# Lightweight Code-based Cryptography: QC-MDPC McEliece Encryption on Reconfigurable Devices

Ingo von Maurich
HGI, Ruhr-Universität Bochum
Germany
ingo.vonmaurich@rub.de

Tim Güneysu
HGI, Ruhr-Universität Bochum
Germany
tim.gueneysu@rub.de

***Abstract*—With the break of RSA and ECC cryptosystems in an era of quantum computing, asymmetric code-based cryptography is an established alternative that can be a potential replacement. A major drawback are large keys in the range between 50 kByte to several MByte 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 a 80-bit security level. Despite of reasonably small key sizes that could also enable small designs, previous work only report high-performance implementations with high resource consumptions of more than 13,000 slices on a large Xilinx Virtex-6 FPGA for a combined en-/decryption unit. In this work we focus on lightweight implementations of code-based cryptography and demonstrate that McEliece encryption using QC-MDPC codes can be implemented with a significantly smaller resource footprint – still achieving reasonable performance sufficient for many applications, e.g., challenge-response protocols or hybrid firmware encryption. More precisely, our design requires just 68 slices for the encryption and around 150 slices for the decryption unit and is able to en-/decrypt an input block in 2.2 ms and 13.4 ms, respectively.**

## I. Introduction

The foundation of virtually all real-world asymmetric cryptosystems are the factoring problem and the (elliptic curve) discrete logarithm problem. In a world with sufficiently powerful quantum computers, both problems can be solved efficiently using Shor's algorithm [1] rendering communication secured by today's asymmetric cryptosystems insecure. Evidently a larger *diversification* and *migration* to other public key primitives is urgently required to prepare security critical systems for this case. As a timeline, IBM recently announced two improvements in quantum computing and estimates that such systems might become practical and available within the next 15 years [2].

The most promising remedies for such a situation are currently relying on code-based, lattice-based, multivariate-quadratic, and hash-based cryptography. A major drawback of many proposed cryptosystems within these classes 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 and lightweight implementations are mandatory. Nevertheless, it was shown that code-based cryptosystems such as the well-established proposals by McEliece [3] and Niederreiter [4] can significantly outperform classical asymmetric cryptosystems on embedded devices [5], [6], [7], [8] – at the cost of very large keys (50-100 kByte for 80-bit security).

Current research is targeting alternative codes that allow more compact key representations but still preserve the security properties of the cryptosystem. Recently, Misoczki et al. proposed to use quasi-cyclic moderate density parity-check (QC-MDPC) codes as such an alternative [9], claiming that a 4,801-bit public key can provide a security level of 80 bit equivalent symmetric security. The scheme was implemented in [10] but the design only aimed for high-throughput. The full potential of this lightweight code-based scheme has not been demonstrated so far.

### A. Contribution

In this work, we present a very lightweight implementation of the McEliece cryptosystem using QC-MDPC codes for Xilinx FPGAs. Since decoding is usually the most expensive operation in code-based cryptosystems, we particularly focus on implementing a lightweight but still efficient variant of the best known decoder for these codes. We show that QC-MDPC codes allow to implement asymmetric cryptography with very few resources while still providing excellent efficiency in terms of computational complexity for encryption and decryption on the FPGA. In addition, this scheme offers practical key sizes of only 4,801 and 9,602 bit for the public and secret key, respectively.

Our lightweight solution can be extremely useful for public key operations that are executed infrequently in the lifetime of long-lasting hardware-based applications (e.g., a key re-establishment or firmware upgrade in elevators or avionic systems). However, we also emphasize that the proposal of QC-MDPC codes is, despite being highly interesting, still novel. We hope to give another incentive for further cryptanalysis of the scheme with the proposal of this very lightweight solution. Furthermore, the source code is freely available to allow verification of our results[1].

### B. Outline

In Section II we provide a background on QC-MDPC codes and the McEliece encryption scheme. Details about designing

978-3-9815370-2-4/DATE14/©2014 EDAA

[1] http://www.sha.rub.de/research/projects/code/

and implementing a lightweight QC-MDPC McEliece are given in Section III. We present our results and compare them to related work in Section IV before we draw a conclusion in Section V.

## II. Background

(QC-)MDPC codes and McEliece encryption based on such codes are fully described in [9]. Below we summarize the description of (QC-)MDPC codes and the McEliece cryptosystem, closely following the notation of the original proposal.

### A. (QC-)MDPC Codes

A binary linear $[n, k]$ code $C$ has length $n$, dimension $k$ and co-dimension $r = n - k$. Code $C$ is defined by its generator matrix $G \in \mathbb{F}_2^{k \times n}$. Messages $m \in \mathbb{F}_2^k$ are transformed into codewords $c \in C$ by computing $c = mG$. The generator matrix is the kernel of parity-check matrix $H \in \mathbb{F}_2^{r \times n}$ and for every codeword $c \in C$ it holds that $s = Hc^T = \mathbf{0}$. The syndrome $s \in \mathbb{F}_2^r$ of any vector $e \in \mathbb{F}_2^n$ can be obtained by computing $s = He^T$.

A linear $[n, k]$ code $C$ is quasi-cyclic (QC), if there exists some integer $n_0$ such that every cyclic shift of a codeword by $n_0$ positions yields again a codeword. If $n = n_0 p$ for some integer $p$, both generator and parity-check matrix are composed of $p \times p$ circulant blocks. The first row of each circulant block is sufficient to completely describe the matrices.

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

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 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. All following rows of the parity-check matrix blocks are obtained by $r - 1$ quasi-cyclic shifts of $h_0, \ldots, h_{n_0-1}$. The parity-check matrix is defined by the concatenation of all parity-check matrix blocks $H = [H_0 | \ldots | H_{n_0-1}]$. The generator matrix $G = [I|Q]$ can be computed from $H$ in row reduced echelon form by concatenating the identity matrix $I$ and matrix

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

### B. QC-MDPC McEliece Encryption

Based on a $t$-error correcting $(n, r, w)$-QC-MDPC code, the McEliece operations key-generation, encryption and decryption are defined as follows:

*1) Key-Generation:* Select the first row $h$ of parity-check matrix $H$ with at most $w$ set bits uniformly at random and generate the corresponding generator matrix $G$ in row reduced echelon form[2]. The public key is matrix $G$ and the secret key is matrix $H$. Since both matrices are quasi-cyclic, it suffices to store the first rows $g$ and $h$ instead of the full matrices.

[2]If this step fails, i.e., if $H_{n_0-1}$ is not invertible, generate a new random first row $h$ and repeat.

*2) Encryption:* A message $m \in \mathbb{F}_2^k$ is encrypted into ciphertext $x \in \mathbb{F}_2^n$ by generating an error vector $e \in \mathbb{F}_2^n$ with at most $t$ set bits uniformly at random and computing $x = mG + e$.

*3) Decryption:* Given a $t$-error correcting (QC-)MDPC decoder $\Psi_H$, compute $mG \leftarrow \Psi_H(x)$. Since $G$ is of systematic form, extract $m$ from the first $k$ positions of $mG$.

Note that the described McEliece cryptosystem slightly deviates from its original form by eliminating permutation matrix $P$ and scrambling matrix $S$. Furthermore, using a CCA2-secure conversion such as [11], [12] allows $G$ to be of systematic form without reducing the security of the scheme.

For a 80-bit security level Misoczki et al. propose to use the following parameters:

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

Hence, a 4801-bit plaintext block is encrypted into a 9602-bit ciphertext and $t = 84$ errors are added. The parity-check matrix $H$ has constant row weight $w = 90$ and consists of $n_0 = 2$ circulant blocks. For a detailed discussion of the security of the proposed system we refer to [9].

Note that co-dimension $r$ was increased by one bit (and hence $n$ by two bits) compared to earlier proposed parameters. According to the authors this is due to an unpublished attack that is possible if $r$ is not prime and hence factorable.

### C. Efficient Decoding

As decoding is the most time-consuming part in code-based cryptography, the selection of an efficient decoding algorithm is crucial to the overall performance. For MDPC codes there exist two families of decoding algorithms [13], [14], out of which the so called bit-flipping algorithms were shown to be the superior choice for handling large codes on embedded platforms [10].

Basically all bit-flipping algorithms follow a similar decoding approach [9], [10], [14], [15]. At first they compute the syndrome of the received codeword, then they count the number of unsatisfied parity-check equations for each codeword bit. If a codeword bit causes more than $b$ unsatisfied parity-check equations, it is flipped. The main difference between the decoders is the way threshold $b$ is determined. Proposals range from using the maximum number of unsatisfied parity-check equations to precomputing the thresholds based on the code parameters.

Recently, all aforementioned decoders were analyzed with respect to their suitability for QC-MDPC codes on embedded platforms in [10]. Decoding algorithm Algo. 1 outperformed its competition both in terms of computation time as well as decoding failure rate.

The decoder precomputes its thresholds $b_i$ based on the code parameters as proposed by Gallager [14] and updates the syndrome while decoding as opposed to other solutions that recompute the syndrome after each iteration.

**Algorithm 1** Decoding (QC-)MPDC Codes

**Input:** $H$, $x = mG + e$, $B = b_0, \ldots, b_{\text{max-1}}$, max
**Output:** Message $m$ or DECODINGFAILURE

```
Compute syndrome s = Hx^T
for i = 0 → max − 1 do
    for every codeword bit j do
        Count unsatisfied parity-check equations #upc
        if #upc ≥ b_i then
            Flip codeword bit x_j
            Update syndrome s = s ⊕ h_j
        end if
    end for
    if s = 0 then
        return x
    end if
end for
return DECODINGFAILURE
```

## III. LIGHTWEIGHT QC-MDPC MCELIECE IMPLEMENTATION

A high-speed implementation of QC-MDPC McEliece at the cost of a high resource consumption was recently published for Xilinx' high-end Virtex-6 XC6VLX240T FPGA. In this section we show how the small keys of QC-MDPC McEliece in combination with embedded block memories of the FPGA can be used to achieve a much smaller and more realistic area footprint implementation that still provides a decent speed sufficient for many applications and can be implemented on Xilinx' low-cost Spartan-6 FPGA family. For fair comparison we also implement our designs on the same Virtex-6 FPGA.

### A. Design Considerations

Intuitively, a small area footprint implementation of QC-MDPC McEliece should be possible due to the comparably small keys. Instead of the need 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 codeword, and the syndrome, with sizes lying in the same range, have to be stored as well. Keeping these parameters in the FPGA logic is possible on large devices, but the high resource consumption is quite expensive.

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 the cells in the same clock cycle after reading their content.

In our design of the encryption and decryption unit we store all inputs, outputs, keys and intermediate values in such 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 units.

*1) Encryption:* During McEliece encryption we have to compute $x = mG + e$ 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 random 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.

*2) Decryption:* McEliece decryption is equal to decoding QC-MDPC codes as described in Sect. II-C. Decoding starts out by computing the syndrome of the received ciphertext, then for each ciphertext bit the number of unsatisfied parity-check equations are counted and if this number exceeds a precomputed threshold, the ciphertext bit is inverted and the corresponding row of the parity-check matrix block is XORed to the syndrome.

For decoding we have to store the secret key (9602 bit), the received ciphertext (9602 bit), and the syndrome (4801 bit). The decoding is performed in-place, i.e., after the decoder is finished the first 4801 bit of the received 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 decided to process them in parallel, in order to not further widening the gap between encryption and decryption performance.

Concerning memory, two 18 kBit BRAMs suffice to store all values. However, 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 in the encryption unit. 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.

### B. Implementation

In the following we detail our implementations of the QC-MDPC McEliece en- and decryption units following the design decisions we reasoned above. Note that the implementation of a CCA2-secure conversion as well as the investigation of a secure implementation of a true random number generator (TRNG) are out of the scope of this work. CCA2 security can be achieved by implementing, e.g., [11], which requires a cryptographic hash function and a TRNG.

*1) Encryption:* Encryption usually starts with setting the redundant part to zero, then the rows of the generator matrix are accumulated depending on the message and 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 can be used to read blocks and the second port can be used to write blocks delayed by one clock cycle. This would require one clock cycles for rotating each 32-bit block and 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 read the content of a memory cell and in the same clock cycle can overwrite the content of the cell with new data. 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 much 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 as the row of the generator matrix. By applying the trick to the redundant part we are able 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.

*2) 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.

Checking if the syndrome is 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 adding the results. We then proceed with the next 32-bit blocks and accumulate the overall hamming weights for both ciphertext bits that are processed in parallel.

[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.

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 unsatisfied parity-check equations (i.e, the computed hamming weight) than are allowed by threshold $b_i$ for the current iteration, we invert the bits and XOR one or both rows of the partiy-check matrix block to the syndrome while rotating them.

After processing 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.

## IV. 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 focusing on QC-MDPC McEliece and lightweight implementations of code-based cryptography.

### A. Implementation Results

Our implementation results are obtained post place-and-route (PAR) and are listed in Table I for a high-end Xilinx Virtex-6 XC6VLX240T and a low-cost Xilinx Spartan-6 XC6SLX4 FPGA (the smallest device in the Spartan-6 family) using Xilinx ISE 14.7.

The encoder occupies 64-68 slices and the decoder 148-159 slices on both devices. As detailed in Section III-A 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.

TABLE I. Resource consumption of our lightweight QC-MDPC McEliece implementations on a high-end Xilinx Virtex-6 XC6VLX240T and a low-cost Xilinx Spartan-6 XC6SLX4 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 |

For encrypting a message, the following cycles counts are required to perform each operation (cf. Table II): 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 block 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. Hence, 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 II. Required cycles for QC-MDPC McEliece en-/decryption.

| **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 |

### B. Comparison

A comparison with a high-speed implementation of QC-MDPC McEliece and other lightweight code-based FPGA implementations is given in Table III.

A fair comparison between the QC-MDPC McEliece implementation of [10] and our work is not possible since the implementation goals are very different. While [10] is aiming for high-speed, we focus on providing a very lightweight asymmetric crypto core. When comparing the occupied resources of both implementations it is fair to say that we achieved our goal by requiring less than 250 slices and four BRAMs for a combined en-/decryption core instead of around 13,000 slices. Hence it is possible to use much smaller and inexpensive devices. Of course our implementation is beaten in terms of time/operation, but we still provide timings in the range of a few milliseconds which is reasonable for a large number of applications.

Previous lightweight McEliece implementations are based on Goppa codes [5], [6]. In [5] the first lightweight implementation of a code-based cryptosystem ("MicroEliece") was proposed 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, [6] proposed a lightweight McEliece decryption co-processor on a Xilinx Spartan-3 and a Virtex-5 FPGA. 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 a Goppa code requires to implement a very different decoder as the one presented in this paper.

Compared to the MicroEliece implementation of [5] 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 [6] we need around nine times less slices in our implementation but also need more time to decrypt. The resource consumption can be compared more or less directly since it is reported for a Virtex-5 FPGA which offers 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 shows its advantage by reducing the required storage space for one public key from 63.5 kByte [6] or even 437.8 kByte [5] to just 0.6 kByte.

A lightweight modular exponentiation core capable of performing 1024-bit RSA operations is offered by Helion [16]. They report a time/op 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 III. PERFORMANCE COMPARISON OF OUR QC-MDPC FPGA IMPLEMENTATIONS WITH OTHER MCELIECE IMPLEMENTATIONS.

| Scheme | Platform | Time/Op | FFs | LUTs | Slices | BRAM |
|---|---|---|---|---|---|---|
| Our (enc) | XC6SLX4 | 3.4 ms | 119 | 226 | 64 | 1 |
| Our (dec) | XC6SLX4 | 23.0 ms | 413 | 605 | 159 | 3 |
| Our (enc) | XC6VLX240T | 2.2 ms | 120 | 224 | 68 | 1 |
| Our (dec) | XC6VLX240T | 13.4 ms | 412 | 568 | 148 | 3 |
| [10] (enc) | XC6VLX240T | 13.7 µs | 14,426 | 8,856 | 2,920 | 0 |
| [10] (dec) | XC6VLX240T | 125.4 µs | 32,974 | 36,554 | 10,271 | 0 |
| [5] (enc) | XC3S1400AN | 2.2 ms | 804 | 1,044 | 668 | 3 |
| [5] (dec) | XC3S1400AN | 21.6 ms | 8,977 | 22,034 | 11,218 | 20 |
| [6] (dec) | XC5VLX110T | 0.5 ms | n/a | n/a | 1,385 | 5 |
| [6] (dec) | XC3S1400AN | 1.02 ms | 2,505 | 4,878 | 2,979 | 5 |
| RSA [16] | Spartan6-3 | 345 ms | n/a | n/a | 135 | 1 |

## V. CONCLUSION

In this work we presented lightweight implementations of the asymmetric cryptosystem McEliece with QC-MDPC codes. In addition to considerably reducing the resource requirements by using embedded block memories that are offered in Xilinx FPGAs, we achieved reasonable performance for both encryption and decryption. Furthermore, the cryptosystem reduces the key sizes to a level that is much more appropriate for real-world usage than the key sizes of previous code-based schemes.

## ACKNOWLEDGMENT

This work was supported by the German Federal Ministry of Economics and Technology (Grant 01ME12025 SecMobil).

## REFERENCES

[1] P. W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms On a Quantum Computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.

[2] K. Chang, "I.B.M. Researchers Inch Toward Quantum Computer," New York Times Article, February 28, 2012, http://www.nytimes.com/2012/02/28/technology/ibm-inch-closer-on-quantum-computer.html?_r=1&hpw.

[3] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory," *Deep Space Network Progress Report*, vol. 44, pp. 114–116, Jan. 1978.

[4] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform.*, vol. 15, no. 2, pp. 159–166, 1986.

[5] T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar, "MicroEliece: McEliece for Embedded Devices," in *CHES*, ser. Lecture Notes in Computer Science, C. Clavier and K. Gaj, Eds., vol. 5747. Springer, 2009, pp. 49–64.

[6] S. Ghosh, J. Delvaux, L. Uhsadel, and I. Verbauwhede, "A Speed Area Optimized Embedded Co-processor for McEliece Cryptosystem," in *Application-Specific Systems, Architectures and Processors (ASAP), 2012 IEEE 23rd International Conference on*, july 2012, pp. 102 –108.

[7] S. Heyse, "Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices," in *Post-Quantum Cryptography*, ser. Lecture Notes in Computer Science, B.-Y. Yang, Ed. Springer Berlin / Heidelberg, 2011, vol. 7071, pp. 143–162.

[8] E. Persichetti, "Compact McEliece Keys based on Quasi-Dyadic Srivastava Codes," *J. Mathematical Cryptology*, vol. 6, no. 2, pp. 149–169, 2012.

[9] R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. L. M. Barreto, "MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes," *IEEE International Symposium on Information Theory*, vol. 2013, 2013.

[10] S. Heyse, I. von Maurich, and T. Güneysu, "Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices," in *CHES*, ser. Lecture Notes in Computer Science, G. Bertoni and J.-S. Coron, Eds., vol. 8086. Springer, 2013, pp. 273–292.

[11] K. Kobara and H. Imai, "Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece," in *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, ser. PKC '01. London, UK: Springer-Verlag, 2001, pp. 19–35.

[12] R. Nojima, H. Imai, K. Kobara, and K. Morozov, "Semantic security for the McEliece cryptosystem without random oracles," *Des. Codes Cryptography*, vol. 49, no. 1-3, pp. 289–305, 2008.

[13] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the Inherent Intractability of Certain Coding Problems (Corresp.)," *Information Theory, IEEE Transactions on*, vol. 24, no. 3, pp. 384 – 386, may 1978.

[14] R. Gallager, "Low-density Parity-check Codes," *Information Theory, IRE Transactions on*, vol. 8, no. 1, pp. 21–28, 1962.

[15] W. C. Huffman and V. Pless, "Fundamentals of Error-Correcting Codes," 2010.

[16] H. T. Inc., "Modular Exponentiation Core Family for Xilinx FPGA," Data Sheet, June 2010, http://www.heliontech.com/downloads/modexp_xilinx_datasheet.pdf.