delta) ≡ gf_zero()) delta++;
      dr −= delta;
   }
   poly_set_deg(u1, du);
   poly_set_deg(r1, dr);      /* return u1 and r1; */
   *u = u1;
   *v = r1;
   poly_free(r0);
   poly_free(u0);
}
```

10.2 Matrix functions

137 ⟨matrix.h 137⟩ ≡

```
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED
#define BITS_PER_LONG (8 * sizeof(unsigned long))
#define mat_coeff(A, i, j)
  (((A)→elem[(i) * A→rwdcnt + (j)/BITS_PER_LONG] ≫ (j % BITS_PER_LONG)) & 1)
#define mat_set_coeff_to_one(A, i, j)
  ((A)→elem[(i) * A→rwdcnt + (j)/BITS_PER_LONG] |= (1UL ≪ ((j) % BITS_PER_LONG)))
#define mat_change_coeff(A, i, j)
  ((A)→elem[(i) * A→rwdcnt + (j)/BITS_PER_LONG] ⊕= (1UL ≪ ((j) % BITS_PER_LONG)))
#define mat_set_to_zero(R)memset((R)→elem, 0, (R)→alloc_size);
  typedef struct matrix {
    int rown;      /* number of rows */
    int coln;      /* number of columns */
```

```
        int rwdcnt;      /* number of words in a row */
        int alloc_size;      /* number of allocated bytes */
        unsigned long *elem;      /* row index */
    } *binmat_t;
    extern binmat_t mat_ini(int rown, int coln);
    extern binmat_t mat_ini_from_string(int rown, int coln, const unsigned char *s);
    extern void mat_free(binmat_t A);
    extern binmat_t mat_copy(binmat_t A);
    extern binmat_t mat_rowxor(binmat_t A, int a, int b);
    extern int *mat_rref(binmat_t A);
    extern void mat_vec_mul(unsigned long *cR, unsigned char *x, binmat_t A);
    extern binmat_t mat_mul(binmat_t A, binmat_t B);
#endif
```

138 ⟨ matrix.c 138 ⟩ ≡

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <assert.h>
#include "matrix.h"
    ⟨ Exported matrix functions 139 ⟩
```

139 ⟨ Exported **matrix** functions 139 ⟩ ≡

```
    binmat_t mat_ini(int rown, int coln)
    { binmat_t A;
        A = (binmat_t) malloc(sizeof (*A));
        if (A ≠ Λ) {
            A→coln = coln;
            A→rown = rown;
            A→rwdcnt = (1 + (coln − 1)/BITS_PER_LONG);
            A→alloc_size = rown * A→rwdcnt * sizeof(unsigned long);
            A→elem = (unsigned long *) malloc(A→alloc_size);
        }
        return A;
    }
```

See also sections 140, 141, 142, 143, 144, 145, and 146.

This code is used in section 138.

140 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
binmat_t mat_ini_from_string(int rown, int coln, const unsigned char *s)
{    /* assumes s has the proper allocated size */
  binmat_t A;
  A = (binmat_t) malloc(sizeof(struct matrix));
  A→coln = coln;
  A→rown = rown;
  A→rwdcnt = (1 + (coln − 1)/BITS_PER_LONG);
  A→alloc_size = rown * A→rwdcnt * sizeof(unsigned long);
  A→elem = (unsigned long *) s;
  return A;
}
```

141 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
void mat_free(binmat_t A)
{
  free(A→elem);
  free(A);
}
```

142 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
binmat_t mat_copy(binmat_t A)
{    /* copying matrix (for the form [G | I] . . . ) */
  binmat_t X;
  int i;
  X = mat_ini(A→rown, A→coln);      /* initialize the matrix */
  for (i = 0; i < ((A→rwdcnt) * (A→rown)); i++) X→elem[i] = A→elem[i];
  return (X);
}
```

143 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
binmat_t mat_rowxor(binmat_t A, int a, int b)
{
  int i;
  assert(a ≥ 0 ∧ b ≥ 0);
  assert(A ∧ a < A→rown ∧ b < A→rown);
  for (i = 0; i < A→rwdcnt; i++) A→elem[a * A→rwdcnt + i] ⊕= A→elem[b * A→rwdcnt + i];
  return A;
}
```

mat_rref transforms the matrix A into systematic form. It contains a fix by the author checking for valid array indices. Otherwise the function did overwrite the pointer *perm* in some cases.

```
144 ⟨ Exported matrix functions 139 ⟩ +≡
  int *mat_rref(binmat_t A)
  {     /* the matrix is reduced from LSB... (from right) */
    int i, j, failcnt, findrow, idx, max = A→coln − 1;
    int *perm;
    perm = malloc(A→coln * sizeof(int));
    if (perm) {
      for (i = 0; i < A→rown; i++) perm[i] = i;     /* initialize permutation */
      failcnt = 0;
      for (i = 0; i < A→rown; i++, max−−) {
        findrow = 0;
        for (j = i; j < A→rown; j++) {
          if (mat_coeff(A, j, max)) {     /* A− > elem[(j * A− > coln) + max] */
                /* max–; */
            if (i ≠ j)     /* not needed as ith row is 0 and jth row is 1. */
              A = mat_rowxor(A, i, j);     /* xor to the row. (swap)? */
            findrow = 1;
            break;
          }     /* largest value found (end if) */     /* break; */
        }
        if (¬findrow) {
              /* if no row with a 1 found then swap last column and the column with no 1 down. */
          idx = A→coln − A→rown − 1 − failcnt;     /* bug fix: check idx */
          if (idx < 0 ∨ idx ≥ A→coln) {
            free(perm);
            return Λ;
          }
          perm[idx] = max;
          failcnt++;
          if (¬max) {
            free(perm);
            return Λ;
          }
          i−−;
        }
        else {
          idx = i + A→coln − A→rown;
          if (idx < 0 ∨ idx ≥ A→coln) {
            free(perm);
            return Λ;
          }
          perm[idx] = max;
          for (j = i + 1; j < A→rown; j++) {     /* fill the column downwards with 0's */
            if (mat_coeff(A, j, (max)))     /* A− > elem[j * A− > coln + max + 1] */
              A = mat_rowxor(A, j, i);     /* check the arg. order. */
            }
            for (j = i − 1; j ≥ 0; j−−) {        /* fill the column with 0's upwards too. */
               if (mat_coeff(A, j, (max)))        /* A− > elem[j * A− > coln + max + 1] */
                  A = mat_rowxor(A, j, i);
            }
         }
      }      /* end for(i) */
   }
   return (perm);
}
```

145 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
void mat_vec_mul(unsigned long *cR, unsigned char *x, binmat_t A)
{
   int i, j;
   unsigned long *pt;
   memset(cR, 0, A→rwdcnt * sizeof(long));
   pt = A→elem;
   for (i = 0; i < A→rown; i++) {        /* extract the first column in the form of char array. */
      if ((x[i/8] ≫ (i % 8)) & 1)
         for (j = 0; j < A→rwdcnt; ++j)  cR[j] ⊕= *pt++;
      else  pt += A→rwdcnt;
   }
}
```

146 ⟨ Exported **matrix** functions 139 ⟩ +≡

```
binmat_t mat_mul(binmat_t A, binmat_t B)
{
   binmat_t C;
   int i, j, k;
   if (A→coln ≠ B→rown)  exit(0);
   C = mat_ini(A→rown, B→coln);
   memset(C→elem, 0, C→alloc_size);
   for (i = 0; i < A→rown; i++)
      for (j = 0; j < B→coln; j++)
         for (k = 0; k < A→coln; ++k)
            if (mat_coeff(A, i, k) ∧ mat_coeff(B, k, j))  mat_change_coeff(C, i, j);
   return C;
}
```

10.3 Utilities

148 ⟨ `utils.h` 148 ⟩ ≡

```
#ifndef UTILS_H_INCLUDED
#define UTILS_H_INCLUDED
#include "gf.h"
#include "matrix.h"
#define MAX_LINE 80
  ⟨ Static utils inline functions 149 ⟩
  extern int read_input(char *fname, int *m, int *n, int *t, int *debug);
  extern void print_matrix(int h, int w, const gf_t *mat, const char *name);
  extern void print_bin_matrix(int h, int w, const binmat_t mat, const char *name);
  extern void next_error_vector(const int *e_old, int *e_new, int *src, int *rndm, int *perm, int n);
#endif
```

149 ⟨ Static *utils* inline functions 149 ⟩ ≡

```
  inline static unsigned rev(unsigned x)
  {    /* reverse the bits contained in x,  [37], p. 102. */
    x = (x & #55555555) ≪ 1 | (x & #AAAAAAAA) ≫ 1;
    x = (x & #33333333) ≪ 2 | (x & #CCCCCCCC) ≫ 2;
    x = (x & #0F0F0F0F) ≪ 4 | (x & #F0F0F0F0) ≫ 4;
    x = (x & #00FF00FF) ≪ 8 | (x & #FF00FF00) ≫ 8;
    x = (x & #0000FFFF) ≪ 16 | (x & #FFFF0000) ≫ 16;
    return x;
  }
```

See also sections 150, 151, and 152.

This code is used in section 148.

150 ⟨ Static *utils* inline functions 149 ⟩ +≡

```
  inline static unsigned pop(unsigned x)
  {    /* count the 1 bits contained in x,  [37], p. 65. */
    x = (x & #55555555U) + ((x ≫ 1) & #55555555U);
    x = (x & #33333333U) + ((x ≫ 2) & #33333333U);
    x = (x & #0F0F0F0FU) + ((x ≫ 4) & #0F0F0F0FU);
    x = (x & #00FF00FFU) + ((x ≫ 8) & #00FF00FFU);
    x = (x & #0000FFFFU) + ((x ≫ 16) & #0000FFFFU);
    return x;
  }
```

151 ⟨ Static *utils* inline functions 149 ⟩ +≡

```
  inline static unsigned long long rdtsc() {
      unsigned long long x;
      __asm__ volatile (".byte␣0x0f,␣0x31": "=A"(x));
          return x; }
```

152 ⟨ Static *utils* inline functions 149 ⟩ +≡

```
static inline void swap(gf_t * x, gf_t * y)
{ gf_t t = *x;
  *x = *y;
  *y = t;
}
```

153 ⟨ `utils.c` 153 ⟩ ≡

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include "utils.h"
  ⟨ Static utils functions 154 ⟩
  ⟨ Exported utils functions 156 ⟩
```

Procedure *open_input* opens the input file *filename*. In case of an error, an error message will be printed on the terminal including some additional information and the application be terminated. Otherwise, a valid file pointer will be returned.

154 ⟨ Static *utils* functions 154 ⟩ ≡

```
static FILE *open_input(char *filename)
{
  FILE *f = Λ;
  errno = 0;
  if (filename ≡ Λ)      /* fopen sometimes has problems with null pointers. */
    filename = "\0";
  if ((f = fopen(filename, "r")) ≡ Λ) {
    fprintf(stderr, "%s(\"%s\")␣failed:␣%s\n", __func__, filename, strerror(errno));
    exit(EXIT_FAILURE);
  }
  return f;
}
```

See also section 155.

This code is used in section 153.

The procedure *close_file* closes the file pointer f. In case an error is detected, the error cause is printed on the terminal and the application terminated.

155 ⟨ Static *utils* functions 154 ⟩ +≡

```
static int close_file(FILE *f)
{
  int s;
```

```
    if (f ≡ Λ) return 0;
    errno = 0;
    s = fclose(f);      /* fclose returns EOF if an error is detected, */
    if (s ≡ EOF) {      /* otherwise it returns zero. */
      fprintf(stderr, "%s␣failed:␣%s\n", __func__, strerror(errno));
      exit(EXIT_FAILURE);
    }
    return s;
  }
```

The procedure *read_input* reads the input file of the application. Entries in the input file consist of key=value pairs, one per line. Comments start with `'#'`. Empty lines are allowed. After closing the input file, a check of the read parameters will be done. In case the parameters are not in valid ranges, the application will be terminated.

156 ⟨ Exported *utils* functions 156 ⟩ ≡

```
  int read_input(char *fname, int *m, int *n, int *t, int *debug)
  {
    FILE *input = open_input(fname);      /* open input file */
    ⟨ Read input file 157 ⟩
    close_file(input);      /* close input file */
    return 1;
  }
```

See also sections 163, 164, and 165.

This code is used in section 153.

157 ⟨ Read input file 157 ⟩ ≡

```
  char * line = malloc(MAX_LINE);
  while ( fgets ( line , MAX_LINE, input ) ) {
  ⟨ Skip C comments 158 ⟩
  ⟨ Terminate each line with '#' 159 ⟩
  ⟨ Fetch the contents of the current line before the first '#' 160 ⟩
  ⟨ Split the line at '=' and search for key = value pairs 161 ⟩
  }free ( line ) ;
```

This code is used in section 156.

CWEB produces section numbers with the following format /*<number>:*/.

158 ⟨ Skip C comments 158 ⟩ ≡

```
  if ( line [0] ≡ '/' ∧ line [1] ≡ '*' ) continue;
```

This code is used in section 157.

159 ⟨ Terminate each line with '#' 159 ⟩ ≡

```
size_t const len = strnlen ( line , MAX_LINE ) ;
line [len − 1] = '#';
```

This code is used in section 157.

160 ⟨ Fetch the contents of the current line before the first '#' 160 ⟩ ≡

```
char ∗data = strchr ( line , '#' ) ;
∗data = '\0';
```

This code is used in section 157.

161 ⟨ Split the line at '=' and search for *key* = *value* pairs 161 ⟩ ≡

```
char ∗sp = strchr ( line , '=' ) ;
if (sp) {
∗sp = '\0';
char ∗key = line ;
char ∗val = sp + 1;
⟨ Search for key − value pairs. Initialize parameters. 162 ⟩}
```

This code is used in section 157.

162 ⟨ Search for *key* − *value* pairs. Initialize parameters. 162 ⟩ ≡

```
if (strstr(key, "extension-degree-m"))
  ∗m = (unsigned int) strtol(val, (char ∗∗) Λ, 10);
else if (strstr(key, "code-length-n"))
  ∗n = (unsigned int) strtol(val, (char ∗∗) Λ, 10);
else if (strstr(key, "correctable-errors-t"))
  ∗t = (unsigned int) strtol(val, (char ∗∗) Λ, 10);
else if (strstr(key, "debug"))
  ∗debug = (unsigned int) strtol(val, (char ∗∗) Λ, 10);
```

This code is used in section 161.

163 ⟨ Exported *utils* functions 156 ⟩ +≡

```
void print_matrix(int m, int n, const gf_t ∗mat, const char ∗name)
{
  printf("\n%s\n", name);       /∗ print  mat ∈ F_q^{m×n} ∗/
  for (int i = 0; i < m; ++i) {
    if (i % m ≡ 0) printf("\n");
    for (int j = 0; j < n; ++j) {
      if (j % m ≡ 0) printf("+␣");
      printf("%04x␣", mat[i ∗ n + j]);
    }
    printf("\n");
  }
}
```

164 ⟨ Exported *utils* functions 156 ⟩ +≡

```
void print_bin_matrix(int $m$, int $n$, const binmat_t mat, const char *name)
{
  printf("\n%s\n", name);      /* print $mat \in \mathbb{F}_2^{m \times n}$ */
  for (int $i = 0$; $i < m$; $++i$) {
    if ($i$ % $m \equiv 0$) printf("\n");
    for (int $j = 0$; $j < n$; $++j$) {
      if ($j$ % $m \equiv 0$) printf("+␣");
      printf("%ld␣", mat_coeff(mat, $i$, $j$));
    }
    printf("\n");
  }
}
```

e_old is an array of length n, staring with t 1's, followed by $n - t$ 0's:

$$e_{old} = [\underbrace{1, 1, \ldots, 1}_{t}, \underbrace{0, 0, \ldots, 0}_{n-t}].$$

Each new error vector *e_new* is just a random permutation of *e_old*. To perform this random permutation, *next_error_vector* is an implementation of Algorithm P (see [16], p. 145) in its *inside−out* version [10, 13].

165 ⟨ Exported *utils* functions 156 ⟩ +≡

```
void next_error_vector(const int *e_old, int *e_new, int *src, int *r, int *p, int n) { static int init = 0;
    int $i$;
    if (init $\equiv 0$) {
      init = 1;
      for ($i = 0$; $i < n$; $++i$) src[$i$] = $i$;
      srand((unsigned) rdtsc());
      $r[0] = 0$;
      for ($i = 1$; $i < n$; $++i$) $r[i]$ = rand() % ($i + 1$);
    }
    ⟨ Shuffle permutation 166 ⟩
    for ($i = 0$; $i < n$; $++i$)      /* Update error vector */
      e_new[$i$] = e_old[$p[i]$];
    }
```

166 ⟨ Shuffle permutation 166 ⟩ ≡

```
$p[0]$ = src[0];
for ($i = 1$; $i < n$; $++i$) {
  register int $j = r[i - 1]$;
  $p[i] = p[j]$;
  $p[j]$ = src[$i$];
}
```

```
for (i = 0; i < n; ++i)        /* Update source vector for next round */
    src[i] = p[i];
```
This code is used in section 165.

10.3.1 Input file

The input file contains *key* − *value* pairs. It is written by `CWEB`, resp. *cweave*.

168 ⟨ `input.txt` 168 ⟩ ≡

```
extension − degree − m = 6
code − length − n = 32
correctable − errors − t = 4
debug = 0
```

11 Putting everything together

11.1 The main program

170

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>
#include <math.h>
#include <string.h>
#include "gf.h"
#include "matrix.h"
#include "poly.h"
#include "utils.h"
#include "alternant-decode.h"
#include "fwht.h"
#include "dyadic-convolution.h"
#include "binary-quasi-dyadic-goppa-code.h"
  int main(int argc, char **argv)
  {
    ⟨ main define variables 171 ⟩
    ⟨ main read the application parameters 172 ⟩
    ⟨ main initialize data 173 ⟩
    ⟨ main generate binary Goppa code of type $[n,k,t]$ 174 ⟩
    ⟨ main perform the encode/decode cycle 175 ⟩
    ⟨ main free resources 190 ⟩
    return (EXIT_SUCCESS);
  }
```

171 ⟨ *main* define variables 171 ⟩ ≡

```
  int $m, q$;     /* extension degree, $q = 2^m$ */
  int $n$;     /* code length */
  int $N$;     /* code length, for future use */
  int $t$;     /* number of correctable errors */
  int debug $= 0$;     /* debug flag */
  int $*b = \Lambda$;     /* auxiliary array, see binary_quasi_dyadic_goppa_code */
  gf_t $*h = \Lambda$;     /* auxiliary array, see binary_quasi_dyadic_goppa_code */
  gf_t omega;     /* auxiliary variable, see binary_quasi_dyadic_goppa_code */
  int $C$;     /* auxiliary variable, see binary_quasi_dyadic_goppa_code */
  int bc;     /* auxiliary variable for block count, see binary_quasi_dyadic_goppa_code */
  gf_t $*H = \Lambda$;     /* auxiliary parity check matrix. used to build $\boldsymbol{G}$ */
  gf_t *H2T $= \Lambda$;     /* private parity check matrix $\boldsymbol{H} \in \mathbb{F}_{2^m}^{2t\times n}$ */
  gf_t $*z = \Lambda$;     /* root set for the Goppa polynomial */
  gf_t $*L = \Lambda$;     /* support of the binary Goppa code */
  binmat_t Hbin $= \Lambda$;     /* auxiliary binary parity check matrix $\boldsymbol{H} \in \mathbb{F}_2^{mt\times n}$ in sys. from */
  binmat_t Gbin $= \Lambda$;     /* binary generator matrix $\boldsymbol{G} \in \mathbb{F}_2^{(n-mt)\times n}$ */
  poly_t poly_sigma $= \Lambda$;     /* error locator polynomial $\sigma(X) \in \mathbb{F}_{2^m}[X]$ */
  poly_t poly_omega $= \Lambda$;     /* error evaluator polynomial $\omega(X) \in \mathbb{F}_{2^m}[X]$ */
  poly_t $g = \Lambda$;     /* the Goppa polynomial $g(X) \in \mathbb{F}_{2^m}[X]$ */
  poly_t g2 $= \Lambda$;     /* square of the Goppa polynomial $g(X)^2 \in \mathbb{F}_{2^m}[X]$ */
  poly_t SyM $= \Lambda$;     /* the syndrome polynomial $S(X) \in \mathbb{F}_{2^m}[X]$ */
  gf_t *cw $= \Lambda$;
  gf_t *dyadic_cw $= \Lambda$;
  gf_t *direct_cw $= \Lambda$;     /* auxiliary arrays for testing the encode/decode cycle */
  int *msg $= \Lambda$;     /* auxiliary message array for testing the encode/decode cycle */
  int *e_old $= \Lambda$;
  int *e_new $= \Lambda$;
  int *src $= \Lambda$;
  int *rndm $= \Lambda$;
  int *perm $= \Lambda$;     /* auxiliary arrays for generating the error vector */
```

This code is used in section 170.

172 ⟨ *main* read the application parameters 172 ⟩ ≡

```
if (argc ≠ 2) {
   fprintf(stderr, "usage:␣%s␣<cmd-file>\n", argv[0]);
   exit(−1);
}
if (read_input(argv[1], &m, &n, &t, &debug)) {
   if (debug ≥ 1) {
      printf("extension-degree-m␣.....␣%5d\n", m);
      printf("code-length-n␣..........␣%5d\n", n);
      printf("correctable-errors-t␣...␣%5d\n", t);
      printf("debug␣..................␣%5d\n", debug);
   }
}
```
This code is used in section 170.

For demonstration purposes fix the values for m, t and n.

173 ⟨ *main* initialize data 173 ⟩ ≡
```
#define DEGREE 6
#define CL 32
#define TN 4
   N = CL;
   n = CL;
   t = TN;
   m = DEGREE;
   q = 1 ≪ DEGREE;
   C = (int) floor(q/t);
   cw = calloc(n, sizeof (*cw));
   dyadic_cw = calloc(n, sizeof (*dyadic_cw));
   direct_cw = calloc(n, sizeof (*direct_cw));
   msg = calloc(n − m * t, sizeof (*msg));
   e_old = calloc(n, sizeof (*e_old));
   e_new = calloc(n, sizeof (*e_new));
   src = calloc(n, sizeof (*src));
   rndm = calloc(n, sizeof (*rndm));
   perm = calloc(n, sizeof (*perm));
   H = calloc(t * n, sizeof (*H));
   h = calloc(q, sizeof (*h));
   b = calloc(C, sizeof (*b));
   z = calloc(t, sizeof (*z));
   L = calloc(N, sizeof (*L));
   gf_init(m);      /* initialize the underlying finite field F_{2^m} */
```
This code is used in section 170.

Run Algorithm 2 from [4] to get $\boldsymbol{z}$ and $\boldsymbol{L}$, then assemble the Goppa polynomial $g(X)$ and build $\boldsymbol{H2T}$, which will be used in the private decoder.

174 ⟨ *main* generate binary Goppa code of type $[n,k,t]$ 174 ⟩ ≡
⟨ *main* build quasi-dyadic Goppa code 76 ⟩
⟨ *main* assemble the Goppa polynomial 82 ⟩
⟨ *main* construct parity check matrix `H2T` for the private decoder 115 ⟩
This code is used in section 170.

For the chosen parameters $(m = 6, t = 4, n = 32)$ there are $2^8 = 256$ possible messages $\boldsymbol{m}$. Encode and decode all of them for a randomly generated binary Goppa code. The codewords $\boldsymbol{mG}$ have errors on $t = 4$ positions.

175 ⟨ *main* perform the encode/decode cycle 175 ⟩ ≡
⟨ *main* generate codewords 183 ⟩
⟨ *main* decode forged codewords 185 ⟩
return 0;
This code is used in section 170.

Reverse the bits in **line**. This is due to the data layout of *HyMES*. Additionally, little endian format is assumed.

176 ⟨ *main* reverse signature bits of **line** 176 ⟩ ≡
line . l = *rev* (**line** . l) ;
This code is used in sections 177 and 178.

For some values of t and depending on the available hardware, it might be more convenient to compute the product $\boldsymbol{mG}$ directly using one of the *vm*-functions.

$$\underbrace{\left[\begin{bmatrix}\boxed{1\,0\,0\,1} & \boxed{1\,0\,1\,1}\end{bmatrix}\right]}_{\boldsymbol{m}} \cdot \underbrace{\begin{bmatrix}
1\,0\,0\,0\,0\,0\,0\,0 & \boxed{1\,1\,0\,1} & \boxed{1\,1\,1\,1} & \boxed{0\,0\,1\,0} & \boxed{1\,0\,0\,1} & \boxed{1\,0\,1\,0} & \boxed{0\,1\,0\,1} \\
0\,1\,0\,0\,0\,0\,0\,0 & 1\,1\,1\,0 & 1\,1\,1\,1 & 0\,0\,0\,1 & 0\,1\,1\,0 & 0\,1\,0\,1 & 1\,0\,1\,0 \\
0\,0\,1\,0\,0\,0\,0\,0 & 0\,1\,1\,1 & 1\,1\,1\,1 & 1\,0\,0\,0 & 0\,1\,1\,0 & 1\,0\,1\,0 & 0\,1\,0\,1 \\
0\,0\,0\,1\,0\,0\,0\,0 & 1\,0\,1\,1 & 1\,1\,1\,1 & 0\,1\,0\,0 & 1\,0\,0\,1 & 0\,1\,0\,1 & 1\,0\,1\,0 \\
0\,0\,0\,0\,1\,0\,0\,0 & \boxed{0\,0\,0\,1} & \boxed{0\,1\,1\,0} & \boxed{1\,0\,1\,1} & \boxed{1\,0\,0\,0} & \boxed{0\,1\,1\,0} & \boxed{1\,1\,1\,0} \\
0\,0\,0\,0\,0\,1\,0\,0 & 0\,0\,1\,0 & 1\,0\,0\,1 & 0\,1\,1\,1 & 0\,1\,0\,0 & 1\,0\,0\,1 & 1\,1\,0\,1 \\
0\,0\,0\,0\,0\,0\,1\,0 & 0\,1\,0\,0 & 1\,0\,0\,1 & 1\,1\,1\,0 & 0\,0\,1\,0 & 1\,0\,0\,1 & 1\,0\,1\,1 \\
0\,0\,0\,0\,0\,0\,0\,1 & 1\,0\,0\,0 & 0\,1\,1\,0 & 1\,1\,0\,1 & 0\,0\,0\,1 & 0\,1\,1\,0 & 0\,1\,1\,1
\end{bmatrix}}_{\boldsymbol{G}} \tag{11.1.1}$$

For the product $\boldsymbol{mG}$ only the parts of the two lines of $\boldsymbol{G}$ indicated by the boxes are necessary. There are $2m$ such boxes for these two lines, where m denotes the extension degree. Due to the data layout of *HyMES*, the bits of the lines have to be reversed, however. The message $\boldsymbol{m}$ is also seen as two blocks of 4 bits. Using *vm4* (because of $t = 4$) allows now to perform $\boldsymbol{mG}$ directly without the need to unfold $\boldsymbol{G}$ first.

177 ⟨ *main* compute codeword directly using *vm4* 177 ⟩ ≡
```
union {
  unsigned int l;
  unsigned char c[4];
} line ;
```

```
unsigned char mesg;
unsigned int codeword = k ≪ 24;
for (int j = 0; j < 2; ++j)
{ line . l = (unsigned int)(Gbin→elem[j * t]);
⟨ main reverse signature bits of line 176 ⟩
 mesg = ((unsigned char)(k) ≫ ((j & 1) ? 0 : 4)) & #0F;      /* pick 4 bits of mesg */
  for (int i = 1; i ≤ m; ++i)      /* loop over the blocks of the line */
  { codeword ⊕= vm4 (mesg, line . c[(m − i)/2] ≫ ((i & 1) ? 4 : 0) & #0F ) ≪ (24 − i * 4); }
}
for (int i = 0; i < 32; ++i)      /* save codeword for following equality check */
   direct_cw[i] = (codeword ≫ (31 − i)) & 1;
```
This code is used in section 183.

The product $\boldsymbol{mG}$ can also be computed using the Walsh-Hadamard transform. As with the direct method (see (11.1.1)), only two lines of $\boldsymbol{G}$ are necessary. However, because we have characteristic 2, the transform can not be directly applied, but has to be lifted to $\mathbb{Z}$. This costs some extra memory.

178 ⟨ *main* compute codeword using *Walsh − Hadamard transform* and *dyadic convolution* 178 ⟩ ≡
```
for (int i = 0; i < n; ++i)      /* first 8 bist of the codeword are known */
   dyadic_cw[i] = (i < 8) ? ((k ≫ (7 − i)) & 1) : 0;
for (int j = 0; j < 2; ++j) {
  line . l = (unsigned int)(Gbin→elem[j * t]);
⟨ main reverse signature bits of line 176 ⟩
  mesg = ((unsigned char)(k) ≫ ((j & 1) ? 0 : 4)) & #0F;
⟨ main handle blocks via Walsh-Hadamard transform 179 ⟩}
for (int i = 0; i < n; ++i)      /* reduce the result from ℤ to 𝔽₂ */
   dyadic_cw[i] &= 1;
```
This code is used in section 183.

The vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ receive 4 bits of the message and one block of the current line, respectively. Then the *Walsh − Hadamardtransform* is applied to deliver the vector $\boldsymbol{w}$ such that $\Delta(\boldsymbol{w}) = \Delta(\boldsymbol{u})\Delta(\boldsymbol{v})$.

179 ⟨ *main* handle blocks via Walsh-Hadamard transform 179 ⟩ ≡
```
fwht_t u[4], v[4], *w;
unsigned char mtrx;
for (int i = 1; i ≤ m; ++i) {      /* loop over the blocks of the line */
mtrx = line . c[(m − i)/2] ≫ ((i & 1) ? 4 : 0) & #0F;
for (int ix = 0; ix < 4; ++ix) { u[3 − ix] = (mesg ≫ ix) & 1; v[3 − ix] = (mtrx ≫ ix) & 1; }
⟨ main compute dyadic convolution via Walsh-Hadamard transform 180 ⟩
for (int ix = 0; ix < 4; ++ix) dyadic_cw[4 + i * 4 + ix] += w[ix]; }
```
This code is used in section 178.

Input: $k \in \mathbb{N}, \boldsymbol{u}, \boldsymbol{v} \in \mathbb{F}^r$ with $r = 2^k$ and $char(\mathbb{F}) \neq 2$.
Output: $\boldsymbol{w} = \boldsymbol{u} \vartriangle \boldsymbol{v} \in \mathbb{F}^r$ such that $\Delta(\boldsymbol{u})\Delta(\boldsymbol{v}) = \Delta(\boldsymbol{u} \vartriangle \boldsymbol{v})$.

180 ⟨ *main* compute dyadic convolution via Walsh-Hadamard transform 180 ⟩ ≡

```
    $w = dyadic\_conv(2, u, v)$;
```

This code is used in section 179.

Perform the product $\boldsymbol{bG}$ using the unfolded $\boldsymbol{G}$.

181 ⟨ *main* compute codeword explicitly as vector-matrix product 181 ⟩ ≡

```
    for (int $i = 0$; $i < 8$; $++i$)
        $msg[7 - i] = ((1 \ll i) \,\&\, k) \gg i$;
    for (int $i = 0$; $i < n$; $++i$)
        $cw[i] = 0$;
    for (int $i = 0$; $i < n - m * t$; $++i$)
        if ($msg[i]$)
            for (int $j = 0$; $j < n$; $++j$)
                $cw[j] = (cw[j] + mat\_coeff(Gbin, i, j)) \,\&\, 1$;
```

This code is used in sections 183 and 185.

182 ⟨ *main* check equality of codewords 182 ⟩ ≡

```
    for (int $i = 0$; $i < n$; $++i$)
        if ($((direct\_cw[i] \neq dyadic\_cw[i]) \lor (direct\_cw[i] \neq cw[i]))$)
            fprintf(stderr, "ERROR␣in␣cw.␣comp.:␣%d␣%X,%X,%X", $i$, $direct\_cw[i]$, $dyadic\_cw[i]$,
                    $cw[i]$);
```

This code is used in section 183.

For demonstration purposes, the three different methods of performing $\boldsymbol{mG}$ are shown.

183 ⟨ *main* generate codewords 183 ⟩ ≡

```
    if ($t \equiv 4 \land n \equiv 32 \land m \equiv 6$) {
      ⟨ Print generator matrix G 93 ⟩
    for (int $k = 0$; $k < 256$; $++k$) {
     ⟨ main compute codeword directly using vm4 177 ⟩
     ⟨ main compute codeword explicitly as vector-matrix product 181 ⟩
     ⟨ main compute codeword using Walsh − Hadamard transform and dyadic convolution 178 ⟩
     ⟨ main check equality of codewords 182 ⟩
    }
    }
```

This code is used in section 175.

184 ⟨ *main* generate next error vector 184 ⟩ ≡

```
    $next\_error\_vector(e\_old, e\_new, src, rndm, perm, n)$;
```

This code is used in section 185.

```
185 ⟨ main decode forged codewords 185 ⟩ ≡
    for (int $k = 0$; $k < t$; $++k$)
        $e\_old[k] = 1$;
    if ($t \equiv 4 \wedge n \equiv 32 \wedge m \equiv 6$) {
    ⟨ main generate next error vector 184 ⟩
    for (int $k = 0$; $k < 256$; $++k$) {
    ⟨ main compute codeword explicitly as vector-matrix product 181 ⟩
    ⟨ main print error positions 186 ⟩
    ⟨ main add some errors 187 ⟩
    ⟨ main compute the syndrome polynomial 117 ⟩
    ⟨ main solve the key equation $\omega(X) = \sigma(X)S(X) \mod g(X)$ 119 ⟩
    ⟨ main correct errors 121 ⟩
    ⟨ main check for correct decoding 188 ⟩
    ⟨ main free polynomials 189 ⟩}
    }
```
This code is used in section 175.

```
186 ⟨ main print error positions 186 ⟩ ≡
    if ($debug \geq 3$) {
        for (int $k = 0$; $k < n$; $++k$) {
            if ($e\_new[k]$) printf("%d␣", $k$);
        }
        printf("\n");
    }
```
This code is used in section 185.

```
187 ⟨ main add some errors 187 ⟩ ≡
    for (int $ix = 0$; $ix < n$; $++ix$)
        $cw[ix] = (cw[ix] + e\_new[ix])$ & 1;
```
This code is used in section 185.

```
188 ⟨ main check for correct decoding 188 ⟩ ≡
    for (int $i = 0$; $i < n - m * t$; $++i$)
        if ($msg[i] \neq cw[i]$)
            fprintf(stderr, "ERROR␣decoded␣wrong␣message␣msg[%d]=%d,␣cw[%d]=%d\n", $i$,
                    $msg[i], i, cw[i]$);
```
This code is used in section 185.

```
189 ⟨ main free polynomials 189 ⟩ ≡
    poly_free(SyM);
    poly_free(poly_sigma);
    poly_free(poly_omega);
```
This code is used in section 185.

190 ⟨ *main* free resources 190 ⟩ ≡
```
    if (e_old ≠ Λ) free(e_old);
    if (e_new ≠ Λ) free(e_new);
    if (src ≠ Λ) free(src);
    if (rndm ≠ Λ) free(rndm);
    if (perm ≠ Λ) free(perm);
    if (cw ≠ Λ) free(cw);
    if (dyadic_cw ≠ Λ) free(dyadic_cw);
    if (direct_cw ≠ Λ) free(direct_cw);
    if (msg ≠ Λ) free(msg);
    if (Hbin ≠ Λ) mat_free(Hbin);
    if (Gbin ≠ Λ) mat_free(Gbin);
    if (z ≠ Λ) free(z);
    if (L ≠ Λ) free(L);
    if (b ≠ Λ) free(b);
    if (h ≠ Λ) free(h);
    if (H ≠ Λ) free(H);
    if (H2T ≠ Λ) free(H2T);
    gf_free();
```
This code is used in section 170.

12 Known issues and further improvements

In order to hide the dyadic code structure, there are some measures taken in [23]. To be compliant, the Goppa code generated by *binary_quasi_dyadic_goppa_code* should have code length N, where $N \gg n$. To arrive at a code with length n, blocks would have to be selected, rearranged and permuted using dyadic permutations. However, we skip this step and rely completely on the secret and permuted support $\boldsymbol{L}$. As already mentioned, one of the reasons for this practice is simplicity. Another factor was the structural attack against the scheme (see part (VI)). The attack was successful in non-binary cases, so it is not clear if those code hiding techniques could be more effective in the binary case.

Furthermore, the algorithm implemented in *binary_quasi_dyadic_goppa_code* spuriously yields an erroneous support $\boldsymbol{L}$, in which case the code generation is restarted. It is at the moment not clear if this is a bug in the code or a design error of the algorithm itself.

Part VIII
Appendix

13 Basic algebraic structures

Coding theory is concerned with the transmission of *messages* over an unreliable channel. Transmission errors can occur and the question arises how to possibly recognize or even correct them.

The messages to be sent are seen as fixed-length sequences of symbols over a fixed alphabet. As we will work with so-called *linear (block) codes*, the messages will be seen as elements of a k-dimensional subvector space over the finite field $\mathbb{F}_q^n$.[27] Thus, each *encoded* message will consist of a fixed number of symbols of $\mathbb{F}_q^n$.

The algebraic structure of linear codes is utilized for efficient *encoding* and *decoding* techniques. Algebra is therefore the most important tool when dealing with linear codes.

For convenience, we will recall some definitions and facts in the Appendix, which will be used below. Details can be found for example in [27, 28, 34, 33].

13.1 Monoid, group and field

Definition 13.1.1 (Semigroup, monoid, group)**.**

(i) *A set* $(\mathrm{M}, *)$ *together with an associative operation* $*$ *is called a* semigroup.

(ii) *If the operation does have a neutral element* e*, then* $(\mathrm{M}, *)$ *is called a* monoid.

(iii) *A subset* $M' \subseteq M$*,* $e \in M'$*, which is closed under the monoid operation is called* submonoid *of* M*.*

(iv) *A* group[28] *is a monoid in* $(\mathrm{M}, *)$ *in which every element* $a \in M$ *is* invertible*, i.e. for all* $a \in M$ *there exists an* $a' \in M$ *such that* $a * a' = a' * a = e$*. Additionally, it holds for all* $a \in M$ *that* $a * e = e * a = a$*.*

(v) *A group* $\mathrm{M}, *$ *such that for all* $a, a' \in M$ *it holds that* $a * a' = a' * a$ *is called a* commutative *or* abelian *group. It is usually denoted by* $(\mathrm{M}, +)$*.*

Definition 13.1.2 (Ring)**.** *Let* $(A, +, \cdot)$ *be a set together with two operations called* addition *and* multiplication*.* A *is called a ring with respect to these operations if the following conditions hold:*

(i) $(A, +)$ *is an abelian group, called the* additive group *of* A*.*

[27]Frequently the scalar ring denoted also as $GF(q)$, the *Galois field* with q elements, named after *Evariste Galois*.
[28]The name *group* goes back again to Galois [28].

(ii) $(A, \cdot)$ *is a monoid, called the* multiplicative monoid *of* A.[29]

(iii) *For all* $a, b, c \in A$ *hold*

1. *a(b + c) = ab + ac*
2. *(b + c)a = ba + ca*

(iv) *The ring A is called* commutative *if* $(A, \cdot)$ *is abelian.*

Definition 13.1.3 (Subring, ring extension)**.** *Let* A *be a ring. A subset* $U \subseteq A$ *is called a* subring *of* A *if* $(U, +, \cdot)$ *is itself a ring. We write* $S \leq A$ *to express that* U *is a subring of* A *and also call* $[A\colon U]$ *a* ring extension. *If* $\mathbf{1}_A \in U$ *then we call* $[A\colon U]$ *a* unital ring extension.

Definition 13.1.4 (Characteristic)**.** *Let* A *be a ring. The* characteristic char A *of* A *is the smallest number* $n \in \mathbb{N}$ *such that* $n \cdot \mathbf{1} = 0$. *If no such* n *exists, we say that* char $A = 0$.

Definition 13.1.5 (Units, unital group)**.** *Let A be a ring.*
Invertible elements $a \in (A, \cdot)$ *are called* units. *They form the* unital group *of A, denoted by* $A^{\times}$.

Definition 13.1.6 (Zero-divisor)**.** *Let* A *be a ring.*
An element $a \neq 0$ *in* A *is called a* zero-divisor *if there is an element* $s \neq 0$ *such that* $rs = 0$ *or* $sr = 0$.

Definition 13.1.7 (Integral domain, field, subfield)**.** *Let* A *be a commutative ring.*

(i) *A is an* integral domain *if it has no zero-divisors.*

(ii) *An integral domain is called a* field *if* $a|b$ *for any two elements* $a \in A \setminus \{0\}$ *and* $b \in A$.

(iii) *Is a subring* U *of a ring* A *a field, we say that* U *is a* subfield *of* A.

Remark 13.1.8. *We will deal only with so-called* finite fields, *i.e. fields with a finite number of* q *elements (*$q \in \mathbb{N}$*). It will be denoted by* $\mathbb{F}_q$. *The number of elements of* $\mathbb{F}_q$ *is called its* order. *As can be shown [27], the characteristic of a finite field is always a prime number* p.

13.2 Direct product and direct sum

Definition 13.2.1 (Direct product)**.** *Let* $I \neq \emptyset$ *a non-empty index set and let* M_i, $i \in I$ *a system of sets with* $M_i \neq \emptyset$ *for all* $i \in I$. *Let* $M := \bigcup_{i \in I} M_i$. *The set of families* $\{f : I \to M \mid f_i := f(i) \in M_i$ *for all* $i \in I\ \}$ *is called the* direct *or* cartesian product *of the* M_i.[30] *It is denoted by* $\prod_{i \in I} M_i$.

If the M_i *are monoids for all* $i \in I$, *then* M *is a monoid relative the following operation:* $M \times M \to M$, $((a_i)_{i \in I}, (b_i)_{i \in I}) \mapsto ((a_i b_i)_{i \in I})$. *It is called the* product monoid *of the* M_i.

If all $M_i = M$ *for a set* M, *then* $\prod_{i \in I} M_i = M^I = \{f \mid f : I \to M\}$, *the set of all mappings from* I *to* M. *It is called the* I-fold product *of* M.

[29]Hence rings as defined here will always have a multiplicative neutral element e, denoted as $\mathbf{1}$.
[30]A common notation for such a family f is $(f_i)_{i \in I}$

Definition 13.2.2 (Direct sum). *Let I an arbitrary set, M_i monoids for all $i \in I$, and let e_i denote the neutral element of M_i for all $i \in I$. $\prod'_{i\in I} M_i := \{(a_i)_{i\in I} \in \prod_{i\in I} M_i \mid a_i \neq e_i$ for finitely many $i \in I\}$ is a submodule of $\prod_{i\in I} M_i$. Is is called the* direct sum *of the monoids M_i and denoted by $\bigoplus_{i\in I} M_i$ or $\coprod_{i\in I} M_i$.*[31]

Remark 13.2.3. *Let M a monoid, I an arbitrary set. Then $\prod_{i\in I} M$ is denoted by $M^{(I)}$ and called the* I-fold direct product *of the monoids M. It is a submonoid of M^I, the direct product of the monoids M. In case $I = \{1,\ldots,n\}$, then $M^I = M^{(I)} = M^n$.*

13.3 Module and vector space

Definition 13.3.1 (Module, vector space). *Let A be a ring. An abelian group V together with a (mulipicatively denoted) operation*[32] *of A on V is called an A-*module *if for all $a, b \in A$ and $x, y \in V$ hold:*

(i) $\mathbf{1}_A \cdot x = x$.

(ii) $a(bx) = (ab)x$.

(iii) $a(x+y) = ax + ay$.

(iv) $(a+b)x = ax + bx$.

If the underlying scalar *ring A is a field, then V is called a* vector space.

Remark 13.3.2 (Direct product and direct sum of modules). *Let $(V_i)_{i\in I}$ a family of A-modules. The direct product $\prod_{i\in I} V_i$ and the direct sum $\bigoplus_{i\in I} V_i = \coprod_{i\in I} V_i$ are A-modules by $a(x_i)_{i\in I} := (ax_i)_{i\in I}$ for $a \in A$ and $(x_i)_{i\in I}$. The addition in $\prod_{i\in I} V_i$ resp. $\bigoplus_{i\in I} V_i$ is defined as $(x_i)_{i\in I} + (y_i)_{i\in I} := (x_i + y_i)_{i\in I}$. If all $V_i = V$ for an A-module V, then $\prod_{i\in I} = V^I$ and $\bigoplus_{i\in I} V_i = V^{(I)}$. If $I = \{1,\ldots,n\}$, then $V^I = V^{(I)} = V^n$.*

Remark 13.3.3 (n-dimensional vector space). *Let $\mathbb{F}$ a field, $n \in \mathbb{N}$. $\mathbb{F}^n$ is then canonically an $\mathbb{F}$-module, i.e. an n-dimensional $\mathbb{F}$-vector space. For $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$ and $\alpha \in \mathbb{F}$ we have*

$$\begin{array}{lcccl} \boldsymbol{x} + \boldsymbol{y} & := & (x_0,\ldots,x_{n-1}) + (y_0,\ldots,y_{n-1}) & := & (x_0+y_0,\ldots,x_{n-1}+y_{n-1}) \\ \alpha\boldsymbol{x} & := & \alpha(x_0,\ldots,x_{n-1}) & := & (\alpha x_0,\ldots,\alpha x_{n-1}) \end{array}$$

[31] If I is finite, then clearly $\prod_{i\in I} M_i = \coprod_{i\in I} M_i$.

[32] An *operation* of a set M on a set X is a map $M \times X \to X$.

13.4 Polynomials

Let A a ring and M a monoide with neutral element ι. The *monoid ring* $A[M]$ is defined as follows: the underlying set is the M-fold direct sum $A^{(M)}$ of A. For each $\sigma \in M$ denote by $\boldsymbol{e}_\sigma$ the canonical basis element $(\delta_{\sigma,\tau})_{\tau\in M}$ of the A-module $A^{(M)}$, where δ denotes the Kronecker symbol. If $\sigma, \tau \in M$, $a, b \in A$, then define

$$(a\boldsymbol{e}_\sigma)(b\boldsymbol{e}_\tau) := ab\boldsymbol{e}_{\sigma\tau}$$

and distributively extend this to get a multiplication on $A[M]$. Hence, a ring structure is introduced on $A[M]$ with $\boldsymbol{e}_\iota$ as $\mathbf{1}$.

Indeed, let $\boldsymbol{a}, \boldsymbol{b} \in A^{(M)}$ with $\boldsymbol{a} = \sum_{\sigma\in M} a_\sigma \boldsymbol{e}_\sigma$ and $\boldsymbol{b} = \sum_{\sigma\in M} b_\sigma \boldsymbol{e}_\sigma = \sum_{\tau\in M} b_\tau \boldsymbol{e}_\tau$. Clearly, $\boldsymbol{a} + \boldsymbol{b} = \sum_{\sigma\in M}(a_\sigma + b_\sigma)\boldsymbol{e}_\sigma$, whereas for the multiplication in $A[M]$ we have $\boldsymbol{ab} = (\sum_{\sigma\in M} b_\sigma \boldsymbol{e}_\sigma)(\sum_{\tau\in M} b_\tau \boldsymbol{e}_\tau) = \sum_{(\sigma,\tau)\in M\times M} a_\sigma b_\tau \boldsymbol{e}_{\sigma\tau}$. Finally, $\boldsymbol{a}\boldsymbol{e}_\iota = (\sum_{\sigma\in M} a_\sigma \boldsymbol{e}_\sigma)\boldsymbol{e}_\iota = \sum_{\iota\in M} a_\iota \boldsymbol{e}_\iota = \boldsymbol{a}$, as desired.

$M \hookrightarrow A[M] \colon \sigma \mapsto \boldsymbol{e}_\sigma$ and $A \hookrightarrow A[M] \colon a \mapsto a\boldsymbol{e}_\iota$ are injective monoid homomophisms and ring homomorphisms, respectively. Therefore, we identify M with a submonoid of $(A[M], \cdot)$ and A with a subring of $A[M]$. Note that the elements of A commute with the elements of the standard basis $\boldsymbol{e}_\sigma$ of $A[M]$: $a\boldsymbol{e}_\sigma = (a\boldsymbol{e}_\iota)(\mathbf{1}\boldsymbol{e}_\sigma) = (a \cdot \mathbf{1})\boldsymbol{e}_\iota\boldsymbol{e}_\sigma = (\mathbf{1} \cdot a)\boldsymbol{e}_\sigma\boldsymbol{e}_\iota = (\mathbf{1}\boldsymbol{e}_\sigma)(a\boldsymbol{e}_\iota) = \boldsymbol{e}_\sigma a$.

Let now A be arbitrary ring $(A \neq 0)$, I a set and $\mathbb{N}^{(I)}$ the I-fold direct sum of the additive monoide $(\mathbb{N}, +)$. Let $\epsilon_i \in \mathbb{N}^{(I)}$ the I-tupel, whose i-th component is 1, and whose other components are 0. Hence, a $\nu \in \mathbb{N}^{(I)}$ can be written as $\nu = \sum_{i\in I} \nu_i \epsilon_i$.

In $A[\mathbb{N}^{(I)}]$ we define

$$X_i := \boldsymbol{e}_{\epsilon_i}$$

and conclude

$$\boldsymbol{e}_\nu = \boldsymbol{e}_{\sum_{i\in I}\nu_i\epsilon_i} = \prod_{i\in I} \boldsymbol{e}_{\nu_i\epsilon_i} = \prod_{i\in I} \boldsymbol{e}_{\sum_{\nu_i}\epsilon_i} = \prod_{i\in I} X_i^{\nu_i}.$$

The elements $\boldsymbol{e}_\nu$ of the standard basis are therefore powers of the elements X_i, $i \in I$. Note again that X_i commutes with X_j $(i \neq j)$ and $a \in A$:

$$X_i X_j = \boldsymbol{e}_{\epsilon_i}\boldsymbol{e}_{\epsilon_j} = \boldsymbol{e}_{\epsilon_i+\epsilon_j} = \boldsymbol{e}_{\epsilon_j+\epsilon_i} = \boldsymbol{e}_{\epsilon_j}\boldsymbol{e}_{\epsilon_i} = X_j X_i$$

$$aX_i = a\boldsymbol{e}_{\epsilon_i} = (a\boldsymbol{e}_0)(\mathbf{1}\boldsymbol{e}_{\epsilon_i}) = (a\mathbf{1})(\boldsymbol{e}_{\epsilon_i}) = (\mathbf{1}a)(\boldsymbol{e}_{\epsilon_i}) = (\mathbf{1}\boldsymbol{e}_{\epsilon_i})(a\boldsymbol{e}_0) = \boldsymbol{e}_{\epsilon_i} a = X_i a$$

The X_i are called *indeterminates* over A.

Definition 13.4.1 (Polynomial ring, polynomials)**.** *The monoid ring $A[\mathbb{N}^{(I)}]$ is called the* polynomial ring *in the* indeterminates *X_i ($i \in I$) over A. It is denoted by $A[X_i \colon i \in I]$ or $A[X_i]_{i\in I}$. Its elements are called* polynomials *in the indeterminates X_i over A.*

Remark 13.4.2. *It is common to write $\boldsymbol{e}_\nu = X^\nu$ for the basis elements $\boldsymbol{e}_\nu = \prod_{i\in I} X_i^{\nu_i}$. Each polynomial f can be written then in the form*

$$f = \sum_{\nu\in\mathbb{N}^{(I)}} a_\nu X^\nu$$

with uniquely determined $a_\nu \in A$ and $a_\nu \neq 0$ for only finitely many ν. A polynomial is therefore a finite sum of elements of the form $a_\nu X^\nu$ which are called monomials. *Finally note that $\nu \in \mathbb{N}^{(I)}$ is a called a* multi-index.[33]

[33] ν runs through $\mathbb{N}^{(I)}$ rather than through $\mathbb{N}$.

If $g = \sum_{\nu \in \mathbb{N}^{(I)}} b_\nu X^\nu$ is another polynomial in $A[\mathbb{N}^{(I)}]$, we have

$$f + g = \sum_{\nu \in \mathbb{N}^{(I)}} (a_\nu + b_\nu) X^\nu$$
$$fg = \sum_{\lambda \in \mathbb{N}^{(I)}} c_\lambda X^\lambda$$

with $c_\lambda := \sum_{\nu+\mu} a_\nu b_\mu$, where λ, μ, ν run through $\mathbb{N}^{(I)}$.

Remark 13.4.3. *Let I a set with one element. In the above construction, one has then only one indeterminate, denoted just by X, and we have:*

$$f = \sum_{\nu \geq 0} a_\nu X^\nu = a_0 + a_1 X^1 + \cdots + a_n X^n,$$
$$g = \sum_{\mu \geq 0} b_\nu X^\nu = b_0 + b_1 X^1 + \cdots + b_m X^m,$$
$$fg = \sum_{\lambda \geq 0} c_\lambda X^\lambda \; with \; c_\lambda = \sum_{j=0}^{\lambda} a_j b_{\lambda - j} = a_0 b_\lambda + \cdots + a_\lambda b_0,$$

if $a_\nu = b_\mu = 0$ for $\nu > n, \mu > m$.

References

[1] M. Braun et. al. A. Betten. *Error-Correcting Linear Codes*, volume 18 of *Algorithms and Computation in Mathematics*. Springer, 2006.

[2] P. S. L. M. Barreto. Post Quantum Cryptography. `http://www.larc.usp.br/~pbarreto/PQC-4.pd`.

[3] P. S. L. M. Barreto. The Fast Walsh-Hadamard Transform. Private communication, 2010.

[4] P. S. L. M. Barreto, P.-L. Cayrel, R. Misoczki, and R. Niebuhr. Quasi-dyadic CFS signatures. In *Inscrypt 2010*, LNCS. Springer, 2010.

[5] P. S. L. M. Barreto, R. Lindner, and R. Misoczki. Decoding square-free Goppa codes over $\mathbb{F}_p$. 2010.

[6] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3):384–386, 1978.

[7] E. R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24(111):713–715, 1970.

[8] D. J. Bernstein. List decoding for binary Goppa codes, 2008. `http://cr.yp.to/codes/goppalist-20110223.pdf`.

[9] R. T. Chien. Cyclic decoding procedure for the bose-chaudhuri-hocquenghem codes. *IEEE Transactions on Information Theory*, 10(4):357–363, 1964.

[10] R. Durstenfeld. Algorithm 235: Random Permutation. *Communications of the ACM*, 7(7):420, 1964.

[11] D. Engelbert, R. Overbeck, and A. Schmidt. A summary of McEliece-type cryptosystems and their security. *Journal of Mathematical Cryptology*, 1:151–199, 2007.

[12] J.-C. Faugère, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic cryptanalysis of McEliece variants with compact keys. In *Advances in Cryptology – Eurocrypt'2010*, LNCS. Springer, 2010.

[13] R. A. Fisher and F. Yates. Statistical tables for biological, agricultural and medical research, 1948, (1938). `http://en.wikipedia.org/wiki/Fisher-Yates_shuffle`.

[14] B. Friedrichs. *Kanalcodierung, Grundlagen und Anwendungen in modernen Kommunikationssystemen*. Springer, 1996.

[15] A. Joux. *Algorithmic Cryptanalysis*. Cryptography and Network Security. Chapman and Hall/CRC, 2009.

[16] D. E. Knuth. *The Art of Computer Programming, 2. Seminumerical Algorithms*. Addison-Wesley, 2005.

[17] D. E. Knuth and S. Levy. *The CWEB System of Structured Documentation*. Addison-Wesley Professional, 1994. `http://www-cs-faculty.stanford.edu/~uno/cweb.html`.

[18] E. Kunz. *Algebra*. Vieweg, 1991.

[19] R. Lidl and H. Niederreiter. *Finite Fields*. Number 20 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, UK, 2nd edition, 1997.

[20] F. J. MacWilliams and N. J. A. Sloane. *The theory of error-correcting codes*, volume 16. North-Holland Mathematical Library, 1977.

[21] R. McEliece. A public-key cryptosystem based on algebraic coding theory. The Deep Space Network Progress Report, DSN PR 42–44, 1978. `http://ipnpr.jpl.nasa.gov/progressreport2/42-44/44N.PDF`.

[22] R. J. McEliece. *The Theory of Information and Coding*. Encyclopedia of Mathematics and its Applications. Addison-Wesley, 1978.

[23] R. Misoczki and P. S. L. M. Barreto. Compact McEliece keys from Goppa codes. In *Selected Areas in Cryptography – SAC'2009*, volume 5867 of *LNCS*, pages 276–392. Springer, 2009.

[24] N. J. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, 1975.

[25] E. Persichetti. Compact McEliece keys based on Quasi-Dyadic Srivastava codes, 2011. `http://eprint.iacr.org/2011/179.pdf`.

[26] S. Roman. *Coding and Information Theory, Graduate Texts in Mathematics*. Springer, 1992.

[27] G. Scheja and U. Storch. *Lehrbuch der Algebra*, volume 1. Teubner, 1980.

[28] G. Scheja and U. Storch. *Lehrbuch der Algebra*, volume 2. Teubner, 1988.

[29] N. Sendrier and B. Biswas. Hymes, The Hybrid McEliece Encription Scheme. `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`.

[30] N. Sendrier and B. Biswas. McEliece Cryptosystem Implementation: Theory and Practice. In *Post-Quantum Cryptography*, volume 5299 of *LNCS*, 2008.

[31] C. E. Shannon. A Mathematical Theory of Communication and Communication, Theory of Secrecy Systems. *Bell System Tech. J.*, 1948.

[32] P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM Journal on Computing*, 26:1484–1509, 1995.

[33] K. Spindler. *Abstract Algebra with Applications: Rings and Fields*. Dekker, 1994.

[34] K. Spindler. *Abstract Algebra with Applications: Vector Spaces and Groups*. Dekker, 1994.

[35] H. Stichtenoth. *Algebraic Function Fields and Codes*. Graduate Texts in Mathematics. Springer, Berlin, Heidelberg, 2nd edition, 2009.

[36] K. K. Tzeng and K. Zimmermann. On extending Goppa codes to cyclic codes. *IEEE Transactions on Information Theory*, 21:721–716, 1975.

[37] H. S. Warren. *Hacker's Delight*. Addison-Wesley, 2003.

Index

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. Error messages are also shown.

List of Refinements