IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter

Ingo von Maurich[1], Lukas Heberle[1] and Tim Güneysu[2]

[1]Horst Görtz Institute for IT-Security, Ruhr University Bochum, Germany
[2]University of Bremen & DFKI, Germany
{ingo.vonmaurich, lukas.heberle}@rub.de, tim.gueneysu@uni-bremen.de

Abstract. QC-MDPC McEliece attracted significant attention as promising alternative public-key encryption scheme believed to be resistant against quantum computing attacks. Compared to binary Goppa codes, it achieves practical key sizes and was shown to perform well on constrained platforms such as embedded microcontrollers and FPGAs.
However, so far none of the published QC-MDPC McEliece/Niederreiter implementations provide indistinguishability under chosen plaintext or chosen ciphertext attacks. Common ways for the McEliece and Niederreiter encryption schemes to achieve IND-CPA/IND-CCA security are surrounding constructions that convert them into secured schemes. In this work we take a slightly different approach presenting (1) an efficient implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers and (2) the first implementation of Persichetti's IND-CCA hybrid encryption scheme from PQCrypto'13 instantiated with QC-MDPC Niederreiter for key encapsulation and AES-CBC/AES-CMAC for data encapsulation. Both implementations achieve practical performance for embedded microcontrollers, at 80-bit security hybrid encryption takes 16.5 ms, decryption 111 ms and key-generation 386.4 ms.

Keywords: Post-quantum cryptography, code-based public key encryption, hybrid encryption, software, microcontroller

1 Introduction

Shor's quantum algorithm [21] efficiently solves the underlying problem of RSA (factoring) and can be adapted to break ECC and DH (discrete logarithms). Although quantum computers can handle only few qubits so far, the proof-of-concept of Shor's algorithm was verified several times with 143 being the largest number which was factored into its prime factors, yet [23]. In this context the NSA Central Security Service recently announced preliminary plans to transition its Suite B family of cryptographic algorithms to quantum-resistant algorithms in the "not too distant future"[1].

[1] See NSA announcement published at https://www.nsa.gov/ia/programs/suiteb_cryptography/.

The code-based public-key encryption schemes by McEliece [15] and Niederreiter [17] are among the most promising alternatives to RSA and ECC. Their security is based on variants of hard problems in coding theory. McEliece encryption instantiated with *quasi-cyclic moderate density parity-check* (QC-MDPC) codes [7] was introduced in [16], followed by QC-MDPC Niederreiter encryption in [3]. Compared to the original proposal of using McEliece and Niederreiter with binary Goppa codes, QC-MDPC codes allow much smaller keys and were shown to achieve good performance on a variety of platforms [9,12,13,14] combined with improved decoding and implementation techniques.

However, none of the previous implementations took into account that the plain McEliece and Niederreiter cryptosystems do not provide *indistinguishability under adaptive chosen-ciphertext attacks* (IND-CCA), using QC-MDPC codes does not change this fact. McEliece/Niederreiter can be integrated into existing frameworks which provide IND-CPA or IND-CCA security (e.g.,[11,18]). Another approach is to plug Niederreiter into an IND-CCA secure hybrid encryption scheme as recently proposed by Persichetti [20]. It is the first hybrid encryption scheme with assumptions from coding theory and it was proven to provide IND-CCA security and *indistinguishability of keys under adaptive chosen-ciphertext attacks* (IK-CCA) in the random oracle model in [20]. Being a hybrid encryption scheme, it furthermore allows efficient encryption of large plaintexts without requiring to share a symmetric secret key beforehand. Still it is not clear how efficient such a system is in practice, especially when implemented for constrained processors of embedded devices.

Contribution. In this work we provide the first implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers for which we also deploy Persichetti's recent hybrid encryption scheme. We base Persichetti's hybrid encryption scheme on QC-MDPC Niederreiter and extend it to handle arbitrary plaintext lengths.

Outline. We summarize the background on QC-MDPC Niederreiter in Sect. 2. Hybrid encryption with Niederreiter based on [20] is presented in Sect. 3. Our implementation of QC-MDPC Niederreiter for ARM Cortex-M4 microcontrollers is detailed in Sect. 4 followed by our implementation of Persichetti's hybrid encryption scheme in Sect. 5. Results and comparisons are given in Sect. 6. We conclude in Sect. 7.

2 QC-MDPC Codes in a Nutshell

In the following we introduce (QC-)MDPC codes, show how the code-based public-key cryptosystem Niederreiter is instantiated with these codes, and explain efficient decoding of (QC-)MDPC codes.

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 is defined by *generator matrix* $G \in \mathbb{F}_2^{k \times n}$ such that $C = \{mG \in \mathbb{F}_2^n \,|\, m \in \mathbb{F}_2^k\}$. Alternatively, the code is defined by *parity-check matrix* $H \in \mathbb{F}_2^{r \times n}$ such that $C = \{c \in \mathbb{F}_2^n \,|\, Hc^T = 0^r\}$. The syndrome of any vector $x \in \mathbb{F}_2^n$ is $s = Hx^T \in \mathbb{F}_2^r$. By definition, $s = 0$ for all codewords of C.

A code C is called *quasi-cyclic* (QC) if there exists an integer n_0 such that cyclic shifts of codewords $c \in C$ by n_0 positions yield codewords $c' \in C$ of the same code. If $n = n_0 \cdot p$ for some integer p, the generator and parity-check matrices are composed of $p \times p$ circulant blocks. Hence, storing one row of each circulant block fully describes the matrices.

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

2.2 The QC-MDPC Niederreiter Cryptosystem

Using QC-MDPC codes in code-based cryptography was proposed in [16] for the McEliece cryptosystem, a corresponding description of QC-MDPC Niederreiter was published in [3]. We introduce the Niederreiter cryptosystem's key-generation, encryption and decryption based on t-error correcting (n, r, w)-QC-MDPC codes.

QC-MDPC Niederreiter Key-Generation Key-generation requires to generate a (n, r, w)-QC-MDPC code $\mathcal{C}$ with $n = n_0 r$. The private key is a composed parity-check matrix of the form $H = [H_0 \,|\, \ldots \,|\, H_{n_0-1}]$ which exposes a decoding trapdoor. The public key is a systematic parity-check matrix $H' = [H_{n_0-1}^{-1} \cdot H] = [H_{n_0-1}^{-1} \cdot H_0 \,|\, \ldots \,|\, H_{n_0-1}^{-1} \cdot H_{n_0-2} \,|\, I]$ which hides the trapdoor but allows to compute syndromes of the public code.

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 Hamming weight $\sum_{i=0}^{n_0-1} \mathrm{wt}(h_i) = w$ at random and

check that H_{n_0-1} is invertible (which is only possible if the row weight d_v is odd). The parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$ are generated by $r-1$ quasi-cyclic shifts of the first rows $h_0, \ldots, h_{n_0-1}$. Their concatenation yields the private parity-check matrix H. The public systematic parity-check matrix H' is computed by multiplication of $H_{n_0-1}^{-1}$ with all blocks H_i. Since the public and private parity-check matrices H' and H are quasi-cyclic, it suffices to store their first rows or columns instead of the full matrices. The identity part I of the public key is usually not stored.

QC-MDPC Niederreiter Encryption Given a public key H' and a message $m \in \mathbb{Z}/\binom{n}{t}\mathbb{Z}$, encode m into an error vector $e \in \mathbb{F}_2^n$ with $wt(e) = t$. The ciphertext is the public syndrome $s' = He^\intercal \in \mathbb{F}_2^r$.

QC-MDPC Niederreiter Decryption Given a public syndrome $s' \in \mathbb{F}_2^r$, recover its error vector using a t-error correcting (QC-)MDPC decoder Ψ_H with private key H. If $e = \Psi_H(s')$ succeeds, return e and transform it back to message m. On failure of $\Psi_H(s')$ return $\perp$.

Parameters The following parameters are proposed in [16] among others for QC-MDPC McEliece to achieve a 80-bit security level: $n_0 = 2, n = 9602, r = 4801, w = 90, t = 84$. For a 128-bit security level the parameters are $n_0 = 2, n = 19714, r = 9857, w = 142, t = 134$. The same parameters achieve the same security levels for QC-MDPC Niederreiter [3].

By $d_v = w/n_0$ we denote the Hamming weight of each row of the n_0 private parity-check matrix blocks[2]. With these parameters the private parity-check matrix H consists of $n_0 = 2$ circulant blocks, each with constant row weight d_v. The public parity-check matrix H' consists of $n_0 - 1 = 1$ circulant block concatenated with the identity matrix. The public key has a size of r bit and the private key has a size of n bit which can be compressed since it is sparse ($w \ll n$). Plaintexts are encoded into vectors of length n and Hamming weight t, ciphertexts have length r. For a detailed discussion of the security of QC-MDPC McEliece and QC-MDPC Niederreiter we refer to [3,16].

[2] 80-bit: $d_v = 45$, 128-bit: $d_v = 71$. Note that $n_0 = 2$ and w is even for the parameters used in this paper.

2.3 Decoding (QC-)MDPC Codes

Compared to encryption, decryption is a more involved operation in both time and memory. Several decoders were proposed for decoding (QC-)MDPC codes [2,7,9,10,16]. Bit-flipping decoders as introduced by Gallager in [7] were, with some modifications, found to be most suitable for constrained devices [9,13,14]. We transfer the decoder and several optimizations to the QC-MDPC Niederreiter setting and introduce the decoder in its basic form in Algorithm 1 in the Appendix.

The decoder receives a private parity-check matrix H and a public syndrome s' as input and computes the private syndrome $s = H_{n_0-1}s'^{\intercal}$. Decoding then runs in several iterations which in general works as follows: the inner loop iterates over all columns of a block of the private-parity check matrix and counts the number of unsatisfied parity-checks $\#_{\text{upc}}$ by counting the number of shared set bits of each column $H_i[j]$ and the private syndrome s. If $\#_{\text{upc}}$ exceeds a certain threshold[3], the decoder likely has found an error position and inverts the corresponding bit in a zero-initialized error candidate $e_{\text{cand}} \in \mathbb{F}_2^n$, thus the name *bit-flipping* decoder. In addition, we include the optimization of directly updating the syndrome s by addition of $H_i[j]$ in case of a bit-flip as proposed in [9]. It was shown in [9,14] that this modification improves the decoding behavior to take less decoding iterations and to reduce the chance of decoding failures. Furthermore, decoding is accelerated because recomputing the syndrome after every decoding iteration is avoided.

The inner loop is repeated for every block H_i of H until all blocks have been processed. Afterwards the public syndrome of the error candidate is computed and compared to the initial public syndrome s'. On a match, the correct error vector was found and is returned. Otherwise the decoder continues with the next iteration. After a fixed maximum of iterations, decoding is restarted with incremented thresholds as proposed in [14] for QC-MDPC McEliece. The failure symbol $\perp$ is returned if even after δ_{max} threshold adaptations the correct error vector is not found.

3 Hybrid Encryption with Niederreiter

Hybrid encryption schemes were introduced in [5]. They are divided into two independent components: (1) a key encapsulation mechanism (KEM) and (2) a data encapsulation mechanism (DEM). The KEM is a public-key encryption scheme that encrypts a randomly generated symmetric

[3] The bit-flipping thresholds used in Algorithm 1 are precomputed from the code parameters as proposed in [7].

session key under the public key of the intended receiver. The DEM then encrypts the plaintext under the randomly generated session key using a symmetric encryption scheme.

Hybrid encryption is usually beneficial in practice because symmetric encryption is orders of magnitude more efficient than pure asymmetric encryption, especially for large plaintexts. On the other hand sole usage of symmetric schemes is not practical due to the symmetric key distribution problem. Hybrid encryption takes the best of two worlds, efficient symmetric data encryption combined with asymmetric key distribution.

3.1 Constructing Hybrid Encryption from Niederreiter

We introduce the Niederreiter hybrid encryption scheme as proposed in [20]. The authors focus on the realization of an IND-CCA secure KEM and assume an IND-CCA symmetric encryption scheme as DEM.

The Niederreiter KEM Let $\mathcal{F}$ be the family of t-error correcting $[n, k]$-linear codes over $\mathbb{F}_q$ and let n, k, q, t be fixed system parameters. The Niederreiter KEM $\pi_{\text{NR\_KEM}} = (\text{Gen}_{\text{NR\_KEM}}, \text{Enc}_{\text{NR\_KEM}}, \text{Dec}_{\text{NR\_KEM}})$ follows the definition of a generic Niederreiter scheme.

- $\textbf{Gen}_{\textbf{NR\_KEM}}$ Pick a random code $\mathcal{C} \in \mathcal{F}$ with parity-check matrix $H' = (M \,|\, I_{n-k})$. Output H' (or M) as public-key and the private code description Δ as private key.
- $\textbf{Enc}_{\textbf{NR\_KEM}}$ Given a public-key H', generate a random error $e \in_R \mathbb{F}_q^n$ of weight $\text{wt}(e) = t$ and compute its public syndrome $s' = H'e^T$. The symmetric key k of length l_k is generated from e by a key-derivation function (KDF) as $k = (k_1 \,||\, k_2) = \text{KDF}(e, l_k)$. The output is (k, s').
- $\textbf{Dec}_{\textbf{NR\_KEM}}$ Decode ciphertext s' to $e = \Psi_\Delta(s')$ using the private code description Δ and decoding algorithm Ψ. Derive symmetric key $k = \text{KDF}(e, l_k)$ if decoding succeeds. Otherwise, k is set to a pseudo-random string of length l_k, [20] suggests to set $k = \text{KDF}(s', l_k)$.

The Standard DEM Let $\text{Enc}_{k_1}^{\text{SE}}(\cdot)$ and $\text{Dec}_{k_1}^{\text{SE}}(\cdot)$ denote en-/decryption operations of a symmetric encryption scheme under key k_1 and let $\text{Ev}_{k_2}(\cdot)$ denote the evaluation of a keyed message authentication code (MAC) under key k_2 that returns a fixed length message authentication tag τ.

The standard DEM $\pi_{\text{DEM}} = (\text{Enc}_{\text{DEM}}, \text{Dec}_{\text{DEM}})$ is the combination of a symmetric encryption scheme with a message authentication code[4].

- **Enc_{DEM}** Given a plaintext m and key $k = (k_1 \,||\, k_2)$, encrypt m to $T = \text{Enc}_{k_1}^{\text{SE}}(m)$ and compute the message authentication tag $\tau = \text{Ev}_{k_2}(T)$ of ciphertext T under k_2. The output is $c^* = (T \,||\, \tau)$.
- **Dec_{DEM}** Given a ciphertext c^* and key k, split c^* into T, τ and k into k_1, k_2. Then verify the correctness of the MAC by evaluating $\text{Ev}_{k_2}(T) \stackrel{?}{=} \tau$. If the MAC is correct, plaintext $m = \text{Dec}_{k_1}^{\text{SE}}(T)$ is decrypted and returned. In case of a MAC mismatch, $\perp$ is returned.

The Niederreiter Hybrid Encryption Scheme The Niederreiter hybrid encryption scheme $\pi_{\text{HY}} = (\text{Gen}_{\text{HY}}, \text{Enc}_{\text{HY}}, \text{Dec}_{\text{HY}})$ is a combination of the Niederreiter KEM $\pi_{\text{NR\_KEM}}$ with the DEM π_{DEM}.

- **Gen_{HY}** invokes $\text{Gen}_{\text{NR\_KEM}}()$ and returns the generated key-pair.
- **Enc_{HY}** is given plaintext m and public key H' and first invokes $\text{Enc}_{\text{NR\_KEM}}(H')$. The returned symmetric keys k_1 and k_2 are used to encrypt the message to $T = \text{Enc}_{k_1}^{\text{SE}}(m)$ and to compute the authentication tag $\tau = \text{Ev}_{k_2}(T)$. The overall ciphertext is $(s' \,||\, T \,||\, \tau)$.
- **Dec_{HY}** receives ciphertext $(s' \,||\, T \,||\, \tau)$ and invokes $\text{Dec}_{\text{NR\_KEM}}(s')$ to decrypt the symmetric key $k = (k_1 \,||\, k_2)$. Then it verifies the correctness of the MAC by evaluating $\text{Ev}_{k_2}(T) \stackrel{?}{=} \tau$. If the MAC is correct, plaintext $m = \text{Dec}_{k_1}^{\text{SE}}(T)$ is decrypted and returned. In case of a MAC mismatch, $\perp$ is returned.

3.2 QC-MDPC Niederreiter Hybrid Encryption

Our instantiation of the Niederreiter hybrid encryption scheme of [20] realizes the KEM using QC-MDPC Niederreiter as defined in Sect. 2.2. We construct the DEM based on AES so that it is capable of handling arbitrary plaintext lengths compared to the impractical one-time pad DEM used in [20]. We target 80-bit and 128-bit security levels in this work. Hence, our DEM uses AES-128 in CBC-mode for message en-/decryption and AES-128 in CMAC-mode for MAC computation following the *encrypt-then-MAC* paradigm. Furthermore, we employ SHA-256 for key derivation of $(k_1 \,||\, k_2)$ from s'.

[4] In [20], the DEM is simply assumed to be a fixed length one-time pad of the size of m combined with a standardized MAC. Hence, $\text{Enc}_{k_1}^{\text{SE}}(m) = m \oplus k_1$ and $\text{Dec}_{k_1}^{\text{SE}}(T) = T \oplus k_1$ with m, T, k_1 having the same fixed length.

For an overall 256-bit security level, appropriate parameters for QC-MDPC Niederreiter should be used (cf. [16]) combined with AES-256-CBC, AES-256-CMAC, and SHA-512.

Hybrid Key-Generation is simply using QC-MDPC Niederreiter key-generation (cf. Sect. 2.2).

Hybrid Encryption generates a random error vector $e \in_R \mathbb{F}_2^n$ with Hamming weight t, encrypts e using QC-MDPC Niederreiter encryption to s' and derives two 128-bit symmetric sessions keys $k = (k_1 \,||\, k_2) =$ SHA-256(e). Message m is encrypted under k_1 by AES-128 in CBC-mode to T starting from a random initialization vector IV. A MAC tag τ is computed over T under k_2 using AES-128 CMAC. The ciphertext is $(s' \,||\, T \,||\, \tau \,||\, IV)$.

Hybrid Decryption extracts the symmetric session keys k_1, k_2 from the QC-MDPC Niederreiter cryptogram, verifies the provided AES-128 CMAC under k_2 and finally decrypts the symmetric ciphertext using k_1 with AES-128 in CBC-mode. The scheme is illustrated in Figure 1.

Security Proof for the IND-CCA security of the hybrid scheme is given in [20] assuming IND-CCA secure symmetric encryption. Furthermore, it was shown in [5] that it is possible to construct IND-CCA symmetric encryption from IND-CPA symmetric encryption (AES-CBC with random IVs [1]) by combining it with a standard MAC (AES-CMAC).

4 QC-MDPC Niederreiter on ARM Cortex-M4

The implementation of QC-MDPC Niederreiter presented in the following targets ARM Cortex-M4 microcontrollers as they are a common modern representative of embedded computing platforms. Our implementation covers key-generation, encryption, and decryption. Details on the implementations of the hybrid encryption scheme based on QC-MDPC Niederreiter are presented in Sect. 5.

To allow fair comparison with previous work we focus on the same microcontroller that was used to implement QC-MDPC McEliece in [13]. The STM32F417VG microcontroller [22] features an ARM Cortex-M4 CPU with a maximum clock frequency of 168 MHz, 1 MB of flash memory and 192 kB of SRAM. The microcontroller is based on a 32-bit architecture and features built-in co-processors for hardware acceleration of

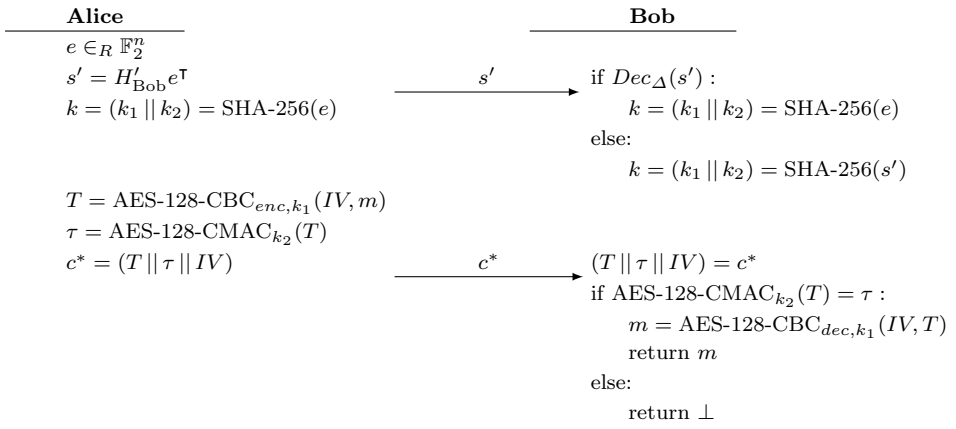

Fig. 1: Alice encrypts plaintext m for Bob using QC-MDPC Niederreiter hybrid encryption with public key H'_{Bob}. Note that we split the transfer of s' and c^* into two steps for illustrative purposes.

AES, Triple DES, MD5, SHA-1 as well as true random number generation (TRNG). Our implementations are written in *Ansi-C* with additional use of Thumb-2 assembly for critical functions. The primary optimization goal is performance, the secondary goal is memory consumption, e.g., we make limited use of unrolling only where it has high performance impacts.

4.1 Polynomial Representation

Our implementations use three different ways for polynomial representation. Each representation has advantages which we exploit in different parts of our implementation.

- *poly_t*: is the naïve way to store a polynomial. It simply stores each bit of the polynomial after each other, its size depends on the polynomial's length and is independent of the polynomial's weight.
- *sparse_t*: stores the positions of set bits of the polynomial. This representation needs less memory than *poly_t* if few bits are set in a polynomial. Furthermore, the *sparse_t* representation allows fast iteration of set bits in the polynomial without having to test all positions.
- *sparse_double_t*: stores the polynomial similarly to the *sparse_t* representation but allocates twice the size of the actually required memory. The yet unused memory is prepended. In addition, it holds a pointer indicating the start of the polynomial. This representation is beneficial when rotating sparse polynomials compared to rotation in *sparse_t*

representation. Its benefits will be explained in more detail when we talk about efficient decoding in Sect. 4.4.

4.2 QC-MDPC Niederreiter Key-Generation

Generating a random first row candidate h_{n_0-1} for block H_{n_0-1} of length r and Hamming weight d_v is done using the microcontroller's TRNG as source of entropy. Its outputs are used as indexes at which we set bits in the polynomial. Since r is prime and hence not a power of two, we use rejection sampling to ensure a uniform distribution of the sampled indexes. The TRNG provides 32 random bits per call but only $\lceil \log_2(r) \rceil$ random bits (13 bit at 80-bit security level, 14 bit at 128-bit security level) are needed to determine an index in the range of $0 \leq i \leq r-1$. Hence we derive two random indexes per TRNG call.

As already stated in Sect. 2.2, we have to ensure that H_{n_0-1} is invertible. We therefor apply the *extended Euclidean algorithm* to newly generated first row candidates until an invertible h_{n_0-1} is found.

We generate the remaining first rows h_i, similar to h_{n_0-1} but skip the inverse checking as only H_{n_0-1} has to be invertible. After private key generation, we compute the corresponding public key which is the systematic parity-check matrix $H' = H_{n_0-1}^{-1} \cdot H = [H_{n_0-1}^{-1} \cdot H_0 | \ldots | I]$, so all we need to do is to compute $H_1^{-1} \cdot H_0$ and append the identity matrix since $n_0 = 2$ in our selected parameter sets. As the private key has few set bits ($d_v \ll r$) we store it in sparse representation. The public key is stored in polynomial representation due to its high density. Since the code is quasi-cyclic, we only need to store the first columns of both matrices. The different representations ease and accelerate later usage.

4.3 QC-MDPC Niederreiter Encryption

Given a public key H' and an error vector[5] $e \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(e) = t$, we compute the public syndrome $s' = H'e^\intercal$. Computing s' is done by iterating over set bits in the error vector and accumulating the corresponding columns of H'. Since the error vector is stored in sparse representation, the index of each bit in the error vector specifies the number of cyclic shifts of the first column of public key H'. To avoid repeated shifting, we reuse the previous shifted column and shift it only by the difference

[5] We do not implement constant weight encoding since it is not needed in the hybrid encryption scheme. Encrypting a message $m \in \mathbb{Z}/\binom{n}{t}\mathbb{Z}$ requires to encode it into an error-vector $e \in \mathbb{F}_2^n$ of weight $\mathrm{wt}(e) = t$ and to reverse the encoding after decryption.

to the next bit index. Multiplication of $e^\intercal$ by the identity part of H' is skipped. As the public syndrome has high density, we store it in *poly_t* representation.

4.4 QC-MDPC Niederreiter Decryption

For decryption we implement two decoder variants, referred to as Dec_A and Dec_B. They differ in their implementation, the decoding behavior of both remains as explained in Sect. 2.3. We start with Dec_A and subsequently look at the improvements made in Dec_B to accelerate decryption. Furthermore, we discuss general implementation optimizations.

$\boldsymbol{Dec_A}$ starts by computing the private syndrome $s = H_{n_0-1}s'^\intercal$ from the public syndrome s' and the private key H. This is basically the same operation as encryption, however we use the *sparse_t* representation for the private key.

Recovery of the error vector e starts from a zero-initialized error candidate e_{cand} of length n. For each column of the private parity-check matrix blocks we observe in how many positions they differ from the private syndrome s, i.e., counting unsatisfied parity-checks. We implement this step by computing the binary *AND* of the current column of the private parity-check matrix block with s followed by a Hamming weight computation of the result. If the Hamming weight exceeds the decoding threshold $b_{\text{iteration}}$, we invert the corresponding bit in e_{cand}. The position is determined by the current column i and block j with $pos = j * r + i$. Additionally, we *XOR* the current column onto the private syndrome for a direct update every time a bit is flipped in e_{cand}. Updating the syndrome while decoding was shown to drastically increase decoding performance in [9,14] for QC-MDPC McEliece, the results similarly apply to QC-MDPC Niederreiter.

We iterate over the private key column by column from the first block to the last by taking the first column of each block and performing successive cyclic shifts. The *sparse_t* representation allows efficient shifting as we only have to increment d_v indexes to effectively shift the polynomial. However, we have to check for overflows of incremented indexes which translate to carry transfers in the regular *poly_t* representation. An overflow results in additional effort, as we have to transfer every value in memory so that the position of the highest bit is always stored in the highest counter.

After iterating over all columns of the private key, we compute the public syndrome of the current error candidate, i.e., we encrypt e_{cand} to

$s'_{\text{cand}} = H'e^{\intercal}_{\text{cand}}$, and compare s'_{cand} to the initial public syndrome s'. On a match, the error vector was found and decryption finishes by returning e. On a mismatch, we continue with the next decoding iteration. After a fixed number of iterations[6], we abort and restart decoding with the original private syndrome and decreased decoding thresholds similar to the optimized decoder for QC-MDPC McEliece presented in [14].

Dec_B The decoding approach of Dec_A has two downsides. First, the public key has to be known during decryption which diverges from standard crypto APIs. Second, costly encryptions have to be performed after each decoding iteration to check whether the current error candidate is the correct error vector. Our decoder Dec_B solves these drawbacks as described in the following.

The first optimization is to transform the private key from *sparse_t* to *sparse_double_t* polynomial representation. This structure allows us to efficiently handle overflows during column rotation. A cyclic shift without carry is equivalent to the *sparse_t* representation in which we increment every bit index of the polynomial. If case of a carry, we pop the last value of the array (with value r), move all array elements by one position, and insert a new value in the beginning (with value 0). We illustrate this operation in Figure 2.

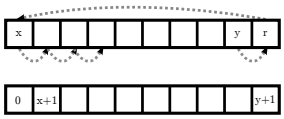

Fig. 2: Carry handling during cyclic rotation in *sparse_t* representation.

Using *sparse_double_t*, we avoid direct manipulation of the array in case of a carry which is the costly part of the *sparse_t* representation. Instead, we decrement the pointer by one and insert a zero at the first element. The last element is ignored since the polynomial has known fixed weight d_v and thereby known length. While the previous approach needs r operations, this approach breaks it down to two operations, independent of the polynomial's length. We illustrate the carry handling in *sparse_double_t* representation in Figure 3.

[6] We found the number of iterations experimentally and set it to five, in line with iteration counts reported in [13,14].

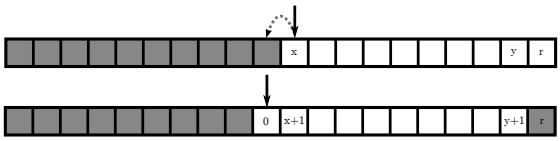

Fig. 3: Carry handling during cyclic rotation in *sparse_double_t* representation. The pointer position is indicated by the black arrow.

Our second optimization checks if the Hamming weight of the error candidate matches the expected Hamming weight $\mathrm{wt}(e) = t$ instead of encrypting e_{cand} after every decoding iteration. If the Hamming weights do not match, we continue with the next decoding iteration immediately. Since Hamming weight computation of a vector is a much cheaper operation than vector matrix multiplication, decryption performance improves.

Our third optimization eliminates the need to encrypt the error candidate to determine whether the correct error vector was found. Instead we test the private syndrome for zero at the end of each decoding iteration. Since the private syndrome is updated every time a bit-flip occurs, it becomes zero once the correct error vector was recovered.

Other general optimizations include writing hot code of the decryption routine in Thumb-2 assembly giving us full control of the executed instructions and allowing us to pay close attention to the instruction execution order to avoid pipeline stalls by interleaving instructions which decreases the number of wasted clock cycles. Furthermore, we store two 16-bit indexes in one 32-bit field of the *sparse_double_t* type[7]. As we indicate the start by a pointer, we do not need to actually shift the values in memory in case of an overflow. A shift by 16 bit would be expensive on a 32-bit architecture. Furthermore, this allows us to increment two values with one `ADD` instruction and we process twice the data with each load and store instruction. To benefit from the burst mode of the load and store instructions (`LDMIA` and `STMIA`), i.e., loading and storing multiple words from/to SRAM, we have to ensure that the memory pointers are 32-bit word aligned. This however is not the case every second overflow since we decrement the *sparse_double_t* pointer in 16-bit steps. To deal with this issue a flag variable is used and, if set, we temporarily decrease the pointer for alignment.

[7] 16 bit are sufficient to store the position for both 80-bit and 128-bit security.

5 QC-MDPC Niederreiter Hybrid Encryption on ARM Cortex-M4

In this section we detail our implementation of the IND-CCA secure QC-MDPC Niederreiter hybrid encryption scheme for ARM Cortex-M4 microcontrollers as introduced in Sect. 3.2. We describe hybrid key-generation, hybrid encryption, as well as hybrid decryption based on our implementation of QC-MDPC Niederreiter (cf. Sect. 4).

5.1 Hybrid Key-Generation

The hybrid encryption scheme requires an asymmetric key-pair for the KEM, and two symmetric keys for the DEM. One symmetric key is used to ensure confidentiality through encryption, the other key is used to ensure message authentication. However, only the asymmetric key pair is permanent, the symmetric keys are randomly generated during encryption. Thus, the implementation of the hybrid key-generation is equal to QC-MDPC Niederreiter key-generation (cf. Sect. 4.2).

5.2 Hybrid Encryption

On input of a plaintext $m \in \mathbb{F}_2^*$ and a QC-MDPC Niederreiter public key H', we generate a random error vector $e \in_R \mathbb{F}_2^n$ with $\mathrm{wt}(e) = t$ using the microcontroller's TRNG and encrypt e under H' using QC-MDPC Niederreiter encryption (cf. Sect. 4.3). Additionally, a hash is derived from e and is split into two 128-bit keys $k = (k_1 \,||\, k_2) = \text{SHA-256}(e)$.

After generation of k_1 and k_2 the key encapsulation is finished and we continue with data encapsulation. We generate a random 16-byte IV using the microcontroller's TRNG and encrypt message m under k_1 to $T = \text{AES-128-CBC}_{enc,k_1}(IV, m)$. Ciphertext T is then fed into AES-128-CMAC, generating a 16-byte tag τ under key k_2. Finally, we concatenate the outputs to $x = (s' \,||\, T \,||\, \tau \,||\, IV)$.

To accelerate AES operations we make use of the AES crypto co-processor featured by the STM32F417 microcontroller for encryption and MAC generation. Unfortunately, the crypto co-processor only offers SHA-1 acceleration which we refrain from to not lower the overall security level. Thus we created a software implementation of SHA-256 for hashing.

5.3 Hybrid Decryption

Hybrid decryption receives ciphertext $x = (s' \,||\, T \,||\, \tau \,||\, IV)$ and decrypts the public syndrome s' using QC-MDPC Niederreiter decryption with

the KEM private key to recover the error vector e (cf. Sect. 4.4). After successful decryption of e, we derive sessions keys k_1 and k_2 by hashing the error vector with SHA-256. We compute the AES-128-CMAC tag τ^* of the symmetric ciphertext T under k_2. If $\tau^* \neq \tau$ we abort decryption, otherwise we AES-128-CBC decrypt T under k_1 to recover plaintext m.

Again we make use of the microcontroller's AES crypto co-processor to accelerate decryption and MAC computation. For SHA-256 we use the same software implementation as during encryption.

6 Implementation Results

In the following we present our implementation results of QC-MDPC Niederreiter and of the hybrid encryption scheme from [20] instantiated with QC-MDPC Niederreiter. Both implementations target ARM Cortex-M4 embedded microcontrollers. We list code size as well as execution time, evaluate the impact of our optimizations and compare the results with previous work. Our code was built with GCC for embedded ARM (arm-eabi v.4.9.3) at optimization level `-O2`.

6.1 QC-MDPC Niederreiter Results

In order to measure the performance of QC-MDPC Niederreiter key-generation, encryption and decryption, we use randomly chosen instances throughout the measurements. We generate 500 random key-pairs and measure for each key-pair 500 en-/decryptions of randomly chosen plaintexts of n-bit length and Hamming weight t, resulting in 250,000 executions over which we average the execution time. Furthermore, we measure cyclic shifting in *poly_t* compared to the sparse polynomial representations to verify our optimizations in more detail. The execution times are listed for 80-bit security, results for 128-bit security are given in parenthesis.

QC-MDPC Niederreiter key-generation takes 376.1 ms (1495.8 ms), encryption 15.6 ms (81.7 ms) and decryption 109.6 ms (477.7 ms) with decoder Dec_B on average. With decoder Dec_A, decryption takes 697.9 ms (3830.2 ms) on average. Both decoders require 2.35 (3.25) decoding iterations on average until decoding succeeds. As embedded microcontrollers usually generate few key pairs in their lifespan, key-generation performance is usually of less practical relevance.

Generating the full private parity-check matrix from its first column in the straightforward *poly_t* representation takes 83.4 ms (345.8 ms). Our *sparse_t* representation accelerates this to 11.6 ms (34.0 ms), even faster

rotations with 7.9 ms (21.2 ms) for the same task are achieved with the *sparse_double_t* representation. By storing private keys in sparse representation with two 16-bit counters in one 32-bit word we reduce the required memory per private key by 85% (88.5%) from 9602 bit (19714 bit) to 1440 bit (2272 bit) compared to simply storing the polynomials in their full length.

The code size of 80-bit QC-MDPC Niederreiter including key-generation, encryption and decryption with Dec_A requires 14 KiB flash memory (1.3%) and additional 4 KiB SRAM (2.0%). For the 128-bit parameter set we need 19 KiB flash memory (1.9%) and 4 KiB SRAM (2.0%). The same implementation with decoder Dec_B requires 16 KiB flash (1.6%) and 3 KiB SRAM (1.5%). For 128-bit security we measured 20 KiB flash memory (2.0%) and 3 KiB SRAM (1.5%) with Dec_B. In Table 1 the code size of each function is listed separately. Note that the sum of the separate code sizes is greater than the combined implementation since we reuse code.

6.2 QC-MDPC Niederreiter Hybrid Encryption Results

The overall execution time of hybrid encryption schemes is dominated by the asymmetric component for key en-/decapsulation. Hence, we focus on QC-MDPC decoder Dec_B for key decapsulation as it operates much faster compared to Dec_A. We generate 500 random key pairs and en-/decrypt 500 randomly chosen plaintexts with a length of 32 byte for each key pair with the hybrid encryption scheme. We measure short plaintexts to get worst-case performance in terms of cycles/byte, longer plaintexts only marginally affect performance since they are only processed by the symmetric components. We list our results for 80-bit security, results for 128-bit security are given in parenthesis.

Key-generation of the hybrid encryption scheme requires 386.4 ms (1511.8 ms), hybrid encryption takes 16.5 ms (83.2 ms), and hybrid decryption 111.0 ms (477.5 ms) on average. Compared to pure QC-MDPC Niederreiter, the symmetric operations (en-/decryption, MACing, hashing) only add very little to the overall execution time ($< 5\%$) although the hybrid encryption scheme seems more complex at first. The AES computations are hardware accelerated which results in further speedup but even if a Cortex-M4 microcontroller without an AES co-processor would be used we would only see a slight increase in the overall execution time. The required code size of the complete hybrid encryption scheme (QC-MDPC Niederreiter, AES-128-CBC, AES-128-CMAC, SHA-256) is 25 KiB flash (2.4%) and 4 KiB SRAM (2.0%) at 80-bit security and 30 KiB flash (2.8%) and 4 KiB SRAM (2.0%) at 128-bit security.

6.3 Comparison with Previous Work

Implementation results reported in other work are listed in Table 1 in the Appendix. A direct comparison of QC-MDPC McEliece [13] with our hybrid QC-MDPC Niederreiter implemented on similar ARM Cortex-M4 microcontrollers shows that hybrid QC-MDPC Niederreiter is around 2.5 times faster at the same security level. In addition it provides IND-CCA security and the possibility to efficiently handle large plaintexts. However, one has to keep in mind that the QC-MDPC McEliece implementation of [13] features constant runtime which adds to its execution time.

Compared to QC-MDPC McEliece implemented on an ATxmega256 [9], our encryption runs 50 times faster and decryption runs 25 times faster, in addition we provide IND-CCA security through hybrid encryption. Comparing implementations on ATxmega256 with implementations on STM32F417 is by no means a fair comparison, however both microcontrollers come at a similar price which makes the comparisons relevant for practical applications.

We refrain from comparing our work to the cyclo-symmetric (CS) MDPC Niederreiter implementation on a PIC24FJ32GA002 microcontroller as presented in [3] because it was shown in [19] that the proposed CS-MDPC parameters do not reach the proclaimed security levels and need adaptation. McEliece implementations based on binary Goppa codes targeting the ATxmega256 microcontroller were presented in [6] and [8]. Again, our implementations outperform both by factors of 5-28. In addition, binary Goppa code public keys are much larger (64 kByte vs. 4801 bit) up to the point of being impractical for embedded devices with constraint memory. The CCA2-secure McEliece implementation based on Srivastava codes presented in [4] also targets the ATxmega256 and is just 4-8 times slower than our hybrid QC-MDPC Niederreiter which seems to make it a good competitor if it would be implemented on the same microcontroller as our work.

7 Conclusion

In this work we presented first implementations of QC-MDPC Niederreiter and of Persichetti's IND-CCA secure hybrid encryption scheme for embedded microcontrollers. We extended the hybrid encryption scheme to handle arbitrary plaintext lengths by choosing well-known symmetric components for data encapsulation and we achieve reasonable performance by combination of new implementation optimizations with transferred known techniques from QC-MDPC McEliece. Furthermore, our

implementations operate with practical key sizes which for a long time was one of the major drawbacks of code-based cryptography.

Acknowledgments

This project has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 645622 (PQCRYPTO). The authors would like to thank Rafael Misoczki for helpful feedback and comments when starting this project.

References

1. M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 394–403. IEEE Computer Society, 1997.
2. E. Berlekamp, R. McEliece, and H. van Tilborg. On the Inherent Intractability of Certain Coding Problems (Corresp.). *Information Theory, IEEE Transactions on*, 24(3):384 – 386, may 1978.
3. F. Biasi, P. Barreto, R. Misoczki, and W. Ruggiero. Scaling efficient code-based cryptosystems for embedded platforms. *Journal of Cryptographic Engineering*, pages 1–12, 2014.
4. P. Cayrel, G. Hoffmann, and E. Persichetti. Efficient Implementation of a CCA2-Secure Variant of McEliece Using Generalized Srivastava Codes. In M. Fischlin, J. Buchmann, and M. Manulis, editors, *Public Key Cryptography - PKC 2012. Proceedings*, volume 7293 of *LNCS*, pages 138–155. Springer, 2012.
5. R. Cramer and V. Shoup. Design and Analysis of Practical Public-Key Encryption Schemes Secure against Adaptive Chosen Ciphertext Attack. *SIAM Journal on Computing*, 33(1):167–226, 2003.
6. T. Eisenbarth, T. Güneysu, S. Heyse, and C. Paar. MicroEliece: McEliece for Embedded Devices. In C. Clavier and K. Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009. Proceedings*, volume 5747 of *LNCS*, pages 49–64. Springer, 2009.
7. R. Gallager. Low-density Parity-check Codes. *Information Theory, IRE Transactions on*, 8(1):21–28, 1962.
8. S. Heyse. Implementation of McEliece Based on Quasi-dyadic Goppa Codes for Embedded Devices. In B. Yang, editor, *Post-Quantum Cryptography - PQCrypto 2011. Proceedings*, volume 7071 of *LNCS*, pages 143–162. Springer, 2011.
9. S. Heyse, I. v. Maurich, and T. Güneysu. Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices. In G. Bertoni and J.-S. Coron, editors, *CHES*, volume 8086 of *Lecture Notes in Computer Science*, pages 273–292. Springer, 2013.
10. W. C. Huffman and V. Pless. Fundamentals of Error-Correcting Codes, 2010.
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*, PKC '01, pages 19–35, London, UK, 2001. Springer-Verlag.

12. I. v. Maurich and T. Güneysu. Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices. In *DATE*, pages 1–6. IEEE, 2014.
13. I. v. Maurich and T. Güneysu. Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices. In M. Mosca, editor, *Post-Quantum Cryptography*, volume 8772 of *Lecture Notes in Computer Science*, pages 266–282. Springer International Publishing, 2014.
14. I. v. Maurich, T. Oder, and T. Güneysu. Implementing QC-MDPC McEliece Encryption. *ACM Transactions on Embedded Computing Systems*, 14(3):1–27, 2015.
15. R. J. McEliece. A Public-Key Cryptosystem Based On Algebraic Coding Theory. *Deep Space Network Progress Report*, 44:114–116, Jan. 1978.
16. R. Misoczki, J.-P. Tillich, N. Sendrier, and P. S. L. M. Barreto. MDPC-McEliece: New McEliece variants from Moderate Density Parity-Check codes. In *ISIT*, pages 2069–2073. IEEE, 2013.
17. H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Problems Control Inform. Theory/Problemy Upravlen. Teor. Inform.*, 15(2):159–166, 1986.
18. R. Nojima, H. Imai, K. Kobara, and K. Morozov. Semantic security for the McEliece cryptosystem without random oracles. *Des. Codes Cryptography*, 49(1-3):289–305, 2008.
19. R. A. Perlner. Optimizing Information Set Decoding Algorithms to Attack Cyclosymmetric MDPC Codes. In M. Mosca, editor, *Post-Quantum Cryptography - 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings*, volume 8772 of *Lecture Notes in Computer Science*, pages 220–228. Springer, 2014.
20. E. Persichetti. Secure and Anonymous Hybrid Encryption from Coding Theory. In P. Gaborit, editor, *Post-Quantum Cryptography*, volume 7932 of *Lecture Notes in Computer Science*, pages 174–187. Springer Berlin Heidelberg, 2013.
21. P. W. Shor. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms On a Quantum Computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.
22. STMicroelectronics. STM32F417VG High-performance foundation line, ARM Cortex-M4 core with DSP and FPU, 1 Mbyte Flash, 168 MHz CPU, ART Accelerator, Ethernet, FSMC, HW crypto - STMicroelectronics. `http://www.st.com/web/en/catalog/mmc/FM141/SC1169/SS1577/LN11/PF252139`, 2015.
23. N. Xu, J. Zhu, D. Lu, X. Zhou, X. Peng, and J. Du. Quantum Factorization of 143 on a Dipolar-Coupling Nuclear Magnetic Resonance System. *Phys. Rev. Lett.*, 108:130501, Mar 2012.

Appendix

Algorithm 1: Syndrome decoder for QC-MDPC codes which returns error vector e or failure $\perp$.

Input H, s', $iterations_{\max}$, $\delta_{\max}$, $threshold$;
Output e ;
Compute the private syndrome $s \leftarrow H_{n_0-1}s'^{\intercal}$;
$\delta \leftarrow 0$;
$e_{\text{cand}} \leftarrow 0^n$;
while $\delta < \delta_{max}$ **do**
    iterations $\leftarrow 0$;
    while $iterations < iterations_{max}$ **do**
        for i *in* n_0 **do**
            for j *in* r **do**
                $hw \leftarrow \text{HammingWeight}(H_i[j]\ \&\ s)$;
                if $hw \geq (threshold[iterations] + \delta)$ **then**
                    $e_{\text{cand}}[i \cdot r + j] \leftarrow e_{\text{cand}}[i \cdot r + j] \oplus 1$;
                    $s \leftarrow H_i[j] \oplus s$;
                end
            end
        end
        $s'_{\text{cand}} \leftarrow H' e^{\intercal}_{\text{cand}}$;
        if $s' = s'_{cand}$ **then**
            return e_{cand};
        end
        iterations++;
    end
    δ++;
    $s \leftarrow H_{n_0-1}s'^{\intercal}$;
end
return $\perp$;

Table 1: Performance and code size of our implementations of QC-MDPC Niederreiter using Dec_B compared to other implementations of similar public-key encryption schemes on embedded microcontrollers. We abbreviate Niederreiter (NR) and McEliece (McE). As code is reused in the combined implementation its size is smaller than the sum of the three separate implementations. [1]Flash and SRAM memory requirements are reported for a combined implementation of key generation, encryption, and decryption.

Scheme	**Platform**	**SRAM [byte]**	**Flash [byte]**	**Cycles/Op**	**Time/Op [ms]**
QC-MDPC NR 80-bit [enc]	STM32F417	2,048	3,064	2,623,432	16
QC-MDPC NR 80-bit [dec]	STM32F417	2,048	8,621	18,416,012	110
QC-MDPC NR 80-bit [keygen]	STM32F417	3,136	8,784	63,185,108	376
QC-MDPC NR 80-bit [combined]	STM32F417	3,136	16,124	-	-
QC-MDPC NR 128-bit [enc]	STM32F417	2,048	4,272	13,725,688	82
QC-MDPC NR 128-bit [dec]	STM32F417	2,048	8,962	80,260,696	478
QC-MDPC NR 128-bit [keygen]	STM32F417	3,136	12,096	251,288,544	1496
QC-MDPC NR 128-bit [combined]	STM32F417	3,136	20,416	-	-
QC-MDPC McE 80-bit [enc] [13]	STM32F407	2,700[1]	5,700[1]	7,018,493	42
QC-MDPC McE 80-bit [dec] [13]	STM32F407	2,700[1]	5,700[1]	42,129,589	251
QC-MDPC McE 80-bit [keygen] [13]	STM32F407	2,700[1]	5,700[1]	148,576,008	884
QC-MDPC McE 80-bit [enc] [9]	ATxmega256	606	5,500	26,767,463	836
QC-MDPC McE 80-bit [dec] [9]	ATxmega256	198	2,200	86,874,388	2,710
Goppa McE [enc] [6]	ATxmega256	512	438,000	14,406,080	450
Goppa McE [dec] [6]	ATxmega256	12,000	130,400	19,751,094	617
Goppa McE [enc] [8]	ATxmega256	3,500	11,000	6,358,400	199
Goppa McE [dec] [8]	ATxmega256	8,600	156,000	33,536,000	1,100
Srivastava McE [enc] [4]	ATxmega256	-	-	4,171,734	130
Srivastava McE [dec] [4]	ATxmega256	-	-	14,497,587	453