QcBits: Constant-Time Small-Key Code-Based Cryptography

Tung Chou

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
blueprint@crypto.tw

Abstract. This paper introduces a constant-time implementation for a quasi-cyclic moderate-density-parity-check (QC-MDPC) code based encryption scheme. At a 2^{80} security level, the software takes 14 679 937 Cortex-M4 and 1 560 072 Haswell cycles to decrypt a short message, while the previous records were 18 416 012 and 3 104 624 (non-constant-time) cycles. Such speed is achieved by combining two techniques: 1) performing each polynomial multiplication in $\mathbb{F}_2[x]/(x^r - 1)$ and $\mathbb{Z}[x]/(x^r - 1)$ using a sequence of "constant-time rotations" and 2) bitslicing.

Keywords: McEliece · Niederreiter · QC-MDPC codes · bitslicing · software implementation

1 Introduction

In 2012, Misoczki, Tillich, Sendrier, and Barreto proposed to use QC-MDPC codes for code-based cryptography [3]. The main benefit of using QC-MDPC codes is that they allow small key sizes, as opposed to using binary Goppa codes as proposed in the original McEliece paper [1]. Since then, implementation papers for various platforms have been published; see [4, 5] (for FPGA and AVR), [7, 9] (for Cortex-M4), and [11] (for Haswell, includes results from [4, 5, 7]).

One problem of QC-MDPC codes is that the most widely used decoding algorithm, when implemented naively, leaks information about secrets through timing. Even though decoding is only used for decryption, the same problem can also arise if the key-generation and encryption are not constant-time. Unfortunately, the only software implementation paper that addresses the timing-attack issue is [7]. [7] offers constant-time encryption and decryption on a platform without caches (for writable-memory).

This work was supported by the Netherlands Organisation for Scientic Research (NWO) under grant 639.073.005 and by the Commission of the European Communities through the Horizon 2020 program under project number 645622 PQCRYPTO. Permanent ID of this document: 172b0e150c3b6be91b0bdaa0870c1e7d. Date: 2016.03.13.

Table 1. Performance results for QcBits, [9], [7], and the vectorized implementation in [11]. The "key-pair" column shows cycle counts for generating a key pair. The "encrypt" column shows cycle counts for encryption. The "decrypt" column shows cycle counts for decryption. For performance numbers of Qcbits, 59-byte plaintexts are used to follow the eBACS [16] convention. For [9] 32-byte plaintexts are used. Cycle counts labeled with * mean that the implementation for the operation is not constant-time on the platform, which means that the worst-case performance can be much worse (especially for decryption). Note that all the results are for 2^{80} security ($r = 4801, w = 90, t = 84$; see Section 2.1).

platform	key-pair	encrypt	decrypt	reference	implementation	scheme
Haswell	784 192	82 732	1 560 072	**(new)** QcBits	clmul	KEM/DEM
	20 339 160	225 948	2 425 516	**(new)** QcBits	ref	KEM/DEM
	*14 234 347	*34 123	*3 104 624	[11]		McEliece
Sandy Bridge	2 497 276	151 204	2 479 616	**(new)** QcBits	clmul	KEM/DEM
	44 180 028	307 064	3 137 088	**(new)** QcBits	ref	KEM/DEM
Cortex-A8	61 544 763	1 696 011	16 169 673	**(new)** QcBits	ref	KEM/DEM
Cortex-M4	140 372 822	2 244 489	14 679 937	**(new)** QcBits	no-cache	KEM/DEM
	*63 185 108	*2 623 432	*18 416 012	[9]		KEM/DEM
	*148 576 008	7 018 493	42 129 589	[7]		McEliece

This paper presents QcBits (pronounced "quick-bits"), a fully constant-time implementation of a QC-MDPC-code-based encryption scheme. QcBits provides constant-time key-pair generation, encryption, and decryption for a wide variety of platforms, including platforms with caches. QcBits follows the McBits paper [17] to use a variant of the hybrid (KEM/DEM) Niederreiter encryption scheme proposed in [13, 14]. (The variant does not exactly follow the KEM/DEM construction since there is an extra "KEM failed" bit passed from the KEM to the DEM; see [17]) As a property of the KEM/DEM encryption scheme, the software is protected against adaptive chosen ciphertext attacks, as opposed to the plain McEliece or Niederreiter [2] encryption scheme. The code is written in C, which makes it easy to understand and verify. Moreover, QcBits outperforms the performance results achieved by all previous implementation papers; see below.

The reader should be aware that QcBits, in the current version, uses a 2^{80}-security parameter set from [3]. Note that with some small modifications QcBits can be used for a 2^{128} security parameter. However, I have not found good "thresholds" for the decoder for 2^{128} security that achieves a low failure rate and therefore decide not to include the code for 2^{128} security in the current version. Also, the key space used is smaller than the one described in [3]. However, this is also true for all previous implementation papers [4, 5, 7, 9, 11]. These design choices are made to reach a low decoding failure rate; see Section 2.1 and 7 for more discussions.

Performance Results. The performance results of QcBits are summarized in Table 1, along with the results for [9], [7], and the vectorized implementation in [11]. In particular, the table shows performance results of the implementations contained in Qcbits for different settings. The implementation "ref" serves as the reference implementation, which can be run on all reasonable 64/32-bit platforms. The implementation "`clmul`" is a specialized implementation that relies on the `PCLMULQDQ` instruction, i.e., the $64 \times 64 \rightarrow 128$-bit carry-less multiplication instruction. The implementation "`no-cache`" is similar to `ref` except that it does not provide full protection against cache-timing attacks. Both "`ref`" and "`clmul`" are constant-time, even on platforms with caches. "`no-cache`" is constant-time only on platforms that do not have cache for *writable* memory. Regarding previous works, both the implementations in [11] for Haswell and [9] for Cortex-M4 are not constant-time. [7] seems to provide constant-time encryption and decryption, even though the paper argues about resistance against simple-power analysis instead of being constant-time.

On the Haswell microarchitecture, `QcBits` is about twice as fast as [11] for decryption and an order of magnitude faster for key-pair generation, even though the implementation of [11] is not constant-time. `QcBits` takes much more cycles on encryption. This is mainly because `QcBits` uses a slow source of randomness; see Section 3.1 for more discussions. A minor reason is that KEM/DEM encryption requires intrinsically some more operations than McEliece encryption, e.g., hashing.

For tests on Cortex-M4, STM32F407 is used for `QcBits` and [7], while [9] uses STM32F417. Note that there is no cache for writable memory (SRAM) on these devices. `QcBits` is faster than [9] for encryption and decryption. The difference is even bigger when compared to [7]. The STM32F407/417 product lines provide from 512 kilobytes to 1 megabyte of flash. [9] reports a flash usage of 16 kilobytes, while the implementation `no-cache` uses 62 kilobytes of flash when the symmetric primitives are included and 38 kilobytes without symmetric primitives. See Section 2.3 for more discussions on the symmetric primitives.

It is important to note that, since the decoding algorithm is probabilistic, each implementation of decryption comes with a failure rate. For `QcBits` no decryption failure occurred in 10^8 trials. I have not found "thresholds" for the decoding algorithm that achieves the same level of failure rate at a 2^{128} security level, which is why QcBits uses a 2^{80}-security parameter set. For [11], no decryption failure occurred in 10^7 trials. For [9] the failure rate is not indicated, but the decoder seems to be the same as [11]. It is unclear what level of failure rate [7] achieves. See Section 7 for more discussions about failure rates.

Table 2 shows performance results for 128-bit security. Using thresholds derived from the formulas in [3, Section A] leads to a failure rate of $6.9 \cdot 10^{-3}$ using 12 decoding iterations. Experiments show that there are some sets of thresholds that achieve a failure rate around 10^{-5} using 19 decoding iterations, but this is still far from 10^{-8}; see Section 6 for the thresholds. Note that [11] and [9] did not specify the failure rates they achieved for 128-bit security, and [7] does not have implementation for 128-bit security. It is reported in [3] that no decryption failure occurred in 10^7 trials for all the parameter sets presented in the paper (including the ones used for Table 1 and Table 2), but they did not provide details such as how many decoding iterations are required to achieve this.

Table 2. Performance results for `QcBits`, [9], and the vectorized implementation in [11] for 128-bit security ($r = 9857, w = 142, t = 134$; see Section 2.1). The cycle counts for `QcBits` decryption are underlined to indicate that these are cycle counts for one decoding iteration. Experiments show that `QcBits` can achieve a failure rate around 10^{-5} using 19 decoding iterations (see Section 6).

platform	key-pair	encrypt	decrypt	reference	implementation	scheme
Haswell	5 824 028	196 836	$\underline{1\,363\,948}$	**(new)** `QcBits`	`clmul`	KEM/DEM
	*54 379 733	*106 871	*18 825 103	[11]		McEliece
Cortex-M4	750 584 383	6 353 732	$\underline{7\,436\,655}$	**(new)** `QcBits`	`no-cache`	KEM/DEM
	*251 288 544	*13 725 688	*80 260 696	[9]		KEM/DEM

Comparison with Other Post-Quantum Public-Key Systems. In 2013, together with Bernstein and Schwabe, I introduced `McBits` (cf. [17]), a constant-time implementation for the KEM/DEM encryption scheme using binary Goppa code. At a 2^{128} security level, the software takes only 60493 Ivy Bridge cycles to decrypt. It might seem that `QcBits` is far slower than `McBits`. However, the reader should keep in mind that `McBits` relies on external parallelism to achieve such speed: the cycle count is the result of dividing the time for running 256 decryption instances in parallel by 256. The speed of `QcBits` relies only on internal parallelism: the timings presented in Table 1 are all results of running only one instance.

Lattice-based systems are known to be pretty efficient, and unfortunately `QcBits` is not able to compete with the best of them. For example, the eBACS website reports that `ntruees439ep1`, at a 2^{128} security level, takes 54940 Haswell cycles (non-constant-time) for encryption and 57008 for Haswell cycles (non-constant-time) for decryption. Also, the recently published "Newhope" paper [32] for post-quantum key exchange, which targets a 2^{128} quantum security level, reports 115414 Haswell cycles (constant-time) for server-side key generation, 23988 Haswell cycles (constant-time) for server-side shared-secret computation, and 144788 Haswell cycles (constant-time) for client-side key generation plus shared-secret computation.

It is worth noticing that using QC-MDPC codes instead of binary Goppa codes allows smaller key sizes. [3] reports a public-key size of 601 bytes for a 2^{80}-security parameter set (the one used for Table 1), 1233 bytes for a 2^{128}-security parameter set (the one used for Table 2), 4097 bytes for a 2^{256}-security parameter set. [17] reports 74624 bytes for a 2^{80}-security parameter set and 221646 bytes for a 2^{128}-security parameter set, and 1046739 bytes for a 2^{256}-security parameter set. The public-key size is 609 bytes for `ntruees439ep1`. The "message sizes" for Newhope are 1824 bytes (server to client) and 2048 bytes (client to server).

The usage of binary Goppa code was proposed by McEliece in [1] in 1978, along with the McEliece cryptosystem. After almost 40 years, nothing has really changed the practical security of the system. The NTRU cryptosystem is almost 20 years old now. QC-MPDC-code-based cryptosystems, however, are still quite young and thus require some time to gain confidence from the public.

2 Preliminaries

This section presents preliminaries for the following sections. Section 2.1 gives a brief review on the definition of QC-MDPC codes. Section 2.2 describes the "bit-flipping" algorithm for decoding QC-MDPC codes. Section 2.3 gives a specification of the KEM/DEM encryption scheme implemented by `QcBits`.

2.1 QC-MDPC Codes

"MDPC" stands for "moderate-density-parity-check". As the name implies, an MDPC code is a linear code with a "moderate" number of non-zero entries in a parity-check matrix H. For ease of discussion, in this paper it is assumed $H \in \mathbb{F}_2^{r \times n}$ where $n = 2r$, even though some parameter sets in [3] use $n = 3r$ or $n = 4r$. H is viewed as the concatenation of two square matrices, i.e., $H = H^{(0)}|H^{(1)}$, where $H^{(i)} \in \mathbb{F}_2^{r \times r}$.

"QC" stands for "quasi-cyclic". Being quasi-cyclic means that each $H^{(i)}$ is "cyclic". For ease of discussion, one can think this means

$$H^{(k)}_{(i+1) \bmod r, (j+1) \bmod r} = H^{(k)}_{i,j},$$

even though the original paper allows a row permutation on H. Note that being quasi-cyclic implies that H has a fixed row weight w. The following is a small parity-check matrix with $r = 5, w = 4$:

$$\begin{pmatrix} 1\,0\,1\,0\,0 & 0\,1\,0\,0\,1 \\ 0\,1\,0\,1\,0 & 1\,0\,1\,0\,0 \\ 0\,0\,1\,0\,1 & 0\,1\,0\,1\,0 \\ 1\,0\,0\,1\,0 & 0\,0\,1\,0\,1 \\ 0\,1\,0\,0\,1 & 1\,0\,0\,1\,0 \end{pmatrix}.$$

The number of errors a code is able to correct is often specified as t. Since there is no good way to figure out the minimum distance for a given QC-MDPC code, t is usually merely an estimated value.

`Qcbits` uses $r = 4801$, $w = 90$, and $t = 84$ matching a 2^{80}-security parameter set proposed in [3]. However, `Qcbits` further requires that $H^{(0)}$ and $H^{(1)}$ have the same row weight, namely $w/2$. This is not new, however, as all the previous implementation papers [4, 5, 7, 9, 11] also restrict H in this way. For `QcBits` this is a decision for achieving low failure rate; see Section 7 for more discussions on this issue. Previous implementation papers did not explain why they restrict H in this way.

2.2 Decoding (QC-)MDPC Codes

As opposed to many other linear codes that allow efficient deterministic decoding, the most popular decoder for (QC-)MDPC code, the "bit-flipping" algorithm, is a probabilistic one. The bit-flipping algorithm shares the same idea with so-called "statistical decoding" [21, 19]. (The term "statistical decoding" historically come later than "bit-flipping", but "statistical decoding" captures way better the idea behind the algorithm.)

Given a vector that is at most t errors away from a codeword, the algorithm aims to output the codeword (or equivalently, the error vector) in a sequence of iterations. Each iteration decides statistically which of the n positions of the input vector v might have a higher chance to be in error and flips the bits at those positions. The flipped vector then becomes the input to the next iteration. In the simplest form of the algorithm, the algorithm terminates when Hv becomes zero.

The presumed chance of each position being in error is indicated by the count of unsatisfied parity checks. The higher the count is, the higher the presumed chance a position is in error. In other words, the chance of position i being in error is indicated by

$$u_i = |\{i \mid H_{i,j} = (Hv)_i = 1\}|.$$

In this paper the syndrome Hv will be called the *private syndrome.*

Now the remaining problem is, which bits should be flipped given the vector u? In [3] two possibilities are given:

- Flip all positions that violate at least $\max(\{u_i\}) - \delta$ parity checks, where δ is a small integer, say 5.
- Flip all positions that violate at least T_i parity checks, where T_i is a precomputed threshold for iteration i.

In previous works several variants have been invented. For example, one variant based on the first approach simply restarts decoding with a new δ if decoding fails in 10 iterations.

`QcBits` uses precomputed thresholds. The number of decoding iterations is set to be 6, and the thresholds are

$$29, 27, 25, 24, 23, 23.$$

These thresholds are obtained by interactive experiments. I do not claim that these are the best thresholds. With this list of thresholds, no iteration failure occurs in 10^8 decoding trials. See Section 6 for more details about the trials.

The best results in previous implementation papers [4, 5, 7, 9, 11] are achieved by a variant of the precomputed-threshold approach. In each iteration of the variant, the u_i's are computed in order. If the current u_i is greater than or equal to the precomputed threshold, v_i is flipped and the syndrome is directly updated by adding the i-th column of H to the syndrome. With this variant, [11] reports that the average number of iterations is only 2.4.

`QcBits` always takes 6 decoding iterations, which is much more than 2.4. However, the algorithms presented in the following sections allow `QcBits` to run each iteration very quickly, albeit being constant-time. As the result, `Qubits` still achieves much better performance results in decryption.

2.3 The Hybrid Niederreiter Encryption System for QC-MDPC Codes

The KEM/DEM encryption uses the Niederreiter encryption scheme for KEM. Niederreiter encryption is used to encrypt a *random* vector e of weight t, which is then fed into a key-derivation function to obtain the symmetric encryption and authentication key. The ciphertext is then the concatenation of the Niederreiter ciphertext, the symmetric ciphertext, and the authentication tag for the symmetric ciphertext. The decryption works in a similar way as encryption; see for example [**17**] for a more detailed description. By default `QcBits` uses the following symmetric primitives: Keccak [**23**], Salsa20 [**25**], and Poly1305 [**24**]. To be more precise, `QcBits` uses Keccak with 512-bit outputs to hash e, and the symmetric encryption and authentication key are defined to be the first and second half of the hash value. For symmetric encryption and authentication, `QcBits` uses Salsa20 with nonce 0 and Poly1305. Note that `QcBits` does not implement Keccak, Salsa20, and Poly1305; it only provides an interface to call these primitives. For the experiments results in table 1, the implementations of the symmetric primitives are from the SUPERCOP benchmarking toolkit. The user can use their own implementations for the primitives, or even use some other symmetric primitives (in this case the user has to change the hard-coded parameters, such as key size of the MAC).

The secret key is a representation of a random parity-check matrix H. Since the first row H gives enough information to form the whole matrix, it suffices to represent H using an array of indices in $\{j \mid H^{(0)}_{0,j} = 1\}$ and an array of indices in $\{j \mid H^{(1)}_{0,j} = 1\}$. In each array the indices should not repeat, but they are *not* required to be sorted. `QcBits` represents each array as a byte stream of length w, where the i-th double byte is the little-endian representation of the i-th index in the array. The secret key is then defined as the concatenation of the two byte streams.

The public key is a representation of the row reduced echelon form of H. The row reduced matrix is denoted as P. Niederreiter requires $P^{(0)}$ to be the identity matrix I_r, or the key pair must be rejected. (Previous papers such as [**7**] use $P^{(1)} = I_r$, but using $P^{(0)} = I_r$ is equivalent in terms of security.) In other words, $P^{(1)}$ contains all information of P (if P is valid). Note that P is also quasi-cyclic; `QcBits` thus defines the public key as a byte stream of length $\lfloor (r+7)/8 \rfloor$, where the byte values are

$$(P^{(1)}_{7,0}P^{(1)}_{6,0}\dots P^{(1)}_{0,0})_2,\ (P^{(1)}_{15,0}P^{(1)}_{14,0}\dots P^{(1)}_{8,0})_2,\ \dots$$

The encryption process begins with generating a random vector e of weight t. The ciphertext for e is then the public syndrome $s = Pe$, which is represented as a byte stream of length $\lfloor (r+7)/8 \rfloor$, where the byte values are

$$(s_7 s_6 \dots s_0)_2,\ (s_{15} s_{14} \dots s_8)_2,\ \dots.$$

For hashing, e is represented as a byte stream of length $\lfloor (n+7)/8 \rfloor$ in a similar way as the public syndrome. The 32-byte symmetric encryption key and the 32-byte authentication key are then generated as the first and second half of the 64-byte hash value of the byte stream. The plaintext m is encrypted and authenticated using the symmetric keys. The ciphertext for the whole KEM/DEM scheme is then the concatenation of the public syndrome, the ciphertext under symmetric encryption, and the tag. In total the ciphertext takes $\lfloor (r+7)/8 \rfloor + |m| + 16$ bytes.

When receiving an input stream, the decryption process parses it as the concatenation of a public syndrome, a ciphertext under symmetric encryption, and a tag. Then an error vector e' is computed by feeding the public syndrome into the decoding algorithm. If $Pe' = s$, decoding is successful. Otherwise, a decoding failure occurs. The symmetric keys are then generated by hashing e' to perform symmetric decryption and verification. `QcBits` reports a decryption failure if and only if the verification fails or the decoding fails.

3 Key-Pair Generation

This section shows how `QcBits` performs key-pair generation using multiplications in $\mathbb{F}_2[x]/(x^r - 1)$. Section 3.1 shows how the private key is generated. Section 3.2 shows how key-pair generation is viewed as multiplications in $\mathbb{F}_2[x]/(x^r - 1)$. Section 3.3 shows how multiplications in $\mathbb{F}_2[x]/(x^r - 1)$ are implemented. Section 3.4 shows how squarings in $\mathbb{F}_2[x]/(x^r - 1)$ are implemented.

3.1 Private-Key Generation

The private-key is defined to be an array of w random 16-bit indices. `QcBits` obtains random bytes by first reading 32 bytes from a source of randomness and then expands the 32 bytes into the required length using `salsa20`. `QcBits` allows the user to choose any source of randomness. To generate the performance numbers on Ivy Bridge, Sandy Bridge, and Cortex-A8 in Table 1, `/dev/urandom` is used as the source of randomness. To generate the performance numbers on Cortex-M4 in Table 1, the TRNG on the board is used as in [9]. The `RDRAND` instruction used by [11] is not considered for there are security concerns about the instruction; see the Wikipedia page of `RDRAND` [26]. One can argue that there is no evidence of a backdoor in `RDRAND`, but I decide not to take the risk.

3.2 Polynomial View: Public-Key Generation

For any matrix M, let $M_{i,:}$ denote the vector $(M_{i,0}, M_{i,1}, \dots)$ and similarly for $M_{:,i}$. In Section 2, the public key is defined as a sequence of bytes representing $P^{(1)}_{:,0}$, where P is the row reduced echelon form of the parity-check matrix H. A simple way to implement constant-time public-key generation is thus to generate H from the private key and then perform a Gaussian elimination. It is not hard to make Gausssian elimination constant-time; see for example, [17]. However, public-key generation can be made much more time- and memory-efficient when considering it as polynomial operations, making use of the quasi-cyclic structure.

For any vector v of length r, let $v(x) = v_0 + v_1 x + \cdots + v_{r-1}x^{r-1}$. As a result of $H^{(0)}$ being cyclic, we have

$$H^{(i)}_{j,:}(x) = x^j H^{(i)}_{0,:}(x) \in \mathbb{F}_2[x]/(x^r - 1).$$

The Gaussian elimination induces a linear combination of the rows of $H^{(0)}$ that results in $P^{(0)}_{0,:}$. In other words, there exists a set I of indices such that

$$\begin{aligned} 1 &= \sum_{i \in I} x^i H^{(0)}_{0,:}(x) = (\sum_{i \in I} x^i) H^{(0)}_{0,:}(x), \\ P^{(1)}_{0,:}(x) &= \sum_{i \in I} x^i H^{(1)}_{0,:}(x) = (\sum_{i \in I} x^i) H^{(1)}_{0,:}(x). \end{aligned}$$

In other words, the public key can be generated by finding the inverse of $H^{(0)}_{0,:}(x)$ in $\mathbb{F}_2[x]/(x^r - 1)$ and then multiplying the inverse by $H^{(1)}_{0,:}(x)$. The previous implementation papers [**4**, **5**, **7**, **9**, **11**] compute the inverse using the extended Euclidean algorithm. The algorithm in its original form is highly non-constant-time. [**30**] provides a way to make extended Euclidean algorithm constant-time; so far it is unclear to me whether their algorithm is faster than simply using exponentiation (see below).

In order to be constant-time, `QcBits` computes the inverse by carrying out a fixed sequence of polynomial multiplications. To see this, first consider the factorization of $x^r - 1 \in \mathbb{F}_2[x]$ as $\prod_i \left(f^{(i)}(x)\right)^{p_i}$, where each $f^{(i)}$ is irreducible. $\mathbb{F}_2[x]/(x^r - 1)$ is then equivalent to

$$\prod_i \mathbb{F}_2[x] \Big/ \left(f^{(i)}(x)\right)^{p_i}$$

Since

$$\begin{aligned} \left| \left(\mathbb{F}_2[x] / \left(f^{(i)}(x)\right)^{p_i} \right)^* \right| &= 2^{\deg(f^{(i)}) \cdot p_i} \cdot (2^{\deg(f^{(i)})} - 1)/2^{\deg(f^{(i)})} \\ &= 2^{\deg(f^{(i)}) \cdot p_i} - 2^{\deg(f^{(i)}) \cdot (p_i - 1)}, \end{aligned}$$

one may compute the inverse of an element in $\mathbb{F}_2[x]/(x^r - 1)$ by raising it to power

$$\mathrm{lcm}\left(2^{\deg(f^{(1)}) \cdot p_1} - 2^{\deg(f^{(1)}) \cdot (p_1 - 1)}, 2^{\deg(f^{(2)}) \cdot (p_2 - 1)} - 2^{\deg(f^{(2)}) \cdot (p_2 - 1)}, \ldots\right) - 1.$$

`QcBits` uses $r = 4801$. The polynomial $x^{4801} - 1$ can be factored into

$$(x + 1) f^{(1)} f^{(2)} f^{(3)} f^{(4)} \in \mathbb{F}_2[x],$$

where each $f^{(i)}$ is an irreducible polynomial of degree 1200. Therefore, `QcBits` computes the inverse of a polynomial modulo $x^{4801} - 1$ by raising it to the power $\mathrm{lcm}(2 - 1, 2^{1200} - 1) - 1 = 2^{1200} - 2$.

Raising an element in $\mathbb{F}_2[x]/(x^{4801}-1)$ to the power $2^{1200}-2$ can be carried out by a sequence of squarings and multiplications. The most naive way is to use the square-and-multiply algorithm, which leads to 1199 squarings and 1198 multiplications. `QcBits` does better by finding a good addition chain for $2^{1200}-2$. First note that there is a systematic way to find a good addition chain for integers of the form 2^k-1. Take $2^{11}-1$ for example, the chain would be

$$1 \to 10_2 \to 11_2 \to 1100_2 \to 1111_2 \to 11110000_2 \to 11111111_2 \to 1111111100_2 \\ \to 1111111111_2 \to 11111111110_2 \to 11111111111_2.$$

This takes 10 doublings and 5 additions. Using the same approach, it is easy to find an addition chain for $2^{109}-1$ that takes 108 doublings and 10 additions. `QcBits` then combines the addition chains for $2^{11}-1$ and $2^{109}-1$ to form an addition chain for $2^{11\cdot 109}-1 = 2^{1199}-1$, which takes $10\cdot 109+108 = 1198$ doubling and $5+10 = 15$ additions. Once the $(2^{1199}-1)$-th power is computed, the $(2^{1200}-2)$-th power can be computed using one squaring. In total, computation of the $(2^{1200}-2)$-th power takes 1199 squarings and 15 multiplications in $\mathbb{F}_2[x]/(x^{4801}-1)$.

Finally, with the inverse, $P^{(1)}_{0,:}(x)$ can be computed using one multiplication. The public key is defined to be a representation of $P^{(1)}_{:,0}$ instead of $P^{(1)}_{0,:}$. `Qcbits` thus derives $P^{(1)}_{0,:}$ from $P^{(1)}_{:,0}$ by noticing

$$P^{(1)}_{0,j} = \begin{cases} P^{(1)}_{0,r-j} & \text{if } j > 0 \\ P^{(1)}_{0,0} & \text{if } j = 0. \end{cases}$$

Note that the conversion from $P^{(1)}_{:,0}$ to $P^{(1)}_{0,:}$ does not need to be constant-time because it can be easily reversed from public data.

3.3 Generic Multiplication in $\mathbb{F}_2[x]/(x^r-1)$

The task here is to compute $h = fg$, where $f, g \in \mathbb{F}_2[x]/(x^r-1)$. In `QcBits`, the polynomials are represented using an array of $\lceil r/b \rceil$ b-bit words in the natural way. Take f for example (the same applies to g and h), the b-bit values are:

$$(f_{b-1}f_{b-2}\cdots f_0)_2, (f_{2b-1}f_{2b-2}\cdots f_b)_2, \ldots.$$

The user can choose b to be 32 or 64, but for the best performance b should be chosen according to the machine architecture. Let $y = x^b$. One can view this representation as storing each coefficient of the radix-y representation of f using one b-bit integer. In this paper this representation is called the "dense representation".

Using the representation, we can compute the coefficients (each being a $2b$-bit value) of the radix-y representation of h, using carry-less multiplications on the b-bit words of f and g. Once the $2b$-bit values are obtained, the dense representation of h can be computed with a bit of post-processing. To be precise, given two b-bit numbers $(\alpha_{b-1}\alpha_{b-2}\cdots\alpha_0)_2$ and $(\beta_{b-1}\beta_{b-2}\cdots\beta_0)_2$, a carry-less multiplication computes the $2b$-bit value (having actually only $2b-1$ bits)

$$\left(\bigoplus_{i+j=2b-2}\alpha_i\beta_j \bigoplus_{i+j=2b-3}\alpha_i\beta_j \dots \bigoplus_{i+j=0}\alpha_i\beta_j\right)_2 .$$

In other words, the input values are considered as elements in $\mathbb{F}_2[x]$, and the output is the product in $\mathbb{F}_2[x]$.

The implementations `clmul` uses the `PCLMULQDQ` instruction to perform carry-less multiplications between two 64-bit values. For the implementation `ref` and `no-cache`, the following C code is used to compute the higher and lower b bits of the $2b$-bit value:

```
low = x * ((y >> 0) & 1);
v1 = x * ((y >> 1) & 1);
low  ^= v1 << 1;
high = v1 >> (b-1);
for (i = 2; i < b; i+=2)
{
        v0 = x * ((y >> i) & 1);
        v1 = x * ((y >> (i+1)) & 1);
        low  ^= v0 << i;
        low  ^= v1 << (i+1);
        high ^= v0 >> (b-i);
        high ^= v1 >> (b-(i+1));
}
```

3.4 Generic Squaring in $\mathbb{F}_2[x]/(x^r - 1)$

Squarings in $\mathbb{F}_2[x]/(x^r - 1)$ can be carried out as multiplications. However, obviously squaring is a much cheaper operation as only $\lceil r/b \rceil$ carry-less multiplications (actually squarings) are required.

The implementation `clmul` again uses the `PCLMULQDQ` instruction to perform carry-less squarings of 64-bit polynomials. Following the section for interleaving bits presented in the “Bit Twiddling Hacks” by Sean Eron Anderson [12], the implementations `ref` and `no-cache` use the following C code twice to compute the square of a 32-bit polynomial represented as 32-bit word:

```
x = (x | (x << 16)) & 0x0000FFFF0000FFFF;
x = (x | (x <<  8)) & 0x00FF00FF00FF00FF;
x = (x | (x <<  4)) & 0x0F0F0F0F0F0F0F0F;
x = (x | (x <<  2)) & 0x3333333333333333;
x = (x | (x <<  1)) & 0x5555555555555555;
```

By using the code twice we can also compute the square of a 64-bit polynomial.

4 KEM Encryption

This section shows how QcBits performs the KEM encryption using multiplications in $\mathbb{F}_2[x]/(x^r - 1)$. Section 4.1 shows how the error vector is generated. Section 4.2 shows how public-syndrome computation is viewed as multiplications in $\mathbb{F}_2[x]/(x^r - 1)$. Section 4.3 shows how these multiplications are implemented.

4.1 Generating the Error Vector

The error vector e is generated in essentially the same way as the private key. The only difference is that for e we need t indices ranging from 0 to $n - 1$, and there is only one list of indices instead of two. Note that for hashing it is still required to generate the dense representation of e.

4.2 Polynomial View: Public-Syndrome Computation

The task here is to compute the public syndrome Pe. Let $e^{(0)}$ and $e^{(1)}$ be the first and second half of e. The public syndrome is then

$$\begin{aligned} s &= P^{(0)}e^{(0)} + P^{(1)}e^{(1)} \\ &= \sum_i P^{(0)}_{:,i} e^{(0)}_i + \sum_i P^{(1)}_{:,i} e^{(1)}_i. \end{aligned}$$

Since P is quasi-cyclic, we have

$$\begin{aligned} s(x) &= \sum_i x^i P^{(0)}_{:,0}(x) e^{(0)}_i + \sum_i x^i P^{(1)}_{:,0}(x) e^{(1)}_i \\ &= P^{(0)}_{:,0}(x) e^{(0)}(x) + P^{(1)}_{:,0}(x) e^{(1)}(x) \\ &= e^{(0)}(x) + P^{(1)}_{:,0}(x) e^{(1)}(x). \end{aligned}$$

In other words, the private syndrome can be computed using one multiplication in $\mathbb{F}_2[x]/(x^r - 1)$. The multiplication is not generic in the sense that $e^{(1)}(x)$ is sparse. See below for how the multiplication is implemented in QcBits.

4.3 Sparse-Times-Dense Multiplications in $\mathbb{F}_2[x]/(x^r - 1)$

The task here can be formalized as computing $f^{(0)} + f^{(1)}g^{(1)} \in \mathbb{F}_2[x]/(x^r - 1)$, where $g^{(1)}$ is represented in the dense representation. $f^{(0)}$ and $f^{(1)}$ are represented together using an array of indices in $I = \{i \mid f^{(0)}_i = 1\} \cup \{i + r \mid f^{(1)}_i = 1\}$, where $|I| = t$.

One can of course perform this multiplication between $f^{(1)}$ and $g^{(1)}$ in a generic way, as shown in Section 3.3. The implementation clmul indeed generates the dense representation of $f^{(1)}$ and then computes $f^{(1)}g^{(1)}$ using the PCLMULQDQ instruction. [11] uses essentially the same technique. The implementations ref and `no-cache` however, make use of the sparsity in $f^{(0)}$ and $f^{(1)}$; see below for details.

Now consider the slightly simpler problem of computing $h = fg \in \mathbb{F}_2[x]/(x^r - 1)$, where f is represented as an array of indices in $I = \{i \mid f_i = 1\}$, and g is in the dense representation. Then we have

$$fg = \sum_{i \in I} x^i g.$$

Therefore, the implementations `ref` and `no-cache` first set $h = 0$. Then, for each $i \in I$, $x^i g$ is computed and then added to h. Note that $x^i g$ is represented as an array of $\lceil r/b \rceil$ b-bit words, so adding $x^i g$ to h can be implemented using $\lceil r/b \rceil$ bitwise-`XOR` instructions on b-bit words.

Now the remaining problem is how to compute $x^i g$. It is obvious that $x^i g$ can be obtained by rotating g by i bits. In order to perform a constant-time rotation, the implementation `ref` makes use of the idea of the Barrel shifter [**27**]. The idea is to first represent i in binary representation

$$(i_{k-1} i_{k-2} \cdots i_0)_2.$$

Since $i \leqslant r-1$, it suffices to use $k = \lfloor \lg(r-1) \rfloor + 1$. Then, for j from $k-1$ to $\lg b$, a rotation by 2^j bits is performed. One of the unshifted vector and the shifted vector is chosen (in a constant-time way) and serves as the input for the next j. After dealing with all $i_{k-1}, i_{k-2}, \ldots, i_{\lg b}$, a rotation of $(i_{\lg b - 1} i_{\lg b - 2} \cdots i_0)_2$ bits is performed using a sequence of logical instructions.

To clarify the idea, here is a toy example for the case $n = 40, b = 8$. The polynomial g is

$$(x^8 + x^{10} + x^{12} + x^{14}) + (x^{16} + x^{17} + x^{20} + x^{21}) + (x^{24} + x^{25} + x^{26} + x^{27}) \\ + (x^{36} + x^{37} + x^{38} + x^{39}),$$

which is represented in an array of 5 bytes as

$$00000000_2, 01010101_2, 00110011_2, 00001111_2, 11110000_2.$$

The goal is to compute $x^i g$ where $i = 010011_2$. Since $\lfloor \lg(40-1) \rfloor + 1 = 6$, the algorithm begins with computing a rotation of $100000_2 = 32$ bits, which can be carried out by moving around the bytes. The result is

$$01010101_2, 00110011_2, 00001111_2, 11110000_2, 00000000_2.$$

Since the most significant bit is not set, the unshifted polynomial is chosen. Next we proceed to perform a rotation of $010000_2 = 16$ bits. The result is

$$00001111_2, 11110000_2, 00000000_2, 01010101_2, 00110011_2.$$

Since the second most significant bit is set, we choose the rotated polynomial. The polynomial is then shifted by $001000_2 = 8$ bits. However, since the third most significant bit is not set, the unshifted polynomial is chosen. To handle the least significant $\lg b = 3$ bits of i, a sequence of logical instructions are used to combine the most significant 011_2 and the least significant 101_2 bits of the bytes, resulting in

$$01100001_2, 11111110_2, 00000000_2, 00001010_2, 10100110_2.$$

Note that in [3] r is required to be a prime (which means r is not divisible by b), so the example is showing an easier case. Roughly speaking, the implementation `ref` performs a rotation as if the vector length is $r - (r \bmod b)$ and then uses more instructions to compensate for the effect of the $r \bmod b$ extra bits. The implementation `no-cache` essentially performs a rotation of $(i_{k-1}i_{k-2} \cdots i_{\lg b}0 \cdots 0)_2$ bits and then performs a rotation of $(i_{\lg b-1}i_{\lg b-2} \cdots i_0)_2$ bits.

With the constant-time rotation, we can now deal with the original problem of computing $f^{(0)} + f^{(1)}g^{(1)} \in \mathbb{F}_2[x]/(x^r - 1)$. `QcBits` first sets $h = 0$. Then for each $i \in I$, one of either 1 or $g^{(1)}$ is chosen according to whether $i < r$ or not, which has to be performed in a constant-time way to hide all information about i. The chosen polynomial is then rotated by $i \bmod r$ bits, and the result is added to h. Note that this means the implementations `ref` and `no-cache` perform a dummy polynomial multiplication to hide information about $f^{(0)}$ and $f^{(1)}$.

5 KEM Decryption

This section shows how `QcBits` performs the KEM decryption using multiplications in $\mathbb{F}_2[x]/(x^r - 1)$ and $\mathbb{Z}[x]/(x^r - 1)$. The KEM decryption is essentially a decoding algorithm. Each decoding iteration computes

- the private syndrome Hv and
- the counts of unsatisfied parity checks, i.e., the vector u, using the private syndrome.

Positions in v are then flipped according the counts. Section 5.1 shows how private-syndrome computation is implemented as multiplications in $\mathbb{F}_2[x]/(x^r - 1)$. Section 5.2 shows how counting unsatisfied parity checks is viewed as multiplications in $\mathbb{Z}[x]/(x^r - 1)$. Section 5.3 shows how these multiplications in $\mathbb{Z}[x]/(x^r - 1)$ are implemented. Section 5.4 shows how bit flipping is implemented.

5.1 Polynomial View: Private-Syndrome Computation

The public syndrome and the private syndrome are similar in the sense that they are both computed by matrix-vector products where the matrices are quasi-cyclic. For the public syndrome the matrix is P and the vector is e. For the private syndrome the matrix is H and the vector is v. Therefore, the computation of the private syndrome can be viewed as polynomial multiplication in the same way as the public syndrome. That is, the private syndrome can be viewed as

$$H^{(0)}_{:,0}(x)v^{(0)}(x) + H^{(1)}_{:,0}(x)v^{(1)}(x) \ \in \mathbb{F}_2[x]/(x^r - 1).$$

The computations of the public syndrome and the private syndrome are still a bit different. For encryption the matrix P is dense, whereas the vector e is sparse. For decryption the matrix H is sparse, whereas the vector v is dense. However, the multiplications $H^{(i)}_{:,0}(x)v^{(i)}(x)$ are still sparse-times-dense multiplications. `QcBits` thus computes the private syndrome using the techniques described in Section 4.3.

Since the secret key is a sparse representation of $H^{(i)}_{0,:}$, we do not immediately have $H^{(i)}_{:,0}$. This is similar to the situation in public-key generation, where $P^{(1)}_{:,0}$ is derived from $P^{(1)}_{0,:}$. `QcBits` thus computes $H^{(i)}_{:,0}$ from $H^{(i)}_{0,:}$ by adjusting each index in the sparse representation in constant time.

5.2 Polynomial View: Counting Unsatisfied Parity Checks

Let $s = Hv$. The vector u of counts of unsatisfied parity checks can be viewed as

$$u_j = \sum_i H_{i,j} \cdot s_i \in \mathbb{Z}^n,$$

where $H_{i,j}$ and s_j are treated as integers. In other words,

$$u = \sum_i H_{i,:} \cdot s_i \in \mathbb{Z}^n.$$

Let $u^{(0)}$ and $u^{(1)}$ be the first and second half of u, respectively. Now we have:

$$\begin{aligned}\left(u^{(0)}(x), u^{(1)}(x)\right) &= \left(\sum_i x^i H^{(0)}_{0,:}(x) \cdot s_i, \sum_i x^i H^{(1)}_{0,:}(x) \cdot s_i\right) \\ &= \left(H^{(0)}_{0,:}(x) \cdot s(x), H^{(1)}_{0,:}(x) \cdot s(x)\right) \in \left(\mathbb{Z}[x]/(x^r - 1)\right)^2.\end{aligned}$$

In other words, the vector u can be computed using 2 multiplications in $\mathbb{Z}[x]/(x^r - 1)$. Note that the multiplications are not generic: $H^{(i)}_{0,:}(x)$ is always sparse, and the coefficients of $H^{(i)}_{0,:}(x)$ and $s(x)$ can only be 0 or 1. See below for how such multiplications are implemented in `QcBits`.

5.3 Sparse-Times-Dense Multiplications in $\mathbb{Z}[x]/(x^r - 1)$

The task can be formalized as computing $fg \in \mathbb{Z}[x]/(x^r - 1)$, where $f_i, g_i \in \{0, 1\}$ for all i, and f is of weight only w. f is represented as an array of indices in $I_f = \{i \mid f_i = 1\}$. g is naturally represented as an array of $\lceil r/b \rceil$ b-bit values as usual. Then we have

$$fg = \sum_{i \in I_f} x^i g.$$

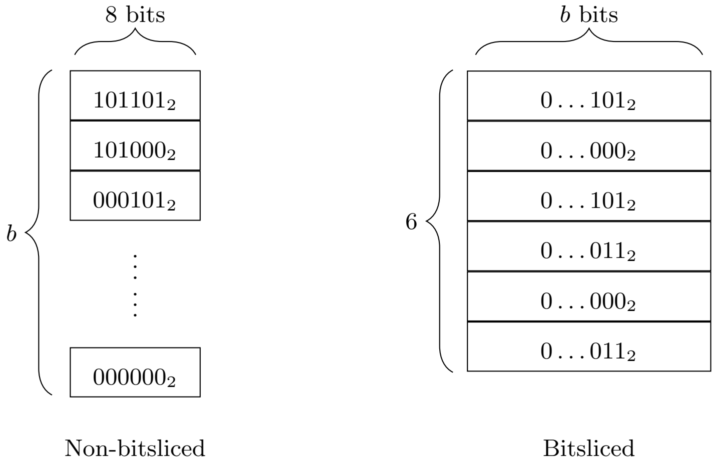

Fig. 1. Storage of b numbers of unsatisfied parity checks in non-bitsliced form and bitsliced format.

Even though all the operations are now in $\mathbb{Z}[x]/(x^r - 1)$ instead of $\mathbb{F}_2[x]/(x^r - 1)$, each $x^i g$ can still be computed using a constant-time rotation as in Section 4.3. Therefore, `QcBits` first sets $h = 0$, and then for each $i \in I$, $x^i g$ is computed using the constant-time rotation and then added to h. After all the elements in I are processed, we have $h = fg$. Note that $x^i g$ is represented as an array of $\lceil r/b \rceil$ b-bit words.

Now the remaining problem is how to add $x^i g$ to h. A direct way to represent h is to use an array of r bytes (it suffices to use 1 byte for each coefficient when $w/2 < 256$, which is true for all parameter sets in [3] with $n = 2r$), each storing one of the r coefficients. To add $x^i g$ to h, the naive way is for each coefficient of h to extract from the corresponding b-bit word the bit required using one bitwise-`AND` instruction and at most one shift instruction, and then to add the bit to the byte using one addition instruction. In other words, it takes around 3 instructions on average to update each coefficient of h.

`QcBits` does better by bitslicing the coefficients of h: Instead of using b bytes, `QcBits` uses several b-bit words to store a group of b coefficients, where the i-th b-bit word stores the i-th least significant bits of the b coefficients. Since the column weight of H is $w/2$, it suffices to use $\lfloor \lg w/2 \rfloor + 1$ b-bit words. To update b coefficients of h, a sequence of logical operations is performed on the $\lfloor \lg w/2 \rfloor + 1$ b-bit words and the corresponding b-bit word in $x^i g$. These logical instructions simulate b copies of a circuit for adding a 1-bit number into a $(\lfloor \lg w/2 \rfloor + 1)$-bit number. Such a circuit requires roughly $\lfloor \lg w/2 \rfloor + 1$ half adders, so updating b coefficients takes roughly $2(\lfloor \lg w/2 \rfloor + 1)$ logical instructions on b-bit words.

Figure 1 illustrates how the b coefficients are stored when $w = 90$. In the non-bitsliced approach b bytes are used. In the bitsliced approach $\lfloor \lg(90/2) \rfloor + 1 = 6$ b-bit words are used, which account for $6b/8$ bytes. Note that this means bitslicing saves memory. Regarding the number of instructions, it takes $(6 \cdot 2)/b$ logical instructions on average to update each coefficient. For either $b = 32$ or $b = 64$, $(6\cdot 2)/b$ is much smaller than 3. Therefore, bitslicing also helps to enhance performance.

The speed that McBits [**17**] achieves relies on bitslicing as well. However, the reader should keep in mind that `QcBits`, as opposed to McBits, makes use of parallelism that lies intrinsically in one single decryption instance.

5.4 Flipping Bits

The last step in each decoding iteration is to flip the bits according to the counts. Since `QcBits` stores the counts in a bitsliced format, bit flipping is also accomplished in a bitsliced fashion. At the beginning of each decoding iteration, the bitsliced form of b copies of $-t$ is generated and stored in $\lfloor \lg w/2 \rfloor + 1$ b-bit words. Once the counts are computed, $-t$ is added to the counts in parallel using logical instructions on b-bit words. These logical instructions simulate copies of a circuit for adding $(\lfloor \lg w/2 \rfloor + 1)$-bit numbers. Such a circuit takes $(\lfloor \lg w/2 \rfloor + 1)$ full adders. Therefore, each $u_i + (-t)$ takes roughly $5(\lfloor \lg w/2 \rfloor + 1)/b$ logical instructions.

The additions are used to generate sign bits for all $u_i - t$, which are stored in two arrays of $\lceil r/b \rceil$ b-bit words. To flip the bits, `QcBits` simply XORs the complement of b-bit words in the two arrays into $v^{(0)}$ and $v^{(1)}$. It then takes roughly $1/b$ logical instructions to update each v_i.

For $w = 90$, we have $5(\lfloor \lg w/2 \rfloor + 1)/b + 1b = 31/b$, which is smaller than 1 for either $b = 32$ or $b = 64$. In contrast, when the non-bitsliced format is used, the naive approach is to use at least one subtraction instruction for each $u_i - t$ and one XOR instruction to flip the bit. One can argue that for the non-bitsliced format there are probably better ways to compute u and perform bit flipping. For example, one can probably perform several additions/subtractions of bytes in parallel in one instruction. However, such an approach seems much more expensive than one might expect as changes of formats between a sequence of bits and bytes are required.

6 Experimental Results for Decoding

This section shows experimental results for QC-MDPC decoding under different parameter sets. The decoding algorithm used is the precomputed-threshold approach introduced in Section 2.2. The codes are restricted: $H^{(0)}$ and $H^{(1)}$ are required to have the same row weight. $\mathtt{r}, \mathtt{w}, \mathtt{t}$ have same meaning as in Section 2.1. `sec` indicates the security level. `T` is the list of thresholds. If not specified otherwise, the thresholds are obtained using the formulas in [**3**, Appendix A]. `S` is a list that denotes how many iterations the tests take. The summation of the numbers in `S` is the total number of tests, which is set to either 10^8 first the first three cases and 10^6 for the last case. The 10^8 (10^6) tests consist of 10^4 (10^3) decoding attempts for each of 10^4 (10^3) key pairs. The first number in the list indicates the number of tests that fail to decode in #`T` iterations (i.e., in the total number of iterations). The second number indicates the number of tests that succeed after 1 iteration. The third number indicates the number of tests that succeed after 2 iterations; etc. `avg` indicates the average number of iterations for the successful tests.

```
r = 4801
w = 90
t = 84
sec = 80
T = [29, 27, 25, 24, 23, 23]
S = [0, 0, 752, 69732674, 30232110, 34417, 47]
avg = 3.30
```

The thresholds are obtained by interactive experiments. `QcBits` uses this setting.

```
r = 4801
w = 90
t = 84
sec = 80
T = [28, 26, 24, 23, 23, 23, 23, 23, 23, 23]
S = [40060, 0, 9794, 87815060, 12079266, 51387, 3833, 519, 70, 10,
1]
avg = 3.12
```

```
r = 9857
w = 142
t = 134
sec = 128
T = [44, 42, 40, 37, 36, 36, 36, 36, 36, 36, 36, 36]
S = [689298, 0, 0, 86592, 53307303, 42797368, 2856446, 235479,
24501, 2651, 333, 26, 3]
avg = 4.46
```

```
r = 9857
w = 142
t = 134
sec = 128
T = [48, 47, 46, 45, 44, 43, 42, 42, 41, 41, 40, 40, 39, 39, 38,
38, 37, 37, 36]
S = [12, 0, 0, 0, 0, 0, 142, 78876, 578963, 290615, 43180, 6363,
1309, 336, 108, 54, 27, 7, 4, 4]
avg = 8.33
```

The thresholds are obtained by interactive experiments.

7 The Future of QC-MDPC-Based Cryptosystems

QcBits provides a way to perform constant-time QC-MDPC decoding, even on platforms with caches. Moreover, decoding in QcBits is much faster than that in previous works. However, the fact that the bit-flipping algorithm is probabilistic can be a security issue. The security proofs in [13, 14] do assume that the KEM is able to decrypt a KEM ciphertext with "overwhelming probability". As there is no good way to estimate the failure rate for a given QC-MDPC code, the best thing people can do is to run a large number of experiments. QcBits manages to achieve no decoding failures in 10^8 trials. Indeed, 10^8 is not a trivial number, but whether such level of failure rate is enough to keep the system secure remains unclear, not to mention that this is for 80-bit security only. See Section 6 for more detailed experimental results on failure rates.

One can probably mitigate the problem by reducing the failure rate. This may be achieved by improving decoding algorithms or designing better parameter sets. However, a more fundamental problem that has not been answered is how low the failure rate should be in order to be secure.

Another probably less serious problem is that QcBits and all previous implementation papers [4, 5, 7, 9, 11] force the parity check matrix H to have equal weights in $H^{(0)}$ and $H^{(1)}$, which is not the same as what was described in [3]. QcBits restricts the key space in this way to reduce the failure rate. Of course, one can argue that even if the key space is not restricted, for a very high probability $H^{(0)}$ and $H^{(1)}$ would still have the same weight. However, such an argument is valid only if the adversary can only target one system. For an adversary who aims to break one out of many systems, it is still unclear whether such restriction affects the security. Hopefully researchers will spend time on this problem also.

References

[1] Robert J. McEliece, *A public-key cryptosystem based on algebraic coding theory*, JPL DSN Progress Report (1978), 114–116. URL: http://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF. Citations in this document: §1, §1.

[2] Harald Niederreiter, *Knapsack-type cryptosystems and algebraic coding theory*, Problems of Control and Information Theory **15** (1986), 159–166. Citations in this document: §1.

[3] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, Paulo S. L. M. Barreto, *MDPC-McEliece: New McEliece variants from moderate density parity-check codes*, IEEE International Symposium on Information Theory (2013), 2069–2073. URL: http://eprint.iacr.org/2012/409.pdf. Citations in this document: §1, §1, §1, §1, §1, §1, §2.1, §2.1, §2.2, §4.3, §5.3, §6, §7.

[4] Stefan Heyse, Ingo von Maurich, Tim Güneysu, *Smaller Keys for Code-Based Cryptography: QC-MDPC McEliece Implementations on Embedded Devices*, in CHES 2013 [18] (2013), 273–292. URL: http://eprint.iacr.org/2015/425.pdf.

[5] Ingo von Maurich, Tim Güneysu, *Lightweight code-based cryptography: QC-MDPC McEliece encryption on reconfigurable devices*, in DATE 2014 [6]

(2014), 1–6. URL: https://www.sha.rub.de/media/sh/veroeffentlichungen/2014/02/11/Lightweight_Code-based_Cryptography.pdf.

[6] Gerhard Fettweis, Wolfgang Nebel (editors), *Design, Automation and Test in Europe Conference and Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, European Design and Automation Association, 2014. ISBN 978-3-9815370-2-4. See [5].

[7] Ingo von Maurich, Tim Güneysu, *Towards Side-Channel Resistant Implementations of QC-MDPC McEliece Encryption on Constrained Devices*, in PQCrypto 2014 [**8**] (2014), 266–282. URL: http://ei.rub.de/media/sh/veroeffentlichungen/2014/07/17/Side-Channel_Resistant_QC-MDPC_McEliece.pdf. Citations in this document: §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §2.3.

[8] Michele Mosca (editor), *Post-quantum cryptography*, Lecture Notes in Computer Science, 8772, Springer, 2014. See [7].

[9] Ingo von Maurich, Lukas Heberle, Tim Güneysu, *IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter*, in PQCrypto 2016 [**10**] (2016). Citations in this document: §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §2, §2, §2, §3.1.

[10] Tsuyoshi Takagi (editor), *Post-quantum cryptography*, Lecture Notes in Computer Science, 9606, Springer, 2016. See [9].

[11] Ingo von Maurich, Tobias Oder, Tim Güneysu, *Implementing QC-MDPC McEliece Encryption*, ACM Transactions on Embedded Computing Systems **14** (2015), 44. Citations in this document: §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §1, §2, §2, §2, §2.2, §3.1, §4.3.

[12] Sean Eron Anderson, *Bit Twiddling Hacks* (1997-2005). URL: https://graphics.stanford.edu/~seander/bithacks.html. Citations in this document: §3.4.

[13] Edoardo Persichetti, *Improving the efficiency of code-based cryptography*, Ph.D. thesis, University of Auckland, 2012. URL: http://persichetti.webs.com/publications.

[14] Edoardo Persichetti, *Secure and Anonymous Hybrid Encryption from Coding Theory*, in PQCrypto 2013 [**15**] (2013), 174–187. URL: http://persichetti.webs.com/publications.

[15] Philippe Gaborit (editor), *Post-quantum cryptography*, Lecture Notes in Computer Science, 7932, Springer, 2013. See [14].

[16] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 2 February 2016 (2016). URL: http://bench.cr.yp.to. Citations in this document: §1, §1.

[17] Daniel J. Bernstein, Tung Chou, Peter Schwabe, *McBits: fast constant-time code-based cryptography*, in CHES 2013 [**18**] (2013), 250–272. Citations in this document: §1, §1, §1, §1, §2.3, §3.2, §5.3.

[18] Guido Bertoni, Jean-Sébastien Coron (editors), *Cryptographic hardware and embedded systems—CHES 2013—15th international workshop, Santa Barbara, CA, USA, August 20–23, 2013, proceedings*, Lecture Notes in Computer Science, 8086, Springer, 2013. ISBN 978-3-642-40348-4. See [4], [17].

[19] A. Kh. Al Jabri, *A statistical decoding algorithm for general linear block codes*, in [**20**] (2001), 1–8.

[20] Bahram Honary (editor), *Cryptography and Coding, 8th IMA International Conference, Cirencester, UK, December 17-19, 2001, Proceeding*, Lecture Notes in Computer Science, 2260, Springer, 2001. ISBN 3-540-43026-1. See [19].

[21] Raphael Overbeck, *Statistical Decoding Revisited*, in ACISP 2006 [**22**] (2006), 283–294.

[22] Lynn Margaret Batten, Reihaneh Safavi-Naini (editors), *Information Security and Privacy, 11th Australasian Conference, ACISP 2006, Melbourne, Australia, July 3-5, 2006, Proceedings*, Lecture Notes in Computer Science, 4058, Springer. ISBN 3-540-35458-1. See [21].

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, *Keccak and the SHA-3 standardization* (2013). URL: http://csrc.nist.gov/groups/ST/hash/sha-3/documents/Keccak-slides-at-NIST.pdf. Citations in this document: §2.3.

[24] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in FSE 2005 [**28**] (2005), 32–49. URL: http://cr.yp.to/papers.html#poly1305. Citations in this document: §2.3.

[25] Daniel J. Bernstein, *The Salsa20 family of stream ciphers*, in [**29**] (2008), 84–97. URL: http://cr.yp.to/papers.html#salsafamily. Citations in this document: §2.3.

[26] Wikipedia, *RdRand — Wikipedia, The Free Encyclopedia*, accessed 2 February 2016 (2016). URL: https://en.wikipedia.org/wiki/RdRand. Citations in this document: §3.1.

[27] Wikipedia, *Barrel shifter — Wikipedia, The Free Encyclopedia*, accessed 2 February 2016 (2016). URL: https://en.wikipedia.org/wiki/Barrel_shifter. Citations in this document: §4.3.

[28] Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3-540-26541-4. See [24].

[29] Matthew Robshaw, Olivier Billet (editors), *New stream cipher designs*, Lecture Notes in Computer Science, 4986, Springer, 2008. ISBN 978-3-540-68350-6. See [25].

[30] Mariya Georgieva, Frédéric de Portzamparc, *Toward Secure Implementation of McEliece Decryption*, in COSADE 2015 [**31**] (2015), 141-156. URL: http://eprint.iacr.org/2015/271.pdf. Citations in this document: §3.2.

[31] Stefan Mangard, Axel Y. Poschmann (editors), *Constructive Side-Channel Analysis and Secure Design*, Lecture Notes in Computer Science, 9064, Springer, 2015. See [30].

[32] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, Peter Schwabe, *Post-quantum key exchange—a new hope* (2015), *The IACR ePrint Archive*, 2015. URL: https://eprint.iacr.org/2015/1092. Citations in this document: §1.