Implementing QC-MDPC McEliece Encryption

INGO VON MAURICH, Ruhr-Universität Bochum
TOBIAS ODER, Ruhr-Universität Bochum
TIM GÜNEYSU, Ruhr-Universität Bochum

With respect to performance, asymmetric code-based cryptography based on binary Goppa codes has been reported as a highly interesting alternative to RSA and ECC. A major drawback are still the large keys in the range between 50-100 kByte that prevented real-world applications of code-based cryptosystems so far. A recent proposal by Misoczki et al. showed that quasi-cyclic moderate density parity-check (QC-MDPC) codes can be used in McEliece encryption – reducing the public key to just 0.6 kByte to achieve an 80-bit security level. In this article we propose optimized decoding techniques for MDPC codes and we provide and survey several efficient implementations of the QC-MDPC McEliece cryptosystem. On the one hand this includes high-speed and lightweight architectures for reconfigurable hardware, efficient coding styles for ARM's Cortex-M4 microcontroller but also novel high-performance software implementations that fully employ vector instructions. Finally, we conclude that McEliece encryption using QC-MDPC codes not only enable high-performance implementations but also extremely lightweight designs on a wide range of different platforms.

Categories and Subject Descriptors: E.3 [**DATA ENCRYPTION**]: Public key cryptosystems; B.7.1 [**INTEGRATED CIRCUITS**]: Algorithms implemented in hardware

General Terms: Algorithms, Performance, Security

Additional Key Words and Phrases: McEliece, Code-based cryptography, MDPC codes, Implementation, FPGA, Microcontroller, Software

ACM Reference Format:
ACM Trans. Embedd. Comput. Syst. V, N, Article A (January YYYY), 24 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Factoring (RSA) and the discrete logarithm problem (DH, ECC) are the foundation of nearly all public key encryption schemes used today. These two classes of problems have suffered some dents and bruises over time (e.g., the general number field sieve and index calculus methods) but no devastating attack on conventional computing platforms has emerged so far that is capable of efficiently solving one of the two. Both are presumed difficult although it seems unlikely to obtain any proofs or reductions to NP-complete problems. Moreover, it is well-known that both problems can be solved in polynomial time by the quantum computing algorithm due to [Shor 1997] – although a quantum computer capable of executing these attacks on large numbers has not been developed yet.

When considering cryptography for the coming decade this is not a comfortable situation. Cryptanalytic improvements or the advent of a powerful quantum computer could instantly render the security assumptions of today's public key encryption schemes useless. Needless to say that alternative cryptosystems which provide (a) the same security services at (b) a comparable level of computational efficiency and (c) costs for storing keys are urgently required.

Author's addresses: Ingo von Maurich, Tobias Oder, and Tim Güneysu, Ruhr-Universität Bochum, Germany, ingo.vonmaurich@rub.de, tobias.oder@rub.de, tim.gueneysu@rub.de

DOI: http://dx.doi.org/10.1145/0000000.0000000

Promising alternatives are currently categorized into code-based, lattice-based, multivariate-quadratic, and hash-based cryptography. Major drawbacks of many proposed cryptosystems within these categories are their low efficiency and practicability due to large keys 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 1978] and [Niederreiter 1986] can significantly outperform classical asymmetric cryptosystems on embedded systems [Eisenbarth et al. 2009; Heyse 2011; Ghosh et al. 2012; Cayrel et al. 2012] – at the cost of very large keys (50-100 kByte for 80-bit security).

Many proposals already tried to address the issue of large keys by replacing the originally used binary Goppa codes with codes that allow more compact representations, e.g., [Misoczki and Barreto 2009; Cayrel et al. 2012]. However, many attempts were broken [Faugère et al. 2010; Faugère et al. 2014a; Faugère et al. 2014b] and for the few survivors hardly any implementations are publicly available [Berger et al. 2009; Heyse 2011]. In the context of this work, low density parity check (LDPC) codes [Gallager 1962] have repeatedly been suggested as candidates for McEliece [Monico et al. 2000; Baldi et al. 2006; Baldi et al. 2007; Baldi and Chiaraluce 2007; Baldi et al. 2008]. The use of quasi-cyclic LDPC codes was suggested for McEliece in [Baldi et al. 2008] but due to the cryptanalytic results of [Monico et al. 2000] and [Baldi and Chiaraluce 2007] in [Otmani et al. 2010], McEliece based on LDPC codes does not seem to be a secure choice.

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. 2012] proposed to use medium-density parity check (MDPC) codes and their quasi-cyclic (QC) variant as such an alternative (published with small changes in the parameter sets in [Misoczki et al. 2013]), claiming that a public key of only 4801 bit can provide a level of 80 bit equivalent symmetric security. In particular, the authors claim that (QC-)MDPC codes resist previous attacks and lack an obvious algebraic structure. Yet it needs to be investigated if all requirements of different computing platforms can be met with McEliece based on QC-MDPC codes.

1.1. Contribution

In this work we survey and extend implementations of QC-MDPC McEliece for reconfigurable hardware, embedded microcontrollers, and software implementations optimized for general-purpose processors. We include high-speed FPGA implementations presented at CHES'13 [Heyse et al. 2013], lightweight FPGA implementations presented at DATE'14 [von Maurich and Güneysu 2014], ARM Cortex-M4 microcontroller implementations to appear at PQCrypto'14[1], as well as novel software implementations that employ vector instructions.

Furthermore, we investigate several ways to efficiently decode erroneous MDPC codewords and propose and evaluate new optimizations that lead to faster computations, less decoding iterations, and a reduced decoding failure probability. The evaluations account for the updated parameters since [Heyse et al. 2013] and include the updated decoder of [Misoczki et al. 2013] that replaced fixed thresholds $b = \max(\#_{\text{upc}}) - \delta$ with an iteratively decrementing $\delta = \delta - 1$ in case of a decoding failure (repeated until $\delta = 0$).

1.2. Outline

In Section 2 we provide preliminaries on (QC-)MDPC codes and McEliece public key encryption. Optimizations for decoding (QC-)MDPC codes are discussed in Section 3. A high-performance FPGA implementation of QC-MDPC McEliece is presented in Section 4, followed by a lightweight FPGA implementation in Section 5. Section 6

[1]A preprint of the paper is made available to the reviewers at http://www.sha.rub.de/media/attachments/files/2014/07/draft_acm_review.pdf.

describes how to implement the scheme for embedded microcontrollers. Finally, we present novel PC implementations using vector instructions (SSE4) in Section 7. A conclusion is drawn in Section 8.

2. PRELIMINARIES

This section introduces (QC-)MDPC codes and describes how the McEliece public key encryption scheme can be instantiated using these codes. Originally, this scheme was proposed in [Misoczki et al. 2013].

2.1. (QC-)MDPC Codes

A binary linear $[n, k]$ error-correcting code C of length n is a subspace of $\mathbb{F}_2^n$ of dimension k and co-dimension $r = n - k$. Code C can either be defined by a generator matrix or by a parity-check matrix. The generator matrix $G \in \mathbb{F}_2^{k \times n}$ defines the code as $C = \{mG \in \mathbb{F}_2^n \mid m \in \mathbb{F}_2^k\}$ and the parity-check matrix $H \in \mathbb{F}_2^{r \times n}$ defines the code as $C = \{c \in \mathbb{F}_2^n | \, cH^T = 0^r\}$. The syndrome $s \in \mathbb{F}_2^r$ of a vector $x \in \mathbb{F}_2^n$ is defined as $s = Hx^T$. It follows that if $x \in C$ then $s = 0^r$, otherwise $s \neq 0^r$.

If there exists some integer n_0 such that every cyclic shift of a codeword $c \in C$ by n_0 positions results in another codeword $c' \in C$ then code C is called quasi-cyclic (QC). If $n = n_0 p$ for some integer p, then both the generator and the parity-check matrix are composed of $p \times p$ circulant blocks. It suffices to store one row (usually the first) of each circulant block to completely describe the matrices.

A (n, r, w)-MDPC code is a binary linear $[n, k]$ code defined by a parity-check matrix with constant row weight w. A (n, r, w)-QC-MDPC code is a (n, r, w)-MDPC code that is quasi-cyclic with $n = n_0 r$.

2.2. The QC-MDPC McEliece Cryptosystem

Key-generation, encryption and decryption for the McEliece cryptosystem based on a t-error correcting (n, r, w)-QC-MDPC code are defined as follows.

Key-Generation. Key-generation requires to generate a (n, r, w)-QC-MDPC code with $n = n_0 r$. The public key is the generator matrix G and the secret key is the parity-check matrix H. In order to generate a (n, r, w)-QC-MDPC code with $n = n_0 r$, select the first rows $h_0, \ldots, h_{n_0-1}$ of the n_0 parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ with weight $\sum_{i=0}^{n_0-1} wt(h_i) \leq w$ uniformly at random. The parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ are then generated by $r - 1$ quasi-cyclic shifts of $h_0, \ldots, h_{n_0-1}$. A horizontal concatenation of $H_0, \ldots, H_{n_0-1}$ forms the parity-check matrix. Generator matrix $G = [I|Q]$ is computed from H in row reduced echelon form by concatenating the identity matrix I_k and matrix

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

Since both matrices are quasi-cyclic, it suffices to store their first rows instead of the full matrices.

Encryption. To encrypt a message $m \in \mathbb{F}_2^k$, generate an error vector $e \in \mathbb{F}_2^n$ with at most t set bits uniformly at random and compute $x = mG \oplus e$.

Decryption. To decrypt a ciphertext $x \in \mathbb{F}_2^n$, compute $mG \leftarrow \Psi_H(x)$ using a t-error correcting (QC-)MDPC decoder Ψ_H. Since G is of systematic form, extract m from the first k positions of mG.

2.3. Security of QC-MDPC McEliece

The description of McEliece based on QC-MDPC codes in Section 2.2 eliminates the scrambling matrix S and the permutation matrix P usually used in the McEliece cryptosystem. A CCA2 conversion (e.g., [Kobara and Imai 2001; Nojima et al. 2008]) allows G to be in systematic-form without introducing security flaws.

Table I. Parameters for different security levels for McEliece with QC-MDPC codes.

Security level	n_0	n	r	w	t	Public key size
80 bit	2	9602	4801	90	84	4801 bit
80 bit	3	10779	3593	153	53	7186 bit
80 bit	4	12316	3079	220	42	9237 bit
128 bit	2	19714	9857	142	134	9857 bit
128 bit	3	22299	7433	243	85	14866 bit
128 bit	4	27212	6803	340	68	20409 bit
256 bit	2	65542	32771	274	264	32771 bit
256 bit	3	67593	22531	465	167	45062 bit
256 bit	4	81932	20483	644	137	61449 bit

[Misoczki et al. 2013] state 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 [Becker et al. 2012] and also the improvements achieved by the DOOM-attack [Sendrier 2011], [Misoczki et al. 2013] suggest parameters as given in Table I. For a detailed discussion of the security of this scheme we refer to [Misoczki et al. 2013].

2.4. Parameter Selection

Table I lists the QC-MDPC parameters proposed by [Misoczki et al. 2013]. Since small public key sizes are particularly crucial for embedded systems, we pick the parameter sets with $n_0 = 2$ from Table I. In this work we mostly focus on a 80-bit security level, hence our implementation use the following parameters:

$$n_0 = 2, n = 9602, r = 4801, w = 90, t = 84.$$

With these parameters a 4801-bit plaintext block is encoded into a 9602-bit codeword to which $t = 84$ errors are added. The parity-check matrix H has constant row weight $w = 90$ and consists of $n_0 = 2$ circulant blocks. The redundant part Q of the generator matrix G consists of $n_0 - 1 = 1$ circulant block.

3. EFFICIENT DECODING OF (QC-)MDPC CODES

Compared to the simple operations involved in encryption (i.e., a vector-matrix multiplication followed by an addition), decoding is a more complex operation. As the selection of an efficient decoding algorithm is crucial to the overall decoding performance, it is imperative to evaluate and compare all available options.

3.1. Decoding (QC-)MDPC Codes

Decoding algorithms for LDPC/MDPC codes are mainly divided into two families. The first class (e.g., [Berlekamp et al. 1978]) 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 [Gallager 1962], seems to be more appropriate. In general, bit-flipping algorithms are based on the following principle:

(1) Compute the syndrome of the received ciphertext $s = Hx^T$.
(2) Count the number of unsatisfied parity-check equations $\#_{\text{upc}}$ associated with each ciphertext bit.
(3) Flip each ciphertext bit that violates more than b equations.
(4) Recompute the syndrome of the updated ciphertext.

This process is repeated until either the syndrome becomes zero or a predefined maximum number of iterations is reached upon which a decoding error is returned.

The main difference between the bit-flipping algorithms is how they determine the threshold b. In [Gallager 1962], thresholds b_i are precomputed for each iteration i. [Huffman and Pless 2010] set the threshold as the maximum of the unsatis-

fied parity-check equations $b = \max(\#_{\text{upc}})$ and [Misoczki et al. 2013] propose to use $b = \max(\#_{\text{upc}}) - \delta$, for some small δ.

3.2. Decoding Optimizations

In the following we propose ways to optimize the number of required decoding iterations, the decoding-failure rate, and to accelerate the syndrome computation.

Improving the Syndrome Computation. All bit-flipping decoders in literature recompute the syndrome after every decoding iteration to decide whether decoding was successful. The cost for one syndrome computation alone can be approximated at around twice the cost of one encoding.

We propose an optimization that can be applied to all bit-flipping decoders based on the following observation: If the number of unsatisfied parity-check equations exceeds threshold b, the corresponding bit in the ciphertext is flipped and the syndrome changes. We stress that the syndrome does not change arbitrarily, but the new syndrome is equal to the old syndrome accumulated with row h_j of the parity check matrix that corresponds to the flipped ciphertext bit j. By keeping track of which ciphertext bits are flipped and updating the syndrome accordingly, the syndrome recomputation can be omitted.

There are two ways to apply this optimization. One is to store all changes to the syndrome in a separate register and add the changes at the end of a decoding iteration to the syndrome. This way, the decoding behavior remains unchanged and only the syndrome computation is accelerated. The other possibility is to directly apply the changes to the syndrome whenever a ciphertext bit is flipped. This affects the decoding behavior as well as accelerates the syndrome computation. We explore both approaches in Section 3.3.

Improving the Decoding Failure Rate. The decoder proposed by [Gallager 1962] precomputes thresholds based on the code parameters. We found that its error-correcting capability can be improved by incrementing the precomputed thresholds by a Δ in case of a decoding failure and restart decoding with the adapted thresholds. When restarting, the initial syndrome does not need to be recomputed as it can be restored from the first decoding try. Incrementing the precomputed thresholds upon a decoding failure can be seen as a similar approach as taken by [Misoczki et al. 2013] when decrementing δ. We achieved the best improvements when setting $\Delta = 1$ and after every decoding failure increasing it to $\Delta = \Delta + 1$ until reaching a predefined $\Delta_{\max}$.

3.3. Investigated Decoding Algorithms

Estimating the error-correction capability of LDPC and MDPC codes generally is nontrivial and influenced by the choice of threshold b. Hence, we derive several bit-flipping algorithms, evaluate their error-correcting capability and count how many iterations are required on average to decode a codeword.

Since we are targeting embedded systems, we omit variants that store n_0 counters for each ciphertext bit to compute their $\#_{\text{upc}}$. This would allow to skip the second computation of $\#_{\text{upc}}$ in some variants, but would blow up memory consumption to an unacceptable amount. The decoders under investigation are:

Decoder $\mathcal{A}$ is given in [Misoczki et al. 2013] and computes the syndrome, then checks the number of unsatisfied parity-check equations once to compute the maximum $\max(\#_{\text{upc}})$ and then a second time to flip all codeword bits that violate $b \geq \max(\#_{\text{upc}}) - \delta$ equations. Afterwards, the syndrome is recomputed and compared to zero. If decoding is not successful after some defined maximum of iterations, δ is reduced to $\delta = \delta - 1$ and the decoding process is restarted. This is repeated until $\delta = 0$ where the decoder becomes equal to [Huffman and Pless 2010].

Decoder $\mathcal{B}$ is given in [Gallager 1962] 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 $\#_{\text{upc}}$ is larger than a precomputed threshold b_i.

Afterwards, the syndrome is recomputed and compared to zero.

In order to evaluate our optimizations of the syndrome computation and the adaptive precomputed thresholds, we derive the following decoders:

Decoder $\mathcal{C}_1$ computes the syndrome, then checks the number of unsatisfied parity-check equations once to compute the maximum $\max(\#_{\text{upc}})$ and then a second time to flip all codeword bits that violate $b \geq \max(\#_{\text{upc}}) - \delta$ 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, resulting in the syndrome of the new codeword without requiring a full recomputation. In case of a decoding error, δ is decremented as in Decoder $\mathcal{A}$.

Decoder $\mathcal{C}_2$ computes the syndrome, then checks the number of unsatisfied parity-check equations once to compute the maximum $\max(\#_{\text{upc}})$ and then a second time to flip all codeword bits that violate $b \geq \max(\#_{\text{upc}}) - \delta$ equations. If a codeword bit j is flipped, the corresponding row h_j of the parity check matrix is *directly* added 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. In case of a decoding error, δ is decremented as in Decoder $\mathcal{A}$.

Decoder $\mathcal{C}_3$ 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 $\mathcal{D}_1$ is similar to Decoder $\mathcal{B}$ with precomputed thresholds b_i, but uses the *direct* update of the syndrome.

Decoder $\mathcal{D}_2$ is similar to Decoder $\mathcal{D}_1$ and in addition increments the precomputed thresholds in case of a decoding failure until $\Delta_{\max} = 5$.

Decoder $\mathcal{D}_3$ is similar to Decoder $\mathcal{D}_2$ and in addition uses the same early exit trick as Decoder $\mathcal{C}_3$.

3.4. Decoding Performance Evaluation

The average number of iterations required to decode a codeword and the decoding failure rate are listed in Table IX in the Appendix for all decoders described in Section 3.3 for different numbers of errors $t = \{84, \ldots, 90\}$. All measurements are taken for QC-MDPC codes with parameters $n_0 = 2, n = 9602, r = 4801, w = 90$. A total of 1,000 random codes and 10,000 random decoding trials per code were evaluated on an AMD Opteron 6276 CPU running at 2.3 GHz.

For decoders with precomputed thresholds b_i we used the formula given in Appendix A of [Misoczki et al. 2012] to precompute the most suitable b_i's for every iteration. For decoders with $b = \max(\#_{\text{upc}}) - \delta$, we found that the smallest number of iterations are required when starting with $\delta = 5$[2]. A decoding failure is returned in case the decoder did not succeed within ten iterations.

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

Comparing the two decoders from literature ($\mathcal{A}$ and $\mathcal{B}$), it is evident that decoder $\mathcal{B}$ requires less decoding iterations and on average just half the time to decode a erroneous codeword. On the other hand it encounters more decoding errors.

[2]In the latest version of [Misoczki et al. 2012] the authors also suggest to use $\delta \approx 5$ for the given parameters.

When comparing decoders $\mathcal{A}$ and $\mathcal{C}_1$, our acceleration of not having to recompute the syndrome becomes apparent in the average execution time which is reduced by 20%.

Our proposal to directly update the syndrome when flipping a ciphertext bit has an even stronger impact on the decoding performance as well as on the decoding failure rate. Not only do we speed up the computation time, but we also reduce the average number of required decoding iterations by 40% (compare decoders $\mathcal{C}_1$ and $\mathcal{C}_2$). Furthermore, the number of decoding failures is highly reduced.

Combining Gallager's precomputed thresholds with a directly updated syndrome results in the lowest number of decoding iterations (decoders $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3$). On average we save 2.9 iterations compared to decoder $\mathcal{A}$ and 0.7 iterations compared to $\mathcal{B}$. Less iterations directly relate to the execution time and combined with the new syndrome update technique decoding is 2-4 times faster.

Adapting the precomputed thresholds upon a decoding error as proposed in Section 3.2 leads to the best decoding failure rates among all decoders under investigation (compare $\mathcal{D}_1$ with $\mathcal{D}_2/\mathcal{D}_3$). The average number of decoding iterations and the average execution time increase only very slightly when using this technique. The small timing advantage of decoders $\mathcal{C}_3/\mathcal{D}_3$ over $\mathcal{C}_2/\mathcal{D}_2$ is due to the immediate termination if the syndrome becomes zero.

Another interesting observation we made for all decoders is that if a ciphertext is decodable, then this is achieved after a small number of iterations. We noticed that if a ciphertext is not decoded within 4-6 iterations, a higher number of iterations rarely leads to a successful decoding without adapting the thresholds. Therefore, an early detection of a decoding failure is possible.

3.5. Decoding Algorithm Selection

Throughout this work we base our implementations on Decoder $\mathcal{D}_1/\mathcal{D}_2$. Even though Decoder $\mathcal{D}_3$ has a small timing advantage, it possibly introduces a timing side-channel. Although we are not aware of a way to exploit the information of how long it takes for the syndrome to become zero, history has shown that it is advisable to avoid leaking any timing information, especially if it can be avoided at low cost.

Decoder $\mathcal{D}_1$ can be summarized by

(1) Compute the syndrome of the ciphertext $s = Hx^T$.
(2) Count the number of unsatisfied parity checks for every ciphertext bit.
(3) If the number of unsatisfied parity checks for a ciphertext bit exceeds a precomputed threshold, flip the ciphertext bit and update the syndrome.
(4) If $s = 0^r$, the codeword was decoded successfully. If $s \neq 0^r$, go to Step (2) or abort after a defined maximum of iterations with a decoding error.

Decoder $\mathcal{D}_2$ can be seen as a wrapper around $\mathcal{D}_1$ that modifies the thresholds upon a decoding error and then calls $\mathcal{D}_1$ again.

4. HIGH-PERFORMANCE QC-MDPC MCELIECE FOR RECONFIGURABLE HARDWARE

The primary goal of our first hardware design is to provide a high-performance public key encryption core for Xilinx FPGAs.

We base our QC-MDPC McEliece decryption implementation on decoder $\mathcal{D}_1/\mathcal{D}_2$ in hardware. The reason for not choosing decoder $\mathcal{D}_3$ is that we sequentially rotate the codewords and secret keys in every cycle of the bit-flipping iterations. If the syndrome 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. Furthermore, an early abort could leak timing information about the point in time when the syndrome became zero, which is undesirable as well.

For our evaluation of QC-MDPC in reconfigurable hardware we target Xilinx's Virtex-6 FPGAs. 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 explain our design choices and describe the implementations of QC-MDPC McEliece. Note that we did not consider the implementation of a CCA2 conversion and true random number generation in the scope of this work.

4.1. 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 Goppa codes. Since we target high-performance, 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 4801-bit vector from a 32-bit BRAM interface would consume at least 151 clock cycles. However, if maximum performance is not required, the use of BRAMs reduces the resource consumption significantly as will be shown in Section 5.

We do not exploit the sparsity of the secret polynomials in this 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 4800 they have be reset to 0. If a bit in the ciphertext x_0 or x_1 is set we have to build a 4801-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.

4.2. High-Performance Implementation

We use a Virtex-6 XC6VLX240T FPGA as target device for 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 ciphertexts are send and received bit-by-bit to keep the I/O overhead small.

QC-MDPC Encryption. In order to implement QC-MDPC encryption 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 $wt(e) \leq 84$ to get the ciphertext $x = c \oplus e$. We are given a 4801-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 require three 4801-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 plaintext can be read from the first half of the decoded codeword. We implement bit-flipping decoder $\mathcal{D}_1$ as described in Section 3.5. 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 9602-bit row $h = [h_0|h_1]$ of H and the 9602-bit codeword $x = [x_0|x_1]$ the syndrome is computed as follows. We sequentially iterate over every bit of the ciphertext 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 test the syndrome for 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, 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 threshold b_i for the current iteration i, the corresponding bit in the ciphertext 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 counting the number of unsatisfied parity check equations for h_0 and h_1 can be done 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 using less resources. We explore both version to evaluate this time/resource trade-off.

Computing the Hamming weight of a 4801-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 4801-bit of the updated ciphertext 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 reached.

4.3. Implementation Results

All our results are obtained post place-and-route (PAR) for a Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 14.7. For the throughput figures we assume an I/O interface capable of these speeds is provided.

In hardware, our QC-MDPC encoder runs at 351.7 MHz and encodes a 4801-bit message in 4801 clock cycles which results in 351.7 Mbit/s. The iterative version of our QC-MDPC decoder runs at 222.5 MHz. Since the time to decode depends on how many decoding iterations are needed, we calculate the average required cycles for iterative decoding as follows. Computing the syndrome for the first time needs 4801 clock cycles and comparing the syndrome to zero takes another 2 clock cycles. For every following bit-flipping iteration we need 9622 plus again 2 clock cycles for checking the syndrome. As shown in Table IX, decoder $\mathcal{D}_1$ needs 2.4019 bit-flipping iterations on average. Thus, the average cycle count for our iterative decoder is $4801+2+2.4019\cdot(9622+2) = 27918.9$ clock cycles.

Our parallel decoder processes both secret polynomials in the bit-flipping step in parallel and runs at 199.3 MHz. We calculate the average cycles as before with the difference that every bit-flipping iteration now takes $4811 + 2$ clock cycles. Thus, the average cycle count for the parallel decoder is $4801 + 2 + 2.4019 \cdot (4811 + 2) = 16363.3$ clock cycles.

The parallel decoder operates 35% faster than the iterative version while occupying 6-26% more resources. Compared to the decoders, the encoder runs 6-9 times faster and occupies 2-5 times less resources. Table II summarizes our results.

Using the formerly proposed decoders that work without our optimizations (i.e., decoders $\mathcal{A}$ and $\mathcal{B}$) would result in much slower decryptions. Decoder $\mathcal{A}$ would need $4803+5.2922\cdot(2\cdot9622+4803) = 132064.5$ cycles in an iterative and $4803+5.2922\cdot(2\cdot4811+4803) = 81143.0$ cycles in a parallel implementation. Decoder $\mathcal{B}$ saves cycles by skipping the $\max(\#_{\text{upc}})$ computation but would still need $4803 + 3.0936 \cdot (9622 + 4803) = 49428.2$ cycles in an iterative and $4803 + 3.0936 \cdot (4811 + 4803) = 34544.9$ cycles in a parallel implementation.

Table II. Implementation results of our QC-MDPC implementations with parameters $n_0 = 2, n = 9602, r = 4801$, $w = 90, t = 84$ (80-bit equivalent symmetric security) on a Xilinx Virtex-6 XC6VLX240T FPGA.

Aspect	Encoder	Decoder (iterative)	Decoder (parallel)
FFs	14,429 (4%)	32,962 (10%)	41,714 (13%)
LUTs	9,201 (6%)	36,502 (24%)	42,274 (28%)
Slices	2,924 (7%)	10,364 (27%)	10,988 (29%)
Frequency	351.7 MHz	222.5 MHz	199.3 MHz
Time/Op	13.7 µs	125.4 µs	82.1 µs
Throughput	351.7 Mbit/s	38.3 Mbit/s	58.5 Mbit/s
Encode	4,801 cycles	-	-
Compute Syndrome	-	4,801 cycles	4,801 cycles
Check Zero	-	2 cycles	2 cycles
Flip Bits	-	9,622 cycles	4,811 cycles
Overall average	4,801 cycles	27,918.9 cycles	16,363.3 cycles

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 X in the Appendix. The most relevant metric for comparing the performance of public key encryption schemes often depends on the application. For key exchange it is usually the required time per operation, given that the symmetric key size is smaller or equal to the block size that can be transmitted in one operation. For data encryption (i.e., more than one block), throughput in Mbit/s is typically the more interesting metric.

A hardware McEliece implementation based on Goppa codes including a CCA2 conversion was presented for a Virtex5-LX110T FPGA in [Shoufan et al. 2009; Shoufan et al. 2010]. Comparing their performance to our implementations shows the advantage of QC-MDPC McEliece in both time per operation and throughput. The occupied resources are similar to our resource requirements but in addition they need 75 block memories. 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 [Shoufan et al. 2010].

Another McEliece co-processor was recently proposed for a Virtex5-LX110T FPGA by [Ghosh et al. 2012]. Their design goal was to optimize the speed/area ratio while we aim for high-performance. Regarding decoding, our implementations outperform their work in both time/operation and throughput. However, they need fewer resources which allows implementation on low-cost devices such as Spartan-3 FPGAs. Their public keys 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 using Goppa codes by [Heyse and Güneysu 2012] for a Virtex6-LX240T FPGA. Their 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 McEliece achieves better results. Furthermore, public keys with a size of 63.5 kByte are a large storage requirement for embedded devices.

FPGA implementations of lattice-based public key encryption were proposed by [Roy et al. 2013; Pöppelmann and Güneysu 2013] for Ring-LWE and by [Kamal and Youssef 2009] for NTRU. The Ring-LWE implementations require 1.5-2 times more time to encrypt a smaller plaintext but decrypts ciphertexts faster and occupies less resources at the cost of using block RAMs and digital signal processors. If high-throughput is required, QC-MDPC McEliece outperforms both implementations for encryption and decryption. NTRU as implemented by [Kamal and Youssef 2009] provides high-performance at moderate resources requirements. However, the selected parameters for this implementation only achieve a security level of around 64 bit. 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 [Dimitrov et al. 2006; Güneysu and Paar 2008; Rebeiro et al. 2012; Roy et al. 2012] which all yield good performance at moderate resource requirements. The most efficient RSA hardware implementation to date was proposed in [Suzuki 2007; Suzuki and Matsumoto 2011]. Both the time to encrypt and decrypt one block as well as the throughput are considerably worse than QC-MDPC McEliece.

5. LIGHTWEIGHT QC-MDPC MCELIECE FOR RECONFIGURABLE HARDWARE

In this section we show how the small keys of QC-MDPC McEliece can be efficiently stored in embedded block memories to achieve a much smaller area footprint implementation that still provides a decent speed sufficient for many applications and targets Xilinx's low-cost Spartan-6 family. For fair comparison we also implement our designs on the same Virtex-6 FPGA as in Section 4.

5.1. Design Considerations

Intuitively, a small area footprint implementation of QC-MDPC McEliece should be possible due to the comparably small keys. Instead of having to provide memory to store 50-100 kByte just for the keys, the public key has a length of 4801 bit and the secret key requires 9602 bit. Apart from keys, additional parameters such as the message, the ciphertext, and the syndrome, with sizes roughly in the same range, have to be stored as well.

FPGAs such as the Xilinx Spartan-6 and Virtex-6 family are equipped with dual-ported block memories (BRAMs) each capable of storing up to 18/36 kBit of data. In each clock cycle two separate 32-bit values can be read from two different memory addresses and when using the READ_FIRST mode, it is even possible to write data to a memory cell in the same clock cycle after reading its content.

In our design of the encryption and decryption unit we store all inputs, outputs, keys and intermediate values in these block memories and process them in a 32-bit fashion to achieve a very compact structure. In the following we detail our design choices for the encryption and decryption cores.

Encryption. During McEliece encryption we have to compute $x = mG \oplus e$ which boils down to an accumulation of the rows of generator matrix G depending on set bits in the message m and an addition of the error vector e. Hence, we have to hold the message (4801 bit), one row of the generator matrix (4801 bit), and the redundant part (second half of x, 4801 bit) in memory. The error vector e is added on-the-fly (provided through a 32-bit interface), to avoid having to store additional 9602 bits out of which at most 84 are set. In total we have to store $3 \cdot 4801$ bit, fitting one 18 kBit BRAM. In addition to the available storage space we also have to consider that only two data ports are available for each BRAM. In a straightforward approach we would need three data ports (and thus 2 BRAMs), one for the message, one for the public key and one for the redundant part.

Since each message bit needs to be processed only once as opposed to the redundant part and the public key which are each accessed 4801 times, we store all values in one BRAM and spend a 32-bit register to hold the current 32-bit message block which we are processing. While the encryption unit is idle, it allows external components to access its internal BRAM to read out the encrypted ciphertext, to write a new message and if required to also change the public key. When starting the encryption, it takes control of the BRAM and allows outside components to access the BRAM only after the encryption is finished.

Decryption. McEliece decryption is equal to decoding QC-MDPC codes as described in Section 3 and hence we have to store the secret key (9602 bit), the received ciphertext (9602 bit), and the syndrome (4801 bit). Decoding is performed in-place, i.e., after the decoder finishes, the first 4801 bit of the decoded ciphertext are equal to the decrypted message. The secret key and the ciphertext consist of two separate 4801-bit

vectors that can either be processed in parallel or iteratively. Since decryption is more complex than encryption we process them in parallel to not further widening the gap between encryption and decryption performance.

Concerning memory, two 18-kBit BRAMs suffice to store all values. As already discussed in the design of the encoder, it is also important to keep in mind that each BRAM only offers two data ports. Since the secret key and the ciphertext consist of two separate 4801-bit vectors that are processed in parallel, this requires four data ports plus one data port for the syndrome. Trading performance at the cost of few additional resources, we spend one additional 18-kBit BRAM to store the syndrome.

When decoding, the first step is the syndrome computation which is similar to encoding a message. Depending on set ciphertext bits, rows of the two parity-check matrix blocks are accumulated. For comparing the syndrome to zero, we compute the OR of all 32-bit blocks of the syndrome. If the OR result is zero, the syndrome is zero, otherwise it is not. For counting the number of unsatisfied parity-check equations we compute the Hamming weight of the binary AND of the syndrome and the two parts of the secret key, again in 32-bit steps.

While the decryption unit idles, it grants access to the BRAM containing the ciphertext so that external components can write new ciphertexts and read out decrypted plaintexts. We do not allow external components to access the secret key in our design. Depending on the application it might be desired to be able to at least write a new secret key. This can be easily accomplished in our design by forwarding the required control signals of the secret key BRAM to external components.

5.2. Lightweight Implementation

Next we detail our lightweight implementations of QC-MDPC McEliece en- and decryption following the design decisions as explained above. Note that the implementation of a CCA2 conversion as well as the investigation of a secure implementation of a true random number generator are out of the scope of this work.

Encryption. Encryption usually starts with setting the redundant part to zero, then the rows of the generator matrix are accumulated depending on the message. In the end an error vector is added to the result. In our implementation we combine resetting the redundant part and adding the error by directly loading the second half of the error vector into the redundant part and starting the accumulation of the rows of G to it afterwards. We rely on being provided a uniformly distributed error vector of weight at most $t = 84$ through a 32-bit interface.

The most performance-critical operation of the encoder is the rotation of 4801-bit vectors. More precisely, the first row g of the generator matrix has to be rotated 4801 times to iterate over all rows of G. In a BRAM-based implementation, each data port can only access 32 bit per clock cycle. Hence, rotating a 4801-bit vector requires to load 152 32-bit cells[3], rotating them by one bit, and storing the result.

In a straightforward approach with one data port two cycles would be needed to rotate each 32-bit block. One cycle to load the value and rotate it, and another cycle to store the result. If two data ports would be used, one port could be used to read blocks and the second port could be used to write blocks delayed by one clock cycle. This would require one clock cycles for rotating each 32-bit block plus a small overhead for loading the least significant bit and introducing the delay required for storing the results. However, by using this approach we encounter a problem when having to add the current row of the generator matrix to the redundant part. Since both data ports are already occupied, we are not able to load the redundant part and XOR the current row to it without spending additional clock cycles.

Instead we implement the following approach that allows to efficiently rotate g and XOR it to the redundant part at the same time if necessary with only two data ports. As described above, Xilinx BRAMs support the READ_FIRST mode which is able to first

[3]Rotating a 4801-bit vector that is stored in 32-bit cells requires $\lceil 4801/32 \rceil = 151$ plus one additional load for extracting the least significant bit.

read the content of a memory cell and can overwrite the content of the cell with new data in the same clock cycle. After loading the least significant bit, we start reading the first memory cell of g. In the next clock cycle we activate the write signal and store the rotated content of the first cell to the next cell while loading its content. Hence by applying this trick we additionally introduce a rotation of the memory cells. The rotated 32-bit value that was previously stored in memory cell 0 is stored to memory cell 1, the rotated value of memory cell 1 is stored in cell 2 and so on. This requires to wrap the addresses around after accessing the last memory cell and also to keep track of which memory cell holds the beginning of the rotated vector. After one rotation the first 32 bit are stored in memory cell 1 instead of memory cell 0, after the second rotation the first 32 bit are stored in cell 2 and so on. This trick allows us to efficiently rotate a 4801-bit vector using just 153 clock cycles instead of nearly twice as many while using only one data port of the BRAM.

We apply the same trick to the redundant part, even though it does not need to be rotated. This allows us to load a 32-bit block of the redundant part, XOR the corresponding 32-bit block of g to it if the current message bit is set, and store the result while rotating g at the same time. Both operations can work in parallel since they only need one data port each.

After 32 rotations of row g, we XOR the current 32-bit message block with its corresponding 32-bit block of the error vector and store the result. Then we load the next 32-bit message block and store it to a 32-bit register. After processing all message bits the resulting ciphertext can be read out from the BRAM by external components.

Decryption. Decryption first computes the syndrome of the received ciphertext. After resetting the syndrome, we rotate both secret keys using the same trick as for rotating the public key when encrypting. Similarly, we apply the same trick to the syndrome that we applied to the redundant part. The syndrome itself does not need to be rotated, but when adding one or even both rows of the secret key to the syndrome we benefit from the same performance gains as when adding one row of the generator matrix to the redundant part. Due to the similar structure of the syndrome computation and the encoding of a message both take nearly the same amount of clock cycles to finish. If we would not process both parts of the secret key and the ciphertext in parallel the computation would take twice as long.

Testing the syndrome for zero is implemented by computing the binary OR of all 32-bit blocks of the syndrome and comparing the results to zero. To count the number of unsatisfied parity-check equations for a ciphertext bit we load 32-bit blocks of the syndrome and of the current rows of the parity-check matrix blocks. Then we compute the Hamming weight of their binary AND to determine if the corresponding ciphertext bits have to be inverted. The Hamming weight is computed by splitting the 32-bit AND result into five 6-bit chunks and one 2-bit chunk, looking up their Hamming weight from tables, and accumulating the results. We then proceed with the next 32-bit blocks and compute the overall Hamming weights for both processed ciphertext bits in parallel.

Next we load the current rows of the parity-check matrix blocks again and rotate them using our previously described rotation technique. If one or both ciphertext bits caused more than b_i unsatisfied parity-check equations for the current iteration i, we invert the ciphertext bit(s) and XOR one or both rows of the parity-check matrix block to the syndrome while rotating them.

After processing $2 \cdot 32$ ciphertext bits, we store both modified parts of the ciphertext back to the BRAM and load the next 32-bit blocks to two 32-bit registers. After processing the last ciphertext bit, we again compute the binary OR of all 32-bit blocks of the syndrome and check if the result is zero. If it is we notify external components that the plaintext can now be read out, otherwise we repeat the bit-flipping step or signal a decoding error if the maximum number of iterations is exceeded.

Table III. Resource consumption of our lightweight QC-MDPC McEliece implementations on a low-cost Xilinx Spartan-6 XC6SLX4 and on a high-end Xilinx Virtex-6 XC6VLX240T FPGA. All results are obtained post place-and-route.

	Virtex-6 XC6VLX240T		Spartan-6 XC6SLX4	
Aspect	**Encryption**	**Decryption**	**Encryption**	**Decryption**
FFs	120	412	119	413
LUTs	224	568	226	605
Slices	68	148	64	159
BRAM	1	3	1	3
Frequency	334 MHz	318 MHz	213 MHz	186 MHz
Time/Op	2.2 ms	13.4 ms	3.4 ms	23.0 ms

5.3. Implementation Results

In the following we present our implementation results in terms of occupied resources and required cycles for Xilinx FPGAs. Furthermore, we compare our results with previous work.

Implementation Results. The implementation results are obtained post place-and-route (PAR) and are listed in Table III for a low-cost Xilinx Spartan-6 XC6SLX4 (the smallest device in the Spartan-6 family) and for a high-end Xilinx Virtex-6 XC6VLX240T FPGA using Xilinx ISE 14.7. The encoder occupies 64-68 slices and the decoder 148-159 slices on these devices. As detailed in Section 5.1, the encoder uses one BRAM and the decoder uses three BRAMs to store inputs, outputs, and intermediate values. While the resource consumption is similar on both devices, the design naturally runs at a higher clock frequency on the Virtex-6.

To encrypt a message, the following cycles counts are required to perform each operation (cf. Table IV): First we need 151 cycles to load the second half of the error vector into the redundant part. Rotating g and XORing it to the redundant part if the current message bit is set takes 153 cycles and has to be repeated 4801 times. After processing 32 message bits we have to load the next message block and store the previous message XOR the corresponding 32 bits of the error vector which takes 3 cycles and has to be repeated 151 times. Last but not least we have to store the least significant bit of the redundant part which takes one additional cycle. Overall we need $151 + 4801 \cdot 153 + 151 \cdot 3 + 1 = 735,158$ cycles to encrypt a 4801-bit message block. On the Virtex-6 FPGA this translates to 2.2 ms and on the Spartan-6 FPGA to 3.4 ms.

Decryption of a ciphertext requires the following cycles for each operation: Resetting the syndrome is finished after 151 cycles. Computing the syndrome is basically the same operation as encoding a message. It takes 153 cycles to rotate both parts of the secret key by one bit and optionally XORing them to the syndrome, which is again repeated 4801 times. Loading the next two 32-bit ciphertext blocks requires one cycle and is repeated 151 times. Overall, we need $4801 \cdot 153 + 151 = 734,704$ cycles to compute the syndrome. Comparing the syndrome to zero takes 151 cycles. Computing the Hamming weight of the binary AND of the syndrome and the two current rows of the parity-check matrix blocks (i.e., counting the number of unsatisfied parity-check equations) takes 154 cycles and is repeated 4801 times. Loading the next two 32-bit ciphertext blocks takes 2 cycles and is repeated 151 times. After computing the Hamming weight, generating the next row of the parity-check matrix takes 153 cycles, which is also repeated 4801 times. Storing modified ciphertext blocks takes one cycle and is done 151 times before the next two 32-bit ciphertext blocks are loaded. Finally, the syndrome is again compared to zero. In summary, one iteration of the bit-flipping step takes $151 \cdot 2 + 4801 \cdot 154 + 4801 \cdot 153 + 151 + 151 = 1,474,511$ cycles. On average 2.4 decoding iterations are needed for successful decoding, hence our overall average cycle count is $151 + 734,704 + 151 + 2.4 \cdot 1,474,511 = 4,273,832$ cycles. On the Virtex-6 FPGA this translates to 13.4 ms and on the Spartan-6 FPGA to 23.0 ms for decrypting one message block.

Table IV. Required cycles for our lightweight QC-MDPC McEliece en-/decryption cores.

Encoder Operations	Cycles	Decoder Operations	Cycles
Load error vector	151	Reset syndrome	151
Rotate PK & XOR	153	Compute syndrome	734,704
Store & load message	3	Check syndrome	151
		Correct ciphertext bits	1,474,511
Overall average	735,000	Overall average	4,274,000

Comparison. A comparison with our high-performance implementation of QC-MDPC McEliece, other lightweight code-based FPGA implementations as well as lightweight Ring-LWE and RSA implementations is presented in Table V.

A fair comparison between the high-performance and the lightweight QC-MDPC McEliece implementations is difficult since the implementations aim for very different goals. When comparing the occupied resources it is fair to say that the lightweight goal was achieved by requiring less than 250 slices and four BRAMs for a combined en-/decryption core instead of using around 13,000 slices. Hence it is possible to use much smaller and inexpensive devices. Of course the lightweight implementation is beaten in terms of time per operation, but still provides timings in the range of a few milliseconds which seems reasonable for a large number of applications.

Previous lightweight McEliece implementations are based on Goppa codes as presented in [Eisenbarth et al. 2009; Ghosh et al. 2012]. [Eisenbarth et al. 2009] proposed the first lightweight implementation of a code-based cryptosystem (“MicroEliece”) for a Xilinx Spartan-3 FPGA. Since the storage capacity of the FPGA did not suffice, external memory had to be used to store the public key. More recently, [Ghosh et al. 2012] proposed a lightweight McEliece decryption co-processor for Xilinx Spartan-3 and Virtex-5 FPGAs. When comparing previous work to our results it is important to keep in mind that even though all works implement McEliece, they are based on different codes. Decoding Goppa codes requires to implement a very different decoder as for (QC-)MDPC codes.

Compared to the MicroEliece implementation of [Eisenbarth et al. 2009] our implementation uses less resources and performs at about the same speed. However, a direct comparison of the consumed resources is difficult since Spartan-3 FPGAs only offer 4-input LUTs as opposed to Spartan-6 and Virtex-6 devices which offer 6-input LUTs. The structure of a slice has changed as well, newer Xilinx FPGAs offer more resources in each slice. But even when reducing the LUT and slice count of the MicroEliece implementation by 50%, our implementations are still smaller, especially when comparing decryption.

Compared to the McEliece decryption co-processor by [Ghosh et al. 2012] we need around nine times less slices in our implementation but also more time to decrypt. The resource consumption can be compared more or less directly since Virtex-5 and Virtex-6 FPGAs offer similar resources.

Besides resource consumption and efficiency an important criteria for real-world applications is the size of the public key. Here, the quasi-cyclic structure of QC-MDPC codes again shows its advantage by reducing the required storage space for one public key from 63.5 kByte [Ghosh et al. 2012] or even 437.8 kByte [Eisenbarth et al. 2009] to just 0.6 kByte.

Recently, [Pöppelmann and Güneysu 2014] presented a lightweight implementation of the lattice-based Ring-LWE scheme for a Spartan-6 XC6SLX9. Their encryption core requires around 50% more resources but takes less time for one operation. As Ring-LWE decryption does not require complex decoders its implementation requires less resources and less time to complete.

A lightweight modular exponentiation core capable of performing 1024-bit RSA operations is offered by [Helion 2010]. They report a time/operation of 345 ms at a resource consumption of 135 slices and one 18 kBit BRAM on a Spartan-6 device for their smallest implementation TINY32.

Table V. Performance comparison of our lightweight QC-MDPC McEliece (McE) implementations with other lightweight public key encryption implementations. [1] Additionally uses 1 DSP48.

Scheme	Platform	Time/Op	FFs	LUTs	Slices	BRAM
Lightweight McE (enc)	XC6SLX4	3.4 ms	119	226	64	1
Lightweight McE (dec)	XC6SLX4	23.0 ms	413	605	159	3
Lightweight McE (enc)	XC6VLX240T	2.2 ms	120	224	68	1
Lightweight McE (dec)	XC6VLX240T	13.4 ms	412	568	148	3
High-performance McE (enc)	XC6VLX240T	13.7 µs	14,429	9,201	2,924	0
High-performance McE (dec)	XC6VLX240T	125.4 µs	32,974	36,554	10,271	0
[Eisenbarth et al. 2009] (enc)	XC3S1400AN	2.2 ms	804	1,044	668	3
[Eisenbarth et al. 2009] (dec)	XC3S1400AN	21.6 ms	8,977	22,034	11,218	20
[Ghosh et al. 2012] (dec)	XC5VLX110T	0.5 ms	n/a	n/a	1,385	5
[Ghosh et al. 2012] (dec)	XC3S1400AN	1.02 ms	2,505	4,878	2,979	5
[Pöppelmann and Güneysu 2014] (enc)	XC6SLX9	0.9 ms	238	317	95	2^1
[Pöppelmann and Güneysu 2014] (dec)	XC6SLX9	0.4 ms	87	112	32	1^1
RSA [Helion 2010]	Spartan6-3	345 ms	n/a	n/a	135	1

6. QC-MDPC MCELIECE FOR EMBEDDED MICROCONTROLLERS

In this section we present QC-MDPC McEliece for embedded microcontrollers, with a focus on ARM Cortex-M4 CPUs. Our implementations are carefully optimized using a mix of C and Thumb2 assembly and we paid particular attention to make their program flow as well as their execution time independent of secret data.

6.1. Target Platform

The STM32F4 Discovery board is equipped with a STM32F407 microcontroller which features a 32-bit ARM Cortex-M4F CPU with 1 Mbyte flash memory, 192 Kbytes SRAM and a maximum clock frequency of 168 MHz. It sells at roughly the same price of USD 5-10 as the popular 8-bit AVR microcontroller ATxmega256A3, depending on the ordered quantity. Instead of a 8-bit architecture it offers 32-bit, can be clocked at higher frequencies, offers more flash and SRAM storage, comes with DSP and floating point instructions, provides communication interfaces such as CAN-, USB-/ and Ethernet controllers, and has a built-in true random number generator (TRNG).

6.2. Implementing QC-MDPC McEliece for the STM32F407

Our implementations of QC-MDPC McEliece for the STM32F407 microcontroller cover key-generation, encryption, and decryption and aim for a reasonable time/memory trade-off.

Key-Generation. Secret key generation starts by selecting a first row candidate for H_{n_0-1} with w/n_0 set bits. The indices in the range of $0 \leq i \leq r-1$ generated using the microcontroller's TRNG determine which bits are set.

The public key computation requires that $H_{n_0-1}^{-1}$ exists. Hence, we apply the extended Euclidean algorithm to the first row candidate and $x^r - 1$. If the inverse does not exist, we select a new first row candidate for H_{n_0-1} and repeat. If the inverse exists, the first row of H_{n_0-1} is converted into a sparse representation where w/n_0 counters point to the positions of set bits and store them as part of the secret key.

Next, we randomly select first rows for $H_0, \ldots, H_{n_0-2}$ as described for H_{n_0-1}, convert and store them in their sparse representation, and compute $(H_{n_0-1}^{-1} H_i)^T, 0 \leq i \leq n_0-2$. Note, since the matrices involved are quasi-cyclic, the result is quasi-cyclic as well so it suffices to multiply the first rows of $H_{n_0-1}^{-1}$ and H_i. The resulting rows of the generator matrix are not sparse and hence they are stored in full length.

Encryption. Encryption is divided into encoding a message and adding an error of weight t to the resulting codeword. To compute the redundant part of the codeword, set bits in message m select rows of the generator matrix G that have to be XORed. Starting from the first row of the generator matrix, we parse m bit-by-bit and decide whether or not to XOR the current row to the redundant part. Then, the next row is

generated by rotating it one bit to the right and the following message bit is processed. This implementation approach was originally introduced in [Heyse et al. 2013].

One has to be careful, however, to not introduce timing dependencies on secret data. For public key encryption the message bits are the secret data and hence the decision of whether or not to XOR a row of matrix G is dependent on secret information. Our proposed countermeasure is to always perform an addition to the redundant part, independent of whether the corresponding message bit is set. Of course we cannot simply accumulate all rows of the generator matrix, as this would map all messages to the same codeword.

Since the addition of a row of G to the redundant part is done in 32-bit steps on the ARM microcontroller, we use the current message bit m_i to compute a 32-bit mask $(0 - m_i)$. For $m_i = 0$ the mask is zero, otherwise all 32 bits of the mask are set. Before the 32-bit blocks of the current row of G are XORed to the redundant part, we compute the logical AND of them with the mask. This either results in the current row being added if the message bit is set, or in zero being added if the message bit was not set. This leads to a runtime that is independent of the message. Furthermore, as the same instructions are executed for set and cleared message bits, a constant program flow is achieved.

After computing the redundant part of the codeword, we append it to the message and generate t random indicies at which we flip bits (i.e., the error addition) to transform the codeword into a ciphertext. We retrieve the required randomness directly from the microcontroller's internal TRNG. If a microcontroller does not provide a dedicated TRNG, a physical source of entropy can be usually obtained by querying the noise from the least-significant bits of an open-ended or internal ADC, or by using clock-jitter [Hlavac et al. 2010]. This entropy can be then regularly mixed into a common PRNG/conditioning function.

Decryption. For decrypting ciphertexts we implement decoder $\mathcal{D}_2$ as described in Sect. 3.5. In a first step we compute the syndrome, which is a similar operation to encoding a codeword, except for the fact that the secret key is stored in a sparse representation. The ciphertext is split into n_0 parts that correspond to the n_0 blocks of the parity-check matrix. The ciphertext blocks are processed bit-by-bit in parallel. If a ciphertext bit is set, the corresponding row of the parity-check matrix is added to the syndrome otherwise the syndrome remains unchanged. The following rows of the parity-check matrix blocks are generated in the sparse representation by incrementing the counters. If a counter overflows (i.e., the counter value equals r), the counter has to be reset to zero.

Again, we want to avoid timing dependencies on secret data. Checking a counter for an overflow and resetting it depending on its occurrence would leak information about set bits in the secret key. A possible countermeasure would be to simply refrain from rotating the rows of the secret key and instead to store the full parity-check matrix in memory. However, storing H would require $2 \times (4801 \times 4801)$ bit = 5.5 Mbyte. Since this is infeasible on embedded platforms, we are left with protecting the rotation of a row of the secret key.

To protect the secret key rotation, we still use counters that point to set bits in the secret key. After incrementing a counter, we check whether an overflow occurred by comparing the value to the maximum counter value r. We load the negative flag `N` of the program status register and use it to compute a 32-bit mask $(0 - \texttt{N})$. Then we compute the logical AND of the counter value and the mask, before we store the result. If the counter value is smaller than r, the `N` flag is set and the incremented counter value is stored. If the counter value is equal to r, the `N` flag is zero and the counter is reset. This removes the timing dependency of an overflowed counter and uses the same program flow independent of whether a counter is reset or not.

If the computed syndrome $s \neq 0^r$ we proceed by counting how many parity-check equations are violated by a codeword bit. This is given by the number of bits that are set in both the syndrome and the row of the parity-check matrix block that corresponds to the ciphertext bit. If the number of unsatisfied parity-check equations

Table VI. Results of our microcontroller implementations of the QC-MDPC McEliece (McE) cryptosystem. The compiler optimization level was set to `-O2` which results in the best code-size/performance trade-off. [1]Flash and SRAM memory requirements are reported for a combined implementation of key-generation, encryption, and decryption. [2]when using a public exponent of the form $e = 2^{16} + 1$. [3]when using a random exponent.

Scheme	Platform	SRAM	Flash	Cycles/Op	Time/Op
This work [enc]	STM32F407	2,792 Byte[1]	4,044 Byte[1]	6,970,817	42 ms
This work [dec]	STM32F407	2,792 Byte[1]	4,044 Byte[1]	31,403,423	187 ms
This work [keygen]	STM32F407	2,792 Byte[1]	4,044 Byte[1]	154,127,467	917 ms
McE [enc] [Heyse et al. 2013]	ATxmega256	606 Byte	5,496 Byte	26,767,463	836 ms
McE [dec] [Heyse et al. 2013]	ATxmega256	198 Byte	2,218 Byte	86,874,388	2.71 s
McE [enc] [Eisenbarth et al. 2009]	ATxmega256	512 Byte	438 kByte	14,406,080	450 ms
McE [dec] [Eisenbarth et al. 2009]	ATxmega256	12 kByte	130.4 kByte	19,751,094	617 ms
McE [enc] [Heyse 2011]	ATxmega256	3.5 kByte	11 kByte	6,358,400	199 ms
McE [dec] [Heyse 2011]	ATxmega256	8.6 kByte	156 kByte	33,536,000	1.1 s
McE [enc] [Cayrel et al. 2012]	ATxmega256	-	-	4,171,734	130 ms
McE [dec] [Cayrel et al. 2012]	ATxmega256	-	-	14,497,587	453 ms
ECC-P160 [Gura et al. 2004]	ATmega128	282 Byte	3682 Byte	6,480,000	810 ms
RSA-1024 [Gura et al. 2004][2]	ATmega128	930 Byte	6292 Byte	3,440,000	430 ms
RSA-1024 [Gura et al. 2004][3]	ATmega128	930 Byte	6292 Byte	87,920,000	11 s

exceeds threshold b_i, then the ciphertext bit is flipped and the row of the parity-check matrix block is added to the syndrome.

If the syndrome is zero after a decoding round, decoding was successful. Otherwise we continue with further rounds until we reach a defined maximum number of iterations in which case a decoding error is issued.

6.3. Implementation Results

The results of our implementations are listed in Table VI. Encrypting a messages takes 42 ms and decrypting a ciphertext takes 187 ms. Key-generation takes 917 ms on average, but usually key-generation performance is not an issue on small embedded devices since they generate few (if even more than one) key-pair(s) in their lifespan. The combined code of key-generation, encryption, and decryption, requires 4 kByte (0.4%) flash memory and 2.8 kByte (1.4%) SRAM, including the public and the secret key. Since $w << r$ for all QC-MDPC parameter sets, storing the secret key in a sparse representation saves memory and at the same time allows fast row rotations. For the 80-bit parameter set with $n_0 = 2$ only $w = 90$ 16-bit counters are needed to store the secret key (1440 bit instead of 9602 bit).

Compared to the QC-MDPC McEliece implementation in [Heyse et al. 2013], our encryption is 20 times faster and includes the generation/addition of a truly random error vector. Decryption performance is improved by more than an order of magnitude taking 187 ms instead of 2.7 s. Furthermore, our implementations are protected against timing and simple power analysis attacks.

Other McEliece microcontroller implementations based on Goppa [Eisenbarth et al. 2009; Heyse 2011] and Srivastava codes [Cayrel et al. 2012] have much higher memory requirements and all need more time per operation. Microcontroller implementations of ECC and RSA were presented by [Gura et al. 2004] for an ATmega128. While their memory requirements are somewhat similar to our work, en-/decryption again take much longer to complete.

Please note that the microarchitecture of the STM32F407 and the ATxmega/ATmega are completely different – but similarly expensive in terms of cost (which is usually a most relevant factor for practical applications).

7. QC-MDPC MCELIECE ON GENERAL-PURPOSE PROCESSORS

In this section we describe our vectorized implementation of the QC-MDPC McEliece encryption scheme for general-purpose processors. The target platform is an Intel Core i7-4770 running at 3.40 GHz. The CPU is based on the Haswell microarchitecture and provides a true random number generator that complies with the standards NIST SP800-90A, B, and C, FIPS-140-2, and ANSI X9.82 [Intel 2014]. The TRNG has a hardware entropy source that samples thermal noise. The output of this entropy source is used as input for an AES-CBC-MAC operation that outputs the seed of a deterministic random bit generator (DRBG). When calling the RDRAND instruction, 16, 32 or 64 random bits are provided by the DRBG.

As our software implementation supports arbitrary parameter sets, we can easily switch from the 80-bit security parameters to parameters that are designed for 128-bit/256-bit security.

7.1. Vectorized Implementation of QC-MDPC McEliece

We provide a vectorized implementation for modern processors that support the Streaming SIMD Extensions 4 (SSE4) and an unvectorized implementation that can be run on older systems. Our implementations are written in C, yet we use several intrinsic functions to access SSE4 instructions in our vectorized implementation. The following description focuses on the vectorization.

While SSE4 features 128-bit integer vectors, Haswell processors also support the AVX2 instruction set that is capable of handling 256-bit integer vectors. However, to ensure a wider compatibility of our vectorized implementation, we decided to apply SSE4. Additionally, we heavily exploit the carry-less multiplication instruction `CLMUL` to accelerate the implementation. This instruction operates on 128-bit vectors and hence would imply expensive conversions if our implementation would be based on 256-bit vectors.

Key-Generation. Keys are generated similarly as for the microcontroller implementation. We generate a random, invertible first row of $H_{n_0-1} = H_1$ with w/n_0 set bits, a random first row of H_0 and the corresponding first row of G. Since a PC provides more memory than a microcontroller, we do not use a compressed, sparse representation for the secret key. All polynomials are stored in full length and we also generate the complete matrix H to avoid polynomial shifts during decryption. Since the public matrix G has to be transmitted to a communication partner, we do not expand G yet.

Encryption. Encryption of a message starts by first expanding public key G. This speeds up the actual encryption and is done only once. All following encryptions under the same public key reuse the already expanded matrix. In contrast to our other implementations, we rotate the first row by 64 positions and store the result. We repeat this step $\lceil N/64 \rceil$ times and end up with a matrix 64 times smaller than a fully expanded matrix.

When multiplying the message by the public key, the omitted intermediate rotations are performed implicitly by using the `CLMUL` instruction that performs a carryless multiplication of two 64-bit values and returns the 128-bit result. By replacing the bit-by-bit checks with this instruction and working with 128-bit vectors, we are able to accelerate the vector-matrix multiplication by 25 times compared to the unvectorized implementation of this subroutine. Additionally, using the `CLMUL` instruction avoids the previously discussed timing dependency on secret data as the carry-less multiplication is always executed.

Afterwards, we append the computed redundant part to the message and add a truly random error vector of weight t. We generate a 64-bit random number using the `RDRAND` instruction and derive four 14-bit, four 15-bit or three 17-bit indexes for the 80/128/256-bit parameter sets, respectively. If the resulting index i lies in the range $0 \leq i < n$, we flip the codeword bit at index i and repeat until t bits are flipped.

Decryption. Decoder $\mathcal{D}_2$ is used in this implementation. We first compute the syndrome of the ciphertext. Similar to the multiplication of the plaintext by the public

Table VII. Cycle counts of our QC-MDPC McEliece implementations on an Intel Core i7-4770 CPU for 100,000 runs en-/decryption and 1,000 runs for the key generation. The compiler optimization level was set to `-O3` since we aim to optimize our implementation for speed. Turbo Boost and hyperthreading were disabled during measurements.

Operation	80-bit non-vectorized	80-bit SSE4	128-bit SSE4	256-bit SSE4
Key Generation	32,139,668	14,234,347	54,379,733	526,096,652
Encryption	292,432	34,123	106,871	971,605
Decryption	10,114,096	3,104,624	18,825,103	193,922,410
Multiply by public key	267,913	10,742	44,114	478,152
Add Error	2,528	11,761	18,837	50,114
Compute Syndrome	1,178,512	26,654	95,144	959,382
Rotate left by one position	586	115	196	562
Rotate left sparse	288	-	-	-
AND-and-Hamming weight	3,723	123	233	735

key, we employ the `CLMUL` instruction to avoid bit-by-bit checks. Since the secret key matrix has been generated by rotating two independent polynomials, the two halves of H are stored separately. Therefore, we have to pay attention to the center element of the ciphertext. As we are processing 64 bits of the ciphertext at once, the center element has to be multiplied with both, H_0 and H_1. While multiplying the center element by H_0, the bits of the center element that will be multiplied by H_1 have to be set to zero, and vice versa. Compared to the unvectorized implementation of the syndrome computation, we achieve a 44 times better performance with this approach. This even exceeds the speed-up with respect to the multiplication during the encryption, since the unvectorized implementation computes the syndrome using a sparse and compressed representation of H.

The plaintext is recovered similarly to the microcontroller implementation. We check whether the syndrome is zero or not. If not, we identify the number of violated parity check equations for each ciphertext bit. For this purpose, we employ the `POPCNT` instruction that returns the Hamming weight of a 64-bit word. The number of violated equations is then compared to a precomputed threshold. If it exceeds the threshold, we flip the responsible bit and the corresponding row of the secret key matrix is added to the syndrome. In case the syndrome does not reach zero after reaching the maximum number of decoding iterations, we slightly increase the thresholds and start another decryption attempt.

7.2. Implementation Results

Table VII lists the cycle counts of our vectorized and non-vectorized implementations for parameter sets designed for 80/128/256-bit equivalent symmetric security. We selected the parameter sets with $n_0 = 2$ from Table I as they result in the smallest public keys.

As expected, the vectorized implementation turns out to be significantly faster than the non-vectorized implementation. Key generation is accelerated by a factor of 2, encryption is nearly 10 times faster and decryption is 3 times faster. The cycle counts naturally rise for higher security levels. Increasing the security level from 80 to 128 bits incurs a performance penalty of a factor of 3-6$\times$. For a 256-bit security level, the cycle counts are about 10 times higher than for the 128-bit security level. The vectorization speeds up almost all subroutines, except for the error addition which is slower since it uses true random number generation.

Table VIII compares our work with implementations of other public key cryptosystems on similar platforms. The optimized implementation of the KEM/DEM scheme based on the Niederreiter cryptosystem with Goppa codes by [Bernstein et al. 2013] is able to decrypt faster compared to our QC-MDPC implementation. Unfortunately, the cycles counts for key-generation and encryption are not reported. However, for real-world applications, public key sizes still play an important role since they need to be transferred to remote parties. At a security level of 128-bit, QC-MDPC McEliece has

Table VIII. Comparison of our QC-MDPC McEliece PC implementation with other McEliece, RSA, and NTRU implementations. We list the required cycles to en-/decrypt one block as well as the required cycles/byte. * eBACS reports cycles for en-/decrypting 59 bytes. We scaled the cycles/byte metric to the full block size.

Implementation	Platform	Security [bit]	Enc. [cycles]	Dec. [cycles]	Enc. [cyc./byte]	Dec. [cyc./byte]	Block Size [bit]
This work	Haswell	80	34,123	3,104,624	56.86	5,173	4801
This work	Haswell	128	106,871	18,825,103	86.74	15,278	9857
This work	Haswell	256	971,605	193,922,410	237.19	47,340	32771
[Bernstein et al. 2013]	Ivy Bridge	81	-	24,051	-	109.88	1751
[Bernstein et al. 2013]	Ivy Bridge	129	-	60,493	-	134.27	3604
[Bernstein et al. 2013]	Ivy Bridge	263	-	306,102	-	452.40	5413
`mceliece` [eBACS 2014]	Haswell	83	63,522	1,139,808	300*	5,376*	1696
`ronald1024` [eBACS 2014]	Haswell	80	45,452	1,288,172	355*	10,064*	1024
`ronald3072` [eBACS 2014]	Haswell	128	165,832	15,181,669	432*	39,536*	3072
`ntruees787ep1` [eBACS 2014]	Haswell	256	322,240	513,852	4,958*	7,905*	520

public keys of size 1.2 kByte while the Goppa code-based Niederreiter implemented by [Bernstein et al. 2013] has a public key of 221 kByte.

The eBACS benchmarking project [eBACS 2014] contains a McEliece implementation by [Biswas and Sendrier 2008] (`mceliece`), RSA implementations (`ronald1024`, `ronald3072`) and an NTRU implementation (`ntruees787ep1`). Compared to the binary Goppa code McEliece implementation by [Biswas and Sendrier 2008], our implementation operates twice as fast for encryption and around three times slower for decryption. With respect to the cycles per byte metric, QC-MDPC McEliece benefits from its larger block sizes although encrypting large data using public key schemes is a rare use case. Again, public keys are considerably larger than for QC-MDPC. The NTRU implementation is only reported for a 256-bit security level and requires less cycles for one operation at this security level.

8. CONCLUSION

In this work we proposed and reviewed optimized decoders for MDPC codes as well as implementations of QC-MDPC McEliece encryption for reconfigurable hardware, embedded microcontrollers and general-purpose microprocessors. In light of the achieved performance, resource consumption and code size, we conclude that QC-MDPC McEliece encryption can not only provide highly efficient encryption and decryption but also extremely lightweight implementations on a wide range of different platforms.

Acknowledgements

This work was supported in part by the German Federal Ministry of Economics and Technology (Grant 01ME12025 SecMobil). We would like to thank Manuel Bluhm and Stefan Heyse for their support on this work.

REFERENCES

Marco Baldi, Marco Bodrato, and Franco Chiaraluce. 2008. A New Analysis of the McEliece Cryptosystem Based on QC-LDPC Codes. In *SCN (Lecture Notes in Computer Science)*, Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti (Eds.), Vol. 5229. Springer, 246–262.

Marco Baldi and Franco Chiaraluce. 2007. Cryptanalysis of a New Instance of McEliece Cryptosystem Based on QC-LDPC Codes. In *Information Theory, 2007. ISIT 2007. IEEE International Symposium on*. 2591–2595. DOI:http://dx.doi.org/10.1109/ISIT.2007.4557609

M. Baldi, F. Chiaraluce, and R. Garello. 2006. 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*. 305–310. DOI:http://dx.doi.org/10.1109/CCE.2006.350824

M. Baldi, F. Chiaraluce, R. Garello, and F. Mininni. 2007. Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem. In *Communications, 2007. ICC '07. IEEE International Conference on*. 951–956. DOI:http://dx.doi.org/10.1109/ICC.2007.161

Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer. 2012. Decoding Random Binary Linear Codes in 2 n/20: How 1 + 1 = 0 Improves Information Set Decoding. In *EUROCRYPT (Lecture Notes in Computer Science)*, David Pointcheval and Thomas Johansson (Eds.), Vol. 7237. Springer, 520–536.

Thierry P. Berger, Pierre-Louis Cayrel, Philippe Gaborit, and Ayoub Otmani. 2009. Reducing Key Length of the McEliece Cryptosystem. In *Proceedings of the 2nd International Conference on Cryptology in Africa: Progress in Cryptology (AFRICACRYPT '09)*. Springer-Verlag, Berlin, Heidelberg, 77–97. DOI:http://dx.doi.org/10.1007/978-3-642-02384-2_6

E. Berlekamp, R. McEliece, and H. van Tilborg. 1978. On the Inherent Intractability of Certain Coding Problems (Corresp.). *Information Theory, IEEE Transactions on* 24, 3 (may 1978), 384 – 386. DOI:http://dx.doi.org/10.1109/TIT.1978.1055873

Daniel J Bernstein, Tung Chou, and Peter Schwabe. 2013. McBits: fast constant-time code-based cryptography. In *Cryptographic Hardware and Embedded Systems-CHES 2013*. Springer, 250–272.

Bhaskar Biswas and Nicolas Sendrier. 2008. McEliece cryptosystem implementation: Theory and practice. In *Post-Quantum Cryptography*. Springer, 47–62.

Pierre-Louis Cayrel, Gerhard Hoffmann, and Edoardo Persichetti. 2012. Efficient Implementation of a CCA2-Secure Variant of McEliece Using Generalized Srivastava Codes. In *Public Key Cryptography (Lecture Notes in Computer Science)*, Marc Fischlin, Johannes Buchmann, and Mark Manulis (Eds.), Vol. 7293. Springer, 138–155.

Vassil S. Dimitrov, Kimmo U. Järvinen, M. J. Jacobson, W. F. Chan, and Zhun Huang. 2006. FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers. In *CHES (Lecture Notes in Computer Science)*, Louis Goubin and Mitsuru Matsui (Eds.), Vol. 4249. Springer, 445–459.

eBACS 2014. eBACS: ECRYPT Benchmarking of Cryptographic Systems. (2014). Accessed: 2014-06-30 URL: http://bench.cr.yp.to/results-encrypt.html.

Thomas Eisenbarth, Tim Güneysu, Stefan Heyse, and Christof Paar. 2009. MicroEliece: McEliece for Embedded Devices. In *CHES '09: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems*. Springer-Verlag, Berlin, Heidelberg, 49–64.

Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Frdric de Portzamparc, and Jean-Pierre Tillich. 2014a. Folding Alternant and Goppa Codes with Non-Trivial Automorphism Groups. Cryptology ePrint Archive, Report 2014/353. (2014). http://eprint.iacr.org/.

Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, Frdric de Portzamparc, and Jean-Pierre Tillich. 2014b. Structural Cryptanalysis of McEliece Schemes with Compact Keys. Cryptology ePrint Archive, Report 2014/210. (2014). http://eprint.iacr.org/.

Jean-Charles Faugère, Ayoub Otmani, Ludovic Perret, and Jean-Pierre Tillich. 2010. Algebraic Cryptanalysis of McEliece Variants with Compact Keys. In *Advances in Cryptology EUROCRYPT 2010 (Lecture Notes in Computer Science)*, Henri Gilbert (Ed.), Vol. 6110. Springer Berlin Heidelberg, 279–298. DOI:http://dx.doi.org/10.1007/978-3-642-13190-5_14

Robert Gallager. 1962. Low-density Parity-check Codes. *Information Theory, IRE Transactions on* 8, 1 (1962), 21–28.

Santosh Ghosh, Jeroen Delvaux, Leif Uhsadel, and Ingrid Verbauwhede. 2012. A Speed Area Optimized Embedded Co-processor for McEliece Cryptosystem. In *ASAP*. IEEE Computer Society, 102–108.

Tim Güneysu and Christof Paar. 2008. Ultra High Performance ECC over NIST Primes on Commercial FPGAs. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2008 (Lecture Notes in Computer Science)*, Vol. 5154. Springer-Verlag, 62–78.

Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. 2004. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems – CHES 2004 (LNCS)*, Vol. 3156. Springer-Verlag, 925–943.

Helion. 2010. Modular Exponentiation Core Family for Xilinx FPGA. Data Sheet. (June 2010). http://www.heliontech.com/downloads/modexp_xilinx_datasheet.pdf.

Stefan Heyse. 2011. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In *Post-Quantum Cryptography*, Bo-Yin Yang (Ed.). Lecture Notes in Computer Science, Vol. 7071. Springer Berlin / Heidelberg, 143–162.

Stefan Heyse and Tim Güneysu. 2012. Towards One Cycle per Bit Asymmetric Encryption: Code-Based Cryptography on Reconfigurable Hardware, See Prouff and Schaumont [2012], 340–355.

Stefan Heyse, Ingo von Maurich, and Tim Güneysu. 2013. Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In *Cryptographic Hardware and Embedded Systems – CHES 2013 (Lecture Notes in Computer Science)*, Guido Bertoni and Jean-Sébastien Coron (Eds.), Vol. 8086. Springer, Berlin Heidelberg, 273–292.

J. Hlavac, R. Lorencz, and M. Hadacek. 2010. True random number generation on an Atmel AVR microcontroller. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, Vol. 2. V2–493–V2–495. DOI:http://dx.doi.org/10.1109/ICCET.2010.5485568

W C Huffman and V Pless. 2010. Fundamentals of Error-Correcting Codes. (2010).

Intel 2014. Intel Digital Random Number Generator (DRNG). (2014). Accessed: 2014-06-30 URL: https://software.intel.com/sites/default/files/managed/4d/91/DRNG_Software_Implementation_Guide_2.0.pdf.

A.A. Kamal and A.M. Youssef. 2009. An FPGA implementation of the NTRUEncrypt cryptosystem. In *Microelectronics (ICM), 2009 International Conference on*. 209–212. DOI:http://dx.doi.org/10.1109/ICM.2009.5418649

Kazukuni Kobara and Hideki Imai. 2001. Semantically Secure McEliece Public-Key Cryptosystems – Conversions for McEliece PKC–. In *Practice and Theory in Public Key Cryptosystems – PKC '01 (Lecture Notes in Computer Science)*, Kwangjo Kim (Ed.), Vol. 1992. Springer, Berlin Heidelberg, 19–35.

R. J. McEliece. 1978. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report* 44 (Jan. 1978), 114–116.

Rafael Misoczki and Paulo S. Barreto. 2009. Compact McEliece Keys From Goppa Codes. In *Selected Areas in Cryptography: 16th Annual International Workshop (SAC 2009)*. Springer-Verlag, Berlin, Heidelberg, 376–392.

Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. 2012. MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes. Cryptology ePrint Archive, Report 2012/409. (2012). http://eprint.iacr.org/.

Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. 2013. MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes. In *Proceedings of the 2013 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2069–2073.

C. Monico, J. Rosenthal, and A. Shokrollahi. 2000. Using Low Density Parity Check Codes in the McEliece Cryptosystem. In *Information Theory, 2000. Proceedings. IEEE International Symposium on*. 215. DOI:http://dx.doi.org/10.1109/ISIT.2000.866513

H. Niederreiter. 1986. Knapsack-type cryptosystems and algebraic coding theory. *Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform.* 15, 2 (1986), 159–166.

Ryo Nojima, Hideki Imai, Kazukuni Kobara, and Kirill Morozov. 2008. Semantic Security for the McEliece Cryptosystem Without Random Oracles. *Des. Codes Cryptography* 49, 1-3 (2008), 289–305.

Ayoub Otmani, Jean-Pierre Tillich, and Léonard Dallot. 2010. Cryptanalysis of Two McEliece Cryptosystems Based on Quasi-Cyclic Codes. *Mathematics in Computer Science* 3, 2 (2010), 129–140.

Thomas Pöppelmann and Tim Güneysu. 2013. Towards Practical Lattice-Based Public-Key Encryption on Reconfigurable Hardware. In *Selected Areas in Cryptography (Lecture Notes in Computer Science)*, Tanja Lange, Kristin Lauter, and Petr Lisonek (Eds.), Vol. 8282. Springer, 68–85.

Thomas Pöppelmann and Tim Güneysu. 2014. Area Optimization of Lightweight Lattice-Based Encryption on Reconfigurable Hardware. In *International Symposium on Circuits and Systems, ISCAS'14*.

Emmanuel Prouff and Patrick Schaumont (Eds.). 2012. *Cryptographic Hardware and Embedded Systems - CHES 2012 - 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings*. Lecture Notes in Computer Science, Vol. 7428. Springer.

Chester Rebeiro, Sujoy Sinha Roy, and Debdeep Mukhopadhyay. 2012. Pushing the Limits of High-Speed GF(2^m) Elliptic Curve Scalar Multiplication on FPGAs, See Prouff and Schaumont [2012], 494–511.

Sujoy Sinha Roy, Chester Rebeiro, and Debdeep Mukhopadhyay. 2012. A Parallel Architecture for Koblitz Curve Scalar Multiplications on FPGA Platforms. In *DSD*. IEEE, 553–559.

Sujoy Sinha Roy, Frederik Vercauteren, Nele Mentens, Donald Donglong Chen, and Ingrid Verbauwhede. 2013. Compact Hardware Implementation of Ring-LWE Cryptosystems. *IACR Cryptology ePrint Archive* 2013 (2013), 866.

Nicolas Sendrier. 2011. Decoding One Out of Many. In *Post-Quantum Cryptography*, Bo-Yin Yang (Ed.). Lecture Notes in Computer Science, Vol. 7071. Springer Berlin Heidelberg, 51–67. DOI:http://dx.doi.org/10.1007/978-3-642-25405-5_4

Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms On a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509.

Abdulhadi Shoufan, Thorsten Wink, H. Gregor Molter, Sorin A. Huss, and Eike Kohnert. 2010. A Novel Cryptoprocessor Architecture for the McEliece Public-Key Cryptosystem. *IEEE Trans. Computers* 59, 11 (2010), 1533–1546.

Abdulhadi Shoufan, Thorsten Wink, H. Gregor Molter, Sorin A. Huss, and Falko Strenzke. 2009. A Novel Processor Architecture for McEliece Cryptosystem and FPGA Platforms. In *20th IEEE International Conference on Application-specific Systems, Architectures and Processors*. DOI:http://dx.doi.org/10.1109/ASAP.2009.29

Daisuke Suzuki. 2007. How to Maximize the Potential of FPGA Resources for Modular Exponentiation. In *CHES (Lecture Notes in Computer Science)*, Pascal Paillier and Ingrid Verbauwhede (Eds.), Vol. 4727. Springer, 272–288.

Daisuke Suzuki and Tsutomu Matsumoto. 2011. How to Maximize the Potential of FPGA-Based DSPs for Modular Exponentiation. *IEICE Transactions* 94-A, 1 (2011), 211–222.

Ingo von Maurich and Tim Güneysu. 2014. Lightweight Code-based Cryptography: QC-MDPC McEliece Encryption on Reconfigurable Devices. In *Design, Automation and Test in Europe – DATE 2014*. IEEE, 1–6.

Appendix

Table IX. Evaluation of the performance and error correcting capability of the decoders described in Section 3.3 for QC-MDPC codes with parameters $n_0 = 2, n = 9602, r = 4801, w = 90$ on an AMD Opteron 6276 CPU at 2.3 GHz. Note, a failure rate of 0 means that no decoding error occurred during our evaluations. The decoders are still probabilistic and will eventually fail to decode an input.

Variant	#errors	time in ms	failure rate	avg. #iterations
Decoder $\mathcal{A}$	84	32.15	0.0000000	5.2922
	85	33.26	0.0000010	5.4027
	86	34.16	0.0000058	5.5234
	87	34.56	0.0000196	5.6792
	88	34.90	0.0000794	5.8728
	89	36.47	0.0002760	6.1311
	90	38.44	0.0008348	6.4876
Decoder $\mathcal{B}$	84	15.41	0.0002957	3.0936
	85	15.93	0.0012654	3.1854
	86	16.67	0.0046348	3.3343
	87	17.67	0.0138536	3.5515
	88	19.07	0.0360551	3.8790
	89	21.47	0.0798088	4.3542
	90	23.36	0.1534663	5.0191
Decoder $\mathcal{C}_1$	84	25.89	0.0000002	5.2961
	85	26.79	0.0000008	5.4014
	86	27.62	0.0000060	5.5250
	87	28.46	0.0000282	5.6822
	88	28.76	0.0000798	5.8730
	89	29.65	0.0002744	6.1354
	90	31.55	0.0008442	6.4895
Decoder $\mathcal{C}_2$	84	16.03	0.0000000	3.3780
	85	16.60	0.0000000	3.4254
	86	16.90	0.0000000	3.4864
	87	17.47	0.0000000	3.5648
	88	18.01	0.0000002	3.6726
	89	18.88	0.0000026	3.8301
	90	19.96	0.0000098	4.0596
Decoder $\mathcal{C}_3$	84	14.83	0.0000000	3.3776
	85	15.42	0.0000000	3.4263
	86	15.74	0.0000000	3.4871
	87	16.26	0.0000004	3.5656
	88	16.77	0.0000004	3.6736
	89	17.65	0.0000020	3.8308
	90	18.90	0.0000096	4.0602
Decoder $\mathcal{D}_1$	84	8.02	0.0000037	2.4019
	85	8.32	0.0000180	2.4985
	86	8.65	0.0000579	2.5975
	87	8.99	0.0001879	2.6965
	88	9.34	0.0005487	2.7928
	89	9.70	0.0014897	2.8914
	90	10.09	0.0036869	2.9992
Decoder $\mathcal{D}_2$	84	8.79	0.0000000	2.4021
	85	9.00	0.0000000	2.4982
	86	9.40	0.0000000	2.5977
	87	9.57	0.0000000	2.6962
	88	10.07	0.0000000	2.7938
	89	10.32	0.0000000	2.8950
	90	10.26	0.0000002	3.0106
Decoder $\mathcal{D}_3$	84	8.10	0.0000000	2.4021
	85	8.17	0.0000000	2.4975
	86	8.47	0.0000000	2.5964
	87	8.71	0.0000000	2.6964
	88	9.06	0.0000000	2.7941
	89	9.45	0.0000000	2.8948
	90	9.99	0.0000000	3.0109

Table X. Performance comparison of our QC-MDPC FPGA implementations with other public key encryption schemes. [1]Occupied resources and BRAMs are given for a combined encryption and decryption core. [2]Additionally uses 1 DSP48. [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	351.7	4,801	13.7 µs	4,801	351.7	14,429	9,201	2,924	0
This work (dec)	XC6VLX240T	199.3	4,801	82.1 µs	16,363	58.5	41,714	42,274	10,988	0
This work (dec iter.)	XC6VLX240T	222.5	4,801	125.4 µs	27,919	38.3	32,962	36,502	10,364	0
McEliece (enc) [Shoufan et al. 2010]	XC5VLX110T	163	512	500 µs	n/a	1.0	n/a	n/a	14,537	75[1]
McEliece (dec) [Shoufan et al. 2010]	XC5VLX110T	163	512	1,290 µs	n/a	0.4	n/a	n/a	14,537	75[1]
McEliece (dec) [Ghosh et al. 2012]	XC5VLX110T	190	1,751	500 µs	94,249	3.5	n/a	n/a	1,385	5
Niederreiter (enc) [Heyse and Güneysu 2012]	XC6VLX240T	300	192	0.66 µs	200	290.9	875	926	315	17
Niederreiter (dec) [Heyse and Güneysu 2012]	XC6VLX240T	250	192	58.78 µs	14,500	3.3	12,861	9,409	3,887	9
Ring-LWE (enc) [Roy et al. 2013]	XC6VLX75T	313	256	20.1 µs	6,300	12.7	860	1,349	n/a	2[1]
Ring-LWE (dec) [Roy et al. 2013]	XC6VLX75T	313	256	9.1 µs	2,800	28.1	860	1,349	n/a	2[1,2]
Ring-LWE (enc) [Pöppelmann and Güneysu 2013]	XC6VLX75T	262	256	26.2 µs	6,861	9.8	3,624	4,549	1,506	12[1,2]
Ring-LWE (enc) [Pöppelmann and Güneysu 2013]	XC6VLX75T	262	256	16.8 µs	4,404	15.2	3,624	4,549	1,506	12[1,2]
NTRU (enc/dec) [Kamal and Youssef 2009]	XCV1600E	62.3	251	1.54/1.41 µs	96/88	163/178	5,160	27,292	14,352	0
ECC-P224 [Güneysu and Paar 2008]	XC4VFX12	487	224	365.10 µs	177,755	0.6	1,892	1,825	1,580	11[3]
ECC-163 [Rebeiro et al. 2012]	XC5VLX85T	167	163	8.60 µs	1436	18.9	n/a	10,176	3,446	0
ECC-163 [Roy et al. 2012]	Virtex-4	45.5	163	12.10 µs	552	13.4	n/a	n/a	12,430	0
ECC-163 [Dimitrov et al. 2006]	Virtex-II	128	163	35.75 µs	4576	4.6	n/a	n/a	2251	6
RSA-1024 [Suzuki and Matsumoto 2011]	XC5VLX30T	450	1,024	1,520 µs	684,000	0.7	n/a	n/a	3,237	5[4]