Smaller Keys for Code-based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices

Stefan Heyse, Ingo von Maurich, Tim Güneysu

Horst Görtz Institute for IT-Security
Ruhr-Universität Bochum
Bochum, Germany
{stefan.heyse,ingo.vonmaurich,tim.gueneysu}@rub.de

Abstract. In the last years code-based cryptosystems were established as promising alternatives for asymmetric cryptography since they base their security on well-known NP-hard problems and still show decent performance on a wide range of computing platforms. The main drawback of code-based schemes, including the popular proposals by McEliece and Niederreiter, are the large keys whose size is inherently determined by the underlying code. In a very recent approach, Misoczki et al. proposed to use quasi-cyclic MDPC (QC-MDPC) codes that allow for a very compact key representation. In this work, we investigate novel implementations of the McEliece scheme using such QC-MDPC codes tailored for embedded devices, namely a Xilinx Virtex-6 FPGA and an 8-bit AVR microcontroller. In particular, we evaluate and improve different approaches to decode QC-MDPC codes. Besides competitive performance for encryption and decryption on the FPGA, we achieved a very compact implementation on the microcontroller using only 4,800 and 9,600 bits for the public and secret key at 80 bits of equivalent symmetric security.

Keywords: MDPC, LDPC, FPGA, microcontroller, McEliece, code-based, public key cryptography

1 Introduction

Nearly all established asymmetric cryptosystems rely on two classes of fundamental problems, namely the factoring problem and the (elliptic curve) discrete logarithm problem. Due to Shor's [37] efficient algorithm which solves both problems on quantum computers, it has become evident that a larger *diversification* of public key primitives is urgently required to be prepared in case quantum computers enter the scene. In this context, IBM announced two improvements in quantum computing [11] and estimates that such systems might become practical and available within the next 15 years.

The most promising alternatives are currently classified into code-based, lattice-based, multivariate-quadratic ($\mathcal{MQ}$-), and hash-based cryptography. A major drawback of many proposed cryptosystems within these classes are their low efficiency and practicability due to large key sizes or complex computations compared to classical RSA and ECC cryptosystems. This is particularly considered an issue for small and embedded systems where memory and processing power are a scarce resource. Nevertheless, it was shown that code-based cryptosystems such as the well-established proposals by McEliece and Niederreiter can significantly outperform classical asymmetric cryptosystems on embedded systems [13, 16, 20, 32] – at the cost of very large keys (often more than 50 kByte). Therefore, current research is targeting alternative codes that allow more compact key representations but still preserve the security properties of the cryptosystem. Recently, Misoczki et al. proposed to use quasi-cyclic medium-density parity check (QC-MDPC) codes as such an alternative [28], claiming that a public key of only 4800 bit can provide a level of 80 bit equivalent symmetric security.

Contribution In this work, we present implementations of the McEliece cryptosystem using QC-MDPC codes for Xilinx FPGAs and AVR microcontrollers. Since decoding is usually the most expensive operation in code-based encryption systems, we particularly focus on evaluations and improvements of different decoders

for QC-MDPC codes and provide implementations for the two embedded platforms under investigation. We show that QC-MDPC codes provide excellent efficiency in terms of computational complexity and key sizes for encryption and decryption on the FPGA and a key size of only 4,800 and 9,600 bit for the public and secret key, respectively. We also show that it is possible to implement QC-MDPC codes with a very small memory footprint on microcontrollers. The source code is available under `http://www.sha.rub.de/research/projects/code/`.

This new McEliece variant has not yet gathered much attention by cryptanalysts. In order to establish the necessary confidence for its deployment in real-world systems we hope to give as early adopters another incentive for further cryptanalysis of this scheme by highlighting the excellent properties of QC-MDPC codes for embedded systems.

Outline This paper is structured as follows: in Section 2 we briefly summarize previous work on code-based public key cryptosystems. Section 3 provides background on MDPC codes, their decoding algorithms, and an introduction to McEliece with QC-MDPC codes. In Section 4 we explain our design considerations and implementations on a Xilinx Virtex-6 FPGA and on a 8-bit AVR microcontroller. Finally, we present and compare our results of both implementations in Section 5 and draw a conclusion in Section 6.

2 Previous Work

Although proposed more than 30 years ago, code-based encryption schemes are hardly found in any (cost-driven) real-world applications due to their large secret and public keys. The original proposal by Robert McEliece for a code-based encryption scheme suggested the use of binary Goppa codes, but in general any other linear code could be used. While other types of codes may have advantages such as a more compact representation, most proposals using different codes were proven less secure (cf. [26, 31]). The Niederreiter cryptosystem is an independently developed variant of McEliece's proposal which is proven to be equivalent in terms of security [25]. In 2009, a first FPGA-based implementation of McEliece's cryptosystem was proposed targeting a Xilinx Spartan-3AN. It encrypts and decrypts data in 1.07 ms and 2.88 ms using security parameters that achieve an equivalence of 80-bit symmetric security [13]. The authors of [39] presented another accelerator for McEliece encryption over binary Goppa codes on a more powerful Virtex5-LX110T, capable to encrypt and decrypt in 0.5 ms and 1.4 ms providing a similar level of security. The latest publication [16] based on a hardware/software co-design for the same Virtex5-LX110T FPGA decrypts a block in 0.5 ms at 190 MHz[1] at the same level of security. For x86-based platforms, a recent implementation of the McEliece scheme over binary Goppa codes is due to Biswas and Sendrier [9] achieving about 83-bit of equivalent symmetric security according to [8].

Many proposals already tried to address the issue of large keys by replacing the originally used binary Goppa codes with (secure) codes that allow more compact representations, e.g, [10, 27]. However, many attempts were broken [14] and for the few survivors hardly any implementations are publicly available [6, 20]. In the context of this work, low density parity check (LDPC) codes [15] have repeatedly been suggested as candidates for McEliece [1–4, 29]. The use of quasi-cyclic LDPC codes was suggested for McEliece in [1] but due to the cryptanalytic results of [2] and [29] in [30], McEliece based on LDPC codes is not considered as a good choice.

Picking up and improving the idea of QC-LDPC codes, medium density parity check (MDPC) codes and a corresponding quasi cyclic variant (QC-MDPC) are introduced in [28]. In particular, the authors claim that (QC-)MDPC codes resist known attacks on LDPC codes and suggest to use such codes in the McEliece public key encryption scheme. To date, neither an attack nor any implementation of cryptography with QC-MDPC codes have been published.

[1] This work does not provide performance results for encryption.

3 Background on MDPC Codes

In the following we introduce (QC-)MDPC codes, closely following the description given in [28]. (QC-)MDPC codes are a special variant of linear codes and are defined as follows:

Definition 1 (Linear codes). *A binary (n, r)-linear code C of length n, dimension $n-r$ and co-dimension r, is a $(n-r)$-dimensional vector subspace of $\mathbb{F}_2^n$. It is spanned by the rows of a matrix $G \in \mathbb{F}_2^{(n-r)\times n}$, called a generator matrix of C. The generator matrix is the kernel of a matrix $H \in \mathbb{F}_2^{r\times n}$ and called the parity-check matrix of C. The codeword $c \in C$ of a vector $m \in \mathbb{F}_2^{(n-r)}$ is given by $c = mG$. Given a vector $e \in \mathbb{F}_2^n$, we obtain the syndrome $s = He^T \in \mathbb{F}_2^r$. The dual $C^\perp$ of C is the linear code spanned by the rows of any parity-check matrix of C.*

A linear code can be quasi-cyclic according to the following definition:

Definition 2 (Quasi-cyclic code). *A (n, r)-linear code is quasi-cyclic (QC) if there is some integer n_0 such that every cyclic shift of a codeword by n_0 positions is again a codeword. When $n = n_0 p$, for some integer p, it is possible and convenient to have both generator and parity check matrices composed by $p \times p$ circulant blocks. A circulant block is completely described by its first row (or column) and the algebra of $p \times p$ binary circulant matrices is isomorphic to the algebra of polynomials modulo $x^p - 1$ in $\mathbb{F}_2$.*

On top we can define the MDPC codes:

Definition 3 (MDPC codes). *A (n, r, w)-MDPC code is a linear code of length n and co-dimension r admitting a parity check matrix with constant row weight w.*

When MDPC codes are quasi-cyclic, they are called (n, r, w)-QC-MDPC codes. LDPC codes typically have small constant row weights (usually, less than 10). For MDPC codes, row weights scaling in $O(\sqrt{n\,log(n)})$ are assumed.

3.1 McEliece Based on QC-MDPC Codes

We now present a variant of the McEliece cryptosystem based on (n, r, w)-QC-MDPC codes with $n = n_0 p$ and $r = p$. To obtain such a code, we first pick a word $h \in \mathbb{F}_2^n$ of length $n = n_0 p$ and weight w at random. Then, the QC-MDPC code is defined by a quasi-cyclic parity-check matrix $H \in \mathbb{F}_2^n$ of first row h and all other $r - 1$ rows are obtained from $r - 1$ quasi-cyclic shifts of h. The parity-check matrix then has the form $H = [H_0|H_1|...|H_{n_0-1}]$. Each block H_i has row weight w_i, such that $w = \sum_{i=0}^{n_0-1} w_i$ with a smooth distribution of w_i's. Finally, the generator matrix G in row reduced echelon form can be easily derived from the H_i blocks. Assuming that H_{n_0-1} is non-singular (this particularly implies w_{n_0-1} being odd, otherwise the rows of H_{n_0-1} would sum up to 0), we compute G of the form $(I|Q)$, where I is the identiy matrix and

$$Q = \begin{pmatrix} (H_{n_0-1}^{-1} \cdot H_0)^T \\ (H_{n_0-1}^{-1} \cdot H_1)^T \\ \cdots \\ (H_{n_0-1}^{-1} \cdot H_{n_0-2})^T \end{pmatrix}.$$

In the following we detail the key-generation as well as encryption and decryption for McEliece based on QC-MDPC codes.

- *Key-Generation:* The public and private keys are generated as follows. First generate a parity-check matrix $H \in \mathbb{F}_2^{r\times n}$ of a t-error-correcting (n, r, w)-QC-MDPC code. Then generate its corresponding generator matrix $G \in \mathbb{F}_2^{(n-r)\times n}$ in row reduced echelon form. The public key is G and the private key is H. Since quasi-cyclic matrices are used, it suffices to store the first rows g and h of the circulant blocks which significantly reduces storage requirements.

– *Encryption:* To encrypt a plaintext $m \in \mathbb{F}_2^{(n-r)}$ into $x \in \mathbb{F}_2^n$, first generate an error vector $e \in \mathbb{F}_2^n$ of $wt(e) \leq t$ at random. Then compute $x \leftarrow mG + e$.

– *Decryption:* Let Ψ_H be a t-error-correcting LDPC/MDPC decoding algorithm equipped with the sparse parity-check matrix H. To decrypt $x \in \mathbb{F}_2^n$ into $m \in \mathbb{F}_2^{(n-r)}$ compute $mG \leftarrow \Psi_H(mG+e)$. Finally extract the plaintext m from the first $(n-r)$ positions of mG.

3.2 Security of QC-MDPC

The description of McEliece based on QC-MDPC codes in Section 3.1 eliminates the scrambling matrix S and the permutation matrix P usually used in the McEliece cryptosystem. The use of a CCA2-secure conversion (e.g., [24]) allows G to be in systematic-form without introducing any security-flaws. Note that [28] states that a quasi-cyclic structure, by itself, does not imply a significant improvement for an adversary. All previous attacks on McEliece schemes are based on the combination of a quasi-cyclic/dyadic structure with some algebraic code information. To resist the best currently known attack of [5] and also the improvements achieved by the DOOM-attack [36], the authors of [28] suggest parameters as given in Table 1.

Table 1. Parameters for different security levels for McEliece with QC-MDPC codes given by [28].

Security Level	$\mathbf{n_0}$	**n**	**r**	**w**	**t**	**Public key size**
80 bit	2	9600	4800	90	84	4800 bit
80 bit	3	10752	3584	153	53	7168 bit
80 bit	4	12288	3072	220	42	9216 bit
128 bit	2	19712	9856	142	134	9856 bit
128 bit	3	22272	7424	243	85	14848 bit
128 bit	4	27200	6800	340	68	20400 bit
256 bit	2	65536	32768	274	264	32768 bit
256 bit	3	67584	22528	465	167	45056 bit
256 bit	4	81920	20480	644	137	61440 bit

3.3 Decoding (QC-)MDPC Codes

For code-based cryptosystems, decoding a codeword (i.e., the syndrome) is usually the most complex task. Decoding algorithms for LDPC/MDPC codes are mainly divided into two families. The first class (e.g., [7]) offers a better error-correction capability but is computationally more complex than the second family. Especially when handling large codes, the second family, called bit-flipping algorithms [15], seems to be more appropriate. In general, they are all based on the following principle:

1. Compute the syndrome s of the received codeword x.
2. Check the number of unsatisfied parity-check-equations $\#_{upc}$ associated with each codeword bit.
3. Flip each codeword bit that violates more than b equations.

This process is iterated until either the syndrome becomes zero or a predefined maximum number of iterations is reached. In that case a decoding error is returned. The main difference of the bit-flipping algorithms is how the threshold b is computed. In the original algorithm of Gallager [15], a new b is computed at each iteration. In [22], b is taken as the maximum of the unsatisfied parity-check-equations Max_{upc} and the authors of the QC-MDPC scheme propose to use $b = Max_{upc} - \delta$, for some small δ.

Since estimating the error-correction capability of LDPC and MDPC codes generally is a hard task and is also influenced by the choice of threshold b, we derive different versions of the bit-flipping algorithm, evaluate their error-correcting capability and count how many iterations are required on average to decode a codeword. Because we are targeting embedded systems, we omit the variant storing n_0 counters for $\#_{upc}$ for each ciphertext bit. This would allow to skip the second computation of $\#_{upc}$ in some variants, but would blow up memory consumption to an unacceptable amount. We now introduce the different decoders under investigation:

Decoder $\mathcal{A}$ is given in [28] and computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum Max_{upc} and afterwards a second time to flip all codeword bits that violate $b \geq Max_{upc} - \delta$ equations. Afterwards the syndrome is recomputed and compared to zero.

Decoder $\mathcal{B}$ is given in [15] and computes the syndrome, then checks the number of unsatisfied parity-check-equations once per iteration i and directly flips the current codeword bit if $\#_{upc}$ is larger than a precomputed threshold b_i. Afterwards the syndrome is recomputed and compared to zero.

We noticed that the previously proposed bit-flipping decoders recompute the syndrome after every iteration. Since this is quite costly we propose an optimization based on the following observation: If the amount of unsatisfied parity-check-equations exceeds threshold b, the corresponding bit in the codeword is flipped and the syndrome changes. We would like to stress that the syndrome does not change arbitrarily, but the new syndrome is equal to the old syndrome accumulated with the row h_j of the parity check matrix that corresponds to the flipped codeword bit j. By keeping track of which codeword bits are flipped and updating the syndrome accordingly, the syndrome recomputation can be omitted. Hence, we propose and evaluate the following decoders:

Decoder $\mathcal{C}_1$ computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum Max_{upc} and afterwards a second time to flip all codeword bits that violate $b \geq Max_{upc} - \delta$ equations. If a codeword bit j is flipped, the corresponding row h_j of the parity check matrix is added to a *temporary syndrome*. At the end of each iteration the temporary syndrome is added to the syndrome, directly resulting in the syndrome of the new codeword without requiring a full recomputation.

Decoder $\mathcal{C}_2$ computes the syndrome, then checks the number of unsatisfied parity-check-equations once to compute the maximum Max_{upc} and afterwards a second time to flip all codeword bits that violate $b \geq Max_{upc} - \delta$ equations. If a codeword bit j is flipped, the corresponding row h_j of the parity check matrix is added *directly* to the current syndrome. Using this method we always work with an up-to-date syndrome and not with the one from the last iteration.

Decoder $\mathcal{D}$ is similar to Decoder $\mathcal{B}$ with precomputed thresholds b_i, but uses the *direct* update of the syndrome as done in Decoder $\mathcal{C}_2$.

Decoder $\mathcal{E}$ is similar to Decoder $\mathcal{C}_2$ but compares the syndrome to zero after each flipped bit and aborts the current bit-flipping iteration immediately if the syndrome becomes zero.

Decoder $\mathcal{F}$ is similar to Decoder $\mathcal{D}$ and in addition uses the same early exit trick as Decoder $\mathcal{E}$.

The average number of iterations required to decode a codeword and the decoding failure rate for the different decoders with different numbers of errors are shown in Table 6 in the appendix for a QC-MDPC code with parameters $n_0 = 2, n = 9600, r = 4800, w = 90$ (cf. first row of Table 1). All measurements are taken for 1000 random codes and 100,000 random decoding tries per code on a Intel Xeon E5345 CPU running at 2.33 GHz. For versions with precomputed thresholds b_i we used the formula given in Appendix A

of [28] to precompute the most suitable b_i's for every iteration. For versions using $b = Max_{upc} - \delta$, we found by exhaustive experiments that the smallest number of iterations are required for $\delta = 5$[2]. A decoding failure is returned when the decoder did not succeed within ten iterations.

The timings given in Table 6 should only be used to compare the decoders among each other. The evaluation was done in software and is not optimized for speed. It is designed to keep only the generating polynomial h and not the whole parity check matrix H in memory which would allow for a time/memory trade-off and faster computations. The corresponding row is derived at runtime by rotating the polynomial.

Our evaluations clearly show the superior error correcting capability of decoders $\mathcal{D}$ and $\mathcal{F}$ which in addition require the lowest number of iterations when compared to the other decoders (cf. Table 6). Decoders $\mathcal{A}$ and $\mathcal{C}_1$ are least efficient with an average of more than 5 bit-flipping iterations. Our new decoders $\mathcal{D}$ and $\mathcal{F}$ on average save 2.9 iterations compared to decoder $\mathcal{A}$ and 0.7 iterations compared to $\mathcal{B}$. This directly relates to the required time for decoding which is up to 4 times faster.

The small timing advantage of decoder $\mathcal{F}$ over $\mathcal{D}$ is due to the immediate termination if the syndrome becomes zero. Another interesting observation we made for all decoders is that if a codeword is decodable, then this is achieved after a small number of iterations. We noticed that if a codeword is not decoded within 4-6 iterations, a higher number of iterations does not lead to a successful decoding. Therefore, a early detection of a decoding failure is possible.

4 Implementation

In this section we discuss decoder and parameter selections and reason design choices for our QC-MDPC McEliece implementations on reconfigurable hardware and microcontrollers. The primary goal for the hardware design is high-performance while the microcontroller implementation aims for a low memory footprint. Note, the implementations of a CCA2-secure conversion and true random number generation are out of the scope of this work.

4.1 Decoder and Parameter Selection

Our implementations aim for a security level of 80 bit, comparable to ECC-160 and RSA-1024. Hence, we select the following QC-MDPC code parameters that provide a 80-bit security level according to Table 1.

$$n_0 = 2, n = 9600, r = 4800, w = 90, t = 84$$

Using these parameters we have a 4800-bit public key and a 9600-bit sparse secret key with 90 set bits. Such key sizes are only a fraction of the key sizes of other code-based public-key encryption schemes. During encryption a 4800-bit plaintext is encoded into a 9600-bit codeword and 84 errors are added to it. It follows from $n_0 = 2$ that the 9600-bit codeword and secret key consist of two separate 4800-bit codewords/secret keys, respectively.

As shown in Section 3 our decoders $\mathcal{D}$ and $\mathcal{F}$ require only one syndrome computation in the beginning and update the syndrome directly in the bit-flipping step. Furthermore, due to the precomputed thresholds b_i the computation of the maximum number of unsatisfied parity check equations can be omitted. The decoders only differ in the way they handle the part where they check if the syndrome is zero. While decoder $\mathcal{F}$ checks the syndrome every time the syndrome is change in the bit-flipping step, decoder $\mathcal{D}$ tests the syndrome at the end of each bit-flipping iteration. Note, the decoding behavior of both decoders is the same, i.e., they require the same amount of bit-flipping iterations with the difference that decoder $\mathcal{F}$ exits as soon as the syndrome is equal to zero.

We base our QC-MDPC McEliece decryption implementation on decoder $\mathcal{D}$ in hardware and on decoder $\mathcal{F}$ for the microcontroller. The reason for choosing decoder $\mathcal{D}$ to be implemented in hardware is that we sequentially rotate the codewords and secret keys in every cycle of the bit-flipping iterations. If the syndrome

[2] In the latest version of [28] the authors also suggest to use $\delta \approx 5$ for the given parameters.

becomes zero during a bit-flipping iteration and we skip further computations immediately, the secret polynomials and the codewords would be misaligned. To fix this we would have to rotate them manually into their correct position which would take roughly the same amount of time as just letting the decoder finish the current iteration.

Both implementations use a maximum of five iterations before returning a decoding error and the corresponding precomputed b_i are $(28, 26, 24, 22, 20)$, which are computed using the formula in the appendix of [28].

4.2 FPGA Implementation

For our evaluation of QC-MDPC in reconfigurable hardware we use Xilinx's Virtex-6 FPGA device family as target platform. Virtex-6 devices are powerful FPGAs offering thousands of slices, where each slice contains four 6-input lookup tables (LUT), eight flip-flops (FF), and surrounding logic. In addition, embedded resources such as block memories (BRAM) and digital signal processors (DSP) are available. In the following we reason our design choices and describe the implementations of the QC-MDPC-based McEliece en- and decryption.

Design Considerations Because of their relatively small size, the public and secret key do not have to be stored in external memory as it was necessary in earlier FPGA implementations of McEliece and Niederreiter using, e.g., Goppa codes. Since we aim for high-speed, we store all operands directly in FPGA logic and refrain from loading/storing them from/to internal block memories or other external memory as this would affect performance. Reading a single 4800-bit vector from a 32-bit BRAM interface would consume 150 clock cycles. However, if maximum performance is not required, the use of BRAMs could certainly reduce resource consumption significantly.

In contrast to the microcontroller implementation we do not exploit the sparsity of the secret polynomials in our FPGA design. Using a sparse representation of the secret polynomials would require to implement $w = 90$ counters with 13 bits, each indicating the position of a set bit in one of the two secret polynomials. To generate the next row of the secret key, all counters have to be increased and in case of exceeding 4799 they have be set to 0. If a bit in the codewords x_0 or x_1 is set we have to build a 4800-bit vector from the counters belonging to the corresponding secret polynomial and XOR this vector to the current syndrome. The alternative is to read out the content of each counter belonging to the corresponding secret polynomial and flip the corresponding bit in the syndrome. These tasks, however, are time and/or resource consuming in hardware.

Implementation We use a Virtex-6 XC6VLX240T FPGA as target device for a fair comparison with previous work – although all our implementations would fit smaller devices as well.

The encryption and decryption unit are equipped with a simple I/O interface. Messages and codewords are send and received bit by bit to keep the I/O overhead of our implementation small and thus get as close as possible to the actual resource consumptions of the en-/decoder.

QC-MDPC Encryption: In order to implement a QC-MDPC encoder we need a vector matrix multiplication to multiply message m with the public key matrix G to retrieve a codeword $c = mG$ and then add an error vector with $hw(e) \leq 84$ to get the ciphertext $x = c + e$. We are given a 4800-bit public key g which is the first row of matrix G. Rotating g by one bit position yields the next row of G and so forth. Since G is of systematic form the first half of c is equal to m. The second half, called redundant part, is computed as follows.

We iterate over the message bit by bit and XOR the current public polynomial to the redundant part if the current message bit is set. To implement this in hardware we need three 4800-bit registers to hold the public polynomial, the message, and the redundant part. Since only one bit of the message has to be accessed in every clock cycle, we store the message in a circulant shift register which can be implemented using shift register LUTs.

QC-MDPC Decryption: Decryption is performed by decoding the received ciphertext, the first half of the decoded codeword is the plaintext. As QC-MDPC decoder we implement the bit-flipping decoder $\mathcal{D}$ as described in Section 3.3. In the first step we need to compute the syndrome $s = Hx^T$ by multiplying parity check matrix $H = [H_0|H_1]$ with the ciphertext x. Given the first 9600-bit row $h = [h_0|h_1]$ of H and the 9600-bit codeword $x = [x_0|x_1]$ we compute the syndrome as follows. We sequentially iterate over every bit of the codewords x_0 and x_1 in parallel and rotate h by rotating h_0 and h_1 accordingly. If a bit in x_0 and/or x_1 is set, we XOR the current h_0 and/or h_1 to the intermediate syndrome which is set to zero in the beginning. The syndrome computation is finished after every bit of the ciphertext has been processed.

Next we need to check if the syndrome is zero. We implement this as a logical OR tree. Since the FPGA offers 6-input LUTs, we split the syndrome into 6-bit chunks and compute their logical OR on the lowest level of the tree. The results are fed into the next level of 6-bit LUTs which again compute the logical OR of the inputs. This is repeated until we are left with a single bit that indicates if the syndrome is zero or not. In addition, we add registers after the second layer of the tree to minimize the critical path.

If the syndrome is zero, the decryption is finished. Otherwise we have to compute the number of unsatisfied parity check equations for each row $h = [h_0|h_1]$. We therefore compute the hamming weight of the logical AND of the syndrome and h_0 and h_1, respectively. If the hamming weight exceeds the threshold b_i for the current iteration i, the corresponding bit in the codeword x_0 and/or x_1 is flipped and the syndrome is directly updated by XORing the current secret polynomial h_0 and/or h_1 to it. Then h_0 and h_1 are rotated by one bit and the process is repeated until all rows of H have been checked.

Since the computation of the number of unsatisfied parity check equations for h_0 and h_1 can be performed independently, we have two options for implementation. Either we compute the parity check violations of the first and second secret polynomial iteratively or we instantiate two hamming weight computation units and process the polynomials in parallel. The iterative version will take twice the time but using less resources. We explore both version to evaluate this time/resource trade-off.

Computing the hamming weight of a 4800-bit vector efficiently is a challenge of its own. Similar to the zero comparator we split the input into 6-bit chunks and determine their hamming weight. We then compute the overall hamming weight by building an adder tree with registers on every layer to minimize the critical path. After all rows of H have been processed, the syndrome is again compared to zero. If the syndrome is zero, the first 4800-bit of the updated codeword (i.e. x_0) are equal to the decoded message m and are returned. Otherwise the bit-flipping is repeated with the next b_i until either the syndrome becomes zero or the maximum number of iterations is exceeded.

4.3 Microcontroller Implementation

As implementation platform we choose a ATxmega256A3 microcontroller for straightforward comparison with previous work. The microcontroller provides 16 kByte SRAM and 256 kByte program memory and can be clocked at up to 32 MHz. The main parts are written in C and we pay careful attention to implement timing critical routines as, e.g., the polynomial rotation and addition using inline assembly.

The encoding operation is straightforward. Since G is of systematic form, the first r ciphertext bits are the message itself and are simply copied. For the multiplication with the redundant part Q, the message bits are parsed and the corresponding rows of G are summed up. Afterwards the current row is rotated by one bit-position to generate the next row. We implemented two different version of the encoder which differ in the way the public polynomial rotation is implemented. In one version we use a loop to rotate the byte of the public polynomial and in the other version we unroll this process.

Usually, smartcard devices communicate over a very slow interface, e.g., 106 kByte/s [40]. In contrast to cryptosystems such as RSA and ECC, we do not need the message as a whole to start with the encryption. Therefore, an interesting option is to directly encode a byte of the message as soon as it arrives while the next message byte is still in transfer. To some extend, this allows to hide the computation time within the latency required to transfer the message.

For decoding, recall that the $n_0 = 2$ involved secret polynomials are sparse and only 45 out of 4800 bits are set. Instead of saving 4800 coefficients in $\frac{4800}{8} = 600$ bytes, it is sufficient to save the indices of the

$w_i = 45$ bits that are set. Each secret polynomial therefore requires only $\lceil \log_2(4800)/8 \rceil \cdot 45 = 2 \cdot 45 = 90$ bytes. Additionally, rotating a polynomial by one bit-position means incrementing the 45 indices by one and handling the overflow from x^{4800} to x^0. We developed a vector-(sparse-matrix) multiplication, which adds a sparse row to the syndrome by flipping the 45 indexed bits in the 4800 bit syndrome. Also the update of the syndrome can be handled this way when a ciphertext bit is flipped. In order to keep the memory consumption low while still achieving good performance we use decoder $\mathcal{F}$, as described in Section 3. Since we store the bit-position in counters, an early exit of the decoding phase can be implemented – unlike to our hardware implementation. The complete secret key therefore requires only $2 \cdot (2 \cdot 45)$ bytes for the secret polynomials and additionally ten bytes for the precomputed thresholds b_i.

Note that the precomputed thresholds b_i can be treated as public system parameter. In contrast to the encoding process, every ciphertext byte is accessed multiple times during decoding so that the "process-while-transfer"-method described above is not applicable. Also note that during decoding no additional memory is required to store the plaintext as the first half of the ciphertext is equal to the plaintext after successful decoding.

5 Results

In the following we present our QC-MDPC implementation results in reconfigurable hardware and in software on a 8-bit microcontroller. Afterwards we give an overview of existing public key encryption implementations for similar platforms and compare them to our results.

5.1 FPGA Results

All our results are obtained post place-and-route (PAR) for a Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 14.5. For the throughput figures we assume a fast enough I/O interface is provided.

In hardware, our QC-MDPC encoder runs at 351.3 MHz and encodes a 4800-bit message in 4800 clock cycles which results in 351.3 Mbit/s. The iterative version of our QC-MDPC decoder runs at 222.5 MHz. Since the decoder does not run in constant time, we calculate the average required cycles for iterative decoding as follows. Computing the syndrome for the first time needs 4800 clock cycles and comparing the syndrome to zero takes another 2 clock cycles. For every following bit-flipping iteration we need 9620 plus again 2 clock cycles for checking the syndrome. As shown in Table 6, decoder $\mathcal{D}$ needs 2.4002 bit-flipping iterations on average. Thus, the average cycle count for our iterative decoder is $4800 + 2 + 2.4002 \cdot (9620 + 2) = 27896.7$ clock cycles.

Our non-iterative decoder processes both secret polynomials in the bit-flipping step in parallel and runs at 190.6 MHz. We calculate the average cycles as before with the difference that every bit-flipping iteration now takes $4810 + 2$ clock cycles. Thus, the average cycle count for our non-iterative decoder is $4800 + 2 + 2.4002 \cdot (4810 + 2) = 16351.8$ clock cycles.

The non-iterative decoder operates 46% faster than the iterative version while occupying 40-65% more resources. Compared to the decoders, the encoder runs 6-9 times faster and occupies 2-6 times less resources. Table 2 summarizes our results.

Using the formerly proposed decoders that work without our syndrome computation optimizations (i.e., decoders $\mathcal{A}$ and $\mathcal{B}$) would result in much slower decryptions. Decoder $\mathcal{A}$ would need $4802 + 5.2964 \cdot (2 \cdot 9620 + 4802) = 132138.0$ cycles in an iterative and $4802 + 5.2964 \cdot (2 \cdot 4810 + 4802) = 81186.7$ cycles in a non-iterative implementation. Decoder $\mathcal{B}$ saves cycles by skipping the Max_{upc} computation but would still need $4802 + 3.1425 \cdot (9620 + 4802) = 50123.1$ cycles in an iterative and $4802 + 3.1425 \cdot (4810 + 4802) = 35007.7$ cycles in a non-iterative implementation.

Comparison A comparison with previously published FPGA implementations of code-based (McEliece, Niederreiter), lattice-based (Ring-LWE, NTRU), and standard public key encryption schemes (RSA, ECC) is given in Table 3. The most relevant metric for comparing the performance of public key encryption schemes

Table 2. Implementation results of our QC-MDPC implementations with parameters $n_0 = 2, n = 9600, r = 4800, w = 90, t = 84$ on a Xilinx Virtex-6 XC6VLX240T FPGA.

Aspect	**Encoder**	**Decoder (iterative)**	**Decoder (non-iterative)**
FFs	14,426 (4%)	32,974 (10%)	46,515 (15%)
LUTs	8,856 (5%)	36,554 (24%)	46,249 (30%)
Slices	2,920 (7%)	10,271 (27%)	17,120 (45%)
Frequency	351.3 MHz	222.5 MHz	190.6 MHz
Time/Op	13.66 µs	125.38 µs	85.79 µs
Throughput	351.3 Mbit/s	38.3 Mbit/s	55.9 Mbit/s
Encode	4,800 cycles	-	-
Compute Syndrome	-	4,800 cycles	4,800 cycles
Check Zero	-	2 cycles	2 cycles
Flip Bits	-	9,620 cycles	4,810 cycles
Overall average	4,800 cycles	27,896.7 cycles	16,351.8 cycles

often depends on the application. For key exchange it is the required time per operation, given the symmetric key size is smaller or equal to the bit size that can be transmitted in one operation. For data encryption (i.e., much more than one block), throughput in Mbit/s is typically the most interesting metric.

A hardware McEliece implementation based on Goppa codes including CCA2 conversion was presented for a Virtex5-LX110T FPGA in [38, 39]. Comparing their performance to our implementations shows the advantage of QC-MDPC McEliece in both time per operation and Mbit/s. The occupied resources are similar to our resource requirements but in addition 75 block memories are required for storage. Even more important for real-world applications is the public key size. QC-MDPC McEliece requires 0.59 kByte which is only a fraction of the 100.5 kByte public key of [38].

A McEliece co-processor was recently proposed for a Virtex5-LX110T FPGA [16]. Their design goal was to optimize the speed/area ratio while we aim for high performance. With respect to decoding performance, our implementations outperform their work in both time/operation and Mbit/s. But the co-processor needs much less resources and can also be implemented on low-cost devices such as Spartan-3 FPGAs. The public keys in this work have a size of 63.5 kByte which is still much larger than the 0.59 kByte of QC-MDPC McEliece.

The Niederreiter public key scheme was implemented in [21] for a Virtex6-LX240T FPGA. The work shows that Niederreiter encryption can provide high performance with a moderate amount of resources. Decryption is more expensive both in computation time as well as in required resources. The Niederreiter encryption is the superior choice for a minimum time per operation, but concerning raw throughput QC-MDPC achieves better results. Furthermore, the public key with 63.5 kByte of the Niederreiter encryption using binary Goppa codes might be to large for real-world applications.

FPGA implementations of lattice-based public key encryption were proposed in [17] for Ring-LWE and in [23] for NTRU. The Ring-LWE implementation requires a huge amount of resources (in particular, exceeding the resources provided by their Virtex6-LX240T FPGA). On the other hand, NTRU as implemented in [23] shows that lattice-based cryptography can provide high performance at moderate resources requirements. Note further that the results are reported for an outdated Virtex-E FPGA which is hardly comparable to modern Virtex-5/-6 devices.

Efficient ECC hardware implementations for curves over $GF(p)$ and $GF(2^m)$ are [12, 18, 34, 35] which all yield good performance at moderate resource requirements. The most efficient RSA hardware implementation to date was proposed in [42, 41]. Both the time to encrypt and decrypt one block as well as the throughput are considerably worse than QC-MDPC McEliece.

Table 3. Performance comparison of our QC-MDPC FPGA implementations with other public key encryption schemes. [1]Occupied slices and BRAMs are only given for encryption and decryption combined. [2]Calculated from synthesis results of a over-mapped device, post-PAR results are not given and will most likely be much slower. [3]Additionally uses 26 DSP48s. [4]Additionally uses 17 DSP48s.

Scheme	Platform	f [MHz]	Bits	Time/Op	Cycles	Mbit/s	FFs	LUTs	Slices	BRAM
This work (enc)	XC6VLX240T	351.3	4,800	13.66 µs	4,800	351.3	14,426	8,856	2,920	0
This work (dec)	XC6VLX240T	190.6	4,800	85.79 µs	16,352	55.9	46,515	46,249	17,120	0
This work (dec iter.)	XC6VLX240T	222.5	4,800	125.38 µs	27,897	38.3	32,974	36,554	10,271	0
McEliece (enc) [38]	XC5VLX110T	163	512	500 µs	n/a	1.0	n/a	n/a	14,537	75[1]
McEliece (dec) [38]	XC5VLX110T	163	512	1,290 µs	n/a	0.4	n/a	n/a	14,537	75[1]
McEliece (dec) [16]	XC5VLX110T	190	1,751	500 µs	94,249	3.5	n/a	n/a	1,385	5
Niederreiter (enc) [21]	XC6VLX240T	300	192	0.66 µs	200	290.9	875	926	315	17
Niederreiter (dec) [21]	XC6VLX240T	250	192	58.78 µs	14,500	3.3	12,861	9,409	3,887	9
Ring-LWE (enc) [17]	XC6VLX240T	n/a	256	8.10 µs	n/a	15.8	143,396	298,016	n/a	0[2]
Ring-LWE (dec) [17]	XC6VLX240T	n/a	256	8.15 µs	n/a	15.7	65,174	124,158	n/a	0[2]
NTRU (enc/dec) [23]	XCV1600E	62.3	251	1.54/1.41 µs	96/88	163/178	5,160	27,292	14,352	0
ECC-P224 [18]	XC4VFX12	487	224	365.10 µs	177,755	0.61	1,892	1,825	1,580	11[3]
ECC-163 [34]	XC5VLX85T	167	163	8.60 µs	1436	18.9	n/a	10,176	3,446	0
ECC-163 [35]	Virtex-4	45.5	163	12.10 µs	552	13.4	n/a	n/a	12,430	0
ECC-163 [12]	Virtex-II	128	163	35.75 µs	4576	4.56	n/a	n/a	2251	6
RSA-1024 [42]	XC5VLX30T	450	1,024	1,520 µs	684,000	0.67	n/a	n/a	3,237	5[4]

5.2 Microcontroller Results

Our QC-MDPC encryption requires 606 byte SRAM and 3,705 byte flash memory for the iterative design and 606 byte SRAM and 5,496 byte flash memory in the unrolled version. Both versions already include the public key. The decryption unit requires 198 byte SRAM and 2,218 byte flash memory including the secret key, which is copied to SRAM at start-up for faster access. The encoder requires 26,767,463 cycles on average or 0.8 seconds at 32 MHz. Most cycles are consumed when adding a row of G to the ciphertext ($\sim$ 6000 cycles each) and when rotating a row to generate the next one ($\sim$ 2400 cycles).

The decoder requires 86,874,388 cycles on average or 2.7 seconds at 32 MHz. Rotating a polynomial in sparse representation takes 720 cycles and adding a sparse polynomial to the syndrome requires 2,285 cycles which clearly shows the advantage of a sparse representation. Nevertheless, computing a syndrome using the vector-(sparse-matrix)-multiplication on average requires 10,379,351 cycles. Because syndrome, ciphertext and the current row of H (even in sparse form) are too large to be held in registers, they have to be stored in SRAM and are continuously loaded and stored.

Comparison Table 4 compares our results with other implementation of McEliece and with implementations of the classical cryptosystems RSA and ECC on a similar microcontroller. For the code-based schemes, the flash memory usage includes the public and secret key, respectively. For RSA and ECC, [19] does not clearly state if the key size is included.

The main advantage of our implementations compared to other code-based schemes is the small memory footprint. Especially our decoder requires much less memory than other McEliece decoders because we only need to store the bit positions of the sparse secret polynomials instead of the full secret key.

We use the cycles/byte metric to compare our results to other implementations that handle different plaintext/ciphertext sizes. Our iterative encoder outperforms the encoders of [10] and [13]. Our unrolled version is nearly as fast as [20] with only half the amount of flash memory and six times less SRAM. Solely

Table 4. Performance comparison of our QC-MDPC microcontroller implementations with other public key encryption schemes.

Scheme	Platform	SRAM	Flash	Cycles/Op	Cycles/byte
This work [enc]	ATxmega256	606 Byte	3,705 Byte	37,440,137	62,400
This work [enc unrolled]	ATxmega256	606 Byte	5,496 Byte	26,767,463	44,612
This work [dec]	ATxmega256	198 Byte	2,218 Byte	86,874,388	146,457
McEliece [enc] [13]	ATxmega256	512 Byte	438 kByte	14,406,080	65,781
McEliece [dec] [13]	ATxmega256	12 kByte	130.4 kByte	19,751,094	90,187
McEliece [enc] [20]	ATxmega256	3.5 kByte	11 kByte	6,358,400	39,493
McEliece [dec] [20]	ATxmega256	8.6 kByte	156 kByte	33,536,000	208,298
McEliece [enc] [10]	ATxmega256	-	-	4,171,734	260,733
McEliece [dec] [10]	ATxmega256	-	-	14,497,587	906,099
ECC-P160 [19]	ATmega128	282 Byte	3682 Byte	6,480,000	324,000
RSA-1024 random [19]	ATmega128	930 Byte	6292 Byte	87,920,000	686,875

the quasi-dyadic McEliece implementation of [20] outperforms our implementation, however requires much more SRAM and flash memory.

6 Conclusions

In this work we presented implementations for the McEliece cryptosystem over QC-MDPC codes for Xilinx Virtex-6 FPGAs and AVR microcontrollers. Our implementations were primarily designed for high throughput and low memory consumption. Since decoding is generally the most expensive operation in code-based cryptography, we analyzed existing decoders and proposed several optimized decoders. We evaluated all decoders and selected the most suitable ones for the corresponding platforms. In addition, we showed that it is indeed possible to realize alternative public-key cryptosystems with moderate key size requirements and high performance or low memory on embedded systems. By demonstrating the excellent properties of this novel construction for embedded applications, we hope to have provided another incentive for further cryptanalytical investigation of QC-MDPC codes in the context of code-based cryptography.

Acknowledgements

Special thanks to Paulo, Rafael and Nicolas for fruitful discussions (Qathlo'). This work was supported in part by the German Federal Ministry of Economics and Technology (Grant 01ME12025 SecMobil) and in part by the Ministry of Economic Affairs and Energy of the State of North Rhine-Westphalia (Grant 315-43-02/2-005-WFBO-009).

References

1. M. Baldi, M. Bodrato, and F. Chiaraluce. A New Analysis of the McEliece Cryptosystem Based on QC-LDPC Codes. In R. Ostrovsky, R. D. Prisco, and I. Visconti, editors, *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 246–262. Springer, 2008.
2. M. Baldi and F. Chiaraluce. Cryptanalysis of a New Instance of McEliece Cryptosystem Based on QC-LDPC Codes. In *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*, pages 2591–2595, june 2007.
3. M. Baldi, F. Chiaraluce, and R. Garello. On the Usage of Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem. In *Communications and Electronics, 2006. ICCE '06. First International Conference on*, pages 305–310, oct. 2006.
4. M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni. Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 951–956, june 2007.
5. A. Becker, A. Joux, A. May, and A. Meurer. Decoding Random Binary Linear Codes in $2^n/20$: How 1+1=0 Improves Information Set Decoding. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 520–536. Springer Berlin Heidelberg, 2012.
6. T. P. Berger, P.-L. Cayrel, P. Gaborit, and A. Otmani. Reducing Key Length of the McEliece Cryptosystem. In *Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology*, AFRICACRYPT '09, pages 77–97, Berlin, Heidelberg, 2009. Springer-Verlag.
7. E. Berlekamp, R. McEliece, and H. van Tilborg. On the Inherent Intractability of Certain Coding Problems (Corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978.
8. D. J. Bernstein, T. Lange, and C. Peters. Attacking and Defending the McEliece Cryptosystem Cryptosystem. In *Proceedings of the International Workshop on Post-Quantum Cryptography – PQCrypto '08*, volume 5299 of *LNCS*, pages 31–46, Berlin, Heidelberg, 2008. Springer-Verlag.
9. B. Biswas and N. Sendrier. McEliece Crypto-system: A Reference Implementation. `http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes`.
10. P.-L. Cayrel, G. Hoffmann, and E. Persichetti. Efficient Implementation of a CCA2-Secure Variant of McEliece using Generalized Srivastava Codes. In *Proceedings of the 15th International Conference on Practice and Theory in Public Key Cryptography*, PKC'12, pages 138–155, Berlin, Heidelberg, 2012. Springer.
11. K. Chang. I.B.M. Researchers Inch Toward Quantum Computer. New York Times Article, February 28, 2012. `http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?_r=1&hpw`.
12. V. S. Dimitrov, K. U. Järvinen, M. J. Jacobson, W. F. Chan, and Z. Huang. FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers. In L. Goubin and M. Matsui, editors, *CHES*, volume 4249 of *Lecture Notes in Computer Science*, pages 445–459. Springer, 2006.
13. T. Eisenbarth, T. Güneysu, S. Heyse, and C. 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.
14. J.-C. Faugre, A. Otmani, L. Perret, and J.-P. Tillich. Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In H. Gilbert, editor, *Advances in Cryptology EUROCRYPT 2010*, volume 6110 of *Lecture Notes in Computer Science*, pages 279–298. Springer Berlin Heidelberg, 2010.
15. R. Gallager. Low-density Parity-check Codes. *Information Theory, IRE Transactions on*, 8(1):21–28, 1962.
16. S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede. A Speed Area Optimized Embedded Co-processor for McEliece Cryptosystem. In *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, pages 102 –108, july 2012.
17. N. Göttert, T. Feller, M. Schneider, J. Buchmann, and S. A. Huss. On the Design of Hardware Building Blocks for Modern Lattice-Based Encryption Schemes. In Prouff and Schaumont [33], pages 512–529.
18. T. Güneysu and C. Paar. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2008*, volume 5154 of *Lecture Notes in Computer Science*, pages 62–78. Springer-Verlag, 2008.
19. N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2004*, volume 3156 of *LNCS*, pages 925–943. Springer-Verlag, 2004.
20. S. Heyse. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In B.-Y. Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 143–162. Springer Berlin / Heidelberg, 2011.

21. S. Heyse and T. Güneysu. Towards One Cycle per Bit Asymmetric Encryption: Code-Based Cryptography on Reconfigurable Hardware. In *CHES*, pages 340–355, 2012.
22. W. C. Huffman and V. Pless. Fundamentals of Error-Correcting Codes, 2010.
23. A. A. Kamal and A. M. Youssef. An FPGA implementation of the NTRUEncrypt cryptosystem. In *Microelectronics (ICM), 2009 International Conference on*, pages 209–212. IEEE, 2009.
24. K. Kobara and H. Imai. Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece PKC. In *PKC '01: Proceedings of the 4th International Workshop on Practiceand Theory in Public Key Cryptography*, pages 19–35, London, UK, 2001. Springer-Verlag.
25. Y. X. Li, R. H. Deng, and X. M. Wang. On the Equivalence of McEliece's and Niederreiter's Public-key Cryptosystems. *IEEE Trans. Inf. Theor.*, 40(1):271–273, Sept. 2006.
26. L. Minder. *Cryptography Based on Error Correcting Codes*. PhD thesis, Ècole Polytechnique Fédérale de Lausanne, July 2007.
27. R. Misoczki and P. 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.
28. R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. L. M. Barreto. MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes. Cryptology ePrint Archive, Report 2012/409, 2012. `http://eprint.iacr.org/`.
29. C. Monico, J. Rosenthal, and A. Shokrollahi. Using Low Density Parity Check Codes in the McEliece Cryptosystem. In *Information Theory, 2000. Proceedings. IEEE International Symposium on*, page 215, 2000.
30. A. Otmani, J.-P. Tillich, and L. Dallot. Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. *Mathematics in Computer Science*, 3(2):129–140, 2010.
31. R. Overbeck and N. Sendrier. Code-based Cryptography. Bernstein, Daniel J. (ed.) et al., Post-quantum cryptography. First international workshop PQCrypto 2006, Leuven, The Netherland, May 23–26, 2006. Selected papers. Berlin: Springer. 95-145 (2009)., 2009.
32. E. Persichetti. Compact McEliece Keys based on Quasi-Dyadic Srivastava Codes. *IACR Cryptology ePrint Archive*, 2011:179, 2011.
33. E. Prouff and P. Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*, volume 7428 of *Lecture Notes in Computer Science*. Springer, 2012.
34. C. Rebeiro, S. S. Roy, and D. Mukhopadhyay. Pushing the Limits of High-Speed GF(2^m) Elliptic Curve Scalar Multiplication on FPGAs. In Prouff and Schaumont [33], pages 494–511.
35. S. S. Roy, C. Rebeiro, and D. Mukhopadhyay. A Parallel Architecture for Koblitz Curve Scalar Multiplications on FPGA Platforms. In *DSD*, pages 553–559. IEEE, 2012.
36. N. Sendrier. Decoding One Out of Many. In B.-Y. Yang, editor, *Post-Quantum Cryptography*, volume 7071 of *Lecture Notes in Computer Science*, pages 51–67. Springer Berlin Heidelberg, 2011.
37. P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms On a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.
38. A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and E. Kohnert. A Novel Cryptoprocessor Architecture for the McEliece Public-Key Cryptosystem. *IEEE Trans. Computers*, 59(11):1533–1546, 2010.
39. A. Shoufan, T. Wink, H. G. Molter, S. A. Huss, and F. Strenzke. A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2009.
40. F. Strenzke. Solutions for the Storage Problem of McEliece Public and Private Keys on Memory-Constrained Platforms. In D. Gollmann and F. C. Freiling, editors, *ISC*, volume 7483 of *Lecture Notes in Computer Science*, pages 120–135. Springer, 2012.
41. D. Suzuki. How to Maximize the Potential of FPGA Resources for Modular Exponentiation. *Cryptographic Hardware and Embedded Systems-CHES 2007*, pages 272–288, 2007.
42. D. Suzuki and T. Matsumoto. How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation. *IEICE Transactions*, 94-A(1):211–222, 2011.

Appendix

Table 5. Evaluation of the performance and error correcting capability of the different decoders for a QC-MDPC code with parameters $n_0 = 2, n = 9600, r = 4800, w = 90$.

Variant	#errors	time in µs	failure rate	avg. #iterations
Decoder $\mathcal{A}$	84	26.8	0.00041	5.2964
	85	27.3	0.00089	5.3857
	86	27.9	0.00221	5.4975
	87	28.7	0.00434	5.6261
	88	29.3	0.00891	5.7679
	89	30.1	0.01802	5.9134
	90	31.0	0.03264	6.0677
Decoder $\mathcal{B}$	84	12.6	0.00051	3.1425
	85	12.9	0.00163	3.1460
	86	13.4	0.00631	3.1607
	87	13.9	0.01952	3.2022
	88	14.6	0.05195	3.4040
	89	15.1	0.11462	3.5009
	90	15.7	0.24080	3.8972
Decoder $\mathcal{C}_1$	84	22.7	0.00044	5.2862
	85	23.2	0.00106	5.3924
	86	23.7	0.00172	5.4924
	87	24.2	0.00480	5.6260
	88	25.1	0.00928	5.7595
	89	25.6	0.01762	5.9078
	90	26.4	0.03315	6.0685
Decoder $\mathcal{C}_2$	84	14.0	0.00018	3.3791
	85	14.1	0.00068	3.4180
	86	14.2	0.00148	3.4643
	87	14.6	0.00378	3.5279
	88	14.8	0.00750	3.5942
	89	15.1	0.01500	3.6542
	90	15.4	0.02877	3.7435
Decoder $\mathcal{D}$	84	7.02	0.00001	2.4002
	85	7.04	0.00003	2.4980
	86	7.24	0.00004	2.5979
	87	7.53	0.00031	2.6958
	88	7.78	0.00093	2.7875
	89	8.13	0.00234	2.8749
	90	8.31	0.00552	2.9670
Decoder $\mathcal{E}$	84	14.15	0.00019	3.3754
	85	14.14	0.00073	3.4218
	86	14.77	0.00153	3.4673
	87	14.63	0.00375	3.5314
	88	15.11	0.00728	3.5886
	89	15.15	0.01529	3.6563
	90	15.68	0.02840	3.7343
Decoder $\mathcal{F}$	84	6.68	0.00000*	2.4047
	85	6.92	0.00002	2.5000
	86	7.11	0.00008	2.5983
	87	7.59	0.00039	2.6939
	88	7.68	0.00094	2.7912
	89	7.99	0.00209	2.8793
	90	8.54	0.00506	2.9630

* Note, this does not mean that Decoder $\mathcal{F}$ always succeeds. It is still a probabilistic decoder that simply did not encounter any decoding failure in our evaluations.