A toolbox for software optimization of QC-MDPC code-based cryptosystems

Nir Drucker[1,2] and Shay Gueron[1,2]

[1] University of Haifa, Israel
[2] Amazon Web Services Inc., Seattle, WA, USA
drucker.nir@gmail.com, shay.gueron@gmail.com

Abstract. The anticipated emergence of quantum computers in the foreseeable future drives the cryptographic community to start considering cryptosystems, which are based on problems that remain intractable even with strong quantum computers. One example is the family of code-based cryptosystems that relies on the Syndrome Decoding Problem (SDP). Recent work by Misoczki et al. [34] showed a variant of McEliece encryption which is based on Quasi Cyclic - Moderate Density Parity Check (MDPC) codes, and has significantly smaller keys than the original McEliece encryption. It was followed by the newly proposed QC-MDPC based cryptosystems CAKE [9] and Ouroboros [13]. These motivate dedicated new software optimizations.
This paper lists the cryptographic primitives that QC-MDPC cryptosystems commonly employ, studies their software optimizations on modern processors, and reports the achieved speedups. It also assesses methods for side channel protection of the implementations, and their performance costs. These optimized primitives offer a useful toolbox that can be used, in various ways, by designers and implementers of QC-MDPC cryptosystems.

1 Introduction

Strong quantum computers are believed to emerge in some foreseeable future. Their potential existence threatens the security current public key cryptosystems that rely on the difficulty of factorization (e. g., RSA) and the discrete log problem (e. g., ECDSA/ECDH), and this has already driven the cryptographic community to start planning for new standardized cryptosystems [1]. Families of problems that are believed to provide an adequate basis for quantum safe cryptosystems are based on codes, lattices, hash functions, and multivariate polynomials. It is conceivable that future standards would include representatives from each family, as a safe strategy for the ecosystem. This paper focuses on code-based cryptography.

The first code-based public key cryptosystem, known (after its inventor) as "McEliece" [31], is based on the difficulty of the SDP: identify and correct errors in a given binary vector to which errors have been added. McEliece cryptosystem is believed to be secure, but its very large associated keys challenge its practicality. Similarly, other error-correcting code based cryptosystems also require large

keys, and are therefore considered less practical (e. g., [12, 29, 39]). Introducing some structure to the codes, reduces the key sizes (e. g., [5, 10, 32]), but may also lead to some security issues (e. g., [14, 36]). For example, Low Density Parity Check (LDPC) codes [15] can leverage the structure to improve performance and decrease the key sizes, but seem to be insecure (e. g., [4, 6, 7, 35]).

A recent study by Misoczki et al. [33, 34] suggests that using QC-MDPC codes with a McEliece encryption type cryptosystems enjoy relatively short keys (shorter than McEliece). However, as an encryption scheme, it was shown to be susceptible to an attack [24] by an adversary with some budget of chosen decryption queries. The attack leverages the fact that there is some probability, termed (by [34]) Decoding Failure Rate (DFR), that the decoding may fail to compute the errors. Indeed, [24] showed that merely knowing if decoding failed with some adversary-crafted ciphertexts, allows the adversary to reveal the private key. Nevertheless, a corresponding Key encapsulate Mechanism (KEM) scheme can still be secure, if ephemeral public-private key pairs are used for each exchange, as proposed by the recent KEM scheme CAKE [9]. For 128=bits quantum security, CAKE is estimated to have a DFR of 10^{-7}. Ouroboros [13] is a different QC-MDPC code based KEM scheme. It has the same parameters, almost the same decoding as MDPC McEliece, and a security reduction to decoding random quasi-cyclic codes in the Random Oracle Model [3]. It also has a public key which is roughly $2x$ smaller than that of CAKE and an estimated DFR varying between 10^{-5} to 10^{-7} (depending on the parameters).

The proposed QC-MDPC codes McEliece-style encryption [33, 34] sparked the study of the practical aspects of such cryptosystems, with a thorough study by Maurich et al. [30]. This comprehensive work covers a variety of optimizations, from FPGA's, lightweight architectures (IoT), ARMs Cortex-M4 microcontroller, and vectorization features of modern general-purpose x86 architectures, specifically for a) 80 bits classical security (FPGA/IoT/ARM); b) 40/64/128-bit quantum security (x86 platforms).

In this paper, we investigate general software optimizations for QC-MDPC codes cryptosystems, focusing mainly on 128 bits quantum security parameter as the primary motivation.

Our contribution. We study several software optimizations that leverage special features of modern processor architectures, and compare our results to those achieved in [30]. By listing the main primitives that are needed for the related protocols, we explain these software optimizations can be used (beyond the McEliece-type encryption) for general QC-MDPC codes cryptosystems, such as CAKE [9], and Ouroboros [13]. We show how different parameter choices affect the performance, demonstrate techniques for side channel protection of such implementations, and analyze their costs.

The paper is organized as follows. Section 2 describes QC-MDPC codes, briefly explains the selection of parameters, and defines some notation. Section 3 lists the related primitives, discusses different types of side channel threats, and shows methods for optimizing the relevant building blocks. Section 4 reports the achieved performance improvements. The paper concludes with Section 5.

2 Preliminaries and notation

2.1 QC-MDPC

Let n, r be positive integers. A binary (n, r) linear-code $\mathcal{C}$ of length n, dimension $k = n - r$, and codimension r is a $(n - r)$ dimension vector subspace of $\mathbb{F}_2^n$. It can be defined, equivalently, by either: 1) a generator-matrix $G \in F_2^{k \times n}$, where $\mathcal{C} = \{mG \in \mathbb{F}_2^n | m \in \mathbb{F}_2^k\}$; 2) a parity check matrix $H \in \mathbb{F}_2^{r \times n}$, where $\mathcal{C} = \{c \in \mathbb{F}_2^n | Hc^T = 0\}$. The syndrome $s \in \mathbb{F}_2^r$ of $x \in \mathbb{F}_2^n$ is defined by $s = Hx^T$, where $s = 0$ iff $x \in \mathcal{C}$.

An (n, r)-linear code is called Quasi Cyclic (QC) if there exists an integer n_0, such that every cyclic shift by n_0 positions, of a codeword $c \in \mathcal{C}$, is also a codeword in $\mathcal{C}$. If $n = n_0 p$ for some integer $p \geq 1$, then both the generator and the parity-check matrices are composed of $p \times p$ circulant blocks. These matrices are uniquely defined by the first row of each circulant block. With such interpretation, this row can be viewed as a polynomial in $\mathbb{F}_2^n[x]/(x^r + 1)$. These equivalent interpretations are used hereafter interchangeably.

LDPC and MDPC codes are (n, r, w)-codes that are defined by a parity-check matrix where the Hamming weight of each row is w. While w is small for LDPC codes (e. g., less than 10), MDPC codes typically have $w = O(\sqrt{n \log n})$. An (n, r, w)-QC-MDPC code is an (n, r, w)-MDPC code that is QC where $n = n_0 r$ (typically, $n_0 = 2$).

2.2 Choices of parameters

To illustrate the parameters' range we are interested in, Table 1 shows a few QC-MDPC codes protocols choices, aiming at $\lambda = 128$-bit quantum security. Table 1 unifies the notation of [13, 33, 34] as follows: w is the weight of a sparse r-bits vector, and w_e is the weight of the error vector (their values are derived from r and λ). The usage and rationale behind these choices, are given therein.

Protocol name	r	w	w_e
McEliece-style Encryption [34]	32,771	137	264
CAKE [9]	32,771	137	264
Ouroboros [13]	40,013	147	294
Ouroboros (optimized) [13]	33,997	131	393

Table 1: Some QC-MDPC parameter choices that correspond to $\lambda = 128$-bit quantum security. Here, r defines an $(2r, r)$-code with 2 circulant blocks, w is the weight of a sparse vector, and w_e is the weight of the error vector.

In general, there are conflicting considerations in choosing r (for a given n_0): increasing r makes the associated public keys larger, and lowers the efficiency of the computations; reducing r can lead to a higher DFR.

We also note that r needs to be a prime, and that it is convenient to also impose the following condition [9,34]: $(x^r+1)/(x+1)$ is an irreducible polynomial (of degree $r-1$). This allows for efficient selection of an invertible element of $F_2[x]/(x^r+1)$, by simply taking any polynomial with odd weight.

2.3 Definitions and notation

QC-MDPC cryptosystems operate on polynomials in $\mathbb{F}_2^r/(x^r+1)$. We view polynomials and strings of bits, interchangeably, as follows.

Definition 1 (Bit strings, polynomials.). *A string is a finite sequence of bits (symbols in $\{0,1\}$). Let $r>0$ be an integer, and let A be a string of length r, $A=a_{r-1}a_{r-2}\dots a_0$, where, for $0\le j\le r-1$, a_j is the bit in position j of A. We call a_{r-1} the most significant bit (of A) and a_0 the least significant bit (of A). By convention, the most significant bit is the bit in the leftmost position (in A), and the least significant bit is the bit in the rightmost position. Let $A=a_{r-1}a_{r-2}\dots a_0$ be a string of r bits, and let $a(x)=a_{r-1}x^{r-1}+\dots a_1x+a_0$ be a (formal) polynomial of degree $r-1$. We view A and $a(x)$, interchangeably, as the same entity, depending on the context.*

Example 1. The length of the string $A=10010100$ is $r=8$ (bits), $a_7=1$, $a_6=0$, $a_4=1$, $a_2=1$, $a_0=0$. The polynomial $a(x)=x^7+x^4+x^2$ can be viewed (identified), interchangeably, as the string $A=10010100$.

For designing concrete QC-MDPC implementations, where the polynomials have degree $r-1$ and $8\not| r$, it is useful to embed the associated strings in arrays of bytes. This allows for conveniently carrying out some operations on the embedding arrays. If B is an array of s bytes, then its j^{th} byte ($0\le j\le s-1$) is denoted by $B[j]$. By convention, values of bytes are hereafter written as (exactly) two hexadecimal digits.

Definition 2 (Embedding in an array of bytes.). *Let $A=a_{r-1}a_{r-2}\dots a_0$ be a string, where $8\not| r$. The following procedure embeds A in an array of $\mathsf{NB}=\lceil r/8\rceil$ bytes, $\bar{\mathsf{A}}=\bar{a}_{\mathsf{NB}-1}\bar{a}_{\mathsf{NB}-2}\dots\bar{a}_0$. Write $r=8u+\delta$ with $u\ge 0$ and $0<\delta<8$. Pad A, from the left, with $8-\delta$ zero bits, so that it has $8(u+1)$ bits. Let A' denote the result, i. e.,*

$$A'=a'_{8u+7}\dots a'_0=\underbrace{00\dots0}_{8-\delta\ bits}\,a_{r-1}a_{r-2}\dots a_0$$

The bytes of $\bar{\mathsf{A}}$ are defined, bitwise, as $\bar{a}_j=\bar{\mathsf{A}}[k]=a'_{8k+7}\dots a'_{8k}$, $k=0,\dots,u$. We say that (the bits string) A is embedded in (the array of bytes) $\bar{\mathsf{A}}$.

Example 2. The polynomial $x^{20}+x^{16}+x^2+x+1$ corresponds to the bits string 10001000000000000111 of length $r=21$ bits. Write $r=21=8\times 2+5$, pad A with $\delta=8-5=3$ zero bits, to 000100010000000000000111 (of $21+3=24$ bits). Then, A is embedded in the array of $\mathsf{NB}=\lceil 21/8\rceil=3$ bytes $\bar{\mathsf{A}}=\mathtt{110007}$, whose byte are $\bar{\mathsf{A}}[2]=\mathtt{11}$, $\bar{\mathsf{A}}[1]=\mathtt{00}$, $\bar{\mathsf{A}}[0]=\mathtt{07}$.

Definition 3 (Redundant representation of strings of bits.). *Let* $A = a_{r-1}a_{r-2}\dots a_0$ *be a string of bits. The redundant representation of* A *is the array* $\tilde{\mathsf{A}}$*, of* r *bytes, where* $\tilde{\mathsf{A}}[i] = \mathtt{01}$ *if* $a_i = 1$ *and* $\tilde{\mathsf{A}}[i] = \mathtt{00}$ *otherwise,* $0 \le i \le r-1$.

Example 3. Let A be the string of bits 1011 (i. e., the polynomial $A = x^3+x+1$). Its redundant bytes representation is $\tilde{\mathsf{A}} = \mathtt{01000101}$

Definition 4 (Blocks.). *An array of* 16 *bytes (a string of* 128 *bits) is called a block. Every integer* $0 \le j \le 2^{128}-1$ *can be encoded as a block, which is denoted by* $\mathsf{encode128(j)}$.

Example 4. $\mathsf{encode128(256)} = \mathtt{00000000000000000000000000000100}$.
$\mathsf{encode128}(2^{128}-5) = \mathtt{fffffffffffffffffffffffffffffffb}$.

Definition 5 (Leftwise cyclic rotation.). *Let* B *be an array of* s *bytes and let* $0 \le i \le s-1$ *be an integer. The (leftwise) cyclic rotation of* B*, by* i *positions, is the array* $\mathsf{rotl(B,i)}$ *of* s *bytes, where* $\mathsf{rotl(B,i)}[j] = B[(i+j) \pmod s]$, $j = 0,\dots,s-1$.

Example 5. Let A be the bytes array $\mathtt{11100100}$ of size 4. Then $\mathsf{rotl(A,3)} = \mathtt{00111001}$

Finally, for two integers x_1, x_2, we define the function $\mathsf{compare}(x_1, x_2)$ to return a single bit: 1 if $x_1 = x_2$, and 0 otherwise, denote concatenation by $\|$, and denote the j's column of H by $h_{j\downarrow}$.

3 The QC-MDPC based cryptographic primitives and their optimizations

3.1 The cryptographic primitives

QC-MDPC code-based cryptosystems can be implemented with (combinations of) the following primitives.

1. A constrained pseudorandom bits stream generator: A PRF that uses a seed and generates a stream of bits that satisfies some constraints.
2. A hash function: used for compressing a sampled value of a random variable into a (short) seed/key.
3. Polynomial multiplication in $\mathbb{F}_2$, and reduction modulo $x^r + 1$.
4. Decoding: an algorithm used for computing an unknown error from a given syndrome.

Example 6. CAKE's [9] key generation generates a pseudorandom string of r bits with a prescribed weight. It also involves two polynomial multiplications (modulo $x^r + 1$). Step 4 of CAKE's encapsulation is $K \leftarrow \mathbf{K}(c, e)$, i. e., compressing (hashing) a $4r$ bits into a (short) shared key. Ouroboros' [13] encapsulation calculates $e_r \leftarrow f_{cw}(Hash(r_1, r_2))$. It can be implemented by combining two primitives: a) Hashing $|r_1| + |r_2| = 2r$ bits into a seed; b) Using the seed with a constrained pseudorandom bits stream generator.

Note that some primitives (e. g., decoding) can be optimized "unilaterally" by one of the protocol's participants, and other optimizations (e. g., deriving a shared secret) need to be agreed at the protocol level, for interoperability. In addition, the cryptographic components themselves should the quantum safe (we use $AES256$, and SHA384)

3.2 Side channel considerations

We consider here two types of side channel adversaries that collect information on code that is executing on a given platform.

- Traffic analysis eavesdropper: this adversary has (passive) access to the network that the platform is using, and can collect timing information of different steps of the protocol, based on the observed traffic.
- A spy program adversary: this adversary is running on the same platform, in parallel to the execution of the victim code, and at the same (or lower) privilege level. It can collect micro-architectural information such as memory access patterns and code and taken (not taken) branches.

We assume that both adversaries obtain their information with absolute accuracy. This implies (for a traffic analysis eavesdropper) that the execution time of protocol steps, of a side channel protected code, should not reveal any secret information, and for a spy program, that the timing, memory access patterns, and the branches of a protected code should not reveal secret information. For short, we call such protected code a "constant time" implementation. Mitigation methods against both adversaries are more expensive than mitigation against traffic analysis eavesdropper alone. Therefore, an implementation needs to choose the appropriate threat model that it needs to address.

3.3 A constrained pseudorandom bits stream generator

We handle three types of pseudorandom bits stream generation $z \xleftarrow{\$} \{0,1\}^{\alpha}$ (for an integer $\alpha > 0$): a) No constraints on z (Alg. 1); b) z has odd weight (Alg. 2); c) z has weight w, for some a-priori prescribed weight w (Alg. 3).

The common building block used in Algorithms 1, 2, and 3 is AES-CTR-PRF. It uses the block cipher AES256, in CTR mode (following NIST SP800-90A guidelines [8]). Specifically, a 256-bit seed (seed) is used as a cipher key, and CTR mode is used for populating an array of bytes (of the required length) with (pseudorandom) values. The typical QC-MDPC protocols need pseudorandom streams of roughly no more than say, 2^{17} bits, so the number of calls to AES with a given key is way below the restrictions on using AES in CTR mode. The details and algorithms (Alg. 9 and 10) are given in Appendix A. The AES-CTR-PRF generator is very efficient on modern recent processors that nowadays have dedicated AES instructions (AES-NI). This is especially the case because the computations can be parallelized and pipelined (see [16,17]): AES256 outputs can be produced at the rate of ~ 0.91 cycles per byte (C/B hereafter).

An example for Algorithm 1 is given in Appendix B. Algorithm 2 is then self explanatory. We explain some details of Algorithm 3.

Generating a bit stream A with a pre-defined weight w. This is equivalent to generating a list of weight random positions (wlist) where the bits in the target string of len bits are set. Each entry in wlist is an integer between 0 and len. When $\log_2(\mathsf{len})$ is not an integer, as in the cases of interest here, we first consume a sample of $b = \lceil \log_2(\mathsf{len}) \rceil$ bit from the AES-CTR-PRF. Note that reducing it modulo len does not give a uniform random distribution (small values are more frequent). Therefore, Alg. 3 uses the rejection method: samples that are smaller than len are considered, and other samples are rejected (see also [30]). Since the rejection probability is $p = 1 - \frac{\mathsf{len}}{2^b} < \frac{1}{2}$, the expected number of samples needed to collect w valid values is at most $2w$. Note that the rejection probability is larger when len is slightly larger than b. For example, if $\mathsf{len} = 32,771 = 2^{15} + 3$ then $p \approx 0.5$, and if $\mathsf{len} = 32,749 = 2^{15} - 19$, $p \approx 0.0006$. Sampling for cases where len is not close to a power of two, is optimized in [23], as in the following example.

Example 7. Let $w = 90$, $\mathsf{len} = 9,602$. Then, $b = \lceil \log_2(\mathsf{len}) \rceil = 14$, $p = 0.41$, so the expected number of samples (14 bits) is $90 \times 1/0.59 \approx 153$ ($153 \times 14 = 2142$ bits). However, choosing an upper bound of $3 \times \mathsf{len} = 28,806$ (a sample may require reduction modulo len), and consuming $b = 15$ pseudorandom bits at a time, gives a rejection rate of $p = 0.12$, and reduces the expected number of such samples to $90 \times 1/0.88 = 103$ ($103 \times 15 = 1545$ bits).

Side channel considerations for Algorithms 2 and 3. Alg. 2 runs in constant time, except for step 3. However, the information that step 3 reveals is not confidential. Alg. 3 generates a wlist (in constant time), with set bit positions for the target string. However, naïvely starting from a zero string and flipping the bits in the relevant positions (per wlist) is not secure against a spy program, because the memory access pattern may leak sensitive information (e. g., the secret key in CAKE and Ouroboros). To this end, we propose Alg. 4.

Algorithm 1 z=GenPseudoRand(seed, len)

Input: seed (32 bytes)
Output: $\bar{\mathsf{z}}$ (pseudorandom stream of len bits z embedded in an array of bytes).
Exception: SeedOverUseError (seed overused).

1: **procedure** GenPseudoRand(seed, len)
2: $\quad \mathsf{s} = \text{AES-CTR-PRF-Init}\left(\mathsf{seed}, 0, 2^{32} - 1\right)$
3: $\quad \mathsf{z} = \mathsf{truncate}_{\mathsf{len}}\left(\text{AES-CTR-PRF}\left(\mathsf{s}, \mathsf{len}\right)\right)$
4: $\quad$ **return** $\bar{\mathsf{z}}$

Algorithm 2 z=GenPseudoRandOddWeight(seed, len)

Input: seed (32 bytes), len
Output: $\bar{z}$ (pseudorandom stream of len bits z with odd weight, embedded in an array of bytes).
Exception: SeedOverUseError (seed overused).

```
1: procedure GenPseudoRandOddWeight(seed, len)
2:     z = GenPseudoRand(seed, len)
3:     if weight(z) is even then
4:         z[0] = z[0] ⊕1
5:     return z̄
```

Algorithm 3 wlist=GenPseudoRandWeightList(s, weight, len)

Input: s (AES-CTR-PRF state), weight (32 bits), len
Output: A list (wlist) of weight bit-positions in $[0, \ldots, \text{len} - 1]$, updated s.
Exception: SeedOverUseError (seed overused).

```
1: procedure GenPseudoRandWeightList(s, weight, len)
2:     wlist= φ
3:     valid_ctr = 0
4:     while valid_ctr < weight do
5:         (pos, s) = AES-CTR-PRF(s, 4)
6:         if ((pos < len) AND (pos ∉ wlist)) then
7:             wlist = wlist ∪ {pos}
8:             valid_ctr = valid_ctr + 1
9:     return wlist, s
```

Algorithm 4 A=ApplyWlist(wlist, len)

Input: A list (wlist) of weight bit-positions in $[0, \ldots, \text{len} - 1]$
Output: $\bar{A}$ a stream of len bits A with weight w embedded in an array of bytes.

```
1: procedure ApplyWlist(wlist, len)
2:     A[len:0] = 0
3:     for i in 0 . . . (len − 1) do
4:         for w in wlist do
5:             A[i] = A[i] BitWiseOr compare(i, w)
6:     return Ā
```

3.4 Efficient hashing

A cryptographic hash function is used as a one way function that generates a seemingly uniform random digest from a sampled random variable that has sufficient (min-)entropy. For efficiency, we use a parallelization technique that is designed to convert serial hashing to a parallelizable process (see [18, 19, 22]).

Let hash be a hash function with digest length of ld bytes. Suppose that it uses a compression function compress that consumes a block of size hbs bytes. Its associated "parallelized hash", $\text{ParallelizedHash}^{\text{hash}}_{\text{s,srem}}$ (or ParallelizedHash for short), with s slices, and pre-padding length srem, is described in Alg. 5.

Algorithm 5 digest=ParallelizedHash$^{\mathsf{hash}}_{\mathsf{s,srem}}$(array, la)

Input: an array of la bytes array[la − 1 : 0], such that la ≥ s > 0
Output: digest (ld bytes)
Context: hash, srem
procedure ComputeSliceLen(la)
    tmp := $\left\lfloor \frac{\mathsf{la}}{\mathsf{s}} \right\rfloor$ − srem
    $\alpha := \left\lfloor \frac{\mathsf{tmp}}{\mathsf{hbs}} \right\rfloor$
    return $\alpha \times$ hbs + srem

procedure ParallelizedHash(array, la)
    ls := ComputeSliceLen(la)
    lrem := la - (ls × s)
    for $i := 0$ to (s -1) **do**
        slice[i] = array[$(i+1) \times$ ls − 1 : $i \times$ ls]
        $X[i]$ = hash(slice[i])
    Y = array[la − 1: ls × s]
    YX= Y ∥ X[s − 1] ∥ X[s − 2] . . . ∥ X[0]
    return hash(YX)

The input to ParallelizedHash is an array of la bytes, array[la − 1 : 0]. It is assumed that $0 < \mathsf{s} \leq \mathsf{la}$. The array is split, logically, to s contiguous disjoint slices of equal (positive) length ls, and (potentially) a remainder buffer Y of length la − ls × s bytes (if this value equals 0, it means that Y is ignored). The length of a slice is ls = $\alpha \times$ hbs + srem where

$$\alpha = \left\lfloor \frac{\left\lfloor \frac{\mathsf{la}}{\mathsf{s}} \right\rfloor - \mathsf{srem}}{\mathsf{hbs}} \right\rfloor \tag{1}$$

The slices are denoted slice[s − 1], slice[s − 2], . . . , slice[0]. The s slices are hashed, independently, and s sub-digests $X[\mathsf{s}-1], X[\mathsf{s}-2], \ldots, X[0]$ are computed, by

$$X[j] = \mathsf{hash}(\mathsf{slice}[j]) \quad j = 0, \ldots, \mathsf{s}-1 \tag{2}$$

Finally, the output of ParallelizedHash is digest, where

$$\mathsf{digest} = \mathsf{hash}(\mathsf{Y} \parallel \mathsf{X}[\mathsf{s}-1] \parallel \mathsf{X}[\mathsf{s}-2] \parallel \ \ldots \ \parallel \mathsf{X}[0]) \tag{3}$$

A specific instantiation of ParallelizedHash**.** This paper uses ParallelizedHash$^{\mathsf{SHA384}}_{8,111}$. This implies ld = 48 and hbs = 128. A concrete example is given in Appendix C.

Example 8. Let the length of the array array be la = 2,000 bytes. Choose s = 8, and srem = 111. Each slice has 128 + 111 = 239 bytes. Denote the SHA384 compression function by SHA384Update. Hashing a slice (of 239 bytes) requires

2 invocations of SHA384Update. The 8 (independent) slices can be hashed in parallel. This requires $2 \times 8 = 16$ parallelizable invocations of SHA384Update, generating 8 digests of 48 bytes, each. The remainder block has $2,000 - 239 \times 8 = 88$ bytes. Together with the 8 digests of the slices, the final step is hashing an array of bytes with $384 + 88 = 472$ bytes. This requires $3 + 1 = 4$ invocations of SHA384Update. The total number of SHA384Update invocations of is $16+4 = 20$. For comparison, note that the serial SHA384 of $2,000$ bytes, requires 16 serialized invocations of SHA384Update. However, on modern platforms, computing 20 calls to SHA384Update, of which 16 can be parallelized, can be optimized (using SIMD architectures), and be faster than 16 serial invocations of SHA384Update.

Remark 1 (The choice of srem $= 111$*).* To motivate the choice srem $= 111$, compare it to the choice s $= 8$ and srem $= 0$, applied to an array of la $= 2,000$ bytes. Here, ls $= 128$, lrem $= 2,000 - 8 \times 128 = 976$. Therefore, the number of SHA384Update invocations is $8 \times 2 + 8 = 24$ (of which 16 can be parallelized). The number of SHA384Update invocations with srem $= 111$, is only 20.

3.5 Polynomial multiplication

Modern general-purpose processors are equipped with the "carry-less multiplication" instruction PCLMULQDQ [20, 21]: it computes the product of two binary polynomials of degree 63. The operands of PCLMULQDQ are two xmm registers, an "immediate" byte, and a destination register. The value of the immediate byte specifies which 64-bit halves (low/high) of the input registers are the multiplicands. The result, which is a polynomial of degree 126, is placed in the destination register. An appropriate software flow can use PCLMULQDQ in order to multiply polynomials with any degree.

For sufficiently high degrees, in particular those used for QC-MDPC cryptosystems, applying a Carry-less Karatsuba algorithm ([20, 21]) improves the performance, compared to the standard "Schoolbook" multiplication. We note that other multiplication algorithms (e. g., Toom-Cook) can also contribute some incremental speedups, but we settled here with a "recursive" Karatsuba multiplication. WLOG, the inputs are two polynomials of degree $\mu - 1$. When $\mu - 1$ is even, the inputs are split into two halves, where the Karatsuba multiplication is applied on the corresponding pieces. This procedure can be invoked recursively. The case where $\mu - 1$ is odd needs to be handled differently (and involves some overheads). This implies that the cases where $\mu - 1 = 2^{\alpha}(2p + 1)$ for some $\alpha > 0$ and $p > 0$, leads to an implementation where the first α iterations require no special handling, and this optimizes the code. Obviously, the case where $p = 0$ is of special interest. In our context, given polynomials of degree $r - 1$ (where r is a relatively large prime), it may be useful to "pad" the input into an array of $2^{\lceil \log_2 r \rceil}$ bytes, i. e., artificially increase the number of polynomial coefficients (adding zero coefficients) up to the next power-of-two boundary. The efficiency of this method depends on how small $2^{\lceil \log_2 r \rceil} - r$ is. After sufficient Karatsuba iterations, when the involved polynomial degrees are sufficiently small, it pays

to revert to the schoolbook multiplication for the final step, as shown in the following example.

Example 9. For $r = 4 \times 64$ a naïve schoolbook algorithm uses $4 \times 4 = 16$ PCLMULQDQ instructions and $3 \times 4 = 12$ additions (XORs), and requires 8×64 bits of storage. A recursive Karatsuba calls itself three times with polynomials of degree $2 \times 64 - 1$ each call uses 3 PCLMULQDQ instructions and 5 XORs. This totals 9 PCLMULQDQ and 15 XORs, plus 10 extra XORs for the parent Karatsuba. The required memory is $(8+2+4) \times 64$ bits. On modern processors PCLMULQDQ has throughput 1 cycle and latency 7 cycles. When the schoolbook multiplication is pipelines (PCLMULQDQs invoked in parallel to XORs), it can theoretically end within $16 + 7 + 1 = 24$ cycles (ignoring memory access overhead). On the other hand, a recursive Karatsuba would complete within $9 + 7 + 10 = 27$ cycles (with more memory access overhead).

The final Schoolbook multiplication. We compare two methods, "Horizontal" and "Vertical", for schoolbook multiplication of two polynomials p_1, p_2 of degree $r-1$, each one padded into a $q = ceil((r-1)/64)$ 64-bit container. They are illustrated in Fig. 1.

1. Horizontal multiplication (a variant of this method is used in the gf2x library [37] that NTL library [38] users can choose to link): calculate $p_3[q:0] = p_1[0] \times p_2[q:0]$. Then, perform $p_3[i+q:i] = p_3[i+q:i] + \big(p_1[:i] \times p_2[q:0]\big)$, for $i = 1, \ldots, q-1$.
2. Vertical multiplication (our proposal): for $0 \leq i, j, k \leq q-1$, calculate (in parallel):

$$p_3[2k+1:2k] = \sum_{i+j=2k} p_1[i] \times p_2[j]$$

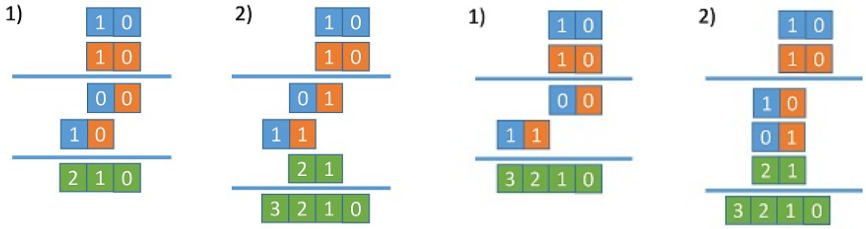
Then, for $k = 0, \ldots, q-2$, perform:

$$p_3[2k+2:2k+1] = p_3[2k+2:2k+1] + \sum_{i+j=2k+1} p_1[i] \times p_2[j]$$

Polynomial reduction modulo x^r+1 in $\mathbb{F}_2$. Reducing a polynomial $p \in \mathbb{F}_2[x]$ of degree r', $r \leq r' \leq 2r$ modulo x^r+1 can be efficiently carried out by calculating $p[r-1:0] = p[r-1:0] \oplus p[r':r]$. Software implementations on modern-processors can be vectorized by using AVX2/AVX512 extensions [26].

3.6 Decoding

We use here the BitFlip algorithm for decoding a QC-MDPC syndrome, proposed by [15] as the baseline for building our optimizations (per [30], other decoding algorithms are more complex). Alg. 6 illustrates a most general variant: it receives the parity check matrix H and a vector x as inputs, and extracts the error



(a) Horizontal multiplication (b) Vertical multiplication

Fig. 1: Schoolbook multiplication (2×2): Horizontal and Vertical. See details in the text.

vector e from x, as the output. Note that other variants (e. g., in [13]) inherit the same general structure. Excluding the first syndrome calculation, Alg. 6 is built upon three repeated main steps: I) For $i = 0, \ldots, r-1$, calculate the number of unsatisfied parity-checks upc_i (step 7); II) Find and compare the thresholds - for $i = 0, \ldots, r-1$, flip the bit in position i of e if $upc_i \geq \tau$ for some threshold τ (steps 8-9); III) Add e to x and recalculate the syndrome s (step 11). Steps I, II, III are repeated until $s = 0$ or until a maximal allowed number of iteration is attained, in which case the algorithm returns a "decoding failure" flag.

Note that there are multiple options to choose τ (the threshold that is used in Alg. 6), and they lead to different DFR and a different number of iterations (i. e., overall performance). Three examples are: 1) a predefined value as in [15]; 2) $\max_i upc_i$ as in [25]; 3) a small value below $\max_i upc_i$ as in [34].

In general, the decoding DFR and performance depend on multiple factors. This has led to various decoding algorithms, and new improved algorithms/-choices are still expected to emerge. We propose below separate optimizations for Steps I and III. Vectorization of Step II is straightforward. Consequently, our optimizations can be used by many variants of the BitFlip decoding algorithm.

BitFlip optimization. Maurich et al. [30] compare several decoders that do not require polynomial multiplications for updating the syndrome (Step III). They all take one of two approaches: a) When finding a bit j for which $upc_j > \tau$, flip $e[j]$, update the syndrome to $s = s + h_{j\downarrow}$, and continue to the next bit. b) Find and correct all the potential error bits in e, and update the syndrome to $s = s + h_{j\downarrow}$ for all the affected bits. The performance of a decoder that utilizes the first approach, on an x86 platform, was reported in [30]. This implementation keeps a copy of $h_{j\downarrow}$ in memory, and rotates and adds it (sometime masked) to s, for each iteration. In this paper, we study decoders that use the second approach.

Optimizing Step I.: Alg. 7 shows our implementation of Step I. The inputs are wlist - a compact representation of $h_{j\downarrow}$ with w indices, and $\tilde{\mathsf{s}}$ - a redundant representation of s. The redundant representation allows us to accumulate the

Algorithm 6 e=BitFlip(x, H)

Input: Parity-check matrix $H \in \mathbb{F}_2^{rxn}$, $x \in \mathbb{F}_2^n$, maxIter (maximal # of iterations).
Output: The error $e \in F_2^n$.
Assumption: A threshold τ is either input or calculated dynamically.
Exception: "decoding failure"

```
1:  procedure BitFlip(x, H)
2:      s = Hx^T;
3:      e = 0;
4:      itr = 0;
5:      while (s ≠ 0) and (itr < maxIter) do
6:          for i in 0 ... n − 1 do
7:              Compute upc_i                                  ▷ Step I
8:              if upc_i > τ then
9:                  e[i] = e[i] ⊕ 1                            ▷ Step II
10:             itr = itr + 1
11:         s = H(x^T + e^T)                                   ▷ Step III
12:     if itr = maxIter then
13:         return "decoding failure"
14:     return e
```

result of (up to 254) additions in each of the $\tilde{\mathsf{s}}$ bytes. Note that in practice, $w < 254$ for QC-MDPC cryptosystems. The output of Alg. 7 is an array of bytes, U, where $U[i] = upc_i$. To speed up rotl($\tilde{\mathsf{s}}$,i), we duplicate $\tilde{\mathsf{s}}$ in memory, into the duplicated syndrome $\tilde{\mathsf{s}}^{\times 2} = \tilde{\mathsf{s}} \| \tilde{\mathsf{s}}$. The rotation rotl($\tilde{\mathsf{s}}$,i) is the memory contents of the r bytes starting from the address of the i-th position, as shown in Fig. 2.

An implementation of Step I can choose two ways to compute the sum of upc_i, $i = 0, \ldots, r-1$: a) Sum for each i separately ("Vertical"; b) sum for all i's in parallel ("Horizontal"). Alg. 7) uses the Vertical summation. To explain why this approach is efficient on modern architectures, let M denote the overall number of bytes that can be stored in the processor's wide-registers. Specifically, $M = 2^9$ bytes for AVX2, and $M = 2^{11}$ bytes for AVX512 architectures. When $r > M$, the array $\tilde{\mathsf{s}}$ is too large to fit in these registers. In this case, Horizontal summation needs to read, accumulate, and store intermediate results in some memory location (for each of the w iterations). This involves $2r$ memory reads plus r memory writes for each iteration - a total of $3wr$ memory operations. By comparison, Vertical summation accumulates the intermediate results in the wide registers, and stores them only in the end of each of the $\lceil r/M \rceil$ iterations. This involves $w \times M$ memory reads and M memory writes (for each iteration) - a total of only $2w + M \times \lceil r/M \rceil \approx 2wr/M$ memory operations. The difference becomes even more noticeable when $2r + M$ bytes do not fit in the last level cache of the processor, which is indeed the case for the typical QC-MDPC r values.

Alignment optimization. An algorithm for CountUPC outputs the array U of r bytes. A Vertical implementation can be optimized to leverage the power of

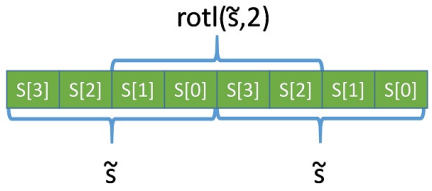


Fig. 2: rotl(s̃,i) example: $\tilde{s} = \tilde{s}[3]\|\tilde{s}[2]\|\tilde{s}[1]\|\tilde{s}[0]$. Duplicating $\tilde{s}$ in memory helps fast rotation (see explanation in the text)

.

Algorithm 7 U=CountUPC(s̃, wlist)

Input: s̃ - a redundant representation of r bits syndrome s, wlist - a w elements of the compact representation of $h_{0\downarrow}$.
Output: U an array of r bytes.

```
1: procedure CountUPC(s̃, wlist)
2:     U=0
3:     for i = 0,...,r − 1 do
4:         for j = 0,...,w − 1 do
5:             U[i] = U[i] + rotl(s̃,wlist[j])[i]
6:     return U
```

SIMD (AVX2 / AVX512) architectures, and parallelize part of the computations. To this end, it needs to operate on $\lfloor r/M \rfloor$ chunks of M bytes, and then handle the remainder "tail" of $\mathsf{tail} = \mathsf{r} - (\lfloor \mathsf{r/M} \rfloor \times \mathsf{M})$ bytes separately. We propose to avoid the cumbersome special handling by padding the duplicated syndrome $\tilde{\mathsf{s}}^{\times 2}$ (from the left) with $M - \mathsf{tail}$ additional zero bytes, and working on the artificially longer array. Obviously, shorter padding leads to smaller overheads, and this makes some values of r preferable over other close values. For example, consider $r = 32,771 = 2^{15} + 3$ (see Table 1), which is slightly above a multiple of M. Padding $\tilde{\mathsf{s}}^{\times 2}$ to the next boundary of $M = 2^9$ bytes (for AVX2), or to $M = 2^{11}$ bytes (for AVX512) adds, respectively, $2^9 - 3 = 509$ or $2^{11} - 3 = 2,045$ bytes. By comparison, consider the close value $r' = 32,749 = 2^{15} - 19$ (smaller than r by only 22), which is slightly below a multiple of M. Here, the padding adds only 19 bytes. This is not saving only memory and memory accesses, but also saves one iteration of the algorithm. It can theoretically improve the performance by $(32,771 + 2,045)/(32,749 + 19) = 1.06$x. Similarly, $r'' = 32,719 = 2^{15} - 49$ (smaller than r by only 52), requires padding of only 49 bytes, leading to savings in memory and computations. Remarkably, it turns out that r' and r'', which are the two largest primes smaller than 2^{15}, also satisfy the requirement that $(x^r + 1)/(x + 1)$ is irreducible (The suggested value r' is due to P. S. L. M. Barreto).

When this approach is applied to smaller values of r, the effect is even greater. For example, compare $r_1 = 22,531 = 2^{14} + 3$ (suggested in [34] for $n_0 = 3$) to

$r_1^{'} = 22,511 = 2^{14} - 17$, which also satisfies the requirements. Here, the potential speedup in the computations is $(22,531 + 2,045)/(22,511 + 17) = 1.09x$.

We point out that the performance gains from using r' or r'' instead of using r, for $QC - MDPC$, should be weighed against the resulting DFR with these values. This DFR is reported in Section 4 (Table 3).

Reduced Weight optimization. We explore the following optimization, called hereafter "Reduced Weight", as an interesting tradeoff between performance and DFR: replace Step 4 of Alg. 7 from `for` $j = 0, \ldots, w$ `do` to `for` $j = 0, \ldots, w - \alpha$ `do`, for some nonnegative integer α. To not skip the same unsatisifed parity checks in each invocation of the algorithm, we rotate $h_{0\downarrow}$ (by a random index) before using it. This reduces the number of iterations by α, and speeds up the CountUPCalgorithm by a factor of $w/(w - \alpha)$. For example, with $w = 137$ and $\alpha = 5$, the speedup is $137/132 = 1.037x$. This comes at the expense of a higher DFR because some parity checks are ignored. This is different from the optimization of [34], where all the unsatisfied parity checks are taken into account, and the threshold is $\max_i upc_i - \delta$. We note that the two optimizations can be applied simultaneously.

3.7 Constant time decoding

Constant time BitFlip. The BitFlip algorithm 6 handles the secret values H (input) and e, s (intermediate results). As a result, memory accesses, conditional execution, and latencies of operations can inadvertently leak information on the weights of e, s. and on the positions of the set bits in H, e, s (e. g., through the number of errors fixed in a specific iteration and the number of iterations). A secure ("constant time") implementation needs to prevent such potential leaks. Note that the number of iterations (either fixed of or variable) is implementation-dependent (e. g., [11]), and we keep it out of our scope.

CountUPC in constant time. The inputs of the CountUPC algorithm (Alg. 7) are H and s. A naïve constant time implementation holds H in memory (entirely), and accesses it accordingly. Here, the required memory can be large: with $n_0 = 2$ and $2^{15} < r < 2^{16}$ H occupies $n_0 r^2/2^3 > (2^{28}) \approx 268$ million bytes. The memory requirement can be reduced by storing only $h_{0\downarrow}$ in memory, at the expense of some performance penalty ($h_{0\downarrow}$ needs to be rotated during each of the r iterations). A different memory-performance trade off is obtained by duplicating $h_{0\downarrow}$ in memory, which speeds up the rotation (similarly to the way s is treated, above). We pursue a faster solution for large values of r. Alg. 8 introduces a new optimization for a constant time implementation of Alg. 7. A parameter $w' < r - w$ is determined (an appropriate choice is proposed below). Then, we choose, uniformly at random, w' positions in $h_{0\downarrow}$, where the bits are not set, call them "fake" bits. We set the bits of $h_{0\downarrow}$ in these positions, which defines a new vector $h'_{0\downarrow}$ with weight $w + w'$. Let wlist' be the compact representation of $h'_{0\downarrow}$, and let b be the "indicator array" of $w + w'$ bits, where $b[i] = h_{0\downarrow}[\mathsf{wlist}'[i]]$, $i = 0, \ldots, w + w' - 1$ (i. e., marking the non-fake positions).

Unlike the list wlist, which is secret, we can choose w' in a way that the list wlist$'$ is not secret (provided that b remains confidential). This can be achieved if $\binom{w'+w}{w} > 2^{2\lambda}$. Such a selection is shown in Table 2, for $\lambda = 128$. With this approach, a secure implementation needs to protect only operations that involve b, which is hopefully efficient because $|b| << r$. Indeed, this allows us to keep only the compact representation of $h'_{0\downarrow}$ in memory, and perform (in Alg. 8) only $|b|$ iterations. This costs $|b| \times r$ memory accesses (to bytes), instead of $r \times r$ as with the alternative.

λ	w	w'
128	137	124
128	155	111
128	161	108
96	99	98
64	67	65

Table 2: Examples for w' such that $\binom{w'+w}{w} > 2^{2\lambda}$.

Remark 2. Our method needs to make a uniform random selection from the set of size $\binom{w'+w}{w}$, of all $w + w'$ indices (positions between 0 and $r - 1$) from which which w' positions are labeled as "fake". In practice, we use Alg. 3 to choose (in constant time) the positions of the $w + w'$ bits. Subsequently, we use Alg. 3 to determine the w' positions (in b) that are labeled as "fake".

Algorithm 8 U=CountUPCConstantTime($\tilde{\mathsf{s}}$, wlist, b)

Input: $\tilde{\mathsf{s}}$ - a redundant representation of r bits syndrome s, wlist$'$ - a w' elements of the compact representation of $h'_{0\downarrow}$. b a flags list of length $w + w'$.
Output: U an array of r bytes.

```
1: procedure CountUPCConstantTime(s̃, wlist', b)
2:     U=0
3:     for i = 0, ..., r − 1 do
4:         for j = 0, ..., w' − 1 do
5:             U[i] = U[i] + (rotl(s̃,wlist'[j])[i] & b[wlist'[j]])
6:     return U
```

Step III. (Recalculate the syndrome) in constant time. The straightforward implementation recalculates the syndrome (in a $(n_0 r, r)$-code) by means of n_0 polynomial multiplications (modulo $x^r + 1$) plus $n_0 - 1$ additions in $\mathbb{F}_2^n$. Two alternative implementations (due to [30]) were discussed above. We found that in most cases, the straightforward implementation is faster. Indeed, implementation must either add all the columns ($h_{j\downarrow}$) of H that correspond to error bits, or

add all the columns while masking out the "unnecessary" ones, to execute in constant time. We point out that applying the fake bits technique to e, introduces additional overheads that, by our experiments, make it non competitive.

4 Results

This section provides the performance results of our study. For this study, we wrote new optimized code for all the algorithms discussed above.

The core functionality was written in x86 assembly, and wrapped by assisting C code. The implementations use the PCLMULQDQ, AES−NI and the AVX2 and AVX512 architecture extensions. The code was compiled with gcc (version 5.4.0) in 64-bit mode, using the "O3" Optimization level, and run on a Linux (Ubuntu 16.04.3 LTS) OS.

The experiments were carried out on a platform equipped with the latest 8^{th} Generation Intel® CoreY^{TM} processor ("Kaby Lake") - Intel® Xeon® Platinum 8124M CPU at 3.00 GHz Core® i5 − 750. The platform has 70 GB RAM, 32K L1d and L1i cache, 1,024K L2 cache, and 25,344K L3 cache. It was configured to disable the Intel® Turbo Boost Technology, and the Enhanced Intel Speedstep® Technology.

The performance is reported in processor cycles counts or in cycles per byte (C/B), where lower is better, reflecting the performance per a single core. The results were obtained with the same measurement methodology, as follows. Each measured function was isolated, run 25 times (warm-up), followed by 100 iterations that were clocked (using the RDTSC instruction) and averaged. To minimize the effect of background tasks running on the system, each such experiment was repeated 10 times, and the minimum result was recorded.

Estimating the DFR To estimate the DFR, we generated (ephemeral) CAKE keys, encrypted a pseudorandom message, and rand the decoder code. This was done for the three values of interest $r = 2^{15} - 49, 2^{15} - 19, 2^{15} + 3$. For each r, the experiments were repeated N times, and recorded the number (n_{fail}) of decoding failures. Aiming for 95% confidence interval, our estimated DFR upper bound is $3/N$ when $n_{fail}/N = 0$, and $\chi^2_{2n_{fail},0.025}/(2N)$ when $0 < n_{fail}/N < 20$. Detailed explanations are given in Appendix D.

Obviously, the DFR depends on the actual decoding algorithm. We used here the decoding algorithm of [34], and also a version optimized by incorporating the Reduced Weight optimization. The results are summarized in Table 3. The DFR upper bounds are roughly the same (around 10^{-7}), with both decoders, for the close r values. This implies that, given the specific decoder(s), it is reasonable to prefer the value of r which also leads to the best decoding performance. This aspect is discussed below.

Efficient hashing Fig. 3 compares OpenSSL's [2] performance for serial SHA384 (and SHA256), to our $\mathsf{ParallelizedHash}^{\mathsf{SHA384}}_{111,8}$ (and $\mathsf{ParallelizedHash}^{\mathsf{SHA256}}_{55,16}$), for different message lengths. It includes results from both AVX2 and AVX512 versions for the algorithms. The choice $\mathsf{s} = 8$ yields the best $\mathsf{ParallelizedHash}^{\mathsf{SHA384}}_{111,8}$ performance on AVX512 enabled platforms: since SHA384 operates on 64-bit operands,

Decoder	r	n_{fail} ($N = 10^8$)	DFR bound
[34]	32,719	0	3×10^{-8}
	32,749	1	3.369×10^{-8}
	32,771	2	5.57×10^{-8}
[34] with Reduced Weight optimization	32,719	7	1.3×10^{-7}
	32,749	4	8.76×10^{-8}
	32,771	2	5.57×10^{-8}

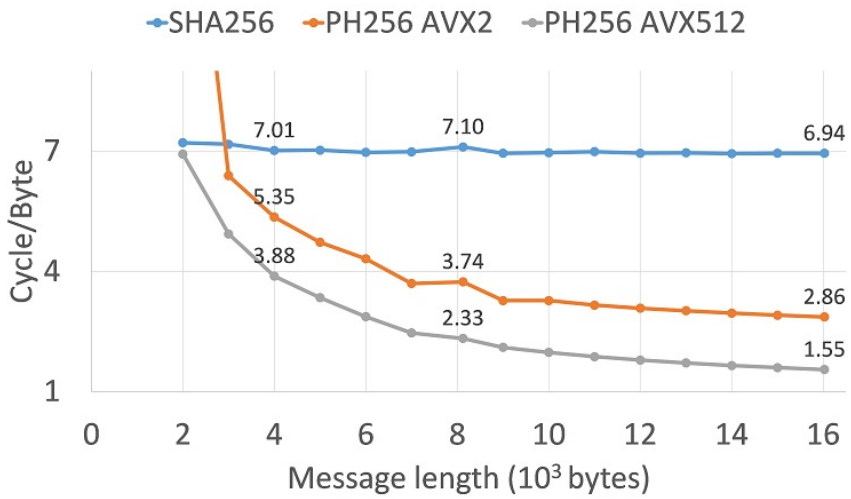
Table 3: DFR estimations (95% confidence interval) for $r = 2^{15} - 49, 2^{15} - 19, 2^{15} + 3$. The first three rows result from implementing the decoder of [34]. The last three rows result from the same decoder, combined with the Reduced Weight optimization.

and a *zmm* register can accommodate 8 lanes, the full capabilities of AVX512 can be realized with $\mathsf{s} = 8$. Similarly, $\mathsf{s} = 16$ yields the best AVX2 performance for $\mathsf{ParallelizedHash}^{\mathsf{SHA256}}_{55,16}$. The graphs show that parallelizing SHA384 (SHA256) contributes significant speedups for sufficiently long messages, e. g., $3.5x$ for a 8KB message (with SHA384). In our context, with $r \approx 2^{15}$, the QC-MDPC protocols indeed hash messages of such lengths (or more), and the savings can be of $\sim 20,000$ cycles.

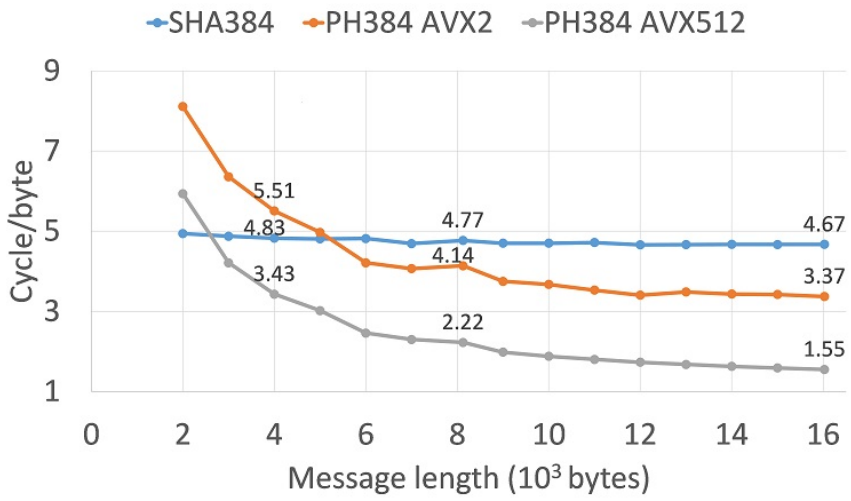
AES-CTR-PRF Alg. 1 uses AES256. An optimized AES256code, that pipelines AES−NI instructions efficiently, can produce output at 0.91 C/B. Thus, the task of generating roughly 4KB of pseudorandom data (for the relevant r values in our context) consumes approximately $\approx 3,700$ cycles. This is much faster than invoking the RDRAND instruction, as proposed in [30], as this instruction has guarantee throughput of 200 cycles [27]. It is also faster than hashing, repeatedly, a 256-bit seed concatenated with a (short) counter (even if this computation is parallelized).

Polynomial multiplication Fig. 4 shows the performance results of our optimized polynomial multiplication (modulo $x^r + 1$), compared to OpenSSL [2] (latest development version), and to NTL [38] (latest version) compiled with the GF2X library [37]. Panel (a) shows all the tested implementations and Panel (b) zooms into the two fastest ones.

NTL and our implementation are $\sim$ 7.5x faster than OpenSSL, because they use Karatsuba multiplication with special tuning for small multiplicand (see Section 3.5). Note that for $r = 2^{14}$ and $r = 2^{15}$ (marked with dashed red lines in Fig. 4) our implementation is, respectively, 1.34x and 1.2x faster than NTL. These are the cases where vertical schoolbook is advantageous. As expected, the performance for r that is slightly below 2^α is better than for r that is slightly above 2^α. Indeed, this can be seen in the figure, where the performance with r=32,719 is $\sim$ 1.2x faster than with r=32,771. We also found performance differences in the reduction (modulo $x^1 + 1$) step. For example, with $r = 32,771$, NTL's reduction consumes $\sim 3,000$, whereas our implementation consumes only 400. We



(a) SHA256



(b) SHA384

Fig. 3: The performance (in C/B) of OpenSSL's [2] serial hashing compared to ParallelizedHash AVX512 and AVX2 implementations. Top panel: serial SHA256 vs. $\mathsf{ParallelizedHash}^{\mathsf{SHA256}}_{55,16}$. Bottom panel: serial SHA384 vs. $\mathsf{ParallelizedHash}^{\mathsf{SHA384}}_{111,8}$.

suspect that the reason is that NTL has special optimization for Trinomials and Pentanomials reductions, but not for polynomials of the form $x^r + 1$.

Finally, we comment the following. We tested our optimized multiplications for $r = 2^{\alpha}(2p+1) \times 64$ and $2p+1 \leq 11$. These use recursive Karatsuba up to the point where the degree of the polynomials is $(2p+1) \times 64 - 1$, and then switch to the schoolbook multiplication (see Section 3.5). We tried both horizontal and vertical methods. The horizontal schoolbook is faster in all cases, except for the case $p = 0$. Here, we found that the vertical schoolbook is faster for polynomials of degree $4 \times 64 - 1$.

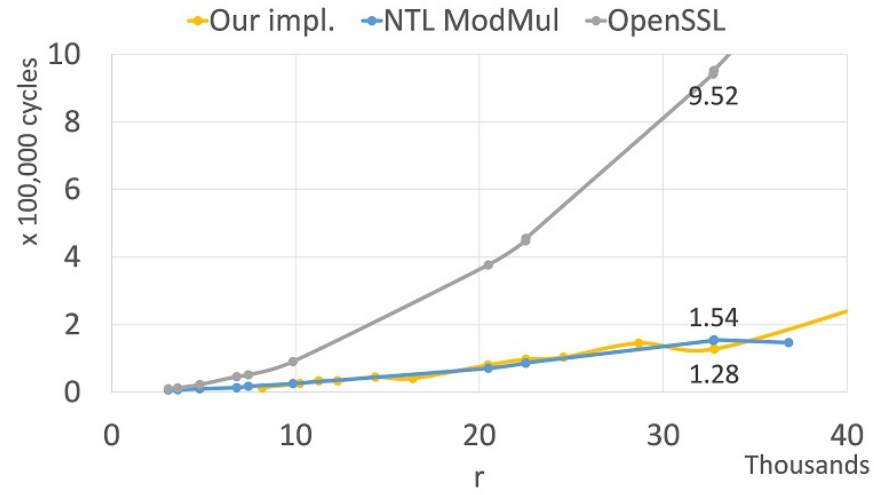
Decoding The two time consuming steps of the BitFlip algorithm are "Calculate the unsatisfied bits" and "Recalculate the syndrome". Table 4 shows the performance of our AVX2 and AVX512 implementations of the first one. For each of these two architectures, we wrote an optimized code (CountUPC) and an optimized constant time implementation (CountUPCConstantTime). Two conclusion can be deduced: 1) AVX512 implementations are consistently ~ 1.5x faster than the AVX2 implementations; 2) The added overheads of incorporating side channel protection are $100 - 120\%$ (AVX2) and $76 - 98\%$ (AVX512), and increase with r.

Table 4 also shows that close values of r may lead to different performance. Consider the case where $r_1 = 2^{11} \times p_1$, and $r_2 = 2^{11} \times p_2$, where $p_1 < 2^{11} < p_2 < 2^{12}$. The AVX512 implementation with r_1 performs better than with r_2. A similar phenomenon occurs with the AVX2 implementations, for $r_1 = 2^9 \times p_1$, and $r_2 = 2^9 \times p_2$, and $p_1 < 2^9 < p_2 < 2^{10}$. For example, the AVX512 implementation with $r_1 = 22,511$, is 1.09x faster than with $r_2 = 22,531$ ($q = 43$, $p_1 = 2^{11} - 17$ and $p_2 = 2^{11} + 3$).

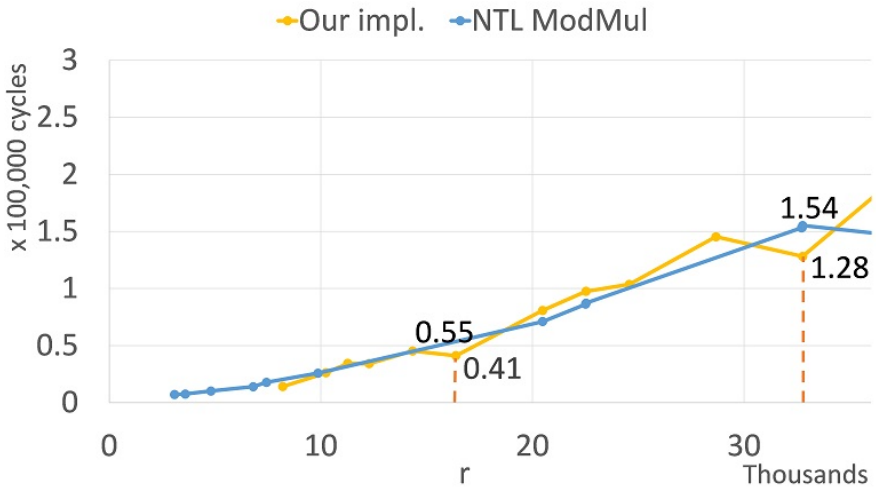
Section 3.6 proposed the Reduced Weight heuristic optimization. The results of this optimization, for an AVX512 implementation, are reported in Table 5, showing both CountUPC and CountUPCConstantTime with Reduced Weight applied. We choose $\alpha = 5$ as a good balance point between performance and DFR.

r	CountUPC		CountUPCConstantTime	
	AVX2 (10^5 cycles)	AVX512 (10^5 cycles)	AVX2 (10^5 cycles)	AVX512 (10^5 cycles)
20,483	2.04	1.38	3.53	2.42
22,511	2.12	1.33	3.76	2.4
22,531	2.18	1.45	3.86	2.61
32,719	2.98	1.91	5.88	3.75
32,749	2.98	1.91	5.88	3.75
32,771	3.04	2.03	5.91	3.91
36,821	3.48	2.32	6.92	4.6

Table 4: Cycle count comparison of AVX2/AVX512 implementations for the CountUPC and CountUPCConstantTime algorithms, lower is better.



(a) All methods



(b) Optimized methods

Fig. 4: Polynomial multiplication reduced modulo $x^r + 1$. Panel (a) shows the cycles count to our optimized implementation. OpenSSL [2] and NTL (compiled with gf2x) [37, 38]. Panel (b) shows only the two fastest implementations. Note the values $r = 2^{14}$ and $r = 2^{15}$ (marked with red dashed vertical lines), where our implementations are, respectively 1.34x and 1.2x faster. See explanation in the text.

r	Reduced Weight CountUPC		Reduced Weight CountUPCConstantTime	
	AVX512 (10^5 cycles)	SpeedUp vs. Table 4	AVX512 (10^5 cycles)	SpeedUp vs. Table 4
22,511	1.16	1.15x	2.37	1.02x
22,531	1.24	1.17x	2.57	1.01x
32,719	1.81	1.06x	3.6	1.04x
32,749	1.81	1.06x	3.63	1.03x
32,771	1.98	1.03x	3.85	1.01x

Table 5: Cycles comparison of CountUPC and CountUPCConstantTime implemented with the Reduced Weight optimization measured for different r values. We compare it to the "normal" implementation in AVX512 (reported in Table 4) for calculating the speedup.

A somewhat surprising result was obtained for step III (Recalculate the syndrome). The AVX2 optimized non-constant time implementation turned out to be slightly faster than the optimized AVX512 implementation. This happens because this step involved many memory accesses. To estimate the latency of step III, note that the performance of both implementations is proportional to the number of the corrected error bits. For example, with r=32,719, correcting 100 error bits consumes $\sim 270 \times 100 = 27,000$ cycles, which is almost 10x less than the latency of the matching constant time implementation. For constant time implementation of step III, our experiments showed that the fastest method is simply a direct calculating. For example, in CAKE (with $r = 32,719$), step III requires two polynomial multiplications and two additions. These consume roughly $2 \times 120,000 = 240,000$ cycles for the multiplication plus some (small) overhead for the two additions.

5 Conclusion

This paper offers a toolbox of primitives that can be leveraged towards efficient and constant time implementations of QC-MDPC cryptosystems. The comparison of our implementations to alternative open source libraries (NTL and OpenSSL), indicates the functionalities that are improved. For example, our polynomial multiplication for degrees $2^{14} - 1$ and $2^{15} - 1$ is 1.34x and 1.2x faster than that of NTL, respectively, and our $\mathsf{ParallelizedHash}_{8,111}^{\mathsf{SHA384}}$ is almost 3x faster than that of SHA384 in OpenSSL for large enough sizes.

Our implementations try to leverage the potential of vectorized processor architectures. For example, the AVX512 implementation of Alg. 4 uses the `VPCMPW` instruction. It operates on two zmmthat are populated with 32 words (of 16-bits), and an "immediate" that determines the comparison operator (e. g., $=, \leq, \geq$). The output is a vector of 32 words set to 0 or 1 accordingly. This is similarly done for the AVX2 implementation, using the instruction `VPCMPEQW`.

We point out that the implementation described in [30] can also enjoy AVX512 capabilities, by using the instruction `VPOPCNTQ` for counting the upc_i (instead of using `POPCNT`).

As we notice, a proper choice of parameters can contribute to the resulting performance. Here are a few examples. With $r = 32,719 = 2^{15} - 49$ optimal memory alignment led to a fast polynomial multiplication and decoding.

To achieve 128-bit quantum pre-image resistance, it is possible to use one of SHA512, SHA384, SHA256, or their parallelized variant, and the optimal choice depends on the length of the hashed buffer. Computing $\mathsf{ParallelizedHash}^{384}_{111,8}$ is faster than computing $\mathsf{ParallelizedHash}^{512}_{111,8}$ (due to the length of the final hash). Furthermore, a sparse buffer (e. g., in the last step of CAKE) can be compressed prior to hashing, in order to optimize the performance. Of course, in such cases, the cost of compression (which can be significant, especially if it needs to be carried out in constant time) needs to be weighed against the potential reduction in the hashing time.

The paper discusses optimization techniques for different building blocks of generic decoding algorithms. Our results can be used for enhancing the performance of a class of decoding algorithms. Given an algorithm, it is possible to predict the performance results by analyzing the cost of one decoding iteration, and multiplying it by the maximum number of iterations, that the algorithm defines. We provide one example.

Example 10. Consider the case $r = 32,719 = 2^{15} - 49$ and $w = 137$. A decoding iteration consists of the above three steps, and the performance cost of each iteration is the corresponding sum. We provide the performance of each step, for a constant time implementation (the numbers in parentheses correspond to non-constant time implementations), based on our results: "CountUPC" performs at $375,000$ $(191,000)$ cycles, "Find and compare the threshold" at $375,000$ $(20,000)$ cycles, and "Recalculate the syndrome" at $250,000$ $(\sim 27,000)$ cycles. The sum is $1,000,000$ $(238,000)$ cycles per iteration.

In this example, we use the decoder of [34] and a constant time implementation. Note that the performance of such implementation depends on the *maximal* number of iterations, whereas a non-constant time implementation depends on the *average* number of iterations. To estimate the maximum / average number of iterations (with $r = 32719$), we set $\delta = 6$ and ran it on 2×10^6 encoded random inputs. The observed maximum and average number of iteration was 20 and 8, respectively. We can therefore estimate that constant-time decoding with this algorithm, choice of parameters, and 20 iterations which leads to $DFR \sim 1.5 \times 10^{-6}$ would perform at $\sim 20,000,000$ cycles ($8 \times 238,000 = 1,904,000$ cycles with a non-constant time implementation), on the "Kaby Lake" platform. Our actual measurements agree with this estimation.

For a rough comparison, consider the optimized constant time decoding performance of $193,922,410$ cycles, reported in [30], for 128-bit quantum security,

when run on an AVX2 platform [3]. Our $\sim$ 9x speedup is due to the optimized primitives that leverage the capabilities of the AVX512 features.

Our choice of 20 iterations is a property of the algorithm in [34], and our the experiments. However, other decoding algorithms can optimize for a reduced maximum number of iteration, and achieve better constant time performance. One example is the result in [11] that reports an optimization of $(9,602,4,801)$-codes with $w = 90$, where DFR of 0.150×10^{-6} is achieved with maximum number of iterations set to 7. A different trade off between performance and DFR is proposed by our Reduced Weightoptimization (see Table 3)).

Acknowledgments

This research was supported by the PQCRYPTO project, which was partially funded by the European Commission Horizon 2020 research Programme, grant #645622, by the Israel Science Foundation (grant No. 1018/16), by the BIU Center for Research in Applied Cryptography and Cyber Security, in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office, and by the Center for Cyber Law and Policy at the University of Haifa.

Opinions, findings, conclusions, and recommendations, expressed in this material, are those of the author(s), and do not necessarily reflect the views of their employers and the granting agencies.

References

1. −: Nist:post-quantum cryptography - call for proposals (September 2017), https://csrc.nist.gov/Projects/Post-Quantum-Cryptography
2. −: OpenSSL, Commit: 2dbfa8444bdf7669a54006c4a83d1e60ba374528. https://github.com/openssl/openssl (Sep 2017)
3. Aguilar, C., Blazy, O., Deneuville, J.C., Gaborit, P., Zémor, G.: Efficient encryption from random quasi-cyclic codes. arXiv preprint arXiv:1612.05572 (2016)
4. Baldi, M., Chiaraluce, F., Garello, R., Mininni, F.: Quasi-cyclic low-density parity-check codes in the McEliece cryptosystem. In: 2007 IEEE International Conference on Communications. pp. 951–956 (June 2007)
5. Baldi, M., Bianchi, M., Chiaraluce, F., Rosenthal, J., Schipani, D.: Using LDGM Codes and Sparse Syndromes to Achieve Digital Signatures, pp. 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg (2013), https://doi.org/10.1007/978-3-642-38616-9_1
6. Baldi, M., Bodrato, M., Chiaraluce, F.: A new analysis of the McEliece cryptosystem based on QC-LDPC codes. Security and Cryptography for Networks pp. 246–262 (2008)
7. Baldi, M., Chiaraluce, F., Garello, R.: On the usage of quasi-cyclic low-density parity-check codes in the McEliece cryptosystem. In: 2006 First International Conference on Communications and Electronics. pp. 305–310 (Oct 2006)

[3] Intel® Core 4770M CPU at 3.40 GHz Core® i7 − 770.

8. Barker, E.B., Kelsey, J.M.: SP 800-90A. recommendation for random number generation using deterministic random bit generators. Tech. rep., Gaithersburg, MD, United States (2012)
9. Barreto, P.S.L.M., Gueron, S., Gueneysu, T., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.P.: CAKE: Code-based Algorithm for Key Encapsulation. Cryptology ePrint Archive, Report 2017/757 (2017), http://eprint.iacr.org/2017/757
10. Cayrel, P.L., Hoffmann, G., Persichetti, E.: Efficient Implementation of a CCA2-Secure Variant of McEliece Using Generalized Srivastava Codes, pp. 138–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), https://doi.org/10.1007/978-3-642-30057-8_9
11. Chaulet, J., Sendrier, N.: Worst case QC-MDPC decoder for McEliece cryptosystem. In: 2016 IEEE International Symposium on Information Theory (ISIT). pp. 1366–1370 (July 2016)
12. Courtois, N.T., Finiasz, M., Sendrier, N.: How to achieve a mceliece-based digital signature scheme. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 157–174. Springer (2001)
13. Deneuville, J.C., Gaborit, P., Zémor, G.: Ouroboros: A Simple, Secure and Efficient Key Exchange Protocol Based on Coding Theory, pp. 18–34. Springer International Publishing, Cham (2017), https://doi.org/10.1007/978-3-319-59879-6_2
14. Faugère, J.C., Otmani, A., Perret, L., de Portzamparc, F., Tillich, J.P.: Structural cryptanalysis of McEliece schemes with compact keys. Designs, Codes and Cryptography 79(1), 87–112 (Apr 2016), https://doi.org/10.1007/s10623-015-0036-z
15. Gallager, R.: Low-density parity-check codes. IRE Transactions on Information Theory 8(1), 21–28 (January 1962)
16. Gueron, S.: Intel's new AES instructions for enhanced performance and security. In: FSE. vol. 5665, pp. 51–66. Springer (2009)
17. Gueron, S.: Intel® advanced encryption standard (AES) new instructions set Rev. 3.01. Intel Software Network (2010)
18. Gueron, S.: A j-lanes tree hashing mode and j-lanes SHA-256. Journal of Information Security 4(01), 7 (2013)
19. Gueron, S.: Parallelized hashing via j-lanes and j-pointers tree modes, with applications to SHA-256. Journal of Information Security 5(03), 91 (2014)
20. Gueron, S., Kounavis, M.: Efficient implementation of the galois counter mode using a carry-less multiplier and a fast reduction algorithm. Information Processing Letters 110(14), 549 – 553 (2010), http://www.sciencedirect.com/science/article/pii/S002001901000092X
21. Gueron, S., Kounavis, M.E.: Intel® carry-less multiplication instruction and its usage for computing the gcm mode. White Paper (2010)
22. Gueron, S., Krasnov, V.: Simultaneous hashing of multiple messages. Journal of Information Security 3(04), 319 (2012)
23. Gueron, S., Schlieker, F.: Speeding up R-LWE Post-quantum Key Exchange, pp. 187–198. Springer International Publishing, Cham (2016), https://doi.org/10.1007/978-3-319-47560-8_12
24. Guo, Q., Johansson, T., Stankovski, P.: A Key Recovery Attack on MDPC with CCA Security Using Decoding Errors, pp. 789–815. Springer Berlin Heidelberg, Berlin, Heidelberg (2016), https://doi.org/10.1007/978-3-662-53887-6_29
25. Huffman, W.C., Pless, V.: Fundamentals of error-correcting codes. Cambridge university press (2010)

26. Intel Corporation: Intel architecture instruction set extensions programming reference. https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf (October 2017)
27. Intel Corporation: Intel intrinsic guide. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=rdrand&expand=4247 (October 2017)
28. Jovanovic, B.D., Levy, P.S.: A look at the rule of three. The American Statistician 51(2), 137–139 (1997), http://www.jstor.org/stable/2685405
29. Kabatianskii, G., Krouk, E., Smeets, B.: A digital signature scheme based on random error-correcting codes, pp. 161–167. Springer Berlin Heidelberg, Berlin, Heidelberg (1997), https://doi.org/10.1007/BFb0024461
30. Maurich, I.V., Oder, T., Güneysu, T.: Implementing QC-MDPC McEliece encryption. ACM Trans. Embed. Comput. Syst. 14(3), 44:1–44:27 (Apr 2015), http://doi.acm.org/10.1145/2700102
31. McEliece, R.: A public-key cryptosystem based on algebraic. Coding Thv 4244, 114–116 (1978)
32. Misoczki, R., Barreto, P.S.L.M.: Compact McEliece Keys from Goppa Codes, pp. 376–392. Springer Berlin Heidelberg, Berlin, Heidelberg (2009), https://doi.org/10.1007/978-3-642-05445-7_24
33. Misoczki, R., Tillich, J.P., Sendrier, N., Barreto, P.S.L.M.: MDPC-McEliece: New McEliece variants from moderate density parity-check codes. Cryptology ePrint Archive, Report 2012/409 (2012), http://eprint.iacr.org/2012/409
34. Misoczki, R., Tillich, J.P., Sendrier, N., Barreto, P.S.: MDPC-McEliece: New McEliece variants from moderate density parity-check codes. In: 2013 IEEE International Symposium on Information Theory. pp. 2069–2073 (July 2013)
35. Monico, C., Rosenthal, J., Shokrollahi, A.: Using low density parity check codes in the McEliece cryptosystem. In: 2000 IEEE International Symposium on Information Theory (Cat. No.00CH37060). pp. 215–. IEEE (2000)
36. Phesso, A., Tillich, J.P.: An Efficient Attack on a Code-Based Signature Scheme, pp. 86–103. Springer International Publishing, Cham (2016), https://doi.org/10.1007/978-3-319-29360-8_7
37. Pierrick Gaudry, Richard Brent, P.Z., Thome, E.: gf2x-1.2. https://gforge.inria.fr/projects/gf2x/ (July 2017)
38. Shoup, V.: Number theory c++ library (ntl) version 10.5.0. http://www.shoup.net/ntl (July 2017)
39. Stern, J.: A new identification scheme based on syndrome decoding. In: Annual International Cryptology Conference. pp. 13–21. Springer (1993)

A AES-CTR-PRF

The statefull algorithm has a state s with the following fields: buffer(16 bytes), pos(1 bytes), $\mathsf{remInvokations}$(4 bytes), seed(32 bytes), j such that $0 \leq \mathsf{j} \leq 2^{128}-1$. The cipher is invokes over plaintext blocks of the form $\mathsf{ctr} = \mathsf{encode128}(\mathsf{s.j})$, for the j^{th} invocation. Initialization is done by calling AES-CTR-PRF-Init (Alg. 9), with input seed and a maximal number of $\mathsf{AES256}$ invocations ($\mathsf{maxInvokation}$) before reseeding. AES-CTR-PRF-Init initializes $\mathsf{s.pos}$ to point to the end of $\mathsf{s.buffer}$, and $\mathsf{s.j}$to 0. An exception flag $\mathsf{SeedOverUseError}$ is raised when the algorithm reaches $\mathsf{maxInvokation}$.

Algorithm 9 s=AES-CTR-PRF-Init(seed, maxInvokation)

Input: seed (32 bytes), maxInvokation (4 bytes)
Output: s (AES-CTR-PRF state)
procedure AES-CTR-PRF-INIT(seed, maxInvokation)
    s.seed = seed
    s.pos = 16
    s.buffer = $NULL$
    s.j= encode128(0)
    s.remInvokations = maxInvokation
    return s

Algorithm 10 A=AES-CTR-PRF(s, len)

Input: s (AES-CTR-PRFstate), len
Output: A (len bytes), the updated AES-CTR-PRF*state* s
Exception: SeedOverUseError (seed overused).
procedure TMP=PERFORMAES(s)
    if s.remInvokations = 0 **then**
        raise SeedOverUseError **exception**
    tmp[15 : 0] = $\mathsf{AES256}_{\mathsf{s.seed}}$(encode128(s.j))
    s.j = s.j + 1
    s.remInvokations = s.remInvokations − 1
    return tmp

procedure AES-CTR-PRF(s, NB)
    if len + s.pos <= 16 **then** ▷ Buffer has enough data.
        A[len − 1 : 0] = s.buffer[s.pos + len − 1 : s.pos]
        s.pos = s.pos + len
    else
        idx = 16 − s.pos ▷ calculate the buffer content length.
        if idx > 0 **then**
            A[idx − 1 : 0] = s.buffer[15 : s.pos]
        s.pos = 0
        while len − *idx* >= 16 **do** ▷ Copy full AES256 blocks
            A[idx + 15 : idx] = PerformAES(s)
            idx = idx + 16
        s.buffer = PerformAES(s) ▷ Handle the tail.
        s.pos = len − idx
        A[len − 1 : idx] = s.buffer[s.pos − 1 : 0]
    return A[len − 1 : 0], s

B GenPseudoRand example

Running Alg. 1 to populate a string A of len = $17 \times 8 + 5 = 141$ bits, embedded in $\overline{\mathsf{A}}[18 : 0]$, with pseudorandom values that stem from using an initialized AES-CTR-PRFwith the input seed seed = encode128(0)||encode128(0).

```
Inputs:
len       = 141
s.seed    = 0000000000000000000000000000000000
            0000000000000000000000000000000000
s.j       = 1
s.pos     = 0
CTR0      = encode128(0)
          = 00000000000000000000000000000000
s.buffer  = AES256(s.seed, CTR0)
          = 8720849214a248ad898940a278c095dc

AES-CTR_PRF internal values:
Ā[15:0]   = 8720849214a248ad898940a278c095dc
CTR1      = encode128(1)
          = 00000000000000000000000000000001
s.buffer  = AES256(s.seed, CTR1)
          = d33ca5bf3e13934568b84f6bd8f37552
s.j       = 2
s.pos     = 2
Ā[17:16]  = 7552

Outputs:
Ā[17:0]   = 15528720849214a248ad898940a278c095dc
```

C ParallelizedHash$^{\mathsf{SHA384}}_{8,111}$ example

ParallelizedHash$^{\mathsf{SHA384}}_{8,111}$ of the array of $\mathsf{la} = 2{,}000$ byte $\mathsf{array}[j] = j \pmod{255}$, $j = 0, \ldots, \mathsf{la} - 1$.

```
la    = 2,000
ls    = 239
lrem  = 88

m     = d6d5d4d3d2d1d0cfcecdcccbcac9c8c7c6c5c4c3c2c1c0bf
        bebdbcbbbab9b8b7b6b5b4b3b2b1b0afaeadacabaaa9a8a7
        a6a5a4a3a2a1a09f9e9d9c9b9a9998979695949392919 08f
        8e8d8c8b8a898887868584838281807f7e7d7c7b7a797877
        767574737271706f6e6d6c6b6a6968676665646362616 05f
        5e5d5c5b5a595857565554535251504f4e4d4c4b4a494847
        464544434241403f3e3d3c3b3a393837363534333231302f
        2e2d2c2b2a292827262524232221201f1e1d1c1b1a191817
        161514131211100f0e0d0c0b0a090807060504030201 00fe
        fdfcfbfaf9f8f7f6f5f4f3f2f1f0efeeedecebeae9e8e7e6
        e5e4e3e2e1e0dfdedddcdbdad9d8d7d6d5d4d3d2d1d0cfce
```

```
          cdcccbcac9c8c7c6c5c4c3c2c1c0bfbebdbcbbbab9b8b7b6
          b5b4b3b2b1b0afaeadacabaaa9a8a7a6a5a4a3a2a1a09f9e
          9d9c9b9a999897969594939291908f8e8d8c8b8a89888786
          8584838281807f7e7d7c7b7a79787776757473727170 6f6e
          6d6c6b6a696867666564636261605f5e5d5c5b5a59585756
          5554535251504f4e4d4c4b4a4948474645444342414 03f3e
          3d3c3b3a393837363534333231302f2e2d2c2b2a29282726
          2524232221201f1e1d1c1b1a1918171615141312111 00f0e
          0d0c0b0a090807060504030201 00fefdfcfbfaf9f8f7f6f5
          ...
          5f5e5d5c5b5a595857565554535251504f4e4d4c4b4a4948
          47464544434241403f3e3d3c3b3a39383736353433323130
          2f2e2d2c2b2a292827262524232221201f1e1d1c1b1a1918
          17161514131211100f0e0d0c0b0a090807060504030201 00

X[0    = 736aed26a8c4ed0add98f587bcaf349f2b748029eeaf3715
         769f162d8343445c63ee3c4a0f606dbb498c787a07cf5625

X[1]   = a24f959fd7b64bf4428ca7947133d1ceb2278f12ab37ee6c
         29298ba72d48d33d3efd3490d84d22b227f78a1454c055a9

X[2]   = 9c6f1f05fd0069788e5e555e1dd1648f61d222728d7c7357
         3c4859b2b84b5d443737a883f9afdfbca5d9bc6bd1bd5f95

X[3]   = 6ea3e8fd041d49db9b96fa39426637d3493dc889e2d5bd86
         faff2ca73e93e57669eccfa46088561529fd3d91d709a240

X[4]   = 4e2335af0345f0f6823cd4b569dcfa4b84515919c6afc150
         844b904b96b64578ad9c375058d5c5f2d0980ccc021e00f6

X[5]   = 0a8736a0be9cc12199207c4ef2df31e12ba32e47fd2ef356
         7ca8230694b0c09c93bb5b029fe51475223f021c201f8b28

X[6]   = 192f5227698c87d8e6d2f704c501757a902629263e57ead6
         958d99aaceccd301019214d0cc6371d9036e76a8b832b5a1

X[7]   = 09d013edf0a6f784c6e3b049069788a91030a9fc39de03db
         6a748ca48f723614ef82533f3ead5b63764b18a5a29a1488

Y      = d6d5d4d3d2d1d0cfcecdcccbcac9c8c7c6c5c4c3c2c1c0bf
         bebdbcbbbab9b8b7b6b5b4b3b2b1b0afaeadacabaaa9a8a7
         a6a5a4a3a2a1a09f9e9d9c9b9a999897969594939291908f
         8e8d8c8b8a898887868584838281807f

YX     = d6d5d4d3d2d1d0cfcecdcccbcac9c8c7c6c5c4c3c2c1c0bf
```

```
        bebdbcbbbab9b8b7b6b5b4b3b2b1b0afaeadacabaaa9a8a7
        a6a5a4a3a2a1a09f9e9d9c9b9a9998979695949392919 08f
        8e8d8c8b8a898887868584838281807f09d013edf0a6f784
        c6e3b049069788a91030a9fc39de03db6a748ca48f723614
        ef82533f3ead5b63764b18a5a29a1488192f5227698c87d8
        e6d2f704c501757a902629263e57ead6958d99aaceccd301
        019214d0cc6371d9036e76a8b832b5a10a8736a0be9cc121
        99207c4ef2df31e12ba32e47fd2ef3567ca8230694b0c09c
        93bb5b029fe51475223f021c201f8b284e2335af0345f0f6
        823cd4b569dcfa4b84515919c6afc150844b904b96b64578
        ad9c375058d5c5f2d0980ccc021e00f66ea3e8fd041d49db
        9b96fa39426637d3493dc889e2d5bd86faff2ca73e93e576
        69eccfa46088561529fd3d91d709a2409c6f1f05fd006978
        8e5e555e1dd1648f61d222728d7c73573c4859b2b84b5d44
        3737a883f9afdfbca5d9bc6bd1bd5f95a24f959fd7b64bf4
        428ca7947133d1ceb2278f12ab37ee6c29298ba72d48d33d
        3efd3490d84d22b227f78a1454c055a9736aed26a8c4ed0a
        dd98f587bcaf349f2b748029eeaf3715769f162d8343445c
        63ee3c4a0f606dbb498c787a07cf5625

digest = 449d8e98c4805b8e551d7520466a3ebdebb5d3230009486a
         1687da888616305ca6fa1d9c5d890835f512e535e651cbcb
```

D Estimating the DFR

To estimate the DFR from N experiments that show n_{fail} decoding failures, with a 95% confidence interval, we use the following methodology.

If $n_{fail} = 0$, we use the "Rule of Three" [28] that places the DFR in the interval $[0, 3/N]$, which implies the upper bound DFR $\leq 3/N$. Let $\hat{p} = \frac{n_{fail}}{N}$ denote the maximum-likelihood estimator for the DFR, and let $X \sim Bin(N,\text{DFR})$ denote the distribution of the failures. This is well approximated by the Poisson distribution $X \sim Poiss(N{\times}\text{DFR})$, for sufficiently large N. If $\hat{p} < 20$, we use the χ^2 distribution as an approximation of the related Poisson distribution $X \sim Poiss(N{\times}\text{DFR})$, getting the confidence interval $\frac{1}{2N} \times [\chi^2_{2(n_{fail}+1),1-\alpha/2}, \chi^2_{2n_{fail},\alpha/2}]$. With $\alpha = 0.05$, this gives the upper bound DFR $\leq \frac{1}{2N} \times \chi^2_{2n_{fail},0.025}$. In case $\hat{p} \geq 20$, the Poisson distribution can be approximated by the Gaussian distribution, giving DFR $\leq \hat{p} + \mathbb{Z}_\alpha \times \sqrt{\hat{p}(1-\hat{p})/N}$.