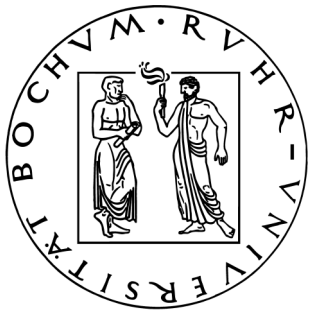

Ruhr-University Bochum

Post Quantum Cryptography on Embedded Devices: An Efficient Implementation of the McEliece Public Key Scheme based on Quasi-Dyadic Goppa Codes

Olga Paustjan

July 1, 2010

Diploma Thesis
Ruhr-University Bochum

Chair for Embedded Security
Prof. Dr.-Ing. Christof Paar
Dipl.-Ing. Stefan Heyse

Gesetzt am July 1, 2010 um 14:59 Uhr.

Eidesstaatliche Erklärung / Statement

Hiermit versichere ich, dass ich meine Diplomarbeit selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

I hereby certify 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 reference to the other author. This theses has not been presented to any other examination board or published before.

Bochum, den July 1, 2010

_____

Olga Paustjan

Contents

1 Introduction

1.1 Motivation

In the last years, the need for embedded systems has arisen continuously. Spanning all aspects of modern life, they are included in almost every electronic device: mobile phones, personal digital assistants (PDAs), domestic appliances, and even in cars. This ubiquity goes hand in hand with increased need for embedded security. For instance, it is crucial to protect a car's electronic door lock from unauthorized use. These security demands can be solved by cryptography. In this context, many symmetric and asymmetric algorithms, such as AES, DES, RSA, ElGamal, and ECC, are implemented on embedded devices.

Most public-key cryptosystems frequently implemented have been proven secure on the basis of the presumed hardness of two mathematical problems: factoring the product of two large primes (FP) and computing discrete logarithms (DLP). Both problems are well known to be closely related. Hence, solving these problems would have significantly ramifications for classical public-key cryptography, and thus, for embedded devices the algorithms are implemented on. Nowadays, both problems are believed to be computationally infeasible with an ordinary computer. However, a quantum-computer having the ability to perform computations on a few thousand qbits could solve both problems by using Shor's algorithm [Sho97]. Although a quantum computer of this dimension has not been reported, development and cryptanalysis of alternative public-key cryptosystems seem suitable. Cryptosystems not breakable using quantum computers are called post-quantum cryptosystems.

Most published post-quantum public-key schemes are focused on the following approaches [BBD08]: Hash-based cryptography (e.g. Merkle's hash-tree public-key signature system [Mer79]), Multivariate-quadratic-equations cryptography (e.g. HFE signature scheme [Pat96]), Lattice-based cryptography (e.g. NTRU encryption scheme [HPS98]), and Code-based cryptography (e.g. McEliece encryption scheme [McE78], Niedereiter encryption scheme [Nie86]).

In this thesis, we concentrate on Code-based cryptography. The first code-based public-key cryptosystem was proposed by Robert McEliece in 1978. The McEliece cryptosystem is based on algebraic error-correcting codes, namely Goppa codes. The hardness assumption of the McEliece cryptosystem is that decoding of Goppa

codes is easily performed by an efficient decoding algorithm, but when disguising a Goppa code as a general linear code by means of several secret transformations, decoding becomes NP-complete. The problem of decoding linear error-correction codes is neither related to the factorization nor to the discrete logarithm problem. Hence, the McEliece scheme is an interesting candidate for post-quantum cryptography, as it is not effected by the computational power of quantum computers.

To achieve acceptance and attention in practice, post-quantum public-key schemes have to be implemented efficiently. Furthermore, the implementations have to perform fast while keeping memory requirements small for security levels comparable to conventional schemes. The McEliece encryption and decryption do not require computationally expensive multiple precision arithmetic. Hence, it is predestined for an implementation on embedded devices. Indeed, there exist efficient implementations of this public-key cryptosystems on a microcontroller and FPGA [EGHP09].

The chief disadvantage of the McEliece public-key cryptosystem is its very large public key of several hundred thousands of bits. For this reason, the McEliece PKC has achieved little attention in the practice, yet. Particularly, with regard to bounded memory capabilities of embedded systems, it is essential to improve the McEliece cryptosystem by finding a way to reduce the public key size. An ongoing research is to replace Goppa codes by other codes having a compact and simple description. For instance, there are proposals based on quasi-cyclic codes [Gab05] and quasi-cyclic low density parity-check codes [BC07]. Unfortunately, all these proposals have been broken by structural attacks [OTD08]. Barreto and Misoczki propose in a recent work [MB09] using Goppa codes in quasi-dyadic form. When constructing a McEliece-type cryptosystem based on quasi-dyadic Goppa codes the public key size is significantly reduced. For instance, for a 80-bit security level the public key used in the original McEliece scheme is 437.75 Kbytes in size. The public key size of the quasi-dyadic variant is 2.5 Kbytes which is a factor 175 smaller compared to the original McEliece PKC. For this reason, it is interesting how the quasi-dyadic McEliece variant performs on embedded devices.

Another disadvantage of the McEliece scheme is that it is not semantical secure. The quasi-dyadic McEliece variant proposed by Barreto and Misoczki is based on systematic coding. It allows to construct CPA and CCA2 secure McEliece variants by using additional conversion schemes, such as Kobara-Imai's specific conversion γ [NIKM08].

In this thesis we provide an implementation of this alternative public-key cryptosystem. In addition, we apply the Kobara-Imai's specific conversion γ on the quasi-dyadic McEliece variant to achieve semantical security. The KIC-γ has also been implemented within the scope of this thesis.

1.2 Existing Implementations

Only few implementations of the original McEliece public-key cryptosystem have been reported. For instance, there exist two software implementations for 32-bit architectures: an i386 assembler implementation [PBGV92] and a C-implementation [Pro09]. Recently, the McEliece PKC achieves more and more attention by researchers analyzing the security of McEliece variants. Hence, we assume that there exist other implementations for frequently used CPUs, but non of them has been published. Two excellent implementations of the original McEliece PKC on an 8-bits AVR microcontroller and an FPGA have been provided by the Chair for Embedded Security at the Ruhr-University Bochum. The microcontroller implementation encrypts with 3,889 bits/second and decrypts with 2,835 bits/second at a 32 MHz clock frequency. The main disadvantage of this implementation is the use of external memory for encryption. As explained above, the public-key of the original McEliece PKC is 437.75 Kbytes in size such that external memory has to be used to store the key. The quasi-dyadic variant should solve the problem of large public keys, increasing the practicability of the McEliece public-key cryptosystem. To the best of our knowledge, no implementations of the quasi-dyadic McEliece variant have been proposed targeting an embedded device.

1.3 Goals of this Study

The aim of this thesis is a proof-of-concept implementation of a McEliece-type cryptosystem based on quasi-dyadic Goppa codes on an 8-bits AVR microcontroller. Particularly, an interesting task is to overperform the implementation of the original McEliece variant in both, encryption and encryption.

1.4 Outline

The remainder of this thesis is organized as follows. Chapter 2 introduces the classical McEliece public key scheme and motivates the need of public key reduction. Chapter 3 introduces some basic concepts of coding theory. In further progress of this chapter we describe how binary dyadic and quasi-dyadic Goppa codes are constructed. Chapter 4 gives the scheme definition of the quasi-dyadic McEliece variant consisting of three basic algorithms: key generation, encryption and decryption. Chapter 5 describes the Kobara-Imai's specific conversion γ also implemented within the scope of this thesis. In Chapter 6, our implementation of the McEliece PKC with quasi-dyadic Goppa codes on an 8-bits AVR microcontroller is explained. We provide the results of our implementation, with respect to memory requirements and performance, in Chapter 8 and conclude in Chapter 9.

2 The McEliece cryptosystem

The McEliece cryptosystem [McE78] was developed by Robert McEliece in 1978 and was the first proposed public-key cryptosystem (PKC) based on error-correcting codes.

The idea behind this scheme is to pick randomly a code from a family of codes with an existing efficient decoding algorithm and to use the description of this code as a private key. To obtain the public key the private key is disguised as a general linear code by means of several secret transformations. The decoding of general linear codes is known to be $\mathcal{NP}$-hard. Hence, the purpose of these transformations is to hide any visible structure of the private key which might be used to identify the underlying code.

McEliece's approach was to use binary Goppa codes as private key. These codes are known to be easy to decode using e.g. Patterson's decoding algorithm [Pat75] or Bernstein's list decoding method [Ber08]. The public key is obtained from the private key by two linear transformations: a scrambling transformation and a permutation transformation. The hardness of the McEliece scheme is the indistinguishability of modified Goppa codes from general linear codes for which decoding is $\mathcal{NP}$-hard.

The McEliece PKC has some advantages compared to classical public-key schemes such as RSA and ECC. In general, the encryption and decryption can be done faster. In addition, with growing keys the security level increases faster. Another major advantage of the McEliece PKC is the resistance to attacks performed by quantum computers.

The remainder of this chapter is organized as follows. Section 2.1 gives an overview about the original McEliece's scheme and presents the key generation, encryption and decryption algorithms. Furthermore, this section provides the correctness proof of the McEliece scheme. Section 2.2 presents recommended parameters and the resulting key sizes while Section 2.3 motivates the need to reduce the size of the public key. In the last Section 2.4 the security of the original McEliece PKC is discussed.

2.1 Scheme definition

The McEliece public-key cryptosystem consists of three algorithms presented in this section:

- Probabilistic polynomial-time key generation algorithm generating public and private parameters
- Probabilistic polynomial-time encryption algorithm that takes as input the recipient's public key K_{pub} and a message M and outputs a ciphertext c
- Deterministic polynomial-time decryption algorithm that takes as input the private key K_{pr} and a ciphertext c and outputs the message M. The decryption must undo the encryption, i.e., $dec(K_{pr}, enc(K_{pub}, m)) = m$ must hold for any generated key pair (K_{pr}, K_{pub}) and any message m (see Section 2.1.4).

2.1.1 Key generation

The common system parameters for the McEliece PKC are parameters of the underlying $[n, k, d]$ binary Goppa code defined by an (irreducible) polynomial of degree t over $GF(2^m)$ called Goppa polynomial. Corresponding to each such polynomial there exist a binary Goppa code of length $n = 2^m$, dimension $k \geq n - mt$ and minimum distance $d = 2t + 1$ where t is the number of errors correctable by an efficient decoding algorithm (see Chapter 3 for more information).

At first, the key generation algorithm (Algorithm 2.1.1) chooses a random binary [n,k,d]-Goppa code $\mathcal{C}$ defined by a Goppa polynomial of degree t. The generator matrix for the code $\mathcal{C}$ can serve as private key. In the next step the key generation algorithm selects a random scrambling matrix S which transforms G and sends it to another matrix $G' = S \cdot G$. As S is an invertible $k \times k$ matrix, the matrix G' is still a generator matrix for the same code C. Furthermore, the key generation algorithm selects randomly an $n \times n$ permutation matrix P which reorders the columns of G' to obtain the matrix $\hat{G} = S \cdot G \cdot P$. As P is a permutation, the resulting matrix $\hat{G}$ is the generator matrix for an equivalent linear code $\hat{\mathcal{C}}$. This code has the same rate and minimum distance as $\mathcal{C}$ but no existing efficient decoding algorithm. The problem of code equivalence can be reduced to the graph isomorphism problem which is supposed to be in $\mathcal{P}/\mathcal{NP}$ [PR97]. Hence, $\hat{G}$ can serve as public key.

Algorithm 2.1.1 Original McEliece PKC: Key generation algorithm

Input: Fixed common system parameters: t, n, k
Output: private key K_{pr}, public key K_{pub}

1. Choose a binary [n,k,d]-Goppa code $\mathcal{C}$ capable of correcting up to t errors
2. $G \leftarrow k \times n$ generator matrix for $\mathcal{C}$
3. Select a random non-singular binary $k \times k$ scrambling matrix S
4. Select a random $n \times n$ permutation matrix P
5. Compute the $k \times n$ matrix $\hat{G} = S \cdot G \cdot P$
6. **return** $K_{pr} = (S, G, P)$, $K_{pub} = (\hat{G}, t)$

The running time (in binary operations) to generate a key pair for the McEliece PKC is $\mathcal{O}(k^2n + n^2 + t^3(n-k) + (n-k)^3)$ [EOS07].

2.1.2 Encryption

The McEliece encryption is done by multiplying a k-bits message vector by the recipient's public generator matrix $\hat{G}$ and adding a random error vector with Hamming weight at most t.

Algorithm 2.1.2 Original McEliece PKC: Encryption algorithm

Input: Message M, recipient's public key $K_{pub} = (\hat{G}, t)$
Output: Ciphertext c

1. Represent M as a binary string m of length k
2. Choose a random error vector e of length n and hamming weight $wt(e) \leq t$
3. Compute the binary ciphertext vector $c = m \cdot \hat{G} + e$
4. **return** c

The time complexity of the McEliece encryption algorithm is $\mathcal{O}(k/2 \cdot n + t)$ [EOS07].

2.1.3 Decryption

The decoding problem is the problem of decoding a linear code $\hat{\mathcal{C}}$ equivalent to a binary Goppa code $\mathcal{C}$. The knowledge of the permutation P is necessary to solve this problem. After reversing the permutation transformation, the decoder for $\mathcal{C}$ can be used to decode the permuted ciphertext $\hat{c}$ to a message $\hat{m} = S \cdot m$. The original message m is then obtained from $\hat{m}$ by reversing the scrambling transformation in step 3 of the decryption algorithm.

Algorithm 2.1.3 Original McEliece PKC: Decryption algorithm

Input: Ciphertext c, private key $K_{pr} = (S, G, P)$
Output: Message M

1. Compute $\hat{c} = c \cdot P^{-1}$
2. Use the decoding algorithm for the code $\mathcal{C}$ to obtain the message vector $\hat{m} = m \cdot S$ from $\hat{c}$
3. Compute $m = \hat{m} \cdot S^{-1}$
4. Represent the binary string m as message M
5. **return** M

The decryption of a ciphertext of a McEliece instance generated by a $[n = 2^m, k, d]$ binary irreducible Goppa code requires $\mathcal{O}(ntm^2)$ binary operations [EOS07].

2.1.4 Correctness

Any cryptosystem should satisfy the correctness property. This means that the decryption works correctly for any honest recipient of the ciphertext c. In the following, we show the correctness proof for the McEliece PKC.

$$\begin{aligned} \hat{c} &= c \cdot P^{-1} \\ &= (m \cdot \hat{G} + e) \cdot P^{-1} \\ &= (m \cdot S \cdot G \cdot P + e) \cdot P^{-}1 \\ &= m \cdot S \cdot G \cdot P \cdot P^{-1} + e \cdot P^{-1} \\ &= m \cdot S \cdot G + e \cdot P^{-1} \end{aligned}$$

As P is a permutation matrix, so is P^{-1}. Hence, the Hamming weight of the permuted error vector $e \cdot P^{-1}$ is still at most t. An efficient decoding algorithm for $\mathcal{C}$ can be used to detect the permuted error, and thus, to obtain $\hat{m}$ from $\hat{c}$. Multiplying $\hat{m}$ by the inverse of S gives $\hat{m} \cdot S^{-1} = m \cdot S \cdot S^{-1} = m$ which yields the message vector. Hence, the McEliece scheme works correctly.

2.2 Recommended parameters and key sizes

The parameters influencing the security of the McEliece PKC are the code length n, the code dimension k, and the number of added errors t. In his original paper [McE78] McEliece suggests using $[n = 2^m, k = n - mt, d = 2t + 1] = [1024, 524, 101]$ Goppa codes over $GF(2^m)$ where $m = 10$ and $t = 50$. In a recent paper [BLP08] Bernstein, Lange and Peters present an improved attack on the McEliece scheme. This new attack reduces the number of operations needed to break the McEliece scheme with original parameters to about 2^{60} instead of

2^{80} being assumed before. To achieve 80-bit, 128-bit, 256-bit security level the authors suggest using [2048,1751,55], [2960,2288,113], and [6624,5129,231] binary Goppa codes, respectively. The new parameters also exploit Bernstein's list decoding algorithm [Ber08] capable of correcting about $n - \sqrt{n(n-2t-2)} \geq t+1$ errors in a binary [n,k,d]-Goppa code. This decoding method allows senders to introduce more errors into ciphertexts, leading to higher security with the same key size, or alternatively the same security with lower key size. Note that the so far most efficient decoding algorithm due to Patterson [Pat75] is capable of correcting t errors at most.

Table 2.1 summarizes all suggested parameters as well as the resulting key sizes for specific security levels.

Security Level	**[n,k,d]-Code**	**Added errors**	**Size of K_{pub} in Kbits**	**Size of $K_{pr} = (G(x), P, S)$ in Kbits**
hardly 80-bit	[1632,1269,67]	34	2022	(0.34,15.94,1573)
80-bit	[2048,1751,55]	27	3502	(0.30,22,2994)
128-bit	[2960,2288,113]	56	6614	(0.61,31.80,5112)
256-bit	[6624,5129,231]	117	33178	(1.38,77.63,25690)

Table 2.1: Recommended parameters and key sizes for the original McEliece PKC

There are several ways to store the secret key. It is advisable to store directly the inverses P^{-1} and S^{-1} of both transformation matrices to enhance the performance of decryption. Since the (inverse) permutation matrix is sparse, it can be stored in the form of a permutation sequence reducing memory requirements from $n \cdot n$ bits to $m \cdot n$ bits. In [CC95] Canteaut and Chabaud pointed out that the scrambling matrix has no cryptographic function in hiding the secret Goppa polynomial $G(x)$. Hence, in [Hey09] Heyse proposes generating the scrambling matrix on-the-fly using a CPRNG and a prestored seed. The only restriction is that the generated matrix has to be invertible. The seed length depends on the block size of the cipher used as CPRNG, e.g. 80 bits for a PRESENT implementation. The storage space occupied by a Goppa polynomial defining a private binary Goppa code $\mathcal{C}$ is $(t+1)m$ bits. Hence, a private key can be stored compactly.

The major disadvantage of the McEliece public-key cryptosystem is its very large public key of several hundred thousand bits. The complete public generator matrix $\hat{G}$ of an (n,k) linear code occupies $n \cdot k$ bits storage space. For this reason, the McEliece PKC has achieved little attention in the practice. Particularly with regard to bounded memory capabilities of embedded devices, it is essential to improve the McEliece cryptosystem by finding a way to reduce the public key size.

2.3 Reducing public key sizes

The first naive approach to reduce the size of the public key is the use of a public generator matrix in systematic (row-echelon) form. A systematic public generator matrix is of the form $\hat{G}_{sys} = \{I_k|Q\}$, where I_k is the $k \times k$ identity matrix. It provides the following advantages:

1. As I_k is an identity matrix, it has not to be stored explicitly. The non-trivial part Q of $\hat{G}_{sys}$ is only $k(n-k)$ bits in size. Hence, the storage space occupied by the public key is reduced. For instance, the representation of a [2048,1751,55] Goppa code occupies about 508 Kbits. That is a factor of 6.89 less than in general

2. The encryption can be performed more efficiently because only the multiplication by a $k \times (n-k)$ submatrix Q of $\hat{G}_{sys}$ is necessary. Since the product of a vector and an identity matrix is the vector itself this multiplication can be omitted.

The problem with a public generator matrix in systematic form is that the multiplication $m \cdot \hat{G}_{sys}$ results in a codeword of the form $cw = (m||pcb)$ where the first part of cw is the message itself and the second part are parity-check bits. This is due to the special form of $\hat{G}_{sys}$ and is common case with systematic coding. As the error vector added during the last step of the encryption algorithm is sparse, the resulting ciphertext c would immediately reveal the message. Hence, additional steps have to be performed on the message before McEliece encryption. For this purpose the same conversions as for achieving CCA2-security can be used (see Chapter 5).

But only using a public generator matrix in systematic form is not enough for public key reduction. There is an ongoing research to replace classical irreducible Goppa codes with ones that can be represented in a more compact way while keeping the security level of the McEliece PKC. Increased efforts were made to find such alternative codes. Most of them tried to replace the class of Goppa codes by a family of other codes. For instance, Shokrollahi, Monico and Rosenthal examined in [SMR00] a possible solution of using low density parity-check codes (LDPC) and showed this solution to be unsafe. Other approaches are based on the idea of using quasi-cyclic codes ([Gab05] proposed by Gaborit) or quasi-cyclic low density parity-check codes ([BC07] proposed by Baldi and Chiaraluce). The first approach considers subcodes of primitive BCH codes and uses a very constrained permutation for hiding the secret code. The second approach uses quasi-cyclic LDP-codes and more general one-to-one mappings instead of permutation matrices. In [OTD08] Otmani, Tillich, Dallot showed that both systems can be broken totally by structural attacks exploiting the quasi-cyclic structure of the code.

In a recent work [MB09] Misoczki and Baretto describe another way to reduce the public key size in McEliece-type cryptosystems by using a subclass of Goppa codes, namely quasi-dyadic Goppa codes. In contrast to many other proposed code families, binary Goppa codes are cryptanalysis-resistant and remain still unbroken.

The aim of this thesis is a proof-of-concept implementation of a McEliece-type cryptosystem based on quasi-dyadic Goppa codes on an embedded microcontroller. In addition, a systematic public generator matrix is used. The construction of this cryptosystem is presented in Chapter 4.

2.4 Security of the McEliece PKC

The security of code-based cryptosystems depends on the difficulty of two basic kinds of attacks:

- **Structural Attack:** Given the public generator matrix $\hat{G}$ recover the secret transformation, also called trapdoor, and the description of the secret code or an equivalent one. The difficulty of this attack is not related to any known coding theoretic problem. It mainly depends on the class of private codes used as well as on the secret transformation.
- **Ciphertext-Only-Attack:** Given a ciphertext and the public key recover the corresponding message. This attack is related to the general decoding problem. The general decoding problem is the problem of decoding a received word to the closest codeword in an arbitrary code and is known to be NP-complete [BMvT78].

Hence, the main security issues in code-based cryptography are

- the use of secure private codes such that no attack exploiting the structure of the private code is possible
- the hiding of the structure of a private code in order to obtain a public code which is indistiguishable from a general linear code.

Using classical irreducible Goppa codes over $\mathbb{F}_2$ derived from codes over $\mathbb{F}_{2^m}$ no efficient polynomial-time algorithm is known which can distinguish these codes from general linear codes. The best known attack uses the support splitting algorithm (SSA) [Sen00] to solve the permutation equivalence problem, and thus, to recover the private key from the public key. This attack tests all possible Goppa polynomials of degree $\leq 2t$ and checks whether the corresponding Goppa code is permutation equivalent to the public code by calling the SSA. If the SSA was able to find a permutation, the attack was successful. The time complexity of this attack can be estimated as $\mathcal{O}(n^{2t}(1 + o(1)))$ [**?**], which is negligible for all suitable McEliece parameters.

Several attacks are known trying to recover a message from a ciphertext only, and thus, to solve the general decoding problem, e.g. the Generalized information-set-decoding attack proposed by McEliece in his original paper as well as Finding-low-weight-codeword attacks (Leon [Leo88], Stern [Ste89], Canteaut and Chabaud [CC98], Bernstein, Lange, Peters [BLP08]). All these attacks require exponential time, hence, no polynomial time algorithm is known which solves the general decoding problem.

Indeed, the McEliece PKC has weaknesses which reduce the complexity of the above ciphertext-only attacks. The first weakness appears when an adversary has partial knowledge of the encrypted message. Let I and J denote two sets of indexes such that $I \subset \{1, \ldots, k\}$ and $J = \{1, \ldots, k\} \backslash I$. Let an adversary $\mathcal{A}$ have knowledge of the message bits $m_{i \in I}$ such that $m\hat{G} = m_I G_I \oplus m_J G_J$. Then $\mathcal{A}$ may try to recover m_J from $c' = c \oplus m_I G_I = m_J G_J \oplus e'$ using one of the above attacks. The attack complexity is obviously reduced.

Another weakness comes out when an adversary $\mathcal{A}$ has knowledge of a relation $\mathcal{R}(m_1, m_2)$ of two messages m_1, m_2, e.g. $\mathcal{R}(m_1, m_2) = m_1 \oplus m_2$. Then the following holds

$$c' = c_1 \oplus c_2 \oplus \mathcal{R}(m_1, m_2) = e_1 \oplus e_2 = e' \text{ with } e'_i = 0 \text{ if } \begin{cases} e_{1,i} = e_{2,i} = 0 \\ e_{1,i} = e_{2,i} = 1 \end{cases}$$

where the second case is hardly probable due to low weights of the error vectors. Hence, the adversary $\mathcal{A}$ can guess error bits efficiently.

Furthermore, the McEliece PKC does not satisfy the non-malleability property even against passive attacks, such as chosen plaintext attacks. Let m and m' denote two messages such that $m' = m \oplus \Delta m$. An adversary $\mathcal{A}$ knows the ciphertext c corresponding to m. Adding rows of the public generator matrix to c the adversary $\mathcal{A}$ can obtain another valid ciphertext c' for the message m' without knowledge of m.

$$c' = c \oplus \hat{G}[i] = (m + \Delta m)\hat{G} \oplus e = m'\hat{G} \oplus e$$

where $i \in I$, and $I = \{i_1, i_2, \cdots\}$ is a set of coordinates of one's in Δm.

In addition, the McEliece PKC is not semantically secure in the random oracle model against adaptive chosen ciphertext attack. The adversary $\mathcal{A}$ can decrypt any ciphertext c in the following way. He asks the decryption oracle for decryption of the ciphertext c' to obtain the corresponding message m'. If the ciphertext c' has been constructed as above, $\mathcal{A}$ can easily obtain the target message $m = m' \oplus \Delta m$.

To harden the McEliece PKC against weaknesses mentioned above and to achieve (IND-)CPA and (IND-)CCA2 security Kobara and Imai proposed three specific conversions [KI01]. Within the scope of this thesis the specific conversion γ has been implemented. This conversion is discussed in Chapter 5.

3 Aspects of coding theory

In this chapter a short introduction to the basic definitions of coding theory based on [HP03], [Sti08], and [MB09] is given.

3.1 Linear codes

Definition 3.1.1 *Let $\mathbb{F}_q$ denote a finite field and $\mathbb{F}_q^n$ a vector space of n tuples over $\mathbb{F}_q$. An* [n,k]-linear code $\mathcal{C}$ *is a k-dimensional vector subspace of $\mathbb{F}_q^n$. The vectors $(a_1, a_2, \ldots, a_{q^k}) \in \mathcal{C}$ are called* codewords *of $\mathcal{C}$.*

An important property of a code is the minimum distance between two codewords.

Definition 3.1.2 *The* Hamming distance $d(x, y)$ *between two vectors $x, y \in \mathbb{F}_q^n$ is defined to be the number of positions at which corresponding symbols x_i, y_i, $\forall 1 \leq i \leq n$ are different. The* Hamming weight $wt(x)$ *of a vector $x \in \mathbb{F}_q^n$ is defined as Hamming distance $d(x, 0)$ between x and the zero-vector.*

The minimum distance of a code $\mathcal{C}$ is the smallest distance between two distinct vectors in $\mathcal{C}$. A code $\mathcal{C}$ is called [n,k,d]-code if its minimum distance is $d = min_{x,y \in \mathcal{C}} d(x, y)$. The error-correcting capability of an [n,k,d]-code is $t = \left\lfloor \frac{d-1}{2} \right\rfloor$.
The two most common ways to represent a code are either the representation by a generator matrix or a parity-check matrix.

Definition 3.1.3 *A matrix $G \in \mathbb{F}_q^{k \times n}$ is called* generator matrix *for an [n,k]-code $\mathcal{C}$ if it's rows form a basis for $\mathcal{C}$ such that $\mathcal{C} = \{x \cdot G \mid x \in \mathbb{F}_q^k\}$. In general there are many generator matrices for a code. An* information set *of $\mathcal{C}$ is a set of coordinates corresponding to any k independent columns of G while the remaining $n - k$ columns of G form the* redundancy set *of $\mathcal{C}$.*

If G is of the form $[I_k|Q]$, where I_k it the $k \times k$ identity matrix, then the first k rows of G form the information set for $\mathcal{C}$. Such a generator matrix G is said to be in standard (systematic) form.

Definition 3.1.4 *For any [n,k]-code $\mathcal{C}$ there exists a matrix $H \in \mathbb{F}_q^{n\times(n-k)}$ with $(n-k)$ independent rows such that $\mathcal{C} = \{y \in \mathbb{F}_q^n \mid H \cdot y^T = 0\}$. Such a matrix H is called* parity-check matrix *for $\mathcal{C}$. In general, there are several possible parity-check matrices for $\mathcal{C}$.*

If G is in systematic form then H can be easily computed and is of the form $[-Q^T|I_{n-k}]$ where I_{n-k} is the $(n-k)\times(n-k)$ identity matrix.

Since the rows of H are independent, H is a generator matrix for a code $\mathcal{C}^\perp$ called dual or orthogonal to $\mathcal{C}$. Hence, if G is generator matrix and H parity-check matrix for $\mathcal{C}$ then H and G are generator and parity-check matrices, respectively, for $\mathcal{C}^\perp$.

Definition 3.1.5 *A* dual code $\mathcal{C}^\perp$ *to $\mathcal{C}$ is an [n,n-k]-code defined by $\mathcal{C}^\perp = \{x \in \mathbb{F}_q^n \mid x \cdot y = 0,\ \forall y \in \mathcal{C}\}$.*

3.2 Punctured and shortened codes

There are many possibilities to obtain new codes by modifying other codes. In this section we present two of them: punctured codes and shortened codes. These types of codes are used for the construction of the quasi-dyadic McEliece variant discussed in Section 4.

Let $\mathcal{C}$ be an [n,k,d]-linear code over $\mathbb{F}_q$. A *punctured code* $\mathcal{C}^*$ can be obtained from $\mathcal{C}$ by deleting the same coordinate i in each codeword. If $\mathcal{C}$ is represented by the generator matrix G then the generator matrix for $\mathcal{C}^*$ can be obtained by deleting the i-th column of the generator matrix for $\mathcal{C}$. The resulting code is an

- $[n-1, k, d-1]$-linear code if $d > 1$ and $\mathcal{C}$ has a minimum weight codeword with a nonzero i-th coordinate
- $[n-1, k, d]$-linear code if $d > 1$ and $\mathcal{C}$ has no minimum weight codeword with a nonzero i-th coordinate
- $[n-1, k, 1]$-linear code if $d = 1$ and $\mathcal{C}$ has no codeword of weight 1 whose nonzero entry is in coordinate i
- $[n-1, k-1, d^*]$-linear code with $d^* \geq 1$ if $d = 1$, $k > 1$ and $\mathcal{C}$ has a codeword of weight 1 whose nonzero entry is in coordinate i

It is also possible to puncture a code $\mathcal{C}$ on several coordinates. Let T denote a coordinate set of size s. The code C^T is obtained from $\mathcal{C}$ by deleting components indexed by the set T in each codeword of $\mathcal{C}$. The resulting code is an $[n-s, k^*, d^*]$-linear code with dimension $k^* \geq k - s$ and minimum distance $d^* \geq d - s$ by introduction.

Punctured codes are closely related to shortened codes. Consider the code $\mathcal{C}$ and a coordinate set T of size s. Let $\mathcal{C}(T) \subseteq \mathcal{C}$ be a subcode of $\mathcal{C}$ with codewords

which are zero on T. A *shortened code* $\mathcal{C}_T$ of length $n - s$ is obtained from $\mathcal{C}$ by puncturing the subcode $\mathcal{C}(T)$ on the set T.

The relationship between shortened and punctured codes is represented by the following theorem.

Theorem 3.2.1 *Let $\mathcal{C}$ be an [n,k,d]-code over $\mathbb{F}_q$ and T a set of s coordinates.*

1. $(\mathcal{C}^\perp)_T = (\mathcal{C}^T)^\perp$ *and* $(\mathcal{C}^\perp)^T = (\mathcal{C}_T)^\perp$, *and*
2. *if $s < d$ then $\mathcal{C}^T$ has dimension k and $(\mathcal{C}^\perp)_T$ has dimension $n - s - k$*
3. *if $s = d$ and T is the set of coordinates where a minimum weight codeword is nonzero, then $\mathcal{C}^T$ has dimension $k-1$ and $(\mathcal{C}^\perp)_T$ has dimension $n-d-k+1$*

3.3 Subfield subcodes and Trace codes

Many well-known and important codes can be constructed over a field $\mathbb{F}_p$ by restricting a code defined over an extension field $\mathbb{F}_q$, where $q = p^d$ for some prime power p and extension degree d.

Definition 3.3.1 *Let $\mathbb{F}_p$ be a subfield of the finite field $\mathbb{F}_q$ and let $\mathcal{C} \subseteq \mathbb{F}_q^n$ be a code of length n over $\mathbb{F}_q$. A subfield subcode $\mathcal{C}_{SUB}$ of $\mathcal{C}$ over $\mathbb{F}_p$ is the vector space $\mathcal{C} \cap \mathbb{F}_p^n$. The dimension of a subfield subcode is $dim(\mathcal{C}_{SUB}) \leq dim(\mathcal{C})$.*

Another way to derive a code over $\mathbb{F}_p$ from a code over $\mathbb{F}_q$ is to use the trace mapping $Tr : \mathbb{F}_q \to \mathbb{F}_p$ which maps an element of $\mathbb{F}_q$ to the corresponding element of $\mathbb{F}_p$.

Definition 3.3.2 *Let $Tr(a)$ denote the trace of an element $a = (a_0, a_1, \ldots, a_n) \in \mathbb{F}_q^n$ such that $Tr(a) = (Tr(a_0), Tr(a_1), \ldots, Tr(a_n)) \in \mathbb{F}_p^n$.* A Trace code *$\mathcal{C}_{Tr} = Tr(\mathcal{C}) := \{Tr(c) \mid c \in \mathcal{C}\} \subseteq \mathbb{F}_p^n$ is a code over $\mathbb{F}_p$ obtained from a code $\mathcal{C}$ over $\mathbb{F}_q$ by the trace construction. The dimension of a Trace code is $dim(\mathcal{C}_{Tr}) \leq d \cdot dim(\mathcal{C})$.*

For instance, let $\mathcal{C}$ be a code over $\mathbb{F}_q$ defined by the parity-check matrix $H \in \mathbb{F}_q^{t \times n}$ with elements $h_{i,j} \in \mathbb{F}_q = \mathbb{F}_p[x]/g(x)$ for some irreducible polynomial $g(x) \in \mathbb{F}_p[x]$ of degree d.

$$H := \begin{pmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n-1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{t-1,0} & h_{t-1,1} & \cdots & h_{t-1,n-1} \end{pmatrix}$$

The elements $h_{i,j} \in \mathbb{F}_q$ of H can be represented as polynomials $h_{i,j}(x) = h_{(i,j),d-1} \cdot x^{d-1} + \cdots + h_{(i,j),1} \cdot x + h_{(i,j),0}$ of degree $d-1$ with coefficients in $\mathbb{F}_p$. The trace construction derives from $\mathcal{C}$ the Trace code $\mathcal{C}_{Tr}$ by writing the $\mathbb{F}_p$ coefficients of each element $h_{i,j}$ onto d successive rows of a parity-check matrix $H_{CTr} \in \mathbb{F}_p^{dt \times n}$ for the Trace code. Consequently, H_{CTr} is the *trace parity-check matrix* for $\mathcal{C}$.

$$H_{CTr} := \begin{pmatrix} h_{(0,0),0} & h_{(0,1),0} & \cdots & h_{(0,n-1),0} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(0,0),d-1} & h_{(0,1),d-1} & \cdots & h_{(0,n-1),d-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(t-1,0),0} & h_{(t-1,1),0} & \cdots & h_{(t-1,n-1),0} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(t-1,0),d-1} & h_{(t-1,1),d-1} & \cdots & h_{(t-1,n-1),d-1} \end{pmatrix}$$

The co-trace parity-check matrix H'_{CTr} for $\mathcal{C}$, which is equivalent to $H_{CTr} \in \mathbb{F}_p^{dt \times n}$ by a left permutation, can be obtained from H analogously, by writing the $\mathbb{F}_p$ coefficients of terms of equal degree from all components on a column of H onto successive rows of H'_{CTr}.

$$H'_{CTr} := \begin{pmatrix} h_{(0,0),0} & h_{(0,1),0} & \cdots & h_{(0,n-1),0} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(t-1,0),0} & h_{(t-1,1),0} & \cdots & h_{(t-1,n-1),0} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(0,0),d-1} & h_{(0,1),d-1} & \cdots & h_{(0,n-1),d-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{(t-1,0),d-1} & h_{(t-1,1),d-1} & \cdots & h_{(t-1,n-1),d-1} \end{pmatrix}$$

Subfield subcodes are closely related to Trace codes by the Delsarte-Theorem.

Theorem 3.3.3 (Delsarte) *For a code $\mathcal{C}$ over $\mathbb{F}_q$, $(\mathcal{C}_{SUB})^\perp = (\mathcal{C}|_{\mathbb{F}_p})^\perp = Tr(\mathcal{C}^\perp)$.*

That means, given an [n,t]-code $\mathcal{C}^\perp$ defined by the parity-check matrix $H \in \mathbb{F}_q^{t \times n}$ dual to an [n,n-t]-code $\mathcal{C}$ defined by the generator matrix $G \in \mathbb{F}_q^{(n-t) \times n}$ the trace construction can be used to efficiently derive from $\mathcal{C}^\perp$ a subfield subcode defined by the parity-check matrix $H_{SUB} \in \mathbb{F}_p^{dt \times n}$.

3.4 Goppa codes

One of the most important families of codes are Goppa codes introduces by V. D. Goppa in 1970 [Gop70]. Binary Goppa codes form a family of binary linear codes

generated by a Goppa polynomial $G(x) = \sum_{i=0}^{t} g_i x^i$ of degree t with coefficients taken in a finite field $\mathbb{F}_q$ where $q = 2^m$ and a subset $L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n$, whose elements L_i are not roots of $G(x)$. Lower bounds on their dimension and minimum distance are known, as well as an efficient polynomial-time decoding algorithm. Goppa codes can be defined in various ways.

Theorem 3.4.1 *Let L be a sequence $L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n$ of distinct elements and $G(x)$ a Goppa polynomial of degree t where $G(L_i) \neq 0$, $\forall\, 0 \leq i \leq n-1$. For any vector $c = (c_0, \ldots, c_{n-1}) \in \mathbb{F}_p^n$ we define the* syndrome *of c by*

$$S_c(x) = -\sum_{i=0}^{n-1} \frac{c_i}{G(L_i)} \frac{G(x) - G(L_i)}{x - L_i} \mod G(x) \equiv \sum_{i=0}^{n-1} \frac{c_i}{x - L_i} \mod G(x).$$

The binary Goppa code $\Gamma(L, G(x))$ *is defined as the following subspace of $\mathbb{F}_p^n$.*

$$\Gamma(L, G(x)) = \{c \in \mathbb{F}_p^n \mid S_c(x) \equiv 0 \mod G(x)\}$$

Equivalently, a Goppa code can be defined by its parity-check matrix.

$$\Gamma(L, G(x)) = \{c \in \mathbb{F}_p^n \mid H \cdot c^T \equiv 0 \mod G(x)\}$$

The parity-check matrix for a binary Goppa code can be derived from the equation $\frac{G(x)-G(L_i)}{G(L_i)(x-L_i)}$ used for the syndrome computation. We see that a vector c is in $\Gamma(L, G(x))$ if and only if

$$\sum_{i=0}^{n-1} \left(\frac{1}{G(L_i)} \sum_{j=s+1}^{t} g_j L_i^{j-s-1} \right) c_i = 0,\ \forall\, 0 \leq s \leq t-1$$

Hence, if $G(x)$ is a monic polynomial then H is of the form

$$H := \begin{pmatrix} \frac{1}{G(L_0)} & \cdots & \frac{1}{G(L_{n-1})} \\ \frac{g_{t-1}+L_0}{G(L_0)} & \cdots & \frac{g_{t-1}+L_{n-1}}{G(L_{n-1})} \\ \vdots & \ddots & \vdots \\ \frac{g_1+g_2L_0+\ldots+g_{t-1}L_0^{t-2}+L_0^{t-1}}{G(L_0)} & \cdots & \frac{g_1+g_2L_{n-1}+\ldots+g_{t-1}L_{n-1}^{t-2}+L_{n-1}^{t-1}}{G(L_{n-1})} \end{pmatrix} =$$

$$= \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ g_{t-1} & 1 & 0 & \cdots & 0 \\ g_{t-2} & g_{t-1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & 1 \end{pmatrix} * \begin{pmatrix} 1 & \cdots & 1 \\ L_0 & \cdots & L_{n-1} \\ L_0^2 & \cdots & L_{n-1}^2 \\ \vdots & \ddots & \vdots \\ L_0^{t-1} & \cdots & L_{n-1}^{t-1} \end{pmatrix} * \begin{pmatrix} \frac{1}{G(L_0)} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \frac{1}{G(L_{n-1})} \end{pmatrix}$$

An alternative way to define Goppa codes is to treat them as subfield subcodes of Generalized Reed-Solomon codes. In that special case Goppa codes are also called alternant codes.

Definition 3.4.2 *Given a sequence $L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n$ of distinct elements and a sequence $D = (D_0, \ldots, D_{n-1}) \in \mathbb{F}_q^n$ of nonzero elements, the Generalized Reed-Solomon code $GRS_t(L, D)$ is the [n,k,t+1] linear error-correcting code defined by the parity-check matrix $H_{L,D} = vdm(t, L) \cdot Diag(D)$ where $vdm(t, L)$ denotes the $t \times n$ Vandermonde matrix with elements $vdm_{ij} = L_j^i$.*

$$H_{L,D} := \begin{pmatrix} D_0 & D_1 & \cdots & D_{n-1} \\ D_0 L_0 & D_1 L_1 & \cdots & D_{n-1} L_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ D_0 L_0^{t-1} & D_1 L_1^{t-1} & \cdots & D_{n-1} L_{n-1}^{t-1} \end{pmatrix}$$

Definition 3.4.3 *Given a prime power $p = 2^s$ for some s, $q = p^d = 2^m$ where $m = s \cdot d$ for some d, a sequence $L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n$ of distinct elements and a polynomial $G(x) \in \mathbb{F}_q[x]$ of degree t such that $G(L_i) \neq 0$ for $0 \leq i < n$, the Goppa code $\Gamma(L, G(x))$ over $\mathbb{F}_p$ is the subfield subcode over $\mathbb{F}_p$ corresponding to $GRS_t(L, D)$ where $D = (G(L_0)^{-1}, \ldots, G(L_{n-1})^{-1})$, and its minimum distance is at least $2t+1$. The Goppa code $\Gamma(L, G(x)) = Tr(GRS_t(L, D))$ over $\mathbb{F}_p$ is derived from $GRS_t(L, D)$ over $\mathbb{F}_q$ through trace construction. A dual code $\Gamma(L, G(x))^\perp \supset GRS_{n-t}(L, D')|_{\mathbb{F}_p}$ is a subfield subcode over $\mathbb{F}_p$ of the Generalized Reed Solomon code $GRS_{n-t}(L, D')$ where $D'_i = G(L_i)/\prod_{i \neq j}(L_i - L_j)$.*

In the original McEliece cryptosystem binary irreducible Goppa codes are used. A Goppa code is *irreducible* if the used Goppa polynomial $G(x)$ is irreducible over $\mathbb{F}_q$. In this case the Goppa code can correct up to t errors.

If $G(x) = \prod_{i=0}^{t-1}(x - z_i)$ is a monic polynomial with t *distinct* roots all in $\mathbb{F}_q$ then it is called *separable* [1] over $\mathbb{F}_q$. In this case the Goppa code can also correct t errors.

A Goppa code generated by a separable polynomial over $\mathbb{F}_q$ admits a parity-check matrix in Cauchy form [MS97].

Definition 3.4.4 *Given two disjoint sequences $z = (z_0, \ldots, z_{t-1}) \in \mathbb{F}_q^t$ and $L = (L_0, \ldots, L_{n-1}) \in \mathbb{F}_q^n$ of distinct elements, the Cauchy matrix $C(z, L)$ is the $t \times n$ matrix with elements $C_{ij} = 1/(z_i - L_j)$, i.e.*

[1] Note that every monic irreducible polynomial $G(x) = x^t + g_{t-1} \cdot x^{t-1} + \cdots + g_0$ with $g_i \neq 0$ for some $i \not\equiv 0 \mod Char(\mathbb{F}_{q^s})$ is separable over an extension $\mathbb{F}_{q^s}$ of $\mathbb{F}_q$

Theorem 3.4.5 *The Goppa code generated by a monic polynomial* $G(x) = (x - z_0) \cdots (x - z_{t-1})$ *without multiple zeros admits a parity-check matrix of the form* $H = C(z, L)$, *i.e.* $H_{ij} = 1/(z_i - L_j)$, $0 \leq i < t, 0 \leq j < n$.

$$C(z, L) := \begin{pmatrix} \frac{1}{z_0 - L_0} & \cdots & \frac{1}{z_0 - L_{n-1}} \\ \vdots & \ddots & \vdots \\ \frac{1}{z_{t-1} - L_0} & \cdots & \frac{1}{z_{t-1} - L_{n-1}} \end{pmatrix}$$

3.5 Dyadic Goppa codes

In [MB09] Barreto and Misoczki have shown how to build binary Goppa codes which admit a parity-check matrix in dyadic form. The family of dyadic Goppa codes offers the advantage of having a compact and simple description.

In this proposal the authors make an extensive use of the fact that using Goppa polynomials separable over $\mathbb{F}_q$ the resulting Goppa code admits a parity-check matrix in Cauchy form by Theorem 3.4.5. Hence, it is possible to construct parity-check matrices which are in Cauchy and dyadic form, simultaneously.

Definition 3.5.1 *Let* $\mathbb{F}_q$ *denote a finite field and* $h = (h_0, h_1, \ldots, h_{n-1}) \in \mathbb{F}_q$ *a sequence of* $\mathbb{F}_q$ *elements. The* dyadic matrix $\Delta(h) \in \mathbb{F}_q^n$ *is the symmetric matrix with elements* $\Delta_{ij} = h_{i \oplus j}$. *The sequence* h *is called* signature *of* $\Delta(h)$ *and coincides with the first row of* $\Delta(h)$. *Given* $t > 0$, $\Delta(h, t)$ *denotes* $\Delta(h)$ *truncated to its first* t *rows.*

When n is a power of 2 every 1×1 matrix is a dyadic matrix, and for $k > 0$ any $2^k \times 2^k$ matrix $\Delta(h)$ is of the form $\Delta(h) := \begin{pmatrix} A & B \\ B & A \end{pmatrix}$ where A and B are dyadic $2^{k-1} \times 2^{k-1}$ matrices.

Theorem 3.5.2 *Let* $H \in \mathbb{F}_q^{n \times n}$ *with* $n > 1$ *be a dyadic matrix* $H = \Delta(h)$ *for some signature* $h \in \mathbb{F}_q^n$ *and a Cauchy matrix* $C(z, L)$ *for two disjoint sequences* $z \in \mathbb{F}_q^n$ *and* $L \in \mathbb{F}_q^n$ *of distinct elements, simultaneously. It follows that*

- $\mathbb{F}_q$ *is a field of characteristic 2*
- h *satisfies* $\frac{1}{h_{i \oplus j}} = \frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}$
- *the elements of* z *are defined as* $z_i = \frac{1}{h_i} + \omega$, *and*
- *the elements of* L *are defined as* $L_i = \frac{1}{h_j} + \frac{1}{h_0} + \omega$ *for some* $\omega \in \mathbb{F}_q$

It is obvious that a signature h describing such a dyadic Cauchy matrix cannot be chosen completely at random. Hence, the authors suggest only choosing nonzero distinct h_0 and h_i at random, where i scans all powers of two smaller than n, and to compute all other values for h by $h_{i\oplus j} = \frac{1}{\frac{1}{h_i}+\frac{1}{h_j}+\frac{1}{h_0}}$ for $0 < j < i$.

In the following an algorithm for the construction of binary Goppa codes in dyadic form is presented.

Algorithm 3.5.1 takes as input three integers: q, N, and t. The first integer $q = p^d = 2^m$ where $m = s \cdot d$ defines the finite field $\mathbb{F}_q$ as degree d extension of $\mathbb{F}_p = \mathbb{F}_{2^s}$. The code length N is a power of two such that $N \leq q/2$. The integer t denotes the number of errors correctable by the Goppa code. Algorithm 3.5.1 outputs the support L, a separable polynomial $G(x)$, as well as the dyadic parity-check matrix $H \in \mathbb{F}_q^{t\times N}$ for the binary Goppa code $\Gamma(L, G(x))$ of length N and designed minimum distance $2t + 1$.

Furthermore, Algorithm 3.5.1 generates the essence η of the signature h of H where $\eta_r = \frac{1}{h_{2^r}} + \frac{1}{h_0}$ for $r = 0, \ldots, \lfloor \lg N \rfloor - 1$ with $\eta_{\lfloor \lg N \rfloor} = \frac{1}{h_0}$, so that, for $i = \sum_{k=0}^{\lfloor \lg N \rfloor - 1} i_k 2^k$, $\frac{1}{h_i} = \eta_{\lfloor \lg N \rfloor} + \sum_{k=0}^{\lfloor \lg N \rfloor - 1} i_k \eta_k$. The first $\lfloor \lg t \rfloor$ elements of η together with $\lfloor \lg N \rfloor$ completely specify the roots of the Goppa polynomial $G(x)$, namely, $z_i = \eta_{\lfloor \lg N \rfloor} + \sum_{k=0}^{\lfloor \lg t \rfloor - 1} i_k \eta_k$.

The number of possible dyadic Goppa codes which can be produced by Algorithm 3.5.1 is the same as the number of distinct essences of dyadic signatures corresponding to Cauchy matrices. This is about $\prod_{i=0}^{\lfloor \lg N \rfloor}(q - 2^i)$. The algorithm also produces equivalent essences where the elements corresponding to the roots of the Goppa polynomial are only permuted. That leads to simple reordering of those roots. As the Goppa polynomial itself is defined by its roots regardless of their order, the actual number of possible Goppa polynomials is $\left(\prod_{i=0}^{\lfloor \lg N \rfloor}(q - 2^i)\right) / (\lfloor \lg N \rfloor)!$.

3.6 Quasi-Dyadic Goppa codes

A cryptosystem cannot be securely defined using completely dyadic Goppa codes which admit a parity-check matrix in Cauchy form. By solving the overdefined linear system $\frac{1}{H_{ij}} = z_i + L_j$ with nt equations and $n + t$ unknowns the Goppa polynomial $G(x)$ would be revealed immediately. Hence, Barreto and Misoczki propose using binary Goppa codes in quasi-dyadic form for cryptographic applications.

Definition 3.6.1 *A quasi-dyadic matrix is a possibly non-dyadic block matrix whose component blocks are dyadic submatrices.*

Algorithm 3.5.1 Construction of binary dyadic Goppa codes

Input: q (a power of 2), $N \leq q/2$, t
Output: L, $G(x)$, H, η

1. $U \leftarrow U \setminus \{0\}$

   ▷ Choose the dyadic signature $(h_0, \ldots, h_{n-1})$. Note that whenever h_j with $j > 0$ is taken from U, so is $1/(1/h_j + 1/h_0)$ to prevent a potential spurious intersection between z and L.
2. $h_0 \xleftarrow{\$} U$
3. $\eta_{\lfloor \lg N \rfloor} \leftarrow \frac{1}{h_0}$
4. $U \leftarrow U \setminus \{h_0\}$
5. **for** $r \leftarrow 0$ **to** $\lfloor \lg N \rfloor - 1$ **do**
6. $\quad i \leftarrow 2^r$
7. $\quad h_i \xleftarrow{\$} U$
8. $\quad \eta_r \leftarrow \frac{1}{h_i} + \frac{1}{h_j}$
9. $\quad U \leftarrow U \setminus \left\{ h_i, \frac{1}{\frac{1}{h_i} + \frac{1}{h_j}} \right\}$
10. $\quad$ **for** $j \leftarrow 1$ **to** $i - 1$ **do**
11. $\quad\quad h_{i \oplus j} \leftarrow \frac{1}{\frac{1}{h_i} + \frac{1}{h_j} + \frac{1}{h_0}}$
12. $\quad\quad U \leftarrow U \setminus \left\{ h_{i \oplus j}, \frac{1}{\frac{1}{h_{i \oplus j}} + \frac{1}{h_0}} \right\}$
13. $\omega \xleftarrow{\$} \mathbb{F}_q$

    ▷ Assemble the Goppa polynomial
14. **for** $i \leftarrow 0$ **to** $t - 1$ **do**
15. $\quad z_i \leftarrow \frac{1}{h_i} + \omega$
16. $G(x) \leftarrow \prod_{i=0}^{t-1} (x - z_i)$

    ▷ Compute the support
17. **for** $j \leftarrow 0$ **to** $N - 1$ **do**
18. $\quad L_j \leftarrow \frac{1}{h_j} + \frac{1}{h_0} + \omega$
19. $h \leftarrow (h_0, \ldots, h_{N-1})$
20. $H \leftarrow \Delta(t, h)$
21. **return** L, $G(x)$, H, η

A quasi-dyadic Goppa code over $\mathbb{F}_p = \mathbb{F}_{2^s}$ for some s is obtained by constructing a dyadic parity-check matrix $H_{dyad} \in \mathbb{F}_q^{t \times n}$ over $\mathbb{F}_q = \mathbb{F}_{p^d} = \mathbb{F}_{2^m}$ of length $n = lt$ where n is a multiple of the desired number of errors t, and then computing the co-trace matrix $H'_{Tr} = Tr'(H_{dyad}) \in \mathbb{F}_p^{dt \times n}$. The resulting parity-check matrix

for the quasi-dyadic Goppa code is a non-dyadic matrix composed of blocks of dyadic submatrices by Theorem 3.6.2.

Theorem 3.6.2 *The co-trace matrix* $H'_{Tr} \in \mathbb{F}_p^{dt \times lt}$ *of a dyadic matrix* $H_{dyad} \in \mathbb{F}_q^{t \times lt}$ *is quasi-dyadic and consists of dyadic blocks of size* $t \times t$ *each.*

Proof sketch:
To prove this theorem we consider a dyadic block B over $\mathbb{F}_q$ of size 2×2 which is the minimum block of a dyadic parity-check matrix for a binary Goppa code.

$$B := \begin{pmatrix} h_0 & h_1 \\ h_1 & h_0 \end{pmatrix}$$

The co-trace construction (see Section 3.3) derives from B a matrix of the following form.

$$B'_{Tr} := \begin{pmatrix} h_{0,0} & h_{1,0} \\ h_{1,0} & h_{0,0} \\ h_{0,1} & h_{1,1} \\ h_{1,1} & h_{0,1} \end{pmatrix}$$

It is not hard to see that B'_{Tr} is no more dyadic but consists of dyadic blocks over $\mathbb{F}_p$ of size 2×2 each. The quasi-dyadicity of $B'_{i,Tr}$ can be shown recursively for all blocks B_i. Consequently, the complete co-trace matrix $Tr'(H_{dyad})$ is quasi-dyadic over $\mathbb{F}_p$.

3.7 Goppa decoding

3.7.1 Patterson's decoding algorithm

In [Pat75] Patterson introduced an efficient algorithm for decoding of binary Goppa codes. The algorithm corrects, in polynomial time, t errors in a classical (irreducible) binary Goppa code of length n and degree t. In this section the functionality of the Patterson's decoding algorithm is explained.

The first step of the decoding algorithm is syndrome computation. For the syndrome of a ciphertext $c = v \oplus e$ where e is an error vector of weight at most t added to the codeword v the following holds

$$S_c(x) = S_v(x) + S_e(x) \equiv S_e(x) \mod G(x)$$

because the syndrome of a codeword $v \in \Gamma(L, G(x))$ is $S_v(x) \equiv 0 \mod G(x)$ by definition.

The goal of the decoding algorithm is to find a polynomial $\sigma(x)$ of degree $t' \leq t$ where t' is the actual number of errors in c. The polynomial $\sigma(x)$ is called error locator polynomial and defined as

$$\sigma(x) = \prod_{i \in E} (x - \gamma_i)$$

where E is a sequence of integers determining error positions in c (equivalent, the positions of ones in e). The roots γ_i of the error locator polynomial are then elements of the support L for the binary Goppa code $\Gamma(L, G(x))$ where the positions of these elements inside of L correspond to error positions in c.

The error locator polynomial $\sigma(x)$ satisfies the equation

$$\sigma(x) S_c(x) \equiv \sigma'(x) \tag{3.1}$$

where $\sigma'(x)$ is the formal derivate of $\sigma(x)$. Since the characteristic of $\mathbb{F}_q$ is 2 the derivate $\sigma'(x)$ is obtained by splitting $\sigma(x)$ in squares and non-squares such that

$$\sigma(x) = a(x)^2 + x \cdot b(x)^2$$

where $\sigma'(x) = b(x)^2$. If c is not a codeword the syndrome $S_c(x)$ has an inverse $T(x) \equiv S_c(x)^{-1}$ in $\mathbb{F}_q$. Hence, the Equation 3.1 can be written as

$$a(x)^2 \equiv (T(x) + x) b(x)^2 \mod G(x) \tag{3.2}$$

To solve Equation 3.2 in order to obtain $a(x)$ and $b(x)$, and thus to compute $\sigma(x)$, the Decoding algorithm 3.7.1 proceeds as follows. It first computes the inverse $T(x)$ of $S_c(x)$. In the next step it computes the square root $R(x)$ of $T(x) + x$ satisfying the equation $a(x) \equiv R(x)b(x) \mod G(x)$. By an observation due to Huber [Hub96] the polynomial $R(x)$ can be represented as $R(x) = R_0(x) + W(x)R_1(x)$ where $W(x)^2 \equiv x \mod G(x)$ which can be precomputed for every Goppa polynomial. Hence, the square root computation can be done by splitting $R(x)^2 = T(x) + x = R_0(x)^2 + x \cdot R_1(x)^2$ in squares and non-squares and computing square roots of both.

The extended Euclidean algorithm (EEA), e.g. presented in [MVO96], can be used to obtain $a(x)$ and $b(x)$ where $deg(a(x)) \leq \lfloor t/2 \rfloor$ and $deg(b(x)) \leq \lfloor (t-1)/2 \rfloor$. We start the computation with $a_{-1}(x) = 0$, $a_0(x) = G(x)$, $b_{-1}(x) = R(x)$, and $b_0(x) = 1$. After each step i of the EEA the degree of $a_i(x)$ decreases while the degree of $b_i(x)$ increases. There is an unique point k where the degree of both polynomials $a_k(x)$ and $b_k(x)$ is below the respective bound.

This is when the degree of $a_k(x)$ drops below $\lfloor (t+1)/2 \rfloor$ for the first time. Thus, we obtain both polynomials $a(x) = a_k(x)$ and $b(x) = b_k(x)$ of desired degrees and can compute $\sigma(x)$.

The next and the most computationally expensive step of the decoding algorithm is root computation of the error locator polynomial. There are several ways to do that, e.g. using Berlekamp trace algorithm, Chien search, Horner scheme, or the simple polynomial evaluation method, explained in the next section.

Then errors can easily be corrected by finding the positions of roots γ_i of $\sigma(x)$ in L and flipping the corresponding bits of c.

Algorithm 3.7.1 Decoding Algorithm for binary Goppa codes

Input: ciphertext vector c, Goppa code $\Gamma(L, G(x))$
Output: message vector m, error vector e

1. Compute the syndrome $S_c(x)$ of c, e.g. by
   $S_c(x) \equiv \sum_{i=0}^{n-1} \frac{c_i}{x - L_i} \mod G(x)$, or by
   $S_c(x) = H \cdot c^T$
2. **if** $S_c(x) \equiv 0 \mod G(x)$ **then**
3. ▷ c is a codeword
4. **return** (c,0)
5. **else**
6. ▷ Get error locator polynomial
7. $T(x) \equiv S_c(x)^{-1} \mod G(x)$
8. $R(x) \equiv \sqrt{T(x) + x} \mod G(x)$
9. Compute $a(x)$ and $b(x)$ such that $a(x) \equiv b(x)R(x) \mod G(x)$.
10. Compute error locator polynomial $\sigma(x) = a(x)^2 + x \cdot b(x)^2$
11. Find roots $\gamma = (\gamma_1, \ldots, \gamma_{t'}) \in \mathbb{F}_q^{t'}$ of $\sigma(x)$ where $t' \leq t$
    ▷ Correct errors
12. Set $e = 0$
13. **for** $i \leftarrow 1$ **to** t' **do**
14. Find the position i of γ_i in L
15. Set $e[i] = 1$
16. **return** (c,e)

3.7.2 Finding roots of the error locator polynomial

Polynomial evaluation method

The easiest way to find roots of the error locator polynomial is the simple polynomial evaluation method. Due to the fact that only elements of the support L for

the binary Goppa code $\Gamma(L, G(x))$ can be roots of $\sigma(x)$ it is sufficient to evaluate $\sigma(x)$ putting in elements $L_i \in L$ for the variable x. If the evaluation result is zero a root has been found. When using intermediate results L_i^j computed in previous evaluation steps this method requires $2t-1$ field multiplications and t field additions for an evaluation of a degree t polynomial at a point L_i. In the worst case all n elements of L must be tested to find all roots of $\sigma(x)$. Hence, the time complexity of this root finding method is $n((2 \cdot C_{mul} + C_{add})t - C_{mul})$ where C_{add} and C_{mul} are the time complexities of one addition and multiplication in $\mathbb{F}_q$, respectively.

Horner scheme

The Horner scheme is a more sophisticated polynomial evaluation method. Every polynomial $\sigma(x) = \sigma_0 + \sigma_1 x + \cdots + \sigma_t x^t$ can also be written as

$$\sigma(x) = \sigma_0 + x\,(\sigma_1 + x\,(\sigma_2 + \cdots + x\,(a_{t-1} + x a_t)\cdots))$$

Making use of this nested form the Horner scheme proceeds as follows.

$$\begin{aligned} a_t &\leftarrow \sigma_t \\ a_{t-1} &\leftarrow \sigma_{t-1} + L_i \cdot a_t \\ &\vdots \\ a_1 &\leftarrow \sigma_1 + L_i \cdot a_2 \\ a_0 &\leftarrow \sigma_0 + L_i \cdot a_1 \\ \sigma(L_i) &= a_0 \end{aligned}$$

The Horner scheme involves t field additions and t field multiplications which is about factor two less compared to the simple evaluation method. In the worst case the Horner scheme scans all n elements of the support L. Hence, the time complexity of the Horner scheme is $(C_{add} + C_{mul}) \cdot t \cdot n$ where C_{add} and C_{mul} are the time complexities of one addition and multiplication in $\mathbb{F}_q$, respectively.

Chien search

Another method frequently implemented in hardware is Chien search [Chi64]. The Chien search makes an extensive use of the following relationship which is true for any element $\beta = \alpha^i \in \mathbb{F}_q \setminus \{0\}$.

This way, the evaluation of $\sigma(x)$ can be done at all points $\beta \in \mathbb{F}_q$. We start the search for roots of $\sigma(x)$ with $i = 0$ and use the intermediate results $\gamma_{i,j}$ computed while evaluation of $\sigma(x)$ at the point α^i to compute the next values $\gamma_{j,i+1} = \gamma_{j,i}\alpha^j$ for the evaluation of $\sigma(x)$ at the next point α^{i+1}. The result of the evaluation at

$$\begin{array}{rcllllllll}
\sigma(\alpha^i) & = & \sigma_0 & + & \sigma_1(\alpha^i) & + & \cdots & + & \sigma_{t-1}(\alpha^i)^{t-1} & + & \sigma_t(\alpha^i)^t \\
 & = & \gamma_{0,i} & + & \gamma_{1,i} & + & \cdots & + & \gamma_{t-1,i} & + & \gamma_{t,i} \\
\sigma(\alpha^{i+1}) & = & \sigma_0 & + & \sigma_1(\alpha^{i+1}) & + & \cdots & + & \sigma_{t-1}(\alpha^{i+1})^{t-1} & + & \sigma_t(\alpha^{i+1})^t \\
 & = & \sigma_0 & + & \sigma_1(\alpha^i)\alpha & + & \cdots & + & \sigma_{t-1}(\alpha^i)^{t-1}\alpha^{t-1} & + & \sigma_t(\alpha^i)^t\alpha^t \\
 & = & \gamma_{0,i} & + & \gamma_{1,i}\alpha & + & \cdots & + & \gamma_{t-1,i}\alpha^{t-1} & + & \gamma_{t,i}\alpha^t \\
 & = & \gamma_{0,i+1} & + & \gamma_{1,i+1} & + & \cdots & + & \gamma_{t-1,i+1} & + & \gamma_{t,i+1}
\end{array}$$

the point α^i is obtained summing up the values $\gamma_{j,i}$ for $0 \leq j \leq t$. If the result is 0 then α^i is a root of $\sigma(x)$.

The values $(1, \alpha, \alpha^2, \ldots, \alpha^t)$ can be precomputed and stored in an array. Hence, Chien search involves t field additions and t field multiplications to evaluate a degree t polynomial at a point β. In the worst case Chien search scans all $q - 1 = 2^m - 1$ field elements of $\mathbb{F}_q = \mathbb{F}_{2^m}$. Hence, the time complexity of the Chien search can be denoted as $(C_{add} + C_{mul}) \cdot t \cdot (2^m - 1)$ where C_{add} and C_{mul} are the time complexities of one addition and multiplication in $\mathbb{F}_q$, respectively.

Berlekamp trace algorithm

The Berlekamp trace algorithm was originally published in [Ber70] and is one of the best known algorithms for the factorization of polynomials over finite fields with small characteristic. As the error locator polynomial does not have multiple roots, we can use this algorithm for root finding. In this section we concentrate on finite fields of characteristic two.

Given a finite field $\mathbb{F}_q$ where $q = 2^m$, the trace function $Tr(\cdot)$ of $\mathbb{F}_q$ over $\mathbb{F}_2$ is defined by

$$Tr(x) = x + x^2 + x^{2^2} + \cdots + x^{2^{m-1}}$$

and maps the field $\mathbb{F}_q$ onto it's base field $\mathbb{F}_2$.

Using the trace function every element $\alpha \in \mathbb{F}_q$ can uniquely be represented by the binary m-Tuple $(Tr(\beta_1\alpha), \ldots, Tr(\beta_m\alpha))$ where $(\beta_1, \ldots, \beta_m) = (\alpha, \alpha^2, \ldots, \alpha^m)$ denotes any basis of $\mathbb{F}_q$ over $\mathbb{F}_2$ and α is a primitive element of $\mathbb{F}_{2^m}$.

The core idea of the Berlekamp trace algorithm is that every polynomial $f(x) \in \mathbb{F}_q[x]$ where $f(x)|(x^{2^m} - x)$ splits into two polynomials

$$g(x) = \gcd(f(x), Tr(\beta \cdot x)), \text{ and}$$

$$h(x) = \gcd(f(x), 1 + Tr(\beta \cdot x)) = f(x)/g(x)$$

for any element $\beta \in (\beta_1, \ldots, \beta_m)$.

Repeating this recursively where β iterates through the basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ all roots of $\sigma(x)$ can be found. The first call of the Berlekamp trace algorithm given below is $BTA(\sigma(x), 1)$.

The Berlekamp trace algorithm has time complexity $\mathcal{O}(m \cdot t^2)$ [BH09].

Algorithm 3.7.2 Berlekamp Trace Algorithm

Input: Polynomial $f(x)$, integer i
Output: Roots $(\gamma_1, \ldots, \gamma_t) \in \mathbb{F}_q^t$ of $\sigma(x)$

1. **if** $deg(f) \leq 1$ **then**
2. **return** rootof(f)
3. **else**
4. $g \leftarrow \gcd(f, Tr(\beta_i \cdot x))$
5. $h \leftarrow f/g$
6. **return** $BTA(g, i+1) \cup BTA(h, i+1)$

4 McEliece-type PKC based on quasi-dyadic Goppa codes

In [MB09] Barreto and Misoczki proposed a new method for the reduction of the public key size in the original McEliece PKC. This proposal is based on quasi-dyadic Goppa codes (see Sections 3.5 and 3.6). The family of dyadic codes offers the advantage of having a compact and simple description. The whole dyadic generator matrix for such a code can be described by only its signature of n elements. A cryptosystem cannot be defined securely by a parity-check matrix of a completely dyadic Goppa code, therefore the authors use quasi-dyadic Goppa codes in their proposal. A quasi-dyadic generator matrix for a binary Goppa code is a non-dyadic matrix consisting of dyadic submatrices of size $t \times t$ each. A quasi-dyadic matrix is described by signatures of its dyadic submatrices of t elements each.

The secret transformations of a private quasi-dyadic Goppa code used to obtain a public code must not destroy the quasi-dyadicity. In this case the size of the public key is roughly a factor t smaller than in general. On the other hand, the secret transformations has to be hard to recover so that no attack identifying the underlying private code is possible.

This chapter first describes the transformations frequently used in McEliece variants to disguise a private code as well as the function of these specific transformations. In Section 4.2 the scheme definition of the quasi-dyadic McEliece variant is given. Section 4.3 presents the recommended parameters while Section 4.4 discusses the security of the quasi-dyadic McEliece-type PKC.

4.1 Hiding the structure of the private code

The main security issue in code-based cryptography is hiding the structure of the private code. Let G denote a generator matrix for a private code $\mathcal{C}$, and let $\hat{\mathcal{C}}$ denote a public code obtained from $\mathcal{C}$ by one or more secret transformations. In the following, some usual transformations are summarized, based on [OS08] and [MB09].

(1) Row Scrambler: Multiply the generator matrix G for the private code

$\mathcal{C}$ by a random invertible matrix $S \in \mathbb{F}_q^{k\times k}$ from the left. As $\langle G\rangle = \langle SG\rangle$, the known error correction algorithm for $\mathcal{C}$ can be used. Publishing a systematic generator matrix provides the same security against structural attacks as a random S.

(2) **Column Scrambler/ Isometry:** Multiply the generator matrix G for the private code $\mathcal{C}$ by a random invertible matrix $P \in \mathbb{F}_q^{n\times n}$ from the right, where P preserves the norm, e.g. P is a permutation matrix. If G and P are known then up to t errors can be corrected in $\langle GP\rangle$.

(3) **Subcode:** Let $0 < l < k$. Multiply the generator matrix G for the private code $\mathcal{C}$ by a random matrix $S \in \mathbb{F}_q^{l\times k}$ of full rank from the left. As $\langle SG\rangle \subseteq \langle G\rangle$, the known error correction algorithm may be used.

(4) **Subfield Subcode:** Take the subfield subcode $\mathcal{C}_{SUB}$ of the secret code $\mathcal{C}$ for a subfield $\mathbb{F}_p$ of $\mathbb{F}_q$. As before, one can correct errors by the error correcting algorithm for the secret code. However, sometimes one can correct errors of larger norm in the subfield subcode than in the original code.

(5) **(Block-)Shortening:** Extract a shortened public code $\mathcal{C}^T$ from a very large private code $\mathcal{C}$ by puncturing $\mathcal{C}$ on the set of coordinates T. In particular, if $\mathcal{C}$ is a code defined by a $t \times N$ matrix H, where $N = l \cdot t$, such that H can be considered as a composition of l blocks of size $t \times t$ each, then T_t contains all those coordinates of blocks which have to be deleted in order to obtain a block-shortened code $\mathcal{C}^{T_t}$.

To protect the secret code, a combination of several transformations is used, as a rule. For instance, in the original McEliece cryptosystem a combination of transformations (1), (2) and (4) is used.

In the following, we explain the role of these transformations in hiding the structure of the private code.

In [CC95] Canteaut and Chabaud pointed out that the scrambling transformation (1) has no cryptographic function. It just sends G to another generator matrix G' for the same code to assure that the public generator matrix $\hat{G}$ is not in systematic form. Otherwise, most bits of the message would be revealed. Our goal is to construct a systematic public generator matrix for a binary quasi-dyadic Goppa code and to use a conversion for CCA2-secure McEliece versions (see Chapter 5) to protect the message. Hence, this transformation is neither useful nor necessary for our purpose.

In contrast, the permutation transformation (2) is essential when constructing a trapdoor function. In the following, we consider the permutation equivalence problem of two codes.

Let the symmetric group S_n of order n be a set of permutations of integers $\{0, \ldots, n-1\}$ and $\sigma \in S_n$ be a permutation. P_σ denotes the $n \times n$ permutation matrix with components $p_{i,j} = 1$ if $\sigma(i) = j$ and $p_{i,j} = 0$ otherwise.

Definition 4.1.1 *Two codes $\mathcal{C}_1$ and $\mathcal{C}_2$ are* permutation equivalent *if there is a permutation matrix P_σ such that G_1 is a generator matrix for $\mathcal{C}_1$ if and only if $P \times G_2$ is the generator matrix for $\mathcal{C}_2$. Thus, P sends $\mathcal{C}_1$ to $\mathcal{C}_2$ by reordering the columns of G_1.*

The permutation equivalence problem is a decisional problem defined as follows.

Definition 4.1.2 *Given two $k \times n$ matrices G_1 and G_2 over $\mathbb{F}_q$, does there exist a permutation σ represented as permutation matrix P_σ such that $G_1 \times P_\sigma = G_2$?*

This problem is closely related to the graph isomorphism problem which is assumed to be in $\mathcal{P}/\mathcal{NP}$ [PR97]. The Support splitting algorithm [Sen00] is the only known algorithm which solves the permutation equivalence problem of two codes in the practice. The success probability of the best known attack using the Support splitting algorithm to distinguish a Goppa code from a general linear code is negligible for all suitable McEliece parameters.

The transformation (4) is used implicitly in every McEliece-type cryptosystem based on Goppa codes because Goppa codes can be considered as subfield subcodes of Generalized Reed Solomon codes.

The last transformation (5) is of great significance for the construction of a CCA2-secure McEliece-type cryptosystem based on quasi-dyadic Goppa codes as introduced in [MB09] where the public code is equivalent to a subcode of a Reed Solomon code. Combining the transformations (2) and (5) the equivalent shortened code problem can be defined as follows.

Definition 4.1.3 *Let H be a $t \times N$ matrix over $\mathbb{F}_q$ and $\tilde{H}$ a $t \times n$ matrix over $\mathbb{F}_q$ with $n < N$, does there exist a set of coordinates T of length $N - n$ and a permutation $\sigma \in S_n$ such that $\tilde{H} = H(T) \times P_\sigma$ where $H(T)$ denotes a matrix obtained by deleting of components indexed by the set T in each row of H?*

The equivalent shortened code problem has been proven to be $\mathcal{NP}$-complete by Wieschbrink in [Wie06]. In contrast to the permutation equivalence problem the equivalent shortened code problem cannot be solved by means of the Support splitting algorithm. Hence, no efficient algorithm is known that solves this problem up to now.

4.2 Scheme definition of QD-McEliece

In this section the scheme definition of the quasi-dyadic McEliece variant, i.e. key generation, encryption, and decryption, is given.

4.2.1 Key generation

The main difference between the original McEliece scheme and the quasi-dyadic variant is the Key generation algorithm 4.2.1 shown below. It takes as input the system parameters t, n, and k and outputs a binary Goppa code in quasi-dyadic form over a subfield $\mathbb{F}_p$ of $\mathbb{F}_q$, where $p = 2^s$ for some s, $q = p^d = 2^m$ for some d with $m = ds$. The code length n must be a multiple of t such that $n = lt$ for some $l > d$.

Algorithm 4.2.1 QD-McEliece: Key generation algorithm

Input: Fixed common system parameters: t, $n = l \cdot t$, $k = n - dt$
Output: private key K_{pr}, public key K_{pub}

1. $(L_{dyad}, G(x)\ H_{dyad}, \eta) \leftarrow$ **Algorithm 3.5.1** $(2^m, N, t)$, where $N >> n$, $N = l' \cdot t < q/2$
2. Select uniformly at random l distinct blocks $\left[B_{i_0}|\cdots|B_{i_{l-1}}\right]$ in any order from H_{dyad}
3. Select l dyadic permutations $\Pi^{j_0}, \cdots, \Pi^{j_{l-1}}$ of size $t \times t$ each
4. Select l nonzero scale factors $\sigma_0, \ldots, \sigma_{l-1} \in \mathbb{F}_p$. Notice that, if $p = 2$, then all scale factors are equal to 1.
5. Compute $H = \left[B_{i_0}\Pi^{j_0}|\cdots|B_{i_{l-1}}\Pi^{j_{l-1}}\right] \in (\mathbb{F}_q^{t\times t})^l$
6. Compute $\Sigma = Diag(\sigma_0 I_t, \ldots, \sigma_{l-1} I_t) \in (\mathbb{F}_p^{t\times t})^{l\times l}$
7. Compute the co-trace matrix $H'_{Tr} = Tr'(H\Sigma) = Tr'(H)\Sigma \in (\mathbb{F}_p^{t\times t})^{l\times l}$
8. Bring H'_{Tr} in systematic form $\hat{H} = [Q|I_{n-k}]$, e.g. by means of Gaussian elimination
9. Compute the public generator matrix $\hat{G} = [I_k|Q^T]$
10. **return** $K_{pub} = (\hat{G}, t)$, $K_{pr} = (H_{dyad}, L_{dyad}, \eta, G(x), (i_0, \ldots, i_{l-1}), (j_0, \ldots, j_{l-1}), (\sigma_0, \ldots, \sigma_{l-1}))$

The key generation algorithm proceeds as follows. It first runs the Algorithm 3.5.1 to produce a dyadic code $\mathcal{C}_{dyad}$ of length $N >> n$, where N is a multiple of t not exceeding the largest possible length $q/2$. The resulting code admits a $t \times N$ parity-check matrix $H_{dyad} = \left[B_0|\cdots|B_{N/t-1}\right]$ which can be viewed as a composition of N/t dyadic blocks B_i of size $t \times t$ each.

In the next step the key generation algorithm uniformly selects l dyadic blocks of H_{dyad} of size $t \times t$ each. This procedure leads to the same result as puncturing the code $\mathcal{C}_{dyad}$ on a random set of block coordinates T_t of size $(N - n)/t$ first, and then permuting the remaining l blocks by changing their order. The block permutation sequence $(i_0, \ldots, i_l)$ is the first part of the trapdoor information. It can also be described as an $N \times n$ permutation matrix P_B. Then the selection and permutation of $t \times t$ blocks can be done by right-side multiplication $H_{dyad} \times P_B$.

Further transformations performed to disguise the structure of the private code are dyadic inner block permutations.

Definition 4.2.1 *A* dyadic permutation Π^j *is a dyadic matrix whose signature is the* j*-th row of the identity matrix. A dyadic permutation is an involution, i.e.* $(\Pi^j)^2 = I$*. The* j*-th row (or equivalently the* j*-th column) of the dyadic matrix defined by a signature* h *can be written as* $\Delta(h)_j = h\Pi^j$*.*

The key generation algorithm first chooses a sequence of integers $(j_0, \ldots, j_{l-1})$ defining the positions of ones in the signatures of the l dyadic permutations. Then each block B_i is multiplied by a corresponding dyadic permutation Π^j to obtain a matrix H which defines a permutation equivalent code $\mathcal{C}_H$ to the punctured code $\mathcal{C}_{dyad}^{T_t}$. Since the dyadic inner-block permutations can be combined to an $n \times n$ permutation matrix $P_{dp} = Diag(\Pi^{j_0}, \cdots, \Pi^{j_{l-1}})$ we can write $H = H_{dyad} \cdot P_B \cdot P_{dp}$.

The last transformation is scaling. Therefore, first a sequence $(\sigma_0, \ldots, \sigma_{l-1}) \in F_p$ is chosen, and then each dyadic block of H is multiplied by a diagonal matrix $\sigma_i I_t$ such that $H' = H \cdot \Sigma = H_{dyad} \cdot P_B \cdot P_{dp} \cdot \Sigma$.

Finally, the co-trace construction derives from H' the parity-check matrix H'_{Tr} for a binary quasi-dyadic permuted subfield subcode over $\mathbb{F}_p$. Bringing H'_{Tr} in systematic form, e.g. by means of Gaussian elimination, we obtain a systematic parity-check matrix $\hat{H}$ for the public code. $\hat{H}$ is still a quasi-dyadic matrix composed of dyadic submatrices which can be represented by a signature of length t each and which are no longer associated to a Cauchy matrix.

The generator matrix $\hat{G}$ obtained from $\hat{H}$ defines the public code $\mathcal{C}_{pub}$ of length n and dimension k over $\mathbb{F}_p$, while $\hat{H}$ defines a dual code $\mathcal{C}_{pub}^{\perp}$ of length n and dimension $k = n - dt$.

The trapdoor information consisting of the essence η of the signature h_{dyad}, the sequence $(i_0, \ldots, i_{l-1})$ of blocks, the sequence $(j_0, \ldots, j_{l-1})$ of dyadic permutation identifiers, and the sequence of scale factors $(\sigma_0, \ldots, \sigma_{l-1})$ relates the public code defined by $\hat{H}$ with the private code defined by H_{dyad}.

The public code defined by $\hat{G}$ admits a further parity-check matrix $V_{L^*,G} = \mathtt{vdm}(L^*, G(x)) \cdot \mathtt{Diag}(G(L_i^*)^{-1})$ where L^* is the permuted support obtained from L_{dyad} by $L^* = L_{dyad} \cdot P_B \cdot P_{db}$. Bringing $V_{L^*,G}$ in systematic form leads to the same quasi-dyadic parity-check matrix $\hat{H}$ for the code $\mathcal{C}_{pub}$.

The matrix $V_{L^*,G}$ is permutation equivalent to the parity-check matrix $V_{L,G} = \mathtt{vdm}(L, G(x)) \cdot \mathtt{Diag}(G(L_i)^{-1})$ for the shortened private code $\mathcal{C}_{pr} = \mathcal{C}_{dyad}^{T_t}$ obtained by puncturing the large private code $\mathcal{C}_{dyad}$ on the set of block coordinates T_t. The support L for the code $\mathcal{C}_{pr}$ is obtained by deleting all components of L_{dyad} at the positions indexed by T_t.

Classical irreducible Goppa codes use support sets containing all elements of $\mathbb{F}_q$. Thus, the support corresponding to such a Goppa code can be published while

only the Goppa polynomial and the (support) permutation are parts of the secret key. In contrast, the support sets L and L^* for $\mathcal{C}_{pr}$ and $\mathcal{C}_{pub}$, respectively, are not full but just subsets of $\mathbb{F}_q$ where L^* is a permuted version of L. Hence, the support sets contain additional information and have to be kept secret.

4.2.2 Encoding

The encryption algorithm of the QD-McEliece variant is the same as that of the original McEliece cryptosystem explained in Section 2.1.2. First a message vector is multiplied by the systematic generator matrix $\hat{G}$ for the quasi-dyadic public code $\mathcal{C}_{pub}$ to obtain the corresponding codeword. Then a random error vector of length n and hamming weight at most t is added to the codeword to obtain a ciphertext.

4.2.3 Decoding

The decryption algorithm of the QD-McEliece version is essentially the same as that of the classical McEliece cryptosystem explained in Section 2.1.3. The following decryption strategies are conceivable.

Permute the ciphertext and undo the inner block dyadic permutation as well as the block permutation to obtain an extended permuted ciphertext of length N such that $ct_{perm} = ct \cdot P_B \cdot P_{dp}$. Then use the decoding algorithm of the large private code $\mathcal{C}_{dyad}$ to obtain the corresponding codeword. Multiplying ct_{perm} by the parity-check matrix for $\mathcal{C}_{dyad}$ yields the same syndrome as reversing the dyadic permutation and the block permutation without extending the length of the ciphertext and using a parity-check matrix for the shortened private code $\mathcal{C}_{pr}$.

A better method is to decrypt the ciphertext directly using the equivalent parity-check matrix $V_{L^*,G}$ for syndrome computation. The Patterson's decoding algorithm can be used to detect the error and to obtain the corresponding codeword. Since $\hat{G}$ is in systematic form, the first k bits of the resulting codeword correspond to the encrypted message.

4.3 Parameter choice and key sizes

For an implementation on an embedded microcontroller it is the best choice to use Goppa codes over the base field $\mathbb{F}_2$. In this case the matrix vector multiplication can be performed most efficiently. Hence, the subfield $\mathbb{F}_p = \mathbb{F}_{2^s}$ should be chosen to be the base field itself where $s = 1$ and $p = 2$. Furthermore, as the register size of embedded microcontrollers is restricted to 8 bits it is advisable to construct

subfield subcodes of codes over $\mathbb{F}_{2^8}$ or $\mathbb{F}_{2^{16}}$. In the following, we consider which of both extension fields is the best choice for the construction of quasi-dyadic subfield subcodes.

There are some restrictions on the choice of parameters l and n. For instance, the security parameter l must be greater than the extension degree d of $\mathbb{F}_q = \mathbb{F}_{p^d} = \mathbb{F}_{2^m}$, and thus greater than m, because $m = s \cdot d = 1 \cdot d = d$ when constructing subfield subcodes over the base field. Furthermore, it is suggested that the designed number of errors t is chosen as a power of two to achieve best possible key size reduction. In addition, in this case further optimizations within the scope of the implementation are possible (see Section 6).

Let N be the largest possible length $q/2$ such that for the code length n of the subfield subcode $N >> n$ is true.

Let us consider the extension field $\mathbb{F}_{2^8}$. If $q = 2^8$ then N is at most 2^7 and $l \overset{!}{>} d = 8$. But in this case the number of errors t is at most 2^3 such that $n = 2^3 \cdot l < N$ where $l \in \{9, \ldots 16\}$. It follows that the resulting subfield subcode is of length $72 \leq n \leq 128$ and has code dimension $k = n - dt$ where $8 \leq k \leq 64$. It is not hard to see that these parameters cannot provide a high level of security. Patently, the extension field $\mathbb{F}_{2^8}$ is too small to securely derive subfield subcodes from codes defined over it.

For subfield subcodes over the base subfield $\mathbb{F}_2$ of $\mathbb{F}_{2^{16}}$ Barreto and Misoczki suggest using the parameters summarized in Table 4.1.

level	**t**	**n = l·t**	**k = n - m·t**	**key size** ($m \cdot k$ **bits**)
80	2^6	$36 \cdot 2^6 = 2304$	$20 \cdot 2^6 = 1280$	$20 \cdot 2^{10}$ bits = 20 Kbits
112	2^7	$28 \cdot 2^7 = 3584$	$12 \cdot 2^7 = 1536$	$12 \cdot 2^{11}$ bits = 24 Kbits
128	2^7	$32 \cdot 2^7 = 4096$	$16 \cdot 2^7 = 2048$	$16 \cdot 2^{11}$ bits = 32 Kbits
192	2^8	$28 \cdot 2^8 = 7168$	$12 \cdot 2^8 = 3072$	$12 \cdot 2^{12}$ bits = 48 Kbits
256	2^8	$32 \cdot 2^8 = 8192$	$16 \cdot 2^8 = 4096$	$16 \cdot 2^{12}$ bits = 64 Kbits

Table 4.1: Suggested parameters for McEliece variants based on quasi-dyadic Goppa codes over $\mathbb{F}_2$

As the public generator matrix $\hat{G}$ is in systematic form, only its non-trivial part Q of length $n - k = m \cdot t$ have to be stored. This part consists of $m(l - m)$ dyadic submatrices of size $t \times t$ each. Storing the t-length signatures of Q only the resulting public key size is $m(l - m)t = m \cdot k$ bits in size. Hence, the public key size is a factor of t smaller compared to the generic McEliece version where the key is $n \cdot k$ bits in size.

4.4 Security of QD-McEliece

A recent work [FOPT09] presents an efficient attack recovering the private key in specific instances of the quasi-dyadic McEliece variant. Due to the structure of a quasi-dyadic Goppa code additional linear equations can be constructed. These equations reduce the algebraic complexity of solving a multidimensional system of equations using Gröbner bases [AL94]. In the case of the quasi-dyadic McEliece variant there are $l - m$ linear equations and $l - 1$ unknowns Y_i. The dimension of the vector space solution for the $Y_i's$ is $m - 1$. Once the unknowns Y_i are found all other unknowns X_i can be obtained by solving a system of linear equations. In our case there are 35 unknowns Y_i, 20 linear equations, and the dimension of the vector space solution for the $Y_i's$ is 15.

The authors remark that the solution space is manageable in practice as long as $m < 16$. The attack was not successful when $m = 16$. Hence, up to day the McEliece variant using subfield subcodes over the base field of large codes over $\mathbb{F}_{2^{16}}$ is still secure.

5 Conversions for CCA2-secure McEliece variants

In [KI01] Kobara and Imai considered some conversions for achieving security against the critical attacks discussed in Section 2.4, and thus CCA2-security, in a restricted class of public-key cryptosystems. The authors reviewed these conversions for applicability to the McEliece public key cryptosystem and showed two of them to be convenient. These are Pointcheval's generic conversion [Poi00] and Fujisaki-Okamoto's generic conversion [FO99]. Both convert partially trapdoor one-way functions (PTOWF) [1] to public key cryptosystems fulfilling the CCA2 indistinguishability.

The main disadvantage of both conversions is their high redundancy of data. Hence, Kobara and Imai developed three further specific conversions decreasing data overhead of the generic conversions even below the values of the original McEliece PKCs for large parameters.

Conversion scheme	**Data redundancy = ciphertext size - plaintext size**		
	(n,k),t,r	(2304,1280),64,160	(2304,1280),64,256
Pointcheval's generic conv.	$n + \|r\|$	2464	2560
Fujisaki-Okamoto's generic conversion	n	2304	2304
Kobara-Imai's specific conv. α and β	$n + \|r\| - k$	1184	1280
Kobara-Imai's specific conversion γ	$n + \|r\| + \|Const\| - \lfloor \log_2 \binom{n}{t} \rfloor - k$	927	1023
McEliece scheme w/o conv.	$n - k$	1024	1024

Table 5.1: Comparison between conversions and their data redundancy

Table 5.1 gives a comparison between the conversions mentioned above and their data overhead where r denotes a random value of typical length $|r|$ equal to the output length of usual hash functions, e.g. SHA-1, SHA-256, and $Const$ denotes a predetermined public constant of suggested length $|Const|$=160 bits. In addition,

[1] A PTOWF is a function $F(x, y) \rightarrow z$ for which no polynomial time algorithm exists recovering x or y from their image z alone, but the knowledge of a secret enables a partial inversion, i.e., finding x from z.

the data redundancy of the original McEliece system is given. The table shows clearly that the Kobara-Imai's specific conversion γ (KIC-γ) provides the lowest data redundancy for large parameters n and k. In particular, for parameters $n = 2304$ and $k = 1280$ used in this thesis for the construction of the quasi-dyadic McEliece-type PKC the data redundancy of the converted variant is even below that of the original scheme without conversion.

In the next section the scheme definition of the Kobara-Imai's specific conversion γ is given, followed by the description of a constant weight coding technique used for the implementation of the KIC-γ in this thesis.

5.1 Kobara-Imai's specific conversion γ

We first define some notations used in the scheme definition of the KIC-γ.

$\|x\|$	Bit length of x
$MSB_y(x)$	The y rightmost bits of x
$LSB_y(x)$	The y leftmost bits of x
$Const$	Predetermined public constant
$\texttt{Prep}(m)$	Preparation of the message m, e.g. padding, data compression. $\texttt{Prep}(\tilde{m})^{-1}$ denotes the inverse function of $\texttt{Prep}(m)$.
`Rand`	Source generating a (pseudo) random sequence of fixed length
$\texttt{Gen}(x)$	PRNG which produces cryptographically secure pseudo-random sequences of arbitrary length from a fixed length seed x.
$\texttt{Hash}(x)$	One-way hash function mapping an arbitrary length binary string x to a fixed length binary string.
$\texttt{Conv}(x)$	Bijective conversion function mapping a message x to the corresponding error vector e of fixed length n and Hamming weight $wt(e) \leq t$. $\texttt{Conv}(y)^{-1}$ denotes the inverse function of $\texttt{Conv}(x)$.
$\mathcal{E}_{MCE}(x, e)$	McEliece encryption function, taking as first argument the message x to be encrypted and as second one the error vector e and outputting the corresponding ciphertext.
$\mathcal{D}_{MCE}(c)$	McEliece decryption function, taking the ciphertext c and outputting the decrypted message and the error vector.

5.1.1 Encryption

Algorithm 5.1.1 encrypts a message m to the ciphertext c. The message m should be of length $|m| \geq \left\lfloor \log_2 \binom{n}{t} \right\rfloor + k - |r| - |Const|$ to decrease data overhead. Otherwise, the preparation function is called first to bring m in desired form.

Algorithm 5.1.1 KIC-γ Encryption

Input: message m, predetermined public constant $Const$
Output: the target ciphertext c

1. $\tilde{m} \leftarrow \texttt{Prep}(m)$
2. $r \leftarrow \texttt{Rand}$
3. $y_1 \leftarrow \texttt{Gen}(r) \oplus (\tilde{m}||Const)$
4. $y_2 \leftarrow r \oplus \texttt{Hash}(y_1)$
5. **if** $|\tilde{m}| + |Const| + |r| > \left\lfloor \log_2 \binom{n}{t} \right\rfloor + k$ **then**
6. $(y_5||y_4||y_3) = (y_2||y_1)$ where $|y_3| = k$, $|y_4| = \left\lfloor \log_2 \binom{n}{t} \right\rfloor$, $|y_5| = |\tilde{m}| + |Const| + |r| - |y_4| - k$
7. $e \leftarrow \texttt{Conv}(y_4)$
8. $c \leftarrow y_5||\mathcal{E}_{MCE}(y_3, e)$
9. **else**
10. $(y_4||y_3) = (y_2||y_1)$ where $|y_3| = k$, $|y_4| = \left\lfloor \log_2 \binom{n}{t} \right\rfloor$
11. $e \leftarrow \texttt{Conv}(y_4)$
12. $c \leftarrow \mathcal{E}_{MCE}(y_3, e)$
13. **return** c

In [NIKM08] it has been proven that padding the plaintext with a random bit-string provides semantic security against chosen plaintext attack for the McEliece-type cryptosystems under standard assumptions. This is based on the observation that if some fixed part of the plaintext is made random then the ciphertext is pseudo random from the attacker's point of view due to the construction of the McEliece cryptosystem. Hence, the first step of the encryption algorithm is randomization of the (possibly) prepared message $\tilde{m}$ padded with a predetermined public constant $Const$. Then the hash value of the randomized message is xored with a random value. The aim of the conversion γ is the reduction of the overhead data. Hence, both the error vector and the "'plaintext"' for the McEliece encryption are taken from $(y_1||y_2)$. When $|r|+|Const| < \left\lfloor \log_2 \binom{n}{t} \right\rfloor$ this reduces the data overhead even below that of the original McEliece scheme without conversion. A k-size part of $(y_1||y_2)$ can be taken as message for McEliece encryption without any changes. The error vector is obtained from another part of $(y_1||y_2)$ by calling a constant weight encoding function $\texttt{Conv}(\cdot)$ which transforms an input string of length about $\left\lfloor \log_2 \binom{n}{t} \right\rfloor$ into a vector of fixed length n and Hamming weight t. A possible method for the realization of the $\texttt{Conv}(\cdot)$ function is discussed in

Section 5.2. If the length of $(y_1||y_2)$ is greater than $\left\lfloor \log_2 \binom{n}{t} \right\rfloor + k$ such that not all input bits are used as plaintext or as error vector all remaining bits are used as padding for the output of $\mathcal{E}_{MCE}$.

5.1.2 Decryption

Algorithm 5.1.2 describes the decryption of a ciphertext c.

Algorithm 5.1.2 KIC-γ Decryption

Input: ciphertext c, predetermined public constant $Const$
Output: the target message m

1. $y_5 \leftarrow MSB_{|c|-n}(c)$
   $\triangleright$ (y_5 may be empty)
2. $(y_3, e) \leftarrow \mathcal{D}_{McEliece}(LSB_n(c))$
3. $y_4 \leftarrow \mathtt{Conv}^{-1}(e)$
4. $(y_2||y_1) = (y_5||y_4||y_3)$ where $|y_1| = |\mathtt{Hash}(\cdot)|$, $|y_2| = |r|$
5. $r \leftarrow y_2 \oplus \mathtt{Hash}(y_1)$
6. $(\tilde{m}||Const') = y_1 \oplus \mathtt{Gen}(r)$
7. **if** $Const' = Const$ **then**
8. **return** $\mathtt{Prep}^{-1}(\tilde{m})$
9. **else**
10. **reject** c

If the ciphertext length is larger than n then c has been padded during the encryption and the $n - |c|$ most significant bits of c correspond to the most significant bits of $(y_2||y_1)$. The remaining n bits of c are then decrypted using the decryption function for the McEliece scheme to a plaintext y_3 and an error vector e. To obtain the message part y_4 from the error e the constant weight decoding function $\mathtt{Conv}(\cdot)^{-1}$ is called.

The remainder of the decryption algorithm undoes the KIC-γ encryption because

$$\begin{aligned} y_1 \oplus \mathtt{Gen}(y_2 \oplus \mathtt{Hash}(y_1)) &= y_1 \oplus ((r \oplus \mathtt{Hash}(y_1)) \oplus \mathtt{Hash}(y_1)) \\ &= y_1 \oplus Gen(r) \\ &= (\mathtt{Gen}(r) \oplus (\tilde{m}||Const)) \oplus \mathtt{Gen}(r) \\ &= \tilde{m}||Const \ \square \end{aligned}$$

Hence, if the decrypted constant $Const'$ is the same as the predetermined constant $Const$ the decryption algorithm returns the message m, after reversing the preparation, if necessary.

5.2 Constant weight coding

In [Sen05] Sendrier proposed an efficient method for encoding information into constant weight words. We use this proposal to implement the $\texttt{Conv}(\cdot)$ function for the KIC-γ. In the following, the essentials of the Sendrier's proposal are given.

Let $W_{n,t}$ denote a set of words of fixed length n and Hamming weight at most t. The aim of the proposal is to construct a function which takes as input any binary string B, and maps it to a sequence of words in $W_{n,t}$ where $W_{n,t}$ is equipped with uniform distribution. This way a message x can be converted into a vector e of fixed length n and weight at most t which can be used as error vector for McEliece encryption.

The proposal is based on Golombs run-length coding [Gol66] which is a form of lossless data compression for a memoryless binary source with highly unbalanced probability law, e.g. such that $p = Prob(0) \geq 1/2$. The run-length encoding routine `encodefd` shown in Algorithm 5.2.1 encodes a source sequence $0^{\delta_1}10^{\delta_2}10\cdots010^{\delta_s}$ by $\texttt{encodefd}(\delta_1)||\cdots||\texttt{encodefd}(\delta_s)$ where δ_is denote the distances between ones in the sequence. The resulting binary stream B is very similar to that produced by a memoryless binary source with uniform distribution. The value d denotes the length of the longest expectable string of consecutive zeros in the source sequence. As the probability of any sequence of zeros to have the length $\geq d$ is $(1-p)^d$ Golomb suggests choosing $d \approx -1/\log_2(1-p)$ in order that $(1-p)^d \approx 1/2$.

Algorithm 5.2.1 Run-length coding: function `encodefd`

Input: two integers δ and d
Output: a binary stream B

1. $u \leftarrow \lceil \log_2(d) \rceil$
2. **if** $\delta < 2^u - d$ **then**
3. $\quad u \leftarrow u - 1$
4. **else**
5. $\quad \delta \leftarrow \delta + (2^u - d)$
6. **return** $LSB_u(\delta|_{\mathbb{F}_2})$

Conversely, the run-length decoding routine `decodefd` shown in Algorithm 5.2.2 transforms a binary stream B into a sequence $(\delta_1, \ldots, \delta_s)$ from which the original binary stream with long runs of zeros is obtained.

The function $\texttt{read}(B, x)$ moves forward and reads x bits in the binary stream B and returns the integer whose binary decomposition has been read.

For the construction of a function, taking any binary string B as input and outputting an element of the uniform binary source $W_{n,t}$, Sendrier makes an extensive use of the fact [Cov73] that every element $e \in W_{n,t}$, represented as an

Algorithm 5.2.2 Run-length coding: function `decodefd`

Input: a binary stream B, an integer d
Output: an integer δ

1. $u \leftarrow \lceil \log_2(d) \rceil$
2. $\delta \leftarrow \texttt{read}(B, u-1)$
3. **if** $\delta \geq 2^u - d$ **then**
4. $\quad \delta \leftarrow 2\delta + \texttt{read}(B, 1) - (2^u - d)$
5. **return** δ

array of integers $(i_1, \ldots, i_t)$ where i_j are coordinates of ones in e, can be written with about $l \leq \left\lceil \log_2 \binom{n}{t} \right\rceil$ bits, in average, by using the run-length encoding function with $d = \binom{n}{t}$. The distances between ones in e are given by $\delta_1 = i_1 - 1$, $\delta_2 = i_2 - i_1 - 1$, ..., $\delta_t = i_t - i_{t-1} - 1$. Hence, the run-length decoding function can be used to transform an arbitrary binary stream B produced by a memoryless binary source into a word of fix length n and hamming weight t in $W_{n,t}$.

In the following, the functions `CWtoB` 5.2.3 and `BtoCW` 5.2.4 are given, converting an element $e \in W_{n,t}$ into a binary string and converting a binary string of length about $\left\lfloor \log_2 \binom{n}{t} \right\rfloor$ into an element $e \in W_{n,t}$ of fixed length and weight, respectively. However, it is very expensive to compute the binomial coefficients for $d = \binom{n}{t}$ on-the-fly as suggested in [Cov73]. On the other hand, the precomputation of these values for d requires a memory quadratic in $\left\lceil \log_2 \binom{n}{t} \right\rceil$. Hence, in Sendrier's proposal the value for d changes depending on the parameters n and t in every recursive call of the functions `CWtoB` and `BtoCW`. The function `best_d` determines an optimal d such that $d \approx \left(n - \frac{t-1}{2}\right)\left(1 - \frac{1}{2^{1/t}}\right) \approx \frac{\ln(2)}{t}\left(n - \frac{t-1}{2}\right)$. Both functions terminate even if d has not been chosen optimal, with only negligible loss of efficiency, as long as the difference to the optimal value is not large.

Algorithm 5.2.3 Constant weight coding: function `CWtoB`

Input: two integers n and t, and a t-tuple $(\delta_1, \ldots, \delta_t)$
Output: a binary string B

1. **if** $t = 0$ or $n \leq t$ **then**
2. $\quad$ **return**
3. $d \leftarrow \texttt{best\_d}(n, t)$
4. **if** $\delta_1 \geq d$ **then**
5. $\quad$ **return** $\texttt{1}||\texttt{CWtoB}(n-d, t, (\delta_1 - d, \delta_2, \ldots, \delta_t))$
6. **else**
7. $\quad s \leftarrow \texttt{0}||\texttt{encodefd}(\delta_1, d)$
8. $\quad$ **return** $\texttt{s}||\texttt{CWtoB}(n - \delta_1 - 1, t-1, (\delta_2, \ldots, \delta_t))$

The first call of `BtoCW` is $\texttt{BtoCW}(n, t, 0, B)$.

Algorithm 5.2.4 Constant weight coding: function `BtoCW`

Input: three integers n, t, and δ, a binary stream B
Output: a t-tuple of integers $(\delta_1, \ldots, \delta_t)$

1. **if** $t = 0$ **then**
2. **return**
3. **else if** $n \leq t$ **then**
4. **return** δ, `BtoCW`$(n-1, t-1, 0, B)$
5. **else**
6. $d \leftarrow$ `best_d`(n, t)
7. **if** `read`(B,1)=1 **then**
8. **return** `BtoCW`$(n-d, t, \delta+d, B)$
9. **else**
10. $i \leftarrow$ `decodefd`(d, B)
11. **return** $\delta + i$, `BtoCW`$(n-i-1, t-1, 0, B)$

The main disadvantage of this proposal is that the length of the input string for the conversion function `BtoCW` is not fix. The number of bits of the binary string B which can be "put" in a word of $W_{n,t}$ is upper bounded by the entropy of $W_{n,t}$ equipped with uniform distribution $H(W_{n,t}) = \log_2 \binom{n}{t}$. Though, it is possible to determine the minimum length of the input string that can be encoded in any case, experimentally.

6 Implementation aspects

In this chapter we discuss aspects of our implementation of the McEliece variant based on quasi-dyadic Goppa codes of length $n = 2304$, dimension $k = 1280$, and correctable number of errors $t = 64$ over the subfield $\mathbb{F}_2$ of $\mathbb{F}_{2^{16}}$ providing 80-bit security level. Target platform is the ATxmega256A1, a RISC microcontroller frequently used in embedded systems. This microcontroller operates at a clock frequency up to 32 MHz, provides 16 Kbytes SRAM and 256 Kbytes Flash memory. We first implemented the quasi-dyadic McEliece variant on a PC using Notepad++ v5.5.1 and the GCC compiler version v3.4.5. Then we ported the implementation to the AVR micro architecture.

This chapter is organized as follows. Section 6.1 describes how field arithmetic can be implemented on embedded microcontrollers. Section 6.2 presents our implementation of the quasi-dyadic McEliece variant. At the beginning of this section we present the key generation algorithm implemented in the Magma computer algebra system. Then we describe the encryption and the decryption algorithms. Section 6.3 describes the implementation of the Kobara-Imai's specific conversion γ and its application to the quasi-dyadic McEliece variant.

6.1 Field arithmetic

Let $\mathbb{F}_q$ denote the finite field $\mathbb{F}_{2^m} \cong \mathbb{F}_2[x]/p(x)$ where $p(x)$ is an irreducible polynomial of degree m over $\mathbb{F}_2$. Furthermore, let α denote a primitive element of $\mathbb{F}_q$.

Every element $a \in \mathbb{F}_q$ has a polynomial representation $a(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0 \mod p(x)$ where $a_i \in \mathbb{F}_2$. The addition of two field elements a and b is done using their polynomial representations such that $a + b = a(x) + b(x) \mod p(x) \equiv c(x)$ with $c_i = a_i \oplus b_i, \forall i \in \{0, \ldots, m-1\}$. The field addition can be implemented efficiently by performing the exclusive or operation of two unsigned m-bits values. For simplicity, the coefficient a_0 should be stored in the least significant bit and a_{m-1} in the most significant bit of an unsigned m-bits value.

Furthermore, any element $a \in \mathbb{F}_q$ except the zero element can be represented as a power of a primitive element $\alpha \in \mathbb{F}_q$ such that $a = \alpha^i$ where $i \in \mathbb{Z}_{2^m-1}$. The exponential representation allows to perform more complex operations such

as multiplication, division, squaring, inversion, and square root extraction more efficiently than using polynomial representation.

The field multiplication of two field elements $a = \alpha^i$ and $b = \alpha^j$ is easily performed by addition of both exponents i and j such that

$$a \cdot b = \alpha^i \cdot \alpha^j \equiv \alpha^{i+j \mod 2^m-1} \equiv c,\ c \in \mathbb{F}_q\,.$$

Analogously, the division of two elements a and b is carried out by subtracting their exponents such that

$$\frac{a}{b} = \frac{\alpha^i}{\alpha^j} \equiv \alpha^{i-j \mod 2^m-1} \equiv c,\ c \in \mathbb{F}_q$$

The squaring of an element $a = \alpha^i$ is done by doubling its exponent and can be implemented by one left shift.

$$a^2 = (\alpha^i)^2 \equiv \alpha^{2i \mod 2^m-1}$$

Analogously, the inversion of a is the negation of its exponent.

$$a^{-1} = (\alpha^i)^{-1} \equiv \alpha^{-i \mod 2^m-1}$$

The square root extraction of an element $a = \alpha^i$ is performed in the following manner.

- If the exponent i of a is even, then $\sqrt{a} = (\alpha^i)^{\frac{1}{2}} \equiv \alpha^{\frac{i}{2} \mod 2^m-1}$.
- If the exponent i of a is odd, then $\sqrt{a} = (\alpha^i)^{\frac{1}{2}} \equiv \alpha^{\frac{i+2^m-1}{2} \mod 2^m-1}$.

If the exponent of a is even the square root extraction can be implemented by one right shift. If the exponent is odd, it is possible to extend it by the modulus 2^m-1, which leads to an even value. Then the square root extraction is performed as before through shifting the exponent to the right for one time.

To implement the field arithmetic on an embedded microcontroller most efficiently both representations of the field elements of $\mathbb{F}_q$, polynomial and exponential, should be precomputed and stored as *log-* and *antilog* table, respectively. Each table occupies $m \cdot 2^m$ bits storage.

Unfortunately, we cannot store the whole log- and antilog tables for $\mathbb{F}_{2^{16}}$ because each table is 128 Kbytes in size. Neither the SRAM memory of the ATXmega256A1 (16 Kbytes) nor the Flash memory (256 Kbytes) would be enough to implement the McEliece PKC when completely storing both tables. Hence, we make use of *tower field arithmetic*. Efficient algorithms for arithmetic over tower fields were proposed in [Afa91], [MK89], and [Paa94].

It is possible to view the field $\mathbb{F}_{2^{2k}}$ as a field extension of degree 2 over $\mathbb{F}_{2^k}$ where $k = 1, 2, 3$. The idea is to perform field arithmetic over $\mathbb{F}_{2^{2k}}$ in terms of operations

in a subfield $\mathbb{F}_{2^k}$. Thus, we can consider the finite field $\mathbb{F}_{2^{16}} = \mathbb{F}_{(2^8)^2}$ as a tower of $\mathbb{F}_{2^8}$ constructed by an irreducible polynomial $p(x) = x^2 + x + p_0$ where $p_0 \in \mathbb{F}_{2^8}$. If β is a root of $p(x)$ in $\mathbb{F}_{2^{16}}$ then $\mathbb{F}_{2^{16}}$ can be represented as a two dimensional vector space over $\mathbb{F}_{2^8}$ and an element $A \in \mathbb{F}_{2^{16}}$ can be written as $A = a_1\beta + a_0$ where $a_1, a_0 \in \mathbb{F}_{2^8}$. To perform field arithmetic over $\mathbb{F}_{2^{16}}$ we store the log- and antilog tables for $\mathbb{F}_{2^8}$ and use them for fast mapping between exponential and polynomial representations of elements of $\mathbb{F}_{2^8}$. Each table occupies only 256 bytes, therefore both tables can smoothly be copied into the fast SRAM memory of the microcontroller at startup time.

The field addition of two elements A and B in $\mathbb{F}_{2^{16}}$ is then performed through

$$A + B = (a_1\beta + a_0) + (b_1\beta + b_0) = (a_1 + b_1)\beta + (a_0 + b_0) = c_1\beta + c_0$$

and involves two field additions over $\mathbb{F}_{2^8}$ which is equal to two xor-operations of 8-bits values.

The field multiplication of two elements $A, B \in \mathbb{F}_{2^{16}}$ is carried out through

$$A \cdot B = (a_1\beta + a_0)(b_1\beta + b_0) \mod p(x) \equiv (a_0b_1 + b_0a_1 + a_1b_1)\beta + (a_0b_0 + a_1b_1p_0).$$

and involves three additions and five multiplications over $\mathbb{F}_{2^8}$ when reusing the value a_1b_1 which already has been computed in the β-term.

The squaring is a simplified version of the multiplication of an element A by itself in a finite field of characteristic 2, and is performed as follows

$$A^2 = (a_1\beta + a_0)^2 \mod p(x) \equiv a_0^2\beta^2 + a_1^2 \mod p(x) \equiv a_0^2\beta + (a_1^2 + p_0).$$

One squaring over $\mathbb{F}_{2^{16}}$ involves two square operations and one addition over $\mathbb{F}_{2^8}$.

The field inversion is more complicated compared to the operations described above. An efficient method for inversion in tower fields of characteristic 2 is presented in [Paa94]. The inversion of an element A is performed through

$$A^{-1} = \left(\frac{a_1}{\Delta}\right)\beta + \frac{a_0 + a_1}{\Delta} = c_1\beta + c_0 \text{ where } \Delta = a_0(a_1 + a_0) + p_0a_1^2$$

and involves two additions, two divisions, one squaring, and two multiplications over $\mathbb{F}_{2^8}$, when reusing the value $(a_0 + a_1)$.

The division of two elements $A, B \in \mathbb{F}_{2^{16}}$ can be performed through multiplication of A by the inverse B^{-1} of B. This approach requires five additions, seven multiplications, two divisions, and one squaring over $\mathbb{F}_{2^8}$. To enhance the performance of the division operation we provide a slightly better method given below.

$$A/B = A \cdot B^{-1} = \left(\frac{a_0(b_0 + b_1) + a_1b_1p_0}{\Delta}\right)\beta + \left(\frac{a_0b_1 + a_1b_0}{\Delta}\right) \text{ where } \Delta = b_0(b_1 + b_0) + p_0b_1^2$$

This method involves one less addition compared to the naive approach mentioned above.

The last operation we need for the implementation of the McEliece scheme is the extraction of square roots. We could not find any formula for square root extraction over tower fields in the literature, therefore, we developed an own one for this purpose. For any element $A \in \mathbb{F}_{2^{16}}$ there exists a *unique* square root, as the field characteristic is 2. Hence, the following holds for the square root of A

$$\begin{aligned}\sqrt{A} &= \sqrt{a_1\beta + a_0} \equiv \sqrt{a_1}\sqrt{\beta} + \sqrt{a_0} \mod p(x) \\ &\equiv \sqrt{a_1}(\beta + \sqrt{p_0}) + \sqrt{a_0} \mod p(x) \\ &= \sqrt{a_1}\beta + (\sqrt{a_1}\sqrt{p_0} + \sqrt{a_0}).\end{aligned}$$

Proof:

As $\mathrm{Char}(\mathbb{F}_{2^{16}})$ is 2,

$$\sqrt{A} = \sqrt{a_1\beta + a_0} \equiv (a_1\beta + a_0)^{2^7 \mod 2^8-1} \equiv a_1^{2^7}\beta^{2^7} + a_0^{2^7} \qquad (6.1)$$

For any element y in $\mathbb{F}_{2^8}$ the trace function is defined by $Tr(y) = \sum_{i=0}^{7} y^{2^i} \equiv \begin{cases}1\\0\end{cases}$

Furthermore, β satisfies $\beta^2 \equiv \beta + p_0$, as β is root of $p(x)$. Hence, we can write

$$\beta^{2^7} = (\beta^2)^{2^6} \equiv (\beta + p_0)^{2^6} \equiv (\beta + p_0^2 + p_0)^{2^5} \equiv \cdots \equiv \beta + \sum_{i=1}^{6} p_0^{2^i} \equiv$$

$$\beta + \sum_{i=0}^{7} p_0^{2^i} + 1 + p_0^{2^7} \equiv \begin{cases}\beta + p_0^{2^7} & \text{, if } Tr(p_0) = 1\\ \beta + 1 + p_0^{2^7} & \text{, if } Tr(p_0) = 0\end{cases}$$

We assume that $Tr(p_0) = 1$. Otherwise, the polynomial $p(x)$ would not be irreducible, and thus, unsuited for the field construction.

Applying the intermediate results to the Equation 6.1 we obtain

$$\begin{aligned}\sqrt{A} &\equiv a_1^{2^7 \mod 2^8-1}(\beta + p_0^{2^7 \mod 2^8-1}) + a_0^{2^7 \mod 2^8-1} \\ &\equiv a_1^{2^{-1} \mod 2^8-1}\beta + a_1^{2^{-1} \mod 2^8-1}p_0^{2^{-1} \mod 2^8-1} + a_0^{2^{-1} \mod 2^8-1} \\ &\equiv \sqrt{a_1}\beta + \sqrt{a_1}\sqrt{p_0} + \sqrt{a_0}\end{aligned}$$

The next question is how to realize the mapping $\varphi\colon\ A \to (a_1, a_0)$ of an element $A \in \mathbb{F}_{2^{16}}$ to two elements $(a_1, a_0) \in \mathbb{F}_{2^8}$, and the inverse mapping $\varphi^{-1}\colon\ a_1, a_0 \to A$ such that $A = a_1\beta + a_0$. Both mappings can be implemented by means of a special transformation matrix and its inverse, respectively [Paa94].

As the input and output for the McEliece scheme are binary vectors, field elements are only used in the scheme internally. Hence, we made an informed choice against the implementation of both mappings. Instead, we represent each field element A of $\mathbb{F}_{2^{16}}$ as a structure of two uint8_t values describing the elements of $\mathbb{F}_{2^8}$ and perform all operations on these elements directly.

```
typedef struct {
   uint8_t highByte;
   uint8_t lowByte;
}gf16_t;
```

An element A of type `gf16_t` is defined by `gf16_t A={A.highByte,A.lowByte}`. The tower field arithmetic can be performed through direct access to the elements a_1 = `A.highByte` and a_0 = `A.lowByte`. The specific operations over $\mathbb{F}_{2^8}$ are carried out through lookups in the precomputed log- and antilog tables for this field. The result of an arithmetic operation is an element of type `gf16_t` again.

Polynomials over $\mathbb{F}_{2^{16}}$ are represented as arrays. For instance, we represent a polynomial $G(x) = G_t x^t + \cdots + G_1 x + G_0$ as an array of type `gf16_t` and size $t+1$ and store the coefficients G_i of $G(x)$ such that `array[i].highByte` $= G_{i,1}$ and `array[i].lowByte` $= G_{i,0}$ where $\varphi(G_i) = (G_{i,1}, G_{i,1})$.

The main problem when generating log- and antilog tables for a finite field is that there exist no exponential representation of the zero element, and thus, no explicit mapping $0 \rightarrow i$ such that $0 \equiv \alpha^i$, and vice versa. Hence, additional steps have to be performed within the functions for specific arithmetic operations to realize a correct zero-mapping. These additional computation steps reduce the performance of the tower field arithmetic but there is no way to avoid them.

In the following, a Magma setup routine is given generating the field $\mathbb{F}_{2^{16}}$ as a tower of $\mathbb{F}_{2^8}$ using the irreducible polynomial $p(x) = x^2 + x + \beta^{11}$. The polynomial $p(x)$ is obtained through testing polynomials of the form $x^2 + x + \beta^i$ for irreducibility.

Listing 6.1: Tower field arithmetic: Field setup

```
poly1<y> := PrimitivePolynomial(GF(2),8);
F<b> := ExtensionField<GF(2),y|poly1>;
P<x> := PolynomialRing(F);
poly2<x> := x^2 + x + b^{11};
GFq<a> := ExtensionField<F,x|poly2>;
PR<z> := PolynomialRing(GFq);
```

To save space, we give a simplified Magma procedure for the generation and printing of the log- and antilog tables for $\mathbb{F}_{2^8}$. The procedure outputs unformatted tables and stores them in a file denoted by the file object `FP` where `FP = Open("filename","w")`. The original procedure implemented in this thesis generates full formated tables. The zero element has a (mathematically incor-

rect) exponential representation α^{255}. Thus, the value 255 is stored in the antilog table at the position 0.

Listing 6.2: Printing the log- and antilog tables for $\mathbb{F}_{2^8}$

```
procedure print_tables(field,FP)
  //Print log table
  logtable :=[Seqint(ChangeUniverse(Eltseq(b^i,GF(2)),Integers()),2):
              i in [0..#F-2]];
  Append(~logtable,0);
  fprintf FP, "%o,", logtable;

  //Print antilog table
  Sort(~logtable);
  for x in logtable do
      if x eq 0 then
          fprintf FP, "%o,", #field-1;
      else
          obj:=Log(Seqelt(ChangeUniverse(Intseq(x,2,8),GF(2)),F));
          fprintf FP, "%o,", obj;
      end if;
  end for;
end procedure;
```

6.2 Implementation of the QD-McEliece variant

6.2.1 Key generation

The Key generation algorithm 4.2.1 for the quasi-dyadic McEliece variant including Algorithm 3.5.1 for the generation of dyadic Goppa codes has been implemented using the Magma computer algebra system. It takes as input the common system parameters $d = 16$, $t = 64$, $l = 36$, $q = 2^d$, $N = 2^{d-1}$, $n = l \cdot t$, $k = n - d \cdot t$ $c = n - k$ and outputs the following parameters: systematic quasi-dyadic generator matrix for the public code as well as its essential part used as public key, secret permutation P, secret Goppa polynomial $G(x)$, permuted support sequence L^* for the Goppa code $\Gamma(L^*, G(x))$. It also precomputes a parity-check matrix `H123` and a scrambling matrix S which can be used to speed up the syndrome computation within the Patterson's decoding algorithm (see Section 3.7.1).

The first steps of the key generation algorithm compute the signature h of a fully dyadic matrix $H = \Delta(h, t)$ of length N and dimension t.

Listing 6.3: QD-McEliece: Key generation algorithm (Magma)

```
nuSize := Ilog2(N);
U := [x : x in GFq];
Remove(~U, Index(U, Zero(GFq)));
h := [GFq|];
nu := [GFq|];

h[1] := Random(U);
nu[nuSize+1] := h[1]^-1;
Remove(~U, Index(U, h[1]));

for s := 0 to nuSize-1 do
    i := 2^s;
    h[i+1] := Random(U);
    nu[s+1] := h[i+1]^-1 + nu[nuSize+1];
    Remove(~U, Index(U, h[i+1]));
    Remove(~U, Index(U, nu[s+1]^-1));
    for j := 1 to i-1 do
        h[i+1+j] := (nu[s+1] + h[j+1]^-1)^-1;
        Remove(~U, Index(U, h[i+1+j]));
        Remove(~U, Index(U, (h[i+1+j]^-1 + nu[nuSize+1])^-1));
    end for;
end for;
```

Then an offset is chosen at random to generate roots of a Goppa polynomial depending on the first t values of h. The Goppa polynomial is obtained by $\prod_{i=1}^{t}(z - roots[i])$.

```
offset := Random(GFq);
roots:=[GFq|h[i]^-1+offset:i in [1..t]];
goppa := One(PR);
for i := 1 to t do
    goppa := goppa*(z-roots[i]);
end for;
```

In the next steps of the key generation algorithm the support L and the parity-check matrix H for the large code C_{dyad} are computed.

```
L:=[GFq|h[j]^-1 + nu[nuSize+1] + offset: j in [1..N]];
H := Zero(KMatrixSpace(GFq, t, N));
seq:=[GFq|h[xor_Integer(i,j)+1]: j in [0..N-1], i in [0..t-1]];
H:=KMatrixSpace(GFq, t, N)!seq;
```

To obtain a permuted binary systematic quasi-dyadic parity-check matrix for the public code the key generation algorithm first choses a block permutation sequence `B_ind` of length n consisting of *distinct* integers $i \in [1, N/t]$. Next, a sequence `P_ind` of integers $j \in [1, t]$ is chosen identifying the positions of ones

in the signatures of dyadic permutations. It is not necessary to choose a random non-zero sequence of scale factors because we generate Goppa codes over the base field and all scale factors have to be 1. Then it holds $\Sigma = I_n$ and $Tr'(H)\Sigma = Tr'(H)$. From `B_ind` and `P_ind` two permutation matrices `PB` and `PD`, respectively, are obtained in the next step of the key generation algorithm. The function `dyadic_Permutation` takes as input an integer x identifying the position of 1 in the signature of a dyadic permutation as well as the target field and generates the corresponding dyadic permutation matrix of size $t \times t$. Through multiplication of both permutation matrices we obtain another permutation matrix P which is used to puncture and to permute H.

From the resulting matrix `pubH` the function `GFq2Bin` derives a binary quasi-dyadic parity-check matrix `pubHbin` for the subfield subcode. The function `get_systematic_matrices` brings `pubHbin`$= [LS|R]$ in systematic form and outputs the systematic parity-check matrix `sysPubHbin`$= [R^{-1}LS|I_{n-k}]$ for the private code. For this reason, the $(n-k) \times (n-k)$ submatrix R of `pubHbin` must be non-singular. Should R be found singular another permutation must be chosen. Furthermore, the function `get_systematic_matrices` outputs the systematic quasi-dyadic binary generator matrix `sysPubGbin` for the public code. The signatures of the quasi-dyadic submatrices of `sysPubGbin` are stored in the array `essPart` which serves as public key for the McEliece encryption.

```
repeat
    //Get random block permutation sequence
    nbrBlocks := N div t;
    B_all := [Integers()|1 .. nbrBlocks];
    B_ind := [Integers()|];
    for i := 1 to l do
        elt := Random(B_all);
        Append(~B_ind, elt);
        Remove(~B_all, Index(B_all,elt));
    end for;

    //Get new dyadic permutation sequence
    P_all := [Integers()|1 .. t];
    P_ind := [Integers()|Random(P_all): i in [1..l]];

    //Get a permutation matrix P:=Pb*Pd
    PB :=Zero(KMatrixSpace(GFq,N,n));
    for i in [1..#B_ind] do
      InsertBlock(~PB,IdentityMatrix(GFq,t),
                  (B_ind[i]-1)*t+1,(i-1)*t+1);
    end for;
    PD := Zero(KMatrixSpace(GFq,n,n));
    i:=1;
    for x in P_ind do
        block:=dyadic_Permutation(x,GFq);
        InsertBlock(~PD,block,i,i);
        i:=i+t;
```

```
    end for;
    P:=PB*PD;

    //Get a permuted punctured dyadic parity-check matrix
    pubH:=H*P;

    //get a systematic quasi-dyadic public generator matrix
    pubHbin := GFq2BinMatrix(pubH);
    echMats := get_systematic_matrices(pubHbin);
until #echMats eq 4;     //pubH = [LS|R] where R is non-singular

    sysPubHbin := echMats[1];
    sysPubGbin := echMats[2];

    //Get the essential part of sysPubGbin serving as K_pub
    essPart := [GF(2)|];
    for i := 1 to Nrows(sysPubGbin) by t do
       essPart := essPart cat
       Eltseq(Submatrix(sysPubGbin,i,k+1,1,n-k));
    end for;
```

To obtain the permuted support sequence L^*=`L_new_perm` for the Goppa code $\Gamma(L^*, G(x))$ the key generation algorithm punctures the large support L using the permutation matrix P. When prepermuting the support sequence the permutation P is required no more for decoding. In this particular case the ciphertext does not have to be permuted before decoding.

```
    L_new_perm:=Eltseq((VectorSpace(GFq,N)!L)*P);
```

To speed up the syndrome computation we can precompute a private parity-check matrix `H123` obtained through multiplication of the three matrices `GoppaMat`, `VdmH`, and `Diag`. The first one is a lower triangular matrix consisting of the coefficients of the Goppa polynomial, the second one is the Vandermonde matrix $\texttt{vdm}(L^*, G(x))$, and the last one is the diagonal matrix $\texttt{Diag}(G(L_i^*)^{-1})$ (see Section 3.4). The binary representation `binH123` of `H123` can then be used for the syndrome computation within the Patterson's decoding algorithm.

```
    seq:=[GFq|L_new_perm[j]^(i-1): j in [1..n], i in [1..t]];
    VdmH := KMatrixSpace(GFq,t,n)!seq;

    seq:=[GFq|1/Evaluate(goppa,L_new_perm[i]): i in [1..n]];
    Diag := DiagonalMatrix(seq);

    seq:=[GFq|Coefficients(goppa)[t+1-i+j]:j in [1..i],i in [1..t]];
    GoppaMat:=LowerTriangularMatrix(seq);

    H123:=GoppaMat*VdmH*Diag;
    binH123:=GFq2BinMatrix(H123);
```

When bringing `binH123` in its systematic form using a scrambling matrix S the same systematic quasi-dyadic parity-check matrix is obtained as for the public code. If we want to use this parity-check matrix for decoding, the scrambling matrix should be precomputed and used as a part of the private key.

```
privmats := get_systematic_matrices(binH123);
S:=privmats[3];
```

6.2.2 Encryption

The first step of the McEliece encryption is codeword computation. This is performed through multiplication of a plaintext p by the public generator matrix $\hat{G}$ which serves as public key. In our case the public generator matrix $\hat{G} = [I_k|M]$ is systematic. Hence, the first k bits of the codeword are the plaintext itself, and only the submatrix M of $\hat{G}$ is used for the computation of the parity-check bits. $M \in (\mathbb{F}_2^{t\times t})^{d\times(l-d)}$ can be considered as a composition of $d \cdot (l-d)$ dyadic submatrices $\Delta(h_{xy})$ of size $t \times t$ each, represented by a signature h_{xy} of length t each. It also can be seen as a composition of $l-d$ dyadic matrices $\Delta(h_x, t)$ of size $dt \times t$ each, represented by a signature of length $dt = n-k$ each.

$$M := \left(\begin{array}{ccc}
\mathbf{m_{0,0}} & \cdots & \mathbf{m_{0,n\text{-}k\text{-}1}} \\
\vdots & \ddots & \vdots \\
m_{t-1,0} & \cdots & m_{t-1,n-k-1} \\
\hline
\mathbf{m_{t,0}} & \cdots & \mathbf{m_{t,n\text{-}k\text{-}1}} \\
\vdots & \ddots & \vdots \\
m_{2t-1,0} & \cdots & m_{2t-1,n-k-1} \\
\hline
\vdots & \ddots & \vdots \\
\hline
\mathbf{m_{(l\text{-}d\text{-}1)t,0}} & \cdots & \mathbf{m_{(l\text{-}d\text{-}1)t,n\text{-}k\text{-}1}} \\
\vdots & \ddots & \vdots \\
m_{(l-d)t-1,0} & \cdots & m_{(l-d)t-1,n-k-1}
\end{array}\right)
\begin{array}{l}
\left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\}\Delta(h_0,t) \\
\left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\}\Delta(h_1,t) \\
\\
\left.\vphantom{\begin{array}{c}a\\a\\a\end{array}}\right\}\Delta(h_{l-d},t)
\end{array}$$

In both cases the compressed representation of M serving as public key K_{pub} for the McEliece encryption is

$$K_{pub} = [(m_{0,0}, \cdots, m_{0,n-k-1}), \cdots, (m_{(l-d-1)t,0}, \cdots, m_{(l-d)t-1,n-k-1})].$$

The public key is $2.5 Kbytes$ in size and can be copied into the SRAM of the microcontroller at startup time for faster encryption.

The plaintext $p = (p_0, \cdots, p_{t-1}, p_t, \cdots, p_{2t-1}, \cdots, p_{(l-d-1)t}, \cdots, p_{(l-d)t-1})$ is a binary vector of length $k = 1280 = 20 \cdot 64 = (l-d)t$. Hence, the codeword computation is done by adding the rows of M corresponding to the non-zero bits of p. As we don't store M but just its compressed representation, only the bits p_{it}

for all $0 \leq i \leq (l-d-1)$ can be encrypted directly by adding the corresponding signatures. To encrypt all other bits of p the corresponding rows of M have to be reconstructed from K_{pub} first.

The components $h_{i,j}$ of a dyadic matrix $\Delta(h,t)$ are normally computed through $h_{i,j} = h_{i \oplus j}$ which is a simple reordering of the elements of the signature h. Unfortunately, we cannot use this equation directly because the public key is stored as an array of $(n-k)(l-d)/8$ elements of type `uint8_t`. Furthermore, for every $t = 64$ bits long substring of the plaintext a different length-$(n-k)$ signature has to be used for encryption.

In the following, we provide an efficient method for the codeword computation using a compressed public key.

Algorithm 6.2.1 QD-McEliece encryption: Codeword computation

Input: plaintext array p of type `uint8_t` and size $k/8$ bytes, public key K_{pub}
Output: codeword array cw of type `uint8_t` and size $n/8$ bytes

1. **INIT:** set the $k/8$ most significant bytes of cw to $MSB_{k/8}(cw) \leftarrow p$. Set the remaining bytes of cw to 0
2. **for** $j \leftarrow 0$ **to** $k/8 - 1$ **by** 8 **do**
3. Read 8 bytes = 64 bits of the plaintext
4. Determine the block key (signature of $\Delta(h_j, t)$)
5. **for** $i \leftarrow 0$ **to** 7 **do**
6. **for all** non-zero bits x of p[i] **do**
7. ▷ compute the $(i \cdot 8 + x)$-th row of $\Delta(h_j, t)$
   ▷ Bit permutations
8. **if** x is odd **then**
9. $r_y \leftarrow (h_{j,y}\texttt{\&0xAA})/2) | ((h_{j,y}\texttt{\&0x55}) \cdot 2)$, $\forall y \in \{0, \ldots, (n-k)/8\}$
10. **else**
11. $r \leftarrow h_j$
12. **if** x&0x02 **then**
13. $r_y \leftarrow ((r_y\texttt{\&0xCC})/4) | ((r_y\texttt{\&0x33}) \cdot 4)$, $\forall y \in \{0, \ldots, (n-k)/8\}$
14. **if** x&0x04 **then**
15. $r_y \leftarrow ((r_y\texttt{\&0xF0})/16) | ((r_y\texttt{\&0x0F}) \cdot 16)$, $\forall y \in \{0, \ldots, (n-k)/8\}$
    ▷ Byte permutations
16. $row_y \leftarrow r_{y \oplus i}$, $\forall y \in \{0, \ldots, (n-k)/8\}$
    ▷ Add the row to the codeword
17. $cw \leftarrow cw + row$

The encryption of a k bits plaintext requires $l - d = 20$ iterations. In each iteration step the encryption function reads an 8 bytes (t bits) long message block $(p_j, \ldots, p_{j+7})$ and determines the signature h_j of the corresponding dyadic submatrix $\Delta(h_j, t)$ serving as block key for this encryption round. Next, the encryption

function scans through all bits of the plaintext block and, if the bit is one, computes the corresponding row of the submatrix $\Delta(h_j, t)$. The row number is used to determine which permutations have to be performed on the bits and bytes of the signature h_j, respectively, to obtain the row looked for. The three most significant bits of the row number identify the block permutation while the three least significant bits are used to determine the bit permutations.

After codeword computation an error vector is added to obtain the ciphertext. We implement a CCA2-secure McEliece variant where the error vector is not chosen at random but computed within the Kobara-Imai's specific conversion γ. Hence, our implementation of the McEliece encryption function takes an arbitrary error vector as input and adds it to the codeword by flipping the corresponding bits. We first tested the encryption function using a fixed error vector. In Section 6.3.1 we describe the modifications of the McEliece encryption function we have done to apply the KIC-γ most efficiently.

6.2.3 Decryption

For decryption we use the equivalent shortened Goppa code $\Gamma(L^*, G(x))$ defined by the Goppa polynomial $G(x)$ and a (permuted) support sequence $L^* \subset \mathbb{F}_{2^{16}}$ (see Chapter 4.2.3). The support sequence consists of $n = 2304$ elements of $\mathbb{F}_{2^{16}}$ and is 4.5 Kbytes in size. We store the support sequence in an array of type `gf16_t` and size 2304.

The Goppa polynomial is a monic separable polynomial of degree $t = 64$. As t is a power of 2, the Goppa polynomial is sparse and of the form $G(x) = G_0 + \sum_{i=0}^{6} G_{2^i} x^{2^i}$. Hence, it occupies just 8·16 bits storage space. We can store both the support sequence and the Goppa polynomial in the SRAM of the microcontroller.

Furthermore, we precompute the sequence $Diag(G(L_0^*)^{-1}, \ldots, G(L_{n-1}^*)^{-1})$ for the parity-check matrix $V_{t,n}(L^*, D)$. Due to the construction of the Goppa polynomial $G(x) = \prod_{i=0}^{t-1}(x - z_i)$ where $z_i = 1/h_i + \omega$ with a random offset ω, the following holds for all $G(L_{jt+i}^*)^{-1}$.

$$G(L_{jt+i}^*)^{-1} = \prod_{r=0}^{t-1}(L_{jt+i}^* + z_r)^{-1} = \prod_{r=0}^{t-1}(1/h_{jt+i}^* + 1/h_r + 1/h_0)^{-1} = \prod_{r=0}^{t-1} h_{jt+r}^* = \prod_{r=jt}^{jt+t-1} h_r^*$$

h^* denotes a signature obtained by puncturing and permuting the signature h for the large code C_{dyad} such that $h^* = h \cdot P$ where P is the secret permutation matrix. Hence, the evaluation of the Goppa polynomial on any element of the support block $(L_{jt}^*, \ldots, L_{jt+t-1}^*)$ where $j \in \{0, \ldots, l-1\}$, $i \in \{0, \ldots, t-1\}$ leads to the same result. For this reason, only $n/t = l = 36$ values of type `gf16_t` need to be stored.

Another polynomial we need for Patterson's decoding algorithm is $W(x)$ satisfying $W(x)^2 \equiv x \mod G(x)$. As the Goppa polynomial $G(x)$ is sparse, the polynomial $W(x)$ is also sparse and of the form $W(x) = W_0 + \sum_{i=0}^{5} W_{2^i} x^{2^i}$.

Proof:
We can split the Goppa polynomial $G(x)$ in squares and non-squares such that $G(x) = \tilde{G}_0^2(x) + x \cdot \tilde{G}_1^2(x)$. Then we obtain

$$\begin{aligned} \tilde{G}_0(x) &= \sqrt{G_0} + \sum_{i=0}^{5} \sqrt{G_{2^{i+1}} x^{2^{i+1}}} \\ &= \sqrt{G_0} + \sum_{i=0}^{5} x^{2^i} \sqrt{G_{2^{i+1}}} \end{aligned}$$

and

$$\tilde{G}_1(x) = \sqrt{G_1}.$$

Hence, in this particular case, we obtain $W(x)$ by

$$W(x) = \tilde{G}_1(x)^{-1} \cdot \tilde{G}_0(x) = \frac{\sqrt{G_0}}{\sqrt{G_1}} + \sum_{i=0}^{5} \frac{\sqrt{G_{2^{i+1}}}}{\sqrt{G_1}} x^{2^i} = W_0 + \sum_{i=0}^{5} W_{2^i} x^{2^i}$$

The polynomial $W(x)$ occupies $7 \cdot 16$ bits storage space.

Syndrome computation

The first step of the decoding algorithm is the syndrome computation. We have implemented two different methods explained in the following section.

Normally, the syndrome computation is performed through solving the equation $S_c(x) = S_e(x) \equiv \sum_{i \in E} \frac{1}{x - L_i^*} \mod G(x)$ where E denotes a set of error positions. The polynomial $\frac{1}{x - L_i^*}$ satisfies the equation

$$\frac{1}{x - L_i^*} \equiv \frac{1}{G(L_i^*)} \sum_{j=s+1}^{t} G_j L_i^{*j-s-1} \mod G(x), \ \forall 0 \leq s \leq t-1 \qquad (6.2)$$

The coefficients of this polynomial are components of the $i-th$ column of the Vandermonde parity-check matrix for the Goppa code $\Gamma(G(x), L^*)$. Hence, to compute the syndrome of a ciphertext c we perform the on-the-fly computation of the rows of the parity-check matrix. As the Goppa polynomial is a sparse monic polynomial of the form $G(x) = G_0 + \sum_{i=0}^{6} G_{2^i} x^{2^i}$ with $G_{64} = 1$, we can simplify the Equation 6.2, and thus, reduce the number of operations needed for the syndrome computation.

Algorithm 6.2.2 presents the syndrome computation procedure implemented in this thesis.

Algorithm 6.2.2 On-the-fly computation of the syndrome polynomial

Input: Ciphertext array c of type `uint8_t` and size $n/8$ bytes, support set L^*, Goppa polynomial $G(x) = G_0 + \sum_{i=0}^{6} G_{2^i} x^{2^i}$ with $G_{64} = 1$

Output: Syndrome $S_c(x) = \sum_{i=0}^{t-1} S_{c,i} x^i$

1. **for** $i = 0$ **to** $n/8$ **do**
2. **for** $j = 0$ **to** 7 **do**
3. **if** $c_{i \cdot 8+j} = 1$ **then**
4. $\triangleright$ compute the polynomial $S'(x) = \frac{1}{x - L^*_{i \cdot 8+j}} \mod G(x)$
5. $S'_{62} \leftarrow 1$
6. $S'_{62} \leftarrow L^*_{i \cdot 8+j}$
7. **for** $r = 61$ **to** 33 **by** -2, $s = 1$ **to** 15 **do**
8. $S'_r \leftarrow {S'_{r+s}}^2$
9. $S'_{r-1} \leftarrow S'_{r+s} \cdot S'_{r+s-1}$
10. **for** $r = 32$ **to** 1 **by** -1 **do**
11. $S'_{r-1} \leftarrow S'_r \cdot L^*_i$
12. **if** $r = 2^s$ **then** $\triangleright$ for all powers of 2 only
13. $S'_{r-1} \leftarrow S'_{r-1} + G_{2^s}$
14. $S_c(x) \leftarrow S_c(x) + S'(x)/G(L^*_i)$
15. **return** $S_c(x)$

The main advantage of this computation method is that it is performed on-the-fly such that no additional storage space is required. To speed-up the syndrome computation the parity-check matrix can be precomputed at the expense of additional $n(n-k) = 288\,Kbytes$ memory. As the size of the Flash memory of ATxmega256A1 is restricted to $256\,Kbytes$, we cannot store the whole parity-check matrix. It is just possible to store 52 coefficients of each syndrome polynomial at most, and to compute the remaining coefficients on-the-fly.

A better possibility is to work with the systematic quasi-dyadic public parity-check matrix $\hat{H} = [Q^T | I_{n-k}]$ from which the public generator matrix $\hat{G} = [I_k | Q]$ is obtained. To compute a syndrome the vector matrix multiplication $\hat{H} \cdot c^T = c \cdot \hat{H}^T$ is performed. For the transpose parity-check matrix $\hat{H}^T = [Q^T | I_{n-k}]^T = \left[\begin{array}{c} Q \\ \hline I_{n-k} \end{array}\right]$ holds, where Q is the quasi-dyadic part composed of dyadic submatrices. Hence, to compute a syndrome we proceed as follows. The first k bits of the ciphertext are multiplied by the part Q which can be represented by the signatures of the dyadic submatrices. The storage space occupied by this part is $2.5\,Kbytes$. The multiplication is performed in the same way as encryption of a plaintext (see Section 6.2.2) and results in a binary vector s' of length $n-k$. The last $n-k$ bits

of the ciphertext are multiplied by the identity matrix I_{n-k}. Hence, we can omit the multiplication and just add the last $n-k$ bits of c to s'. To obtain a valid syndrome the vector s' first have to be multiplied by a scrambling matrix S (see Section 6.2.1). We stress that this matrix brings the Vandermonde parity-check matrix for the private code $\Gamma(G(x), L^*)$ in systematic form which is the same as the public parity-check matrix. Hence, S has to be kept secret. We generate S over $\mathbb{F}_2$ and afterwards represent it over $\mathbb{F}_{2^{16}}$. Thus, the multiplication of a binary vector s' by S results in a polynomial $S_c(x) \in \mathbb{F}_{2^{16}}[x]$ which is a valid syndrome. The matrix S is 128 Kbytes in size and can be stored in the Flash memory of the microcontroller (see Section 7.1 for more information).

Syndrome inversion

The next step of the decoding algorithm is syndrome inversion $T(x) = S_c(x)^{-1} \mod G(x)$. The syndrome inversion is done using the binary Extended Euclidean Algorithm 6.2.3.

Algorithm 6.2.3 Binary Extended Euclidean Algorithm over $GF(2^{16})$

INPUT: $G(x) \in \mathbb{F}_{2^{16}}[x], S_c(x) \in \mathbb{F}_{2^{16}}[x]$ *with degree* $S_c < G$
OUTPUT: $S_c(x)^{-1} \mod G(z)$

1. $u \leftarrow S_c, v \leftarrow G, f \leftarrow 1, g \leftarrow 0$
2. **while** $u \neq const$ *AND* $v \neq const$ **do** $\triangleright$ $degree(u) > 0$ and $degree(v) > 0$
3. **while** $u_0 = 0$ **do**
4. $u \leftarrow \frac{u}{x}$
5. $f \leftarrow \frac{f+t \cdot G}{x} \triangleright t = \frac{f_0}{G_0}, t \in \mathbb{F}_{2^{16}}$
6. **while** $v_0 = 0$ **do**
7. $v \leftarrow \frac{v}{x}$
8. $g \leftarrow \frac{g+t \cdot G}{x} \triangleright t = \frac{g_0}{G_0}, t \in \mathbb{F}_{2^{16}}$
9. **if** $degree(u) \geq degree(v)$ **then**
10. $u \leftarrow u - t \cdot v \triangleright t = \frac{u_0}{v_0}, t \in \mathbb{F}_{2^{16}}$
11. $f \leftarrow f - t \cdot g$
12. **else**
13. $v \leftarrow v - t \cdot u \triangleright t = \frac{v_0}{u_0}, t \in \mathbb{F}_{2^{16}}$
14. $g \leftarrow g - t \cdot f$
15. **return** $\frac{f}{u_0}$

We are thankful to Stefan Heyse for providing an efficient implementation of the Extended Euclidean Algorithm over $\mathbb{F}_{2^{10}}$, also presented in [EGHP09]. We modified the implementation to work over the tower field $\mathbb{F}_{2^{16}} = \mathbb{F}_{(2^8)^2}$ where any field element is represented by two elements of $\mathbb{F}_{2^8}$. In addition, using the fact

that the Goppa polynomial $G(x)$ is sparse, we optimized the implementation for a more efficient multiplication by the Goppa polynomial.

Computation of the error locator polynomial

In the next step of the decoding algorithm the error locator polynomial $\sigma(x)$ is obtained.

To do so, we first check if the inverse syndrome polynomial $T(x)$ equals x. In this case it holds for the error locator polynomial $\sigma(x) = x$.

Otherwise, we compute the square root $R(x)$ of $T(x) + x = R(x)^2$ satisfying

$$R(x) = \sqrt{T(x) + x} \mod G(x) \equiv R_0(x) + W(x)R_1(x) \mod G(x)$$

where $W(x) = W_0 + \sum_{i=0}^{5} W_{2^i} x^{2^i}$ has been precomputed (see introduction to the current Chapter).

Therefore, we split the polynomial $T(x) + x = R(x)^2 \equiv R_0(x)^2 + x \cdot R_1(x)^2$ in squares and non-squares and obtain the polynomials $R_0(x)$ and $R_1(x)$ through square root extraction of the corresponding coefficients.

```
for (i=(POLYSIZE>>1);0<i--;)
{
    sqrt_gf16(&Rq[i<<1],&R0[i]);
    sqrt_gf16(&Rq[(i<<1)+1],&R1[i]);
}
```

The function `sqrt_gf16(gf16_t *A, gf16_t *B)` computes a square root of an element `A` over the tower field $\mathbb{F}_{(2^8)^2}$, as explained in Section 6.1, and stores the result in `B`. The above listing presents a procedure computing square roots of the coefficients of the polynomial `Rq`$=T(x) + x$. The value `POLYSIZE` is equal to the number of errors t. As the polynomial $T(x) + x$ is of degree at most $t - 1 = 63$, both polynomials $R_0(x)$ and $R_1(x)$ are of degree at most $t/2 - 1 = 31$.

The multiplication of $R_1(x)$ by $W(x)$ can be performed very efficient because $W(x)$ is of degree $t/2 = 32$ such that no reduction modulo Goppa polynomial is necessary. In addition $W(x)$ is sparse and has only 7 non-zero coefficients. Equation 6.3 summarizes the simplified multiplication procedure where W_i and

$R_{1,j}$ denote the coefficients of $W(x)$ and $R_1(x)$, respectively.

$$\begin{aligned} R_1(x) \cdot W(x) &= W_{32} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x^{32} \\ &+ W_{16} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x^{16} \\ &+ W_{8} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x^{8} \\ &+ W_{4} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x^{4} \\ &+ W_{2} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x^{2} \\ &+ W_{1} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \cdot x \\ &+ W_{0} \cdot (R_{1,31}x^{31} + R_{1,30}x^{30} + \cdots + R_{1,1}x + R_{1,0}) \end{aligned}$$

By adding the polynomial $R_0(x)$ to $R_1(x) \cdot W(x)$ we finally obtain the square root $R(x)$ of $T(x) + x$.

Using a slightly modified version of the Extended Euclidean Algorithm 6.2.4 we can compute the polynomials $a(x)$ and $b(x)$ with $deg(a(x)) \leq \lfloor t/2 \rfloor$ and $deg(b(x)) \leq \lfloor (t-1)/2 \rfloor$ satisfying the equation $a(x) \equiv R(x)b(x) \mod G(x)$. The algorithm takes as input both polynomials $R(x)$ and $G(x)$ and terminates the computation when the degree of $a(x)$ drops below the bound $\lfloor (t+1)/2 \rfloor$ for the first time. In our case that is when $deg(a(x)) = 32$ and $deg(b(x)) = 31$.

Algorithm 6.2.4 Binary EEA over $GF(2^{16})$ with stop value

INPUT: $G(x) \in \mathbb{F}_{2^{16}}[x], R(x) \in \mathbb{F}_{2^{16}}[x]$ *with degree* $R < G$,
stop value $stop = \lfloor (t+1)/2 \rfloor$
OUTPUT: $a(x), b(x)$ with $a(x) \equiv R(x)b(x) \mod G(x)$

1. $A \leftarrow G, B \leftarrow R, v \leftarrow 1, u \leftarrow 0$
2. **while** $deg(A) \geq stop$ **do**
3. $\quad q \leftarrow \frac{A_k}{B_k}$ $\triangleright$ A_k, B_k are leading coefficients of A and B, respectively, in the iteration step k
4. $\quad A \leftarrow A - q \cdot B \cdot x^{deg(A)-deg(B)}$
5. $\quad u \leftarrow u - q \cdot v \cdot x^{deg(A)-deg(B)}$
6. $\quad$ **if** $deg(A) < deg(B)$ **then**
7. $\quad\quad Swap(A, B)$
8. $\quad\quad Swap(u, v)$
9. **return** (B, v)

The start values for Algorithm 6.2.4 are $A = a_0(x)$, $B = b_{-1}(x)$, $v = b_0(x)$, and $u = a_{-1}(x)$ where $a_i(x), b_i(x)$ denote the polynomials $a(x)$ and $b(x)$, respectively, after iteration i of the algorithm. As before, the implementation of the Extended Euclidean Algorithm over $\mathbb{F}_{2^{10}}$ with stop value has been provided by Stefan Heyse. We modified and optimized the implementation so that it works over the tower field $\mathbb{F}_{(2^8)^2}$ with a sparse Goppa polynomial.

Now, as the polynomials $a(x)$ and $b(x)$ are known, we can compute the error locator polynomial $\sigma(x)$. The function `square_gf16(gf16_t *A, gf16_t *B)` in the listing denotes the squaring of an element `A` over the tower field $\mathbb{F}_{(2^8)^2}$. The result is stored in the value `B`.

```
for (i=0;i<(POLYSIZE>>1);i++)
{
  square_gf16(&a[i],&sigma[i<<1]);
  square_gf16(&b[i],&sigma[(i<<1)+1]);
}
square_gf16(&a[i],&sigma[POLYSIZE]);
```

Searching for roots of $\sigma(x)$

The last and the most computationally expensive step of the decoding algorithm is the search for roots of the error locator polynomial $\sigma(x)$. For this purpose, we first planed to implement the Berlekamp trace algorithm which is known to be the best algorithm for finding roots of polynomials over finite fields with small characteristic. Considering the complexity of this algorithm we found out that it is absolutely unsuitable for punctured codes over a large field. The first step of the Berlekamp trace algorithm is the computation of the trace $Tr(\beta_i \cdot x)$ defined by

$$Tr(\beta_i \cdot x) = \beta_i x + \beta_i x^2 + \beta_i x^{2^2} + \cdots + \beta_i x^{2^{15}}$$

where β_i is an element of the $\mathbb{F}_2$ basis of $\mathbb{F}_{2^{16}}$. Hence, $Tr(\beta_i \cdot x)$ is a polynomial of degree 32768, occupying about 64 Kbytes storage space. We cannot precompute the trace functions for all basis elements β_i, as proposed in [BH09]. Furthermore, we cannot work with polynomials of such high degree on the microcontroller due to the size of the SRAM restricted to 16 Kbytes. Therefore, the reduction modulo polynomial $\sigma(x)$ of degree at most 64 is necessary after each step of the trace computation performed on-the-fly. Another computationally expensive operation is the computation of GCDs. The complexity of the Berlekamp trace algorithm exceeds that of the simple polynomial evaluation method on the fixed support L^*. The reason is that only $n/(2^m) = 2304/2^{16} \approx 0.035$ field elements are in the shortened support sequence, and thus, possible roots of the error locator polynomial. Hence, it would be pointless to factorize the polynomial $\sigma(x)$ using trace functions over the large field $\mathbb{F}_{2^{16}}$.

The next root finding method we analyzed is the Chien search which has theoretical complexity $\mathcal{O}(n \cdot t)$ if $n = 2^m$. The Chien search scans automatically all $2^m - 1$ field elements, in a more sophisticated manner than the simple polynomial evaluation method. Unfortunately, in our case $n << 2^m$ such that the complexity of the Chien search becomes $\mathcal{O}(2^{16} \cdot t)$ which is enormous compared to the complexity of the simple polynomial evaluation method.

Another disadvantage of both the Berlekamp trace algorithm and the Chien search is that after root extraction the found roots have to be located within the support sequence to identify error positions. That is not the case when evaluating the error locator polynomial on the support set directly. In this case we know the positions of the elements L_i^* and can correct errors directly by flipping the corresponding bits in the ciphertext.

The only algorithm which actually decreases the computation costs of the simple evaluation method in the case of punctured codes is the Horner scheme. The complexity of the Horner scheme does not depend on the extension degree of the field but on the number of possible root candidates, which is n. In addition, as the Horner scheme evaluates the error locator polynomial on the support set L^*, the root positions within L^* are known such that errors can be corrected more efficiently. Hence, we have implemented this root finding algorithm in this thesis.

To speed-up the evaluation of the polynomial $\sigma(x)$ on all support elements using the Horner scheme we provide the following algorithm.

Algorithm 6.2.5 Error correction: Horner scheme with polynomial division

Input: Ciphertext ct, error locator polynomial $\sigma(x) = \sum_{i=0}^{t} \sigma_i x^i$, support set L^*

Output: Codeword cw

1. $\sigma(x) \leftarrow \sigma(x)/\sigma_t \triangleright\ \sigma_t$ is the leading coefficient of $\sigma(x)$
2. $cw \leftarrow ct$
3. $i \leftarrow 0$
4. **while** $deg(\sigma(x)) > 1$ **do**
5. Evaluate $\sigma(L_i^*)$ using Horner scheme
6. **if** $\sigma(L_i^*) = 0$ **then**
7. $\sigma(x) \leftarrow \frac{\sigma(x)}{x - L_i^*} \triangleright$ Decrease $deg(\sigma(x))$
8. $cw_i \leftarrow cw_i \oplus 1 \triangleright$ Toggle bit i
9. $i \leftarrow i + 1$
   $\triangleright$ Now $\sigma(x) = x + \sigma_0 \Rightarrow \sigma_0$ is the last root
10. Find σ_0 in L^* scanning the remaining $n - i$ elements
11. Correct the last error bit cw_j at the position $j = L^*[\sigma_0]$
12. **return** cw

After a root L_i^* of $\sigma(x)$ has been found we perform the polynomial division of $\sigma(x)$ by $(x - L_i^*)$. To simplify the polynomial division we first divide $\sigma(x)$ by its leading coefficient σ_t so that $\sigma(x)$ becomes monic. We can do so, because searching for roots is solving the equation $\sigma(x) = 0$. As both, $\sigma(x)$ and $(x - L_i^*)$, are monic, the polynomial division can be performed very efficiently.

$$
\begin{aligned}
y(x) = \frac{\sigma(x)}{x - L_i^*} &= \frac{x^t + \sigma_{t-1}x^{t-1} + \cdots + \sigma_1 x + \sigma_0}{x - L_i^*} \\
&= x^{t-1} + \underbrace{(\sigma_{t-2} + L_i^*)}_{y_{t-2}} x^{t-2} + \underbrace{[\underbrace{(\sigma_{t-2} + L_i^*)}_{y_{t-2}} L_i^* + \sigma_{t-3}]}_{y_{t-3}} x^{t-3} + \cdots \\
&= x^{t-1} + y_{t-2}x^{t-2} + \underbrace{(y_{t-2}L_i^* + \sigma_{t-3})}_{y_{t-3}} x^{t-3} + \cdots + \underbrace{(y_1 L_i^* + \sigma_0)}_{y_0}
\end{aligned}
$$

We observed that the polynomial division by $(x - L_i^*)$ can be performed sequentially reusing values computed in previous iteration steps. In the first step we compute the coefficient y_{t-2} of the polynomial $y(x)$ searched for. In every iteration step j we use the previous coefficient y_{t-j+1} to compute $y_{t-j} = y_{t-j+1}L_i^* + \sigma_{t-j}$. The whole procedure requires $t-3$ multiplications and $t-2$ additions to divide a degree-t polynomial by $x - L_i^*$.

The main advantage of performing polynomial division each time a root has been found is that the degree of the error locator polynomial decreases. Hence, the next evaluation steps require less operations. For instance, after $t/2$ roots of $\sigma(x)$ have been found only a polynomial of degree $t/2$ must be evaluated to find the remaining $t/2$ roots.

6.3 Implementation of the KIC-γ

For the implementation of the Kobara-Imai's specific conversion γ [KI01] introduced in Section 5.1 two parameters have to be chosen: the length of the random value r and the length of the public constant *Const*. The authors propose that the length of r and *Const* are both 160 bits. We disagree to this proposal. The length of r should be equal to the output length of the used hash function. Nowadays, there is no secure hash function producing an output of 160 bits. Hence, we prefer to use a hash function with 256 bits output length, e.g. SHA-256 or one of the SHA-3 candidates. For the implementation of the KIC-γ on ATxmega256A1 we used an efficient assembler implementation of the Blue Midnight Wish (BMW) hash function which has been provided by the Chair for embedded security at the Ruhr-University Bochum.

As we have $|r| = 256$ and $|Const| = 160$, the message to be encrypted should be of the length $|m| \geq \left\lfloor \log_2 \binom{n}{t} \right\rfloor + k + |r| - |Const| = 1281\,bits$. Hence, we encrypt messages of length $1288\,bits = 161\,bytes$. In this case the data redundancy is even below of that of the McEliece scheme without conversion: $1288/2304 \leq 1280/2304$.

6.3.1 Encryption

The first steps of the KIC-γ encryption function are the generation of a random seed r for the function $Gen(r)$, as well as the one-time-pad encryption of the message m padded with the public constant $Const$ and the output of $Gen(r)$. The result is a 1288 bits+160 bits=1448 bits = 181 bytes big value y_1. The generation of random values can be implemented using the C-*rand* function. In the next step the hash value of y_1 is added to the random seed r by the xor operation to obtain the value y_2.

The most important step of the KIC-γ is the constant weight encoding function $Conv$ explained in Section 5.2. To optimize the computation we restrict the values for d to powers of two. Then the function `decodefd` can be simplified. Algorithm 6.3.1 presents the modified constant weight encoding function implemented in this thesis. The function $best\_u(n,t)$ computes the optimal exponent u for $d = 2^u$.

Algorithm 6.3.1 Constant weight coding: function `BtoCW`

Input: three integers n, t, and δ, a binary stream B
Output: a t-tuple of integers $(\delta_1, \ldots, \delta_t)$

1. **if** $t = 0$ **then**
2. **return**
3. **else if** $n \leq t$ **then**
4. **return** δ, `BtoCW`$(n-1, t-1, 0, B)$
5. **else**
6. $u \leftarrow best\_u(n,t)$
7. $d \leftarrow 2^u$
8. **if** `read`(B,1)=1 **then**
9. **return** `BtoCW`$(n-d, t, \delta+d, B)$
10. **else**
11. $i \leftarrow$ `read`(B, u)
12. **return** $\delta + i$, `BtoCW`$(n-i-1, t-1, 0, B)$

While the $k = 1280$ least significant bits of $(y_2||y_1)$ are used as input for the McEliece encryption function, the remaining 424 bits can be used as input for the $Conv$ function. The input for the constant weight encoding function is not fixed. We determined that our implementation takes at least 408 bits=51 bytes as input. Hence, we have to store the remaining 2 bytes in y_5.

As mentioned above, the McEliece encryption function takes as input a k-bits long part of $(y_2||y_1)$ as well as the delta-array produced by the constant weight encoding function. To add an error vector denoted by the delta-array to a codeword the modified McEliece encryption function proceeds as follows: it first computes the error positions and then flips the corresponding bits of the codeword.

```
int16_t pos = -1;
for(i=0;i<errors;i++){
    pos += delta_array[i] + 1;
    ct[pos>>3] ^= (0x80>>(pos&0x07));
}
```

6.3.2 Decryption

To decrypt a ciphertext the KIC-γ first stores the first two bytes of the ciphertext in y_5. Then it calls the McEliece decryption function which returns the encrypted plaintext y_3 and the error vector in form of the delta-array. For this purpose we modified the error correction function within the McEliece decoding in the following manner. The McEliece decryption function first corrects errors and then returns the delta-array with components computed by $\delta_j = i_j - i_{j-1} - 1$ where i_r denote the error positions.

To obtain part y_4 from the delta-array constant weight decoding function is used. As the values for d are restricted to the powers of two this function can also be simplified.

Algorithm 6.3.2 Constant weight coding: function `CWtoB`

Input: two integers n and t, and a t-tuple $(\delta_1, \ldots, \delta_t)$
Output: a binary string B

1. **if** $t = 0$ or $n \leq t$ **then**
2. **return**
3. $u \leftarrow best\_u(n, t)$
4. $d \leftarrow 2^u$
5. **if** $\delta_1 \geq d$ **then**
6. **return** `1||CWtoB`$(n - d, t, (\delta_1 - d, \delta_2, \ldots, \delta_t))$
7. **else**
8. $s \leftarrow 0||LSB_u(\delta_1|_{\mathbb{F}_2})$
9. **return** `s||CWtoB`$(n - \delta_1 - 1, t - 1, (\delta_2, \ldots, \delta_t))$

Now $(y_2||y_1) = (y_5||y_4||y_3)$ is known and the message m can be obtained as introduced in Section 5.1.2.

7 Implementation on an 8-bits AVR microcontroller

This chapter discusses our implementation of the quasi-dyadic McEliece variant including the Kobara-Imai's specific conversion γ on an 8-bits AVR microcontroller. For porting our implementation described in Section 6 to the AVR micro architecture from Atmel we used the AVR Studio version 4.18 and the WinAVR-20090313 IDE as plugin for C development. The target device is the ATXmega256A1 microcontroller with 256 Kbytes Flash memory (access time 3 cycles/operation) and 16 Kbytes SRAM (access time 2 cycles/operation). This RISC microcontroller operates at a clock frequency up to 32 MHz.

In Section 7.1 we discuss all problems occurring while porting our implementation to the AVR micro architecture. In the last Section 7.2 we give an analysis of the side channel security of our implementation.

7.1 Porting to AVR

Most of the implemented functions could be ported to the AVR micro architecture without any problem. Unfortunately, the AVR micro architecture does not have a pseudo random number generator (PRNG). Hence, we used fixed random values for the implementation of the KIC-γ. Later, the hardware based DES or AES acceleration of the target device can be used to implement a PRNG.

Another problem occurs when implementing the syndrome computation method using the matrix S (see Section 6.2.3). The matrix S is 128 Kbytes in size and cannot be stored in the SRAM of the microcontroller. Hence, we have to use the Flash ROM for this purpose. To forbid the copying of this matrix to the SRAM when starting up the code the `PROGMEM` directive is used. Note that without this directive the C-compiler tries to copy all data to the SRAM at startup time. As the matrix S does not fit into the SRAM the copying process would result in an error message.

```
const gf16_t Smat[] PROGMEM = {...};
```

By doing so, the matrix S is loaded to the Flash ROM together with the program data. Our first naive approach was to store the matrix S as array of size

$64 \cdot 2 \cdot 1024 = 131072\,bytes$. Unfortunately, the AVR uses 16 bit pointers to access its Flash ROM where a pointer is a signed integer. Hence, a pointer can address an array of size (32 Kbytes-1 byte) at most. Therefore, the matrix S need to be split into multiple parts at the expense of additional overhead in the program code. We took the decision to split S into 8 arrays of size $128 \cdot 64 \cdot 2\,bytes$ each. Then the multiplication of a binary vector s' by S is performed as presented in Algorithm 7.1.1.

Algorithm 7.1.1 Multiplication of s' by S with 8 tables

Input: binary vector s' of length 64, tables S_{table_i} where $0 \leq i \leq 7$
Output: a syndrome polynomial $S_c(x)$

1. $S_c(x) \leftarrow 0$
2. **for** $i = 0$ **to** 15 **do**
3. **for** $j = 0$ **to** 7 **do**
4. **if** $s'_{i \cdot 8+j} = 1$ **then**
5. $S_c(x) \leftarrow S_c(x) + S_{table_0}[i \cdot 8 + j]$
6. **if** $s'_{(i+16) \cdot 8+j} = 1$ **then**
7. $S_c(x) \leftarrow S_c(x) + S_{table_1}[i \cdot 8 + j]$
8. **if** $s'_{(i+32) \cdot 8+j} = 1$ **then**
9. $S_c(x) \leftarrow S_c(x) + S_{table_2}[i \cdot 8 + j]$
10. **if** $s'_{(i+48) \cdot 8+j} = 1$ **then**
11. $S_c(x) \leftarrow S_c(x) + S_{table_3}[i \cdot 8 + j]$
12. **if** $s'_{(i+64) \cdot 8+j} = 1$ **then**
13. $S_c(x) \leftarrow S_c(x) + S_{table_4}[i \cdot 8 + j]$
14. **if** $s'_{(i+80) \cdot 8+j} = 1$ **then**
15. $S_c(x) \leftarrow S_c(x) + S_{table_5}[i \cdot 8 + j]$
16. **if** $s'_{(i+96) \cdot 8+j} = 1$ **then**
17. $S_c(x) \leftarrow S_c(x) + S_{table_6}[i \cdot 8 + j]$
18. **if** $s'_{(i+112) \cdot 8+j} = 1$ **then**
19. $S_c(x) \leftarrow S_c(x) + S_{table_7}[i \cdot 8 + j]$

Another problem is that when storing the S-tables in the Flash ROM the rows $S_{table_r}[i \cdot 8 + j]$ cannot be accessed directly. The cause is that AVR is a Harvard architecture with separated buses for program data stored in the Flash ROM and other data stored in the SRAM. To access data stored in the program space we include the header file `pgmspace.h` [pgm] and use the macro `pgm_read_byte_far` defined there. We can use this macro because the target platform supports the ELPM instructions. This macro takes as input a signed int32 pointer to the requested array. To obtain this pointer another macro `FAR` provided by [Sag] is used.

The components of the S-tables are of type `gf16_t` which is a structure consisting of two `uint8_t` elements. However, the macro `pgm_read_byte_far` is able to read

a byte only. The following listing makes clear how to read the $(i \cdot 8 + j)$-th row of a S-table of type `gf16_t`.

```
uint8_t len = n/8-k/8;
for(y=126;y>=0;y-=2)
{
  SynPoly[(y/2)].highByte^=pgm_read_byte_far(FAR(Smat0)+(i*8+j)*len+y);
  SynPoly[(y/2)].lowByte^=pgm_read_byte_far(FAR(Smat0)+(i*8+j)*len+y+1);
}
```

7.2 Side channel security

Embedded devices can always be subject of passive attacks, e.g. timing attacks and power analysis attacks. The goal is either to recover the secret information L^* and $G(x)$ or to recover the error vector without the knowledge of the secret information. Within our implementation there are two critical functions where side channel attacks seem conceivable: syndrome computation and the extraction of roots of the error locator polynomial.

Neither the implementation of the Horner scheme without nor with polynomial division is resistant against side channel attacks.

We first consider the implementation without polynomial division. This method has a constant running time for all degree-t polynomials. An attacker may try to toggle an arbitrary bit in the received ciphertext in the hope that the flipped bit was an error bit. That can be verified by running the decoding algorithm and measuring the running time. If the flipped bit was an error bit then the degree of the error locator polynomial $\sigma(x)$ is at most $t-1$. In this case the evaluation of $\sigma(x)$ takes less time compared to the evaluation of an degree-t polynomial. Scanning through all bits of the ciphertext the attacker can efficiently find all errors.

In our implementation we use error vectors of Hamming weight t only, constructed by the constant weight encoding function within the KIC-γ. Hence, to protect the implementation against the timing attack explained above we first check whether the degree of the error locator polynomial is exactly t. If the degree of $\sigma(x) = t' < t$, we evaluate the error locator polynomial on an arbitrary element of the support sequence (equivalently an arbitrary element of $\mathbb{F}_{2^{16}}$) $t - t'$ times without error correction. The time taken for the error correction is negligible. Hence, the running time of the Horner scheme is constant, as it is independent of the degree of $\sigma(x)$.

The implementation of the Horner scheme with polynomial division is not resistant against timing attacks, as it does not have a constant running time. Let us assume that the first t bits of the ciphertext contain errors. As we perform

polynomial division after a root has been found, the degree of $\sigma(x)$ decreases fast. After t evaluation steps all roots are known and the algorithm terminates. In contrast, if all t errors are in the last t bits of the ciphertext, $n-t$ evaluation steps of a degree-t polynomial must be performed before the degree of the error locator polynomial decreases. In this case the running time of the algorithm is a lot longer. Hence, the attacker is able to find the error vector by measuring the running time of the algorithm.

Another problem is that the power consumption is not constant. The evaluation of a degree-t polynomial involves more operations compared to the evaluation of a polynomial of less degree. As the degree of the error locator polynomial decreases after every root found, the attacker is able to identify the error positions by means of the power trace.

To protect the implementation against both the timing attack and the simple power analysis the implementation can be modified in the following manner. Instead of incrementing the index i in each iteration step of Algorithm 6.2.5 a random index must be chosen. A pseudo random number generator implemented on the microcontroller can be used to generate a random sequence Ind of distinct elements j with $0 \leq j \leq n-1$. Taking the index i of Algorithm 6.2.5 from the random index set Ind the root evaluation is carried out in a random order. By this means the correlation between the power consumption / running time and the error positions is destroyed.

The next critical point within our implementation is syndrome computation. As explained in Section 6.2.3, we have implemented two different methods for this purpose: on-the-fly syndrome computation and syndrome computation using a precomputed matrix S. It seems not possible to perform a timing attack on both syndrome computation methods, as the degree of the syndrome polynomial is always $t-1$ and independent from the number of induced errors. Hence, the running time is constant. The question is whether it is possible to recover the support L^* and the Goppa polynomial $G(x)$ used in the syndrome computation methods by means of the power analysis. The security against power analysis has not been explicitly studied within the scope of this thesis. Nevertheless, we provide some ideas for further investigation.

The first method uses elements of the prepermuted support directly to compute the rows of the private (transpose) parity-check matrix on-the-fly. Hence, an idea is to try to determine the Hamming weights of the elements $L_i^*, (L_i^*)^2, \cdots, (L_i^*)^{31}$ and to use the correlation between these elements to reduce the number of possible candidates. We recall that an element $A \in F_{2^{16}}$ is represented by two elements A_l and A_h in $\mathbb{F}_{2^8}$. Hence, $C = A^2$ satisfies the equation $C_h^2 = A_h^2$ and $C_l^2 = A_h^2 p_0 + A_l^2$ where the defining polynomial of $F_{2^{16}}$ is $p(x) = x^2 + x + p_0$. We may try to recover the elements A_h first. Once these elements are known, they can be used to recover the elements A_l.

An element A_h has a polynomial representation $A_h = A_{h_0} + A_{h_1}x + \cdots + A_{h_7}x^7$. We used the polynomial $x^8+x^4+x^3+x^2+1$ for the construction of the subfield $\mathbb{F}_{2^8}$. Hence, if α is root of this polynomial, $\alpha^8 \equiv \alpha^4+\alpha^3+\alpha^2+1$ holds. The polynomial representation of A_h^2 can be denoted by $A_h^2 = (a_0+a_4+a_6)+a^7x+(a_1+a_4+a_5+a_6+a_7)x^2+(a_4+a_0)x^3+(a_2+a_4+a_5+a_6+a_7)x^4+a_5x^5+(a_3+a_5+a_6)x^6+a_6x^7$. Using this fact we can see that, for instance, if A_h has Hamming weight 1 then A_h^2 has Hamming weight 1 (3 candidates), 2 (1 candidate), 3 (1 candidate), 4 (2 candidates), or 5 (1 candidate). With knowledge of the Hamming weights for A_h^2 the number of candidates is reduced significantly. We believe that when considering further powers of A_h an efficient attack can be performed on the on-the-fly syndrome computation method. By doing so, all elements of the support sequence might be found. We leave the realization of the attack as a subject for further investigation.

Furthermore, it should be investigated whether the attacks presented in the recent work [HMP10] are applicable to our syndrome computation methods. This work discusses possible power analysis attacks on the original McEliece scheme implemented on an 8-bits AVR microcontroller. As we use a prepermuted support sequence, only the Attack II of this work is interesting. This attack recovers the permuted support sequence and the permuted parity-check matrix, respectively, for a private Goppa code. We do not store the whole parity-check matrix. However, the matrix S used for the syndrome computation with precomputation has a similar structure as the parity-check matrix. Hence, it should be investigated whether the precomputed matrix S can be recovered by means of Attack II or similar. In addition, the fact that the Goppa polynomial is sparse might be useful. We recall that the evaluation of the Goppa polynomial on any element of the support block $(L^*_{jt}, \ldots, L^*_{jt+t-1})$ where $j \in \{0, \ldots, l-1\}$, $i \in \{0, \ldots, t-1\}$ leads to the same result.

8 Results

This chapter presents the results of our implementation of the McEliece variant based on [2304,1280,129] quasi-dyadic Goppa codes providing an 80-bit security level for the 8-bits AVR microcontroller. As we use a systematic generator matrix for the Goppa code, we also implemented the Kobara-Imai's specific conversion γ developed for CCA2-secure McEliece variants (see Chapter 5). Note that the KIC-γ first randomizes and transforms a message to be encrypted into a fixed length and Hamming weight vector which is used as error vector for the quasi-dyadic encryption function. Due to the parameters chosen for the KIC-γ the actual length of the message to be encrypted increases to 1288 bytes while the ciphertext length increases to 2312 bytes (2 bytes overhead for the conversion).

Table 8.1 summarizes the sizes of all parameters being precomputed and used for the encryption and decryption algorithms. As we work with field elements over $\mathbb{F}_{2^{16}}$ the real size of all parameters equals the actual storage space occupied by the parameters.

	Parameter	**Size**
QD-McEliece encryption	K_{pub}	2560 byte
QD-McEliece decryption	log table for $\mathbb{F}_{2^8}$	256 byte
	antilog table for $\mathbb{F}_{2^8}$	256 byte
	Goppa polynomial $G(x)$	16 byte
	Polynomial $W(x)$	14 byte
	Support sequence L^*	4608 byte
	Array with elements $1/G(L_i^*)$	72 byte
	Matrix S	131072 byte
KIC-γ	Public constant *Const*	20 byte

Table 8.1: Sizes of tables and values in memory

Except the matrix S used only within the syndrome computation method with precomputation, all precomputed values can be copied into the faster SRAM of the microcontroller at startup time resulting in faster encryption and decryption.

The performance results of our implementation were obtained from the AVR Studio in version 4.18. Table 8.2 summarizes the clock cycles needed for specific

operations and sub-operations for the conversion and encryption of a message. Note that we used fixed random values for the implementation of the KIC-γ. The encryption of a 1288 bits message requires 6,358,952 cycles. Hence, when running at 32 MHz, the encryption takes about 0.1987 seconds while the throughput is 6482 bits/second.

Operation	**Sub-operation**	**Clock cycles**
BMW Hash		15,083
BtoCW		50,667
Other		8,927
QD-McEliece encryption	Vector-matrix multiplication	6,279,662
	Add error vector (delta-array)	4,613

Table 8.2: Performance of the QD-McEliece encryption including KIC-γ on the AVR μC ATxmega256@32 MHz

Table 8.3 presents the simulation results of the operations and sub-operations of the QD-McEliece decryption function including KIC-γ.

Operation	**Sub-operation**	**Clock cycles**
QD-McEliece decryption	Syndrome computation on-the-fly	25,745,284
	Syndrome computation with S	9,118,828
	Syndrome inversion	3,460,823
	Computing $\sigma(x)$	1,625,090
	Error correction (HS)	31,943,688
	Error correction (HS with PD)	19,234,171
CWtoB		61,479
BMW Hash		15,111
Other		19,785

Table 8.3: Performance of the QD-McEliece decryption including KIC-γ on the AVR μC ATxmega256@32 MHz

The above table shows clearly that the error correction using the Horner scheme with polynomial division is about 40% faster then the Horner scheme without polynomial division. Considering the fact that the error correction is one of the most compuationally expensive functions within the decryption algorithm the polynomial division provides a significant speed gain for this operation. In the case that the syndrome is computed using the precomputed matrix S and the error correction is performed using the Horner scheme with polynomial division decoding of a 2312 bits ciphertext requires 33,535,287 cycles. Running

at 32 MHz the decryption takes 1.0480 seconds while the ciphertext rate is 2206 bits/second[1]. Decryption with the on-the-fly syndrome computation method takes 50,161,743 cycles. Hence, running at 32 MHz the decryption of a ciphertext takes 1.5676 seconds in this case while the cophertext rate is 1475 bits/second. Although the on-the-fly decryption is about 1.5 times slower, no additional Flash memory is required so that a migration to cheaper devices is possible.

Table 8.4 summarizes the resource requirements for our implementation. The third column of the table refer to the decryption method with precomputed matrix S, the fourth to the on-the-fly syndrome decoding method. For a comparison we also provide the resource requirements for the original McEliece version based on [2048,1751,55]-Goppa codes [EGHP09].

	Operation	**SRAM**	**Flash memory**	**External memory**
QD-McEliece with KIC-γ	Encryption	3.5 Kbyte	11 Kbyte	–
	Decryption (with S)	8.6 Kbyte	156 Kbyte	–
	Decryption (on-the-fly)	6 Kbyte	21 Kbyte	–
Original McEliece	Encryption	512 byte	684 byte	438 Kbyte
	Decryption	12 Kbyte	130.4 Kbyte	–

Table 8.4: Resource requirements of QD-McEliece including KIC-γ on the AVR μC ATxmega256@32 MHz

As we can see, the memory requirements of the quasi-dyadic encryption routine including KIC-γ are minimal because of the compact representation of the public key. Hence, much cheaper microcontrollers such as ATxmega32 with only 4 Kbytes SRAM and 32 Kbytes Flash ROM could be used for encryption. In contrast, the implementation of the original McEliece version even requires 438 Kbyte external memory. The implementation of the decryption method with on-the-fly syndrome computation could also be migrated to a slightly cheaper microcontroller such as ATxmega128 with 8 Kbyte SRAM and 128 Kbyte Flash memory.

Table 8.5 gives a comparison of our implementation of the quasi-dyadic McEliece variant including KIC-γ with the implementation of the original McEliece PKC and the implementations of other public-key cryptosystems providing an 80-bit security level. RSA-1024 and ECC-160 [GPW+04] were implemented on a Atmel ATmega128 microcontroller at 8 MHz while the original McEliece version was implemented on a Atmel ATxmega192 microcontroller at 32 MHz. For a fair comparison with our implementation running at 32 MHz, we scale the timings at lower frequencies accordingly. To be fair we also take into account that the message length is 1288 bits in our case where the RSA encrypts messages of length

[1] Chiphertext rate denotes number of ciphertext bits processed per second

1024 bits, ECC of length 160 bits and the original McEliece version of length 1751 bits. Hence, we also provide the throughput of the implementations. In contrast to RSA and ECC, the plaintext and ciphertext lengths in both, the original McEliece PKC and the quasi-dyadic variant, are not equal. In the throughput figures we do not take into account any message expansion in the ciphertext, but only consider the number of plaintext bits processed by each cryptosystem.

Method	**Time** ops/sec	**Throughput** bits/sec
QD-McEliece encryption	0.1987	6482
QD-McEliece decryption (with S)	1.0480	1229
QD-McEliece decryption (on-the-fly)	1.5676	822
Original McEliece encryption [EGHP09]	0.4501	3889
Original McEliece decryption [EGHP09]	0.6172	2835
ECC-160 [GPW$^+$04]	0.2025	790
RSA-1024 $2^{16}+1$ [GPW$^+$04]	0.1075	9525
RSA-1024 w. CRT [GPW$^+$04]	2.7475	373

Table 8.5: Comparison of the quasi-dyadic McEliece variant including KIC-γ (n'=2312, k'=1288, t=64) with original McEliece PKC (n=2048, k=1751, t=27), ECC-P160, and RSA-1024

Although we additionally include KIC-γ in the quasi-dyadic McEliece encryption, we were able to overperform both, the original McEliece version and ECC-160, in terms of number of operations per second. In particular, the throughput of our implementation significantly exceeds that of ECC-160.

Unfortunately, we could not overperform the original McEliece scheme neither in throughput nor in number of operations per second for the decryption. The reason is that the original McEliece version is based on Goppa codes with much smaller number of errors $t = 27$. Due to this fact, this McEliece version works with polynomials of smaller degree such that most operations within the decoding algorithm can be performed more efficiently. Another disadvantage of our implementation is that all parameters are defined over the large field $\mathbb{F}_{2^{16}}$. As we could not store the log- and antilog tables for this field in the Flash memory, we had to implement the tower field arithmetic which significantly reduces performance. For instance, one multiplication over a tower $\mathbb{F}_{(2^8)^2}$ involves 5 multiplications over the subfield $\mathbb{F}_{2^8}$. Hence, much more arithmetic operations have to be performed to decrypt a ciphertext.

Nevertheless, the decryption function is still faster than the RSA-1024 private key operation and exceeds the throughput of ECC-160. Furthermore, although slower, the on-the-fly decoding algorithm requires 81% less memory compared to the original McEliece version such that migration to cheaper devices is possible.

9 Conclusion and further research

In this thesis we have implemented a McEliece variant based on quasi-dyadic Goppa codes on a 8-bits AVR microcontroller. The family of quasi-dyadic Goppa codes offers the advantage of having a compact and simple description. Using quasi-dyadic Goppa codes the public key for the McEliece encryption is significantly reduced. Furthermore, we used a generator matrix for the public code in systematic form resulting in an additional key reduction. As a result, the public key size is a factor t less compared to generic Goppa codes used in the original McEliece PKC. Moreover, the public key can be kept in this compact size not only for storing but for processing as well. However, the systematic coding necessitates further conversion to protect the message. Without any conversions the encrypted message would be revealed immediately from the ciphertext. Hence, we have implemented the Kobara-Imai's specific conversion γ: a conversion scheme developed specially for CCA2 secure McEliece variants. We also provided the side-channel analysis of our implementation in Section 7.2.

For public-key cryptosystems implemented on embedded systems parameters providing a mid-term security level are often regarded sufficient. The reason is that many embedded systems have short life cycles. Hence, we chose [2304,1280,129] quasi-dyadic Goppa codes for our implementation providing an 80-bit security level. We generate the public Goppa codes as punctured permuted subfield subcodes over $\mathbb{F}_2$ of large private Goppa codes over $\mathbb{F}_{2^{16}}$. As we could not perform the field arithmetic over $\mathbb{F}_{2^{16}}$ directly due to memory size limitations, we have implemented the tower field arithmetic. All operations over $\mathbb{F}_{2^{16}}$ are then performed in terms of operations in the subfield $\mathbb{F}_{2^8}$ which are realized through table lookups. For the implementation of the Patterson's decoding algorithm square root extraction of polynomials over the tower field $\mathbb{F}_{2^{16}}$ is necessary. As we did not find any formula for square root extraction in the literature we developed and implemented a new one for this purpose. We provide the formal correctness proof of this formula in Section 6.1.

Although our implementation leaves room for further optimizations, we overperform the implementations of the original McEliece PKC and ECC-160 in encryption. In particular, the quasi-dyadic McEliece encryption is 2.3 times faster than the original McEliece PKC and exceeds the throughput of both, the origi-

nal McEliece PKC and ECC-160, by 1.7 and 8.2 times, respectively. In addition, our encryption algorithm requires 96,7% less memory compared to the original McEliece version and can be migrated to much cheaper devices.

The performance of the McEliece decryption algorithm is closely related to the number of errors added within the encryption. In our case the number of errors is 64 which is 2.4 times greater compared to the original McEliece PKC. Hence, the polynomials used are huge and the parity-check matrix is too large to be completely precomputed and stored in the Flash memory. In addition, the error correction requires more time because a polynomial of degree 64 has to be evaluated. We showed in Section 6.2.3 that non of the frequently used error correction algorithms, such as the Berlekamp trace algorithm and the Chien search, is suitable for punctured and shortened codes obtained from codes over very large fields. Furthermore, the tower field arithmetic significantly reduces the performance of the decoding algorithm. All facts considered, it is clear that our implementation cannot overperform the original McEliece variant in decoding. Nevertheless, the decryption algorithms with precomputation and on-the-fly are 2.6 and 1.8 times, respectively, faster than the RSA-1024 private key operation and exceed the throughput of ECC-160. Furthermore, although slower, the on-the-fly decoding algorithm requires 81% less memory compared to the original McEliece version such that migration to cheaper devices is possible.

For future work, we propose the investigation whether the decoding algorithm for the quasi-dyadic McEliece variant can be optimized to achieve better performance. For instance, the tower field arithmetic could be implemented in the assembly language. Alternatively, we believe that when implementing the quasi-dyadic McEliece variant on a 16-bits microcontroller with more memory the performance increases significantly.

Furthermore, nowadays it is not clear whether the quasi-dyadic McEliece variant is actually secure. The best known attack is able to break some instances of this McEliece variant. This does not hold if the quasi-dyadic Goppa codes are defined over the base field as subfield subcodes of codes over very large extension fields with extension degree ≥ 16. Further research on this topic should answer this question. In the positive case that the implemented quasi-dyadic variant is secure against structural attacks, the quasi-dyadic McEliece variant should be implemented on an FPGA which should result in a major speed enhancement.

A Bibliography

[Afa91] V.B. Afanasyev. On the complexity of finite field arithmetic. *Fifth Joint Soviet-Swedish Intern. Workshop Information Theory*, pages 9–12, January 1991.

[AL94] W. Adams and P. Loustaunau. *An introduction to Gröbner Bases*, volume 3. 1994.

[BBD08] Daniel J. Bernstein, Johannes Buchmann, and Erik Dahmen. *Post Quantum Cryptography*. Springer Publishing Company, Incorporated, 2008.

[BC07] M. Baldi and G. F. Chiaraluce. Cryptanalysis of a new instance of mceliece cryptosystem based on qc-ldpc codes. In *IEEE International Symposium on Information Theory*, pages 2591–2595, March 2007.

[Ber70] E. R. Berlekamp. Factoring polynomials over large finite fields. *Mathematics of Computation*, 24(111):713–715, 1970.

[Ber08] Daniel J. Bernstein. List decoding for binary goppa codes, 2008.

[BH09] Bhaskar Biswas and Vincent Herbert. Efficient root finding of polynomials over fields of characteristic 2. In *WEWoRC 2009*, July 7-9 2009.

[BLP08] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In *PQCrypto '08: Proceedings of the 2nd International Workshop on Post-Quantum Cryptography*, pages 31–46, Berlin, Heidelberg, 2008. Springer-Verlag.

[BMvT78] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Trans. Inf. Theory*, 24(3):384–386, May 1978.

[CC95] Anne Canteaut and Florent Chabaud. Improvements of the attacks on cryptosystems based on error-correcting codes, 1995.

[CC98] Florent Chabaud and Anne Canteaut. A new algorithm for finding minimum-weight words in a linear code: application to primitive narrow-sense bch codes of length 511. *IEEE Transactions on Information Theory*, 44:367–378, 1998.

[Chi64] R. Chien. Cyclic decoding procedure for the bose-chaudhuri-hocquenghem codes. *IEEE Transactions on Information Theory*, IT-10(10):357–363, 1964.

[Cov73] T. Cover. Enumerative source encoding. *IEEE Transactions on Information Theory*, 19(1):73–77, January 1973.

[EGHP09] Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. Microeliece: Mceliece for embedded devices. In *CHES '09: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*, pages 49–64, Berlin, Heidelberg, 2009. Springer-Verlag.

[EOS07] D. Engelbert, R. Overbeck, and A. Schmidt. A summary of mceliece-type cryptosystems and their security. *Journal of Mathematical Cryptology*, 1(2):151–199, 2007.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In *CRYPTO '99: Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 537–554, London, UK, 1999. Springer-Verlag.

[FOPT09] Jean-Charles Faugere, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. Algebraic cryptanalysis of mceliece variants with compact keys. 2009.

[Gab05] P. Gaborit. Shorter keys for code based cryptography. In *The 2005 International Workshop on Coding and Cryptography (WCC 2005)*, pages 81–91, March 2005.

[Gol66] S. Golomb. Run-length encoding. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[Gop70] V.D. Goppa. A new class of linear correcting codes. *Probl. Pered. Info.*, 6(3):24–30, 1970.

[GPW+04] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *Cryptographic hardware and embedded systems. CHES 2004: 6th international workshop*, pages 119–132, 2004.

[Hey09] Stefan Heyse. Code-based cryptography: Implementing the mceliece scheme on reconfigurable hardware. Diploma thesis, Ruhr-University Bochum, 2009.

[HMP10] S. Heyse, A. Moradi, and C. Paar. Practical power analysis attacks on software implementations of mceliece. 2010.

[HP03] W. C. Huffman and V. Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. Ntru: A ring-based public key cryptosystem. In *Lecture Notes in Computer Science*, pages 267–288. Springer-Verlag, 1998.

[Hub96] K. Huber. Note on decoding binary goppa codes. *Electronics Letters*, pages 102–103, 1996.

[KI01] Kazukuni Kobara and Hideki Imai. Semantically secure mceliece public-key cryptosystems-conversions for mceliece pkc. In *PKC '01: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography*, pages 19–35, London, UK, 2001. Springer-Verlag.

[Leo88] Jeffrey S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, 1988.

[MB09] Rafael Misoczki and Paulo S. Barreto. Compact mceliece keys from goppa codes. In *Selected Areas in Cryptography: 16th Annual International Workshop (SAC 2009)*, pages 376–392, Berlin, Heidelberg, August 13-14 2009. Springer-Verlag.

[McE78] Robert J. McEliece. A public-key cryptosystem based on algebraic coding theory. DSN Progress Report 42-44, Jet Propulsion Laboratory, January-February 1978.

[Mer79] Ralph Merkle. *Secrecy, authentication and public key systems / A certified digital signature*. Dissertation, Dept. of Electrical Engineering, Stanford University, 1979.

[MK89] M Morii and M. Kasahara. Efficient construction of gate circuit for computing multiplicative inverses over $gf(2^m)$. *Transactions of the IEICE*, E 72:37–42, January 1989.

[MS97] F. J. MacWilliams and N.J.A. Sloane. *The theory of error-correcting codes*, volume 16 of *North-Holland Mathematical Library*. 1997.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996. Chapter 2, page 67.

[Nie86] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems of Control and Information Theory*, 15:159–166, 1986.

[NIKM08] Ryo Nojima, Hideki Imai, Kazukuni Kobara, and Kirill Morozov. Semantic security for the mceliece cryptosystem without random oracles. *Des. Codes Cryptography*, 49(1-3):289–305, 2008.

[OS08] R. Overbeck and N. Sendrier. Code-based cryptography. In D. Bernstein, J. Buchmann, and J. Ding, editors, *Post-Quantum Cryptography*, pages 95–145. Springer, 2008.

[OTD08] Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. Cryptanalysis of two mceliece cryptosystems based on quasi-cyclic codes. *CoRR*, abs/0804.0409, 2008.

[Paa94] Christof Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. Dissertation, Institute for Experimental Mathematics, Universität Essen, 1994.

[Pat75] N.J. Patterson. The algebraic decoding of goppa codes. *IEEE Transactions on Information Theory*, IT-21(2):203–207, March 1975.

[Pat96] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In *EUROCRYPT'96: Proceedings of the 15th annual international conference on Theory and application of cryptographic techniques*, pages 33–48, Berlin, Heidelberg, 1996. Springer-Verlag.

[PBGV92] B. Preneel, A. Bosselaers, R. Govaerts, and J. Vandewalle. A software implementation of the mceliece public-key cryptosystem. In *Proceedings of the 13th Symposium on Information Theory in the Benelux, Werkgemeenschap voor Informatieen Communicatietheorie*, pages 119–126. Springer-Verlag, 1992.

[pgm] http://www.cs.mun.ca/∼paul/cs4723/material/atmel/avr-libc-user-manual-1.6.5/group__avr__pgmspace.html#gd7082c45c2c96f015c76eff1ad00a99a.

[Poi00] David Pointcheval. Chosen-ciphertext security for any one-way cryptosystem. In *PKC '00: Proceedings of the Third International Workshop on Practice and Theory in Public Key Cryptography*, pages 129–146, London, UK, 2000. Springer-Verlag.

[PR97] E. Petrank and R.M. Roth. Is code equivalence easy to decide? *IEEE Transactions on Information Theory*, 43(5):1602–1604, September 1997.

[Pro09] Prometheus. Implementation of mceliece cryptosystem for 32-bit microprocessors (c-source). http://www.eccpage.com/goppacode.c, 2009.

[Sag] Peter Sager. http://www.mikrocontroller.net/topic/30859.

[Sen00] Nicolas Sendrier. Finding the permutation between equivalent linear codes: The support splitting algorithm. *IEEE Transactions on Information Theory*, 46(4):1193–1203, 2000.

[Sen05] N. Sendrier. Encoding information into constant weight words. In *IEEE Conference, ISIT'2005*, pages 435–438, September 2005.

[Sho97] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[SMR00] A. Shokrollahi, C. Monico, and J. Rosenthal. Using low density parity check codes in the mceliece cryptosystem. In *IEEE International Symposium on Information Theory (ISIT 2000)*, page 215, 2000.

[Ste89] J. Stern. A method for finding codewords of small weight. In *Proceedings of the third international colloquium on Coding theory and applications*, pages 106–113, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[Sti08] Henning Stichtenoth. *Algebraic Function Fields and Codes*. Graduate Texts in Mathematics. Springer Publishing Company, Incorporated, 2 edition, 2008.

[Wie06] Christian Wieschebrink. Two np-complete problems in coding theory with an application in code based cryptography. In *2006 IEEE International Symposium on Information Theory*, pages 1733–1737, July 2006.

B List of Tables

C Listings

D List of Algorithms