Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices

Stefan Heyse

Horst Görtz Institute for IT Security
Ruhr University Bochum
44780 Bochum, Germany
heyse@crypto.rub.de

Abstract. 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). At present, 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 using Shor's algorithm [23]. Although a quantum computer of this dimension has not been reported, development and cryptanalysis of alternative public-key cryptosystems seem suitable. To achieve acceptance and attention in practice, they have to be implemented efficiently. Furthermore, the implementations have to perform fast while keeping memory requirements low 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. The major disadvantage of the McEliece public-key cryptosystem(PKC) is its very large public key of several hundred thousands bits. For this reason, the McEliece PKC has achieved little attention in the practice. Another disadvantage of the McEliece scheme, like many other schemes, is that it is not semantically secure. The quasi-dyadic McEliece variant proposed by Barreto and Misoczki addresses both problems. In this work we provide an implementation of this alternative public-key cryptosystem, which is semantically secure and uses a 40 times smaller public key and a five times smaller secret key compared to a previously published implementation [6].

Keywords: McEliece, Goppa Code, Quasi-Dyadic, Embedded Device, Post-Quantum.

1 Introduction

Only few implementations of the original McEliece public-key cryptosystem have been reported. For instance, there exist two software implementations for 32-bit

B.-Y. Yang (Ed.): PQCrypto 2011, LNCS 7071, pp. 143–162, 2011.

architectures: an i386 assembler implementation [20] and a C-implementation [21]. Two implementations of the McEliece PKC on an 8-bits AVR microcontroller and an FPGA have been provided by [6]. The microcontroller implementation encrypts with 3,889 bits/second and decrypts with 2,835 bits/second at a clock frequency of 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 McEliece PKC in [6] 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.

The remainder of this work is organized as follows. Section 2 introduces the classical McEliece public key scheme. In further progress we describe how binary dyadic and quasi-dyadic Goppa codes are constructed. Section 3 gives the scheme definition of the quasi-dyadic McEliece variant and describes the Kobara-Imai's specific conversion γ. In Section 4, 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 Section 5 and conclude in Section 6.

2 Background on the McEliece Cryptosystem

The McEliece cryptosystem [15] 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 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.

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. The public key is $K_{pub} = (\hat{G}, t)$, where $\hat{G} = S \cdot G \cdot P$. The private key is $K_{pr} = (S, G, P)$, where G is a $k \times n$ generator matrix for the code $\mathcal{C}$, S is a $k \times k$ scrambling matrix and P is a $n \times n$ permutation matrix. The McEliece encryption is done by multiplying a k-bit message vector by the recipient's public generator matrix $\hat{G}$ and adding a random error vector e with Hamming weight at most t. 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 $m = \hat{m}S^{-1}$.

2.1 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 [15] 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 [4] the authors 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} which was assumed before. To achieve 80-bit, 128-bit, and 256-bit security level the authors suggest using [2048,1751,55], [2960,2288,113], and [6624,5129,231] binary Goppa codes, respectively.

Table 1 summarizes all suggested parameters as well as the resulting key sizes for specific security levels. It is very common to give the public key in systematic form as a $(n - k) \times k$ matrix. But all published implementations targeting embedded devices choose to store the full $(n \times k)$ public key. This has the advantage of a smaller secret key, which cannot be stored in external memory. If the public key is non systematic, the matrix S in the secret key is completely random and can be generated at runtime form a small seed. For this reason column four gives the size of non-systematic public keys.

Table 1. Recommended parameters and key sizes for the original McEliece PKC

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)

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.

2.2 Goppa Codes

Goppa codes were introduces by V. D. Goppa in 1970 [9]. 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.

Theorem 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 $\mathbb{F}_p^n$.*

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

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 1. *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 $vdm_{ij} = L_j^i$.*

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

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* over $\mathbb{F}_q$. In case of $q = 2^m$ 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 [14].

Definition 2. *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 $C_{ij} = 1/(z_i - L_j)$.*

Theorem 2. *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. $H_{ij} = 1/(z_i - L_j)$, $0 \leq i < t, 0 \leq j < n$.*

2.3 Dyadic Goppa Codes

In [16] Barreto and Misoczki show 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 their proposal the authors make 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 2. Hence, it is possible to construct parity-check matrices which are in Cauchy and dyadic form, simultaneously.

Definition 3. *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 $\Delta_{ij} = h_{i \oplus j}$, where $\oplus$ is the bitwise exclusive-or. 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. *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 $0 < j < i$.

Algorithm 1 in [16] shows how binary dyadic Goppa codes are constructed. It 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. The algorithm 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 1 in [16] 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 these algorithm 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 $\left(\prod_{i=0}^{\lfloor \lg N \rfloor}(q-2^i)\right)/(\lfloor \lg N \rfloor !)$.

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

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 [16].

3 Scheme Definition of QD-McEliece

The main difference between the original McEliece scheme and the quasi-dyadic variant is the key generation algorithm 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 $l > d$.

The key generation algorithm proceeds as follows. It first runs **Algorithm 1 in [16]** 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

Algorithm 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 1 in [16]** $(2^m, N, t)$, 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 $\sigma_0, \ldots, \sigma_{l-1} \in \mathbb{F}_p$. 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** $K_{pub} = (\hat{G}, t)$, $K_{pr} = (H_{dyad}, L_{dyad}, \eta, G(x), (i_0, \ldots, i_{l-1}), (j_0, \ldots, j_{l-1}), (\sigma_0, \ldots, \sigma_{l-1}))$

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 5. *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.

The encryption algorithm of the QD-McEliece variant is the same as that of the original McEliece cryptosystem. 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.

The decryption algorithm of the QD-McEliece version is essentially the same as that of the classical McEliece cryptosystem. 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. 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.

3.1 Parameter Choice and Key Sizes

For an implementation on an embedded microcontroller the best choice is 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 $\mathbb{F}_{2^{16}}$. But the extension field $\mathbb{F}_{2^8}$ is too small to derive secure subfield subcodes from codes defined over it.

Over the base subfield $\mathbb{F}_2$ of $\mathbb{F}_{2^{16}}$ [16] suggests using the parameters summarized in Table 2.

Table 2. Suggested parameters for McEliece variants based on quasi-dyadic Goppa codes over $\mathbb{F}_2$

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

As the public generator matrix $\hat{G}$ is in systematic form, only its non-trivial part Q of length $n - k = m \cdot t$ has to be stored. This part consists of $m(l - m)$ dyadic submatrices of size $t \times t$ each. Storing only the t-length signatures of Q, 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 even in systematic form is $(n - k) \cdot k$ bits in size.

3.2 Security of QD-McEliece

A recent work [7] 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 Groebner bases [1]. 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 with $m = 16$. Hence, up to now the McEliece variant using subfield subcodes over the base field of large codes over $\mathbb{F}_{2^{16}}$ is still secure.

3.3 Conversions for CCA2-Secure McEliece Variants

In [13] Kobara and Imai considered conversions for achieving 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 [19] and Fujisaki-Okamoto's generic conversion [8]. 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 (α, β, γ) decreasing data overhead of the generic conversions even below the values of the original McEliece PKCs for large parameters. Their work 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 work 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.

4 Implementational Aspects

In this section 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 a security level of 80 bit. Target platform is the ATxmega256A1, a RISC microcontroller frequently used in embedded systems. This microcontroller operates at a clock frequency of up to 32 MHz, provides 16 Kbytes SRAM and 256 Kbytes Flash memory.

4.1 Field Arithmetic

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 of 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 are proposed in [2], [17], and [18]. It is possible to view the field $\mathbb{F}_{2^{2k}}$ as a field extension of degree 2 over $\mathbb{F}_{2^k}$. 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

[1] A PTOWF is a function $F(x, y) \to z$ 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.

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 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 [18]. 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.

4.2 Implementation of the QD-McEliece Variant

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.

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

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 do not 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 as $h_{i,j} = h_{i \oplus j}$ 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.

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 $L^* \subset \mathbb{F}_{2^{16}}$. 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 $8 \cdot 16$ 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 $W(x) = W_0 + \sum_{i=0}^{5} W_{2^i} x^{2^i}$. $W(x)$ occupies $7 \cdot 16$ bits storage space.

Syndrome Computation. The first step of the decoding algorithm is the syndrome computation. 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

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

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 1, and thus, reduce the number of operations needed for the syndrome computation. 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 $\hat{H}^T = [Q^T | I_{n-k}]^T$ 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 4.2) 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 syndrome for the efficiently decodable code the vector s' first has to be multiplied by a scrambling matrix S. 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. The next step, which is computing the error locator polynomial $\sigma(x)$, is implemented straightforward using Patterson's algorithm.

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 [3] which is known to be one of 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, because of the required computation of traces and gcds. The next root finding method we analyzed is the Chien search [5] which has a theoretical complexity of $\mathcal{O}(n \cdot t)$ 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 [12]. 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. After a root L_i^* of $\sigma(x)$ has been found we perform the polynomial division of $\sigma(x)$ by $(x - L_i^*)$. 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 searched polynomial $y(x)$. 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.

4.3 Implementation of the KIC-γ

For the implementation of Kobara-Imai's specific conversion γ [13] two parameters have to be chosen: the length of the random value r and the length of the public constant $Const$. The length of r should be equal to the output length of the used hash function. Here we choose the Blue Midnight Wish (BMW) hash function, because of the availability of a fast assembly implementation. As we have $|r| = 256$ and $|Const| = 160$, the message to be encrypted should be of the length $|m| \geq \lfloor \log_2 \binom{n}{t} \rfloor + k + |r| - |Const| = 1281$ 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: $1288/2304 \leq 1280/2304$.

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 $1288 + 160 = 1448$ bits = 181 bytes value y_1. 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. $k = 1280$ bits from $(y_2||Y_1)$ are used as input for McEliece and from the remaining 424 bits the error vector is constructed by the constant weight encoding function $Conv$[22,11].

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 $\delta_j = i_j - i_{j-1} - 1$ where i_r denote the error positions. To obtain part y_4 from the error vector constant weight decoding function is used. Now $(y_2||y_1) = (y_5||y_4||y_3)$ is known and the message m can be obtained.

5 Results

This section 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 Kobara-Imai's specific conversion γ developed for CCA2-secure McEliece variants. Due to the parameters chosen for KIC-γ the actual length of the message to be encrypted increases to 1288 bytes while the ciphertext length increases to 2312 bytes. Table 3 summarizes the sizes of all parameters being precomputed and used for the encryption and decryption algorithms.

Table 3. Sizes of tables and values in memory

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

Except for the matrix S which is 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 AVR Studio in version 4.18. Table 4 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 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.

Table 4. Performance of the QD-McEliece encryption including KIC-γ on the AVR μC ATxmega256@32 MHz

Operation	Sub-operation	Clock cycles
Hash		15,083
CWencoding		50,667
Other		8,927
QD-McEliece encryption	Vector-matrix multiplication	6,279,662
	Add error vector	4,613

Table 5. Performance of the QD-McEliece decryption on the AVR μC ATxmega256@ 32 MHz

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
CWdecoding		61,479
Hash		15,111
Other		19,785

Table 5 presents the results of the operations and sub-operations of the QD-McEliece decryption function including KIC-γ.

Table 5 shows clearly that the error correction using the Horner scheme with polynomial division (PD) is about 40% faster then the Horner scheme without polynomial division. Considering the fact that the error correction is one of the most computationally 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[2]. 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 ciphertext 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.

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

Table 6 summarizes the resource requirements of our implementation. The third column of the table refers 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 McEliece version based on [2048,1751,55]-Goppa codes [6].

Table 6. Resource requirements of QD-McEliece on the AVR μC ATxmega256@ 32 MHz

	Operation	**Flash memory**	**External memory**
QD-McEliece with KIC-γ	Encryption	11 Kbyte	–
	Decryption (with S)	156 Kbyte	–
	Decryption (on-the-fly)	21 Kbyte	–
McEliece[6]	Encryption	684 byte	438 Kbyte
	Decryption	130.4 Kbyte	–

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 7 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 [10] 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 timings at lower frequencies accordingly.

Although we additionally include KIC-γ in the quasi-dyadic McEliece encryption, we were able to out perform 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 out perform 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 $\mathbb{F}_{2^{16}}$.

Table 7. 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

Method	**Time** 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
McEliece encryption [6]	0.4501	3889
McEliece decryption [6]	0.6172	2835
ECC-160 [10]	0.2025	790
RSA-1024 $2^{16}+1$ [10]	0.1075	9525
RSA-1024 w. CRT [10]	2.7475	373

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.

6 Conclusion and Further Research

In this work 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 Kobara-Imai's specific conversion γ: a conversion scheme developed specially for CCA2 secure McEliece variants.

Our implementation out performs 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 original 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 4.2 that none of the frequently used error correction algorithms, such as the Berlekamp trace algorithm and the Chien search, are 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. Nevertheless, the decryption algorithms with precomputation and on-the-fly computation are 2.6 and 1.8 times 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.

Acknowledgement. I would like to thank Olga Paustjan and Paulo Barreto for fruitful discussions. Special thanks to an anonymous reviewer for many useful comments.

References

1. Adams, W., Loustaunau, P.: An Introduction to Gröbner Bases, vol. 3 (1994)
2. Afanasyev, V.B.: On the complexity of finite field arithmetic. In: Fifth Joint Soviet-Swedish Intern. Workshop Information Theory, pp. 9–12 (January 1991)
3. Berlekamp, E.R.: Factoring polynomials over large finite fields. Mathematics of Computation 24(111), 713–715 (1970)
4. Bernstein, D.J., Lange, T., Peters, C.: Attacking and Defending the McEliece Cryptosystem. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 31–46. Springer, Heidelberg (2008)
5. Chien, R.: Cyclic decoding procedure for the bose-chaudhuri-hocquenghem codes. IEEE Transactions on Information Theory IT-10(10), 357–363 (1964)
6. Eisenbarth, T., Güneysu, T., Heyse, S., Paar, C.: MicroEliece: McEliece for Embedded Devices. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 49–64. Springer, Heidelberg (2009)
7. Faugère, J.-C., Otmani, A., Perret, L., Tillich, J.-P.: Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 279–298. Springer, Heidelberg (2010)
8. Fujisaki, E., Okamoto, T.: Secure Integration of Asymmetric and Symmetric Encryption Schemes. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 537–554. Springer, Heidelberg (1999)
9. Goppa, V.D.: A New Class of Linear Correcting Codes. Probl. Pered. Info. 6(3), 24–30 (1970)

10. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-Bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
11. Heyse, S.: Low-Reiter: Niederreiter Encryption Scheme for Embedded Microcontrollers. In: Sendrier, N. (ed.) PQCrypto 2010. LNCS, vol. 6061, pp. 165–181. Springer, Heidelberg (2010)
12. Horner, W.G.: A new method of solving numerical equations of all orders, by continuous approximation. Philosophical Transactions of the Royal Society of London 109, 308–335 (1981)
13. Kobara, K., Imai, H.: Semantically Secure McEliece Public-key Cryptosystems-conversions for McEliece PKC. In: Kim, K.-c. (ed.) PKC 2001. LNCS, vol. 1992, pp. 19–35. Springer, Heidelberg (2001)
14. MacWilliams, F.J., Sloane, N.: The Theory of Error-Correcting Codes. North-Holland Mathematical Library, vol. 16 (1997)
15. McEliece, R.J.: A public-key cryptosystem based on algebraic coding theory. DSN Progress Report 42-44, Jet Propulsion Laboratory (January-February 1978)
16. Misoczki, R., Barreto, P.S.: Compact McEliece Keys from Goppa Codes. In: Jacobson Jr., M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 376–392. Springer, Heidelberg (2009)
17. Morii, M., Kasahara, M.: Efficient construction of gate circuit for computing multiplicative inverses over $gf(2^m)$. Transactions of the IEICE E72, 37–42 (1989)
18. Paar, C.: Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. Dissertation, Institute for Experimental Mathematics, Universität Essen (1994)
19. Pointcheval, D.: Chosen-Ciphertext Security for Any One-Way Cryptosystem. In: Imai, H., Zheng, Y. (eds.) PKC 2000. LNCS, vol. 1751, pp. 129–146. Springer, Heidelberg (2000)
20. Preneel, B., Bosselaers, A., Govaerts, R., Vandewalle, J.: 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, pp. 119–126. Springer, Heidelberg (1992)
21. Prometheus. Implementation of McEliece cryptosystem for 32-bit microprocessors (c-source), `http://www.eccpage.com/`
22. Sendrier, N.: Encoding information into constant weight words. In: IEEE Conference, ISIT 2005, pp. 435–438 (September 2005)
23. Shor, P.W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput. 26(5), 1484–1509 (1997)