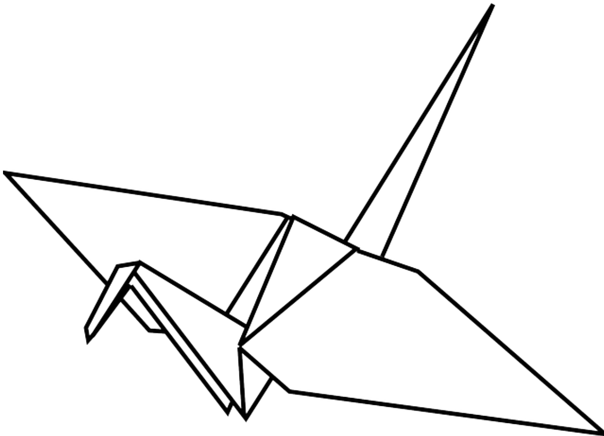
LEDAcrypt: Low-dEnsity parity-check coDe-bAsed cryptographic systems

Specification revision 3.0 – April, 2020



Name of the proposed cryptosystem

LEDAcrypt (Low-dEnsity parity-check coDe-bAsed cryptographic systems)

Submitters

This submission is from the following team, listed in alphabetical order:

- Marco Baldi, Università Politecnica delle Marche, Ancona, Italy
- Alessandro Barenghi, Politecnico di Milano, Milano, Italy
- Franco Chiaraluce, Università Politecnica delle Marche, Ancona, Italy
- Gerardo Pelosi, Politecnico di Milano, Milano, Italy
- Paolo Santini, Università Politecnica delle Marche, Ancona, Italy

E-mail addresses: m.baldi@univpm.it, alessandro.barenghi@polimi.it, f.chiaraluce@univpm.it, gerardo.pelosi@polimi.it, p.santini@pm.univpm.it.

Contact telephone and address

Marco Baldi (phone: +39 071 220 4894), Università Politecnica delle Marche, Dipartimento di Ingegneria dell'Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy.

Names of auxiliary submitters

There are no auxiliary submitters. The principal submitter is the team listed above.

Name of the inventors/developers of the cryptosystem

Same as submitters.

Name of the owner, if any, of the cryptosystem

Same as submitters.

Backup contact telephone and address

Gerardo Pelosi (phone: +39 02 2399 3476), Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via G. Ponzio 34/5, I-20133, Milano, Italy.

Signature of the submitter[1]

×

[1] See also printed version of "Statement by Each Submitter".

Contents

Foreword **9**

1 Complete written specification **11**

1.1 Preliminaries 11

1.1.1 Finite fields and circulant matrix algebra 11

1.1.2 Quasi-cyclic low-density parity-check codes and their efficient decoding 15

1.1.3 Classic code-based cryptosystems 19

1.2 QC-LDPC code-based McEliece and Niederreiter cryptosystems 21

1.3 Description of LEDAcrypt Key Encapsulation Methods 27

1.3.1 LEDAcrypt-KEM: encapsulation and decapsulation algorithms 27

1.3.2 LEDAcrypt-KEM-CPA: encapsulation and decapsulation algorithms 30

1.4 Description of LEDAcrypt Public Key Cryptosystem 32

1.4.1 LEDAcrypt-PKC: encryption and decryption transformations 32

1.4.2 Constant weight encoding/decoding 36

2 Security analysis of LEDAcrypt **37**

2.1 Quantitative security level goals 37

2.2 Hardness of the underlying problem 39

2.3 Attacks based on information set decoding 40

2.3.1 Key recovery attacks via codeword finding 41

2.4 Attacks based on weak keys 43

2.5 Attacks based on exhaustive key search 43

2.6 Attacks based on the receiver's reactions 44

2.7 Side channel attacks 45

3 Decoders for LEDAcrypt **46**

3.1 Preliminary analyses 50

3.2 Residual distribution of the mis-matches between the sought error vector and the input estimated one after the execution of a single in-place iteration function 53

3.3 Residual distribution of the mis-matches between the sought error vector and the input estimated one after the execution of a single out-of-place iteration function . . 60

3.4 Maximum number of mismatches corrected with certainty by executing either an in-place or an out-of-place iteration function 64

3.4.1 Efficient computation of Γ and $\mu(t)$ 66

3.5 Efficient choice of out-of place decoder thresholds for the numerically simulated DFR of LEDAcrypt-KEM-CPA 69

4 LEDAcrypt code parameters from a security standpoint **72**

4.1 Parameters for LEDAcrypt-KEM and LEDAcrypt-PKC 76

4.2 Parameters for LEDAcrypt-KEM-CPA 82

5 Optimized LEDAcrypt Implementation **84**

5.1 Selection of the LEDAdecoder strategy 85

5.2 Binary Polynomial Multiplications 88

5.2.1 Selection of Polynomial Multiplication Algorithms 90

5.3 Optimized Out-Of-Place Iteration Function 93

5.4 Binary Polynomial Inversion 96

5.4.1 Schoolbook Euclid's Algorithm 96

5.4.2 Optimized Polynomial Inversions 98

5.4.3 Experimental Evaluation 105

5.5 Parameter Tuning for LEDAcrypt-KEM-CPA 108

6 LEDAcrypt performance evaluation and recommended parameters **111**

6.1 Key-lengths, ciphertext and transmitted data sizes 111

6.2 Performance evaluation considering execution times in milliseconds 115

6.3 Performance evaluation considering execution times in clock cycles and their combination with the size of transmitted data 118

6.4 Recommended parameters 121

6.4.1 Recommended parameters for LEDAcrypt-KEM-CPA 121
6.4.2 Recommended parameters for the IND-CCA2 LEDAcrypt-KEM and LEDAcrypt-PKC systems 122

Bibliography **124**

A Deriving the bit-flipping probabilities for the in-place iteration function of the LEDAdecoder **130**

Acronyms

ASIC	Application-Specific Integrated Circuit
BF	Bit Flipping
BQP	Bounded-error Quantum Polynomial
CFP	Codeword Finding Problem
DFR	Decoding Failure Rate
DRBG	Deterministic Random Bit Generator
GE	Gate Equivalent
GRS	Generalized Reed-Solomon
IND-CCA2	Indistinguishability Under Adaptive Chosen Ciphertext Attack
IND-CPA	Indistinguishability Under Chosen Plaintext Attack
ISD	Information Set Decoding
KDF	Key Derivation Function
KEM	Key Encapsulation Module
KEM+DEM	Key Encapsulation Module + Data Encapsulation Mechanism
KI	Kobara-Imai
LDPC	Low-Density Parity-Check
NP	Nondeterministic-Polynomial
OW-CPA	One Wayness against Chosen Plaintext Attack
PFS	Perfect Forward Secrecy
PKE	Public-Key Encryption
PRNG	Pseudo Random Number Generator
QC	Quasi-Cyclic
QC-LDPC	Quasi-Cyclic Low-Density Parity-Check

ROM Random Oracle Model

SDP Syndrome Decoding Problem

TM Turing Machine

TRNG True Random Number Generator

XOF Extensible Output Function

Foreword

This document provides a complete and detailed specification of the post-quantum cryptographic primitives named LEDAcrypt (Low-dEnsity parity-check coDe-bAsed cryptographic systems), submitted to the 2nd round of the NIST post-quantum contest [1] and resulting from the merger between the LEDAkem and LEDApkc proposals submitted to the 1st round of the contest [54]. LEDAcrypt provides a set of cryptographic primitives based on binary linear error-correcting codes. In particular, the following cryptographic primitives are proposed:

i. An IND-CCA2 key encapsulation method, named LEDAcrypt-KEM.

ii. An IND-CCA2 public key encryption scheme, named LEDAcrypt-PKC.

iii. An IND-CPA key encapsulation method optimized for use in an ephemeral key scenario, while providing resistance against accidental key reuse, named LEDAcrypt-KEM-CPA.

LEDAcrypt exploits the advantages of relying on Quasi-Cyclic Low-Density Parity-Check (QC-LDPC) codes to provide high decoding speeds and compact key pairs [6]. In particular:

i. We present a theoretical model to provide sound upper bounds to the probability of a decryption failure of the underlying Niederreiter/McEliece encryption schemes, when employing QC-LDPC codes as the code family of choice.

ii. We employ the constructions provided by [39, 45] to attain provable IND-CCA2 guarantees in the proposed LEDAcrypt-KEM and LEDAcrypt-PKC primitives, respectively.

iii. We propose a method to allow the use of the classic definition of Decoding Failure Rate (DFR) from coding theory as a proxy for the notion of δ-correctness required by the construction in [39] for IND-CCA2 KEMs, under the assumption of a preimage resistant extensible output function (XOF) being available.

iv. We provide three choices for the rate of the underlying QC-LDPC codes, and two choices of DFR for each security category specified by NIST. We highlight the engineering tradeoffs between key size, overall transmitted data, and execution time.

v. We employ a reproducible parameter design procedure relying on finite-regime estimates of the computational effort required to carry out attacks against LEDAcrypt and jointly optimize the parameters for security and DFR.

vi. We provide a **constant time** software implementation for the proposed LEDAcrypt-KEM and LEDAcrypt-PKC, optimized for the Intel Haswell Instruction Set Architecture, which exploits the presence of the Intel AVX2 instruction set extension.

The main innovations with respect to the second round specification are as follows:

- We propose, and analyze from the DFR standpoint two decoding strategies, namely in-place and out-of-place Bit Flipping (BF) decoding. We provide closed form worst case estimates for the DFR of both of them, under consolidated assumptions from the literature on iteratively decoded codes. Our worst-case estimates for the DFR allow us to match the requirements for an IND-CCA2 construction with the choice of either one of the decoding strategies. Our models for the worst case behaviour of the DFR consider a finite number of decoder iterations, enabling the constant time implementation of the decoder. In particular, we choose to consider BF decoders with two iterations for performance reasons.
- We provide an IND-CCA2 construction for LEDAcrypt-KEM which allows employing the common DFR notion from the coding theory lexicon, matching it with the requirement of the δ-correctness from the [39] proofs, and we provide a proof of its IND-CCA2 guarantees, under the (provided) bounded DFR of the employed QC-LDPC code.
- A recent attack [2] highlighted that the product structure of the secret key matrix, HQ, can be exploited to lower the security margin provided by LEDAcrypt. We adopt a conservative solution, that is consider $Q = I$, removing the effects of the product structure altogether. A brief justification of this rationale is provided in Section 2.4, from which the current specification assumes that the secret key is a randomly drawn block circulant matrix H. While making this choice reduces the speed advantages coming from the performance-oriented criterion of using of a product-based secret key, we consider the importance of the security of the scheme paramount with respect to the speed gains.
- New parameterizations for LEDAcrypt instances considering the case $Q = I$ are provided. In addition to the previous specification, we also provide parameters for all the code rates $(\frac{1}{2}, \frac{2}{3}, \frac{3}{4})$ also for the case of IND-CCA2 primitives. The parameters are obtained through a joint optimization of the circulant matrix size p, the error weight t and the column weight of the parity-check matrix v. Our previous strategy relied only on enlarging the circulant matrix size to attain the desired DFR, starting from IND-CPA secure parameters. Our new approach allows further reductions of the key size.
- We performed an exploration of the design space for the implementation of the low level primitives involved in the computation of the LEDAcrypt ciphers, namely the binary polynomial multiplication, the modular binary polynomial inversion and the BF decoder. We motivate the implementation choices made through benchmarking of the different alternatives, providing a *constant time* implementation of LEDAcrypt-KEM and LEDAcrypt-PKC, and a speed-oriented, variable time implementation of LEDAcrypt-KEM-CPA for ephemeral key use.
- Despite forgoing the speedup provided by the use of the product structure, the improvements in the cryptosystem engineering allowed us to improve by 20% to 100% (depending on the NIST category) the computation speed of the LEDAcrypt-KEM-CPA primitive with respect to the one reported in the NIST Second PQC Standardization Conference. Furthermore, despite the additional overhead imposed by the constant time implementation requirement, we either match, or improve by 30% the computation speed of LEDAcrypt-KEM with respect to the results reported in the aforementioned venue. Analyzing the obtained results, we provide recommendations on the choices for the code rate to be made, depending on which figure of merit (computation speed, bandwidth, or a combined metric) is chosen.

Chapter 1

Complete written specification

This chapter recalls some basic concepts and provides a functional description of the LEDAcrypt primitives.

1.1 Preliminaries

Let us introduce a set of background notions and nomenclature concerning finite fields and circulant matrix algebra, binary error correcting codes, and code-based cryptosystems, which will be employed in LEDAcrypt.

1.1.1 Finite fields and circulant matrix algebra

A $v \times v$ circulant matrix A is a matrix having the following form

$$A = \begin{bmatrix} a_0 & a_1 & a_2 & \cdots & a_{v-1} \\ a_{v-1} & a_0 & a_1 & \cdots & a_{v-2} \\ a_{v-2} & a_{v-1} & a_0 & \cdots & a_{v-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_1 & a_2 & a_3 & \cdots & a_0 \end{bmatrix}. \quad (1.1)$$

According to its definition, any circulant matrix has a constant row and column weight, i.e., it is *regular*, since all its rows and columns are cyclic shifts of the first row and column, respectively.

A Quasi-Cyclic (QC) matrix is a matrix having the following form

$$B = \begin{bmatrix} B_{0,0} & B_{0,1} & \dots & B_{0,w-1} \\ B_{1,0} & B_{1,1} & \dots & B_{1,w-1} \\ \vdots & \vdots & \ddots & \vdots \\ B_{z-1,0} & B_{z-1,1} & \dots & B_{z-1,w-1} \end{bmatrix}, \quad (1.2)$$

where w and z are two positive integers and each block $B_{i,j}$ is a circulant matrix.

The set of $v \times v$ binary circulant matrices forms an algebraic ring under the standard operations of modulo-2 matrix addition and multiplication. The zero element is the all-zero matrix, and the

identity element is the $v \times v$ identity matrix. The algebra of the polynomial ring $\mathbb{F}_2[x]/\langle x^v+1\rangle$ is isomorphic to the ring of $v \times v$ circulant matrices over $\mathbb{F}_2$ with the following map

$$A \leftrightarrow a(x) = \sum_{i=0}^{v-1} a_i x^i. \tag{1.3}$$

According to (1.3), any binary circulant matrix is associated to a polynomial in the variable x having coefficients over $\mathbb{F}_2$ that coincide with the entries in the first row of the matrix

$$a(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_{v-1} x^{v-1}. \tag{1.4}$$

In addition, according to (1.3), the all-zero circulant matrix corresponds to the null polynomial and the identity matrix to the unitary polynomial.

The ring of polynomials $\mathbb{F}_2[x]/\langle x^v+1\rangle$ includes elements that are zero divisors: such elements are mapped onto singular circulant matrices over $\mathbb{F}_2$. Avoiding such matrices is important in the computation of the LEDAcrypt primitives, as non-singular $v \times v$ circulant matrices are required. However, a proper selection of the size v of a circulant matrix allows to easily generate invertible instances of it as described next.

Polynomial inversion in a finite field In order to provide efficient execution for the LEDAcrypt primitives, it is crucial to be able to efficiently check invertibility of a binary circulant matrix, and to generate a non-singular circulant matrix efficiently. To this end, we exploit the isomorphism (1.3) between $p \times p$ binary circulant matrices and polynomials in $\mathbb{F}_2[x]/\langle x^p+1\rangle$, turning the problem into providing an efficient criterion for the invertibility of an element of $\mathbb{F}_2[x]/\langle x^p+1\rangle$ and describing an efficient way to generate such invertible polynomials. In the following, we recall some facts from finite field theory, and we derive a necessary and sufficient condition for the invertibility of an element of $\mathbb{F}_2[x]/\langle x^p+1\rangle$, provided p is chosen according to the described criterion. Let $\mathbb{F}_{q^m}$ be the finite field of cardinality q^m, with q a prime power and m a positive integer; given an element $\alpha \in \mathbb{F}_{q^m}$, the following propositions hold [75]:

(i) The minimal polynomial of α with respect to $\mathbb{F}_q$, i.e., the nonzero monic polynomial $f(x) \in \mathbb{F}_q[x]$ of the least degree such that $f(\alpha) = 0$, always exists, it is unique, and it is also irreducible over $\mathbb{F}_q$.

(ii) If a monic irreducible polynomial $g(x) \in \mathbb{F}_q[x]$ has $\alpha \in \mathbb{F}_{q^m}$ as a root, then it is the minimal polynomial of α with respect to $\mathbb{F}_q$.

Definition 1.1.1 Let n be a positive integer and q a prime power such that $\mathsf{gcd}(n, q) = 1$, which means that n and q are coprime. A cyclotomic coset of $q \bmod n$ containing the value $a \in \mathbb{Z}_n$ is defined as

$$C_a = \{aq^j \bmod n : j = 0, 1, \ldots\}.$$

A subset $\{a_1, \ldots, a_s\} \subseteq \mathbb{Z}_n$ is named as a *complete set of representatives* of cyclotomic cosets of $q \bmod n$ if $\forall\, i \neq j\ C_{a_i} \cap C_{a_j} = ?$ and $\bigcup_{j}^{s} C_{a_j} = \mathbb{Z}_n$.

It is worth noting that the previous definition allows to easily infer that two cyclotomic cosets are either equal or disjoint. Indeed, given two cyclotomic cosets C_{a_1} and C_{a_2}, with $a_1 \neq a_2 \bmod n$, if

$C_{a_1} \cap C_{a_2} \neq ?$, two positive integers j and k such that $a_1 q^j = a_2 q^k \bmod n$ should exist. Assuming (without loss of generality) that $k \geq j$, the condition $\mathsf{gcd}(n, q) = 1$ would ensure the existence of the multiplicative inverse of q and consequentially that $a_1 = a_2 q^{k-j} \bmod n$, which in turn would imply that the cyclotomic coset including a_1 is a subset of the coset including a_2, i.e., $C_{a_1} \subseteq C_{a_2}$. However, as the previous equality can be rewritten as $a_2 = a_1 (q^{-1})^{k-j} \bmod n$, it would also imply $C_{a_2} \subseteq C_{a_1}$, leading to conclude that $a_1 = a_2 \bmod n$, which is a contradiction of the initial assumption about them being different.

Two notable theorems that make use of the cyclotomic coset definition to determine the minimal polynomials of every element in a finite field can be stated as follows [75].

Theorem 1.1.1 Let α be a primitive element of F_{q^m}, the minimal polynomial of α^i in $\mathsf{F}_q[x]$ is $g^{(i)}(x) = \prod_{j \in C_i} (x - \alpha^j)$, where C_i is the unique cyclotomic coset of $q \bmod q^m - 1$ containing i.

Theorem 1.1.2 Given a positive integer n and a prime power q, with $\mathsf{gcd}(n, q) = 1$, let m be a positive integer such that $n \,|\, (q^m - 1)$. Let α be a primitive element of F_{q^m} and let $g^{(i)}(x) \in \mathsf{F}_q[x]$ be the minimal polynomial of $\alpha^i \in \mathsf{F}_{q^m}$. Denoting as $\{a_1, \ldots, a_s\} \subseteq \mathsf{Z}_n$ a complete set of representatives of cyclotomic cosets of $q \bmod n$, the polynomial $x^n - 1 \in \mathsf{F}_q[x]$ can be factorized as the product of monic irreducible polynomials over F_q, i.e.,

$$x^n - 1 = \prod_{i=1}^{s} g^{\left(\frac{(q^m-1)a_i}{n}\right)}(x).$$

Corollary 1.1.1 Given a positive integer n and a prime power q, with $\mathsf{gcd}(n, q) = 1$, the number of monic irreducible factors of $x^n - 1 \in \mathsf{F}_q[x]$ is equal to the number of cyclotomic cosets of q mod n.

From the previous propositions on the properties of finite fields, it is possible to derive the following results.

Corollary 1.1.2 Given an odd prime number p, if 2 is a primitive element in the finite field Z_p then the irreducible (non trivial) polynomials being a factor of $x^p + 1 \in \mathsf{F}_2[x]$ are $x + 1$ and $\Phi(x) = x^{p-1} + x^{p-2} + \cdots + x + 1$.

Proof. Considering the ring of polynomials with binary coefficients $\mathsf{F}_2[x]$ and picking a positive integer n as an odd prime number (i.e., $n = p$), Corollary 1.1.1 ensures that the number of factors of $x^p + 1 \in \mathsf{F}_2[x]$ equals the number of cyclotomic cosets of 2 mod p.
If 2 is a primitive element of Z_p, its order, $\mathrm{ord}_p(2)$, is equal to the order of the (cyclic) multiplicative group of the field, i.e., $\mathrm{ord}_p(2) = |\,(\mathsf{Z}_p \setminus \{0\}, \cdot)\,| = p - 1$; thus, the said cyclotomic cosets can be listed as: $C_0 = \{0 \cdot 2^j \bmod p : j = 0, 1, \ldots\} = \{0\}$ and $C_1 = \{1 \cdot 2^j \bmod p : j = 0, 1, \ldots\} = \mathsf{Z}_p \setminus \{0\}$.
The polynomial $x^p - 1 \in \mathsf{F}_2[x]$ admits $\alpha = 1$ as a root, therefore its two (non trivial) factors can be listed as: $x + 1$ and $\frac{x^p+1}{x+1} = x^{p-1} + x^{p-2} + \cdots + x + 1$. ■

Theorem 1.1.3 (Invertible elements in $\mathsf{F}_2[x]/\langle x^p + 1\rangle$) Let p be a prime number such that $\mathrm{ord}_p(2) = p - 1$ (i.e., 2 is primitive element in the field Z_p). Let $g(x)$ be a binary polynomial in $\mathsf{F}_2[x]/\langle x^p + 1\rangle$, with $\mathsf{deg}(g(x)) > 0$.
$g(x)$ has a multiplicative inverse in $\mathsf{F}_2[x]/\langle x^p + 1\rangle$ if and only if it contains an odd number of terms and $g(x) \neq \Phi(x)$, with $\Phi(x) = x^{p-1} + x^{p-2} + \cdots + x + 1$.

Proof. If $g(x) \in \mathbb{F}_2[x]/\langle x^p+1\rangle$ contains an odd number of terms and $g(x) \neq \Phi(x)$, to prove it is invertible modulo x^p+1 we need to consider that $\mathsf{gcd}(g(x), x^p+1) = \mathsf{gcd}(g(x), (x+1)\Phi(x))$.
It is easy to observe that $x+1$ does not divide $g(x)$, i.e., $(x+1) \nmid g(x)$, as $g(1)=1$, thus they are coprime. Considering $\Phi(x)$, we know by hypothesis that $\mathrm{ord}_p(2) = p-1$, therefore $\Phi(x)$ is irreducible over $\mathbb{F}_2[x]$ (see Corollary 1.1.2), which excludes that $g(x) \mid \Phi(x)$.
To the end of proving that $g(x)$ and $\Phi(x)$ are coprime, it has to hold that $\Phi(x) \nmid g(x)$. To this end assume, by contradiction, that $g(x)h(x) = \Phi(x)$ for a proper choice of $h(x) \in \mathbb{F}_2[x]$. The previous equality entails that $\mathsf{deg}(g(x)) + \mathsf{deg}(h(x)) = p-1$, while $\mathsf{deg}(g(x)) \leq p-1$, which in turn leaves $\mathsf{deg}(h(x)) = 0$ as the only option, leading to conclude $h(x)=0$ or $h(x)=1$. In case $h(x)=0$, the equality $g(x)\cdot 0 = x^{p-1}+x^{p-2}+\cdots+x+1$ is false, while in case $h(x)=1$, the equality $g(x)\cdot 1 = \Phi(x)$ contradicts the hypothesis. Since we proved that $g(x) \nmid \Phi(x)$ and $\Phi(x) \nmid g(x)$, $g(x) \neq \Phi(x)$ by hypothesis, we can infer that they are coprime.
Finally, being $\mathsf{gcd}(g(x), x^p+1) = \mathsf{gcd}(g(x), (x+1)\Phi(x)) = 1$ we conclude that $g(x)$ is invertible.
To prove the other implication of the theorem, if $g(x) \in \mathbb{F}_2[x]/\langle x^p+1\rangle$, with $\mathsf{deg}(g(x)) > 0$, is invertible we need to derive that $g(x)$ must have an odd number of terms and be different from $\Phi(x)$. Being $g(x)$ invertible, this means that $\mathsf{gcd}(g(x), x^p+1) = \mathsf{gcd}(g(x), (x+1)\Phi(x)) = 1$, which in turn means that $\mathsf{gcd}(g(x), x+1) = 1$ and $\mathsf{gcd}(g(x), \Phi(x)) = 1$ that guarantees that $g(x) \neq \Phi(x)$ and that $g(1) = 1$. Willing to prove that $g(x)$ must have an odd number of terms, assume, by contradiction, it has an even number of terms. Regardless of which terms are contained in $g(x)$ this means that it admits 1 as a root, which contradicts the premise. ■

Invertibility of square quasi cyclic binary matrices In the construction of the LEDAcrypt primitives we will need to generate QC binary square matrices that must be invertible. To remove the need to compute the matrix determinant, as it is computationally demanding, we prove and employ the following theorem. We denote as $\mathrm{perm}(\cdot)$ the permanent of a matrix, and as $w(\cdot)$ the number of the nonzero coefficients of a polynomial, a quantity also known as its weight.

Theorem 1.1.4 Let $p > 2$ be a prime such that $\mathrm{ord}_p(2) = p-1$ and Q is an $n_0 \times n_0$ matrix of elements of $\mathbb{F}_2[x]/\langle x^p+1\rangle$; if $\mathrm{perm}(w(Q))$ is odd and $\mathrm{perm}(w(Q)) < p$, then Q is non-singular.

Proof. Since each block Q_{ij} is isomorphic to a polynomial $q_{ij}(x) \in \mathbb{F}_2[x]/\langle x^p+1\rangle$, the determinant of the matrix Q is represented as an element of $\mathbb{F}_2[x]/\langle x^p+1\rangle$ as well. Let us denote by $d(x)$ the polynomial associated to the determinant. If the inverse of $d(x)$ exists, then Q is non-singular. According to Theorem 1.1.3, showing that $d(x)$ has odd weight and $d(x) \neq \Phi(x) = x^{p-1}+x^{p-2}+\cdots+1$ is enough to guarantee that it is invertible. In general, when we are considering two polynomials $a(x)$ and $b(x)$, with $w(a(x)) = w_a$ and $w(b(x)) = w_b$, the following statements hold:

i. $w(a(x)b(x)) = w_a w_b - 2c_1$, where c_1 is the number of cancellations of pairs of monomials with the same exponent resulting from multiplication;

ii. $w(a(x)+b(x)) = w_a + w_b - 2c_2$, where c_2 is the number of cancellations due to monomials with the same exponent appearing in both polynomials.

The determinant $d(x)$ is obtained through multiplications and sums of the elements $q_{ij}(x)$ and, in case of no cancellations, has weight equal to $\mathrm{perm}(w(Q))$. If some cancellations occur, considering statements i) and ii) above, we have that $w(d(x)) = \mathrm{perm}(w(Q)) - 2c$, where c is the overall number of cancellations. So, even when cancellations occur, $d(x)$ has odd weight only if $\mathrm{perm}(w(Q))$ is

odd. In addition, the condition $\text{perm}(w(Q)) < p$ guarantees that $d(x) \neq \Phi(x)$, since $w(\Phi(x)) = p$. ■

With this result, we can guarantee that a QC matrix is non-singular, provided that the weights of its circulant blocks are chosen properly.

1.1.2 Quasi-cyclic low-density parity-check codes and their efficient decoding

Binary error correcting codes rely on a redundant representation of information in the form of binary strings to be able to detect and correct accidental bit errors which may happen during transmission or storage. We employ binary codes acting on a finite binary sequence at once, known as the information word, which are known as block codes. We will refer to them simply as binary codes in the following.

In this setting, let F_2 be the binary finite field with the addition and multiplication operations that correspond to the usual exclusive-or and logical product between two Boolean values. Let F_2^k denote the k-dimensional vector space defined on F_2. A binary code, denoted as $\mathsf{C}(n,k)$, is defined as a bijective map $\mathsf{C}(n,k) : \mathsf{F}_2^k \rightarrow \mathsf{F}_2^n$, $n,k \in \mathsf{N}$, $0 < k < n$, between any binary k-tuple (i.e., an information word) and a binary n-tuple (denoted as codeword). The value n is known as the length of the code, while k is denoted as its dimension.
Encoding through $\mathsf{C}(n,k)$ means converting an information word $u \in \mathsf{F}_2^k$ into its corresponding codeword $c \in \mathsf{F}_2^n$. Given a codeword $\hat{c}$ corrupted by an error vector $e \in \mathsf{F}_2^n$ with Hamming weight $t > 0$ ($\hat{c} = c + e$), decoding aims to recover both the value of the information word u and the value of the error vector e. A code is said to be t-error correcting if, for any value of e, given $\hat{c}$ there is a decoding procedure to retrieve both the error vector e and the original information word u.

Definition 1.1.2 (Linear Code) The code $\mathsf{C}(n,k)$ is linear if and only if the set of its 2^k codewords is a k-dimensional subspace of the vector space F_2^n.

A property of linear block codes that follows from Definition 1.1.2 is that the sum modulo 2, i.e., the component wise exclusive-or, of two codewords is also a codeword.

Definition 1.1.3 (Minimum distance) Given a linear binary code $\mathsf{C}(n,k)$, the minimum distance of $\mathsf{C}(n,k)$ is the minimum Hamming distance among all the distances which can be computed between a pair of its codewords.

If the code is linear, its minimum distance coincides with the minimum Hamming weight of its nonzero codewords.

Definition 1.1.4 (Encoding) Given $\mathsf{C}(n,k)$, a linear error correcting code, and $\Gamma \subset \mathsf{F}_2^n$ the vector subspace containing its 2^k codewords, it is possible to represent it choosing k linearly independent codewords $\{g_0, g_1, \ldots g_{k-1}\} \in \mathsf{F}_2^n$ to form a basis of Γ. Any codeword $c = [c_0, c_1, \ldots, c_{n-1}]$ can be expressed as a linear combination of the vectors of the basis

$$c = u_0 g_0 + \mathrm{u}_1 g_1 + \ldots + u_{k-1} g_{k-1}, \tag{1.5}$$

where the binary coefficients u_i can be thought as the elements of an information vector $u = [u_0, u_1, \ldots, u_{k-1}]$, which the code maps into c. We then say that u is encoded into c.

Equation (1.5) can be rewritten as $c = uG$, where G is a $k \times n$ binary matrix known as the generator matrix of the code $\mathsf{C}(n, k)$, i.e.,

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix}.$$

Since any set of k linearly independent codewords can be used to form G, a code can be represented by different generator matrices. Among the possible generator matrices for a linear code, one known as systematic can always be derived.

Definition 1.1.5 (Systematic Encoding) A linear error correcting code $\mathsf{C}(n, k)$ is said to have systematic encoding, or to be systematic in short, if each one of its codewords contains the information vector it is associated to.

A conventional way to express a systematic code is the one where each n-bit codeword, c, is obtained by appending $r = n - k$ redundancy bits $(c_k, c_{k+1}, \ldots, c_{n-1})$ to its corresponding k-bit information word (i.e., $c_0, c_1, \ldots, c_{k-1}$, with $c_i = u_i$, $0 \leq i < k$): $c = [u_0, u_1, \ldots, u_{k-1} | c_k, c_{k+1}, \ldots, c_{n-1}]$. It follows that the associated $k \times n$ generator matrix G can be written as $G = [I_k | P]$, where I_k denotes the $k \times k$ identity matrix and P is a $k \times r$ binary matrix.
Let us consider the set of all n-bit vectors in $\mathbb{F}_2^n$ that are orthogonal to any codeword of the code subspace Γ, known as its orthogonal complement $\Gamma^{\perp}$. Its dimension is $\dim\left(\Gamma^{\perp}\right) = n - \dim(\Gamma) = n - k = r$. A basis of $\Gamma^{\perp}$ is readily obtained choosing r linearly independent vector in $\Gamma^{\perp}$ as

$$H = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_{r-1} \end{bmatrix}.$$

The $r \times n$ matrix H is known as a parity-check matrix of the code $\mathsf{C}(n, k)$, while, for any n-bit vector $x \in \mathbb{F}_2^n$, the $r \times 1$ vector $s = Hx^T$, where ^{T} denotes transposition, is known as the syndrome of x through H. Given that H is a basis of $\Gamma^{\perp}$, every codeword $c \in \Gamma$ satisfies the equality $Hc^T = 0_{r \times 1}$ where $0_{r \times 1}$ is the $r \times 1$ zero vector, i.e., a codeword belonging to $\mathsf{C}(n, k)$ has a null syndrome through H.

It is easy to show that the generator matrix G and the parity-check matrix H are two equivalent descriptions of a linear code. Indeed, we have that $Hc^T = HG^T u^T = 0_{r \times 1}$, $\forall\, u \in \mathbb{F}_2^k$, yielding in turn that $HG^T = 0_{r \times k}$. Exploiting the aforementioned relation, it is possible to derive H from G and vice versa. Let us consider, for the sake of clarity, the case of a systematic code $\mathsf{C}(n, k)$ with $G = [I_k | P]$. It is possible to obtain the corresponding parity-check matrix H as $\left[P^T | I_r\right]$, which satisfies $HG^T = P^T + P^T = 0_{r \times k}$. Finally, considering a generic parity-check matrix $H = [A|B]$, with A an $r \times k$ matrix and B an $r \times r$ non-singular matrix, a systematic generator matrix of the corresponding code is computed as $G = \left[I_k | \left(B^{-1}A\right)^T\right]$, being B^{-1} the inverse of matrix B.

A QC code is defined as a linear block code $\mathsf{C}(n, k)$ having information word size $k = pk_0$ and codeword size $n = pn_0$, where n_0 is denoted as *basic block length* of the code and each cyclic shift of a codeword by n_0 symbols results in another valid codeword [72].

LEDAcrypt hinges on a QC code $\mathtt{C}(pn_0, pk_0)$ having the generator and parity-check matrices composed by $p \times p$ circulant sub-matrices (blocks).

A Low-Density Parity-Check (LDPC) code $\mathtt{C}\,(n, k)$ is a special type of linear block code characterized by a sparse parity-check matrix H. In particular, the Hamming weight of a column of H, denoted as d_v, is much smaller than its length r and increases sub-linearly with it. In terms of error correction capability, LDPC codes having a non-constant weight for either the rows or the columns of H, hence known as irregular LDPC codes, were proven to approach the channel capacity [50]. Considering the parity-check matrix H of an LDPC code as the incidence matrix of a graph, such a graph is known as Tanner graph, and it has been shown that keeping the number of short cycles as small as possible in such a graph is beneficial to the error correction performance of the code.

The peculiar form of LDPC codes allows to devise an efficient decoding procedure, provided their parity-check matrix H is known, via algorithms known as BF decoders [33]. Indeed, BF algorithms perform decoding with a fixed-point procedure which exploits the form of H to iteratively deduce which bits of an error-affected codeword should be flipped in order to obtain a zero-valued syndrome for it. If the fixed-point procedure converges within a desired amount of iterations to a zero-valued syndrome, the decoding action is deemed successful.

The rationale of BF decoders is in considering the parity-check matrix H as the description of a set of r equations in the codeword bits yielding the syndrome bits as their results. Such equations are known as parity-check equations, or parity checks, in short. In this context, the one-valued coefficients of the i-th column of a parity matrix H can be thought of as the indicators of which parity checks of the code are involving the i-th bit of the received codeword. The result of each one of the said parity checks is a bit in the syndrome, hence a zero-valued syndrome indicates a set of successful parity checks, and thus a correct codeword. The convergence of the fixed-point decoder is influenced by the number of parity checks in which each codeword element is involved: in particular, being involved in a small number of parity checks speeds up the convergence.

An LDPC code may also be a QC code, expressed with a QC parity-check or generator matrix, hence being named a QC-LDPC code, which is indeed the case of the codes employed in LEDAcrypt.

An efficient BF decoding procedure for QC-LDPC codes can be devised relying on the number of unsatisfied parity checks to which a codeword bit concurs as an estimate of it being affected by an error. We describe such a procedure in Algorithm 1, where the sparse and QC nature of the matrix H is explicitly exploited.

To this end H is represented as $r_0 \times n_0$ sparse $p \times p$ circulant blocks, and only the positions of the first column of each block are memorized in `Hsparse`. Algorithm 1 receives, alongside `Hsparse`, the error-affected codeword to be corrected x, its syndrome computed as $s = Hx^T$, and performs the fixed-point decoding procedure for a maximum of `imax` iterations. The algorithm outputs its best estimate for the correct codeword c and a Boolean variable `decodeOk` reporting the success, or not, of the decoding procedure. The procedure iterates at fixed-point (loop at lines 4–20) the decoding procedure, which starts by counting how many unsatisfied parity checks a codeword bit is involved into (lines 5–12). Such a value is obtained considering which are the asserted bits in a given column of H, taking care of accounting for its sparse representation, and the cyclic nature of its blocks (line 10). Whenever a bit in the i-th column and `assertedHbitPos`-th row of H is set, it is pointing to the fact that the i-th bit of the codeword is involved in the `assertedHbitPos`-th parity-check equation. Thus, if the `assertedHbitPos`-th bit of the syndrome is unsatisfied, i.e., equal to 1, the number of unsatisfied parity checks of the i-th bit is incremented (lines 11–12).

Algorithm 1: BF decoding

Input: x: QC-LDPC error-affected codeword as a $1 \times pn_0$ binary vector.
s: QC-LDPC syndrome. It is a $pr_0 \times 1$ binary vector obtained as $s = Hx^T$.
Hsparse: sparse version of the parity-check matrix $H_{pr_0 \times pn_0}$, represented as a $d_v \times n_0$ integer matrix containing for each of its n_0 columns, the positions in $\{0, 1, \ldots, pr_0 - 1\}$ of the asserted binary coefficients in the first column of each circulant block in the sequence of n_0 submatrices in $H = [H_0 \mid H_1 \mid \ldots \mid H_{n_0-1}]$ (any of which with size $p \times p$).

Output: c: error-free $1 \times pn_0$ codeword
decodeOk: Boolean value denoting the successful outcome of the decoding action

Data: imax: maximum number of allowed iterations before reporting a decoding failure

```
1  codeword ← x                                       // bitvector with size pn_0
2  syndrome ← s                                       // bitvector with size pr_0
3  iter ← 0                      // scalar variable denoting the number of iterations
4  repeat
5      iter ← iter + 1
6      unsatParityChecks ← 0_{1×pr_0} // counters of unsatisfied parity checks
7      for i = 0 to n_0 − 1 do
8          for exp = 0 to p − 1 do
9              for j = 0 to d_v − 1 do
10                 assertedHbitPos←(exp+Hsparse[j][i] ) mod p+p·⌊Hsparse[j][i] div p⌋
11                 if syndrome[assertedHbitPos] = 1 then
12                     unsatParityChecks[i · p + exp] ← 1 + unsatParityChecks[i · p + exp]
13     threshold ← ThresholdChoice(iterationCounter, syndrome)
14     for i = 0 to pn_0 − 1 do
15         if unsatParityChecks[i] ≥ threshold then
16             BitToggle(codeword[i])                                // codeword update
17             for j = 0 to d_v − 1 do
18                 assertedHbitPos ← (exp + Hsparse[j][i]) mod p + p · ⌊Hsparse[j][i] div p⌋
19                 BitToggle(syndrome[assertedHbitPos])
20 until syndrome ≠ 0_{1×pr_0}  AND  iter < imax
21 if syndrome = 0_{1×pr_0} then
22     return codeword, true
23 else
24     return codeword, false
```

Once the computation of the number of unsatisfied parity checks per codeword bit is completed, a decision must be taken on which of them are to be flipped, as they are deemed error affected. The choice of the threshold that the number of unsatisfied parity checks should exceed, can be done a-priori from the code parameters, or determined taking into account the iteration reached by the decoder and the current weight of the syndrome.

Thus, the procedure toggles the values of all the codeword bits for which the number of unsatisfied parity checks matches the maximum one (lines 14–16). Once this step is completed, the values of the parity checks should be recomputed according to the new value of the codeword. While this can be accomplished by pre-multiplying the transposed codeword by H, it is more efficient to exploit the knowledge of which bits of the codeword were toggled to change only the parity-check values in the syndrome affected by such toggles. Lines 17–19 of Algorithm 1 update the syndrome according to the aforementioned procedure, i.e., for a given i-th codeword bit being toggled, all the syndrome values corresponding to the positions of the asserted coefficients in the i-th column of H are also toggled. Once either the decoding procedure has reached its intended fixed-point, i.e., the syndrome is a zero-filled vector, or the maximum number of iterations has been reached, Algorithm 1 returns its best estimate for the corrected codeword, together with the outcome of the decoding procedure (lines 21–24).

1.1.3 Classic code-based cryptosystems

The McEliece cryptosystem is a Public-Key Encryption (PKE) scheme proposed by Robert McEliece in 1978 [52] and exploiting the hardness of the problem of decoding a random-like linear block code. In the original proposal, the McEliece cryptosystem used irreducible Goppa codes as secret codes, but its construction can be generalized to other families of codes. The triple of polynomial-time algorithms Π^{McE}=($\mathsf{Keygen}^{\mathsf{McE}}$, $\mathsf{Enc}^{\mathsf{McE}}$, $\mathsf{Dec}^{\mathsf{McE}}$) defining the scheme are as follows:

- The *key-generation* algorithm considers a binary linear block code $\mathsf{C}(n, k)$, with codeword length n, information word length k and outputs a secret key sk^{McE} defined as the generator matrix $G_{k\times n}$ of a code $\mathsf{C}(n, k)$ able to correct $t \geq 1$ or less bit errors, plus a randomly chosen invertible binary matrix $S_{k\times k}$, named scrambling matrix, and a binary permutation matrix $P_{n\times n}$:
$$sk^{\mathsf{McE}} \leftarrow \{S, G, P\}. \quad (1.6)$$
The corresponding public key pk^{McE} is computed as the generator matrix $G'_{k\times n}$ of a permutation-equivalent code with the same size and correction capability of the original code:
$$pk^{\mathsf{McE}} \leftarrow \{G'\}, \text{ with } G' = SGP. \quad (1.7)$$
- The *encryption* algorithm takes as input a public key pk^{McE} and a message composed as a $1 \times k$ binary vector u, and outputs a ciphertext $x_{1\times n} \leftarrow \mathsf{Enc}^{\mathsf{McE}}\left(pk^{\mathsf{McE}}, u\right)$ computed as:
$$x = uG' + e, \quad (1.8)$$
where e is a $1 \times n$ random binary error vector with weight t (i.e., with exactly t asserted bits).
- The *decryption* algorithm takes as input a secret key sk^{McE} and a ciphertext $x_{1\times n}$ and outputs a message $u_{1\times k} \leftarrow \mathsf{Dec}^{\mathsf{McE}}\left(sk^{\mathsf{McE}}, x\right)$ computed as the result of a known error correction decoding

algorithm (Decode) able to remove t errors present in xP^{-1} and subsequently multiplying by the inverse of the matrix S:

$$\mathsf{Decode}(xP^{-1})S^{-1} = \mathsf{Decode}\left((uS)G + (eP^{-1})\right)S^{-1} = (uS)S^{-1} = u. \quad (1.9)$$

As mentioned, in the original McEliece cryptosystem, algebraic code families (namely, Goppa codes) provided with bounded-distance decoders were used. In such a case, since the number of errors correctable by the secret code is t, the correction of the error vector eP^{-1} is ensured by design and the cryptosystem exhibits a zero DFR.

It is also worth noticing that the original McEliece cryptosystem only provides One Wayness against Chosen Plaintext Attack (OW-CPA) guarantees, which means that, given a ciphertext, it is computationally impossible to recover the plaintext without knowing the private key. Suitable conversions of the cryptosystem must be exploited in order to achieve Indistinguishability Under Adaptive Chosen Ciphertext Attack (IND-CCA2), which means that an adversary with access to a decryption oracle (that knows the private key) cannot distinguish whether a string is a decryption of a legitimate given ciphertext or a random valid plaintext message. The decryption oracle cannot be queried on the given ciphertext. When these conversions are used, some constraints on the public code can be relaxed.

The Niederreiter cryptosystem [55] is a code-based cryptosystem exploiting the same trapdoor introduced in the McEliece PKE [52] with an alternative formulation. The Niederreiter PKE employs syndromes and parity-check matrices in place of codewords and generator matrices, as employed by the algorithms in the McEliece PKE. The first proposal of such a scheme used Generalized Reed-Solomon (GRS) codes that were proven to make the whole construction vulnerable [67]. However, when the same family of codes is used, Niederreiter and McEliece cryptosystems exhibit the same cryptographic guarantees [49]. The triplet of polynomial-time algorithms Π^{Nie}=($\mathsf{Keygen}^{\mathsf{Nie}}$, $\mathsf{Enc}^{\mathsf{Nie}}$, $\mathsf{Dec}^{\mathsf{Nie}}$) defining the scheme are as follows:

- The *key-generation* algorithm considers a binary linear block code $\mathsf{C}(n, k)$, with codeword length n, information word length k and outputs a secret key sk^{Nie} defined as the parity-check matrix $H_{r\times n}$ of a code $\mathsf{C}(n, k)$, with $r = n - k$, able to correct $t \geq 1$ or less bit errors, plus a randomly chosen invertible binary matrix $S_{r\times r}$, named scrambling matrix:

$$sk^{\mathsf{Nie}} \leftarrow \{H, S\}. \quad (1.10)$$

  The corresponding public key pk^{Nie} is computed as the parity-check matrix $H'_{r\times n}$ obtained as the product of the two secret matrices and is equivalent to H:

$$pk^{\mathsf{Nie}} \leftarrow \{H'\}, \text{ with } H' = SH. \quad (1.11)$$

  Note that the knowledge of H' is not amenable to be employed with an efficient decoding algorithm as it actually hides the structure of the selected code.

- The *encryption* algorithm takes as input a public key pk^{Nie} and a message composed as a $1\times n$ binary vector e with exactly t asserted bits, and outputs a ciphertext $x_{r\times 1} \leftarrow \mathsf{Enc}^{\mathsf{Nie}}\left(pk^{\mathsf{Nie}}, e\right)$ computed as the syndrome of the original message:

$$x = H'e^T = SHe^T. \quad (1.12)$$

- The *decryption* algorithm takes as input a secret key sk^{Nie} and a ciphertext $x_{r\times 1}$ and outputs a message $e_{1\times n} \leftarrow \mathsf{Dec}^{\mathsf{Nie}}\left(sk^{\mathsf{Nie}}, x\right)$ computed as the result of a known error correction syndrome decoding algorithm ($\mathsf{SynDecoding}$) applied to the vector $S^{-1}x$ and able to recover the original error vector $e_{1\times n}$:

$$e = \mathsf{SynDecoding}(S^{-1}x) = \mathsf{SynDecoding}(He^T). \tag{1.13}$$

  In the original Niederreiter cryptosystem, algebraic code families provided with bounded-distance decoders were considered. In such a case, the syndrome decoding algorithm allows to deterministically recover the original error vector with weight t and the cryptosystem exhibits a zero DFR.

1.2 QC-LDPC code-based McEliece and Niederreiter cryptosystems

In the following we describe the algorithms of both McEliece Π^{McE}=($\mathsf{Keygen}^{\mathsf{McE}}$, $\mathsf{Enc}^{\mathsf{McE}}$, $\mathsf{Dec}^{\mathsf{McE}}$) and Niederreiter Π^{Nie}=($\mathsf{Keygen}^{\mathsf{Nie}}$, $\mathsf{Enc}^{\mathsf{Nie}}$, $\mathsf{Dec}^{\mathsf{Nie}}$) cryptosystems instantiated with QC-LDPC codes.

- The *key-generation* algorithms $\text{KEYGEN}^{\mathsf{Nie}}$ in Fig. 1.1(a) and $\text{KEYGEN}^{\mathsf{McE}}$ in Fig. 1.1(b) consider a QC-LDPC code $\mathsf{C}(n, k)$, with codeword length $n = pn_0$, information word length $k = p(n_0 - 1)$, where $n_0 \in \{2, 3, 4\}$, p is a prime number such that $\mathsf{ord}_p(2) = p - 1$.

  The first step in the key generation procedures is to find two random binary matrices that correspond to the (secret) quasi-cyclic $p \times pn_0$ parity-check matrix H of a QC-LDPC code with the mentioned parameters and a $pn_0 \times pn_0$ quasi-cyclic sparse binary matrix Q. The matrix H is structured as $1\times n_0$ circulant blocks, each of which with size $p\times p$ and with a fixed **odd** number of asserted elements per row/column, denoted as d_v (to guarantee invertibility of each block – see Th. 1.1.3)

$$H = [H_0, H_1, \ldots, H_{n_0-1}],\ w(H_i) = d_v,\ 0 \le i < n_0.$$

  The matrix Q is structured as an $n_0 \times n_0$ block matrix, where each block is a $p \times p$ binary circulant matrix (note that the existence of a multiplicative inverse of any of these blocks is not guaranteed). The weights of each block of Q define a circulant matrix of integers denoted as $w(Q)$ such that the sum of all elements in any row/column of Q amounts to the same value $m = \sum_{i=0}^{n_0-1} m_i$. Explicitly

$$Q = \begin{bmatrix} Q_{0,0} & Q_{0,1} & \cdots & Q_{0,n_0-1} \\ Q_{1,0} & Q_{1,1} & \cdots & Q_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ Q_{n_0-1,0} & Q_{n_0-1,1} & \cdots & Q_{n_0-1,n_0-1} \end{bmatrix} \quad w(Q) = \begin{bmatrix} m_0 & m_1 & \cdots & m_{n_0-1} \\ m_{n_0-1} & m_0 & \cdots & m_{n_0-2} \\ \vdots & \vdots & \ddots & \vdots \\ m_1 & m_{n_0-1} & \cdots & m_0 \end{bmatrix}.$$

  The choice of the weights $m_0, m_1, \cdots, m_{n_0-1}$ is constrained according to Theorem 1.1.4 to ensure the invertibility of Q, even if any individual block may not be invertible.

  In computing the product HQ (see line 3 of both Fig. 1.1(a) and Fig. 1.1(b)), the multiplication of H by the (compatible) full-rank matrix Q ($\mathsf{rank}(Q) = pn_0$) yields a matrix

$$L = HQ \tag{1.14}$$

Algorithm 2: KEYGEN$^{\mathsf{Nie}}$

```
Output: (sk^Nie, pk^Nie)
Data: p>2 prime, ord_p(2)=p−1,
      n_0 ≥ 2
1  seed ← TRNG()
2  {H, Q} ← GenerateHQ(seed)
   // H = [H_0, ..., H_{n0−1}],
   // H_i is a p × p circulant block
   // with wt(H_i) = d_v, 0 ≤ i < n_0
   // Q is an n_0 × n_0 block matrix
   // in accordance to Th. 1.1.4
   // wt([Q_{i,0}, ..., Q_{i,n0−1}])=m, 0≤i<n_0
3  L ← HQ
   // L = [L_0, L_1, ..., L_{n0−1}],
   // L_j = Σ_i H_i Q_ij is a p × p
   circulant block
4  if ∃ 0≤j<n_0 s.t. wt(L_j) ≠ d_v × m then
5      goto 2
6  LInv ← ComputeInverse(L_{n0−1})
7  for i = 0 to n_0 − 2 do
8      M_i ← LInv L_i
   // M_i is a p × p circulant block
9  pk^Nie ← [M_0 | ... | M_{n0−2} | I]
10 sk^Nie ← {H, Q}
   // I is a p × p identity block
11 return (sk^Nie, pk^Nie)
```

(a)

Algorithm 3: KEYGEN$^{\mathsf{McE}}$

```
Output: (sk^McE, pk^McE)
Data: p>2 prime, ord_p(2)=p−1,
      n_0 ≥ 2
1  seed ← TRNG()
2  {H, Q} ← GenerateHQ(seed)
   // H = [H_0, ..., H_{n0−1}],
   // H_i is a p × p circulant block
   // with wt(H_i) = d_v, 0 ≤ i < n_0
   // Q is an n_0 × n_0 block matrix
   // in accordance to Th. 1.1.4
   // wt([Q_{i,0}, ..., Q_{i,n0−1}])=m, 0≤i<n_0
3  L ← HQ
   // L = [L_0, L_1, ..., L_{n0−1}],
   // L_j = Σ_i H_i Q_ij is a p × p
   circulant block
4  if ∃ 0≤j<n_0 s.t. wt(L_j) ≠ d_v × m then
5      goto 2
6  LInv ← ComputeInverse(L_{n0−1})
7  for i = 0 to n_0 − 2 do
8      M_i ← LInv L_i
   // M_i is a p × p circulant block
9  Z ← diag([I, ..., I])
   // p(n_0−1)×p(n_0−1) identity
   // matrix, composed as a
   diagonal
   // block matrix with (n_0 − 1)
   // replicas of I, where I is
   // a p × p identity matrix
10 pk^McE ← [Z | [M_0 | ... | M_{n0−2}]^T]
11 sk^McE ← {H, Q}
12 return (sk^McE, pk^McE)
```

(b)

Figure 1.1: Summary of the key generation algorithms of both Niederreiter (a) and McEliece (b) cryptosystems, instantiated with QC-LDPC codes

with the same rank of H, thus $\mathsf{rank}(L) = p$. This implies that every $p \times p$ block of $L = [L_0, L_1, \ldots, L_{n_0-1}]$ has the same rank (i.e., $\mathsf{rank}(L_i) = p$) and is therefore invertible. As a consequence, in line 6 of both Fig. 1.1(a) and Fig. 1.1(b), there is no need to check whether L_{n_0-1} admits a multiplicative inverse or not.

It is worth noting that the necessary and sufficient conditions in Th. 1.1.3 state that an

invertible binary circulant block matrix must exhibit a weight of any row/column equal to an odd number. Since $L_j = \sum_{i=0}^{n_0-1} H_i Q_{ij}$, the weight of L_j satisfies also the following $wt(L_j) = d_v \times m - 2c$, where the parameter $c \geq 0$ is justified with an argument analogous to the one reported in the proof of Th. 1.1.4. From this, it is easy to conclude that $m = \sum_{i=0}^{n_0-1} m_i$ must also be an odd number.

Lines 4–5 in both Fig. 1.1(a) and Fig. 1.1(b) check that all the blocks of the matrix L have a weight equal to $d_v \times m$, and repeat the generation process until this condition does hold. Such a constraint is imposed as the weight of the blocks of L is a security parameter in LEDAcrypt, and then must be appropriately constrained.

Starting from the multiplicative inverse of L_{n_0-1}, the following matrix can be computed

$$M = L_{n_0-1}^{-1} L = [M_0|M_1|M_2|\ldots|M_{n_0-2}|I_p] = [M_l|I] \tag{1.15}$$

where I denotes the $p \times p$ identity matrix. The matrix M in (1.15) is the parity-check matrix of the public code in systematic form, and can easily be converted into the systematic generator matrix of the same code.

The private key for each of the schemes at hand consists of the two matrices H and Q: $sk^{\mathsf{Nie}} \leftarrow \{H, Q\}$, $sk^{\mathsf{McE}} \leftarrow \{H, Q\}$. Comparing these with the keypair definitions in the original Niederreiter and McEliece schemes, it is worth noting that the original non-singular scrambling matrix is replaced with the multiplicative inverse of the last circulant block resulting from the multiplication of the secret matrices, i.e., $L_{n_0-1}^{-1}$ with $L = HQ = [L_0, \ldots, L_{n_0-1}]$. The secret permutation matrix introduced in the original McEliece scheme has no counterpart in our scheme because we will use a public generator matrix in systematic form and will embed the result into a CCA2 construction, thus making it redundant from a security perspective. The computation of the public key depends on whether the scheme is Niederreiter-based or McEliece-based. Specifically, the computation of the public key values will yield either a parity-check matrix or a generator matrix, both in systematic form, referring to a public code without a computationally efficient decoding/decryption algorithm (see lines 7–9 in both Fig. 1.1(a) and Fig. 1.1(b)).

It is worth noting that in practice there is no need to store a public key including circulant matrices (blocks) that are identity $p \times p$ matrices. Moreover, since both H and Q are formed by sparse circulant matrices (blocks), it is convenient to store each block as the set of integers representing the positions of non-zero elements of the first row of each block. The said set of integers requires at least $\lceil \log_2(p) \rceil$ bits to be stored.

If we consider that the circulant blocks in any block row of Q have overall weight $m = \sum_{i=0}^{n_0-1} m_i$, the size of sk^{Nie} (equiv., sk^{McE}) is $|sk^{\mathsf{Nie}}| = n_0\,(d_v + m)\,\lceil \log_2(p) \rceil$ bits.

We note that, given the fast generation procedure for H and Q, a viable option to reduce the size of the private key *at rest* is to store the seed of a cryptographically secure Pseudo Random Number Generator (PRNG) from which H and Q are generated.

- The *encryption* algorithm $\textsc{Encrypt}^{\mathsf{Nie}}$ in Fig. 1.2(a) takes as input a public key $pk^{\mathsf{Nie}} = [M_0 \mid \ldots M_j \mid \ldots \mid M_{n_0-2}|I]$ and a message composed as a randomly chosen $1 \times pn_0$ binary vector e with exactly t asserted bits, and outputs a ciphertext that is the $p \times 1$ syndrome of the message computed multiplying the sequence of n_0 message blocks of size $1 \times p$ by the QC parity-check matrix of the public code included in the public key

$$s = [M_0 \mid \ldots M_j \mid \ldots \mid M_{n_0-2}|I]e^T.$$

Algorithm 4: ENCRYPT$^{\mathsf{Nie}}$

Input: $e = [e_0, \ldots, e_{n_0-1}]$: plaintext message; randomly chosen $1 \times pn_0$ binary vector, with t asserted bits, where each e_j is a $1 \times p$ vector with $0 \leq j < n_0$.
$pk^{\mathsf{Nie}} = [M_0 \mid \ldots \mid M_{n_0-2} | I]$ public key: sequence of n_0-1 $p \times p$ circulant blocks M_j, with $0 \leq j < n_0-1$, followed by an identity block

Output: s: syndrome; $p \times 1$ binary vector

Data: $p>2$ prime, $\mathsf{ord}_p(2)=p-1$, $n_0 \geq 2$

```
1 s ← 0_{p×1}                  // zero vector
2 for j = 0 to n_0 − 1 do
3     s ← s + M_j e_j^T
4 s ← s + e_{n_0−1}^T
5 return s
```

(a)

Algorithm 5: ENCRYPT$^{\mathsf{McE}}$

Input: $u = [u_0, \ldots, u_{n_0-2}]$: plaintext message; n_0 $1 \times p$ binary vectors;
$e = [e_0, \ldots, e_{n_0-1}]$: error message with $wt(e)=t$; composed as n_0 $1 \times p$ binary vectors;
$pk^{\mathsf{McE}}=[Z \mid [M_0 | \ldots | M_{n_0-2}]^T]$: public key; composed as sequence of n_0-1 $p \times p$ circulant blocks M_j, with $0 \leq j < n_0-1$ juxtaposed to compose a systematic $(n_0-1) \times n_0$ generation matrix with $p \times p$ circulant blocks. Z: a diagonal block matrix with n_0-1 replicas of the identity circular block I

Output: $c = [c_0, \ldots, c_{n_0-1}]$: error affected codeword; $1 \times pn_0$ binary vector, where each c_j is a $1 \times p$ vector with $0 \leq j < n_0$.

Data: $p>2$ prime, $\mathsf{ord}_p(2)=p-1$, $n_0 \geq 2$

```
1 blk ← 0                // p × p zero block
2 for j = 0 to n_0 − 2 do
3     blk ← blk + u_j^T M_j
4 c ← [u_0, u_1, ..., u_{n_0−2}, blk]
5 for j = 0 to n_0 − 1 do
6     c_j ← c_j + e_j
7 return c
```

(b)

Figure 1.2: Encryption algorithm of Niederreiter (a) and McEliece (b) cryptosystems, instantiated with QC-LDPC codes

The *encryption* algorithm ENCRYPT$^{\mathsf{McE}}$ in Fig. 1.2(b) takes three input parameters: a plaintext message composed as a $1 \times p(n_0 - 1)$ binary vector u, an *error vector* composed as a $1 \times pn_0$ binary vector e with exactly t asserted bits (supposed to be uniformly and randomly picked from the set of binary vectors with the same weight), and a public key pk^{McE} structured as a QC generator matrix with size $(n_0 - 1) \times n_0$, i.e., $pk^{\mathsf{McE}} = \left[Z \mid [M_0 \mid \ldots \mid M_{n_0-2}]^T\right]$, where $Z = \mathsf{diag}(I, \ldots, I)$ is a diagonal block matrix composed as $n_0 - 1$ replicas of the identity circulant block I (i.e., a $p \times p$ identity matrix), which coincides with an $(n_0 - 1)p \times (n_0 - 1)p$ identity matrix. The algorithm outputs a ciphertext c that is an error affected $1 \times pn_0$

binary vector composed as follows. The sequence of the first (n_0-1) binary vectors from the plaintext message (each with size $1 \times p$) followed by the binary vector obtained as the sum of products between the corresponding $1 \times p$ blocks in u and the ones in the public key portion $[M_0 \mid \ldots \mid M_{n_0-2}]$, is added to the sequence of n_0 binary vectors of the error vector e

$$c = [e_0|\ldots|e_{n_0-2}|e_{n_0-1}] + \left[u_0 \mid \ldots \mid u_{n_0-2} \mid \sum_{j=0}^{n_0-2} u_j M_j \right].$$

- The *decryption* algorithm $\textsc{Decrypt}^{\mathsf{Nie}}$ in Fig. 1.3(a) takes as input a secret key $sk^{\mathsf{Nie}} = \{H, Q\}$ and a ciphertext that is identified with a $p \times 1$ binary syndrome that is supposed to be computed as $s = [M_0 \mid \ldots \mid M_{n_0-2}|I]e^T = \left(L_{n_0-1}^{-1}[L_0, \ldots, L_{n_0-1}]\right) e^T$. The outcome of the decryption algorithm is the original binary vector e. The initial steps of the algorithm (lines 1-2) consist in taking the secret matrices, re-computing $L = HQ$ and executing the multiplication between the received syndrome s and the last block of $L = [L_0, \ldots, L_{n_0-1}]$ to obtain a new $p \times 1$ binary vector $s' = L_{n_0-1}s = HQ\, e^T = H\left(Qe^T\right) = H\left(eQ^T\right)^T$.

  Defining the *expanded error vector* as $e' = eQ^T$, the binary vector s' can be thought of as $s' = He'^T$ to the end of applying a QC-LDPC decoding procedure and recover both e' and subsequently the original error vector $e = e'\left(Q^T\right)^{-1}$.

  QC-LDPC decoders are not bounded distance decoders, and some non-zero DFR must be tolerated. The system parameters can be chosen such that the DFR is acceptably small; for this purpose, the average decoding radius of the private code must be sufficiently larger than the Hamming weight of e', which is $t' \leq mt$ and approximately equal to mt, due to the sparsity of Q and e. As shown in line 3 of algorithm $\textsc{Decrypt}^{\mathsf{Nie}}$ in Fig. 1.3(a), the LEDAdecoder algorithm performs the computation of the error vector e saving altogether the computation of both $\left(Q^T\right)^{-1}$ and the subsequent vector-matrix multiplication.

  The decoder adopted in versions < 2.5 of the LEDAcrypt specification allows the error vector e to be recovered directly from the syndrome s and the secret matrices H and Q, while in the current and future specifications and implementations of LEDAcrypt (i.e., the ones with ver. ≥ 2.5) we take the matrix Q equal to an identity matrix of the same size, while the values of the parameters of the cryptoschemes are chosen accordingly as per the NIST security requirements (see Chap. 4). As a consequence, the analyses and algorithms detailed in Chap. 3 to describe the LEDAdecoder allow e to be recovered from the syndrome s and the secret matrix $L = H$.

  If the decoding procedure terminates successfully (i.e., the decoder verifies that the syndrome corresponding to the recovered error is equal to zero), the procedure returns also a Boolean variable $\mathsf{res} = \mathsf{true}$ along with the recovered value e (see line 3 of algorithm $\textsc{Decrypt}^{\mathsf{Nie}}$ in Fig. 1.3(a)). On the other hand, when the decoding fails, the algorithm $\textsc{Decrypt}^{\mathsf{Nie}}$ returns the value of the Boolean variable res set to false and a null value for the original message $e = \perp$ (lines 4–5).

  The *decryption* algorithm $\textsc{Decrypt}^{\mathsf{McE}}$ in Fig. 1.3(b), takes as input a secret key $sk^{\mathsf{McE}} = \{H, Q\}$ and a ciphertext that is identified with an error affected codeword $c = [c_0, \ldots, c_{n_0-1}]$ that was computed as the element-wise addition between a $1 \times pn_0$ error vector with exactly t asserted bits and a $1 \times pn_0$ binary vector $[u_0, \ldots, u_{n_0-2}, \mathsf{blk}]$, where $\mathsf{blk} = \sum_{j=0}^{n_0-2} u_j(L_{n_0-1}HQ)$. The outcomes of the decryption algorithm are the original error vector e and message u.

  The initial steps of the algorithm (lines 1-2 in Fig. 1.3(b)) consist in taking the secret matrices, re-computing $L = HQ$ and executing the multiplication between the parity-check matrix L

Algorithm 6: DECRYPT$^{\mathsf{Nie}}$

Input: s: syndrome; $1 \times p$ binary vector.
$sk^{\mathsf{Nie}} = \{H, Q\}$ private key;

Output: $e = [e_0, \ldots, e_{n_0-1}]$: error; sequence of n_0 binary vectors with size $1 \times p$.
res: Boolean value denoting if the decryption ended successfully (true) or not (false)

Data: $p{>}2$ prime, $\mathsf{ord}_p(2){=}p{-}1$, $n_0 \geq 2$

1 $L \leftarrow HQ$ // $L = [L_0|\ldots|L_{n_0-1}]$
2 $s' \leftarrow L_{n_0-1}s$ // $M{=}L_{n_0-1}L, s{=}Me^T, s'{=}Le^T$
3 $\{e, \mathsf{res}\} \leftarrow$ LEDADECODER(s', sk^{Nie})
4 **if** res = false **then**
5 $\quad e \leftarrow \perp$
6 **return** (e, res)

(a)

Algorithm 7: DECRYPT$^{\mathsf{McE}}$

Input: $c = [c_0, \ldots, c_{n_0-1}]$: error affected codeword; $1 \times pn_0$ binary vector, where each c_j is a $1 \times p$ vector with $0 \leq j < n_0$;
$sk^{\mathsf{McE}} = \{H, Q\}$ private key;

Output: $u = [u_0, \ldots, u_{n_0-2}]$: message; sequence of $n_0{-}1$ binary vectors with size $1 \times p$.
$e = [e_0, \ldots, e_{n_0-1}]$: error; sequence of n_0 binary vectors with size $1 \times p$.
res: Boolean value denoting if the decryption ended successfully (true) or not (false).

Data: $p{>}2$ prime, $\mathsf{ord}_p(2){=}p{-}1$, $n_0 \geq 2$

1 $L \leftarrow HQ$ // $L = [L_0|\ldots|L_{n_0-1}]$
2 $s \leftarrow Lc^T$ // $p \times 1$ binary syndrome
3 $\{e, \mathsf{res}\} \leftarrow$ LEDADECODER(s, sk^{McE})
4 **if** res = true **then**
5 $\quad$ **for** $j = 0$ **to** $n_0 - 1$ **do**
6 $\quad\quad u_j \leftarrow c_j + e_j$
7 **else**
8 $\quad e \leftarrow \perp$; $u \leftarrow \perp$
9 **return** (u, e, res)

(b)

Figure 1.3: Decryption algorithm of Niederreiter (a) and McEliece (b) cryptosystems, instantiated with QC-LDPC codes

and the received error-affected codeword c to obtain a syndrome s as $p \times 1$ binary vector $s = Lc^T$.

If the syndrome decoding phase on line 3 in Fig. 1.3(b) is successful, then res = true and the original message is computed by adding element-wise the first $n_0 - 1$ binary blocks of the error-affected codeword (i.e., the ciphertext) with the recovered error vector (see lines 5–6 of algorithm DECRYPT$^{\mathsf{McE}}$ in Fig. 1.3(b)); otherwise, the variable res is set to false, while the message and error vectors are set to null values, $u{=}\perp$, $e{=}\perp$ (see line 8).

1.3 Description of LEDAcrypt Key Encapsulation Methods

In this section we describe the structure of the Key Encapsulation Method of LEDAcrypt, where the QC-LDPC Niederreiter cryptoscheme described in the previous section, is employed as a *passive secure* (aka *one wayness* CPA secure or OW-CPA secure) building block[1] to design the LEDAcrypt-KEM scheme with a long term public/private keypair, and the LEDAcrypt-KEM-CPA scheme with an ephemeral public/private keypair. In particular, each of the said schemes is composed by a triple of algorithms for the generation of the public/private keypair, the encapsulation of a secret session-key and the decapsulation of a secret session-key, respectively. The key generation algorithms of both schemes follow the definitions and steps already shown in Figure 1.1(a) as Algorithm 2, while they differ in the size of the generated keypairs and in the numerical values of the specific parameters, as described in the following chapters. An encapsulation algorithm enables a sender to compute a secret session-key and the corresponding encrypted (or encapsulated) payload, i.e., a ciphertext, employing the public key of the intended receiver. A decapsulation algorithm enables the intended receiver to employ her private key to decrypt a ciphertext computed with her public key and subsequently derive the secret session-key chosen by the sender.

1.3.1 LEDAcrypt-KEM: encapsulation and decapsulation algorithms

The LEDAcrypt-KEM scheme employs long term keys for scenarios where key reuse is desirable. Assuming the use of a QC-LDPC decoder with appropriately low DFR, the encryption and decryption transformations of the OW-CPA Niederreiter scheme reported in Figure 1.2(a) and in Figure 1.3(a), respectively, are used in the construction reported in Figure 1.4 to obtain the encapsulation and decapsulation algorithms of a Key Encapsulation Module (KEM) with IND-CCA2 guarantees.

The IND-CCA2 security guarantees of LEDAcrypt-KEM are obtained by employing a QC-LDPC decoder with DFR=$2^{-\lambda}$ and the construction described by algorithms Encap and Decap in Figure 1.4, ensuring protection with a security level 2^{λ}, against passive and active adversaries. The security against a passive adversary is provided by the OW-CPA property of the underlying Niederreiter scheme and the fact that session key is derived by a Key Derivation Function (KDF). The security against active adversaries, including the ones considered in the so called reaction attacks [56] – which induces a decryption failure choosing plaintext messages that are in the message space, to recover the private key – is provided building on one of the constructions reported in [39], augmented with a message confirmation mechanism to prevent the choice of a specific plaintext message.

In particular, lines 3–4 of Algorithm Encap in Figure 1.4(a), and lines 1–2, 7–10 of Algorithm Decap in Figure 1.4(b) coincide with the $U_m^{\perp}$ construction of an IND-CCA2 KEM defined in [39]. Specifically, Theorem 3.6 in [39] establishes that, in the random oracle model (ROM), the IND-CCA2 security of the construction $U_m^{\perp}$ tightly reduces to the OW-CPA security of the underlying deterministic public key cryptoscheme, with a bound on the probability of observing a decryption failure when the OW-CPA scheme decrypts a legitimate ciphertext, that is formalized as δ-correctness of the OW-CPA public key cryptoscheme. In our case, the said OW-CPA public key

[1] A deterministic (*key-generation*, *encrypt*, *decrypt*) triple of algorithms is said to be one wayness (OW) if the probability that a polynomial time attacker can invert a randomly generated ciphertext $c = Encrypt(m, pk)$ (where m is chosen at random in the plaintext space) is a negligible function of the cryptosystem parameters. The term "chosen plaintext attack" (CPA) is used because the attacker is not allowed to make decryption queries.

Algorithm 8: LEDAcrypt-KEM ENCAP

Input: pk^{Nie}: public key.
Output: K: ephemeral key;
c: encapsulated secret;
x: tag.

1 $\mathsf{seed} \leftarrow \mathrm{TRNG}()$
2 $e \leftarrow \mathrm{XOF}\,(\mathsf{Seed})$
3 $K \leftarrow \mathrm{KDF}(e)$
4 $c \leftarrow \mathrm{ENCRYPT}^{\mathsf{Nie}}\left(e, pk^{\mathsf{Nie}}\right)$
5 $\mathsf{mask} \leftarrow \mathrm{HASH}(e)$
6 $x \leftarrow \mathsf{seed} \oplus \mathsf{mask}$
7 **return** (K, c, x)

(a)

Algorithm 9: LEDAcrypt-KEM DECAP

Input: sk^{Nie}: secret key
$\mathsf{LongTermSeed}$: a secret random bitstring;
c: encapsulated secret;
x: tag.
Output: K: ephemeral key.

1 $\{e, \mathsf{res}\} \leftarrow \mathrm{DECRYPT}^{\mathsf{Nie}}\left(c, sk^{\mathsf{Nie}}\right)$
2 $\mathsf{tmp} \leftarrow \mathrm{CONCATENATE}(\mathsf{LongTermSeed}, c)$
3 $\mathsf{mask} \leftarrow \mathrm{HASH}(e)$
4 $\mathsf{seed} \leftarrow \mathsf{mask} \oplus x$
5 $\mathsf{test_e} \leftarrow \mathrm{XOF}\,(\mathsf{seed})$
6 **if** $\mathsf{res} = \mathsf{true}$ and $wt(e) = t$ and $\mathsf{test_e} = e$ **then**
7 $\quad K \leftarrow \mathrm{KDF}(e)$
8 **else**
9 $\quad K \leftarrow \mathrm{KDF}(\mathsf{tmp})$
10 **return** K

(b)

Figure 1.4: Description of the key encapsulation and decapsulation primitives of LEDAcrypt-KEM

cryptoscheme coincides with the Niederreiter scheme described in Section 1.2 (see Figure 1.1(a), Figure 1.2(a), Figure 1.3(a)).
The same construction and the same assumption about the OW-CPA security of the underlying deterministic cryptosystem were proven to achieve IND-CCA2 also in the quantum oracle model (QROM) in [41], with a tighter security reduction being reported in [42].

The proofs in [39, 41, 42] define the δ-correctness of the underlying OW-CPA scheme as the probability that a (possibly unbounded) adversary is able to induce a decryption failure on a valid ciphertext, taken as the **maximum** over all possible plaintexts, and averaged over all the keypairs. The proofs require that $\delta \leq 2^{-\lambda}$ for the KEM construction to exhibit IND-CCA2 assurances with a security level 2^{λ}.

The definition of δ-correctness does not match the one usually employed to analyze the failure probability of a Niederreiter public key cryptosystem relying on QC-LDPC codes. Indeed, it is customary, in the coding theory community, to evaluate the decoding failure probability (or decoding failure rate – DFR) of a given code (i.e., Niederreiter keypair) **averaged** over all the possible plaintexts (i.e., error vectors).

We solve the mismatch between the said definitions, proposing an augmentation of the $U_m^{\perp}$ construction, where we add a mechanism to prevent the adversary from exploiting the advantage which he may have in selecting a set of input messages, i.e., error vectors, causing a decryption failure

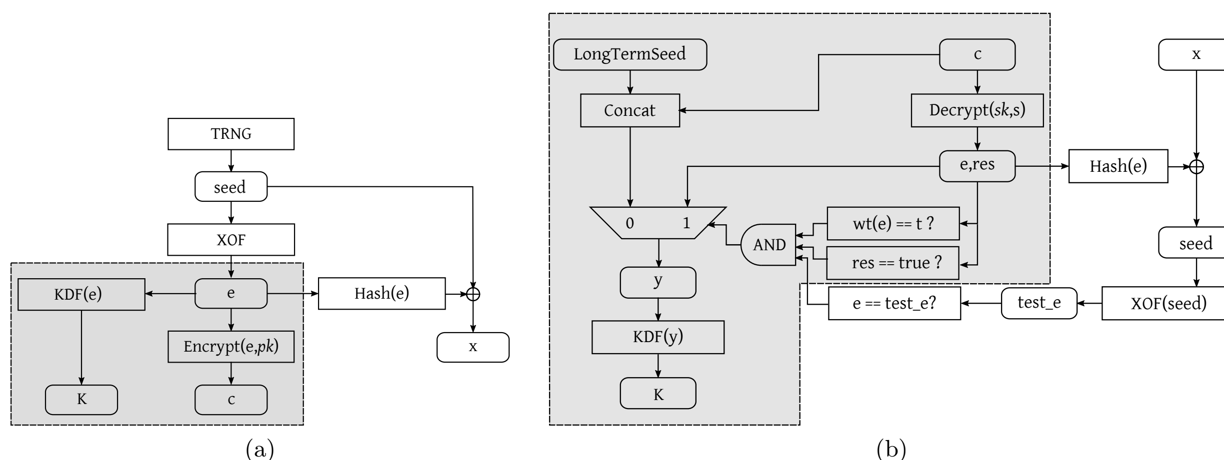


Figure 1.5: Graphical representation of the IND-CCA2 LEDAcrypt-KEM Encap (a), and LEDAcrypt-KEM Decap primitives (b)

with probability different from (and possibly higher than) the average over all the error vectors. The said mechanism makes the maximum probability of a decryption failure over all plaintexts coincide with the average failure probability over all plaintexts, by taking off from the adversary the opportunity to feed the KEM encapsulation function with a plaintext message of her choice.

Figure 1.5 shows a graphical description of the operations of Algorithms Encap and Decap in Figure 1.4, where the gray boxes highlight the operations coinciding with the IND-CCA2 KEM based on the $U_m^{\perp}$ construction in [39].

As shown in Figure 1.5(a), our augmentation generates the confidential message (i.e., the error vector), e, to be processed by the underlying OW-CPA Niederreiter encryption function as the output of an Extensible Output Function (XOF) fed with an ephemeral secret value `seed`. The error vector e is subsequently encapsulated in the encrypted payload, c, following the $U_m^{\perp}$ construction. The rest of the mechanism computes a tag, x, as the bitwise-`xor` between the `seed` and the hash-digest of the error vector itself. The recipient of the message is meant to receive both the encrypted payload c and the tag x to verify that the error vector was actually generated as the output of an XOF.

As shown in Figure 1.5(b), the recipient of c and x, after recovering the error vector, e, via the QC-LDPC syndrome decoding performed by the underlying OW-CPA Niederreiter decryption function fed with c and the private key, proceeds to recover the XOF input employed by the sender (`seed`) by computing the bitwise-`xor` between the received tag x and the hash-digest of the recovered error vector. Subsequently, a test error vector e' is computed as XOF(`seed`) at the recipient end. If the test error vector e' matches the one output from the syndrome decoding, e, there are only two possibilities. The first one makes the recipient confident that the recovered error vector e (i.e. the received KEM plaintext message) was actually generated as the output of an XOF by a sender choosing an ephemeral `seed` value and following the proposed construction. The second possibility is that an adversary is able to obtain a preimage of the XOF for an arbitrary error vector e. Indeed, if the adversary is able to do so, he can choose a plaintext message e for the OW-CPA Niederreiter encapsulation function at his own will and can compute the accompanying tag x as $\mathrm{XOF}^{-1}(e) \oplus \textsc{Hash}(e)$. Therefore, if the computation of a preimage of the XOF is possible, the adversary can induce a decryption failure probability of the OW-CPA Niederreiter scheme, during

the execution of the decapsulation function, that is different from the one averaged over all error vectors, and the overall construction cannot be claimed to be IND-CCA2.

We summarize the security guarantees of our augmented construction in the following lemma.

Lemma 1.3.1 Given an OW-CPA Niederreiter public key cryptosystem, based on QC-LDPC codes, with DFR=δ, and a preimage-resistant XOF, the LEDAcrypt-KEM construction shown in Figure 1.4, or equivalently in Figure 1.5, provides IND-CCA2 guarantees in the ROM and QROM model.

Proof. In all the trivial cases where an adversary abides by the error vector generation mechanism shown in Figure 1.4, he will pick a random error vector among all the possible ones, therefore matching the maximum probability of a decryption failure with the average one over all error vectors.

We now show that, being able to derive the value x for a chosen value of e is at least as hard as finding a preimage of XOF, reducing the problem of obtaining a preimage of XOF for an error-vector looking output to finding a valid x passing the check expressed as the third clause at line 7, in Figure 1.4. To this end, consider the algorithm $\mathsf{A}_{\mathsf{find\,tag}}(e)$, which computes a tag value x so that e passes the check expressed as the third clause at line 7, in Figure 1.4. Given that the said check validates the equality $e = \text{XOF}(x \oplus \textsc{Hash}(e))$, the value x computed by $\mathsf{A}_{\mathsf{find\,tag}}(e)$ should be such that $x = \text{XOF}^{-1}(e) \oplus \textsc{Hash}(e)$. We are now able to provide an algorithm $\mathsf{A}_{\mathsf{inv\,XOF}}(b)$, which requires only a single call to $\mathsf{A}_{\mathsf{find\,tag}}(e)$ to find a preimage a for a given b through XOF, that is, $\text{XOF}(a) = b$. This, in turn, proves that computing $\mathsf{A}_{\mathsf{inv\,XOF}}(b)$ is no harder than $\mathsf{A}_{\mathsf{find\,tag}}(e)$, or, in other words, that computing $\mathsf{A}_{\mathsf{find\,tag}}(e)$ is at least as hard as computing $\mathsf{A}_{\mathsf{inv\,XOF}}(b)$.
The procedural description of $\mathsf{A}_{\mathsf{inv\,XOF}}(b)$ is as follows:

(i) $y \leftarrow \mathsf{A}_{\mathsf{find\,tag}}(b)$

(ii) $h \leftarrow \textsc{Hash}(b)$

(iii) **return** $h \oplus y$

The correctness of the provided $\mathsf{A}_{\mathsf{inv\,XOF}}(b)$ is proven observing that $y = \text{XOF}^{-1}(b) \oplus \textsc{Hash}(b)$, therefore $h \oplus y$ can be rewritten as $\textsc{Hash}(b) \oplus \text{XOF}^{-1}(b) \oplus \textsc{Hash}(b) = \text{XOF}^{-1}(b)$, obtaining a preimage of b through XOF. Therefore, for an attacker to be able to employ a chosen value for the error vector e instead of a randomly drawn one it is necessary for him to be able to find a preimage of XOF, which is not feasible by hypothesis. Therefore, the best correctness error δ the attacker may achieve matches the DFR of the underlying OW-CPA Niederreiter scheme, in turn making the proofs in the ROM and QROM model in [39,41,42] hold for the steps denoted as $U_m^{\perp}$ contained in the gray part in Figure 1.5. ■

1.3.2 LEDAcrypt-KEM-CPA: encapsulation and decapsulation algorithms

The LEDAcrypt-KEM-CPA scheme employs ephemeral public/private keypairs for scenarios where the security property of indistinguishability against chosen ciphertext attacks (IND-CPA) is usually required and the Perfect Forward Secrecy (PFS) property is desirable.

The encryption and decryption transformations of the LEDAcrypt-KEM-CPA scheme employ the same IND-CCA2 construction defined in the previous section. The differences with respect to the scheme with long-term keys will be in terms of DFR and execution times. Indeed, LEDAcrypt-KEM-CPA will exhibit, w.r.t. the LEDAcrypt-KEM scheme with long-term keys, a DFR=10^{-9} per public/private keypair as well as faster key-generation times and faster decryption times due to different choices of the underlying QC-LDPC code parameters and the choice of the decoder applied by the OW-CPA Niederreiter building block.

As the construction reported in Figure 1.5 allows to exhibit IND-CCA2 guarantees also for the LEDAcrypt-KEM-CPA scheme, this implies that IND-CPA guarantees are provided as well. A simple lemma declaring the IND-CPA property of the LEDAcrypt-KEM-CPA scheme is as follows.

Lemma 1.3.2 Given an OW-CPA Niederreiter public key cryptosystem, based on QC-LDPC, and a preimage-resistant KDF, the LEDAcrypt-KEM-CPA construction shown in Figure 1.5 is IND-CPA secure.

Proof. The IND-CPA definition for a KEM states the computational indistinguishability of the two following games: Game1 and Game2.

In Game1 a challenger generates a pair of public/private keys (pk, sk). Subsequently, he picks a confidential message e (i.e., the error vector in our case) in a uniformly random way, and applies the KEM encapsulation function to get a session key K_0 and the corresponding encrypted payload c (i.e., $(c, K_0) \leftarrow \textsc{Encap}(e, pk)$). Finally the challenger in Game1 outputs the tuple (pk,c,K_0).

In Game2 a challenger generates a pair of public/private keys (pk, sk). Subsequently, he picks a confidential message e (i.e., the error vector in our case) in a uniformly random way, and applies the KEM encapsulation function to get a session key K_0 and the corresponding encrypted payload c (i.e., $(c, K_0) \leftarrow \textsc{Encap}(e, pk)$). Finally the challenger in Game2 outputs the tuple (pk,c,K_1), where K_1 is a session key that is uniformly random picked.

In case of Game1, an honest run of the KEM encapsulation function is executed and the session key is derived as the output of a KDF fed with the output of an XOF which is in turn fed with an ephemeral random seed. Since the outputs of a KDF and an XOF are indistinguishable from uniformly random sequences, the only way the adversary could distinguish K_0 from K_1 would be to derive the error vector e by inverting the KDF and subsequently match c with the outcome of the KEM encapsulation function fed with the recovered e and the public key. Alternatively, the adversary should guess the private key to correctly decapsulate the ciphertext c, derive the error vector and compute the corresponding session key. Since, the guessing of the private key of the Niederreiter public key cryptosystem is prevented by the OW-CPA hypothesis and also the KDF is preimage resistant by hypothesis, the given games are computationally indistinguishable applying the key generation and encapsulation functions of the LEDAcrypt-KEM-CPA scheme. ■

Concerning the additional security guarantees of the LEDAcrypt-KEM-CPA scheme, it is easy to acknowledge that the IND-CCA2 construction (see Figure 1.5) of the scheme is sufficient to foil the private key or message recovery attempts executed by active adversaries able to exploit the *accidental or moderate* reuse of the ephemeral public/private keypairs, Indeed, in the event of a keypair reuse, a (possibly active) attacker [56] will not be able to craft messages inducing decoding failures faster than drawing them randomly, due to the message confirmation tag included in the IND-CCA2 construction. As a consequence, reusing an ephemeral keypair for a number of times $\ll \frac{1}{\mathrm{DFR}}$ is unlikely to provide any advantage to an attacker.

1.4 Description of LEDAcrypt Public Key Cryptosystem

In this section we describe the structure of the Public Key Cryptosystem of LEDAcrypt, where the QC-LDPC McEliece cryptoscheme described in Section 1.2 is employed as a building block in a construction allowing to exhibit IND-CCA2 guarantees. While it is possible to employ LEDAcrypt-KEM in combination with a symmetric encryption primitive to get a Key Encapsulation Module + Data Encapsulation Mechanism (KEM+DEM), we note that such an approach may lead to a non-negligible ciphertext expansion in case plaintexts are small in size.

In particular, the LEDAcrypt-PKC scheme is composed by a triple of algorithms for the generation of the public/private keypair, the encryption transformation of a **plaintext message with arbitrary length**, and the decryption transformation of a **ciphertext with arbitrary length**, respectively. The key generation algorithm of LEDAcrypt-PKC follows the definitions and steps already shown in Figure 1.1(b) as Algorithm 3, while the encryption and decryption transformations are defined augmenting the McEliece encryption and decryption primitives in Figure 1.2(b) and Figure 1.3(b) as prescribed by the IND-CCA2 γ-construction introduced in [45]. It is worth noting that the systematic form of the generator matrix of the public code included in the public key, pk^{McE}, would easily allow any observer to recover the information word u embedded in an encrypted message c, without recovering the private key of the cipher (i.e., $sk^{\mathsf{McE}} = \{H, Q\}$). Nevertheless, the conversion proposed by Kobara and Imai in [45], with the purpose of maximizing the amount of message encrypted via a McEliece encryption transformation, allows IND-CCA2 guarantees to be provided in the Random Oracle Model (ROM). Indeed, one of the elements of the Kobara and Imai γ-construction [45] is to obfuscate the plaintext message, before encrypting it with the McEliece transformation, employing a bit-wise xor between the plaintext message and a secret pseudo-random mask that can be re-computed also at the recipient side. As a consequence, the confidentiality of the information word as well as the secrecy of the private key remain guaranteed by the hardness of the NP-hard syndrome decoding problem that an adversary should face in recovering the error vector knowing the ciphertext (codeword) and the systematic generator matrix employed as public key.

Figure 1.6 and Figure 1.7 describe the encryption and decryption transformations of the IND-CCA2 LEDAcrypt-PKC, respectively. As the Kobara-Imai (KI) γ-conversion [45] is based on bit-wise manipulations, to provide a clear and detailed description, we introduce some specific naming conventions. Bit-string values will be reported with a teletype font name (e.g., the plaintext to be encrypted will be denoted as ptx, and the resulting ciphertext will be denoted as ctx) while, the length of a bit-string, s, will also be expressed in bits and denoted as l_{s}.

1.4.1 LEDAcrypt-PKC: encryption and decryption transformations

A pseudo-code description of the KI-γ encryption algorithm is shown as Algorithm 10 in Fig. 1.6. The main intuition of the KI conversion relies on xor-combining a padded plaintext message with the output of a Deterministic Random Bit Generator (DRBG), seeded with the random bit-string extracted from a True Random Number Generator (TRNG). To this end, we chose the NIST standard PRNG CTR-DRBG, instantiated with AES-256 as its core block cipher. Specifically, the original plaintext message (with size l_{ptx} bits) is prefixed with a bit-string named constant, having l_{const} bits, and a bit-string named lenField, having l_{lenField} bits, containing the value of l_{ptx}.

This message is then concatenated, if necessary, with a sequence of null bits to align the over-

Algorithm 10: LEDAcrypt-PKC encryption transformation

Data: n, k, t: QC-LDPC code parameters. $n = pn_0$ codeword size, $k = p(n_0 - 1)$ information word size, t error correction capability, n_0 basic block length of the code, p circulant block size.
HASH: hash function with digest length in bits l_{hash}
$l_{\mathsf{obfuscatedPtx}} = \max\left(p(n_0-1),\ \left\lceil \frac{l_{\mathsf{const}}+l_{\mathsf{ptx}}}{8} \right\rceil \cdot 8\right)$
$l_{\mathsf{const}} = l_{\mathsf{seed}}$
$l_{\mathsf{iword}} = p(n_0 - 1)$
$l_{\mathsf{eword}} = l_{\mathsf{hash}}$
$\mathsf{const} = 0^{l_{\mathsf{seed}}}$

Input: ptx: plaintext bit-string, with $l_{\mathsf{ptx}} \geq 0$
pk^{McE}: QC-LDPC based McEliece public key

Output: ctx ciphertext bit-string

```
1  seed ← TRNG()                                          // bit-string with length l_seed
2  pad ← DRBG(seed)                                       // bit-string with length l_obfuscatedPtx
3  obfuscatedPtx ← ZeroPadByteAligned(const||lenField||ptx) ⊕ pad
4  obfuscatedSeed ← ZeroExtendByteAligned(seed, l_hash) ⊕ Hash(obfuscatedPtx)
5  {iword, leftOver} ← Split(obfuscatedPtx, l_iword, l_obfuscatedPtx − l_iword)
6  u ← ToVector(iword)                                    // 1 × p(n_0 − 1) information word vector
7  e ← ConstantWeightEncoder(obfuscatedSeed)
8  c ← Encrypt^McE(u, e, pk^McE)                          // 1 × pn_0 codeword
9  ctx ← ToBitString(c)||leftover                         // bit-string with l_ctx = pn_0
10 return ctx
```

Figure 1.6: Description of the KI-γ encryption function adopted to define the encryption primitive of LEDAcrypt PKC

all length of such an *extended plaintext* to a byte-multiple for implementation convenience (i.e., $l_{\mathsf{extendedPtx}} = 8 \cdot \left\lceil \frac{l_{\mathsf{const}}+l_{\mathsf{lenField}}+l_{\mathsf{ptx}}}{8} \right\rceil$). The extended plaintext is then xor-combined with the output of a DRBG to allow the resulting *obfuscated plaintext* to be safely enciphered. This ensures the ciphertext to appear perfectly random.

In order for the IND-CCA2 property to hold, a fresh random value (i.e., a seed), encoded with l_{seed} bits, should be available from a TRNG for each message encryption. In our instance of the the KI-γ construction, the bit-length of the DRBG seed equals 128, 192, 256 bits, depending on the security level prescribed by NIST for the DRBG primitive, addressing category 1, 3, or 5, respectively. The lenField bit-string is commonly encoded employing 64 bits, while the arbitrarily chosen value encoded into the bit-string named constant equals, in our instance of the construction, a sequence of l_{seed} zero bits, i.e., $\mathsf{constant} = 0^{l_{\mathsf{seed}}}$. To be able to successfully decrypt the plaintext, the seed of the DRBG is also enciphered alongside the message. The secret seed is xor-combined with the digest computed by a hash function fed with the obfuscated plaintext. The correctness of the resulting *obfuscated seed* is guaranteed by concatenating to the original seed value enough zero bits to match the length of the hash digest in case $l_{\mathsf{seed}} < l_{\mathsf{hash}}$, thus obtaining $l_{\mathsf{obfuscatedSeed}} = l_{\mathsf{hash}}$.

The extended plaintext bits are subsequently split into two bit-strings, namely iword and leftOver. The former, having size of l_{iword} bits, is employed as the information word u to be encoded by the QC-LDPC code underlying the processing performed by the McEliece encryption function (i.e.,

Algorithm 11: LEDAcrypt-PKC decryption transformation

Data: n, k, t: QC-LDPC code parameters. $n = pn_0$ codeword size, $k = p(n_0 - 1)$ information word size, t error correction capability, n_0 basic block length of the code, p circulant block size.
$\texttt{const} = 0^{l_{\texttt{seed}}}$
HASH: hash function with digest length in bits $l_{\texttt{hash}}$

Input: ctx: ciphertext bit-string.
sk^{McE}: LEDAcrypt-PKC private key.

Output: ptx plaintext bit-string

```
1  cword, leftOver ← SPLIT(ctx, pn_0, l_ctx − pn_0)
2  c ← TOVECTOR(cword)
3  {u, e, res} ← DECRYPT^McE(c, sk^McE)
4  if res = true and wt(e) = t then
5      iword ← TOBITSTRING(u)
6      obfuscatedSeed ← CONSTANTWEIGHTDECODE(e)
7      seed ← ZEROTRIM(obfuscatedSeed ⊕ HASH(iword||leftOver), l_seed)
8      pad ← DRBG(seed)
9      extendedPtx ← obfuscatedPtx ⊕ pad        // extendedPtx should equal const||lenField||ptx
10     {retrievedConst, ptx} ← ZEROTRIMANDSPLIT(extendedPtx, l_const, l_lenField)
11     if retrievedConst = const then
12         return ptx
13 return ⊥
```

Figure 1.7: Description of the KI-γ decryption function adopted to define the decryption primitive of LEDAcrypt PKC

$l_{\texttt{iword}} = p(n_0 - 1)$), while the latter (with size $l_{\texttt{leftOver}}$ bits) is concatenated with the output of the McEliece encryption function to yield the final ciphertext (ctx) of the KI-γ construction.

The obfuscated secret seed bits are employed as input to the constant-weight encoding function to obtain a binary error vector e, with exactly t asserted bits, which is in turn fed into the McEliece encryption function to complete a proper initialization of the encryption process. In case the number of bits of the obfuscated seed (i.e., $l_{\texttt{obfuscatedSeed}} = l_{\texttt{hash}}$) are less than the ones required by the constant-weight encoding procedure, random bits are extracted from the DRBG (or from a new random source) to match the requirements.

The final ciphertext is composed by a bit-string made of the binary elements of the codeword computed by the McEliece encryption function, concatenated with the leftOver bit-string. In case the bit-length of leftOver is not a multiple of 8, leftOver is prefixed with a zero pad up to the nearest byte for the sake of implementation ease and efficiency.

It is worth noting that the KI-γ construction allows a plaintext message of arbitrary length to be encrypted. Employing the LEDAcrypt-PKC construction, plaintext messages longer than $p(n_0 - 1) - l_{\texttt{const}} - l_{\texttt{lenField}}$ have a fixed ciphertext expansion of $p + l_{\texttt{const}} + l_{\texttt{lenField}}$.
In contrast a LEDAcrypt-KEM+DEM approach requires a fixed ciphertext expansion of pn_0 bits.

The described procedure employs the QC-LDPC code parameters of choice (i.e., the ones of

$\mathtt{C}(pn_0, p(n_0-1))$ that allows up to t bit errors to be corrected), and a hash function, HASH(), having a digest with a byte-aligned bit-length l_{hash} as configuration parameters. We chose as our hash function, to instantiate the construction, the NIST Standard SHA-3 hash function.

Algorithm 11 in Fig. 1.7 reports the pseudocode of the LEDAcrypt-PKC decryption procedure. The algorithm performs, in reverse order, the steps dual to the ones in the encryption procedure. The value in the first l_{const} bits of the retrieved extended plaintext message is compared with the fixed value of the constant considered in the KI-γ to reject the decryption outcome in case they mismatch. The rationale underlying the design choice of prefixing to the plaintext with the `constant` string is that the probability of the DRBG generating the same initial l_{const} bits when it is fed with distinct seeds is negligible (2^{-128}, 2^{-192}, 2^{-256}, respectively) and the probability of failing the detection of a correct decryption of the *de-obfuscated* concatenation of `constant`, `lenField`, and plaintext by checking only the value `constant` is also negligible.

As the McEliece decryption transformation employs a QC-LDPC code with an intrinsically non-bounded decoding radius, the decryption process of the KI-γ may have an additional reason to fail. As shown in the pseudocode, the IND-CCA2 property[2] is preserved making a decoding failure (consider the failed check at line 3 of Algorithm 11) indistinguishable from an accidental error (consider the failed check at line 11 of Algorithm 11), and preventing active adversaries to freely choose both the seed and the plaintext.

The security guarantees of the LEDAcrypt-PKC scheme are summarized in the following lemma.

Lemma 1.4.1 Given an OW-CPA McEliece cryptosystem based on the QC-LDPC syndrome decoding problem as the one in Section 1.2, with DFR=δ, the LEDAcrypt-PKC construction provides IND-CCA2 guarantees in the ROM model.

Proof. (Sketch)
The Fujisaki-Okamoto (FO) generic conversion presented in [30,31] enhances a correct (i.e., DFR $=0$) OW-CPA public key encryption scheme to exhibit indistinguishability against adaptive chosen ciphertext (IND-CCA2) attacks in the random oracle model. In [45], Kobara and Imai introduced three equivalent constructions (KI-α, KI-β, KI-γ) tailored specifically for the McEliece cryptossystem rather than a general OW-CPA encryption scheme, showing considerable savings in terms of ciphertext expansion and the possibility to employ also a systematic generator matrix without affecting the IND-CCA2 guarantees. In [39] the authors solves the issues of the FO and KI conversions related to the need of a perfectly correct OW-CPA public key scheme and tight security reductions, leveraging the notion of δ-correctness. The δ-correctness definition in [39] considers the probability of failure of the decryption primitive on a legitimate ciphertext, averaged over all possible key pairs and taken as the maximum achievable by an adversary who knows both the public and private key of the scheme. The decoding failure analysis adopted by the McEliece cryptosystem, underlying the LEDAcrypt-PKC scheme, considers an upper bound on the decoding failure rate for any given keypair, assuming that the adversary is randomly picking error vectors. This analysis matches the requirements of the definitions by [39]. In fact, due to the KI-γ construction the maximization of the decoding failure rate over all the plaintexts has no effects, since the plaintext message in the KI-γ is employed as a codeword of the underlying systematic McEliece scheme, and thus does not influence the decoding failure probability. Indeed, the error vector in the KI-γ construction is picked as the output of a hash (encoded with a constant weight encoding procedure) and thus the

[2] given that the decoding failure rate is low enough

attacker is not able to choose a specific error pattern unless he finds a preimage for the said hash. ∎

1.4.2 Constant weight encoding/decoding

A direct approach to implement the constant-weight encoding and decoding procedures may consist in applying the conversion mandated by a combinatorial number system [44], which maps (bijectively) any combination of n bits with t asserted bits to an integer value in the range $\{0, \ldots, \binom{n}{t}\}$ and viceversa thus, building an enumeration of the set of combinations. However, computing the said bijection requires a computational effort in the same range as computing the binomial $\binom{n}{t}$, which adversely impacts the performance of LEDAcrypt when n is in a range of tens of thousands and t in the hundreds range.

In [7], we introduced an efficiently computable, bijective function which yields the desired mapping with a computational complexity linear in the length of the string at hand, and delineate how the said function can be computed in constant time. The details of the implemented construction and the related experimental campaign showing speed-ups from three to five orders of magnitude with respect to other state-of-the-art solutions, can be found either in [7] or in the pre-print draft available at `https://www.ledacrypt.org/documents/cf2020.pdf`.

Chapter 2

Security analysis of LEDAcrypt

In order to analyze the security of the LEDAcrypt primitives, we start by providing a quantitative assessment of NIST's security level targets in terms of the number of classical and quantum elementary operations required to perform an exhaustive key search against AES.

Then, we analyze the best known attacks against LEDAcrypt from the perspective of a passive attacker, willing to perform either a message or key recovery attack given a ciphertext or the public key, respectively. Finally, we analyze the best known attacks in the context of an active attacker with the aim of providing IND-CCA2 guarantees for the LEDAcrypt cryptosystem.

The main known attacks against LEDAcrypt are those applicable against QC-LDPC code-based cryptosystems [5], which have been studied for twelve years since the first proposal appeared in [4], plus statistical attacks recently introduced in [27, 38]. We carefully analyze their capabilities and address the design of parameters for the LEDAcrypt primitives to provide the required security guarantees taking into account the computational cost reduction following from the use of a quantum computer in the solution of the underlying computationally hard problems.

In addition to the aforementioned attacks, it has recently been pointed out [57] that the occurrence of some weak keys may follow from the product structure of L in Eq. (1.14). Such a vulnerability is also taken into account in this chapter, and suitably countered.

2.1 Quantitative security level goals

The bar to be cleared to design parameters for post-quantum cryptosystems is the computational effort required on either a classical or a quantum computer to break the AES with a key size of λ bits, $\lambda \in \{128, 192, 256\}$, through an exhaustive key search. The three pairs of computational efforts required on a classical and quantum computer correspond to NIST Category 1, 3, and 5, respectively [53]. Throughout the design of the parameters for the LEDA cryptosystems we ignore Categories 2 and 4: if a cipher matching those security levels is required, we advise to employ the parameters for Categories 3 and 5, respectively.

The computational worst-case complexity of breaking AES on a classical computer can be estimated as $2^{\lambda}\mathsf{C}_{\mathsf{AES}}$, where $\mathsf{C}_{\mathsf{AES}}$ is the amount of binary operations required to compute AES on a classical computer on a small set of plaintexts, and match them with a small set of corresponding ciphertexts

to validate the correct key retrieval. Indeed, more than a single plaintext-ciphertext pair is required to retrieve AES keys [34]. In particular, a validation on three plaintext-ciphertext pairs should be performed for AES-128, on four pairs for AES-192 and on five for AES-256.

Willing to consider a realistic AES implementation for exhaustive key search purposes, we refer to [73], where the authors survey the state-of-the-art of Application-Specific Integrated Circuit (ASIC) AES implementations, employing the throughput per Gate Equivalent (GE) as their figure of merit. The most performing AES implementations are the ones proposed in [73], and require around 16ki GEs. We thus deem reasonable to estimate the computational complexity of an execution of AES as 16ki binary operations. We are aware of the fact that this is still a conservative estimate, as we ignore the cost of the interconnections required to carry the required data to the AES cores.

The computational complexity of performing an AES key retrieval employing a quantum computer was measured first in [34], where a detailed implementation of an AES breaker is provided. The computation considers an implementation of Grover's algorithm [35] seeking the zeros of the function given by the binary comparison of a set of AES ciphertexts with the encryption of their corresponding plaintexts for all the possible key values. In a more recent work, the authors of [40] proposed a more optimized version of the Grover oracle function to perform a key recovery attack on AES, and provided complexity figures for the full circuit depth of the breaker of both their proposals, and the one by [34], which we summarize in Table 2.1.

The NIST call takes into account the fact that quantum cryptanalytic machines implementing a Grover-based key finding strategy may be limited more severely by a long sequential computation, due to the stability of the qubits, and the (square-root) reduced effectiveness of parallelizing Grover's algorithm. As a consequence, the call specifies an expected maximum circuit depth for the quantum codebreaking circuits, and expresses the corresponding AES codebreaking complexities taking the effects of such a bound on the circuit depth into account.

Currently, no gate level description of an effective oracle function to perform quantum codebreaking on a code-based cryptosystem is publicly available. It is therefore hard to assess directly how much a limit on the circuit depth would affect the entire codebreaking circuit. To overcome this lack of information, we compare the depth of the AES breaker in the ideal case (i.e., without limiting the maximum depth of the quantum circuit) with a similarly maximum-depth unlimited circuit to perform a Grover-based attack on LEDAcrypt. We consider this comparison sufficiently fair, given the current state-of-the-art of the description of quantum circuits to attack code-based cryptosystems, as we expect that a reduction of the maximum effective depth to have a similar adverse effect on the performance of both the AES and the LEDAcrypt quantum codebreaking circuits.

In Table 2.1 we summarize the computational cost of performing exhaustive key searches on all three AES variants (i.e., with 128, 192, and 256 bits long keys), both considering classical and quantum computers. For the sake of simplicity, in the following we will round up fractional exponents of the reported complexities to the next integer value.

Table 2.1: Classical and quantum computational costs to perform an exhaustive key search on AES. The quantum computational cost is expressed as the full depth of the AES circuit, assuming no limits on the maximum constructible quantum circuit are present

NIST Category	AES Key Size (bits)	Classical Cost (binary operations)	Quantum Cost [34] (quantum gates)	Quantum Cost [40] (quantum gates)
1	128	$2^{128} \cdot 2^{14} \cdot 3 = 2^{143.5}$	$1.16 \cdot 2^{81} \approx 2^{81.2}$	$1.08 \cdot 2^{75} \approx 2^{75.1}$
3	192	$2^{192} \cdot 2^{14} \cdot 4 = 2^{208}$	$1.33 \cdot 2^{113} \approx 2^{113.4}$	$1.33 \cdot 2^{107} \approx 2^{107.4}$
5	256	$2^{256} \cdot 2^{14} \cdot 5 = 2^{272.3}$	$1.57 \cdot 2^{145} \approx 2^{145.6}$	$1.57 \cdot 2^{139} \approx 2^{139.6}$

2.2 Hardness of the underlying problem

The set of computational decision problems for which an efficient solution algorithm can be devised for a non-deterministic Turing Machine (TM) represents a fruitful computational class from which primitives for asymmetric cryptosystems have been designed. Such a computational class, known as the Nondeterministic-Polynomial (NP) class, is characterized by problems for which it is efficient (i.e., there is a polynomial-time algorithm) to verify the correctness of a solution on a deterministic TM, while finding a solution to the problem does not have in general an efficient algorithm on a deterministic machine, hence the computational asymmetry required to build a cryptosystem.

When considering a quantum TM, i.e., the abstract computational model for a quantum computer, the class of problems which can be solved in polynomial time, with the quantum TM providing the correct answer with probability $> \frac{2}{3}$, is known as the Bounded-error Quantum Polynomial (BQP) time class [15].

In 1997 Peter Shor proved that the integer factoring problem, which has its decisional version in NP, is effectively in BQP [66], in turn demonstrating that a widely adopted cryptographic trapdoor function can be broken in polynomial time by a quantum computer. Consequentially, to devise a proper post-quantum asymmetric primitive, it is crucial to choose a computational problem which resides outside BQP as its underlying foundation. While there is no current formal proof, a sub-class of NP, the NP-complete problem class, is widely believed to contain computational problems not belonging to BQP, thus allowing only a polynomial speedup in their solution with a quantum TM.

LEDAcrypt is constructed starting from the computational search problems: *i)* performing the decoding of a codeword of a random linear code, i.e., finding the error vector affecting a codeword, known as Syndrome Decoding Problem (SDP), and *ii)* finding a bounded weight codeword of a generic random linear code, known as the Codeword Finding Problem (CFP). The decision problems corresponding to the SDP and CFP were shown to be NP-complete in [12], and, as for all NP-complete problems, we have a natural search-to-decision reduction, as noted in [3, §2.5].

In addition to SDP and CFP, it is also known that determining the minimum distance of a random code is also NP-hard, since its corresponding decision problem was proven to be NP-complete in [74].

Finally, in [11] was proven that the syndrome decoding problem remains NP-hard, with its decision version being NP-complete, even in the case of a random quasi-cyclic code.

LEDAcrypt primitives rely on the indistinguishability of their generator matrix G and parity-check matrix H from the ones of a random, quasi-cyclic linear code. Provided that this indistinguishability holds, the problems of performing a key recovery attack, or a message recovery attack against LEDAcrypt are equivalent to solving either a CFP or an SDP respectively. This in turn makes the Niederreiter and McEliece cryptosystems described in Section 1.2 OW-CPA under the assumption that no efficient method to solve the CFP, or the SDP exists.

At the moment of writing, the only technique known to exploit the non random nature of the H and G matrices of LEDAcrypt is to rely on the low-weight of the corresponding secret QC-LDPC code.

2.3 Attacks based on information set decoding

The most computationally effective technique known to attack the LEDAcrypt schemes is the same technique solving the SDP and CFP on random binary linear block codes with the same length, rate and number of errors. Among the available set of techniques, the ones which are most effective are known as Information Set Decoding (ISD). ISD was invented as a general efficient decoding technique for a random binary linear block code by Eugene Prange in [58]. While ISD is indeed more efficient than guessing the error affected positions in an incorrect codeword, its complexity is still exponential in the number of errors affecting the codeword.

ISD can thus be employed as a *message recovery attack* in both McEliece and Niederreiter cryptosystems, solving the corresponding SDPs. In the former case, it will recover the error pattern from the error affected codeword constituting the ciphertext, allowing to recover the original message; in the Niederreiter case, ISD will derive the error vector (which corresponds to the message itself) from the syndrome constituting the ciphertext. When a message recovery attack of this kind is performed against a cryptosystem exploiting quasi-cyclic codes, such as the case of LEDAcrypt, it is known that a polynomial speedup factor equal to the square root of the circulant block size can be achieved [65].

ISD algorithms have a long development history, dating back to the early '60s [58], and provide a way to recover the error pattern affecting a codeword of a generic random linear block code given a representation of the code in the form of either its generator or parity-check matrix.

Despite the fact that the improvement provided by ISD over the straightforward enumeration of all the possible error vectors affecting the codeword is only polynomial, employing an ISD technique provides substantial speedups. It is customary for ISD variant proposers to evaluate the effectiveness of their attacks considering the improvement on a worst-case scenario as far as the code rate and number of corrected errors goes (see, for instance [9]). Such an approach allows to derive the computational complexity as a function of a single variable, typically taken to be the code length n, and obtaining asymptotic bounds for the behavior of the algorithms.

In our parameter design, however, we chose to employ non-asymptotic estimates of the computational complexity of the ISD attacks. Therefore, we explicitly compute the amount of time employing a non-asymptotic analysis of the complexity of ISD algorithms, given the candidate parameters of the code at hand. This approach permits us to retain the freedom to pick rates for our codes which are different from the worst-case one for decoding, thus exploring different trade-offs in the choice of the system parameters. It is common for ISD variants to have free parameters,

which should be tuned to achieve optimal performance. We sought the optimal case by explicitly computing the complexity of the ISD variant for a large region of the parameter space, where the minimum complexity resides. In particular, we note that all such free parameters are tuning points for computational tradeoffs.

We consider the ISD variants proposed by Prange [58], Lee and Brickell [47], Leon [48], Stern [69], Finiasz and Sendrier [29], and Becker, Joux, May and Meurer (BJMM) [9], in our computational complexity evaluation on classical computers. The reason for considering all of them is to avoid concerns on whether their computational complexity in the finite-length regime is already well approximated by their asymptotic behavior. For all the attacks, we consider the complexity as obtained with a logarithmic memory access cost, considering as the amount of required memory only the one taken by the lists in case of collision-based ISD techniques. Such a choice is mutuated by the fact that the size of such lists is exponential in the code parameters, and thus has a small, but non negligible, impact on the computation complexity.ISD In order to estimate the computational complexity of ISD on quantum computing machines, we consider the results reported in [22], which are the same employed in the original specification [6]. Since complete and detailed formulas are available only for the ISD algorithms proposed by Lee and Brickell [47], and Stern [69], we consider those as our computational complexity bound. While asymptotic bounds show that executing a quantum ISD derived from the May-Meurer-Thomae (MMT) algorithm [51] is faster than a quantum version of Stern's [43], we note that there is no computational complexity formulas showing this in the finite regime.

We note that solving the CFP problem employing a given ISD variant to search for a codeword of weight w in a random code with an $r \times n$ parity-check matrix is slightly faster than solving an SDP searching for an error of weight w on the same code. In the following, we will denote with $\mathsf{C\text{-}ISD}_{\mathsf{sdp}}(n, r, t)$ and $\mathsf{Q\text{-}ISD}_{\mathsf{sdp}}(n, r, t)$ the cost of solving the SDP on a code represented by an $r \times n$ parity-check matrix, finding an error of weight t, with a classical and quantum computer, respectively. We denote with $\mathsf{C\text{-}ISD}_{\mathsf{cfp}}(n, r, t)$ and $\mathsf{Q\text{-}ISD}_{\mathsf{cfp}}(n, r, t)$ the complexities of solving the CFP on a code represented by an $r \times n$ parity-check matrix, finding a codeword of weight w, with a classical and quantum computer, respectively. In the following, we will omit the C- and Q- prefixes whenever the statements apply to both computational costs.

2.3.1 Key recovery attacks via codeword finding

Next, we describe how ISD techniques can be employed to perform a key recovery attack against the codes employed in LEDAcrypt. More precisely, we consider three different approaches and discuss their computational complexities.

Finding low weight codewords in C. In LEDAcrypt, the public key is the representation of a code C whose parity-check matrix L in (1.14) is in the form

$$L = \left[L_0, L_1, \cdots, L_{n_0-1}\right], \tag{2.1}$$

where each block L_i is a $p \times p$ circulant matrix with weight $v \leq d_v m$.

It can be shown that C has minimum distance $2v$; indeed, let us consider two distinct integers

$0 \leq i_0, i_1 \leq n_0 - 1$, and define the $p \times n_0 p$ matrix C as follows

$$C = [C_0, \cdots, C_{n_0-1}], \text{ with } C_i \in \mathbb{F}_2^{p\times p}, \; C_i = \begin{cases} 0 & \text{if } i \neq i_0, i_1 \\ L_{i_1}^T & \text{if } i = i_0 \\ L_{i_0}^T & \text{if } i = i_1 \end{cases}. \tag{2.2}$$

It is easy to see that the rows of C are codewords of C, as

$$CL^T = C_{i_0} L_{i_0}^T + C_{i_1} L_{i_1}^T = L_{i_1}^T L_{i_0}^T + L_{i_0}^T L_{i_1}^T = 0, \tag{2.3}$$

where the last equality is justified by the fact that multiplication between circulant matrices is commutative.

If an attacker succeeds in determining one of the rows of C, which are codewords of the public code C, he will be able to recover two blocks L_i, L_j of the secret parity-check matrix L. From such values, the attacker will be able to determine the value of $L_{n_0-1}^{-1}$ from either one of the public key blocks M_i, M_j, and consequently derive the full secret parity-check matrix L.

Each one of the codeword c of C has weight $2v$, and can thus be searched for by exploiting an ISD to solve the CFP with a cost $\mathsf{ISD}_{\mathsf{cfp}}(n_0 p, p, 2v)$. We note that more than a codeword with weight $2v$ is present, thus resulting in a speedup of the codeword finding attack. To quantify the number of codewords, consider that we have $\binom{n_0}{2}$ possible matrices C as in (2.2), each one containing p rows: thus, the number of codewords in C with weight $2v$ (and the subsequent speedup in ISD) is obtained as $p\binom{n_0}{2}$.

Finding low weight codewords in a smaller code C'. Consider the systematic generator matrix for C, with the form:

$$G_{\mathsf{sys}} = \left[\begin{array}{cccc|c} I_p & & & & G_0 \\ & I_p & & & G_1 \\ & & \ddots & & \vdots \\ & & & I_p & G_{n_0-2} \end{array}\right] = \left[\begin{array}{cccc|c} I_p & & & & \left(L_0 L_{n_0-1}^{-1}\right)^T \\ & I_p & & & \left(L_1 L_{n_0-1}^{-1}\right)^T \\ & & \ddots & & \vdots \\ & & & I_p & \left(L_{n_0-2} L_{n_0-1}^{-1}\right)^T \end{array}\right]. \tag{2.4}$$

Pick a a block G_i and constitute the matrix $G' = [I_p, G_i]$. G' can be thought as the generator matrix of a code C', with length $2p$ and dimension p, which, we show, contains codewords of weight $2v$. Indeed, we have:

$$L_{n_0-1}^T G' = [L_{n_0-1}^T; L_{n_0-1}^T \left(L_i L_{n_0-1}^T\right)^T] = [L_{n_0-1}^T; L_i^T]. \tag{2.5}$$

We thus have that the low weight codewords of G reveal enough information to reconstruct the secret key from the public key. It is thus possible to apply an ISD technique to solve the CFP problem with complexity $\mathsf{ISD}_{\mathsf{cfp}}(2p, p, 2v)$. Note that there are p codewords of weight $2v$ for each block G_i employed to build G', therefore resulting in a speedup for the ISD equal to $n_0 p$.

Codeword finding in the dual code $\mathsf{C}^\perp$. An attacker can consider the dual code of C, $\mathsf{C}^\perp$. A valid generator matrix for such a code is thus the matrix L which, by construction, has low weight codewords. Indeed, the rows of L have weight $n_0 v$. It is thus possible to solve the codeword finding problem on $\mathsf{C}^\perp$ with a cost $\mathsf{ISD}_{\mathsf{cfp}}(n_0 p, (n_0 - 1)p, n_0 v)$. Due to the quasi cyclic nature of L, the ISD cost will be reduced by a factor p.

We thus have that the cost of key recovery attacks taken into account in our parameter design procedure is $\min\left\{\frac{\mathsf{ISD}_{\mathsf{cfp}}(n_0 p, p, 2v)}{p\binom{n_0}{2}}, \frac{\mathsf{ISD}_{\mathsf{cfp}}(2p, p, 2v)}{n_0 p}, \frac{\mathsf{ISD}_{\mathsf{cfp}}(n_0 p, (n_0-1)p, n_0 v)}{p}\right\}$.

2.4 Attacks based on weak keys

A recent attack against LEDAcrypt primitives exploits the product structure of the parity-check matrix of the private code, i.e., $L = HQ$ to search for weak keys that make decoding easier than in the general case.

This attack procedure has been introduced in [2] and leverage the product structure of the public code parity-check matrix to make guess separately on H and Q and projecting them onto H'. This accelerates the ISD procedures with respect to a direct application of them directly on H'.

According to such an approach, one key is considered weak if occurs with probability 2^{-x}, requires the equivalent of 2^y AES operations for ISD and $x + y < \lambda$, being λ the claimed security level.

Some of such weak keys have been found for the following LEDAcrypt instances:

- $n_0 = 2$, cat. 5, Indistinguishability Under Chosen Plaintext Attack (IND-CPA): $x \approx 44$, $y \approx 52$;
- $n_0 = 4$, cat. 1, IND-CPA: $x \approx 40$, $y \approx 50$.

This attack has been tested for $n_0 = 2$, and works well when n_0 is small and the weights of H and Q are well balanced. Hence, it can be countered by increasing n_0 or choosing unbalanced weights for H and Q. In order to completely avoid attacks of this type, in the following we focus on LEDAcrypt instances with $m = 1$, thus yielding maximally unbalanced weights for H and Q. This coincides with considering those special instances of LEDAcrypt in which the matrix Q boils down to a quasi cyclic permutation matrix and, hence, can be neglected. Therefore, from this point onward, we will simply consider the case $L = HI = H$. We note that taking $Q = I$ *does not otherwise adversely* impact the one-wayness of the Niederreiter and McEliece cryptosystems as described in Section 1.2, provided that the weight of a column of $L = H$ is chosen taking $Q = I$ into account (i.e., taking $v \approx d_v \cdot m$, where d_v is the column weight of the old H matrix and m the column weight of the old Q matrix).

2.5 Attacks based on exhaustive key search

Considering the choice of obtaining the parity-check matrix H as a randomly generated, block circulant matrix made of $1 \times n_0$ blocks with size p, we note that performing exhaustive key search attacks will always be more expensive than solving the codeword finding problem via advanced ISD solvers. Indeed, picking H as a random block circulant matrix with column weight v, and row weight $w = n_0 v$, does not offer any additional possibility of enumerating its components, as opposed to obtaining it as the product of two low weight block circulant matrices. This results in a complexity of performing an exhaustive key search equal to $\binom{n}{w} \times c_{test}$, where c_{test} is the cost of testing a candidate key for correctness. We note that such a computational cost is higher than even the one required by the simplest ISD techniques. Taking as an example Lee and Brickell's ISD [47], the correct key is found in $\frac{\binom{n}{w}}{\binom{k}{x}\binom{r}{w-x}} \times c_{test}$ attempts, where x is a free positive integer parameter to be optimized.

2.6 Attacks based on the receiver's reactions

In addition to the proper sizing of the parameters of LEDAcrypt so that it withstands the aforementioned attacks, attacks performed by an active attacker should be taken into account.

The best known case of active attacks against LEDAcrypt, and, more in general, non null decoding failure probability cryptosystems are the so-called *reaction attacks*.

In these attacks, an adversary has access to a decryption oracle, to which he may pose a large amount of queries; when the scheme's DFR is non-zero, then events of decryption failures leak information about the secret key. Therefore, the secret key can be retrieved through a statistical analysis on the decryption outcome can be performed [27,28,37,38,62]. In particular, these attacks exploit the relation between the DFR of the code and the supports of the private matrices and the error vector used for encryption. Indeed, whenever e and H have pairs of ones placed at the same distance, the decoder exhibits a DFR smaller than the average one over all the possible error vectors.

Reaction attacks require the collection of the outcome of decoding (success or failure) on a ciphertext for which the attacker knows the distances between the set bits in the error vector for a significant number of ciphertexts, to achieve statistical confidence in the result. The information on the decoding status is commonly referred to as the *reaction* of the decoder, hence the name of the attack. We note that employing an appropriate construction, so that the resulting primitive enjoys IND-CCA2 guarantees provably prevents reaction attacks, as the attacker model fits the CCA2 one.

In the following we briefly recall the work of [38]. Given a binary vector a of length p, having $\Psi_a = \{p_0, p_1, \cdots, p_w\}$ as its support, i.e., the set of positions of a containing a set bit, we define its distance spectrum $DS(a)$ as:

$$DS(a) = \{\min\{|p_i - p_j|, p - |p_i - p_j|\}, \quad p_i, p_j \in \Psi_a\}. \tag{2.6}$$

For a circulant matrix C, the distance spectrum $DS(C)$ is defined as the distance spectrum of its first row, since all the rows share the same spectrum. Indeed, it can be easily shown that the cyclic shift of a vector does not change its distance spectrum. As proven in [27], it is possible to reconstruct a vector a once its distance spectrum and number of set symbols is known.

The attacker can aim at recovering the distance spectrum of ones of the circulant blocks in the secret key; for LEDAcrypt schemes, such an information is enough to recover the whole secret key from the public one. To do this, the adversary generates a large number of valid plaintext/ciphertext pairs and then queries the oracle with the produced ciphertexts which, to each received query, replies with the outcome of the corresponding decryption phase. Then, the statistical analysis of the oracle's replies (in terms of decoding success or failure) is exploited by Eve to recover the distance spectra she is interested in. When an IND-CCA2 secure conversion is adopted, the error vector used during encryption cannot be chosen by Eve, and can be seen as a randomly extracted vector among all the possible $\binom{n}{t}$ n-tuples with weight t. A critical parameter for these attacks is T, which is the number of collected ciphertexts. In fact, after observing T ciphertexts, the average number of failures observed by Eve is $T \cdot \mathrm{DFR}$, which is the basis for her statistical analysis. For LEDAcrypt instances, we consider that any key pair has a lifetime equal to $T = \mathrm{DFR}^{-1}$, which means allowing that only one decryption failure is observed by Eve during the whole lifetime of each key pair, on average.

It has been recently pointed out in [56] that some mechanisms exist to generate error vectors able to artificially increase the DFR of these systems. However, they start from the observation of at least one failure, which is unlikely when the original DFR is sufficiently low. In addition, these methods require the manipulation of error vectors, which is not feasible when an IND-CCA2 secure conversion is adopted.

2.7 Side channel attacks

Side channel attacks against LEDAcrypt may be performed exploiting either the computation time or the energy required to perform a computation as a valid side channel. Attacks exploiting the latter information leakage, or any common proxy of the energy information (e.g., electromagnetic emissions) follow the usual attack methodology, common to all asymmetric ciphers [59, 68]. In this regard, the highly parallelizable nature of the out-of-place iteration function, or the intrinsically randomized nature of the described randomized-in-place iteration function provide room for a natural *hiding in time* countermeasure, at a very limited overhead. Such an approach provides an useful complement to a masking strategy which can be employed on the Boolean and simple arithmetic computations of the decoder itself.

Concerning timing side channel attacks, the decoder iterations are amenable to a constant time implementation, and the fixed number of iterations of the decoder ensures that no obstacles are present to carry it out. While the fixed number of iterations suppresses the most evident timing side channel leakage [26] altogether, we note that more subtle timing leakages due to the number of flips performed during an iteration itself may yield equally informative content [63]. Indeed, both the number of flips, and the number of iterations can be exploited by an attacker in the same fashion the information coming from a decoding failure is. The motivation behind these attacks is in the fact that many features of the QC-LDPC/MDPC decoding phase (together with the failure probability) can be related to the number of overlapping ones between the columns of the secret parity-check matrix that are indexed by the support of the error vector. This number is indeed strictly correlated to the number of unsatisfied parity-check equations, which is the key quantity that is used in BF decoders to estimate error affected bits. In the case of QC codes, the number of overlapping ones between columns is directly related to the distance spectrum of the circulant blocks in the secret key. Thus, combining such side-channel attacks with the key reconstruction described in [38] allows for a complete key-recovery attack. To solve the aforementioned issues concerning timing attacks, a consolidated study of constant-time implementation of QC-LDPC/QC-MDPC decoders is currently present in open literature, starting with [18], and counting, amount the most recent works [23, 36].

Chapter 3

Decoders for LEDAcrypt

In this chapter we provide the rationale of the design and the description of the LEDAdecoder procedure for the three cryptographic primitives. In this design, our aim is twofold:

- Provide a *predictable-DFR, constant time implementable*, decoding strategy for the primitives LEDAcrypt-KEM and LEDAcrypt-PKC where IND-CCA2 guarantees are desired;
- provide a *low execution latency optimized* decoding strategy, with a DFR low enough to be acceptable in practical cases ($\approx 10^{-9}$) for LEDAcrypt-KEM-CPA.

The LEDAdecoder decoding strategies are derived from the classic BF-decoder, originally proposed by Gallager [33]. This choice is motivated by both the computational efficiency of this family of algorithms and the possibility of providing a reliable and simple theoretical model for the DFR, which becomes significantly involved when more complicated decoders are taken into account.

BF-decoders rely on a simple procedure which iterates the computation of a function employing the syndrome s and the parity-check matrix H. Such a function starts from an estimate, $\hat{e}$, of the sought error vector, e, and computes a new estimate of it, $\bar{e}$, from the available data.
The initial value of the error estimate $\hat{e}$, in the case of the first iteration, is set to the all-zero vector of length n. Therefore, such a function outputs an updated estimate $\bar{e}$ of the error vector, and the syndrome $\bar{s}$ of the vector obtained as the difference between the sought error vector e and $\bar{e}$, that is: $\bar{s} = H(\bar{e} \oplus e)^T$.
From now on, we will refer to such a function as an *iteration* of the BF-decoder.

The BF-decoder computes repeatedly an iteration function, providing as an input to the next execution of it an estimate of the sought error vector $\hat{e}$ and a syndrome s, with values equal to the updated error vector estimate $\bar{e}$ and the updated syndrome $\bar{s}$ computed at the end of the current execution, respectively (i.e., $\hat{e} = \bar{e}$, and $s = \bar{s}$), as well as the parity-check matrix H, until the computed value of $\bar{s}$ is the null vector, or a pre-defined number of iterations, imax, has been performed.

Different strategies to compute the iteration function can be employed. Indeed, a decoder can be designed to execute a sequence of iteration functions of different kinds. In our approach to the design of the LEDAdecoder, we will employ two iteration functions which go by the name of *in-place* and *out-of-place*, and for which we provide:

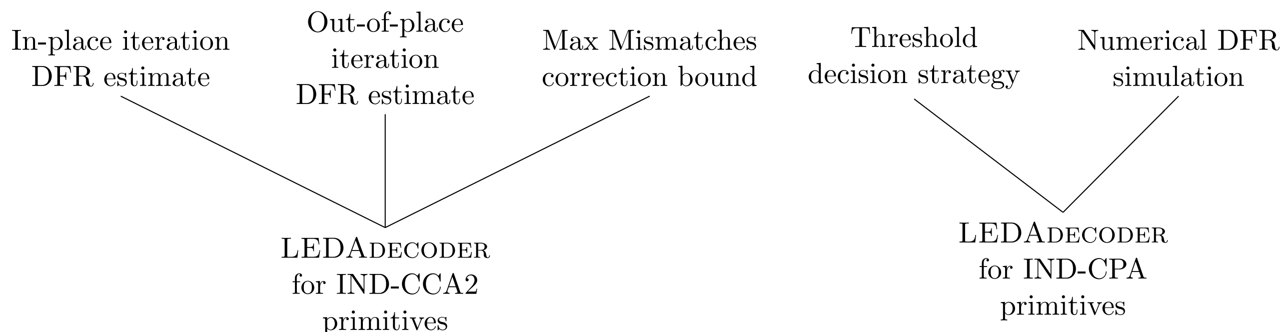


Figure 3.1: A schematic overview of the LEDADECODER design.

i. a **closed form procedure** to compute a **conservative estimate** of the number of mismatches (discrepancies) between the actual error vector e and the output of the iteration $\bar{e}$ (i.e., the Hamming weight of $e \oplus \bar{e}$: $wt(e \oplus \bar{e})$), at the end of an **in-place iteration function** computation (Section 3.2);

ii. a **closed form procedure** to compute a **conservative estimate** of the number of mismatches (discrepancies) between the actual error vector e and the output of the iteration $\bar{e}$ (i.e., the Hamming weight of $e \oplus \bar{e}$: $wt(e \oplus \bar{e})$), at the end of an **out-of-place iteration function** computation (Section 3.3);

iii. a code-specific **closed form procedure** to compute the maximum number of mismatches between e and $\bar{e}$ that can be corrected **with certainty** by a single computation of an iteration function of any of the two kinds (Section 3.4);

iv. a **threshold decision strategy** trading off an improvement in the correction capability of the out-of-place iteration function with its closed form DFR estimation (Section 3.5), which is replaced by numerical DFR simulations.

Figure 3.1 summarizes our choices in designing the LEDADECODER for both the IND-CCA2 primitives and INC-CPA primitives in LEDAcrypt.

Decoder design for IND-CCA2 LEDAcrypt-KEM and IND-CCA2 LEDAcrypt-PKC

We design a closed form predictable DFR LEDADECODER combining our ability to derive a closed form DFR estimate (Section 3.2 and Section 3.3) with the procedure to determine the maximum number of mismatches which can be corrected with a single computation of an iteration (Section 3.4). Indeed, we determine what is the probability of the decoder ending up in computing an error estimated $\bar{e}$ with a given number of discrepancies $wt(e \oplus \bar{e})$, lesser or equal to a given amount, τ, after computing all but the last iteration (employing the closed form procedures stated in i) and ii) for the in-place and out-of-place iteration functions, respectively).
The last iteration of the LEDADECODER is then chosen to be either in-place or out-of-place with the following criterion:

- An in-place iteration allows us to employ the tighter upper bound on the correction capability provided by item i), at the cost of a higher decoder latency as in-place iterations are less amenable to parallel implementations.

Algorithm 12: LEDADECODER

Input: s: QC-LDPC syndrome, binary vector of size p
`Htr`: transposed parity-check matrix, represented as an $n_0 \times v$ integer matrix containing in each row the positions (ranging in $\{0, 1, \ldots, p-1\}$) of the set coefficients in the first column of each n_0 $p \times p$ block of the circulant-block matrix $H = [H_0 \mid H_1 \mid \ldots \mid H_{n_0-1}]$

Output: e: the decoded error vector with size $n_0 p$
`decodeOk`: Boolean value denoting the successful outcome of the decoding action

Data: `imax`$^{\mathtt{In}}$: maximum number of in-place iteration function executions
`imax`$^{\mathtt{Out}}$: maximum number of out-of-place iteration function executions

```
1  ê ← 0_n                        // Error vector initialized as the null length-n vector
2  for iter^In ← 0 to imax^In − 1 do
3      if s = 0 then break // removed in constant time implementations
4      {ê, s} ← RandomizedInPlaceIteration(ê, Htr, s, iter^In)
5  for iter^Out ← 0 to imax^Out − 1 do
6      if s = 0 then break // removed in constant time implementations
7      {ê, s} ← OutOfPlaceIteration(ê, Htr, s, iter^Out)
8  if s = 0 then
9      return ê, true
10 return ê, false
```

- An out-of-place iteration allows us to obtain an efficiently parallelizable implementation, at the cost of some correction power as the out-of-place strategy is affected by lower error correction performances w.r.t the in-place one. In this case, we employ the bound from item `iii`) to obtain the amount of mismatches τ which the last iteration will correct with certainty at code (i.e., keypair) generation time. We design code parameters such that during the key-generation procedure the rejection ratio of the codes is below 50%.

Decoder design for LEDAcrypt-KEM-CPA

The LEDADECODER design for IND-CPA primitives aimed at an ephemeral-key use scenario aims at obtaining the most compact key pairs and the fastest decoding strategy (among the ones we propose), exploiting the advantage of requiring only a practically usable DFR (i.e., DFR$= 10^{-9}$), which can be determined via numeric simulations.
We therefore design a variable iteration number decoder, where the maximum number of iterations is capped experimentally to the highest one obtained in the numerical simulations. This decoder employs only out-of-place iterations to minimize the decoder latency, and exploits the syndrome-weight dependent threshold determination rule (item `iv`)) described in Section 3.5. Such a threshold selection criterion is experimentally determined to attain a higher correction capability w.r.t. a fixed threshold choice depending only on the number of computed iterations. This, in turn, allows to further reduce the key size, improving the computation performance.

LEDAcrypt Decoder

Algorithm 12 provides a procedural description of the LEDADECODER stub, fitting both the case of the closed-form predictable DFR decoder (for LEDAcrypt-KEM and LEDAcrypt-PKC), and the case of the computation performance optimized decoder (for LEDAcrypt-KEM-CPA).
The decoder structure is simple: it first computes `imax`$^{\mathtt{In}}$ in-place iteration functions, followed by `imax`$^{\mathtt{Out}}$ out-of-place ones. If constant time implementation is not required (e.g., in the LEDAcrypt-

Table 3.1: Summary of the LEDAdecoder instances depending on the LEDAcrypt primitive

	LEDAcrypt-KEM-CPA	**LEDAcrypt-KEM & -PKC**
$(\mathtt{imax}^{\mathtt{In}}, \mathtt{imax}^{\mathtt{Out}})$	(0, max among simulated)	$\mathtt{imax}^{\mathtt{In}} + \mathtt{imax}^{\mathtt{Out}} = 2$
Threshold selection	function of syndrome weight	fixed per iteration
Obtained DFR	10^{-9}	2^{-64} or $2^{-\lambda}$, $\lambda \in \{128, 192, 256\}$

KEM-CPA case) the early exits after an iteration (lines 3 and 6) can be employed to obtain a faster execution. By contrast, in constant time implementations all the iterations are always computed – in practice not altering the value of $\hat{e}$ during their computation.
Note that Algorithm 12 returns the value of $\hat{e}$ deeming it correct if the value of s is null.
Indeed, there is a probability of the decoder ending up in providing an error vector $\bar{e}$ which does not match the actual one e, even if the syndrome is null. We take into account these failure cases too in our analysis, as we monitor the number of discrepancies between $\bar{e}$ and e, and deem a decoding successful if and only if there are none.
Finally, we provide a quick summary of how the appropriate choices of $\mathtt{imax}^{\mathtt{In}}$, $\mathtt{imax}^{\mathtt{Out}}$ and the bit-flipping thresholds result in the LEDAdecoder variant for IND-CPA or IND-CCA2 primitives in Table 3.1.

In particular, in addition to the choices which have been described, we pick the total number of iterations for the LEDAdecoder employed in IND-CCA2 primitives to be equal to 2, as it provides a good engineering tradeoff between key size and execution speed and makes the primitives amenable to a constant time implementation.

Conventions adopted in this chapter

In the remainder of this chapter, we will describe in detail the contributions i)–iv). To this end, we recall from Section 2.4 that from the specification reported in this manuscript on, the parity-check matrix of LEDAcrypt secret code, $L = HQ$, is chosen taking $Q = I$; thus, the Q matrix is neglected for all practical purposes. As a consequence, the LEDAdecoder procedure is expected to recover the value of an error vector e of weight t, given the secret parity-check matrix H and a syndrome s of the error e through H, that is, $e \leftarrow \text{LEDAdecoder}(H, s)$.

In particular, during the Niederreiter-based decryption procedure of LEDAcrypt-KEM (see Figure 1.3(a) in Section 1.2), s is identified with the private syndrome s', obtained multiplying the (public) syndrome contained in the ciphertext by the last block of $L = H$, $L_{n_0-1} = H_{n_0-1}$, $s' = H_{n_0-1}s$. During the decryption of LEDAcrypt-PKC, s corresponds to the syndrome of the corrupted codeword c, contained in the ciphertext, computed through the secret parity-check matrix.

3.1 Preliminary analyses

In the following we describe the assumptions and derivations at the core of our statistical model for the behaviour of the iteration functions employed in the LEDADECODER.

The set of simultaneous parity-check equations, in the unknown error vector e with n_0p bits and weight t, tackled by the decoder on an input parity-check matrix H, with p rows and n_0p columns, and a syndrome s with length p, is captured by the following relation $He^T = s^T$.
To the end of finding the bit-values of the unknown error vector the decoder starts taking an estimated error vector $\hat{e}$ with all bits set to zero and applies a sequence of iteration functions, each of which will evaluate whether or not to flip (i.e., change) the j-th bit of $\hat{e}$, $\hat{e}_j$, if the corresponding number of unsatisfied parity-check equations (upc_j) exceeds a predefined threshold th, updating the syndrome value accordingly. The decoder will stop when the updated syndrome value is equal to zero. Note that the parity-check equations (i.e., the rows in the $H\hat{e}^T = s^T$ set of simultaneous equations) influenced by the j-th bit of the estimated error vector are the ones having an asserted bit in their j-th column position, and the subset of unsatisfied parity-check equations includes the ones having a constant term that is also asserted (as it coincides with the non-null value in the corresponding bit-position of the syndrome), therefore the value of any upc_j can be computed summing $s_i \cdot h_{i,j}$ for all the equations, that is $0 \leq i \leq p-1$. The distinguishing point between an in-place and an out-of-place iteration function lies on when the update of the syndrome value is executed. In the former iteration function, the syndrome is updated just after each test establishing if the j-th bit value in the estimated error vector (i.e., $\hat{e}_j$) should be flipped or not (i.e., if $\mathsf{upc}_j \geq \mathsf{th}$ or not). In the latter iteration function, the syndrome is updated after the test and possible flip of all the bits of the estimated error vector have been performed.

The analyses of the in-place and of the out-of-place iteration functions, reported in the next subsections, are all based on the following statements and Lemma.

Assumption 3.1.1 *The probability* $\mathrm{P}^{\mathsf{th}}_{f|1} = \Pr\left[\mathsf{upc}_j \geq \mathsf{th} \mid e_j \neq \hat{e}_j\right]$*, with* $j \in \{0, 1, \ldots, n-1\}$*, that the number of the unsatis ed parity checks involving the* j-th *bit of the error vector, i.e.,* upc_j*, is large enough to trigger a ip of* $\hat{e}_j$*, given that its current value does not match the value of the* j-th *bit in the unknown error vector, i.e.,* $\tilde{e}_j = e_j \oplus \hat{e}_j = 1$*, is a function of only the total number* $\hat{t} = wt\,(\hat{e} \oplus e)$ *of positions over which the estimated error vector* $\hat{e}$ *and the unknown error vector* e *di er.*
Analogously, the probability $\mathrm{P}^{\mathsf{th}}_{m|0} = \Pr\left[\mathsf{upc}_j < \mathsf{th} \mid e_j = \hat{e}_j\right]$ *that the number of the unsatis ed parity checks involving the* j-th *bit of the error vector, i.e.,* upc_j*, is low enough to maintain the current value* $\hat{e}_j$ *of the* j-th *estimated error vector bit, given that its current value matches the value of the* j-th *bit in the unknown error vector, i.e.,* $\tilde{e}_j = e_j \oplus \hat{e}_j = 0$*, is a function of only the total number* $\hat{t} = wt\,(\hat{e} \oplus e)$ *of positions over which the estimated error vector* $\hat{e}$ *and the unknown error vector* e *di er.*

Informally, we are stating that the statistical behaviour of the single given upc_j does not depend on its location j, but only on the number of discrepancies between the estimated error vector and the actual one, and the fact that the j-th position of $\hat{e}$ is in accordance or not with e.

The following probabilities referred to flipping (f) or maintaining (m) the value of each bit of $\hat{e}$

will be used to characterize the iteration behaviour

$$
\begin{aligned}
\mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}) &= \Pr\left[\mathsf{upc}_j \geq \mathsf{th} \mid e_j \neq \hat{e}_j,\ wt\left(e \oplus \hat{e}\right) = \hat{t}\right],\\
\mathrm{P}^{\mathsf{th}}_{m|1}(\hat{t}) = 1 - \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}) &= \Pr\left[\mathsf{upc}_j < \mathsf{th} \mid e_j \neq \hat{e}_j,\ wt\left(e \oplus \hat{e}\right) = \hat{t}\right],\\
\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}) &= \Pr\left[\mathsf{upc}_j < \mathsf{th} \mid e_j = \hat{e}_j,\ wt\left(e \oplus \hat{e}\right) = \hat{t}\right],\\
\mathrm{P}^{\mathsf{th}}_{f|0}(\hat{t}) = 1 - \mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}) &= \Pr\left[\mathsf{upc}_j \geq \mathsf{th} \mid e_j = \hat{e}_j,\ wt\left(e \oplus \hat{e}\right) = \hat{t}\right].
\end{aligned}
\tag{3.1}
$$

To derive closed form expressions for both $\mathrm{P}^{\mathsf{th}}_{f|1}$ and $\mathrm{P}^{\mathsf{th}}_{m|0}$, we remember that the parity-check matrices employed as secret keys in LEDAcrypt have column weight $v = d_v$ and row weight $w = n_0 v = n_0 d_v$.

According to this, in the following we assume that each row of the parity-check matrix H is independent of the others and modeled as a sample of a uniform random variable, distributed over all possible sequences of n bits with weight w, and name a parity-check matrix (v, w)-regular if all its columns have weight v and all its rows have weight w. We share this assumption with a significant amount of literature on the prediction of the DFR of QC-LDPC decoders, ranging from the original work by Gallager on LDPCs [32, Section 4.2] to more recent ones, namely [71, Section 3] and [61, Section 4].

Formally, the following statement is assumed to hold:

Assumption 3.1.2 *Let H be a $p \times n_0 p$ quasi-cyclic block-circulant (v, w)-regular parity-check matrix and let s be the $1 \times p$ syndrome corresponding to a $1 \times n_0 p$ error vector $\tilde{e} = e \oplus \hat{e}$ that is modeled as a sample from a uniform random variable distributed over the elements in $\mathbb{F}_2^{1 \times n_0 p}$ with weight $\hat{t}$. We assume that each row $h_{i,:}$, $0 \leq i \leq p-1$, of the parity-check matrix H is well modeled as a sample from a uniform random variable distributed over the elements of $\mathbb{F}_2^{1 \times n_0 p}$ with weight w.*

Note that the assumption on the fact that the syndrome at hand is obtained from a vector $\tilde{e} = \hat{e} \oplus e$ of weight $\hat{t}$ is trivially true if the iteration of the decoder being considered is the first one being computed, since the error estimate $\hat{e}$ is null and the error vector e is drawn at random with weight $t = \hat{t}$. This in turn states that, when employing Assumption 3.1.2 in estimating the correction capability of the first iteration of a decoder, we are only relying on the fact that the rows of the matrix H can be considered independent, neglecting the effects of the quasi-cyclic structure.

In the following Lemma we establish how, upon relying on the previous assumption, the probabilities that characterize the choices on the bits of the estimated error vector, made by a either an in-place or an out-of-place iteration function, can be expressed.

Lemma 3.1.1 Let H be a $p \times n_0 p$ quasi-cyclic block-circulant (v, w)-regular parity-check matrix; let $\tilde{e} = \hat{e} \oplus e$ be an unknown vector of length n and weight $\hat{t}$ such that $H(\tilde{e})^T = s$.
From Assumption 3.1.1 and Assumption 3.1.2, the probabilities $\rho_{0,\mathsf{u}}(\hat{t}) = \Pr\left[s_i = 1 | \tilde{e}_z = 0\right]$ and $\rho_{1,\mathsf{u}}(\hat{t}) = \Pr\left[s_i = 1 | \tilde{e}_z = 1\right]$ that the i-th parity-check equation having $h_{i,z} = 1$, for any $0 \leq z \leq n-1$, is unsatisfied (i.e., $s_i = h_{i,:}(\tilde{e})^T = 1$) given the value of $\tilde{e}_z$, can be derived as follows

$$
\rho_{0,\mathsf{u}}(\hat{t}) = \Pr\left[h_{i,:}(\tilde{e})^T = 1 \mid \tilde{e}_z = 0\right] = \frac{\sum_{l=1,\ l \text{ odd}}^{\min\{w-1,\hat{t}\}} \binom{w-1}{l}\binom{n_0 p - w}{\hat{t}-l}}{\binom{n_0 p - 1}{\hat{t}}}
$$

$$\rho_{1,\mathsf{u}}(\hat{t}) = \Pr\left[h_{i,:}(\tilde{e})^T = 1 \mid \tilde{e}_z = 1\right] = \frac{\sum_{l=0,\ l \text{ even}}^{\min\{w-1,\hat{t}-1\}} \binom{w-1}{l}\binom{n_0p-w}{\hat{t}-1-l}}{\binom{n_0p-1}{\hat{t}-1}}$$

Consequently, the probability $\mathrm{P}_{f|1}^{\mathsf{th}}(\hat{t}) = \Pr\left[\mathsf{upc}_z \geq \mathsf{th} \mid \tilde{e}_z = \hat{e}_z \oplus e_z = 1\right]$ of changing (flipping) the z-th bit of the estimated error vector $\hat{e}_z$ assuming that $\tilde{e}_z = 1$, and the probability $\mathrm{P}_{m|0}^{\mathsf{th}}(\hat{t}) = \Pr\left[\mathsf{upc}_z < \mathsf{th} \mid \tilde{e}_z = \hat{e}_z \oplus e_z = 0\right]$ of maintaining $\hat{e}_z$ assuming that $\tilde{e}_z = 0$, are computed as follows

$$\mathrm{P}_{f|1}^{\mathsf{th}}(\hat{t}) = \sum_{\sigma=\mathsf{th}}^{v} \binom{v}{\sigma} \left(\rho_{1,\mathsf{u}}(\hat{t})\right)^{\sigma} \left(1 - \rho_{1,\mathsf{u}}(\hat{t})\right)^{v-\sigma},$$

$$\mathrm{P}_{m|0}^{\mathsf{th}}(\hat{t}) = \sum_{\sigma=0}^{\mathsf{th}-1} \binom{v}{\sigma} \left(\rho_{0,\mathsf{u}}(\hat{t})\right)^{\sigma} \left(1 - \rho_{0,\mathsf{u}}(\hat{t})\right)^{v-\sigma}.$$

Proof. For the sake of brevity, we consider the case of $\tilde{e}_z = 1$ deriving the expression of $\mathrm{P}_{f|1}^{\mathsf{th}}(\hat{t})$; the proof for $\mathrm{P}_{m|0}^{\mathsf{th}}(\hat{t})$ can be carried out with similar arguments. Given a row $h_{i,:}$ of the parity-check matrix H, such that $z \in \mathrm{Supp}(h_{i,:})$, the equation $\bigoplus_{j=0}^{n_0p-1} h_{i,j}\tilde{e}_j$ (in the unknown $\tilde{e}$) yields a non-null value for the i-th bit of the syndrome, $\hat{s}_i$, (i.e., the equation is unsatisfied) if and only if the support of $\tilde{e}$ is such that $\bigoplus_{j=0}^{n_0p-1} h_{i,j}\tilde{e}_j = 2a+1, a \geq 0$, including the term having $j = z$, i.e., $h_{i,z}\tilde{e}_z = 1$. This implies that the cardinality of the set obtained intersecting the support of $h_{i,:}$ with the one of $\tilde{e}$, $|(\mathrm{Supp}(h_{i,:}) \setminus \{z\}) \cap (\mathrm{Supp}(\tilde{e}) \setminus \{z\})|$, must be an even number, which in turn cannot be larger than the minimum between $|\mathrm{Supp}(h_{i,:}) \setminus \{i\}| = w-1$ and $|\mathrm{Supp}(\tilde{e}) \setminus \{i\}| = \hat{t}-1$.

The probability $\rho_{1,\mathsf{u}}(\hat{t})$ is obtained considering the fraction of the number of values of $\tilde{e}$ having an even number of asserted bits matching the asserted bits ones in a row of H (noting that, for the z-th bit position, both the error and the row of H are set) on the number of $\tilde{e}$ values having $\hat{t}-1$ asserted bits over n_0p-1 positions, i.e., $\binom{n_0p-1}{\hat{t}-1}$. The numerator of the said fraction is easily computed as the sum of all $\tilde{e}$ configurations having an even number $0 \leq l \leq \min\{w-1, \hat{t}-1\}$ of asserted bits. Considering a given value for l, the counting of $\tilde{e}$ values is derived as follows. Picking one vector with l asserted bits over w possible positions, i.e., one vector over $\binom{w-1}{l}$ possible ones, there are $\binom{n_0p-w}{\hat{t}-1-l}$ possible values of the error vector exhibiting $\hat{t}-1-l$ null bits in the remaining n_0p-w positions; therefore, the total number of vectors with weigh l is $\binom{w-1}{l}\binom{n_0p-w}{\hat{t}-1-l}$.
Repeating the same line of reasoning for each value of l allows to derive the numerator of the formula defining $\rho_{1,\mathsf{u}}(\hat{t})$.

From Assumption 3.1.2, the observation of any unsatisfied parity check involving the z-th bit of the error vector $\tilde{e}_z$ (i.e., $h_{i,:}(\tilde{e})^T = 1$) given that $\tilde{e}_z = \hat{e}_z \oplus e_z = 1$, is modeled as a random variable with a Bernoulli distribution having parameter (or expected value) $\rho_{1,\mathsf{u}}(\hat{t})$, and each of these random variables is independent of the others. Consequentially, the probability that the decoder performs a bit-flip of an element of the estimated error vector when the corresponding bit of the error vector is asserted and the counter of the unsatisfied parity checks (upc) is above or equal to a given threshold th, is derived as the binomial probability obtained adding the outcomes of v (column-weight of H) i.i.d. Bernoulli trials. ■

Algorithm 13: RandomizedInPlaceIteration

```
Input: s: QC-LDPC syndrome, binary vector of size p
       Htr: transposed parity-check matrix, represented as an n0 × v integer matrix
            containing in each row the positions (ranging in {0, 1, ..., p − 1}) of the set
            coefficients in the first column of each n0 p × p block of the
            circulant-block matrix H = [H_0 | H_1 | ... | H_{n0−1}]
       ê: initial error vector estimate, binary vector of size n = n0 p
       iter^In: number of iterations computed before
Output: ē: updated error vector estimate
        s: updated syndrome
1  π <-$- S_n                                    // Random permutation of size n = n0 p
2  J ← π(⟨0, 1, ··· , n − 1⟩)                    // Permutation of the sequence ⟨0, 1, ··· , n − 1⟩
3  th ← ComputeThreshold(iter^In)
4  for j ∈ J do
5      i ← ⌊j/p⌋, offset ← j mod p               // Circulant block index & offset j = i·p + offset
6      upc_j ← 0                                  // Counter of unsatisfied parity checks
7      for k = 0 to v − 1 do
8          if getSyndromeBit(s, (Htr[i][k] + offset) mod p) = 1 then
9              upc_j ← upc_j + 1
10     if upc_j ≥ th then
11         ē_j ← ê_j ⊕ 1                          // Error vector update
12         for k = 0 to v − 1 do
13             idx ← (Htr[i][k] + offset) mod p
14             s_idx ← s_idx ⊕ 1                  // Syndrome update
15     else
16         ē_j ← ê_j
17 return {ē, s}
```

3.2 Residual distribution of the mis-matches between the sought error vector and the input estimated one after the execution of a single in-place iteration function

We consider the in-place iteration function described in Algorithm 13, which takes as input the p-bit syndrome s, the initial n-bit error vector estimate $\hat{e}$, and the transposed parity-check matrix of the code represented, for performance reasons, as an $n_0 \times v$ integer matrix containing in each row the positions (ranging in $\{0, 1, \dots, p-1\}$) of the set coefficients in the first column of each n_0 $p \times p$ block of the circulant-block matrix $H = [H_0 \mid H_1 \mid \dots \mid H_{n_0-1}]$. The algorithms outputs the updated n-bit error vector estimate, $\bar{e}$, and the corresponding updated syndrome. In the following analysis, the number of mismatches between the error estimate $\hat{e}$ and the sought error vector e that are going to be corrected is denoted as $\hat{t} = wt(\hat{e} \oplus e)$, while the (eXclusive-OR) difference between e and $\hat{e}$ is denoted as $\tilde{e} = \hat{e} \oplus e$ (thus, $\hat{t} = wt(\tilde{e})$).
It is worth noting that, if the computation of Algorithm 13 corresponds to the first iteration performed by the LEDAdecoder (that is, $\mathtt{iter}^{\mathtt{In}} = 0$ in Algorithm 12) then $\hat{t} = t$ since $\hat{e}$ is the null vector of length $n_0 p$, and there are exactly t mismatches with e.

Our aim in this section is to provide a way to compute the probability of $\hat{t}$ taking a given value at the end of the in-place iteration performed by Algorithm 13. Note that the probability of $\hat{t}$ being

zero is the probability that the iteration completes correctly the error estimation, i.e., that the algorithm of the in-place iteration function succeeds in retrieving e at the end of its computation.

Algorithm 13 starts by drawing a random permutation π and applying it to the sequence of integer indexes $\{0, 1, \ldots, n_0p - 1\}$ (line 1–2), then it continues by determining the flipping threshold according to the number of the iteration being computed, by means of a lookup (line 3).
The outer loop of the algorithm at lines 4–16 iterates over the entries of $\hat{e}$, following the order given by the permuted index sequence J.
Each execution of the loop body computes the number of unsatisfied parity-check equations, $\mathsf{upc_j}$, to which the bit $\tilde{e}$ contributes (lines 5–9). If $\mathsf{upc_j}$ is above or equals the chosen threshold, th, (line 10) we obtain the output estimate $\bar{e}_j$ by flipping $\hat{e}_j$ (line 11), while the output syndrome is obtained updating the current one accordingly (lines 12–14). If $\mathsf{upc_j}$ is below the chosen threshold, the output estimate $\bar{e}_j$ remains equal to $\hat{e}_j$, while also the syndrome is not modified. Note that, as j is supposed to denote a column index of the $p \times n_0p$ matrix H, the algorithm needs to compute both the block index, $\mathsf{i} \leftarrow \lfloor \mathsf{j}/p \rfloor$, and the offset, offset, of the position of the column at hand ($\mathsf{j} = \mathsf{i} \cdot p + \mathsf{offset}$) to correctly determine the v positions of the set coefficients on the j-th column of H as: $(\mathsf{offset} + \mathsf{Htr}[\mathsf{i}][\mathsf{k}]) \bmod p$, for all k denoting the position of a set coefficient along the first column of the block i.

Using the probabilities $\mathrm{P}^{\mathsf{th}}_{f|1}$ and $\mathrm{P}^{\mathsf{th}}_{m|0}$, we derive a statistical model for one in-place iteration as described in Algorithm 13. In particular, we consider a *worst-case* evolution of the iteration from the correction standpoint, i.e., the computation path leading to a decoding success at the end of the iteration with the lowest probability. Indeed, the success probability of the in-place decoder also depends on the order in which the $\hat{e}$ bits are processed.

To describe the worst-case computation path (from a correction standpoint), we consider a subset of the permutations π picked at the beginning of the iteration (line 1 of Algorithm 13).
Let $S_n^* \subseteq S_n$ be the set of all permutations $\pi^* \in S_n^*$ such that[1]

$$\mathrm{Supp}(\pi^*(e) \oplus \pi^*(\hat{e})) = \{n - \hat{t}, n - \hat{t} + 1, \cdots, n - 1\}, \ \forall \pi^* \in S_n^*,$$

that is the set of permutations which cause all the positions having discrepancies between e and $\hat{e}$ to be analyzed last by the iteration.

We denote with $\Pr[\bar{e} \neq e | \ \pi \in S_n]$ the probability that the estimated error vector provided as output by the RandomizedInPlaceIteration algorithm is different from e, conditioned by the fact that the permutation π was picked. Similarly, we define $\Pr[\bar{e} \neq e | \ \pi^* \in S_n^*]$.

It can be verified that $\mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}) \geq \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}+1)$ and $\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}) \geq \mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}+1)$, $\forall \hat{t}$, as increasing the number of current mis-estimated error bits, increases the likelihood of a wrong decoder decision. By leveraging Assumption 3.1.1, we now prove that the in-place iteration function reaches a correct decoding at the end of the outer loop with the minimum probability each time a $\pi^* \in S_n^*$ is applied to the sequence of indices.

Lemma 3.2.1 Let s and $\hat{e}$, with $s = H(e \oplus \hat{e})^T$ and $\hat{t} = wt\,(e \oplus \hat{e})$ be the input of Algorithm 13 (RandomizedInPlaceIteration), and let $\bar{e}$ be the error vector estimate provided as output. Then

$$\forall \pi \in S_n, \forall \pi^* \in S_n^*, \quad \Pr[\bar{e} \neq e | \ \pi \in S_n] \leq \Pr[\bar{e} \neq e | \ \pi^* \in S_n^*].$$

[1] The notation Supp($\mathtt{w}$), where $\mathtt{w}$ is a vector, denotes the set of indexes(positions) corresponding to a non-null component of $\mathtt{w}$

Proof. For the sake of readability, we rewrite $\Pr[\bar{e} \neq e|\ \pi \in S_n]$ as $1 - \beta(\pi)$, where $\beta(\pi)$ is the probability that all the positions of $\hat{e}$, evaluated in the order specified by π, are correctly flipped or not, so that $\bar{e} = e$. To visualize the effect of a permutation $\pi^* \in S_n^*$ on the discrepancies between $\hat{e}$ and e, we can consider the following representation

$$\forall \pi^* \in S_n^*, \pi^*(e \oplus \hat{e}) = [\underbrace{0, 0, \cdots, 0}_{\text{length } n-\hat{t}}, \underbrace{1, 1, \cdots, 1}_{\text{length } \hat{t}}].$$

The in place iteration will thus analyze first a run of $n - \hat{t}$ positions where the difference vector $\tilde{e}$ between the permuted error $\pi^*(e)$ vector and $\pi^*(\hat{e})$ contains only zeroes, followed by a run of $\hat{t}$ positions containing only ones. The expression of $\beta(\pi^*)$ can be derived, thanks to Assumption 3.1.1, to be the following

$$\beta(\pi^*) = \left(\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right)^{n-\hat{t}} \cdot \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}) \cdot \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-1) \cdots \mathrm{P}^{\mathsf{th}}_{f|1}(1)$$

To demonstrate this expression, note that the elements in the first $n - \hat{t}$ positions of $\pi^*(e)$ and $\pi^*(\hat{e})$ match; therefore, the decoder takes the correct decision if it does not change the value of $\pi^*(\hat{e})$. Then, if a sequence of $n - \hat{t}$ correct decisions are taken in the first $n - \hat{t}$ evaluations, $\hat{t}$ will not change, thus each decision will be correct with probability $\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})$. This leads to a probability of taking the first $n - \hat{t}$ decisions equal to $\left(\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right)^{n-\hat{t}}$. Through an analogous line of reasoning, observe that the decoder will need to flip the value of the last $\hat{t}$ of $\hat{e}$ to complete the iteration with no residual discrepancies. Therefore, at each evaluation during the computation path we are interested in computing the probability, the value of $\hat{t}$ will decrease by one unit, yielding the remaining part of the expression.

Consider now a generic permutation π, such that the resulting $\pi(e \oplus e')$ has support $\{u_0, \cdots, u_{t'-1}\}$ Visually, the permuted discrepancies vector $\pi(e \oplus \hat{e})$ appears as

$$\forall \pi \in S_n, \pi(e \oplus \hat{e}) = [\underbrace{0, 0, \cdots, 0}_{u_0}, 1, \underbrace{0, 0, \cdots, 0}_{u_1 - u_0 - 1}, 1, \underbrace{0, 0, \cdots, 0}_{u_2 - (u_1 + u_0 + 2)}, 1, \ldots, 1, \underbrace{0, 0, \cdots, 0}_{n-1-u_{\hat{t}-1}}].$$

We thus derive the expression of $\beta(\pi)$ as

$$\begin{aligned}
\beta(\pi) &= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_0} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}) \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}-1)\right]^{u_1-u_0-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-1) \cdots \mathrm{P}^{\mathsf{th}}_{f|1}(1) \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \\
&= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(t)\right]^{u_0} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \prod_{j=1}^{\hat{t}-1} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}-j)\right]^{u_j-u_{j-1}-1} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l).
\end{aligned}$$

We now show that we always have $\beta(\pi) \geq \beta(\pi^*)$. First of all, we observe that $\mathrm{P}^{\mathsf{th}}_{m|0}(0) = 1$ since the decoder, in case $\hat{t} = 0$, will act on a null syndrome and thus will always maintain the same value for the elements of $\hat{e}$. Then, exploiting the monotonicity of $\mathrm{P}^{\mathsf{th}}_{m|0}$ and $\mathrm{P}^{\mathsf{th}}_{f|1}$, the following chain of

inequalities can be derived

$$\begin{aligned}
\beta(\pi) &= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_0} \prod_{j=1}^{\hat{t}-1} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t}-j)\right]^{u_j-u_{j-1}-1} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l) \\
&\geq \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_0} \prod_{j=1}^{\hat{t}-1} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_j-u_{j-1}-1} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l) \\
&= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_0} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_{\hat{t}-1}-u_0-(\hat{t}-1)} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l)) \\
&= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(0)\right]^{n-1-u_{\hat{t}-1}} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_{\hat{t}-1}-(\hat{t}-1)} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l) \\
&\geq \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{n-1-u_{\hat{t}-1}} \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{u_{\hat{t}-1}-(\hat{t}-1)} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l) \\
&= \left[\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right]^{n-\hat{t}} \prod_{l=0}^{\hat{t}-1} \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t}-l) = \beta(\pi^*).
\end{aligned}$$

We therefore have that

$$\Pr\left[\bar{e} \neq e |\ \pi \in S_n\right] = \quad 1-\beta(\pi) \leq 1-\beta(\pi^*) \quad = \Pr\left[\bar{e} \neq e |\ \pi^* \in S^*_n\right],$$

which concludes the proof. ■

Let us define the following two sets: $E_1 = \mathrm{Supp}(e \oplus \hat{e})$, and $E_0 = \{0, \ldots, n-1\} \setminus \mathrm{Supp}(e \oplus \hat{e})$, and consider the following probabilities:

i. $\Pr_{S^*_n}\left[\omega \xrightarrow{E_0} x\right]$: defined as the probability that the execution of one RandomizedInPlaceIteration acting on a syndrome s and an error vector estimate $\hat{e}$ such that $wt\left(e \oplus \hat{e}\right) = \omega$, in the order specified by a worst case permutation $\pi^* \in S^*_n$, outputs an error estimate $\bar{e}$ such that $|\mathrm{Supp}(e \oplus \bar{e}) \cap E_0| = x$. Note that, in this case $0 \leq x \leq n_0 p - \omega$;

ii. $\Pr_{S^*_n}\left[\omega \xrightarrow{E_1} x\right]$: defined as the probability that the execution of one RandomizedInPlaceIteration acting on a syndrome s and an error vector estimate $\hat{e}$ such that $wt\left(e \oplus \hat{e}\right) = \omega$, in the order specified by a worst case permutation $\pi^* \in S^*_n$, outputs an error estimate $\bar{e}$ such that $|\mathrm{Supp}(e \oplus \bar{e}) \cap E_1| = x$. Note that, in this case $0 \leq x \leq \omega$;

iii. $\Pr_{S^*_n}\left[\omega \underset{1}{\rightarrow} x\right]$: defined as the probability that the execution of one RandomizedInPlaceIteration (denoted by the 1 subscript to the arrow) acting on a syndrome s and an error vector estimate $\hat{e}$ such that $wt\left(e \oplus \hat{e}\right) = \omega$, in the order specified by a worst case permutation $\pi^* \in S^*_n$, outputs an error estimate $\bar{e}$ such that $wt\left(e \oplus \bar{e}\right) = x$.

The expressions of the probabilities i. and ii. are derived in Appendix A, and only depend on the probabilities $\mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t})$ and $\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})$. Given those, we obtain probability iii. as:

$$\text{for } \omega \geq x,\ \Pr_{S^*_n}\left[\omega \underset{1}{\rightarrow} x\right] = \sum_{\delta=\max\{0\ ;\ x-(n-\omega)\}}^{\omega} \Pr_{S^*_n}\left[\omega \xrightarrow{E_0} x-\delta\right] \Pr_{S^*_n}\left[\omega \xrightarrow{E_1} \delta\right]. \tag{3.2}$$

Being able to compute any $\Pr_{S_n^*}\left[\omega \xrightarrow[1]{} x\right]$ of our choice allows us to extend our analysis to any number of consecutive calls to the RandomizedInPlaceIteration function.

To clarify this point, denote with a $\cdot^{(i)}$ superscript the fact that a given value pertains to the i-th consecutive execution of Algorithm 13 after the first (i.e., $i \geq 1$). For the first iteration (i.e., $i = 0$), the number of discrepancies between e and $\hat{e}$ just before the execution of the RandomizedInPlaceIteration function is expressed as: $\hat{t}^{(i-1)} = t$.

Since a different permutation π is applied at the beginning of each execution of Algorithm 13, we have that the worst-case computation path (from a correction capability standpoint) is the one where a permutation among the ones packing the discrepancies between e and $\hat{e}^{(i)}$, i.e., $\pi \in \mathsf{S}_n^*$, is taken at each iteration. Each of these permutations will pack the $\hat{t}^{(i)}$ discrepancies in the positions which will be considered last by the RandomizedInPlaceIteration function.
Clearly, $\hat{t}^{(i)}$ also expresses the weight of the error vector corresponding to the syndrome which is given as input to the $(i + 1)$-th iteration.

We denote with $\Pr_{S_n^*}\left[t \xrightarrow[\mathsf{imax}^{\mathsf{In}}]{} x\right]$ the probability that the number of differences between $\bar{e}^{(\mathsf{imax}^{\mathsf{In}}-1)}$ and e is equal to x, after a maximum of $\mathsf{imax}^{\mathsf{In}} \geq 1$ consecutive calls to the RandomizedInPlaceIteration function (each call labeled with $\mathsf{iter}^{\mathsf{In}} = 0$, $\mathsf{iter}^{\mathsf{In}} = 1$, $\mathsf{iter}^{\mathsf{In}} = 2, \ldots, \mathsf{iter}^{\mathsf{In}} = \mathsf{imax}^{\mathsf{In}} - 1$, respectively – see Algorithm 12 at lines 2–4).

Then, by considering all possible configurations of the residual errors $\hat{t}^{(i)}$, we have

$$\Pr_{S_n^*}\left[t \xrightarrow[\mathsf{imax}^{\mathsf{In}}]{} x\right] = \sum_{\hat{t}^{(0)}=0}^{n} \cdots \sum_{\hat{t}^{(\mathsf{imax}^{\mathsf{In}}-1)}=0}^{n} \prod_{j=0}^{\mathsf{imax}^{\mathsf{In}}-1} \Pr_{S_n^*}\left[\hat{t}^{(j-1)} \xrightarrow[1]{} \hat{t}^{(j)}\right].$$

The above formula is very simple and, essentially, it takes into account all possible transitions starting from an initial number of residual errors equal to t and ending with a final a number of residual errors equal to x.

This result allows us to determine what is the probability of leaving at most x discrepancies between the error estimate output by the last in-place iteration function out of a sequence of $\mathsf{imax}^{\mathsf{In}}$ ones. We will exploit this result,in two different ways for the LEDAcrypt IND-CCA2 primitives:

i. to provide an upper bound on the DFR for a decoder having $\mathsf{imax}^{\mathsf{Out}} = 0$, simply estimating the probability that $\mathsf{imax}^{\mathsf{In}}$ in-place iterations leave no discrepancies and computing the complement to one as: $\mathrm{DFR} \leq 1 - \Pr_{S_n^*}\left[t \xrightarrow[\mathsf{imax}^{\mathsf{In}}]{} 0\right]$.

ii. together with the result providing a code-specific bound to the maximum number of discrepancies which can be corrected by an iteration function, either in-place or out-of-place (see Section 3.4), to derive the LEDAdecoder DFR.
In this case, we provide a conservative DFR estimation, for an $\mathsf{imax}^{\mathsf{Out}} = 1$ decoder, computing the maximum number of discrepancies corrected with certainty, τ, according to section Section 3.4, and subsequently determining the probability that the first $\mathsf{imax}^{\mathsf{In}}$ iterations leave at most τ discrepancies as the sum of the probabilities of leaving exactly x discrepancies for all the values of x between 0 and τ: $\mathrm{DFR} \leq 1 - \sum_{x=0}^{\tau} \Pr_{S_n^*}\left[t \xrightarrow[\mathsf{imax}^{\mathsf{In}}]{} x\right]$.

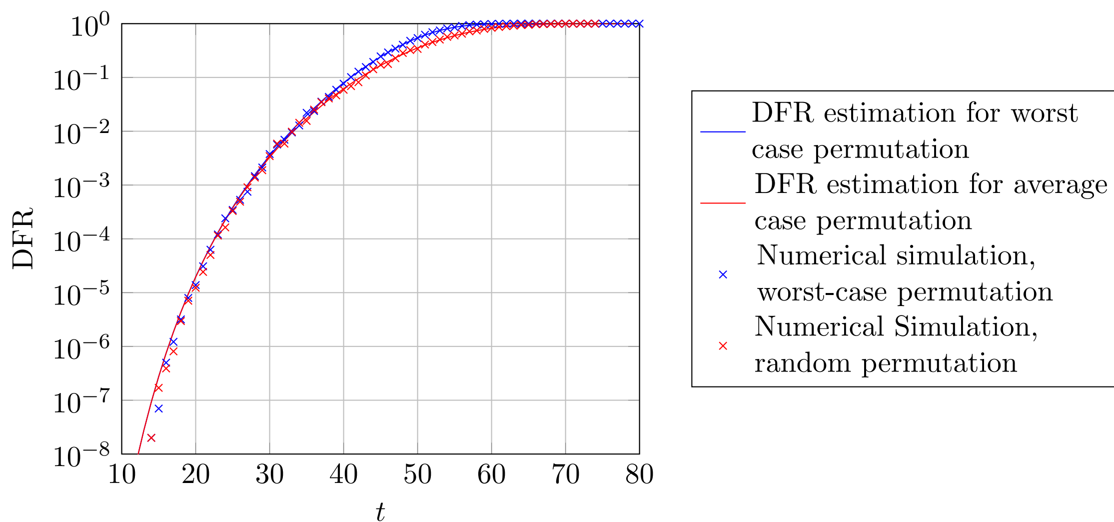


Figure 3.2: Comparison of our worst-case DFR estimation technique with numerical simulations, for a LEDADECODER with $\mathtt{imax}^{\mathtt{In}}= 1$ $\mathtt{imax}^{\mathtt{Out}}= 0$, on a QC-LDPC code with parameters $n_0 = 2, p = 4801, v = 45$, employing $\mathtt{th} = 27$ as the bit flipping threshold. The numerical DFR simulation is obtained decoding random error vectors of weight t, until either 100 decoding errors are reached, or until 10^8 decoding operations have been performed, whichever happens first.

A bonus point of the analysis we proposed is that it is also allows to obtain an estimate for the average DFR of a code, i.e., an estimate of the correction capability whenever a random permutation π is being used. Indeed, let $\pi(\hat{e})$ be the vector obtained by applying the permutation π on $\hat{e}$, with support $\mathrm{Supp}(\pi(\hat{e}))$.

Considering two consecutive elements of elements $\mathrm{Supp}(\pi(\hat{e}))$, a_i, a_{i+1} with $0 \leq i \leq t-2$, we denote the length of the 0-run between them as $a_{i+1} - a_i$. The value d of the average denote the average zero-run length in $\hat{e}$, i.e., $d{=}\mathsf{E}\left[a_{i+1} - a_i\right], \forall i{\in}\{0, 1, \ldots, t-2\}$, where $\mathsf{E}[\cdot]$ denotes the expected value, can be derived to be $\frac{n-\hat{t}}{\hat{t}+1}$. Consequently, we obtain as the estimate for the average DFR after the computation of one in-place iteration as:

$$\forall \hat{t} > 0, \quad \mathrm{DFR} = 1 - \left(\prod_{j=1}^{\hat{t}} \left(\mathrm{P}^{\mathsf{th}}_{m|0}(j) \right)^{d} \right) \prod_{l=1}^{\hat{t}} \mathrm{P}^{\mathsf{th}}_{f|1}(l). \tag{3.3}$$

It is interesting to note that the estimate for the average case DFR converges to the one of the worst case DFR after one in-place iteration function execution, as $\hat{t}$ tends to zero. Indeed, such estimate can be obtained as $\mathrm{DFR} \leq 1 - \Pr_{S_n^*}\left[t \underset{0}{\rightarrow} 0\right]$, where $\Pr_{S_n^*}\left[t \underset{0}{\rightarrow} 0\right]$ can be derived from Appendix A to be $\prod_{j=1}^{\hat{t}} \left(\mathrm{P}^{\mathsf{th}}_{m|0}(j) \right)^{n_0 p - \hat{t}} \cdot \prod_{l=1}^{\hat{t}} \mathrm{P}^{\mathsf{th}}_{f|1}(l)$, and it is easy to note that $n_0 p - t$ converges to the same value as $d = \frac{n-\hat{t}}{\hat{t}+1}$ when $\hat{t}$ tends to zero.

To provide numerical evidence, further backing the soundness of our assumptions, we report in Figure 3.2 the results of performing a numerical simulation down to values of DFR equal to $\approx 10^{-8}$ for a code with $p = 4801, v = 45$. The code parameters were chosen to obtain an acceptable simulation time (about 2k to $5k$ core hours per data set). These parameters require the same effort

to solve either SDP or one of the CFP reported in Chapter 2, assuming $t = 45$: $\approx 2^{96}$ classical operations or $\approx 2^{64}$ quantum gates. We depict the results of the worst-case and average-case DFR estimation technique presented in this section as solid lines. We depict as + signs the results of the simulations of the LEDAdecoder with $\mathtt{imax}^{\mathtt{In}} = 1$, $\mathtt{imax}^{\mathtt{Out}} = 0$, both employing a randomly picked permutation, and selecting ad hoc a worst-case permutation (since we actually know the value of $\tilde{e}$ during the simulations). As it can be seen, our estimation techniques provide a very good fit for the actual behavior of the LEDAdecoder with $\mathtt{imax}^{\mathtt{In}} = 1$, $\mathtt{imax}^{\mathtt{Out}} = 0$, in particular matching also the fact that the average case behavior of the decoder tends to the worst case one when t is decreasing.

Algorithm 14: OutOfPlaceIteration

Input: s: QC-LDPC syndrome, binary vector of size p
Htr: transposed parity-check matrix, represented as an $n_0 \times v$ integer matrix containing in each row the positions (ranging in $\{0, 1, \ldots, p-1\}$) of the set coefficients in the first column of each n_0 $p \times p$ block of the circulant-block matrix $H = [H_0 \mid H_1 \mid \ldots \mid H_{n_0-1}]$
$\hat{e}$: initial error vector estimate, binary vector of size $n = n_0 p$
$\mathtt{iter}^{\mathtt{Out}}$: number of iterations computed before

Output: $\bar{e}$: updated error vector estimate
$\bar{s}$: updated syndrome

```
1  currSynd ← s                                              // Syndrome copy
2  ē ← ê, s̄ ← s
3  th ← ComputeThreshold(iter^Out, s)
4  for i = 0 to n0 − 1 do
5      for offset = 0 to p − 1 do
6          upc ← 0                              // Counter of unsatisfied parity checks
7          for k = 0 to v − 1 do
8              if getSyndromeBit(currSynd, (offset + Htr[i][k]) mod p) = 1 then
9                  upc ← upc + 1
10         if upc ≥ th then
11             ē_{i·p+offset} ← ē_{i·p+offset} ⊕ 1             // Error vector update
12             for k = 0 to v − 1 do
13                 idx ← (Htr[i][k] + offset) mod p
14                 s̄_idx ← s̄_idx ⊕ 1                            // Syndrome update
15 return {ē, s̄}
```

3.3 Residual distribution of the mis-matches between the sought error vector and the input estimated one after the execution of a single out-of-place iteration function

We consider the in-place iteration function described in Algorithm 14, which takes as input the p-bit syndrome s, the initial n-bit error vector estimate $\hat{e}$, and the transposed parity-check matrix of the code represented, for performance reasons, as an $n_0 \times v$ integer matrix containing in each row the positions (ranging in $\{0, 1, \ldots, p-1\}$) of the set coefficients in the first column of each n_0 $p \times p$ block of the circulant-block matrix $H = [H_0 \mid H_1 \mid \ldots \mid H_{n_0-1}]$. The algorithms outputs the updated n-bit error vector estimate, $\bar{e}$, and the corresponding updated syndrome. In the following analysis, the number of mismatches between the error estimate $\hat{e}$ and the sought error vector e that are going to be corrected is denoted as $\hat{t} = wt(\hat{e} \oplus e)$, while the (eXclusive-OR) difference between e and $\hat{e}$ is denoted as $\tilde{e} = \hat{e} \oplus e$ (thus, $\hat{t} = wt(\tilde{e})$).

Our aim is to assess the probability that, at the end of the iteration function computation, the number of mismatches between the output error vector $\bar{e}$ and the actual error vector e is below or equal to a sufficiently small number τ, i.e., that $wt(e \oplus \bar{e}) \leq \tau$.

Algorithm 14 describes the OutOfPlaceIteration function. The computation starts by keeping a copy of the input syndrome s, CurrSynd, on which the unsatisfied parity-check (upc) computations of all the bits of the estimated error vector $\hat{e}$ will be based (line 1), and by initializing the output

error estimate $\bar{e}$ and the output syndrome with the input estimate $\hat{e}$ and the input syndrome, respectively (lines 2). Subsequently, the threshold to test whether to flip or not each bit in the estimated error vector is determined – employing the number of iterations previously computed as input (for possible use in the LEDAcrypt-KEM and LEDAcrypt-PKC systems), or the syndrome weight (for possible use in the LEDAcrypt-KEM-CPA system). Each execution of the body of the nested loops at lines 4–5 considers the j-th element of $\hat{e}$ and the j-th column of the parity-check matrix ($0 \leq j \leq n_0 p - 1$), assuming that $j = \mathtt{i} \cdot p + \mathtt{offset}$, where $\mathtt{i} \in \{0, \ldots, n_0 - 1\}$ is the block index of the column at hand, while $0 \leq \mathtt{offset} \leq p - 1$ is the distance of the column at hand from the first column of the block where it belongs. In particular, the instructions on lines 6–9 compute the number of unsatisfied parity-check equations, upc, to which the bit $\hat{e}$ contributes. If upc is above or equals the chosen threshold, th, (line 10) the j-th bit of the output error estimate $\bar{e}$ (which has been initialized with a copy of $\hat{e}$ at line 2) is flipped and the output syndrome updated accordingly (lines 11–14). The distinguishing behavior w.r.t. the in-place iteration function is determined by the use of the vector CurrSynd to compute the number of unsatisfied parity-check equations and the use of $\bar{s}$ as separate vector for the computation of the output syndrome, that jointly make each bit-flip performed on in the output estimated error vector independent from the others.

We consider Assumption 3.1.1 and Assumption 3.1.2 to hold, and rely on Lemma 3.1.1 to estimate the probabilities that characterize the output error vector estimate of the out-of-place iteration function. Specifically, we use $\mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t})$ to characterize the probability that the iteration function flips any bit position i of $\hat{e}$ which has a discrepancy (i.e., with $\tilde{e}_i = e_i \oplus \hat{e}_i = 1$), and $\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})$ to characterize the probability that the iteration function does not flip any bit position i which is not affected by a discrepancy between $\hat{e}$ and e (i.e., with $\tilde{e}_i = e_i \oplus \hat{e}_i = 0$).

We model the number of error mismatches at the end of the iteration function computation, that is $wt(e \oplus \bar{e})$, as a random variable over the discrete domain of integers $\{0, \ldots, n\}$, $n = n_0 p$, having a probability mass function that we determine as follows.
Let us denote with f_{correct} the number of bit-flips that are correctly performed by a single run of the out-of-place iteration function, i.e., the number of positions j such that $e_j = 1$ and $\bar{e}_j = 1$; analogously, let us denote with f_{wrong} the number of bit-flips that are wrongly performed by a single run of the out-of-place iteration function, i.e., the number of positions j such that $e_j = 0$ and $\bar{e}_j = 1$.

Because of Assumption 3.1.1, the bit-flip actions performed by the out-of-place iteration function are statistically independent, i.e.,

$$\Pr\left[f_{\mathsf{correct}} = x_c, f_{\mathsf{wrong}} = x_w\right] = \Pr\left[f_{\mathsf{correct}} = x_c\right] \cdot \Pr\left[f_{\mathsf{wrong}} = x_w\right], \tag{3.4}$$

while, given Lemma 3.1.1, the probability of the function performing $x_c \in \{0, \ldots, t\}$ correct bit-flips out of t, and the probability of the function performing $x_w \in \{0, \ldots, n-t\}$ wrong bit-flips out of $n - t$ can be quantified as:

$$\begin{aligned} \Pr\left[f_{\mathsf{correct}} = x_c\right] &= \binom{t}{x_c} \left(\mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t})\right)^{x_c} \left(1 - \mathrm{P}^{\mathsf{th}}_{f|1}(\hat{t})\right)^{t - x_c}, \\ \Pr\left[f_{\mathsf{wrong}} = x_w\right] &= \binom{n-t}{x_w} \left(1 - \mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right)^{x_w} \left(\mathrm{P}^{\mathsf{th}}_{m|0}(\hat{t})\right)^{n-t-x_w}. \end{aligned} \tag{3.5}$$

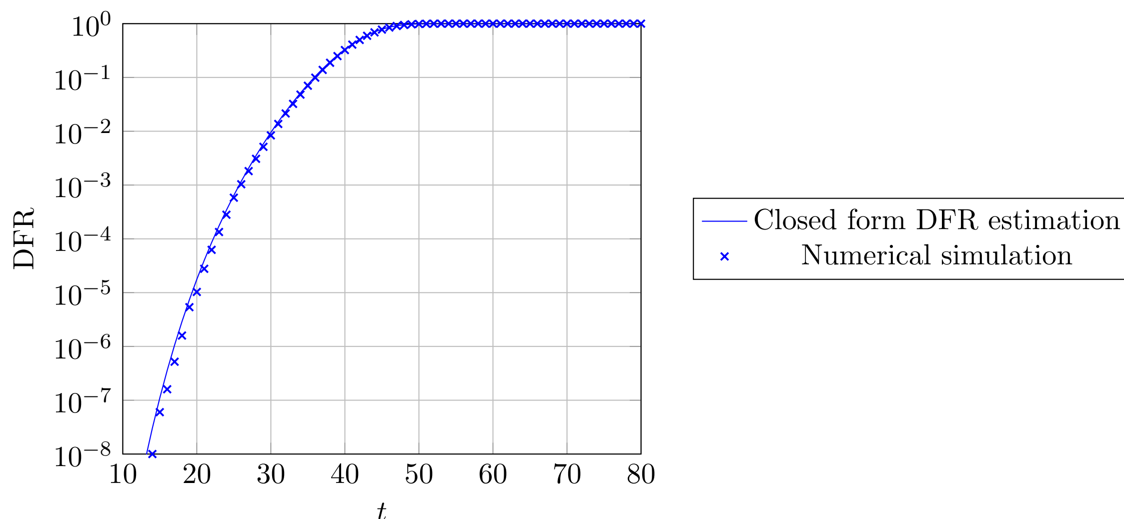


Figure 3.3: Comparison of our worst-case DFR estimation technique with numerical simulations, for a LEDAdecoder with $\mathtt{imax}^{\mathtt{In}} = 0$ $\mathtt{imax}^{\mathtt{Out}} = 1$, on a QC-LDPC code with parametrs $n_0 = 2, p = 4801, v = 45$, employing $\mathtt{th} = 25$ as the bit flipping threshold. The numerical DFR simulation is obtained decoding random error vectors of weight t, until either 100 decoding errors are reached, or until 10^8 decoding operations have been performed, whichever happens first.

It is easy to see that, in the case of $f_{\mathtt{correct}} = x_c$ and $f_{\mathtt{wrong}} = x_w$, we have $wt\,(e \oplus \bar{e}) = t + x_w - x_c$: thus, the probability that the output error vector $\bar{e}$ differs from the actual error vector e in $x \in \{0, \ldots, t\}$ positions can be expressed as:

$$\Pr\left[wt(e \oplus \bar{e}) = x\right] = \sum_{x_c = t-x}^{t} \Pr\left[f_{\mathtt{correct}} = x_c\right] \cdot \Pr\left[f_{\mathtt{wrong}} = x + x_c - t\right]. \tag{3.6}$$

where the number of correct bit-flips that must be performed ranges from t minus the number of residual mismatches x up to the initial number of mismatches t. This, in turn, allows us to derive the probability that the total number of discrepancies left at the end of the out-of-place iteration are at most $\tau \in \{0, \ldots, n\}$ as $\sum_{x=0}^{\tau} \Pr\left[wt(e \oplus \bar{e}) = x\right]$.

This result allows us to obtain the DFR of a decoder employing one or more out-of-place iterations in a fashion similar to the one for the in-place decoder described in Section 3.2, i.e., we determine the maximum number of discrepancies τ which can be corrected with certainty in a single iteration employing the result from Section 3.4, and subsequently determine the probability that running $\mathtt{imax}^{\mathtt{Out}} - 1$ out-of-place iteration functions leave at most τ error employing the method we just described.

The DFR of the entire LEDAdecoder in the case there are only $\mathtt{imax}^{\mathtt{Out}} = 2$ out-of-place iterations (for the parameter design of the IND-CCA2 primitives) is obtained as the complement to one of the aforementioned probability of leaving at most τ discrepancies as follows:

$$\mathrm{DFR} \leq 1 - \sum_{x=0}^{\tau} \Pr\left[wt(e \oplus \bar{e}) = x\right].$$

To provide numerical evidence, further backing the soundness of our assumptions, we report in

Figure 3.3 the results of performing a numerical simulation down to values of DFR equal to $\approx 10^{-8}$ for a code with $p = 4801, v = 45$. The code parameters were chosen to obtain an acceptable simulation time (about 2k to 5k core hours per data set). These parameters require the same effort to solve either SDP or one of the CFP reported in Chapter 2, assuming $t = 45$: $\approx 2^{96}$ classical operations or $\approx 2^{64}$ quantum gates. We depict the results of the conservative DFR estimation technique presented in this section as a solid line and depict as + signs the results of the simulations of the LEDAdecoder with $\mathsf{imax}^{\mathsf{In}} = 0, \mathsf{imax}^{\mathsf{Out}} = 1$. As it can be seen, our estimation techniques provide a very good fit for the actual behavior of the LEDAdecoder with $\mathsf{imax}^{\mathsf{In}} = 0, \mathsf{imax}^{\mathsf{Out}} = 1$.

3.4 Maximum number of mismatches corrected with certainty by executing either an in-place or an out-of-place iteration function

In the following, we consider the execution of one iteration function (either in-place or out-of-place) on an input syndrome s corresponding to $s = H(e \oplus \hat{e})^T$, and an estimated error vector $\hat{e}$. We denote with $\tilde{e}$ the bitwise difference between $\hat{e}$ and e (mismatch), that is $\tilde{e} = \hat{e} \oplus e$, and with $\hat{t} = wt(\tilde{e}) = wt(e \oplus \hat{e})$ the number of discrepancies between the unknown error vector and the input estimated one.

Specifically, we recall the results of [61] and provide a way to determine the code-specific maximum admissible value, τ, for $\hat{t}$ so that the execution of the iteration function outputs an estimated error vector equal to the unknown error vector, i.e.: $\bar{e} = e$.
Note that, in determining such a bound, we rely on no assumptions.

We use s and H to denote the syndrome and parity-check matrix lifted into the integers, respectively, while s_i denotes the i-th entry of s and $\mathsf{h}_{i,j}$ denotes the entry in the i-th row and j-th column of H. The number of unsatisfied parity-checks for the i-th bit of the error vector is denoted as upc_i, and is computed as

$$\mathsf{upc}_i = \sum_{j=0}^{p-1} \mathsf{s}_j \mathsf{h}_{j,i} = \sum_{j=0}^{p-1} \mathsf{h}_{j,i} \left(\bigoplus_{u=0}^{n_0 p-1} \tilde{e}_u h_{j,u} \right) = \sum_{\substack{j \in \{0,\dots,p-1\} \\ h_{j,i}=1}} \left(\bigoplus_{u=0}^{n_0 p-1} \tilde{e}_u h_{j,u} \right). \tag{3.7}$$

We denote with $h_{:,i}$ the i-th column of H which, in LEDAcrypt instances, has weight equal to v for all $0 \leq i \leq pn_0 - 1$. Let $H_{(A)}$ denote the matrix composed by the rows of H indexed by a set of integers $A \subseteq \{0, \dots, p-1\}$ and $h_{(A)\ :,i}$ denote its i-th column, with $0 \leq i \leq pn_0 - 1$.
We thus have that $H_{(\mathrm{Supp}(h_{:,i}))}$ is a rectangular matrix with v rows of $n_0 p$ elements, where the i-th column, $h_{(\mathrm{Supp}(h_{:,i}))\ :,i}$ is constituted by all ones.
$H_{(\mathrm{Supp}(h_{:,i}))}$ thus corresponds to the set of parity-check equations in which the i-th component of the error vector corresponding to the given syndrome is involved.

Equation (3.7) can be thus rewritten as

$$\mathsf{upc}_i = wt\left(\bigoplus_{j \in \mathrm{Supp}(\tilde{e})} h_{(\mathrm{Supp}(h_{:,i}))\ :,j} \right) = wt\left(H_{(\mathrm{Supp}(h_{:,i}))} \tilde{e}^T \right). \tag{3.8}$$

Recall that the iteration function will change the i-th bit of the input error estimate $\hat{e}_i$ if and only if upc_i is greater or equal than threshold th.
Depending on whether $\tilde{e}_i = 0$ or $\tilde{e}_i = 1$, the function will make a correct choice if it does not flip the i-th bit, or if it does, respectively. If $\tilde{e}_i = 0$, the function will make a correct decision if $\mathsf{upc}_i < \mathsf{th}$: we thus consider the following upper bound on upc_i, assuming $\tilde{e}_i = 0$:

$$\mathsf{upc}_i = wt\left(H_{(\mathrm{Supp}(h_{:,i}))} \tilde{e}^T \right) = wt\left(\bigoplus_{j \in \mathrm{Supp}(\tilde{e})} h_{(\mathrm{Supp}(h_{:,i}))\ :,j} \right) \leq \sum_{j \in \mathrm{Supp}(\tilde{e})} wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,j} \right). \tag{3.9}$$

By contrast, if $\tilde{e}_i = 1$ the function will make a correct decision if $\mathsf{upc}_i \geq \mathsf{th}$.
We thus consider the following lower bound on upc_i, assuming $\tilde{e}_i = 1$:

$$\begin{aligned}\mathsf{upc}_i = wt\left(\bigoplus_{j\in \mathrm{Supp}(\tilde{e})} h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right) = wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,i} \oplus \left(\bigoplus_{j\in \mathrm{Supp}(\tilde{e})\setminus\{i\}} h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right)\right)\\ = v - wt\left(\bigoplus_{j\in \mathrm{Supp}(\tilde{e})\setminus\{i\}} h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right) \geq v - \sum_{j\in \mathrm{Supp}(\tilde{e})\setminus\{i\}} wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right).\end{aligned} \tag{3.10}$$

A relevant quantity in Eq. (3.10) is the Hamming weight of the columns $h_{(\mathrm{Supp}(h_{:,i}))\ :,j}$ with $i \neq j$. Indeed, it represents the amount of interference on the upc_i value induced by the presence of error bits in positions different from i.

To ease the computation of $wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right)$, we observe that it is equivalent to compute the cardinality of the set obtained from the intersection of the i-th and j-th column of H, i.e., $|\mathrm{Supp}(h_{:,i}) \cap \mathrm{Supp}(h_{:,j})|$, since $H_{(\mathrm{Supp}(h_{:,i}))}$ is formed by the rows of H in which the column h_i contains a one, and only the ones present in such rows of the j-th column of H will contribute to $wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,j}\right)$.
We therefore define Γ as an $n \times n$ positive integer matrix in which the entry in row i and column j, denoted as $\gamma_{i,j}$, amounts to $|\mathrm{Supp}(h_{:,i}) \cap \mathrm{Supp}(h_{:,j})|$.

After this definition, Eq. (3.9) and Eq. (3.10) can be rewritten as follows:

$$\mathsf{upc}_i \leq \sum_{j\in \mathrm{Supp}(\tilde{e})} \gamma_{i,j}, \quad \text{when } \tilde{e}_i = 0 \text{ (i.e., } e_i \oplus \hat{e}_i = 0), \tag{3.11}$$

$$\mathsf{upc}_i \geq v - \sum_{j\in \mathrm{Supp}(\tilde{e})\setminus\{i\}} \gamma_{i,j}, \quad \text{when } \tilde{e}_i = 1 \text{ (i.e., } e_i \oplus \hat{e}_i = 1). \tag{3.12}$$

We thus have just provided a way to compute a code-specific (Γ depends on the specific H instance) upper bound to the value of upc_i in case the corresponding position in $\hat{e}$ is correctly estimated (i.e., $\tilde{e}_i = 0$), and a lower bound to the value of upc_i in case $\hat{e}$ is incorrectly estimated (i.e., $\tilde{e}_i = 1$).

Since we are willing to determine the case where the iteration function corrects with certainty the discrepancies in $\tilde{e}$, we need to consider the case where the upper bound on upc_i for each position corresponding to a correct $\hat{e}_i$ (see Eq. (3.11)) is strictly lower than the lower bound on the value of upc_i for the case of an incorrect $\hat{e}_i$ (see Eq. (3.12)).
Therefore, given a value $0 \leq i \leq n_0p - 1$, it must hold that:

$$\sum_{j\in \mathrm{Supp}(\tilde{e})} \gamma_{i,j} < v - \sum_{j\in \mathrm{Supp}(\tilde{e})\setminus\{i\}} \gamma_{i,j}, \quad \forall\, \tilde{e} \in \mathsf{F}_2^n \text{ s.t. } wt\left(\tilde{e}\right) = \hat{t}. \tag{3.13}$$

Indeed, in this case, it is possible to use as the bit-flipping threshold (e.g., th in Algorithm 14 on line 3) any value strictly greater than the left quantity in Eq. (3.13), resulting in the iteration function executing a bit-flip of all and only the incorrectly estimated $\hat{e}_i$ received as input.

For the sake of a more compact notation, we now define the quantity $\mu(z), 0 \leq z \leq n_0p$. Informally, $\mu(z)$ is the maximum change in the value of any single $\mathtt{upc}_i$ induced by z discrepancies being present in locations different from the i-th itself. We therefore have that:

$$\mu(z) = \max_{\substack{0 \leq i \leq n_0p-1 \\ \tilde{e} \in \mathbb{F}_2^n, wt(\tilde{e})=z,\ \tilde{e}_i=0}} \left\{ \sum_{j \in \mathrm{Supp}(\tilde{e})} \gamma_{i,j} \right\}$$

The definition of this quantity, allows to rewrite Eq. (3.11) as follows:

$$\mathtt{upc}_i \leq \mu(\hat{t}), \quad \text{when } \tilde{e}_i = 0 \text{ (i.e., } e_i \oplus \hat{e}_i = 0) \tag{3.14}$$

since the contributions to $\mathtt{upc}_i$ whenever no discrepancy is present in the i-th position (i.e., $\tilde{e}_i = 0$), only come from the $\hat{t}$ discrepancies being present in the other locations.

Similarly, observing that, whenever a discrepancy is present in the i-th position, (i.e., $\tilde{e}_i = 0$), the value of $\mathtt{upc}_i$ is obtained subtracting from v (as $wt\left(h_{(\mathrm{Supp}(h_{:,i}))\ :,i}\right) = v$) the effects of the other $\hat{t}-1$ columns selected by the positions of of asserted bits in the error vector, allowing to rewrite Eq. (3.12) as:

$$\mathtt{upc}_i \geq v - \mu(\hat{t}-1), \quad \text{when } \tilde{e}_i = 1 \text{ (i.e., } e_i \oplus \hat{e}_i = 1). \tag{3.15}$$

Consequentially, also Eq. (3.13) can be rewritten as:

$$\mu(\hat{t}) + \mu(\hat{t}-1) < v. \tag{3.16}$$

Considering Eq. (3.16), the largest value of $\hat{t} = wt\,(\tilde{e}) = wt\,(e \oplus \hat{e})$ for which it holds is the maximum number of residual mismatches between $\hat{e}$ and e that be corrected with certainty by the execution of the iteration function at hand, employing a bit-flip threshold $\mathtt{th}$ such that:

$$\mu(\hat{t}) + 1 \leq \mathtt{th} \leq v - \mu(\hat{t}).$$

Therefore, given a specific code, which means a specific instance of the parity-check matrix H, it is possible to compute the threshold $\mathtt{th}$ to be set during the execution of an iteration function in such a way that it is able to correct with certainty a number of mismatches between e and the input $\hat{e}$, which amounts to $\hat{t}$. To this end, performance concerns on design and implementation of the LEDADecoder mandates to find an efficient strategy to compute both the matrix Γ and $\hat{t}$, as described in the next sub-section.

3.4.1 Efficient computation of Γ and $\mu(t)$

The computation of the largest value of mismatches between the sought error vector and the input estimated error vector, $\hat{t}$, that is surely corrected by an out-of-place iteration function requires a significant computational effort for the calculus of the matrix Γ, which is approximately cubic in the value of n_0p.

We now prove that Γ is quasi cyclic, thus allowing a significant reduction in the time required by its computation. Note that Γ is a symmetric matrix, since $\gamma_{i,j} = \gamma_{j,i}$, for all pairs of indexes i, j due to the symmetry of the column intersection. Consider the indexes $i, j \in \{0, \ldots, n_0p-1\}$ obtained as $i = pi_p + i'$ and $j = pj_p + j'$, respectively, with $i_p, j_p \in \{0, \ldots n_0 - 1\}$ and $i', j' \in \{0, \ldots, p-1\}$.

Denoting with P the $p \times p$ circulant permutation matrix associated to the cyclic shift of one position, and observing that $P^p = I_p$, and that $(P^a)^T = P^{p-a}$, we have that

$$h_{:,i} = P^{i'} h_{:,i_p}, \qquad h_{:,j} = P^{j'} h_{:,j_p} \tag{3.17}$$

From this observation, we can derive

$$\begin{aligned}\gamma_{i,j} = h_{:,i}^T h_{:,j} &= \left(P^{i'} h_{:,i_p}\right)^T P^{j'} h_{:,j_p} \\ &= h_{:,i_p}^T P^{p-i'} P^{j'} h_{:,j_p} \\ &= \gamma_{p i_p, p j_p + (j'-i' \mod p)}\end{aligned}$$

which proves that the matrix Γ is quasi cyclic with block size p, and can thus be written as

$$\Gamma = \begin{bmatrix} \Gamma_{0,0} & \Gamma_{0,1} & \cdots & \Gamma_{0,n_0-1} \\ \Gamma_{1,0} & \Gamma_{1,1} & \cdots & \Gamma_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ \Gamma_{n_0-1,0} & \Gamma_{n_0-1,1} & \cdots & \Gamma_{n_0-1,n_0-1} \end{bmatrix}, \tag{3.18}$$

where each $\Gamma_{i,j}$ is a $p \times p$ circulant integer matrix; in particular, due to the commutativity of the set intersection that defines the $\gamma_{i,j}$ values, we also have that $\Gamma_{i,j}^T = \Gamma_{j,i}$.

Based on the above analysis, it can be easily shown that the matrix Γ is fully described by a subset of only $\frac{p-1}{2} n_0(n_0+1) - n_0$ entries. Indeed, all the blocks on the main block-diagonal of Γ, i.e. the $\Gamma_{i,i}, i \in \{0, \ldots, n_0-1\}$ blocks, are both symmetrical and circulant, and recalling that we do not need to store the elements on the main diagonal, i.e. the $\gamma_{i,i}, i \in \{0, \ldots, n_0-1\}$ values we obtain that $\frac{p-1}{2}$ elements are sufficient to fully describe each $\Gamma_{i,i}$. Since there are n_0 blocks on the diagonal, those can be represented by $n_0 \frac{p-1}{2}$ elements of Γ. Thanks to the fact that Γ is symmetric, it is sufficient to represent, besides the blocks on the diagonal, only the blocks $\Gamma_{i,j}, i \in \{0, \ldots, n_0-1\}, j \in \{1, \ldots, n_0-1\}, i < j$. Since there are $\binom{n_0}{2}$ such blocks, and each one requires p elements to be fully described, we obtain a total number of elements equal to

$$p \cdot \frac{n_0(n_0-1)}{2} + n_0 \frac{p-1}{2} = \frac{n_0(n_0 p - 1)}{2}.$$

This property can be used to obtain an efficient method to evaluate the largest value of $\hat{t}$ for which the condition in Eq. (3.16) still holds.

Algorithm 15 provides a procedural description of such an efficient computation, calculating only the first rows of the circulant blocks of Γ (lines 2–15), and subsequently testing the admissible values for $\hat{t}$ (lines 19–30). Note that it is possible to further cut down the computational complexity of the algorithm by computing directly the histograms containing the occurrences of unique values in each row of Γ. Such a method is not reported for the sake of brevity, but it allows a significant speedup (around two orders of magnitude) in practice.

Algorithm 15: COMPUTING MAXIMUM ADMISSIBLE $\hat{t}$

Input: H: a $1 \times n_0$ blocks, block circulant parity-check matrix with $p \times p$ wide blocks $H = [H_0 \ldots H_{n_0-1}]$.
Output: $\hat{t}$: maximum admissible number of discrepancies between e and $\hat{e}$
th: out of place iteration threshold
Data: $y_{(i,j)}$: first row of the (i,j)-th block of the Γ matrix

```
1  for i ← 0 to n_0 − 2 do
2      for j ← i to n_0 − 1 do
3          A ← Supp(H_{i :,1})
4          B ← Supp(H_{j :,1})
5          for z ← 0 to p − 1 do
6              if i ≠ j ∨ z ≠ 0 then
7                  C ← ?
8                  foreach b ∈ B do
9                      C ← C ∪ {(b + z) mod p}
10                 y_(i,j)[z] ← |A ∩ C|
11             else
12                 y_(i,j)[z] ← 0
13 for i ← 1 to n_0 − 1 do
14     for j ← i + 1 to n_0 − 1 do
15         y_(j,i) ← y_(i,j)
16 mu_t← 0
17 mu_t_min_one← 0
18 t̂ ← 0
19 while mu_t+mu_t_min_one< v do
20     valid_mu_t← mu_t_min_one
21     for i ← 0 to n_0 − 1 do
22         for j ← 0 to n_0 − 1 do
23             tmp← Concatenate(tmp, y_(i,j))
24         tmp← SortDecreasing(tmp)
25         if mu_t< SumRange(tmp, 0, t̂ − 1) then
26             mu_t← SumRange(tmp, 0, t̂ − 1)
27         if mu_t_min_one< SumRange(tmp, 0, t̂ − 1) then
28             mu_t_min_one← SumRange(tmp, 0, t̂ − 1)
29     t̂ ← t̂ + 1
30 th←valid_mu_t+1
31 return ⟨t̂ − 1, th⟩
```

3.5 Efficient choice of out-of place decoder thresholds for the numerically simulated DFR of LEDAcrypt-KEM-CPA

The choice of the flipping threshold for the out-of-place iteration function influences its error correction capability. While the previous section described how to determine a threshold such that a number of discrepancies in the error estimate would be corrected with certainty, the present section aims at providing a more sophisticated threshold selection criterion. While the criterion we introduce in this section does not allow for a closed form analysis of the DFR, it practically provides very fast convergence of a decoding algorithm relying only on out-of-place iterations. We thus exploit this criterion for LEDAcrypt-KEM-CPA, where ephemeral keys are employed and a practically low enough DFR is sufficient. This also motivates our choice of employing a decoder that only exploits out-of-place iterations for LEDAcrypt instances with ephemeral keys. We assess its DFR by means of numerical simulations targeting parameters sets with a DFR= 10^{-9}, and derive the value of $\mathsf{imax}^{\mathsf{out}}$ as the maximum number of iterations found to be required for achieving the target DFR. These engineering choices allow the decoder to be fully parallelized, as all the upc values can be computed independently, and the flipping decisions applied to the $\hat{e}$ and s likewise.

In the following, we describe our rationale for the computation of the ComputeThreshold function required in the OutOfPlaceIteration Algorithm 14 at line 3.

A natural choice is to set the threshold used at the l-th iteration of the whole function equal to the largest among the upc values. This strategy ensures that only those few bits that have maximum likelihood of being mismatched between $\hat{e}$ and e are flipped during each function execution, thus achieving the lowest DFR. However, it has some drawbacks in terms of complexity, since the computation of the maximum upc value, for each column of the H matrix, entails additional memory storage and some repeated operations. For such reasons, we consider an approach with reduced complexity, according to which the threshold values are computed on the basis of the syndrome weight $wt(s)$ at each execution of the iteration function. The function relating the syndrome weight to the threshold to be chosen can be precomputed and tabulated, allowing an efficient lookup to take place at runtime.

Coherently with the notation adopted in the previous sections, we denote with $\hat{e}$ and s, respectively, the input error vector estimate and the syndrome, and recall that $s = H\,(e \oplus \hat{e})^T$, where $wt\,(e \oplus \hat{e}) = \hat{t}$. Let us denote with $wt(s)|_{\hat{t}}$ the average syndrome weight when $wt\,(e \oplus \hat{e}) = \hat{t}$.
To this end, let $\mathsf{p}_{\mathsf{unsatisfied}}(\hat{t})$ denote the probability that a randomly selected parity-check equation is unsatisfied. Assuming that $e \oplus \hat{e}$ is distributed as a random vector of weight $\hat{t}$, we have:

$$\mathsf{p}_{\mathsf{unsatisfied}}(\hat{t}) = \sum_{j=1,\, j\ \mathsf{odd}}^{\min\{n_0 v, \hat{t}\}} \frac{\binom{n_0 v}{j}\binom{n_0 p - n_0 v}{\hat{t}-j}}{\binom{n_0 p}{\hat{t}}}.$$

Then, under the assumption that the parity-check equations are statistically independent, the average syndrome weight is estimated as:

$$wt(s)|_{\hat{t}} = p \cdot \mathsf{p}_{\mathsf{unsatisfied}}(\hat{t}). \tag{3.19}$$

The aforementioned relation allows us to estimate $\hat{t}$, the amount of set bits in $\tilde{e} = e \oplus \hat{e}$, as the value $\tilde{t}$ for which $wt(s)|_{\tilde{t}}$ is closer to the (available) $wt(s)|_{\hat{t}}$. Relying on this estimate to approximate the value of $\hat{t}$ as $\tilde{t}$, we now define a strategy to pick a threshold value depending on the estimated number of discrepancies $\tilde{t}$ itself.

We now consider $\tilde{e}_i$, i.e., the i-th element of the vector of differences between e and $\hat{e}$, $\tilde{e} = \hat{e} \oplus e$ and assume that the number of discrepancies $wt(\tilde{e}) = \hat{t}$ matches our estimate $\tilde{t}$.
Denote the number of unsatisfied parity-check equations in which the i-th bit participates as $\mathsf{upc}_i \in \{0.\ldots, v\}$, with $0 \le i \le n_0p-1$. The probability that such a position contains a discrepancy can be written as:

$$\begin{aligned}\Pr\left[\tilde{e}_i = 1 \,|\mathsf{upc}_i = \sigma\right] &= \frac{\Pr\left[\tilde{e}_i = 1, \mathsf{upc}_i = \sigma\right]}{\Pr\left[\mathsf{upc}_i = \sigma\right]} = \\ &= \frac{\Pr\left[\tilde{e}_i = 1, \mathsf{upc}_i = \sigma\right]}{\Pr\left[\tilde{e}_i = 1, \mathsf{upc}_i = \sigma\right] + \Pr\left[\tilde{e}_i = 0, \mathsf{upc}_i = \sigma\right]} = \\ &= \left(1 + \frac{\Pr\left[\tilde{e}_i = 0, \mathsf{upc}_i = \sigma\right]}{\Pr\left[\tilde{e}_i = 1, \mathsf{upc}_i = \sigma\right]}\right)^{-1}.\end{aligned}$$

By elaborating the previous result, we obtain:

$$\Pr\left[\tilde{e}_i = 1 \,|\mathsf{upc}_i = \sigma\right] = \frac{1}{1 + \frac{n_0p-\tilde{t}}{\tilde{t}}\left(\frac{\rho_{0,\mathsf{u}}(\tilde{t})}{\rho_{1,\mathsf{u}}(\tilde{t})}\right)^{\sigma}\left(\frac{1-\rho_{0,\mathsf{u}}(\tilde{t})}{1-\rho_{1,\mathsf{u}}(\tilde{t})}\right)^{v-\sigma}}, \tag{3.20}$$

where $\rho_{0,\mathsf{u}}(\tilde{t})$ and $\rho_{1,\mathsf{u}}(\tilde{t})$ are given in Lemma 3.1.1, and note that (as expected intuitively) it is a monotonically increasing function as σ increases over the integers $\{0, \ldots, v\}$.

Willing to define a threshold for the value of upc_i such that it is advisable to flip the i-th bit of $\hat{e}$, we have that the probability of performing a flip when $\tilde{e}_i = 1$ should exceed the one when $\tilde{e}_i = 0$, in turn for a given value σ taken by upc_i, yielding:

$$\left\{\begin{array}{lcl}\Pr\left[\tilde{e}_i = 1 \mid \mathsf{upc}_i = \sigma\right] & > & (1+\Delta)\Pr\left[\tilde{e}_i = 0 \mid \mathsf{upc}_i = \sigma\right] \\ \Pr\left[\tilde{e}_i = 0 \mid \mathsf{upc}_i = \sigma\right] & = & 1 - \Pr\left[\tilde{e}_i = 1 \mid \mathsf{upc}_i = \sigma\right]\end{array}\right. \Rightarrow \left\{\begin{array}{lcl}\Pr\left[\tilde{e}_i = 1 \mid \mathsf{upc}_i = \sigma\right] & > & \frac{1+\Delta}{2+\Delta} \\ \Pr\left[\tilde{e}_i = 0 \mid \mathsf{upc}_i = \sigma\right] & < & \frac{1}{2+\Delta}\end{array}\right.$$

where $\Delta \ge 0$ represents a margin which can be arbitrarily chosen. The minimum value of σ such that the inequalities in the previous relation are satisfied can be computed as:

$$\mathsf{th} = \min\left\{\sigma \in \{0, \ldots, v\} \;:\; \Pr\left[\tilde{e}_i = 1 | \mathsf{upc}_i = \sigma\right] > \frac{1+\Delta}{2+\Delta}\right\} \tag{3.21}$$

and used as the decision threshold during the execution of the out-of-place iteration function.

We now combine the results of Eq. (3.19) (linking the the weight of the syndrome to the estimated number of discrepancies $\tilde{t}$), Eq. (3.20) (linking the value of the estimated number of discrepancies $\tilde{t}$, to the correct flipping probability given a certain threshold), to obtain the function mapping a given(available) value for $wt(s)$ onto the corresponding threshold value to be adopted for the out-of-place iteration. Such a function, of which we provide a graphical representation in Figure 3.4, splits the domain of possible values for $wt(s)$ into t regions, according to the value $\tilde{t}$ for which the value $wt(s)|_{\tilde{t}}$ is closest to the available value of $wt(s)$.
We therefore determine $t+1$ delimiting values, $\overline{w}_s^{(j)}, j \in \{0, \ldots, t\}$ as:

$$\overline{w}_s^{(0)} = 0; \qquad \forall\, j \in \{1, \ldots, t\},\;\; \overline{w}_s^{(j)} = \frac{wt(s)|_{j-1} + wt(s)|_j}{2}, \tag{3.22}$$

Each of the t regions, each one delimited by a pair $\overline{w}_s^{(j)}, \overline{w}_s^{(j+1)}, j \in \{0, \ldots, t\}$, is thus considered to be the one where we assume $\tilde{t} = j$, and employ this value of $\tilde{t}$ to derive the value of the threshold to be used via Eq. (3.21).

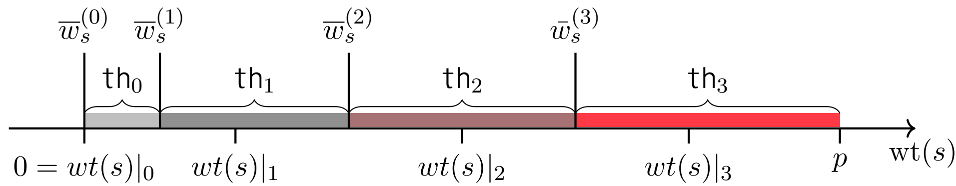


Figure 3.4: Graphical representation of the relations between $wt\,(s)$, $wt(s)|_j$, $\overline{w}_s^{(j)}$ and th_j. The example is referred to the case of $t = 3$.

It can be easily shown that, if j is the largest integer such that $wt\,(s) \geq \overline{w}_s^{(j)}$, then j minimizes $|wt\,(s) - \overline{w}_s^{(j)}|$. The function described by Eq. (3.22) can be efficiently stored as a lookup table containing pairs $\langle \overline{w}_s^{(j)}, \mathsf{th}_j \rangle$ for all $j \in \{0, \dots, t\}$.
Therefore, the ComputeThreshold function will start by computing the value of $wt(s)$, and subsequently find the required threshold $\mathsf{th}_{\tilde{t}}$. Algorithm 16 provides a procedural description of a constant-time lookup into the table.

We have observed that, moving from the largest values of $\overline{w}_s^{(j)}$ to the smallest ones, the threshold values computed this way firstly exhibit a decreasing trend, then start to increase. According to numerical simulations, neglecting the final increase has no impact from the performance standpoint. Therefore, in the look-up table we replace the threshold values after the minimum with a constant value equal to the minimum itself. In addition, the so-obtained table usually contains consecutive rows with the same threshold value: we can clearly eliminate all such rows, apart from the first one (i.e., the one containing $\overline{w}_s^{(0)}$). By means of numerical simulations, we have also observed that the lowest DFR within a reasonable amount of iterations, for LEDAcrypt instances, is obtained with the choice $\Delta = 0$.

Algorithm 16: ComputeThreshold($\mathsf{iter}^{\mathsf{In}}, s$)

Input: $\mathsf{iter}^{\mathsf{In}}$: Number of iterations computed before the current one. This input parameter is not used to compute the thresholds of the out-of-place-decoder for LEDAcrypt-KEM-CPA. Formally, Algorithm 12 and Algorithm 14 take it as an input to be generic w.r.t. the specific instance of the LEDAcrypt primitive.
s: syndrome, binary vector of size p

Output: th: decision threshold, integer in $\{\lceil v/2 \rceil, \dots, v\}$

Data: LutS: Lookup table containing the pairs $\langle \overline{w}_s^{(j)}, \mathsf{th}_j \rangle$, $j \in \{0, \dots, t\}$

```
1 th ← ⌈v/2⌉
2 foreach ⟨w̄_s^(j), th_j⟩ ∈ LutS do
3     th ← th · (wt(s) > w̄_s^(j)) + th_j · (wt(s) ≤ w̄_s^(j))
4 return th
```

Chapter 4

LEDAcrypt code parameters from a security standpoint

In this chapter we describe the reproducible parameter design procedure employed to obtain the code parameters in LEDAcrypt cryptosystems, provide the parameter sets themselves, and report our choices for the symmetric primitives required in the cryptosystems. We provide the parameter generation procedure as public domain software at https://github.com/ledacrypt.

Symmetric primitives. Concerning the IND-CCA2 construction for LEDAcrypt-KEM (Section 1.3), we chose to employ as Extendable Output Function (XOF), the NIST standard SHAKE-128 [25] for NIST Category 1 and SHAKE-256 [25] for Category 3 and Category 5. We chose to employ as cryptographic hash the NIST standard SHA-3 [25] with digest length, l_{hash}, equal to 256, 384, and 512 bits for NIST Category 1, 3, and 5 of the LEDAcrypt parameters, respectively. Taking as input the error vector, two distinct instances of such a cryptographic hash are employed in the IND-CCA2 construction of LEDAcrypt-KEM. Indeed, an instance is employed to implement the Key Derivation Function (KDF) that outputs the encapsulated secret key, while the other one is employed to build the confirmation tag to be transmitted along with the ciphertext. To achieve domain separation between the said instances of the cryptographic hash, we prefix the input to the former instance (i.e., the one employed for the KDF) with a zero-valued byte, while we prefix the input to the second instance with the binary encoding of 1 over one byte. This technique, called *pre xing* in the recent analysis of many of the IND-CCA2 constructions employed in the NIST PQC contest [10], allows to derive in a sound fashion two different hash functions starting from the same primitive.
We note that SHA-3 and SHAKE are already domain separated due to their definition, as they have a different suffix added to the message by construction.

Concerning the IND-CCA2 construction for LEDAcrypt-PKC (Section 1.4), we chose to employ as Deterministic Random Bit Generator (DRBG) the NIST recommended CTR-DRBG [8] instantiated with AES-256 and fed with a TRNG-extracted seed having bit-length, l_{seed}, equal to 192, 256, and 320 bits for the NIST Category 1, 3, 5 of the LEDAcrypt parameters, respectively.

Designing code parameters. The LEDAcrypt design procedure described in this section takes as input the desired security level λ_c and λ_q, expressed as the base-2 logarithm of the number of operations of the desired computational effort on a classical and quantum computer, respectively. In addition to λ_c and λ_q, the procedure also takes as input the number of circulant blocks, $n_0 \in \{2, 3, 4\}$,

forming the parity-check matrix H, allowing tuning of the code rate. Finally, the design procedure takes the maximum acceptable DFR, and the number of in-place, $\mathsf{imax}^{\mathsf{In}}$, and out-of place, $\mathsf{imax}^{\mathsf{Out}}$, decoder iterations.

The parameter design procedure outputs the size of the circulant blocks, p, the weight of a column of H, v, and the number of intentional errors, t. We recall that Q is chosen as I, as highlighted in Section 2.4 and is therefore neglected in the entire design procedure, and in the code.

The procedure enforces the following constraints on the parameter choice:

- The minimum cost for a message recovery via Information Set Decoding (ISD) on both quantum and classical computers must exceed 2^{λ_q} and 2^{λ_c} operations, respectively. This constraint binds the values of the code length $n = n_0 p$, the code dimension $k = (n_0 - 1)p$ and the number of errors t to be chosen such that an ISD on the public code $\mathsf{C}(n, k)$ aiming at decoding a random error vector of weight t requires more than 2^{λ_q} or 2^{λ_c} operations on a quantum and a classical computer.
- The minimum cost for a key recovery attack via ISD on both quantum and classical computers must exceed 2^{λ_q} and 2^{λ_c} operations, respectively. This constraint binds the values of the code length $n = n_0 p$, the code redundancy $r = p$ and the number of ones in a row of H, $w = vn_0$, to be chosen such that the effort of an recovering the private QC-LDPC/QC-MDPC code, i.e., $\min\left\{\frac{\mathsf{ISD_{cfp}}(n_0 p, p, 2v)}{p\binom{n_0}{2}}, \frac{\mathsf{ISD_{cfp}}(2p, p, 2v)}{n_0 p}, \frac{\mathsf{ISD_{cfp}}(n_0 p, (n_0-1)p, n_0 v)}{p}\right\}$ (see Section 2.3.1), requires more than 2^{λ_q} or 2^{λ_c} operations on a quantum and classical computer.
- The choice of the circulant block size, p, should be such that p is a prime number and such that $\mathrm{ord}_2(p) = p - 1$ (see Theorem 1.1.3).
- The choice of the circulant block size, p, and parity-check matrix density, $w = n_0 v$, must allow the code to correct the required amount of errors, with the specified DFR, depending on the IND-CCA2 or IND-CPA requirements of the primitive.

We report a synthetic description of the procedure implemented in the publicly available code as Algorithm 17. The rationale of the procedure[1] is to proceed in refining the choice for p, t, v at fix point, considering only values of p respecting $\mathrm{ord}_2(p) = p - 1$.

Since there are cyclic dependencies among the constraints on p, t, and v, the search for the parameter set is structured as a fix point solver iterating the computation of appropriate values for the three desired parameters (Algorithm 17, lines 2–18), starting from an arbitrary choice of the value of p, so to be able to compute the ISD costs.

The loop body starts by deriving the length, n, dimension, k, and redundancy, $r = n - k$, of the code, assigning them to obtain a code rate equal to $1 - \frac{1}{n_0}$ (line 3). Subsequently, the procedure for the parameter choice proceeds executing a loop (lines 5–7) to determine a value t, with $t < r$, such that a message recovery attack on a generic code $\mathsf{C}(n, k)$, looking for an error vector with weight t, requires more than the specified amount of computational efforts on both classical and quantum computers.

[1] Note that, in the pseudocode of Algorithm 17, the loop construct **while**(< condition >) . . . iterates the execution of instructions in the loop body when the condition is **true**, while the loop construct **Repeat** . . . **until**(< condition >) iterates the instructions in the loop body when the condition is **false**.

Algorithm 17: LEDAcrypt Parameter Generation

Input: λ_c, λ_q:desired security levels against classical and quantum attacks, respectively;
ϵ: either $\log_2$(DFR), with DFR being the maximum allowed DFR (IND-CCA2 parameters), or an enlargement factor for the heuristic estimate of a simulatable DFR set (IND-CPA parameters);
n_0: number of circulant blocks of the $p \times n_0 p$ parity-check matrix H of the code.

Output: p: size of a circulant block; t: number of errors; v: weight of a column of the parity matrix H; τ: number of errors to be corrected with certainty by a single iteration of the decoder.

Data: NextPrime(x): subroutine returning the first prime p larger than the value of the input parameter and such that $\mathrm{ord}_2(p) = p - 1$;
$\mathsf{C\text{-}ISD_{sdp}}(n, r, t)$,$\mathsf{C\text{-}ISD_{cfp}}(n, r, t)$,$\mathsf{Q\text{-}ISD_{sdp}}(n, r, t)$,$\mathsf{Q\text{-}ISD_{sdp}}(n, r, t)$: cost of the syndrome decoding and codeword finding ISD procedures, on a classical and quantum computer, respectively;

```
1  p ← starting_prime
2  repeat
3      n ← n0p, k ← (n0 − 1)p, r ← p
4      t ← 0
5      repeat
6          t ← t + 1
7      until (t ≥ r ∨ (C-ISD_sdp(n, r, t) ≥ 2^λc ∧ Q-ISD_sdp(n, r, t) ≥ 2^λq))
8      v ← 1
9      repeat
10         v ← v + 2
11         SecureOk ← C-ISD_cfp(n0p,p,2v) / (p·binom(n0,2)) ≥ 2^λc ∧ Q-ISD_cfp(n0p,p,2v) / (p·binom(n0,2)) ≥ 2^λq
12         SecureOk ← SecureOk ∧ C-ISD_cfp(2p,p,2v) / (n0p) ≥ 2^λc ∧ Q-ISD_cfp(2p,p,2v) / (n0p) ≥ 2^λq
13         SecureOk ← SecureOk ∧ C-ISD_cfp(n0p,(n0−1)p,n0v) / p ≥ 2^λc ∧ Q-ISD_cfp(n0p,(n0−1)p,n0v) / p ≥ 2^λq
14     until (SecureOk = true ∨ vn0 ≥ p)
15     if (SecureOk = true) then
16         p, τ ← ComputepDFRok(n0, v, t)
17     return (p, t, v)
18 until p,v,t do not change
```

To determine the weight of a column of H, i.e., v, the procedure moves on searching for a candidate value of v. The loop at lines 9–14 tests the resistance of the cryptosystem against key recovery attacks attacks on both classical and quantum computers. The final value of v is computed as the smallest odd integer such that resistance against such attacks is obtained. We remember that the condition of v being odd is sufficient to guarantee the non singularity of the circulant blocks of H.

In both the evaluation of the key recovery attacks, which drives the choice of v, and the evaluation of the message recovery attack, which drives the choice of t, we consider the best (i.e., lowest) complexity among the ones of solving the Codeword Finding Problem / Syndrome Decoding Problem (CFP/SDP) with the ISD techniques proposed by Prange [58], Lee and Brickell [47], Leon [48], Stern [69], Finiasz and Sendrier [29], and Becker, Joux, May and Meurer (BJMM) [9], and Prange [58], Lee and Brickell [47], Stern [69] for the quantum computing platform as discussed in Section 2.3. For all the aforementioned algorithm, we perform an exhaustive search for the best ISD parameters, ensuring that the found parameter sets do not lie at the boundary of our exploration range. Since all the ISD parameters are the result of a computational tradeoff, we expect

the found solution to be the actual minimum computation effort for an attack.

When suitable values for the code parameters from a security standpoint are found, the algorithm computes the minimum value of p, such that the decoding algorithm is expected to correct t errors, with the desired DFR, depending on the guarantees required by the primitive. The same procedure computing the appropriate value for p also derives, in case an estimate for a LEDAdecoder to be used in IND-CCA2 primitives is desired, the value τ of errors to be corrected by the decoder in a single iteration with certainty.
In order to achieve the maximum correction efficiency from the code at hand, the procedure estimating the value of p yielding the desired DFR computes the best (i.e., lowest) DFR value exploring exhaustively all the values for the decoding thresholds in the decoders.
As a consequence, we employ at key generation phase a rejection sampling procedure, which performs the analysis from Section 3.4, and discards the keypair if the code does not correct the amount of errors τ determined at parameter design time. To prevent this rejection sampling from significantly diminishing the keyspace, we estimate the value of τ at design time by running the analysis from Section 3.4 on 1000 randomly extracted codes, and pick the value τ which is corrected by at least a half of the randomly generated codes.

We note that such a fixpoint optimization procedure, with respect to the one proposed during the first round of the NIST post quantum standardization process, simultaneously optimizes the parameters for achieving the desired security margin and DFR, while the previous procedure to obtain long-term keys (i.e., suitable for IND-CCA2 primitives) started from the optimization of the values of t and v to match the desired security margin against ISD attacks, and then gradually increased the value of p until a desirable DFR was achieved. The current procedure instead, allows to exploit the additional security margin coming from an increase of the value of p, in turn allowing a reduction of the keysizes.

The C++ tool provided relies on Victor Shoup's NTL library (available at `https://www.shoup.net/ntl/`), in particular for the arbitrary precision integer computations and the tunable precision floating point computations, and requires a compiler supporting the C++11 standard.

The tool follows the computation logic described in Algorithm 17, but it is optimized to reduce the computational effort as follows:

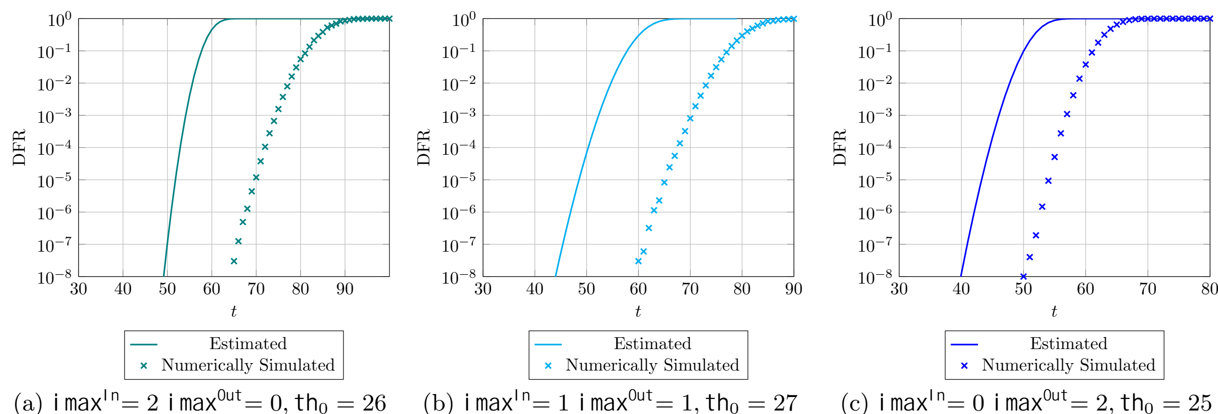
- The values of p respecting the constraint $\text{ord}_2(p) = p - 1$ are pre-computed up to $149,939$ and stored in a lookup table.
- The search for the values of p, t and v respecting the constraints is performed by means of a dichotomic search instead of a linear scan of the range.
- The computations of the binomial coefficients employs a precomputation of all the factorials as arbitrary precision integers up to a tunable number. When a factorial is not available as a precomputed value, we switch to a direct, incremental calculation starting from the largest factorial available in the precomputed table. We resort to this technique since Stirling's approximation, considering the approximation up to the eight term of the series, was proven to be lacking accuracy in the determination of extremely low DFR values (i.e., $\leq 2^{-100}$). Namely, employing Stirling's approximation lead to excessively conservative DFR estimates.
- We chose to precompute all the values of the factorials up to 90000!, 150000! and 250000! to perform the parameter design for Category 1, 3 and 5, and limit the maximum code lengths accordingly. The values were chosen so that Stirling's approximation is not used when the

final code parameter results are obtained. We note that this results in a memory occupation of around 6.9 GiB, 19.7 GiB, and 57.4 GiB per optimization process, respectively. The value can be tuned to accomodate parameter generation replication for smaller parameter sets with less demanding hardware.

- We left Stirling's approximation available in the public code to provide also reproducibility of the previous parameter designs.
- All floating point values are memorized and manipulated as 704-bits mantissas (≈ 200 decimal digits) and 64-bit exponents, as per provided by the NTL library.

4.1 Parameters for LEDAcrypt-KEM and LEDAcrypt-PKC

In this section, we present the code parameters for IND-CCA2 LEDAcrypt-KEM and IND-CCA2 LEDAcrypt-PKC, evaluating the engineering tradeoff given by performing two, one or no out-of-place iterations in a decoder, preceded by a number of in-place iterations such that the total decoder iteration number is fixed at two. The choice to pick such a low number of iterations allows the implementation of the decoder to benefit from a very fast, constant time execution, and allow us to employ our DFR estimation techniques in a regime where they provide a close, conservative, fit to the values which can be obtained through numerical simulations, in turn keeping a reasonable size of the parameter design space.
To provide a numerical confirmation of such a fact, we report the outcome of numerical simulations on a $p = 4801$, $n_0 = 2$, $v = 45$ QC-LDPC/QC-MDPC code in Figure 4.1.



(a) $\mathtt{imax}^{\mathtt{In}} = 2$ $\mathtt{imax}^{\mathtt{Out}} = 0, \mathtt{th}_0 = 26$ (b) $\mathtt{imax}^{\mathtt{In}} = 1$ $\mathtt{imax}^{\mathtt{Out}} = 1, \mathtt{th}_0 = 27$ (c) $\mathtt{imax}^{\mathtt{In}} = 0$ $\mathtt{imax}^{\mathtt{Out}} = 2, \mathtt{th}_0 = 25$

Figure 4.1: Comparison of our worst-case DFR estimation technique with numerical simulations, for the LEDAdecoder choices employed to provide predictable DFR for IND-CCA primitives (i.e., $\mathtt{imax}^{\mathtt{In}} + \mathtt{imax}^{\mathtt{Out}} = 2$). Results obtained on a QC-LDPC/QC-MDPC code with parameters $n_0 = 2, p = 4801, v = 45$ (taking $t = 45$, it exhibits a complexity of $\approx 2^{96}$ classical operations or $\approx 2^{64}$ quantum gates against the best SDP/CFP solvers) employing thresholds $\mathtt{th}_0$ for the first iteration computation found computing all the DFR estimates according to Chapter 3, and picking the first iteration thresholds providing the best DFR for each case. The numerical DFR simulation is obtained decoding random error vectors of weight t, until either 100 decoding errors are reached, or until 10^8 decoding operations have been performed, whichever happens first.

The code parameters were chosen to obtain an acceptable simulation time (from $2 \cdot 10^3$ to $5 \cdot 10^3$ core hours per data set). For the record, these parameters require the same effort to solve either the Syndrome Decoding Problem (SDP) or one of the Codeword Finding Problems (CFPs) reported in Chapter 2, and allow the code to exhibit a resistance of $\approx 2^{96}$ classical operations or $\approx 2^{64}$ quantum gates when $t = 45$.

In order to choose the bit-flipping thresholds for the first iteration to be used in these numerical simulations, we employed the DFR estimation techniques reported in Chapter 3, computing the DFR for all the possible threshold values in $\{\lceil \frac{v+1}{2} \rfloor, \ldots, v\}$, and picking the one providing the best estimated DFR. The second iteration threshold is chosen with the code-specific criterion reported in Section 3.4, for the LEDAdecoder with $\mathtt{imax}^{\mathtt{Out}} > 0$, performing a rejection sampling at key generation time so to achieve a number of errors corrected with certainty by the last out-of-place iteration $\tau = 7$. For the case of the $\mathtt{imax}^{\mathtt{In}} = 2, \mathtt{imax}^{\mathtt{Out}} = 0$ LEDAdecoder, we simply picked the second iteration threshold equal to the one for the first iteration. As it can be seen in Figure 4.1, our worst case DFR estimation techniques provide a good fit of the decoder behaviour, while retaining a conservative estimate of the actual decoder DFR.

Table 4.1 and Table 4.2 report the values of the parameters of the codes designed to be used with a LEDAdecoder employing ($\mathtt{imax}^{\mathtt{In}} = 0$, $\mathtt{imax}^{\mathtt{Out}} = 2$) and ($\mathtt{imax}^{\mathtt{In}} = 1$, $\mathtt{imax}^{\mathtt{Out}} = 1$), respectively. Specifically, for each NIST category, guaranteed DFR values and number of circulant blocks $n_0 \in \{2, 3, 4\}$, the tables report: the size p of the circulant blocks of H, the weight v of a column of H, the number t of intentional errors, and the value of τ (weight of error corrected with certainty in a single iteration) to be employed during the *rejection sampling* phase in the keyGeneration procedure, and the ratio of keypairs accepted on average during an execution of the KeyGeneration procedure. Furthermore, for each NIST category of the reported codes, the tables also show the base-2 logarithm of the estimated computational cost of the best algorithms for solving the following problems on a classical and a quantum computer (denoted with the prefixes C- and Q-, respectively), described in Chapter 2: syndrome decoding problem for key recovery (SDP); codeword finding problem in the hidden LDPC/MDPC code (CFP-1); codeword finding problem in the length-reduced hidden LDPC/MDPC code (CFP-2); codeword finding in the dual code of the hidden code (CFP-3).
A noteworthy point is that the third strategy of key recovery attack (i.e., the one denoted as CFP-3) is effectively the one being the most effective for all the parameter sets having n_0 equal to either 3 or 4, while the key recovery attack complexities are almost all the same for $n_0 = 2$ bar for a single unit advantage of the codeword finding problem in length-reduced hidden LDPC/MDPC code (CFP-2), which is easily ascribed to an extra factor 2 in the complexity denominator (see Section 2.3.1). In Table 4.1, we observe a significant reduction in the size of the p parameter, with respect to the parameter sets proposed in the LEDAcrypt specification at the beginning of the second round of the NIST PQC contest. Note that such parameters were proposed only for the case $n_0 = 2$ employing a LEDAdecoder with a number of iterations with $\mathtt{imax}^{\mathtt{In}} = 0$, and $\mathtt{imax}^{\mathtt{Out}} = 2$. The reason for the $\approx 30\%$ decrease in the size of the p parameter for the sets with DFR equal to 2^{-64}, and the $\approx 50\%$ decrease in size for the sets with DFR equal to $2^{-\lambda}$ is due to a combined effect of the joint optimization of p, t, and v provided by the new parameter design procedure, and the higher correction power of a sparse code where $L = HQ$ and $Q = I$. The former effect can be seen from the slight decrease of the values of t and v, which previously was kept unchanged from the one for ephemeral key use parameters. The latter effect is evident observing that the values reported for τ in the LEDAcrypt specification provided for the second round of the NIST PQC contest, were in the 4–6 range, while the current values for τ are in the 9–18 range, indicating that more errors

with more than twice the weight can be corrected with certainty by the given code parameters.

Table 4.3 reports the values of the parameters of the codes designed to be used with a LEDAdecoder employing ($\mathsf{imax}^{\mathsf{In}} = 2$, $\mathsf{imax}^{\mathsf{Out}} = 0$). A notable difference of this table w.r.t. Table 4.1 and Table 4.2 is that it does not show the values of the parameter τ and the percentage of keypairs accepted during the execution of the KeyGeneration procedure, because the DFR of the codes are conservatively estimated for the two executions of the in-place iteration function applying the results of the analysis in Section 3.2.

Comparing the results in Table 4.1 (decoder with $\mathsf{imax}^{\mathsf{In}} = 0$, $\mathsf{imax}^{\mathsf{Out}} = 2$), Table 4.2 (decoder with $\mathsf{imax}^{\mathsf{In}} = 1$, $\mathsf{imax}^{\mathsf{Out}} = 1$), and Table 4.3 (decoder with $\mathsf{imax}^{\mathsf{In}} = 2$, $\mathsf{imax}^{\mathsf{Out}} = 0$) we can observe a steady decrease in the code length due to the use of in-place iteration functions. Such a decrease is more evident in the case of $\mathsf{imax}^{\mathsf{In}} = 2$, since the DFR estimate in that case is not employing the result from Section 3.4 to estimate the correction power of the last iteration, instead employing the method in Section 3.2. Indeed, when two in-place iterations are considered, instead of assuming that a decoding failure takes place whenever the first in-place iteration leaves more than τ errors, the analysis in Section 3.2 considers the non-unitary probability of decoding in such a case too, yielding a tighter DFR estimate.

Table 4.1: Parameter sizes for IND-CCA2 LEDAcrypt-KEM, and IND-CCA2 LEDAcrypt-PKC, with an $\texttt{imax}^{\texttt{In}}$=0, $\texttt{imax}^{\texttt{Out}}$=2 LEDADECODER executing **two times the out-of-place iteration function**. Attack complexities are reported as the base-2 logarithm of the estimated computational cost of the best algorithm for solving the following problems on a classical and a quantum computer (denoted with the prefixes C- and Q-, respectively).
SDP: syndrome decoding problem for key recovery;
CFP-1: codeword finding problem in the hidden LDPC/MDPC code;
CFP-2: codeword finding problem in the length-reduced hidden LDPC/MDPC code;
CFP-3: codeword finding on dual code of the of hidden code.
All attack complexities are equal or greater to the ones mandated by NIST categories as per Chapter 2, Table 2.1

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> **143.5**)	Q-SDP (> **81.2**)	C-CFP-1 (> **143.5**)	Q-CFP-1 (> **81.2**)	C-CFP-2 (> **143.5**)	Q-CFP-2 (> **81.2**)	C-CFP-3 (> **143.5**)	Q-CFP-3 (> **81.2**)
1	2^{-64}	**2**	23,371	71	130	10	85.3	144.2	95.7	148.3	94.4	147.3	93.4	148.3	94.4
		3	16,067	79	83	9	86.5	145.3	96.1	250.8	146.4	161.0	99.7	145.7	93.9
		4	13,397	83	66	8	93.7	145.4	96.3	329.3	186.1	167.9	102.7	145.3	94.4
	2^{-128}	**2**	28,277	69	129	11	68.2	144.8	96.4	145.0	92.9	144.0	91.97	145.0	92.9
		3	19,709	79	82	10	66.5	144.6	96.1	251.3	146.9	161.5	100.2	146.2	94.5
		4	16,229	83	65	9	63.5	144.2	96.1	329.6	186.5	168.3	103.2	145.8	95.0

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> **208.0**)	Q-SDP (> **113.4**)	C-CFP-1 (> **208.0**)	Q-CFP-1 (> **113.4**)	C-CFP-2 (> **208.0**)	Q-CFP-2 (> **113.4**)	C-CFP-3 (> **208.0**)	Q-CFP-3 (> **113.4**)
3	2^{-64}	**2**	40,787	103	195	13	81.9	208.9	130.2	210.8	127.5	209.8	126.5	210.8	127.5
		3	28,411	117	124	11	99.0	209.0	130.2	369.8	207.8	235.5	138.8	210.8	128.4
		4	22,901	123	98	11	61.4	208.0	129.8	487.8	267.3	246.4	143.9	210.2	128.7
	2^{-192}	**2**	52,667	103	195	15	59.8	208.7	130.6	211.5	128.2	210.5	127.2	211.5	128.2
		3	36,629	115	123	13	81.9	208.3	130.2	364.0	205.2	232.1	137.5	208.0	127.3
		4	30,803	123	98	12	75.9	209.0	130.8	488.2	267.9	246.9	144.6	210.8	129.5

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> **272.3**)	Q-SDP (> **145.6**)	C-CFP-1 (> **272.3**)	Q-CFP-1 (> **145.6**)	C-CFP-2 (> **272.3**)	Q-CFP-2 (> **145.6**)	C-CFP-3 (> **272.3**)	Q-CFP-3 (> **145.6**)
5	2^{-64}	**2**	61,717	137	261	17	55.2	273.0	163.8	277.7	162.3	276.7	161.3	277.7	162.3
		3	42,677	153	165	14	96.4	273.1	163.8	482.9	265.7	306.5	175.7	273.0	160.9
		4	35,507	163	131	13	95.9	273.2	164.0	646.9	348.2	325.4	184.8	275.5	162.9
	2^{-256}	**2**	83,579	135	260	18	51,6	273.1	164.4	274.6	161.2	273.6	160.2	274.6	161.2
		3	58,171	153	165	16	65,2	274.1	164.8	483.5	266.4	307.1	176.5	273.7	161.6
		4	48,371	161	131	15	78,9	274.3	165.1	647.2	348.9	325.9	185.5	276.2	163.6

Table 4.2: Parameter sizes for IND-CCA2 LEDAcrypt-KEM, and IND-CCA2 LEDAcrypt-PKC, with an $\mathtt{imax}^{\mathtt{In}}$=1, $\mathtt{imax}^{\mathtt{Out}}$=1 LEDADecoder executing **one time the in-place iteration function followed by one execution of the out-of-place iteration function**. Attack complexities are reported as the base-2 logarithm of the estimated computational cost of the best algorithm for solving the following problems on a classical and a quantum computer (denoted with the prefixes C- and Q-, respectively).
SDP: syndrome decoding problem for key recovery;
CFP-1: codeword finding problem in the hidden LDPC/MDPC code;
CFP-2: codeword finding problem in the length-reduced hidden LDPC/MDPC code;
CFP-3: codeword finding on dual code of the of hidden code.
All attack complexities are equal or greater to the ones mandated by NIST categories as per Chapter 2, Table 2.1

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> 143.5)	Q-SDP (> 81.2)	C-CFP-1 (> 143.5)	Q-CFP-1 (> 81.2)	C-CFP-2 (> 143.5)	Q-CFP-2 (> 81.2)	C-CFP-3 (> 143.5)	Q-CFP-3 (> 81.2)
1	2^{-64}	2	21,701	71	130	10	75.56	144.0	95.4	148.1	94.2	147.1	93.2	148.1	94.2
		3	15,053	79	83	9	75.74	145.1	95.9	250.7	146.2	160.9	99.5	145.5	93.7
		4	12,739	83	66	8	89.69	145.2	96.2	329.2	186.0	167.8	102.6	145.2	94.3
	2^{-128}	2	27,077	69	130	11	62.61	144.7	96.2	148.6	94.8	147.6	93.8	148.6	94.8
		3	19,541	79	82	10	45.10	144.5	96.0	251.2	146.8	161.4	100.1	146.1	94.4
		4	15,773	83	65	9	53.33	144.2	96.0	329.6	186.5	168.3	103.2	145.8	94.9

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> 208.0)	Q-SDP (> 113.4)	C-CFP-1 (> 208.0)	Q-CFP-1 (> 113.4)	C-CFP-2 (> 208.0)	Q-CFP-2 (> 113.4)	C-CFP-3 (> 208.0)	Q-CFP-3 (> 113.4)
3	2^{-64}	2	37,813	103	196	13	70.6	208.6	130.0	210.7	127.3	209.7	126.3	210.7	127.3
		3	26,597	117	124	11	97.4	208.8	129.9	369.7	207.6	235.4	138.7	210.7	128.2
		4	22,229	123	98	11	57.0	208.0	129.7	487.8	267.2	246.4	143.8	210.1	128.7
	2^{-192}	2	50,363	103	195	15	50.1	208.7	130.5	211.4	128.1	210.4	127.1	211.4	128.1
		3	35,419	115	123	13	73.8	208.3	130.1	370.2	208.3	236.0	139.4	211.3	129.0
		4	29,221	123	98	13	61.2	209.0	130.7	488.2	267.9	246.9	144.5	210.7	129.4

NIST Cat.	DFR	n_0	p	v	t	τ	key % ok	C-SDP (> 272.3)	Q-SDP (> 145.6)	C-CFP-1 (> 272.3)	Q-CFP-1 (> 145.6)	C-CFP-2 (> 272.3)	Q-CFP-2 (> 145.6)	C-CFP-3 (> 272.3)	Q-CFP-3 (> 145.6)
5	2^{-64}	2	58,171	137	262	17	53.1	273.8	164.1	277.6	162.2	276.6	161.2	277.6	162.2
		3	40,387	155	165	14	90.8	274.5	164.3	489.1	268.7	310.3	177.5	276.3	162.4
		4	33,493	163	131	13	88.8	273.0	163.8	646.8	348.1	325.3	184.7	275.4	162.7
	2^{-256}	2	81,773	135	260	17	87.0	273.0	164.3	274.5	161.1	273.5	160.1	274.5	161.1
		3	56,731	153	165	16	52.9	274.1	164.8	483.5	266.4	307.1	176.4	273.7	161.6
		4	47,459	161	131	15	73.0	274.2	165.0	647.2	348.9	325.9	185.5	276.1	163.6

Table 4.3: Parameter sizes for IND-CCA2 LEDAcrypt-KEM, and IND-CCA2 LEDAcrypt-PKC, with an $\mathsf{imax}^{\mathsf{In}}$=2, $\mathsf{imax}^{\mathsf{Out}}$=0 LEDADECODER executing **two times the in-place iteration function**. Attack complexities are reported as the base-2 logarithm of the estimated computational cost of the best algorithm for solving the following problems on a classical and a quantum computer (denoted with the prefixes C- and Q-, respectively).
SDP: syndrome decoding problem for key recovery;
CFP-1: codeword finding problem in the hidden LDPC/MDPC code;
CFP-2: codeword finding problem in the length-reduced hidden LDPC/MDPC code;
CFP-3: codeword finding on dual code of the of hidden code.
All attack complexities are equal or greater to the ones mandated by NIST categories as per Chapter 2, Table 2.1

NIST Cat.	DFR	n_0	p	v	t	C-SDP (> **143.5**)	Q-SDP (> **81.2**)	C-CFP-1 (> **143.5**)	Q-CFP-1 (> **81.2**)	C-CFP-2 (> **143.5**)	Q-CFP-2 (> **81.2**)	C-CFP-3 (> **143.5**)	Q-CFP-3 (> **81.2**)
1	$\mathbf{2^{-64}}$	**2**	15,461	71	132	144.9	95.4	147.4	93.3	146.4	92.3	147.4	93.3
		3	10,891	79	83	144.1	94.9	250.3	145.5	160.3	98.8	144.9	93.0
		4	9,283	83	66	144.3	95.2	329.0	185.3	167.4	101.9	144.7	93.6
	$\mathbf{2^{-128}}$	**2**	19,813	71	131	144.7	95.7	147.9	94.0	146.9	93.0	147.9	94.0
		3	14,051	79	83	145.0	95.8	250.6	146.1	160.8	99.4	145.4	93.6
		4	11,909	83	66	145.1	96.1	329.2	185.9	167.8	102.5	145.2	94.2

NIST Cat.	DFR	n_0	p	v	t	C-SDP (> **208.0**)	Q-SDP (> **113.4**)	C-CFP-1 (> **208.0**)	Q-CFP-1 (> **113.4**)	C-CFP-2 (> **208.0**)	Q-CFP-2 (> **113.4**)	C-CFP-3 (> **208.0**)	Q-CFP-3 (> **113.4**)
3	$\mathbf{2^{-64}}$	**2**	27,701	103	197	208.7	129.4	210.0	126.5	209.0	125.5	210.0	126.5
		3	19,949	117	125	209.5	129.8	369.3	207.0	234.9	138.0	210.1	127.6
		4	16,747	123	99	209.0	129.8	487.6	266.7	246.0	143.2	209.6	128.0
	$\mathbf{2^{-192}}$	**2**	38,069	103	196	208.7	130.0	210.7	127.4	209.7	126.4	210.7	127.4
		3	27,179	117	124	209.0	130.1	369.8	207.7	235.5	138.8	210.7	128.4
		4	22,853	123	98	208.1	129.9	487.9	267.3	246.4	143.9	210.2	128.8

NIST Cat.	DFR	n_0	p	v	t	C-SDP (> **272.3**)	Q-SDP (> **145.6**)	C-CFP-1 (> **272.3**)	Q-CFP-1 (> **145.6**)	C-CFP-2 (> **272.3**)	Q-CFP-2 (> **145.6**)	C-CFP-3 (> **272.3**)	Q-CFP-3 (> **145.6**)
5	$\mathbf{2^{-64}}$	**2**	43,397	137	263	273.9	163.7	276.9	161.4	275.9	160.4	276.9	161.4
		3	31,069	155	166	273.7	163.5	488.8	268.2	309.9	177.0	275.8	161.8
		4	26,251	163	132	274.2	164.0	645.4	347.7	325.0	184.2	275.0	162.2
	$\mathbf{2^{-256}}$	**2**	61,211	137	261	273.0	163.8	277.7	162.3	276.7	161.3	277.7	162.3
		3	43,499	153	165	273.2	163.9	483.0	265.8	306.6	175.8	273.1	160.9
		4	36,877	163	131	273.4	164.2	646.9	348.3	325.5	184.9	275.6	163.0

Table 4.4: Parameter sizes for the LEDAcrypt-KEM-CPA cryptosystem together with the related attack complexities – tailored for the scenario where ephemeral private/public keypairs are desired. Each parameter set is chosen targeting a **DFR**=$\mathbf{10^{-9}}$. Experimentally, the execution of the encapsulation and decapsulation procedures, for a given keypair, 10^9 times exhibits zero decryption/decoding failures. The column labeled as $\mathtt{imax}^{\mathtt{Out}}$ shows the maximum number of times that the out-of-place iteration function is executed by the LEDADECODER.
Attack complexities are reported as the base-2 logarithm of the estimated computational cost of the best algorithm for solving the following problems on a classical and a quantum computer (denoted with the prefixes C- and Q-, respectively).
SDP: syndrome decoding problem for key recovery;
CFP-1: codeword finding problem on hidden LDPC/MDPC code;
CFP-2: codeword finding problem on length-reduced hidden LDPC/MDPC code;
CFP-3: codeword finding on dual of hidden code.
All attack complexities are equal or greater to the ones mandated by NIST categories as per Chapter 2, Table 2.1

NIST Cat.	n_0	p	v	t	$\mathtt{imax}^{\mathtt{Out}}$	C-SDP (> **143.5**)	Q-SDP (> **81.2**)	C-CFP-1 (> **143.5**)	Q-CFP-1 (> **81.2**)	C-CFP-2 (> **143.5**)	Q-CFP-2 (> **81.2**)	C-CFP-3 (> **143.5**)	Q-CFP-3 (> **81.2**)
	2	10,853	71	133	6	144.8	94.7	146.6	92.4	145.6	91.4	146.6	92.4
1	**3**	8,237	79	84	5	144.8	94.7	250.0	144.8	159.9	98.1	144.4	92.3
	4	7,187	83	67	4	145.3	95.3	328.2	184.8	167.1	101.3	144.2	92.9

NIST Cat.	n_0	p	v	t	$\mathtt{imax}^{\mathtt{Out}}$	C-SDP (> **208.0**)	Q-SDP (> **113.4**)	C-CFP-1 (> **208.0**)	Q-CFP-1 (> **113.4**)	C-CFP-2 (> **208.0**)	Q-CFP-2 (> **113.4**)	C-CFP-3 (> **208.0**)	Q-CFP-3 (> **113.4**)
	2	20,981	103	198	6	208.8	129.0	209.4	125.8	208.4	124.8	209.4	125.8
3	**3**	15,331	117	125	5	208.7	128.9	369.1	206.4	234.6	137.4	209.6	126.9
	4	13,109	123	99	4	208.2	128.9	485.1	266.2	245.8	142.7	209.2	127.4

NIST Cat.	n_0	p	v	t	$\mathtt{imax}^{\mathtt{Out}}$	C-SDP (> **272.3**)	Q-SDP (> **145.6**)	C-CFP-1 (> **272.3**)	Q-CFP-1 (> **145.6**)	C-CFP-2 (> **272.3**)	Q-CFP-2 (> **145.6**)	C-CFP-3 (> **272.3**)	Q-CFP-3 (> **145.6**)
	2	35,117	137	263	4	273.3	163.0	276.5	160.9	275.5	159.9	276.5	160.9
5	**3**	25,579	155	166	4	273.1	162.8	488.7	267.8	309.7	176.5	275.4	161.4
	4	21,611	163	132	4	273.6	163.3	644.9	347.3	324.9	183.7	274.7	161.7

4.2 Parameters for LEDAcrypt-KEM-CPA

Table 4.4 reports the code parameters obtained for LEDAcrypt-KEM-CPA, and the relative work factor of the four attacks considered, solving the SDP to recover a message, and the three CFP solutions for key recovery, with the fastest ISD technique available. We note that, in our complexity estimation we found out that the quantum variant of the Stern's algorithm as described by de Vries [22] does not achieve an effective speedup when compared against a quantum transposition of Lee and Brickell's ISD. Such a result can be ascribed to the fact that the speedup obtained by the reduction in the number of ISD iterations which can be obtained by Stern's ISD is mitigated by the fact that the overall number of iterations to run is quadratically reduced by applying Grover's algorithm to execute them. This result effectively matches the point made at the end of [13].

The parameters in the table are obtained as the output of the parameter design procedure described by Algorithm 17, where the value of p is obtained through a simple heuristic. We subsequently simulate the decoding of 10^9 syndromes obtained from randomly generated error vectors to estimate the code DFR for each parameter set, and increase the value of p alone by 5% until no more decoding errors are reported. During this procedure we needed to increase the value of p only in the category 1 parameters, raising it by 5% in the $n_0 = 2$ case, and by 10% in the $n_0 = 3$ and $n_0 = 4$ cases.

The parameter sets for LEDAcrypt-KEM-CPA provide slightly smaller values of p than the one present in the round-2 submission of LEDAcrypt. Indeed, the size improvement of 5% to 25%, depending on the category and code rate is due to the greater correction power of the sparse code in case the Q matrix is taken to be an identity, as we are doing to prevent weak keys attacks. Table 4.4 also reports the maximum observed number of iterations during the decoding, although we point out that most ($> 50\%$) of the decoding processes terminate before the stated $\mathtt{imax}^{\mathtt{Out}}$.

Chapter 5

Optimized LEDAcrypt Implementation

In this chapter, we will report the strategies employed to realize an efficient implementation of the LEDAcrypt primitives, together with quantitative motivations for their choices. We employ, as an experimental assessment platform, a desktop machine equipped with an Intel Core i5-6600, clocked at 3.2 GHz, and endowed with 16 GiB of DDR-4, running Debian GNU/Linux 10.3 for `amd64` architectures. The `C11` source code is compiled with `gcc` 8.3.0, specifying the highest optimization level (`-O3`). The number of clock cycles is measured via the `rtdscp` instruction, which provides an integrated instruction barrier for the instructions in flight in the pipeline. All the timings reported are the averages of 1000 executions. All the measured implementations are optimized employing the Intel supplied compiler intrinsics for the AVX2 ISA extensions, available in the Intel Haswell ISA, as per NIST indications. Our aim is to obtain a constant time codebase for the IND-CCA LEDAcrypt primitives, while the codebase for LEDAcrypt-KEM-CPA is optimized without considering constant-time constraints, as its ephemeral key use scenario does not allow to exploit the variable timing leakage.

In particular, in this chapter, we analyze:

- the results of the experimental evaluation of the computational performance of LEDAdecoder, for the choices of $\mathsf{imax}^{\mathsf{In}}$ and $\mathsf{imax}^{\mathsf{Out}}$ described in Chapter 4, motivating our choice of $\mathsf{imax}^{\mathsf{In}} = 0, \mathsf{imax}^{\mathsf{Out}} = 2$ from the efficiency of this LEDAdecoder variant;
- the strategies adopted for both constant- and variable-time binary polynomial multiplications, where we select a Sparse-By-Dense multiplication strategy for variable time multiplications, and a Dense-By-Dense multiplication strategy for constant time execution, as we measure them to be the most efficient in our implementations;
- the strategy to implement the LEDAdecoder employing the *bit-slicing* technique to obtain a fully constant time and highly parallel decoding strategy;
- a set of four different modular polynomial inversion strategies, picking the most efficient one in our scenario, i.e. the approach relying on Fermat's little theorem, specialized for the $x^p + 1$ modulus;

- the tradeoffs which can be achieved employing a value of p with low Hamming weight, speeding up the modular inverse computation, at the cost of increasing multiplication and decoding time for the case of LEDAcrypt-KEM-CPA.

5.1 Selection of the LEDAdecoder strategy

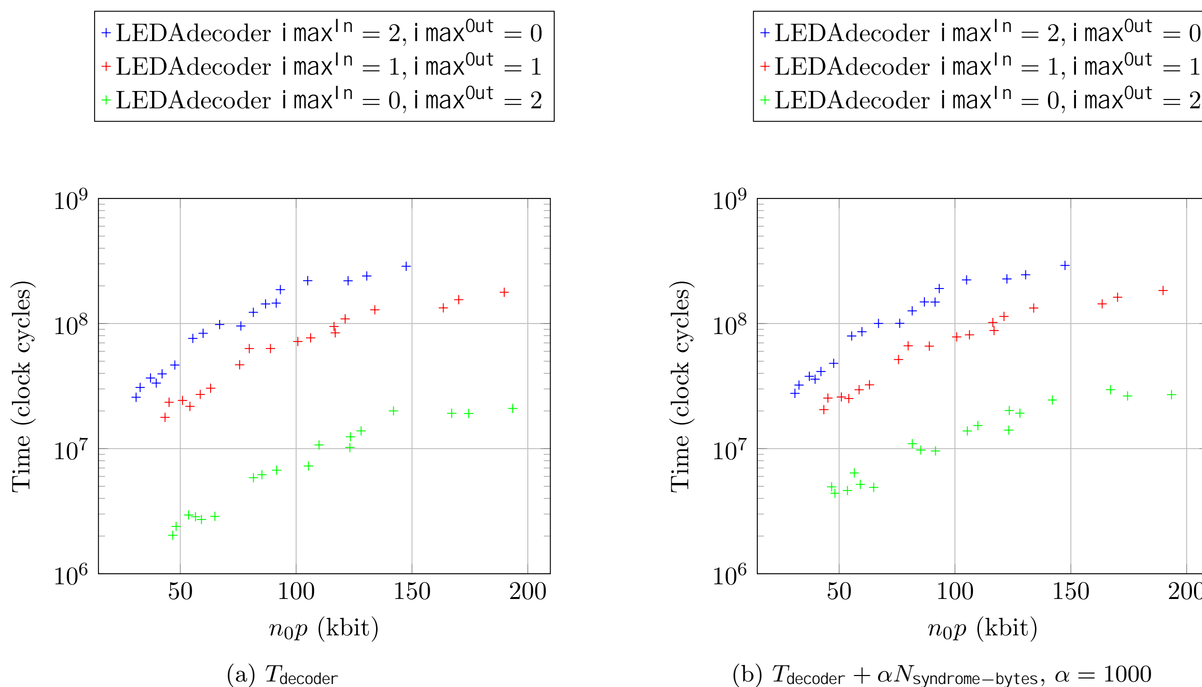
The figures of merit to be considered in the performance evaluation of LEDAcrypt systems with the QC-MDPC code configurations and parameters reported in Tables 4.1, 4.2 and 4.3 for the IND-CCA2 LEDAcrypt-KEM and IND-CCA2 LEDAcrypt-PKC as well as with the ones in Table 4.4 for LEDAcrypt-KEM-CPA (proposed for use-cases where ephemeral public/private keypairs are employed), must take into account both the execution times of optimized implementations targeting x86_64 platforms and the byte-size of data exchanged between sender and receiver. Assuming both sender and receiver equipped with two instances of the same computational platform, the definition of the figures of merit must distinguish between the implementations of IND-CCA2 systems, with their strict requirement to be immune to timing analyses (a.k.a. constant-time implementations), and the ephemeral-key system which is allowed to run also non constant-time implementations of its primitives.

Specifically, for the IND-CCA2 LEDAcrypt-KEM (resp., LEDAcrypt-PKC) system, the first figure of merit is defined as the sum of the execution timings of the ENCAP (resp, ENCRYPT) and DECAP (resp. DECRYPT) algorithms, both measured as the required number of machine clock cycles, i.e.: $T_{\mathsf{KEM}} = T_{\mathsf{Encap}} + T_{\mathsf{Decap}}$ (resp. $T_{\mathsf{PKC}} = T_{\mathsf{Encrypt}} + T_{\mathsf{Decrypt}}$). For the LEDAcrypt-KEM-CPA, the first figure of merit is defined as the sum of the execution timings of the KEYGENERATION, ENCAP and DECAP algorithms, measured as the required number of machine clock cycles, i.e.: $T_{\mathsf{KEM-CPA}} = T_{\mathsf{KeyGen}} + T_{\mathsf{Encap}} + T_{\mathsf{Decap}}$.

A second figure of merit that keeps into account also the computational overhead experienced at both the receiver and the sender side to process the byte-streams exchanged over the network is defined by adding to the total execution time (i.e., T_{KEM}, T_{PKC} or $T_{\mathsf{KEM-CPA}}$, respectively) also the total number of transmitted bytes, $\mathsf{N}_{\mathsf{bytes}}$, multiplied by a normalization coefficient α (measured as machine clock cycles over bytes). In particular for the IND-CCA2 LEDAcrypt-KEM (resp., LEDAcrypt-PKC) system, the second figure of merit is defined as $T_{\mathsf{combined-KEM}} = T_{\mathsf{KEM}} + \alpha \cdot \mathsf{N}_{\mathsf{bytes}}$ (resp. $T_{\mathsf{combined-PKC}} = T_{\mathsf{PKC}} + \alpha \cdot \mathsf{N}_{\mathsf{bytes}}$), while for the LEDAcrypt-KEM-CPA is defined as $T_{\mathsf{combined-KEM-CPA}} = T_{\mathsf{KEM-CPA}} + \alpha \cdot \mathsf{N}_{\mathsf{bytes}}$. An empirical value for the coefficient α, which is usually accepted as well-modeling the computational overhead introduced by the execution of the routines in the software-stack employed for communication, is $\alpha = 1000$.

In every LEDAcrypt instance the foremost role is played by the decoding procedure designed in Chapter 3, as Algorithm 12. Indeed, for IND-CCA2 systems, the performance of the KEYGENERATION algorithm is a (relatively) limited concern as it does not influence the aforementioned figures of merit. As a consequence, the most performance demanding algorithm w.r.t. the total execution time T_{KEM} (resp., T_{PKC}) is the QC-MDPC decoder (see Algorithm 12).

In the case of the LEDAcrypt-KEM-CPA system, the parameter design took advantage of the requirement of only a practically usable DFR (i.e., DFR= 10^{-9}), which in turn can be determined via numerical simulations and not via closed-form formulae. The decoder implemented for the LEDAcrypt-KEM-CPA system is a variable iteration number decoder, where the maximum number of iterations was capped experimentally to the highest one obtained in the numerical simulations.



(a) T_{decoder}

(b) $T_{\mathsf{decoder}} + \alpha N_{\mathsf{syndrome-bytes}}$, $\alpha = 1000$

Figure 5.1: Performance of each LEDADECODER variant as a function of the error vector sizes (n_0p) which correspond to the code configurations reported in Tables 4.1, 4.2, 4.3 in Section 4.1 (for each NIST category and $n_0 \in \{2, 3, 4\}$)

The LEDAcrypt-KEM-CPA decoder employs only out-of-place iterations (see Section 5.3) to minimize the decoder latency, and exploits the syndrome-weight dependent threshold determination rule described in Section 3.5. Such a threshold selection criterion is experimentally determined to attain a higher correction capability w.r.t. a fixed threshold choice depending only on the number of computed iterations. This, in turn, allows to further reduce the key size, improving the performance figures of merit. An implementation analysis that keeps into account the aforementioned figures of merit to perform a further tuning of LEDAcrypt-KEM-CPA parameters is presented in Section 5.5.

In the following we analyze the execution timings of the three LEDADECODER variants designed and described in Section 4.1 for IND-CCA2 instances of LEDAcrypt, as well as the figures of merit obtained by adding to the execution timings the number of bytes encoding the syndrome multiplied by a normalization coefficient α, with $\alpha = 1000$. Such a choice is going to evaluate better the performance differences among the three variants of the LEDADECODER, highlighting the impact of the parameter sizes derived from the DFR analyses presented in the previous chapters also on the size of the transmitted syndrome. It is worth noting that all three variants of the LEDADECODER provide strong IND-CCA2 security guarantees featuring closed-form formulae to assess the DFR provided by the execution of two out-of-place iteration functions, of one out-of-place iteration function followed by one in-place iteration function, or of two in-place iteration functions.

It is useful to note that the in-place decoding strategy does not allow to parallelize the computation of its intermediate values, namely, the unsatisfied parity check counts, as the computation of each one of them depends on the result of the computation of the previous one, in the decoder processing

order. Figure 5.1(a) compares the execution times of each decoding strategy as a function of the values of the circulant-block size p of the underlying code. Figure 5.1(b) takes into account also the byte-size of the syndrome. Conservatively, Figure 5.1 shows the performance of the three LEDAdecoder variants employing a constant-time implementation of the out-of-place iteration function, and non constant-time implementation (and thus faster) of the in-place iteration function. As it is clear, from Figure 5.1, the most efficient LEDAdecoder strategy executes two times an out-of-place iteration function. Indeed, despite the computational advantage of a non-constant time implementation the strategies employing in-place iterations are significantly slower than the purely out of place one. It is interesting to note, though, that the reduced data being processed by them (as a result of the smaller values of p, thanks to the higher correction power of an in-place iteration), effectively causes a reduction in the gap in the combined execution time-transmitted data metric.

On the basis of these evaluations, **we implement the IND-CCA2 LEDAcrypt-KEM and the IND-CCA2 LEDAcrypt-PKC systems keeping the code configurations and parameters shown in Table 4.1 in Section 4.1 and a decoder executing two out-of-place iteration functions**.

5.2 Binary Polynomial Multiplications

The KEY-GENERATION, ENCRYPT, and DECRYPT algorithms of LEDAcrypt-KEM-CPA, LEDAcrypt-KEM and LEDAcrypt-PKC primitives require at several steps the computation of binary polynomial multiplications that have been implemented following two different strategies. Given two binary invertible polynomials $f(x), g(x) \in \mathbb{Z}_2[x]/(x^p - 1)$, the former strategy, named SPARSE-BY-DENSE MULTIPLICATION, assumes the first factor $f(x)$ to be represented as the set of its asserted-bit positions, i.e., $f(x) = \{b_0, b_1, \dots, b_{s-1}\}$, where $s \geq 1$, $b_i \in \{0, 1, \dots, p-1\}$, $0 \leq i < s$, while the latter, named DENSE-BY-DENSE MULTIPLICATION, assumes both $f(x)$ and $g(x)$ to be represented as p-bit strings.

The implementation of the DENSE-BY-DENSE MULTIPLICATION exploits the availability of the *carryless multiplication instruction* (e.g., pclmulqdq on x86_64 platforms, pmull on ARM platforms implementing the Armv8-A profile), which performs a polynomial multiplication of two 64-bit elements into a 128-bit one. In particular, the said algorithm takes as input two binary polynomials $f(x), g(x) \in \mathbb{Z}_2[x]/(x^p - 1)$, each of which encoded with p bits (where p ranges from $\approx$ 7k to $\approx$ 80k bits). Then, to compute the result $d(x) = f(x) \cdot g(x) \in \mathbb{Z}_2[x]/(x^p - 1)$, it performs the computation of the polynomial $c(x) = f(x) \times g(x)$, encoded with $2 \times p$ bits, followed by the execution of the modulo $(x^p - 1)$ operation, which in turn consists in computing a bit-wise xor between the most significant p-bits of $c(x)$ and the least significant p-bits of $c(x)$. All the optimizations implemented to speed up the execution of the DENSE-BY-DENSE MULTIPLICATION focus on the execution of the product $c(x) = f(x) \times g(x)$.
We employed a careful combination of the well-known Karatsuba and Toom-Cook-3 divide-and-conquer strategies, tailored on the bitsize range of the LEDAcrypt polynomials, to take advantage of their $O(n^{\log_2(3)})$ and $O(n^{\log_3(5)})$ asymptotic complexities, respectively, where n denotes the bit-size of the factors. In a nutshell, the Toom-Cook-3 method splits up the factors in three parts and, instead of multiplying the obtained smaller polynomials directly, it evaluates them at a set of points and multiplies together the values obtained from such evaluations. Based on the values computed at the said points the final polynomial is interpolated. We applied the Toom-Cook-3 (TC) optimized interpolation proposed in [16] until the recursive operand-splitting computation is no longer profitable with respect to the Karatsuba multiplication technique. The Karatsuba multiplication technique is then employed to act on smaller and smaller operands until optimized Karatsuba formulae for all multiplicands up to 9 machine words were implemented according to the strategies described in [70]. We determined the number of machine words where the multiplication strategy is changed (between the TC and Karatsuba strategies) via exhaustive exploration of the tradeoff points.
The execution of the described implementation of the DENSE-BY-DENSE MULTIPLICATION algorithm is guaranteed to not leak any information from the timing side-channel, as the input polynomials are always considered as having p binary coefficients, the sequence of performed operations does not present any early exit condition, and the carryless multiplication instructions between operands having machine word size, w, are also executed in a number of machine cycles that does not depend on the values encoded in each of them.

The constant-time implementation of the SPARSE-BY-DENSE MULTIPLICATION is also provided to allow the secure and efficient running of the LEDAcrypt primitives on computational platforms having an ISA without the *carryless multiplication instruction*. In particular, the said algorithm takes as input two binary polynomials $f(x), g(x) \in \mathbb{Z}_2[x]/(x^p - 1)$, with the former represented, as said above, as a set of asserted-bit positions, i.e., $f(x) = \{b_0, b_1, \dots, b_{s-1}\}$, where $s \geq 1$,

Algorithm 18: GF2X_MODULAR_MONOMIAL_MUL_CT

Input: b_i: exponent value of a monomial $x^{b_i} \in \mathbb{Z}_2[x]$, with $b_i \in \{0, 1, \dots, p-1\}$;
$g(x)$: dense p-bit representation of an invertible element in $\mathbb{Z}_2[x]/(x^p - 1)$

Output: $c(x) = \left(x^{b_i} \cdot g(x)\right) \bmod (x^p - 1)$

Data: $m = \lceil \log_2(p-1) \rceil$: number of bits needed to encode an exponent value b_i in binary, i.e., $b_i = (\beta_{m-1}, \beta_{m-2}, \dots, \beta_1, \beta_0)_{\texttt{bin}}$.
The algorithm always executes the same sequence of operations regardless of the value of b_i, with a computational cost: $\mathcal{O}(p \cdot (\log_2(p) - \log_2(w) + 1))$ bit operations

1 **begin**
2 $\quad c(x) \leftarrow g(x) \cap_p (b_i \bmod w)$ // (intra-word) circular left shift of $j<w$ bits (constant-time)
3 $\quad$ **for** $j \leftarrow \log_2(w)$ **to** $m-1$ **do**
4 $\quad\quad \texttt{tmp}(x) \leftarrow g(x) \cap_p j$ // (inter-word) circular left shift of a public amount j
5 $\quad\quad \texttt{cond} \leftarrow (\beta_j == 1)$ // cond equals 1 in case the equality test is true, 0 otherwise
6 $\quad\quad$ IF-CONSTANT_TIME($c(x)$, cond, tmp(x), $c(x)$) // if (cond ==true) then $c(x) \leftarrow \texttt{tmp}(x)$;
7 $\quad$ **return** $c(x)$
8 **end**
9 IF-CONSTANT_TIME($Z(x)$, BoolVal, $U(x)$, $V(x)$)
$\quad$ // output $Z(x) = z_{l-1}(x^w)^{l-1} + \dots + z_1(x^w)^1 + z_0$, with z_k a w-bit machine word, $l = \lceil \log_w(p) \rceil$
$\quad$ // input $U(x) = u_{l-1}(x^w)^{l-1} + \dots + u_1(x^w)^1 + u_0$, with u_k a w-bit machine word, $l = \lceil \log_w(p) \rceil$
$\quad$ // input $V(x) = v_{l-1}(x^w)^{l-1} + \dots + v_1(x^w)^1 + v_0$, with v_k a w-bit machine word, $l = \lceil \log_w(p) \rceil$
10 $\quad$ mask $\leftarrow 0 - (w\text{-bituint})(\text{BoolVal})$ // mask is an all 0s or all 1s w-bit value
11 $\quad$ **for** $j \leftarrow 0$ **to** $l-1$ **do**
12 $\quad\quad z_j \leftarrow z_j \oplus (u_j$ AND mask) OR (v_j AND mask) // AND, OR, $\oplus$ operations are computed bitwise
13 $\quad$ **return** Z(x)

$b_i \in \{0, 1, \dots, p-1\}$, $0 \le i < s$, and the latter encoded with p bits. From a high-level perspective, the algorithm computes the result $d(x) \equiv f(x) \cdot g(x) \in \mathbb{Z}_2[x]/(x^p - 1)$ by initializing it to the null polynomial and then executing s iterations of a loop where in each iteration $d(x)$ is updated as: $d(x) \leftarrow \left(d(x) + x^{b_i} \cdot g(x)\right) \bmod (x^p - 1),\ 0 \le i < s$. We note that although the number of loop iterations coincides with the number of asserted bits in $f(x)$, this is not a secret information for the use of the SPARSE-BY-DENSE MULTIPLICATION algorithm in LEDAcrypt. Indeed, the value s will be (approximately) equal to either the square root of a fixed fraction of the row/column weight v of the secret $p \times pn_0$ parity-check matrix (i.e., $s = \frac{v}{n_0}$, $v \approx \sqrt{p \cdot n_0} \Rightarrow s \approx \sqrt{p/n_0}$), which is a publicly known parameter, or the number t of asserted bits in the error vector, which is also a publicly known parameter of the cryptoscheme.
Concerning the operations performed in the body of the loop executed by the SPARSE-BY-DENSE MULTIPLICATION algorithm, the modular multiplication of a monomial by a dense p-bit polynomial, i.e., $c(x) \leftarrow x^{b_i} \cdot g(x) \bmod (x^p - 1)$, must be computed by a carefully designed subroutine to guarantee that no information on the exponent of the monomial b_i leaks from the timing side-channel. The subsequent modular addition with the partial value of $d(x)$ is implemented trivially in constant-time performing a bitwise `xor` between the operands.
Algorithm 18 shows the pseudo-code of the constant-time GF2X_MODULAR_MONOMIAL_MUL_CT algorithm introduced for the first time in [18]. The exponent value of the input monomial (i.e., the position of an asserted bit in $f(x)$) is considered in binary encoding, i.e., $b_i = (\beta_{m-1}, \beta_{m-2}, \dots, \beta_1, \beta_0)_{\texttt{bin}}$,

Table 5.1: Computational complexities of the multiplication algorithms

Algorithm	Implementation	Computational Complexity (bit ops.)
Dense-By-Dense Mul.	constant-time	$p^{1+\epsilon}$, with $0 < \epsilon < 1$
Sparse-By-Dense Mul.	constant-time	$s{\cdot}(8{\cdot}p{\cdot}(m{-}\log_2(w)){+}3{\cdot}p)$, with $s{\approx}\sqrt{p/n_0}$, $n_0{\in}\{2,3,4\}$, $m{=}\lceil\log_2(p-1)\rceil$
	variable-time	$s{\cdot}(4{\cdot}p)$, with $s{\approx}\sqrt{p/n_0}$, $n_0{\in}\{2,3,4\}$

with $m{=}\lceil\log_2(p{-}1)\rceil$, to the end of replacing the computation $c(x) \leftarrow \left(x^{b_i} \cdot g(x)\right) \bmod (x^p - 1)$ as

$$c(x) \leftarrow \bigoplus_{j=0}^{m-1} \left((x^{2^j} \cdot g(x) \bmod (x^p - 1)) \cdot (\beta_j == 1) \right).$$

The computation of m circular bit shifts (i.e., $\left(x^{2^j} \cdot g(x)\right) \bmod (x^p - 1)$ – equivalently denoted as $g(x) \cap_p j$ in Algorithm 18, lines 4) guarantees no information leakage from the timing side-channel provided that the modular addition of the result of each of the said circular shifts to the partial value of the returned polynomial $c(x)$ is also performed in a constant-time fashion as shown in Algorithm 18, lines 5-6 and lines 9–12. It is worth noting that $\log_2(w)$ iterations of the loop at lines 3-6 in Algorithm 18, are effectively saved in our implementation by performing the intra-word circular-bit shift (under the assumption that a single word bit shift is performed in constant time), mandated by the $\log_2(w)$ least significant bits of the b_i in $(\beta_{m-1}, \beta_{m-2}, \ldots, \beta_1, \beta_0)_{\mathtt{bin}}$, with a single scan of the input operand $g(x)$ before the beginning of the loop iterations, which in turn can be started from the $\log_2(w)$-th bit of the binary encoding of b_i.

5.2.1 Selection of Polynomial Multiplication Algorithms

The choice of the most fitting multiplication strategy for each LEDAcrypt cryptosystem, i.e., LEDAcrypt-KEM-CPA, LEDAcrypt-KEM and LEDAcrypt-PKC must keep into account the requirement for constant-time execution mandated for the IND-CCA2 systems LEDAcrypt-KEM and LEDAcrypt-PKC and the performance figures exhibited by the Dense-By-Dense Multiplication and Sparse-By-Dense Multiplication algorithms. Considering the computational complexities of the said algorithms, also shown in Table 5.1, the specific values of the LEDAcrypt parameters, and the fact that different computational platforms feature ISAs with specific opportunities for optimizations, the selection of the most fitting algorithm/implementation calls for an exhaustive design space exploration.

It is worth noting that in the implementation of any LEDAcrypt instance one of the two factors of every multiplication is always managed with a sparse representation for global efficiency reasons. Therefore, willing to use a Dense-By-Dense Multiplication strategy to realize the multiplications, an additional constant-time routine, named Densify_Polinomial, must be implemented to convert the sparse representation of a polynomial into a dense one. Specifically, the constant-time implementation of the Densify_Polinomial routine takes as input a set of s integer values, representing the positions of the asserted bits among the coefficients of a p-bit polynomial and computes the i-th coefficient of the output polynomial, for all $0 \leq i < p$, by scanning all the input values and adding to the zero-initialized value of the i-th coefficient, via an inclusive `or`, either a set bit or a zero bit, depending on i being equal to the input value at hand or not. The inclusive `or` operation

is performed by means of a IF-CONSTANT_TIME subroutine following a logic similar to the one applied in Algorithm 18. The computational complexity of the described procedure amounts to $O(s{\cdot}p)$ bit operations and is further reduced exploiting the possibility to update the coefficients of the output polynomial in sequential chunks, each of which including a number of bits/coefficient equal to the machine word size w, obtaining a running time proportional to $O(s{\cdot}p/w)$.

x86_64 Platforms with AVX2 ISA Extension

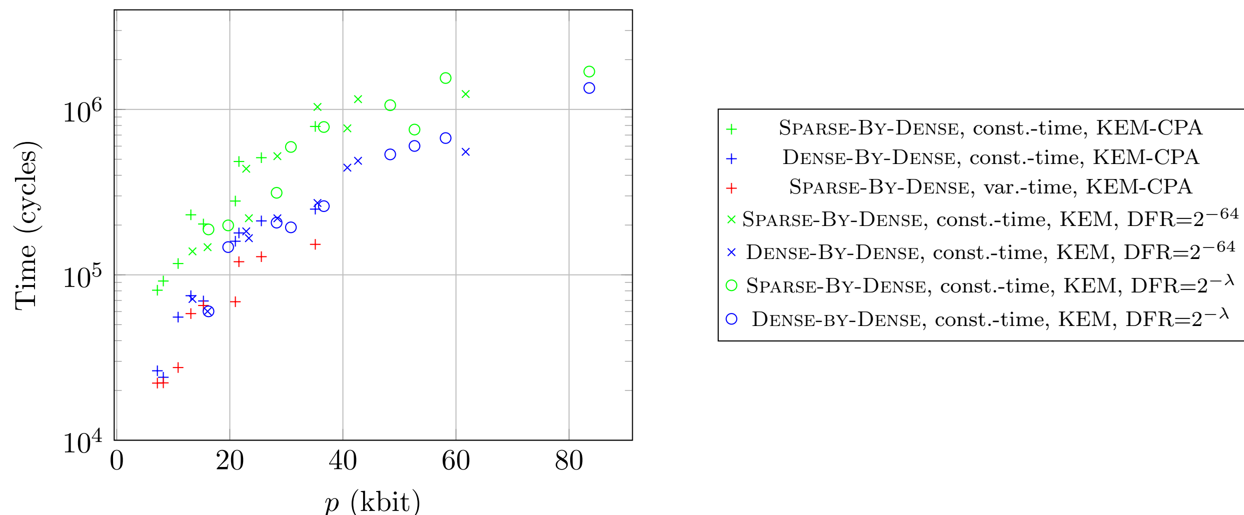


Figure 5.2: Execution times of the polynomial multiplication algorithms plotted against the bit-lengths (i.e., p values) of a circulant block of the QC-MDPC code designed for both the IND-CCA2 LEDAcrypt-KEM instance and the LEDAcrypt-KEM-CPA instance. Plus marks (+) refer to prime sizes employed in LEDAcrypt-KEM-CPA; Cross marks($\times$) refer to prime sizes employed in the IND-CCA2 LEDAcrypt-KEM with DFR=2^{-64}; Circular marks ($\circ$) refer to prime sizes employed in the IND-CCA2 LEDAcrypt-KEM with DFR=$2^{-\lambda}$, with $\lambda \in \{128, 192, 256\}$ being the security-equivalent AES keylength

Figure 5.2 depicts the running time of the described algorithms on our benchmark platform, endowed with the `pclmuldq` carryless multiplication instruction listed in the AVX2 ISA extension and a machine word size $w = 64$. It shows, on the same plot, for each multiplication strategy, the total number of machine cycles needed to execute each of them with polynomial sizes and Hamming weight employed in the LEDAcrypt-KEM-CPA and the IND-CCA2 LEDAcrypt-KEM, when the cryptosystem parameters guarantee either a DFR$= 2^{-64}$ or a DFR$= 2^{-\lambda}$, respectively, with λ being the security-equivalent AES key-length, ranging in $\{128, 192, 256\}$. The timings for the multiplications involved in the IND-CCA2 LEDAcrypt-PKC match the ones of the IND-CCA2 LEDAcrypt-KEM they share the values of the p.
In Figure 5.2 the total numbers of machine cycles are shown as a function of the bit-length of the polynomials corresponding to the bit-length of the circulant-block side of the QC-MDPC parity check matrix adopted in each LEDAcrypt instance. Finally, the timings of the DENSE-BY-DENSE MULTIPLICATION in the figure includes also the machine cycles required by the DENSIFY_POLINOMIAL routine to convert the sparse factor into a dense one.

Counter-intuitively, Figure 5.2 makes clear that the best choice for the selection of the most fitting multiplication strategy, in terms of execution times, for all LEDAcrypt requiring constant time execution and for all possible rates of the underlying QC-MDPC code (i.e., $n_0 \in \{2, 3, 4\}$), is to pick the proposed Dense-By-Dense Multiplication algorithm even if it mandates the execution of the Densify_Polinomial routine before every operation. Contrariwise, for all scenarios where LEDAcrypt-KEM-CPA is involved, and therefore no constant time execution requirements are needed, the variable time Sparse-By-Dense Multiplication is the most convenient approach.

Algorithm 19: OPTIMIZED OUTOFPLACEITERATION

Input: s: QC-LDPC syndrome, binary vector of size p
H: parity-check matrix, represented as an $n_0 \times v$ integer matrix containing in each row the positions (ranging in $\{0, 1, \dots, p-1\}$) of the set coefficients in the first row of each n_0 $p \times p$ block of the circulant-block matrix $H = [H_0 \mid H_1 \mid \dots \mid H_{n_0-1}]$.
H^T: transposed parity-check matrix, represented also as n_0 binary polynomials, each corresponding to the first row of H_i^T, $0 \leq i < n_0$.
$\hat{e} = (\hat{e}_0, \hat{e}, \dots, \hat{e}_{n_0-1})$: $(n_0 p)$-bit initial error vector estimate ($\hat{e}_i$, $0 \leq i < n_0$: p-bit polyn.)
$\mathtt{iter}^{\mathtt{Out}}$: number of iterations computed before

Output: $\bar{e} = (\bar{e}_0, \bar{e}_1, \dots, \bar{e}_{n_0-1})$: $(n_0 p)$-bit updated error vector estimate ($\bar{e}_i$, $0 \leq i < n_0$: p-bit polyn.)
$\bar{s}$: p-bit updated syndrome

Data: $\mathtt{bs_upc}_i$, $0 \leq i < n_0$, $(\lceil \log_2(v) \rceil + 1) \times p$ bit-matrix derived from a vector of p integers, each of which encoded in two's-complement over $(\lceil \log_2(v) \rceil + 1)$ bits, with the least significant bits (lsb) stored in row 0, and the most significant bits (msb) stored in row $\lceil \log_2(v) \rceil$

```
1  th ← ComputeThreshold(iter^Out, s)
2  for i = 0 to n_0 − 1 do
3      bs_upc_i ← Replicate_and_bitslice(−th)
       /* bs_upc_i initialized as a (⌈log_2(v)⌉ + 1) × p bit-matrix having all columns equal
          to the 2's complement encoding of the integer −th; msb stored in row 0.      */
4      for k = 0 to v − 1 do
5          s′ ← gf2x_modular_monomial_mul_CT(H[i][k], s)
6          bs_upc_i ← BitSliced_increment(bs_upc_i, s′)
           /* s′ is processed as a (⌈log_2(v)⌉ + 1) × p) zero bit-matrix except for the
              first row that actually stores each syndrome bit as the msb in a column;
              bs_upc_i is updated performing an addition between each pair of corresponding
              columns in bs_upc_i and the s′ bit-matrix; the addition is vectorized to act
              on the bits stored in an entire row of bs_upc_i at once                   */
7  for i = 0 to n_0 − 1 do
8      ẽ_i ← BitWise_NOT(Extract_Row_with_Msb(bs_upc_i))
       /* if the j-th msb is set (0≤j<p), then the unsatisfied parity-check counting
          for e_i[j] was greater or equal to th                                        */
9      ē_i ← ẽ_i ⊕ ê_i  // update of the input p-bit ê_i to get the output ē_i
10     s̄_i ← s_i ⊕ Dense-By-Dense_Multiplication(ẽ_i, H_i^T)  // s̄_i=s_i⊕ẽ_i·H_i^T
11 return {ē, s̄}
```

5.3 Optimized Out-Of-Place Iteration Function

The optimized implementation of the OUTOFPLACEITERATION in Algorithm 19 has been designed applying the *bit-slicing* parallelization strategy introduced in [18] to fully exploit the *vector* instructions featured by the ISA of the underlying computational platform. The semantics of the computed values are equivalent to the ones of the non optimized algorithm presented in Section 3.3 as Algorithm 14 and follows the same high-level decoding process. Algorithm 19 takes as input the received p-bit syndrome and the secret parity-check matrix consisting in a sequence of n_0 non-singular circulant blocks $H = [H_0 \mid H_1 \mid \dots \mid H_{n_0-1}]$, each defined as a $p \times p$ circulant bit-matrix with fixed row/column weight v, and represented as the set of asserted bit positions in the first row

of each of them; the polynomial representation of each block of the transposed parity-check matrix H^T is also assumed to be available for efficiency reasons. The decoding process strategy starts from an $(n_0 p)$-bit error vector estimate $\hat{e} = (\hat{e}_0, \hat{e}_1, \ldots, \hat{e}_{n_0-1})$, where $\hat{e}_i$, $0 \leq i < n_0$, is a p-bit polynomial, and computes the number of parity equations ($\hat{e}H^T = H\hat{e}^T = s$) with both an asserted constant value (i.e., $s_k = 1$, $0 \leq k < p$) and a term in correspondence with the l-th bit of the error vector, for all bits $0 \leq l < pn_0$. Subsequently, each of the said numbers of unsatisfied parity-check equations, upcs for short, is compared with a pre-determined threshold th $\in \{\lceil \frac{v}{2} \rceil, \ldots, v\}$ to determine an auxiliary $(n_0 p)$-bit error vector $\tilde{e}$ having an asserted bit when the upc value is greater than or equal than th and a clear bit otherwise. The auxiliary error vector $\tilde{e}$ is in turn employed to update the input error vector estimate $\hat{e}$ to get the output error vector estimate $\bar{e}$ as $\bar{e} = \tilde{e} \oplus \hat{e}$. Finally, the value of the syndrome is also updated to take into account the updates on the estimated error vector, and returned as an output $\bar{s}$, by computing $\bar{s} = \bar{s} \oplus \tilde{e}H^T$ (or equivalently, as shown in Algorithm 14, by adding to the input syndrome all columns of H corresponding to upc values greater or equal to th).

The optimized out-of-place iteration function of the LEDAdecoder shown in Algorithm 19 replaces the aforementioned array of $n_0 p$ upc counters (employed to compute the number of unsatisfied parity-check equations including as a term the l-th bit, $0 \leq l < n_0 p$, of the error vector and an asserted bit syndrome as constant term) with a sequence of n_0 bit-matrices bs_upc$_i$, $0 \leq i < n_0$, with p-columns, each of which is thought to be in one-to-one correspondence with the analogous bit position in the i-th p-bit portion of the initial error vector estimate $\hat{e}_i$. The number of rows in the bit-matrix bs_upc$_i$ equals the number of binary digits needed to encode in two's-complement the signed integer value of each upc counter, i.e.: $\lceil \log_2(v) \rceil + 1$. The actual implementation of Algorithm 19 assumes, conventionally, the first row of each bs_upc$_i$, $0 \leq i < n_0$, (i.e., bs_upc$_i$[0][:]) to store the least significant bits (lsb) of the upc values.

The computation of the upc values is performed by the steps shown in Algorithm 19, lines 2–6, considering them as a sequence of n_0 groups of p consecutive upcs, where the i-th group is represented by the p columns of the bit-matrix bs_upc$_i$. For each bs_upc$_i$ (line 2), the first step of the algorithm (line 3) initializes each of its columns with the two's-complement encoding of $-$th, i.e., the opposite of the integer value kept as threshold to establish if a given upc counter should determine a bit-flip in the corresponding position of the estimated error vector or not. The idea is to establish if the bs_upc$_i$ column (counter) is greater than or equal to 0 by looking at the elements in the last row of a bit-matrix and to proceed by updating the estimate error vector and the syndrome value (lines 7–10) accordingly.

The actual computation of the upc values stored in the columns of bs_upc$_i$ proceeds observing that for each position a of an asserted coefficient of the i-th block, H_i, of the parity-check matrix ($a =$ H[i][k], $0 \leq k < v$), the constant term of the parity-check equation, including the said coefficient as a term, can be obtained rotating the p-bit syndrome polynomial as $s' \leftarrow x^a s \bmod (x^p - 1)$. This monomial by polynomial multiplication in $\mathbb{Z}_2[x]/(x^p - 1)$ must be performed in constant-time to prevent the leakage of information related to the secret parity-check matrix from the timing side-channel. To this end, the GF2X_MODULAR_MONOMIAL_MUL_CT routine presented in Section 5.2 is employed (see Algorithm 19, line 5). As a consequence of the computation of the rotated syndrome s', the update of the upc values stored in the j-th column, $0 \leq j < p$, of bs_upc$_i$ can proceed by incrementing it by either one or zero, depending on the value of the j-th bit in s'. To perform such an increment, s' is processed as a $(\lceil \log 2(v) \rceil + 1) \times p)$ zero bit-matrix except for the first row that stores each bit of s' as the lsb of the zero or one two's complement value represented in a column. The update of the j-th column in bs_upc$_i$ is performed by adding to it the j-th column of

the bit-matrix representing s'. The implementation of the BitSliced_increment routine shown in Algorithm 19, line 6, considers the Boolean formulae to realize the increment operation and parallelizes its computation over multiple upc values by applying the same Boolean operation simultaneously on all the bits in the same row of the matrices, exploiting the vector instructions of the computational platform at hand.

Once the upc values has been computed as columns of $\mathtt{bs_upc}_i$, $0 \leq i < n_0$, the final steps of Algorithm 19 (lines 7–10) extract the last row of the i-th bit-matrix, $\mathtt{bs_upc}_i[\lceil \log_2(v) \rceil][:]$ and compute the i-th portion of the estimated error vector corrections $\tilde{e}_i$ as the bitwise NOT of $\mathtt{bs_upc}_i[\lceil \log_2(v) \rceil][:]$. Subsequently, the i-th portion of the input estimated error vector $\hat{e}_i$ is xored with the i-th portion of the error vector corrections $\tilde{e}_i$, to obtain the corresponding portion of the output error vector estimate $\bar{e}_i$ (lines 8-9). Finally, the output syndrome is evaluated by subtracting (via bitwise-xor) from the input syndrome the product between the estimated error vector corrections $\tilde{e}$ and the transposed parity-check matrix (line 10).

A final remark on the memory access patterns during the computation of line 6, suggests that a memory representation of $\mathtt{bs_upc}_i$ providing greater cache-friendliness is a different one from the canonical linearization of the $\mathtt{bs_upc}_i$ matrix by row, as operated by the C language. Indeed, we found out by experimental assessment, that splitting the $\mathtt{bs_upc}_i$ matrix in blocks of columns as wide as the vector instructions offered by the underlying platform (i.e., 256 bits for the AVX2 instructions in Intel Haswell CPUs and successors), and linearizing each block by row, yielded a speedup of around $2\times$. We ascribe this speedup to the fact that when BitSliced_Increment operation is computed, on a portion of 256 columns of the $\mathtt{bs_upc}_i$ matrix, the said portion resides in a single cache line, therefore minimizing the performance impact of the store operations.

5.4 Binary Polynomial Inversion

The KEY-GENERATION algorithm of all LEDAcrypt primitives require $n_0 - 1$ multiplications between $p \times p$ binary blocks, each of which is represented as an invertible polynomial in the ring $\mathbb{Z}_2[x]/(x^p - 1)$ with an odd and small number of asserted coefficients $v \approx \sqrt{p/n_0}$. In particular, the first factor of each multiplication is given by the multiplicative inverse of the polynomial $a(x) \in \mathbb{Z}_2[x]/(x^p - 1)$ corresponding to the first row of the last circulant-block of the secret parity-check matrix $H{=}[H_0, \ldots, H_{n_0-1}]$.

Although the computation of a multiplicative inverse of a polynomial is required only once during the execution of the KEY-GENERATION algorithm for the IND-CCA2 LEDAcrypt-KEM and the IND-CCA2 LEDAcrypt-PKC system, the length/degree of the involved operand $a(x)$, $7 \cdot 10^3 < \mathsf{deg}(a(x)) < 9 \cdot 10^4$, requires to implement carefully this operation by establishing the inversion strategy that fits best the length of the involved operand. Furthermore, the LEDAcrypt-KEM-CPA system generates ephemeral private/public key pairs to transmit a session key at each run, requiring the computation of a multiplicative inverse polynomial each time. Finally, another feature steering the choice of the best polynomial inversion algorithm for a given primitive and operand size is the possibility to exhibit a constant-time implementation to prevent timing based side-channel analyses aimed at the recovering of the secret key $sk = \{H\}$.

5.4.1 Schoolbook Euclid's Algorithm

Euclid's algorithm to compute the greatest common divisor, gcd, between two polynomials is commonly employed to derive a polynomial time algorithm yielding the multiplicative inverse of an element in the multiplicative group of a Galois field represented in polynomial basis, e.g., the multiplicative group of $\mathrm{GF}(2^m), m \geq 2$ with $\mathrm{GF}(2^m) \cong \mathbb{Z}_2[x]/(f(x))$, where $f(x)$ is an irreducible polynomial with $\mathsf{deg}(f(x)) = m$.

A schoolbook analysis considers a polynomial $a(x)$ and the irreducible polynomial $f(x)$, employed to represent the field elements $(\mathsf{deg}(a(x)) < \mathsf{deg}(f(x)))$, to iteratively apply the equality $\mathsf{gcd}(f(x), a(x)){=}\ \mathsf{gcd}(a(x), f(x) \bmod a(x))$ and derive the computation path leading to the non-null constant term representing the greatest common divisor d (e.g., $d = 1$, when $\mathbb{Z}_2[x]$ is considered):

$$
\begin{array}{ll}
\mathsf{deg}(f(x)) > \mathsf{deg}(a(x)) > 0 & d(x){=}\mathsf{gcd}(f(x), a(x)) \\
r_0(x){=}f(x),\ r_1(x){=}a(x) & d(x){=}\mathsf{gcd}(r_0(x), r_1(x)) \\
r_2(x){=}r_0(x) \bmod r_1(x), \quad 0{\leq}\mathsf{deg}(r_2(x)){<}\mathsf{deg}(r_1(x)) & d(x){=}\mathsf{gcd}(r_1(x), r_2(x)) \\
\ldots & \ldots \\
r_i(x){=}r_{i-2}(x) \bmod r_{i-1}(x), \quad 0{\leq}\mathsf{deg}(r_i(x)){<}\mathsf{deg}(r_{i-1}(x)) & d(x) = \mathsf{gcd}(r_{i-1}(x), r_i(x)) \\
\ldots & \ldots \\
r_z(x){=}r_{z-2}(x) \bmod r_{z-1}(x), \quad r_z(x){=}0 & d{=}\mathsf{gcd}(r_{z-1}(x), 0){=}r_{z-1}(x)
\end{array}
$$

for a proper number of iterations, $z \geq 2$. A close look to the previous derivation shows that two series of auxiliary polynomials $w_i(x)$ and $u_i(x)$, $0 \leq i \leq z-1$, can be defined to derive the series of remainders $r_i(x)$ as: $r_i(x) = r_{i-2}(x) - q_i(x) \cdot r_{i-1}(x) = w_i(x) \cdot f(x) + u_i(x) \cdot a(x)$, where $q_i(x) = \left\lfloor \frac{r_{i-2}(x)}{r_{i-1}(x)} \right\rfloor$, with $r_0(x) = f(x)$ and $r_1(x) = a(x)$. Specifically, the computations shown above can be rewritten as follows:

Algorithm 20: Inversion using Euclid's gcd Algorithm

Input: $f(x)$ irreducible polynomial of GF(2^m), m 2,
$a(x)$ invertible element of GF(2^m)

Output: $\mathsf{V}(x)$, polynomial such that $a(x)^{-1}$ $\mathsf{V}(x) \bmod f(x)$

1 **begin**
2 $\quad \mathsf{S}(x)$ $f(x)$, $\mathsf{R}(x)$ $a(x)$
3 $\quad \mathsf{V}(x)$ 0, $\mathsf{U}(x)$ 1
4 $\quad$ **repeat**
5 $\qquad \mathsf{Q}(x)$ $b\mathsf{S}(x)/\mathsf{R}(x)c$
6 $\qquad \mathsf{tmp}(x)$ $\mathsf{S}(x)$ $\mathsf{Q}(x)$ $\mathsf{R}(x)$, $\mathsf{S}(x)$ $\mathsf{R}(x)$, $\mathsf{R}(x)$ $\mathsf{tmp}(x)$
7 $\qquad \mathsf{tmp}(x)$ $\mathsf{V}(x)$ $\mathsf{Q}(x)$ $\mathsf{U}(x)$, $\mathsf{V}(x)$ $\mathsf{U}(x)$, $\mathsf{U}(x)$ $\mathsf{tmp}(x)$
8 $\quad$ **until** ($\mathsf{R}(x) = 0$)
9 $\quad$ **return** $\mathsf{V}(x)$

$\mathsf{deg}(f(x)) > \mathsf{deg}(a(x)) > 0$
$r_0(x) = 1 \cdot f(x) + 0 \cdot a(x) = w_0(x) \cdot f(x) + u_0(x) \cdot a(x)$
$r_1(x) = 0 \cdot f(x) + 1 \cdot a(x) = w_1(x) \cdot f(x) + u_1(x) \cdot a(x)$
$r_2(x) = r_0(x) \bmod r_1(x) =$
$\quad = (w_0(x) \cdot f(x) + u_0(x) \cdot a(x)) - \left\lfloor \frac{r_0(x)}{r_1(x)} \right\rfloor (w_1(x) \cdot f(x) + u_1(x) \cdot a(x)) =$
$\quad = w_2(x) \cdot f(x) + u_2(x) \cdot a(x)$
$\ldots$
$r_i(x) = r_{i-2}(x) \bmod r_{i-1}(x) = w_i(x) \cdot f(x) + u_i(x) \cdot a(x)$
$\ldots$
$r_{z-1}(x) = r_{z-3}(x) \bmod r_{z-2}(x) = w_{z-1}(x) \cdot f(x) + u_{z-1}(x) \cdot a(x)$
$r_z = r_{z-2}(x) \bmod r_{z-1}(x) = 0$

Finally, the multiplicative inverse of $a(x)$ is obtained from the equality $d = w_{z-1}(x) \cdot f(x) + u_{z-1}(x) \cdot a(x)$, by computing:

$$a(x)^{-1} \equiv \left(d^{-1} \cdot u_{z-1}(x) \bmod f(x)\right).$$

In the last derivation of remainder polynomials, it is worth noting that $w_i(x) = w_{i-2}(x) - q_i(x) \cdot w_{i-1}(x)$, and $u_i(x) = u_{i-2}(x) - q_i(x) \cdot u_{i-1}(x)$, with $w_0(x) = 1$, $u_0(x) = 0$ and $w_1(x) = 0$, $u_1(x) = 1$.

Restricting ourselves to the case of binary polynomials, Algorithm 20 shows the pseudo-code for computing an inverse employing only the strictly needed operations. Note that the execution of an actual division would prevent the efficient implementation of the algorithm, therefore in the following section we consider the optimizations to this algorithm that circumvent its computation. The naming conventions adopted in Algorithm 20 denote as $\mathsf{R}(x)$ the i-th remainder of the gcd procedure described before, and as $\mathsf{S}(x)$ the $(i-1)$-th remainder, with $i \geq 1$. Furthermore, the i-th $u(x)$ value is denoted as $\mathsf{U}(x)$, while the value it took at the previous iteration (i.e., the $(i-1)$-th $u(x)$ value) is denoted as $\mathsf{V}(x)$, with $i \geq 1$.
Note that in the last iteration (the z-th one, counting the first as 1) $\mathsf{R}(x) = r_z(x) = 0$, while $\mathsf{S}(x) = r_{z-1}(x) = 1$.
The rationale to keep the notation of the pseudo-code variables as polynomials in x lies on the willingness of not specifying the implementation dependent choice for the endianness of values stored in array variables, i.e., if the least significant coefficient of a polynomial is stored in the first or last cell of an array.

In the case of LEDAcrypt where the arithmetic operations are performed in $\mathbb{Z}_2[x]/(x^p - 1)$, with

Algorithm 21: Brunner *et al.* (BCH) Inversion Algorithm [17]

Input: $f(x)$ irreducible polynomial of $\mathrm{GF}(2^m)$, $m \quad 2$,
$a(x)$ invertible element of $\mathrm{GF}(2^m)$

Output: $\mathsf{V}(x)$, such that $a(x)^{-1} \quad \mathsf{V}(x)$ 2 $\mathrm{GF}(2^m)$

Data: polynomial variables $\mathsf{R}(x), \mathsf{S}(x), \mathsf{U}(x), \mathsf{V}(x)$ are assumed to have at most $m{+}1$ coefficients each (e.g., $\mathsf{R}(x){=}\mathsf{R}_m\ x^m{+}\mathsf{R}_{m-1}\ x^{m-1}{+} \quad {+}\mathsf{R}_1\ x{+}\mathsf{R}_0$, with R_i 2 $\mathbb{Z}_2, 0 \quad i \quad m$) to prevent reduction operations (i.e., mod $f(x)$) among the intermediate values of the computation;
δ stores a signed integer number

```
1  begin
2      S(x)   f(x), R(x)   a(x)
3      V(x)   0, U(x)   1, δ   0
4      for i   1 to 2  m do
5          if (R_m = 0) then
6              R(x)   x  R(x)
7              U(x)   x  U(x)
8              δ   δ + 1
9          else
10             if (S_m = 1) then
11                 S(x)   S(x)   R(x)
12                 V(x)   V(x)   U(x)
13             else
14                 S(x)   x  S(x)
15                 if (δ = 0) then
16                     tmp(x)   R(x), R(x)   S(x), S(x)   tmp(x)
17                     tmp(x)   U(x), U(x)   V(x), V(x)   tmp(x)
18                     U(x)   x  U(x)
19                     δ   1
20                 else
21                     U(x)   U(x)/x
22                     δ   δ   1
23     return V(x)
```

$\mathsf{ord}_2(p) = p-1$, it is worth noting that Euclid's algorithm can still be applied to compute the inverse of unitary elements as the factorization of $f(x) = x^p - 1$ in irreducible factors, i.e., $f(x) = (x+1) \cdot (\sum_{i=0}^{p-1} x^i)$, shows that every binary polynomial $a(x)$ with degree less than p and an odd number of asserted coefficients, that is, also different from the said factors, (i.e., any polynomial representing a circular block in the LEDAcrypt) always admits a greater common divisor equal to one (i.e., $\mathsf{gcd}(a(x), f(x)) = 1$). As a consequence, the steps of Euclid's algorithm to compute inverses remain the same shown before.

5.4.2 Optimized Polynomial Inversions

In [17] Brunner *et al.* optimized the computation performed by Algorithm 20 embedding into it the evaluation of the quotient and remainder of the polynomial division $\lfloor \mathsf{S}(x)/\mathsf{R}(x) \rfloor$. Indeed, Algorithm 21 performs the division operation by repeated shifts and subtractions, while keeping track of the difference δ between the degrees of polynomials in $\mathsf{S}(x)$ (which is the dividend and starts by containing the highest degree polynomial, that is, the modulus $f(x)$) and $\mathsf{R}(x)$ (which is the divisor, and is initialized to the polynomial whose inverse should be computed, $a(x)$). The difference between the degrees, $\delta = \mathsf{deg}(\mathsf{S}) - \mathsf{deg}(\mathsf{R})$ is updated each time the degree of either $\mathsf{S}(x)$ or $\mathsf{R}(x)$ is altered, via shifting.

Brunner *et al.* in [17] observed that, since either a single bit-shift or a subtraction is performed at each iteration, the number of iterations equals $2m$, that is the number m of single bit-shifts needed to consider every bit of the element to be inverted, plus the number of substractions to be performed after each alignment. The last iteration of Algorithm 21 computes $\mathsf{R}(x) = 0$, $\mathsf{S}(x) = 1$ (i.e., $\mathsf{deg}(\mathsf{R}) = \mathsf{deg}(\mathsf{S}) = 0$), and outputs the result in $\mathsf{V}(x)$. The part of Algorithm 21 dealing with computations on $\mathsf{U}(x)$ and $\mathsf{V}(x)$ mimics the same operations, in the same order, that are applied to $\mathsf{R}(x)$ and $\mathsf{S}(x)$, respectively, as the schoolbook algorithm does (Alg. 20).

Kobayashi-Takagi-Takagi Algorithm

The target implementation of Algorithm 21 is a dedicated hardware one, therefore the algorithm simplifies the computations to be performed in it, reducing them to single bit-shifts, bitwise xors and single bit comparison. While this approach effectively shortens the combinatorial cones of a hardware circuit, yielding advantages in timing, when directly transposed in a software implementation it may fail to exploit the wide computation units that are present in a CPU to the utmost. In particular, modern desktops and high-end mobile CPUs are endowed with a so-called carryless multiplier computation unit. Such a unit computes the product of two binary polynomials with degree lower than the architecture word size, w, with a latency which is definitely smaller than computing the same operation with repeated shifts and xors (3–7 cycles on Intel CPUs [20]). To maximize the exploitation of the available carryless multipliers, Kobayashi *et al.* [46] optimized further the gcd based Algorithm 21. The main assumption in this sense is that a w-bit input carryless multiplier is available on the target CPU architecture. Algorithm 22 reports the result of the optimization proposed in [46], which still takes as input an irreducible polynomial of $\mathrm{GF}(2^m)$, employed to build a representation for all its elements, and one of its invertible elements. However, these polynomials and the ones computed as intermediate values of the algorithm, are now thought as polynomials of degree at most $M-1$, with $M = \lceil (m+1)/w \rceil$, having as coefficients binary polynomials with degree $w-1$. For example, the polynomial $S(x) = s_m \cdot x^m + s_{m-1} \cdot x^{m-1} + \cdots + s_1 \cdot x + s_0$, with $s_j \in \{0, 1\}$ and $0 \le j \le m$, is processed by considering it as $S(x) = S_{M-1}(x) \cdot (x^w)^{M-1} + \cdots + S_1(x) \cdot (x^w)^1 + S_0(x)$, $S_i(x) = s_{iw+w-1} \cdot x^{w-1} + \cdots + s_{iw+1} \cdot x + s_{iw}$, where $s_{iw+l} = 0$ if $iw + l > m$, with $0 \le l \le w-1$, and $0 \le i \le M-1$.

Employing this approach, the loop at lines 5–22 of Algorithm 21 is rewritten to perform fewer iterations, each one of them corresponding to the computation made in w iterations of Algorithm 21. The crucial observation is that the computations performed by Algorithm 21 on $\mathsf{U}(x), \mathsf{V}(x), \mathsf{R}(x)$,and $\mathsf{S}(x)$ at each iteration can be represented as a linear transformation of the vectors $\begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix}$ and $\begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix}$. Indeed, the authors represent the computation portions of Algorithm 21, driven by the values of $\mathsf{S}(x)$ and $\mathsf{R}(x)$, present at lines 6–7 as $\begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix} = \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \begin{pmatrix}\mathsf{U}(x)\\ \mathsf{V}(x)\end{pmatrix}$, $\begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix} = \begin{pmatrix}x & 0\\ 0 & 1\end{pmatrix} \times \begin{pmatrix}\mathsf{R}(x)\\ \mathsf{S}(x)\end{pmatrix}$, and rewrite similarly also lines 11–12, line 14, lines 16–18 and line 21.

With this representation of the computation, Algorithm 22 selects a portion of $\mathsf{R}(x)$ and $\mathsf{S}(x)$ as large as an architectural word (line 5 of Algorithm 22) and makes a copy of them (variables $\mathsf{C}(x)$ and $\mathsf{D}(x)$). Employing these copies, Algorithm 22 computes a single linear transformation, H, cumulating the effects of w computations of the main loop of Algorithm 21, to apply them all at once (line 28, Algorithm 22). While this approach appears more expensive, as polynomial multiplications are employed to compute the values of $\mathsf{U}(x), \mathsf{V}(x), \mathsf{R}(x)$ and $\mathsf{S}(x)$, at the end of each iteration of the loop at lines 5–33, we recall that such multiplications can be carried out at the cost

Algorithm 22: Kobayashi *et al.* (KTT) Inversion Algorithm [46]

Input: $f(x)$ irreducible polynomial of $\mathrm{GF}(2^m)$, $m \quad 2$,
$a(x)$ invertible element of $\mathrm{GF}(2^m)$

Output: $\mathsf{V}(x)$, such that $a(x)^{-1} \quad \mathsf{V}(x) \; 2 \; \mathrm{GF}(2^m)$

Data: Polynomials $\mathsf{R}(x), \mathsf{S}(x), \mathsf{U}(x), \mathsf{V}(x)$ have at most $m+1$ binary coefficients to prevent the execution of modulo operations in the intermediate computations. Furthermore, they are assumed to be processed a w-bit chunk at a time, i.e., $M = d(m+1)/we$ and
$\mathsf{R}(x) = \mathsf{R}_{M-1}(x) \; (x^w)^{M-1} + \quad + \mathsf{R}_1(x) \; (x^w)^1 + \mathsf{R}_0(x)$, with $0 \quad \deg(\mathsf{R}_i(x)) < w, 0 \quad i \quad M \quad 1$;
Polynomials $\mathsf{C}(x)$ and $\mathsf{D}(x)$ are such that $0 \quad \deg(\mathsf{C}), \deg(\mathsf{D}) < w$.

1 **begin**
2 $\quad \mathsf{S}(x) \quad f(x)$, $\mathsf{R}(x) \quad a(x)$
3 $\quad \mathsf{V}(x) \quad 0$, $\mathsf{U}(x) \quad x$, $\mathsf{degR} \quad M \; w \quad 1$, $\mathsf{degS} \quad M \; w \quad 1$
4 $\quad$ **while** $\mathsf{degR} > 0$ **do**
5 $\quad\quad \mathsf{C}(x) \quad \mathsf{R}_{M-1}(x)$, $\mathsf{D}(x) \quad \mathsf{S}_{M-1}(x)$
6 $\quad\quad$ **if** $\mathsf{C}(x) = 0$ **then**
7 $\quad\quad\quad \mathsf{R}(x) \quad x^w \quad \mathsf{R}(x)$, $\mathsf{U}(x) \quad x^w \quad \mathsf{U}(x)$
8 $\quad\quad\quad \mathsf{degR} \quad \mathsf{degS} \quad w$
9 $\quad\quad\quad$ **goto** line **4**
10 $\quad\quad \mathsf{H} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $j \quad 1$
11 $\quad\quad$ **while** $j < w$ **and** $\mathsf{degR} > 0$ **do**
12 $\quad\quad\quad j \quad j+1$
13 $\quad\quad\quad$ **if** $\mathsf{C}_{w-1} = 0$ **then**
14 $\quad\quad\quad\quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix} \quad \begin{pmatrix} x & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix}$; $\mathsf{H} \quad \begin{pmatrix} x & 0 \\ 0 & 1 \end{pmatrix} \quad \mathsf{H}$
15 $\quad\quad\quad\quad \mathsf{degR} \quad \mathsf{degR} \quad 1$
16 $\quad\quad\quad$ **else if** $(\mathsf{degR} = \mathsf{degS})$ **then**
17 $\quad\quad\quad\quad \mathsf{degR} \quad \mathsf{degR} \quad 1$
18 $\quad\quad\quad\quad$ **if** $\mathsf{D}_{w-1} = 1$ **then**
19 $\quad\quad\quad\quad\quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix} \quad \begin{pmatrix} x & x \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix}$; $\mathsf{H} \quad \begin{pmatrix} x & x \\ 1 & 0 \end{pmatrix} \quad \mathsf{H}$
20 $\quad\quad\quad\quad$ **else**
21 $\quad\quad\quad\quad\quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix} \quad \begin{pmatrix} 0 & x \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix}$; $\mathsf{H} \quad \begin{pmatrix} 0 & x \\ 1 & 0 \end{pmatrix} \quad \mathsf{H}$
22 $\quad\quad\quad$ **else**
23 $\quad\quad\quad\quad \mathsf{degS} \quad \mathsf{degS} \quad 1$
24 $\quad\quad\quad\quad$ **if** $\mathsf{D}_{w-1} = 1$ **then**
25 $\quad\quad\quad\quad\quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ x & x \end{pmatrix} \quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix}$; $\mathsf{H} \quad \begin{pmatrix} 1 & 0 \\ x & x \end{pmatrix} \quad \mathsf{H}$
26 $\quad\quad\quad\quad$ **else**
27 $\quad\quad\quad\quad\quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & x \end{pmatrix} \quad \begin{pmatrix} \mathsf{C}(x) \\ \mathsf{D}(x) \end{pmatrix}$; $\mathsf{H} \quad \begin{pmatrix} 1 & 0 \\ 0 & x \end{pmatrix} \quad \mathsf{H}$
28 $\quad\quad \begin{pmatrix} \mathsf{R}(x) \\ \mathsf{S}(x) \end{pmatrix} \quad \mathsf{H} \quad \begin{pmatrix} \mathsf{R}(x) \\ \mathsf{S}(x) \end{pmatrix}$; $\begin{pmatrix} \mathsf{U}(x) \\ \mathsf{V}(x) \end{pmatrix} \quad \mathsf{H} \quad \begin{pmatrix} \mathsf{U}(x) \\ \mathsf{V}(x) \end{pmatrix}$
29 $\quad$ **if** $\deg(\mathsf{R}) = 0$ **then**
30 $\quad\quad$ **return** $\mathsf{U}(x)/x^{wM}$
31 $\quad$ **return** $\mathsf{V}(x)/x^{wM}$

of a handful of xors by the carryless multipliers present on modern CPUs, effectively resulting in a favourable tradeoff for the strategy described by Algorithm 22. To provide a further speedup, Algorithm 22 has a shortcut in case entire word-sized portions of $\mathsf{R}(x)$ (which is initialized with the input polynomial to be inverted) are filled with zeroes, a fact which can be efficiently checked in one or a few CPU operations in software. Indeed, in that case, the resulting transformation to be

applied is given by $\mathsf{H} = \begin{pmatrix} x^w & 0 \\ 0 & 1 \end{pmatrix}$; therefore the authors of Algorithm 22 insert a shortcut (lines 6–9) to skip the expensive execution of the loop body computing a trivial transformation, simply applying the aforementioned H, which boils down to two word-sized operand shifts (line 7). We note that this efficiency improvement trick, while effectively making the inversion faster, also provides an information leakage via the timing side channel. Indeed, due to this shortcut, the inversion algorithm will be running faster if it is consuming words of the polynomial to be inverted which contain only zeroes. As a result, some information on the length of the zero runs of the operand is encoded in the timing side channel. We note that, in many code-based cryptoschemes, the position of the asserted coefficients of the polynomial to be inverted represent the private key of the scheme. As a final step, Algorithm 22 returns either the value of $\mathsf{U}(x)$ or the value of $\mathsf{V}(x)$, detecting which is the variable containing the actual pseudo inverse (i.e., $a(x)^{-1}x^{wM}$), and performing an exact division by x^{wM} (i.e., a shift by M words).

Bernstein-Yang Algorithm

The last approach to polynomial inversion via improved Euclid algorithms, is the one recently proposed in [14]. Bernstein and Yang provide a comprehensive study [14] of modular inversion and greatest common divisor (`gcd`) computation both for integer and polynomial Euclidean domains. In this document, we specialize the divide-et-impera strategy devised for the computation of `gcd`s and modular inverses for polynomial rings having coefficients over a generic GF(p) onto one which only performs the computation of modular inverses of binary polynomials. We report the said specialized algorithm as Algorithm 23. A full proof of the correctness of the algorithmic approach is provided in [14], and we will omit it from the current document for space reasons. However, to provide an intuition of the inner working of Algorithm 23, we note that it is possible to obtain Algorithm 22 as a special case of it, highlighting the conceptual similarities. The key observation made in [14] is that it is possible to split the operands recursively up to a point where the operands become as small as the designer deems useful (a machine word, in our case) and then proceed to compute a portion of the linear transform over GF(2^m) which operates on the inputs to provide the modular inverse. Once such a transform is computed for the operand fragments, the transform can be recombined by means of a multiplication of matrices over GF(2^m) yielding the modular inverse. Algorithm 23 performs the operand splitting phase in the `jumpdivstep` function (defined at lines 17–25), taking two polynomials $f(x), g(x)$ of maximum degree n and the difference between their degrees δ, and picking a splitting point j (line 20). The splitting point can be chosen as any non null portion of the maximum degree n, although splitting the operands in half is likely to be optimal. We report that the intuition of optimality of splitting the operands in half was verified to be the optimal choice in our implementations. Two subsequent recursive calls to the `jumpdivstep` function are made, splitting the input polynomials at their j-th degree term (lines 21 and 24), until the maximum degree n is equal or smaller than the machine word size (w in the algorithm). When this happens ("branch taken" at line 18 of Algorithm 23), the function handling the base case of the recursion, `divstep` is invoked. `divstep` can be seen as a clever reformulation of n iterations, of the loop body of Algorithm 21, with n being the parameter taken as input by `divstep`. The computation of the loop body is decomposed into a conditional swap (lines 5–9), depending only on the value of the difference between the operand degrees δ being positive and the constant term of $g(x)$ being equal to 1, and a sequence of steps (lines 10–13) performing the correct operation between the two portions of the source operands $f(x), g(x)$ and the auxiliary values.

Algorithm 23: Bernstein and Yang (BY) Inverse [14] specialized for GF(2^m)

Input: $f(x)$ irreducible polynomial of GF(2^m), m 2,
$a(x)$ invertible element of GF(2^m)

Output: $\mathsf{V}(x)$, such that $a(x)^{-1}$ $\mathsf{V}(x)$ 2 GF(2^m)

Data: w: machine word size

```
                                         // Base case, solved iteratively on n   w
1  function divstep(n, δ, f(x), g(x)):
2      U(x)   x^{n-1}; V(x)   0
3      Q(x)   0; R(x)   x^{n-1}
4      for i   0 to n   1 do
5          if (δ > 0 ^ g_0 = 1) then
6              δ     δ
7              SWAP(f(x), g(x))
8              SWAP(U(x), Q(x))
9              SWAP(R(x), V(x))
10         δ = δ + 1
11         Q(x)   (f_0  Q(x)   g_0  U(x))/x          // dropping the remainder
12         R(x)   (f_0  R(x)   g_0  V(x))/x          // dropping the remainder
13         g(x)   (f_0  g(x)   g_0  f(x))/x          // dropping the remainder
14     H   ( U(x)  V(x) )
           ( Q(x)  R(x) )
15     return δ, H
16
                              // Splitting case, shortens operands until n   w
17 function jumpdivstep(n, δ, f(x), g(x)):
18     if n   w then
19         return divstep(n, δ, f(x), g(x))
                  // any j > 0 is admissible, intuitive optimum at b n/2 e
20     j   b n/2 e                                    // integer part of n/2
21     δ, P   jumpdivstep(j, δ, f(x) mod x^j, g(x) mod x^j)
22     f'(x)   P_{0,0}  f(x) + P_{0,1}  g(x)
23     g'(x)   P_{1,0}  f(x) + P_{1,1}  g(x)
24     δ, Q   jumpdivstep(n   j, δ, f'(x)/x^j, g'(x)/x^j)   // dropping remainders
25     return δ, (P   Q)
26
                     // Main inverse function calling recursion splitting case
27 S(x)   MIRROR(f(x)), R(x)   MIRROR(a(x))
28 δ, H   jumpdivstep(2m   1, 1, S(x), R(x))
29 V(x)   MIRROR(H_{0,1})
30 return V(x)
```

This approach allows a constant-time implementation using for the `if` construct Boolean-predicate operations (lines 5–9).

To compute the polynomial inverse, Algorithm 23 invokes the `jumpdivstep` function on the reflected representation of both the modulus and the polynomial to be inverted, that is it considers the polynomials $\mathsf{S}(x), \mathsf{R}(x)$ of the same degree of $f(x)$ and $a(x)$, respectively, obtained swapping the coefficient of the x^i term with the one of the $x^{\mathsf{deg}(\mathsf{S}(x))-i}$ (resp., $x^{\mathsf{deg}(\mathsf{R}(x))-i}$) one. Both polynomials, represented as degree $2m-1$ ones, adding the appropriate null terms, are processed by the `jumpdivstep` call, which returns the final difference in degrees (expected to be null), and the reflected representation of the polynomial inverse in the second element of the first row of H.

Inversion with Fermat's Little Theorem

Finally, an approach to perform a constant-time implementation of the binary polynomial inversion is to employ the Fermat's little theorem. While this procedure is usually slower on a polynomial ring with a generic modulus, we obtain an efficient implementation of Fermat's method to compute inverses in $\mathbb{Z}_2[x]/(x^p-1)$, in case p is prime and $\mathrm{ord}_2(p)=p-1$ (i.e., 2 is a primitive element of $\mathrm{GF}(p)$), as it is the case in the LEDAcrypt parameters. Indeed, $\mathbb{Z}_2[x]/(x^m-1), m\geq 1$ can be seen as $\prod_i \mathbb{Z}_2[x]/((f^{(i)}(x))^{\lambda_i})$, where $f^{(i)}(x)$ are the irreducible factors of $x^m-1\in\mathbb{Z}_2[x]$, each with its own multiplicity $\lambda_i>0$. As a consequence, the number of elements in each $\mathbb{Z}_2[x]/\left(\left(f^{(i)}(x)\right)^{\lambda_i}\right)$ admitting an inverse is: $\left|\left(\mathbb{Z}_2[x]/(f^{(i)}(x))^{\lambda_i}\right)^*\right| = 2^{\deg(f^{(i)})\cdot\lambda_i} - 2^{\deg(f^{(i)})(\lambda_i-1)}$, and the multiplicative inverse of a unitary element in $\mathbb{Z}_2[x]/(x^m-1)$ can be obtained by raising it to the least common multiple ($\mathtt{lcm}$) of the said quantities:
$\mathtt{lcm}(2^{\deg(f^{(1)})\cdot\lambda_1} - 2^{\deg(f^{(1)})(\lambda_1-1)}, 2^{\deg(f^{(2)})\cdot\lambda_2} - 2^{\deg(f^{(2)})(\lambda_2-1)}, \ldots)-1$.

In our case, where $m=p$ and p is a prime number such that $\mathtt{ord}_2(p)=p-1$, the polynomial x^p-1 factors as the product $(x+1)\cdot(\sum_{i=0}^{p-1}x^i)$, and consequently the number of invertible elements is $\left(2^{1\cdot 1}-2^{1\cdot(1-1)}\right)\cdot\left(2^{(p-1)\cdot 1}-2^{(p-1)\cdot(1-1)}\right) = 2^{p-1}-1$, while the computation of the inverse of $a(x)\in((\mathbb{Z}_2[x]/(x^p-1)^*,\cdot)$ is obtained raising it to $2^{p-1}-2$, i.e., $a(x)^{-1}=a(x)^{2^{p-1}-2}$.

Noting that the binary representation of $2^{p-1}-2$ is obtained as a sequence of $p-2$ set bits followed by a null bit (i.e., $2^{p-1}-2=(11,\ldots,10)_{\mathtt{bin}}$), a simple (right-to-left) *square & multiply* strategy would compute the inverse employing $p-2$ modular squarings and $p-3$ multiplications (i.e., $a(x)^{-1}=a(x)^2\cdot a(x)^{2^2}\cdots a(x)^{2^{p-1}}$). However, as reported first in [19], it is possible to devise a more efficient square and multiply chain, tailoring it to the specific value of the exponent. Indeed, we are able to obtain a dedicated algorithm, reported as Algorithm 24, which computes the inversion with only $\lceil\log_2(p-2)\rceil+\mathtt{HammingWeight}(p-2)$ multiplications and $p-1$ squarings. Since the squaring operations are significantly more frequent than the multiplications, it is useful to exploit the fact that polynomial squaring can be computed very efficiently in $\mathbb{Z}_2[x]/(x^p-1)$. Indeed, two approaches are possible.

The first approach observes the fact that computing a square of an element of $\mathbb{Z}_2[x]$ is equivalent to the interleaving of its (binary) coefficients with zeroes, an operation which has linear complexity in the number of polynomial terms (as opposed to the quadratic complexity of a multiplication).

The second approach builds on the observation made in [24]: given an element $a(x)\in\mathbb{Z}_2[x]/(x^p-1)$, considering the set of exponents of its non-zero coefficient monomials S allows to rewrite $a(x)$ as $\sum_{j\in S}x^j$. It is known that, on characteristic 2 polynomial rings, we have that $\forall i\in\mathbb{N}, \left(\sum_{j\in S}(x^j)\right)^{2^i}\equiv\sum_{j\in S}(x^j)^{2^i}$, thus permitting us to obtain the 2^i-th power of $a(x)$ by computing the 2^i-th powers of a set of monomials, and then adding them together. To efficiently compute the 2^i-th power of a monomial, observe that the order of x in $\mathbb{Z}_2[x]/(x^p-1)$ is p (indeed, $x^p-1\equiv 0$ in our ring, therefore $x^p\equiv 1$). We therefore have that $(x^j)^{2^i}=(x^j)^{2^i \bmod p}$ in $\mathbb{Z}_2[x]/(x^p-1)$. This in turn implies that it is possible to compute the 2^i-th power of an element $a(x)\in\mathbb{Z}_2[x]/(x^p-1)$ simply permuting its coefficients according to the following permutation: the j-th coefficient of the polynomial $a(x)$, $0\leq j\leq p-1$, becomes the $((j\cdot 2^i)\bmod p)$-th coefficient of the polynomial $a(x)^{2^i}$. This observation allows to compute the 2^i-th power of an element in $\mathbb{Z}_2[x]/(x^p-1)$, again, at a linear cost (indeed, the one of permuting the coefficients); moreover the

Algorithm 24: Inverse based on Fermat's Little Theorem

Input: $a(x)$: element of $\mathbb{Z}_2[x]/(x^p \quad 1)$ with a multiplicative inverse.
Output: $c(x) = (a(x)^e)^2$, and $e{=}2^{p-2} \quad 1{=}(11\ldots1)_{\mathtt{bin}}$. The binary encoding of e is a sequence of $p \quad 2$ set bits.
Data: p: a prime such that $\mathrm{ord}_2(p) = p \quad 1$ (i.e., 2 is a primitive element of GF(p)).
The algorithm is instrinsically constant-time w.r.t. to the value of $a(x)$, as the control flow depends only on the value of p, which is not a secret.
Computational cost: $d\log_2(p \quad 2)e \quad 1 +$ `HammingWeight`$(p \quad 2)$ polynomial multiplications, plus $p \quad 1$ squarings

1 `exp` BinEncoding(p 2)
2 `expLength` $d\log_2(p \quad 2)e$
  `// scan of exp from right to left;` $l{=}0$ \$ `LSB`
3 $b(x) \quad a(x)$ `// exp`$_0$`=1 as` p `2 is an odd number`
4 $c(x) \quad a(x)$
5 **for** $l \quad 1$ **to** `expLength` 1 **do**
6   $c(x) \quad c(x)^{2^{2^{l-1}}} \quad c(x)$ `//` 2^{l-1} `squarings, 1 mul.`
7   **if** `exp`$_l$ $= 1$ **then**
8     $b(x) \quad b(x)^{2^{2^i}}$ `//` 2^l `squarings`
9     $b(x) \quad b(x) \quad c(x)$ `// 1 multiplication`
10 $c(x) \quad (b(x))^2$ `// 1 squaring`
11 **return** $c(x)$

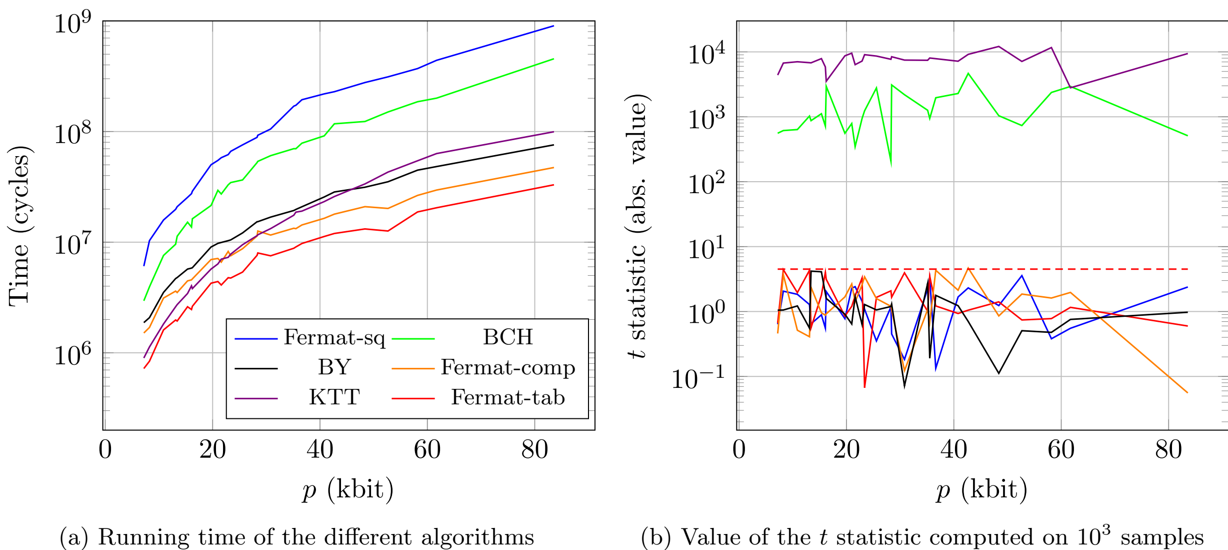
permutation which must be computed is fixed, and depends only on the values p and 2^i, which are both public and fixed, thus avoiding any meaningful information leakage via the timing side channel. The authors of [24] observe that, since the required permutations, one for each value of $(2^i \bmod p)$, with $i \in \{2^1, 2^2, \ldots, 2^{\lceil \log_2(p-2) \rceil}\}$ are fixed, they can be precomputed, and stored in lookup tables.

We note that the precomputation of such tables, while feasible, is likely to take a non-negligible amount of memory. Indeed, such tables require $p \cdot (\lceil \log_2(p-2) \rceil - 1) \cdot \lceil \log_2(p) \rceil$ bits to be stored, assuming optimal packing in memory. This translates into tables ranging between 136 kiB and 2,856 kiB for the recommended prime sizes in LEDAcrypt [6]. While these table sizes are not problematic for an implementation targeting a desktop platform, more constrained execution environments, such as microcontrollers, may not have enough memory to store the full tables.

To this end, we introduce and examine a computational tradeoff, where only a small lookup table comprising the $(\lceil \log_2(p-2) \rceil - 1)$ values obtained as $(2^i \bmod p)$, with $i \in \{2^1, 2^2, \ldots, 2^{\lceil \log_2(p-2) \rceil}\}$, is precomputed and stored, while the position of the j-th coefficient, $0 \leq j \leq p-1$, of $a(x)^{2^i}$ is computed via a multiplication and a (single-precision) modulus operation online, as $(j \cdot (2^i \bmod p)) \bmod p$.

This effectively reduces the tables to a size equal to $(\lceil \log_2(p-2) \rceil - 1) \cdot \lceil \log_2(p) \rceil$ bits, which corresponds to less than 1 kiB for all the LEDAcrypt parameters.

Finally, the authors of [24] also note that, on `x86_64` platforms, it is faster to precompute the inverse permutation with respect to the one corresponding to raising $a(x)$ to 2^i, as it allows the collection of binary coefficients of the result $a(x)^{2^i}$ that are contiguous in their memory representation. This is done determining, for each coefficient of $a(x)^{2^i}$, which was its corresponding one in $a(x)$ through the inverse permutation. The said inverse permutation maps the coefficient of the a-th degree term in $a(x)^{2^i}$, $0 \leq a \leq p-1$, back to the coefficient of the $(a \cdot (2^{-i} \bmod p) \bmod p)$-th term in $a(x)$. We



(a) Running time of the different algorithms

(b) Value of the t statistic computed on 10^3 samples

Figure 5.3: Experimental evaluation of the four modular inverse algorithms. (a) average running time in clock cycles over 10^3 computations. (b) absolute value of the t-statistic for a Student's t with two populations of 10^3 execution times for each algorithm. The dashed horizontal line highlights the upper bound of the range $[-4.5, 4.5]$, pointing out no data-dependent changes in the timing behaviors of the algorithms below it with a confidence of 99.999% (significance level $\alpha = 10^{-5}$)

note that, also in this case, it is possible to either tabulate the entire set of inverse permutations, as suggested by [24], or simply tabulate the values of $(2^{-i} \bmod p)$ and determine each position of the permutation at hand partially at runtime.

5.4.3 Experimental Evaluation

We implemented in `C11` all the aforementioned inversion algorithms, employing appropriate compiler intrinsics to exploit the features of the `AVX2` ISA extensions available on the Intel Haswell microarchitecture, selected by NIST as the standard evaluation platform for desktop applications. In particular, we employed vector Boolean instructions to speed up the required `xor` and `shift` operations which make up a significant portion of the reported algorithms.

We exploited the presence of the *carryless multiplication instruction* (`pclmulqdq`) which performs a polynomial multiplication of two 64-bit elements in a 128-bit one. We implemented the polynomial multiplication primitive applying the Toom-Cook (TC) optimized interpolation proposed in [16] until the recursive operand-splitting computation path yields operands below 50, 64-bit machine words. Then we employ a Karatsuba multiplication technique, with all multiplications involving operands below 9 machine words performed picking the optimal sequence of addition & multiplication instructions according to [70]. We determined the number of machine words where the multiplication strategy is changed (between the TC and Karatsuba strategies) via exhaustive exploration of the tradeoff points.

Concerning the exponentiations to 2^i, as required by the optimized Fermat's little based inverse, we investigated the effects of the tradeoff reported in [24], where it is noted that, for small values of i, it can be faster to compute the exponentiation to 2^i by repeated squaring, instead of resorting

to a bitwise permutation. Indeed, it is possible to obtain a fast squaring exploiting the pclmulqdq instruction which performs a binary polynomial multiplication of two, 64 terms, polynomials in a 128 terms one. While it may appear counterintuitive, this approach is faster than the use of the dedicated *Parallel Bits Deposit* (pdep) instruction available in the AVX2 instruction set. Indeed, the throughput obtained with the parallel bit deposit instruction is lower than the one obtained via pclmulqdq. This is due to the possibility of performing two pclmulqdq bit-interleaving starting from a 128-bit operand which can be loaded in a single instruction, as opposed to two pdep which need to expand a 64-bit wide operand only. This fact, combined with the low latency of the pclmulqdq instruction (5 to 7 cycles, depending on the microarchitecture, with a number of Clock cycles Per Instruction (CPI) of 1 when the pipeline is full, while the pdep instruction has a 3 cycles latency, to act on operands having half the size) provides a fast zero interleaving strategy. We also recall that the implementation of the pdep instruction on AMD CPUs is quite slow (50+ cycles) and non constant-time, as it is implemented in microcode. As a consequence, the use of the pdep instruction would lead to a scenario where the non constant-time execution on an (AVX2 ISA extension compliant) AMD CPU can compromise the security of the implementation. Finally, we note that the pclmulqdq bit-interleaving strategy performs better than the one relying on interleaving by Morton numbers [64], employing the 256-bit registers available via the AVX2 ISA extension, plus the corresponding vector shift and vector or instructions.

We computed the polynomial inversions picking the modulus $f(x){=}x^p{-}1{\in}\mathrm{GF}(2^p)[x]$ for all the values of p indicated in use in the ephemeral key exchange LEDAcrypt-KEM-CPA (Table 4.4) , and in the IND-CCA2 LEDAcrypt-KEM and LEDAcrypt-PKC with their two-iterations out-of-place decoder (Table 4.1).

Figure 5.3 (a) provides a depiction of the average number of clock cycles taken to compute each polynomial inversion algorithm, computed over 10^3 runs of it, acting on a randomly drawn, invertible, polynomial on the Intel based machine. As it can be seen, the inverse computation strategy exploiting large tables and Fermat's little theorem proves to be the most efficient one for the entire range of primes which are employed in the LEDAcrypt specification. A further noteworthy point is that, if a compact memory footprint is desired, and the constant-time property is not strictly required, as it is the case for ephemeral keypairs, Algorithm 22 provides a valid alternative to the method relying on Fermat's little theorem, with compact lookup tables (Fermat-comp in the legend) for the low range of prime sizes. We also note that an implementation of Fermat's little polynomial inverse, employing only repeated squarings via bit interleaving, instead of a permutation based computation of the exponentiation to 2^i is significantly slower (Fermat-sq in Figure 5.3) than all other methods. Precomputing only the smaller table of the values $2^i \bmod p, i \in 2^1, 2^2, \ldots, 2^{\lceil \log_2(p-2) \rceil}$ (the Fermat-comp line in Figure 5.3) still provides a good amount of the speedup obtained by the precomputation,the full speedup being the one obtained by Fermat-tab, while employing very little memory $<$ 1kiB; Finally, we note that, if the modular inverse is not being computed on a polynomial ring having a modulus with a peculiar structure, such as $\mathbb{Z}_2[x]/(x^p-1)$, the benefits of performing the inversion via Fermat's little theorem may be smaller, or nullified, with respect to a completely general purpose inversion algorithm such as the one by Bernstein and Yang which, from our analysis, provides competitive performance and is constant time.

We note that the implementation of the algorithms expected to be running in constant time (Algorithm 23 (BY in the legend) and Algorithm 24 (Fermat-sq, Fermat-comp, and Fermat-tab in the legend)) relies on their implementation not containing any secret-dependent branches, nor any memory lookups whose address depends on a secret value. To provide an experimental validation of the said fact Figure 5.3 (b) reports the result of the validation of the constant-time property

by means of a Student t-test, for each value of p, done on two timing populations of 10^3 samples each taken, one when computing inverses of random invertible polynomials, and the other when computing the inverse of the $x^2 + x + 1$ trinomial. The choice of a trinomial is intended to elicit the leakage of the position of the (very sparse) set coefficients, which represents the secret not to be disclosed in cryptoschemes such as LEDAcrypt, as the position of the few set coefficients is clustered at one end of the polynomial.

The values of the t-statistic, represented in absolute value in Figure 5.3 (b), for Algorithm 23 (BY in the legend) and Algorithm 24 (all Fermat variants in the legend) are within the range $[-4.5, 4.5]$ (upper bound indicated with a dashed horizontal line in the figure), in turn implying that the t-test is not detecting data-dependent changes in the timing behaviors with a confidence of 99.999% (significance level $\alpha = 10^{-5}$). By contrast, both the method by Brunner *et al.* (BCH in the legend), and the method by Kobayashi *et al.* (KTT in the legend) show t statistic values far out of the interval $[-4.5, 4.5]$, exposing (as expected) their non constant time nature.

Finally, we note that on a desktop platform, given a polynomial to be inverted $a(x)$, making the KTT and BCH algorithms immune to the timing side channel by replacing the computation of $a(x)^{-1}$ with $\lambda(x) \cdot (\lambda(x) \cdot a(x))^{-1}$, where $\lambda(x)$ is a randomly chosen unitary polynomial, will increase further their performance penalty w.r.t. the Fermat-tab solution.

5.5 Parameter Tuning for LEDAcrypt-KEM-CPA

The LEDAcrypt-KEM-CPA cryptosystem as described in Chapter 1 and Section 4.2 has been designed with QC-MDPC code parameters targeting use-cases featuring ephemeral public/private keypairs. Indeed, these use-cases involve a one round-protocol where *i)* the sender generates her own public/private keypair and sends the public key to the receiver, together with the other public information on the scheme; *ii)* the receiver employs the Encapsulation algorithm (ENCAP) fed with the public key of the sender to generate both a secret symmetric-key and an encrypted payload corresponding to the encryption of the said secret key; the payload is in turn sent back to the sender to complete the protocol and to allow her to derive the same secret key value executing the decapsulation (DECAP) algorithm fed with her own private key and the encrypted payload.

Assuming both sender and receiver equipped with two instances of the same computational platform, a first figure of merit assessing the performance of the implementation of the ephemeral-key KEM scheme is defined as the sum of the execution timings of the KEYGENERATION (T_{KeyGen}), ENCAP (T_{Encap}) and DECAP (T_{Decap}) algorithms, measured as the required number of machine clock cycles, i.e.: $T_{\mathsf{KEM-CPA}} = T_{\mathsf{KeyGen}} + T_{\mathsf{Encap}} + T_{\mathsf{Decap}}$.

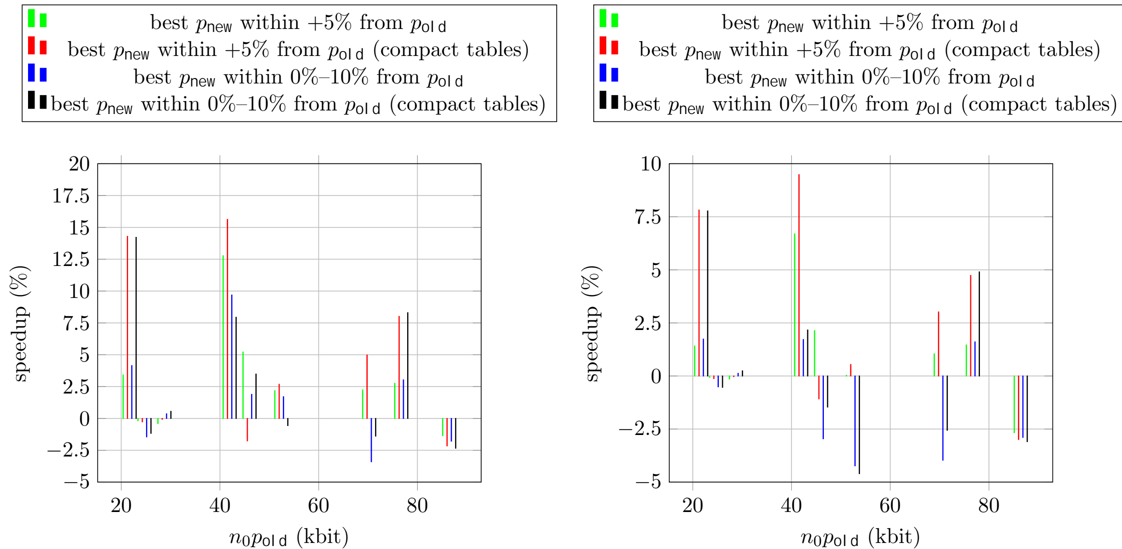
A second figure of merit, which we employed also in the decoder performance analysis in Section 5.1, keeping into account also the computational overhead experienced at both the receiver and sender side to process the byte-streams exchanged over the network to transmit the public key and the encrypted payload, respectively, is defined by adding to $T_{\mathsf{KEM-CPA}}$ also the total number of transmitted bytes, $\mathsf{N_{bytes}}$, multiplied by a normalization coefficient β (measured as machine clock cycles over bytes), i.e.: $T_{\mathsf{combined}} = T_{\mathsf{KEM-CPA}} + \beta \cdot \mathsf{N_{bytes}}$. An empirical value for such coefficient, which is typically accepted as modeling the computational overhead due to the execution of the routines in the software-stack employed for communication is $\beta = 1000$.

The use of a code profiling tool to analyze the software implementation of LEDAcrypt-KEM-CPA reveals that the time to compute the single multiplicative inverse required during the execution of the KEYGENERATION algorithm takes from 40% to 60% of the total execution time $T_{\mathsf{KEM-CPA}}$, as the rate $\frac{n_0-1}{n_0}$, $n_0 \in \{2,3,4\}$, of the underlying QC-MDPC code varies among the code configurations reported in Table 4.4 in Section 4.2. In Section 5.4.2 and Section 5.4.3, we presented the algorithm based on the application of the Fermat's little theorem (see Algorithm 24) as the most effective strategy to compute the multiplicative inverse of an element in $\mathbb{Z}_2[x]/(x^p-1)$, given the LEDAcrypt-KEM-CPA parameters in Section 4.2. The computational complexity of the optimized implementation of the Algorithm 24 highlights the need to compute the following operations to invert an element $a(x) \in \mathbb{Z}_2[x]/(x^p-1)$:

$$(\lceil(\log_2(p-2)\rceil - 1 + \mathsf{HammingWeight}(p-2)) \times (\textsc{PolMul} + \textsc{ExpPowerTwo}(i)) + 1 \times \textsc{PolSquaring}$$

where POLMUL denotes a polynomial multiplication, POLSQUARING denotes a polynomial squaring, while EXPPOWERTWO(i) denotes the computation of $a(x)^{2^i}$, with $i \in \{2^1, 2^2, \ldots, 2^{\lceil \log_2(p-2) \rceil}\}$. The optimized implementation of the inversion algorithm features the EXPPOWERTWO(i) routine implemented with the aid of either a compact or a full permutation table (refer to Sections 5.4.2 and 5.4.3 for further details) that in turn exhibit the same execution time, respectively, regardless of the input parameter i. Concerning the polynomial multiplication algorithm, the DENSE-BY-DENSE_MULTIPLICATION strategy described in Section 5.2 is applied, as the SPARSE-BY-DENSE_MULTIPLICATION strategy does not fit the operands involved in the inversion algorithms. From these observations it is easy to infer that an inversion algorithm featuring values for the p parameter that exhibit a low $\mathsf{HammingWeight}(p-2)$ may provide better execution timings.

It is worth noting that only p values with $\mathsf{ord}_2(p) = p-1$ (i.e., such that 2 is a primitive element of $\mathbb{Z}_p$) that are larger than the ones shown in Section 4.2 allow to keep the same security guarantees. This consideration, together with the one that increasing the size of p will inevitably introduce a performance penalty in the execution of both the decoding and the multiplications algorithm, justifies our choice of restricting to the range $[p, p \times 1.1]$ the search for new values to be assigned to the p parameters of the distinct instances of LEDAcrypt-KEM-CPA. Indeed, the potential speedup of T_{combined} and/or $T_{\mathsf{KEM-CPA}}$ figures of merit will quickly decrease as p increases.



(a) Figure of merit: total execution time, $T_{\mathsf{KEM-CPA}}$

(b) Figure of merit: total execution time plus the time needed to process the transmitted/received byte-streams, $T_{\mathsf{combined}} = T_{\mathsf{KEM-CPA}} + \beta\ \mathsf{N_{bytes}}$, with $\beta = 1000$

Figure 5.4: Speedups or slowdowns obtained for the LEDAcrypt-KEM-CPA figures of merit (key generation+encryption+decryption time,$T_{\mathsf{KEM-CPA}}$, and related $T_{\mathsf{combined}} = T_{\mathsf{KEM-CPA}} + \beta \cdot \mathsf{N_{bytes}}$), when employing the values of p larger than the ones required by security requirements alone

Figure 5.4 shows the speedups/slowdowns of the performance of all LEDAcrypt-KEM-CPA instances, with respect to the figures of merit $T_{\mathsf{KEM-CPA}}$ (Figure 5.4(a)) and T_{combined} (Figure 5.4(b)), when the parameter p in each code configuration in Table 4.4 in Section 4.2, denoted as p_{old} from now on, is increased up to 10%. Specifically, the x-axes report the error vector size $n_0 p_{\mathsf{old}}$ for all QC-MDPC code configurations in Table 4.4 (i.e., $n_0 \in \{2, 3, 4\}$ and NIST Category equal to 1, 3, 5), while the y-axes denotes the speedups/slowdowns of the figure of merit at hand computed as $100 \times \frac{T_{p_{\mathsf{new}}} - T_{p_{\mathsf{old}}}}{T_{p_{\mathsf{old}}}}$.

The green and blue bars consider LEDAcrypt-KEM-CPA instances featuring an inverse algorithm implemented with the aid of full permutation tables and refer to p_{new} values that yield the best results when picked in a range within 0%-5% and 0%-10% from p_{old}, respectively. The red and black bars consider LEDAcrypt-KEM-CPA instances featuring an inverse algorithm implemented with the aid of compact permutation tables (useful to have a portable implementation for platforms with limited memory requirements), and refer to p_{new} values that yield the best results when picked in a range within 0%-5% and 0%-10% from p_{old}, respectively.

The three macro-groups of bars, from left to right, in each figure, refer to QC-MDPC code param-

Table 5.2: New p values for the LEDAcrypt-KEM-CPA instances replacing the ones reported in Table 4.4 in Section 4.2

NIST Category	n_0	p_{old}	p_{new}	$p_{new} - p_{old}$	$\mathrm{HW}(p_{new} - 2) - \mathrm{HW}(p_{old} - 2)$
1	2	10,853	10,883	30	-2
	3	8,237	8,237	0	0
	4	7,187	7,187	0	0
3	2	20,981	21,011	30	-4
	3	15,331	15,373	42	-2
	4	13,109	13,109	0	0
5	2	35,117	35,339	222	-2
	3	25,579	25,603	24	-5
	4	21,611	21,611	0	0

eters addressing the requirements of the NIST Categories 1, 3 and 5, respectively. Within each macro-group of bars, from left to right, in each figure, bundles of four bars (green, red, blue, black) refer to QC-MDPC code parameters with n_0 equal to 2, 3 and 4, respectively.

As it is evident from both Figure 5.4(a) and Figure 5.4(b), not all LEDAcrypt-KEM-CPA instances benefit from an enlargement of the parameter p. In particular, focusing on the figure of merit T_{combined} shown in Figure 5.4(b), we updated the values of p reported in Table 4.4 in Section 4.2 as shown in Table 5.2, raising the value of p where either a gain above 2.5% in the metric can be achieved by the solution with the compact-tables inverse, or the increase in the value of p is negligible (i.e. < 50).

Chapter 6

LEDAcrypt performance evaluation and recommended parameters

In this chapter, we provide an analysis of the overall performance of the LEDAcrypt primitives, and provide recommendations on the choices of the code rate to be made to optimize two key figures of merit: the computation speed of the primitive and the combined execution time-transmitted data metric. The timing results reported for the IND-CCA2 primitives LEDAcrypt-KEM and LEDAcrypt-PKC relate to a **constant time** implementation.

6.1 Key-lengths, ciphertext and transmitted data sizes

In this section, we consider the key and ciphertext lengths for the LEDAcrypt primitives. Concerning LEDAcrypt-KEM-CPA, Table 6.1 reports all the key sizes, ciphertext sizes and overall transmitted data during a key agreement, as a function of the NIST category and n_0. It can be noticed that the amount of transmitted data grows with the increase of n_0, while the ciphertext alone of the LEDAcrypt-KEM-CPA shrinks. Indeed, this fact is due to the shrinking of the value of p for codes sharing the same NIST Category and with increasing n_0, as the ciphertext of LEDAcrypt-KEM-CPA is indeed a p-bit long syndrome (plus some negligible overhead), while the entirety of transmitted data also includes the public key (as it is intended for ephemeral use).

The growth trend of the transmitted amount of data is reversed in LEDAcrypt-KEM (data reported in Table 6.2), where there is no need to retransmit the public key at each communication, indeed limiting the ciphertext to the p-bit syndrome alone. This in turn allows LEDAcrypt-KEM to transmit smaller amounts of data (for the same NIST Category) than LEDAcrypt-KEM-CPA. However, the transmitted overhead of providing the ephemeral key service employing LEDAcrypt-KEM-CPA, in scenarios where perfect forward secrecy is desired, such as TLS 1.3, is limited to 704 extra bytes to be sent.

We note that the configurations of LEDAcrypt-KEM-CPA and LEDAcrypt-KEM keep the amount of transmitted data well below the 20,480 B mark (down to a quantity between a half and a third of it), which was indicated by Stebila *et al.* in [21] to be causing issues in some implementations of OpenSSL, and definitely below the $2^{16} - 1 = 65,535$ B mark size of the public key field in the TLS 1.3 handshake protocol as reported in [21]. Indeed, in [21] the authors report the successful

Table 6.1: Key, ciphertext and transmitted data sizes for **LEDAcrypt-KEM-CPA** instances employing the QC-MDPC parameters reported in Table 4.4 in Section 4.2, and partially updated as shown in Table 5.2 in Section 5.5

NIST Category	n_0	Private Key (B)	Public Key(B)	Ciphertext (B)	Shared Secret (B)	Transmitted data (B)
1	2	1,160	1,368	1,392	32	2,760
	3	1,920	2,064	1,056	32	3,120
	4	2,680	2,712	928	32	3,640
3	2	1,680	2,632	2,664	48	5,296
	3	2,840	3,856	1,960	48	5,816
	4	3,968	4,920	1,672	48	6,592
5	2	2,232	4,424	4,464	64	8,888
	3	3,760	6,416	3,248	64	9,664
	4	5,256	8,112	2,744	64	10,856

integration of LEDAcrypt-KEM-CPA and LEDAcrypt-KEM primitives at all its security levels, with the parameter sets indicated in the second version of the LEDAcrypt specification, upon which we improved in keysize in this LEDAcrypt specification release.

As a final remark, it is worth noting that the sizes of private keys reported in Table 6.1 for the LEDAcrypt-KEM-CPA instances are larger than the ones in Table 6.2 or Table 6.3 for the IND-CCA2 LEDAcrypt instances. The reason for the minimally sized private keys for the IND-CCA2 LEDAcrypt primitives is that the amount of time required to regenerate the private key, i.e., the sparse matrix H, from either a XOF seed or a DRBG seed, is negligible, therefore allowing the users to store only the seed itself. In an ephemeral key use scenario, such as the one where LEDAcrypt-KEM-CPA is intended to be used, such a procedure does not yield practical advantages since the private key is already generated in its expanded form in memory, and such a form is the one required to perform the decryption operation. Since the decryption operation is expected to be taking place on the same machine, temporally close to the key generation, and with the explicit requirement of not storing private key material on mass memory, employing the key compression technique applied to LEDAcrypt-KEM would be detrimental (although by a negligible amount) to the computational performance of LEDAcrypt-KEM-CPA, without yielding any advantage (as the full private key needs to be present in memory for the decryption).

Table 6.2: Key, ciphertext and transmitted data sizes for the **IND-CCA2 LEDAcrypt-KEM** instances employing the QC-MDPC parameters reported in Table 4.1 in Section 4.1 and the LEDAdecoder designed to execute two out-of-place iteration functions

NIST Category	DFR	n_0	Private Key (B)	Public Key (B)	Ciphertext (B)	Shared Secret (B)	Transmitted data (B)
1	2^{-64}	2	50	2,928	2,952	32	2,952
		3	50	4,032	2,040	32	2,040
		4	50	5,040	1,704	32	1,704
	2^{-128}	2	50	3,536	3,560	32	3,560
		3	50	4,928	2,488	32	2,488
		4	50	6,096	2,056	32	2,056
3	2^{-64}	2	66	5,104	5,136	48	5,136
		3	66	7,104	3,584	48	3,584
		4	66	8,592	2,896	48	2,896
	2^{-192}	2	66	6,584	6,616	48	6,616
		3	66	9,168	4,616	48	4,616
		4	66	11,568	3,888	48	3,888
5	2^{-64}	2	82	7,720	7,760	64	7,760
		3	82	10,672	5,376	64	5,376
		4	82	13,320	4,480	64	4,480
	2^{-256}	2	82	10,448	10,488	64	10,488
		3	82	14,544	7,312	64	7,312
		4	82	18,144	6,088	64	6,088

Table 6.3: Key, ciphertext and transmitted data sizes for the **IND-CCA2 LEDAcrypt-PKC** instances employing the QC-MDPC parameters reported in Table 4.1 in Section 4.1 and the LEDAdecoder designed to execute two out-of-place iteration functions

NIST Category	DFR	n_0	Private Key (B)	Public Key (B)	Min. Ciphertext Overhead (B)	Max. Ciphertext Overhead (B)
1	2^{-64}	2	26	2,928	2,967	5,856
		3	26	4,032	2,064	6,048
		4	26	5,040	1,729	6,720
	2^{-128}	2	26	3,536	3,570	7,072
		3	26	4,928	2,497	7,392
		4	26	6,096	2,075	8,128
3	2^{-64}	2	34	5,104	5,150	10,208
		3	34	7,104	3,594	10,656
		4	34	8,592	2,909	11,456
	2^{-192}	2	34	6,584	6,625	13,168
		3	34	9,168	4,635	13,752
		4	34	11,568	3,913	15,424
5	2^{-64}	2	42	7,720	7,774	15,440
		3	42	10,672	5,387	16,008
		4	42	13,320	4,493	17,760
	2^{-256}	2	42	10,448	10,497	20,896
		3	42	14,544	7,322	21,816
		4	42	18,144	6,101	24,192

6.2 Performance evaluation considering execution times in milliseconds

In this section, we report the measured running times for LEDAcrypt-KEM, LEDAcrypt-PKC, and LEDAcrypt-KEM-CPA on our experimental evaluation setup, constituted by an Intel Core i5-6600, clocked at 3.2 GHz, where the turbo-boost feature was disabled.

We report that LEDAcrypt-KEM-CPA takes around 600μs for the computation of an entire ephemeral key exchange (namely, key generation, encapsulation and decapsulation) for NIST category 1, with the best timing achieved by the solution featuring a code with $n_0 = 4$. Indeed, picking $n_0 > 2$ is the favourable solution to minimize the execution time of LEDAcrypt-KEM-CPA for all the NIST categories. To put the computation time of LEDAcrypt-KEM-CPA in perspective, the OpenSSL 1.1.1d benchmark for the elliptic curve Diffie-Hellman key exchange for the NIST-P-384 curve, reports an execution time of 1.08 ms on the same experimental workbench, and the computation of the encryption and decryption primitives alone (without key generation) of RSA take 645 μs with a 2048 bit modulus, and 1.93 ms with a 3072 bit modulus. LEDAcrypt-KEM-CPA thus provides comparable computation times with current ephemeral key exchange primitives.

The execution times reported for LEDAcrypt-KEM, (Table 6.5) show that, for parameters meeting the strict requirement providing a DFR of $2^{-\lambda}$ (where $2^{\lambda}, \lambda \in \{128, 192, 256\}$, is the expected security margin of the scheme) to attain IND-CCA2 guarantees, LEDAcrypt-KEM takes around 1ms to complete the encapsulation and decapsulation procedure, a time indeed comparable with the RSA-3072 key exchange, which provides the same expected security margin.

We note that, in the case of LEDAcrypt-KEM, the best timings for the overall long term key exchange (encapsulation plus decapsulation) are achieved with choices of $n_0 \in \{2, 3\}$, depending on the NIST category. This change with respect to LEDAcrypt-KEM-CPA is due to the fact that the increase in the code length $n_0 p$ turns in a computational requirement increase of the decoder, which is not compensated by the smaller size of the multiplication operands for the n_0 overall multiplications performed in the encryption and decryption primitives.

We note that the entire implementation of the LEDAcrypt-KEM and LEDAcrypt-PKC primitives provides **constant time execution**, preventing the information leakage via the timing side channel altogether. The high variance measured in the LEDAcrypt-KEM and LEDAcrypt-PKC key generation processes is entirely due to the key rejection sampling procedure, which restarts the *constant time* key generation procedure altogether if the produced key does not meet the requirements described in Chapter 3, effectively discarding all previous key related information. Therefore, such a time variation, does not provide any information on the generated keypair.

Finally, we report the execution times for the LEDAcrypt-PKC primitive in Table 6.6, taken when encrypting a random, 1 kiB sized message. A message with the said size is always less than the information word size (pn_0-bit) in input to the encryption primitive. From the reported timings it can be seen that the impact of the Kobara-Imai construction on the overall running time of the KEM is around 30%, and mostly due to the increase in the encryption time required by the *constant-time* constant weight encoding primitive, implemented according to [7], as well as to the increase in the decryption time required by the computation of the syndrome to be fed to the decoder (as LEDAcrypt-PKC is McEliece-based scheme).

Table 6.4: Execution times in **milliseconds** for the **LEDAcrypt-KEM-CPA** instances

NIST Category	n_0	KeyGen (ms)	Encap (ms)	Decap (ms)	KeyGen+Encap+Decap (ms)
1	2	0.390 (± 0.009)	0.035 (± 0.001)	0.247 (± 0.011)	0.671
	3	0.313 (± 0.002)	0.032 (± 0.001)	0.276 (± 0.035)	0.620
	4	0.271 (± 0.008)	0.029 (± 0.003)	0.305 (± 0.016)	0.605
3	2	1.016 (± 0.003)	0.083 (± 0.002)	0.716 (± 0.027)	1.816
	3	0.810 (± 0.002)	0.070 (± 0.001)	0.876 (± 0.026)	1.755
	4	0.614 (± 0.002)	0.068 (± 0.001)	0.976 (± 0.087)	1.658
5	2	2.282 (± 0.006)	0.168 (± 0.004)	1.759 (± 0.014)	4.209
	3	1.181 (± 0.018)	0.155 (± 0.001)	1.879 (± 0.078)	3.215
	4	1.316 (± 0.017)	0.151 (± 0.008)	2.141 (± 0.199)	3.608

Table 6.5: Execution times in **milliseconds** for the IND-CCA2 **LEDAcrypt-KEM** instances

NIST Category	DFR	n_0	KeyGen (ms)	Encap (ms)	Decap (ms)	Encap+Decap (ms)
1	2^{-64}	2	1.393 (± 0.087)	0.076 (± 0.001)	0.561 (± 0.002)	0.637
		3	1.277 (± 0.128)	0.088 (± 0.001)	0.653 (± 0.001)	0.741
		4	1.009 (± 0.120)	0.078 (± 0.006)	0.684 (± 0.011)	0.762
	2^{-128}	2	2.281 (± 0.162)	0.110 (± 0.001)	0.939 (± 0.002)	1.049
		3	1.851 (± 0.527)	0.099 (± 0.001)	0.880 (± 0.002)	0.979
		4	1.833 (± 0.433)	0.124 (± 0.004)	0.956 (± 0.014)	1.080
3	2^{-64}	2	3.479 (± 0.193)	0.194 (± 0.001)	1.691 (± 0.004)	1.885
		3	2.717 (± 0.061)	0.192 (± 0.001)	1.698 (± 0.008)	1.890
		4	2.630 (± 0.787)	0.167 (± 0.010)	1.911 (± 0.018)	2.078
	2^{-192}	2	4.775 (± 0.438)	0.248 (± 0.001)	2.207 (± 0.008)	2.454
		3	3.610 (± 0.360)	0.267 (± 0.008)	2.659 (± 0.024)	2.926
		4	3.822 (± 0.750)	0.318 (± 0.001)	2.900 (± 0.009)	3.219
5	2^{-64}	2	7.253 (± 0.625)	0.433 (± 0.002)	3.971 (± 0.006)	4.404
		3	4.378 (± 0.150)	0.375 (± 0.003)	3.658 (± 0.015)	4.033
		4	4.117 (± 0.290)	0.413 (± 0.001)	4.821 (± 0.006)	5.234
	2^{-256}	2	10.746 (± 0.865)	0.605 (± 0.002)	4.870 (± 0.007)	5.476
		3	7.476 (± 1.079)	0.587 (± 0.001)	5.779 (± 0.011)	6.365
		4	5.695 (± 1.008)	0.506 (± 0.001)	5.889 (± 0.022)	6.395

Table 6.6: Execution times in **milliseconds** for the IND-CCA2 **LEDAcrypt-PKC** instances

NIST Category	DFR	n_0	KeyGen (ms)	Encypt (ms)	Decrypt (ms)	Encypt+Decrypt (ms)
1	2^{-64}	2	1.355 (± 0.095)	0.217 (± 0.011)	0.692 (± 0.011)	0.909
		3	1.263 (± 0.086)	0.267 (± 0.011)	0.859 (± 0.011)	1.126
		4	1.094 (± 0.184)	0.301 (± 0.002)	0.941 (± 0.015)	1.242
	2^{-128}	2	2.311 (± 0.224)	0.273 (± 0.011)	1.080 (± 0.012)	1.353
		3	1.973 (± 0.617)	0.344 (± 0.002)	1.079 (± 0.018)	1.424
		4	1.759 (± 0.335)	0.434 (± 0.010)	1.285 (± 0.015)	1.719
3	2^{-64}	2	3.548 (± 0.162)	0.449 (± 0.002)	1.923 (± 0.032)	2.372
		3	2.768 (± 0.154)	0.487 (± 0.012)	2.103 (± 0.028)	2.590
		4	2.706 (± 0.909)	0.514 (± 0.011)	2.399 (± 0.020)	2.913
	2^{-192}	2	4.781 (± 0.398)	0.524 (± 0.004)	2.484 (± 0.032)	3.008
		3	3.642 (± 0.306)	0.677 (± 0.003)	2.854 (± 0.011)	3.531
		4	3.633 (± 0.407)	0.808 (± 0.011)	3.604 (± 0.028)	4.412
5	2^{-64}	2	7.484 (± 0.861)	0.800 (± 0.003)	3.949 (± 0.035)	4.749
		3	4.427 (± 0.181)	0.829 (± 0.015)	4.266 (± 0.028)	5.095
		4	4.175 (± 0.347)	0.929 (± 0.016)	5.375 (± 0.033)	6.304
	2^{-256}	2	11.217 (± 1.323)	1.004 (± 0.011)	5.481 (± 0.033)	6.485
		3	7.517 (± 0.895)	1.147 (± 0.004)	6.076 (± 0.014)	7.223
		4	5.434 (± 0.592)	1.209 (± 0.004)	7.093 (± 0.010)	8.302

6.3 Performance evaluation considering execution times in clock cycles and their combination with the size of transmitted data

In this section, we analyze the running times of the LEDAcrypt primitives, expressed as clock cycle counts on the benchmarking platform, together with the combined time-transmitted data metric, to highlight which choices of the parameter n_0 provide the best results.

Concerning the case of LEDAcrypt-KEM-CPA (Table 6.7), we observe that the fastest solutions for the entire ephemeral KEM are the ones relying on $n_0 = 4$ for category 1 and 3, and on $n_0 = 3$ for category 5. This effect is due to the significant speed gains obtained by reducing the size of the operand of the modular polynomial inverse operation. While evaluating speed alone leads to the aforementioned n_0 choices, if the amount of transmitted data is taken into account combining it with the computation time by multiplying the number of transmitted bytes by $\beta = 1000$ and adding it to the clock cycle count, the most efficient solutions become the ones based on $n_0 = 2$ or $n_0 = 3$. This change is due to the impact of the increase in the transmitted data which, considering a normalization coefficient $\beta = 1000$ overcomes the speed advantage. While this provides a reasonable hint in terms of the best tradeoff when considering the computation time and cost of transmission together, we recall that such a metric may not fit extreme scenarios where the communication link throughputs and latencies are either very good (i.e., high throughput, low latency, as on an intra-device communication bus), or very bad (e.g., over the air links with limited bandwidth, such as the ones employed by the LoRa protocol in a low-power wide-area network).

Analyzing the speed and combined metric results for LEDAcrypt-KEM (Table 6.8), where the latency of the key exchange does not consider the key generation action (as it is supposed to be widely amortized over its use), the best, i.e., lowest, computational requirements are obtained picking $n_0 = 2$ or $n_0 = 3$, depending on the NIST category at hand, since the side-effect of increasing the code length, which comes with increasing the code rate, raises the computational load of the decoder, and picking $n_0 > 2$ increases the number of multiplications to be computed during encryption. However, when evaluating the overall efficiency including the amount of transmitted data, following the same metric described before, we obtain that the reduction in the amount of data to be sent, coming from the smaller values of p, which are admissible when $n_0 > 2$, is chosen are enough to offset the additional computational requirements. Indeed, when considering a combined metric, the best choices for LEDAcrypt-KEM are the ones picking $n_0 = 3$ or $n_0 = 4$ depending on the NIST category at hand.

Finally, we provide the computational load required by LEDAcrypt-PKC (Table 6.9), and evaluate the extent of the overhead of computing the PKC with respect to the KEM alone. We observe that, in computing the LEDAcrypt-PKC the overheads imposed by the computation of the Kobara-Imai construction, as opposed to the simpler one required by LEDAcrypt-KEM are however relatively contained ($\approx 30\%$), especially for the LEDAcrypt instances providing higher security margins.

Table 6.7: Performance figures of merit for **LEDAcrypt-KEM-CPA**. Execution times are reported in **clock cycles (cc)**, while values for T_{combined} are reported in **equivalent cc**, being it computed by increasing the overall execution time with $\beta \times \mathtt{N}$, where $\beta = 1000$ and $\mathtt{N}$ is the total number of bytes transmitted over the communication channel (i.e., public key+ciphertext)

NIST Category	n_0	KeyGen (10^3cc)	Encap (10^3cc)	Decap (10^3cc)	KeyGen+Encap+Decap (10^3cc)	T_{combined} (10^3equiv. cc)
1	2	1,241.4 (± 28.2)	109.7 (± 2.2)	784.8 (± 32.2)	2,135.9	4,895.9
	3	999.5 (± 2.8)	99.5 (± 0.8)	879.4 (± 112.8)	1,978.4	5,098.4
	4	852.4 (± 3.0)	90.4 (± 1.8)	972.8 (± 71.5)	1,915.7	5,555.7
3	2	3,285.3 (± 65.5)	266.1 (± 12.2)	2,294.5 (± 84.6)	5,845.9	11,141.9
	3	2,592.0 (± 3.6)	221.0 (± 1.1)	2,799.6 (± 85.5)	5,612.6	11,428.6
	4	1,961.2 (± 6.0)	215.8 (± 1.4)	3,083.4 (± 302.8)	5,260.4	11,852.4
5	2	7,284.3 (± 13.2)	531.3 (± 11.5)	5,640.4 (± 38.0)	13,456.0	22,344.0
	3	3,761.0 (± 84.1)	494.8 (± 18.4)	6,038.5 (± 237.2)	10,294.2	19,958.2
	4	4,140.4 (± 8.1)	474.9 (± 2.1)	6,895.8 (± 541.2)	11,511.1	22,367.1

Table 6.8: Performance figures of merit for the IND-CCA2 **LEDAcrypt-KEM** instances. Execution times are reported in **clock cycles (cc)**, while values for T_{combined} are reported in **equivalent cc**, being it computed by increasing the overall execution time with $\beta \times \mathtt{N}$, where $\beta = 1000$ and $\mathtt{N}$ is the total number of bytes transmitted over the communication channel (i.e., ciphertext)

NIST Category	DFR	n_0	KeyGen (10^3cc)	Encap (10^3cc)	Decap (10^3cc)	Encap+Decap (10^3cc)	T_{combined} (10^3cc)
1	2^{-64}	2	4,399.4 (± 232.6)	240.0 (± 1.0)	1,788.6 (± 4.4)	2,028.6	4,980.6
		3	4,062.0 (± 389.4)	277.5 (± 0.9)	2,076.0 (± 4.0)	2,353.5	4,393.5
		4	3,198.8 (± 331.1)	241.0 (± 1.1)	2,174.1 (± 3.7)	2,415.1	4,119.1
	2^{-128}	2	7,336.7 (± 613.7)	349.3 (± 1.5)	2,995.2 (± 7.5)	3,344.5	6,904.5
		3	6,130.5 (± 1,815.8)	312.5 (± 1.1)	2,788.0 (± 11.6)	3,100.5	5,588.5
		4	5,793.7 (± 1,627.9)	386.0 (± 1.6)	2,935.1 (± 3.6)	3,321.2	5,377.2
3	2^{-64}	2	11,057.1 (± 526.5)	617.2 (± 2.5)	5,358.7 (± 11.9)	5,975.9	11,111.9
		3	8,680.2 (± 228.6)	610.1 (± 6.8)	5,400.3 (± 26.1)	6,010.4	9,594.4
		4	8,680.1 (± 3,143.8)	523.1 (± 7.0)	6,197.7 (± 16.3)	6,720.8	9,616.8
	2^{-192}	2	15,386.1 (± 1,305.7)	789.8 (± 1.8)	7,209.6 (± 26.0)	7,999.3	14,615.3
		3	11,495.2 (± 1,189.1)	874.9 (± 2.1)	8,535.6 (± 26.5)	9,410.6	14,026.6
		4	11,820.7 (± 2,125.3)	1,019.8 (± 2.0)	9,516.8 (± 37.5)	10,536.6	14,424.6
5	2^{-64}	2	23,304.5 (± 1,848.6)	1,374.2 (± 2.7)	12,592.4 (± 15.3)	13,966.6	21,726.6
		3	14,044.0 (± 713.0)	1,197.9 (± 7.2)	11,697.4 (± 43.0)	12,895.3	18,271.3
		4	13,005.2 (± 651.7)	1,314.9 (± 3.1)	15,291.3 (± 25.0)	16,606.2	21,086.2
	2^{-256}	2	34,592.0 (± 3,150.4)	1,919.3 (± 3.5)	15,640.7 (± 22.7)	17,560.0	28,048.0
		3	23,792.9 (± 2,824.9)	1,872.9 (± 3.4)	18,299.1 (± 30.9)	20,171.9	27,483.9
		4	18,513.5 (± 3,975.2)	1,602.4 (± 5.1)	18,780.1 (± 73.9)	20,382.5	26,470.5

Table 6.9: Execution times in **clock cycles (cc)** for the IND-CCA2 **LEDAcrypt-PKC** instances

NIST Category	DFR	n_0	KeyGen (10^3cc)	Encrypt (10^3cc)	Decrypt (10^3cc)	Encrypt+Decrypt (10^3cc)	Overhead w.r.t. KEM (10^3cc)
1	2^{-64}	2	4,208.6 (± 229.5)	682.0 (± 1.4)	2,185.9 (± 10.4)	2,867.9	839.3
		3	4,151.9 (± 479.7)	841.8 (± 2.5)	2,743.5 (± 3.1)	3,585.3	1,231.8
		4	3,139.8 (± 7.4)	961.0 (± 7.0)	2,938.0 (± 4.8)	3,899.0	1,483.9
	2^{-128}	2	7,096.2 (± 385.2)	868.9 (± 21.2)	3,457.5 (± 6.2)	4,326.4	981.9
		3	6,195.1 (± 1743.3)	1,123.6 (± 99.0)	3,433.8 (± 34.1)	4,557.4	1,456.9
		4	5,577.2 (± 978.0)	1,376.9 (± 18.1)	4,080.7 (± 56.2)	5,457.6	2,136.4
3	2^{-64}	2	10,851.4 (± 185.1)	1,429.2 (± 2.2)	6,094.2 (± 7.2)	7,523.4	1,547.5
		3	8,700.7 (± 9.1)	1,537.7 (± 5.8)	6,533.8 (± 9.2)	8,071.5	2,061.1
		4	9,402.6 (± 4,076.9)	1,618.8 (± 2.8)	7,888.2 (± 11.4)	9,507.0	2,786.2
	2^{-192}	2	15,703.4 (± 1,829.2)	1,670.7 (± 15.7)	8,033.4 (± 59.2)	9,704.0	1,704.7
		3	11,437.9 (± 1,038.8)	2,157.6 (± 3.1)	9,192.3 (± 39.7)	11,350.0	1,939.4
		4	12,431.3 (± 2,651.9)	2,575.9 (± 47.1)	11,455.6 (± 85.0)	14,031.6	3,495
5	2^{-64}	2	23,694.9 (± 2,020.4)	2,551.5 (± 3.5)	12,551.0 (± 71.4)	15,102.5	1,135.9
		3	13,960.0 (± 466.0)	2,622.0 (± 5.6)	13,458.3 (± 23.1)	16,080.3	3,185
		4	13,063.9 (± 738.6)	2,954.1 (± 6.7)	17,159.3 (± 32.5)	20,113.4	3,507.2
	2^{-256}	2	33,877.3 (± 2,185.3)	3,218.6 (± 3.6)	17,255.9 (± 27.1)	20,474.5	2,914.5
		3	23,450.0 (± 2,156.5)	3,657.6 (± 78.1)	19,680.5 (± 216.5)	23,338.1	3,166.2
		4	19,106.7 (± 3,817.8)	3,854.1 (± 5.6)	22,500.1 (± 36.1)	26,354.2	5,971.7

Table 6.10: **LEDAcrypt-KEM-CPA recommended parameter choices**, for Category 1, 3, 5, respectively (extracted from Table 4.4, Table 6.1, and Table 6.7)

NIST Category	Figure of Merit	Value of Figure of Merit	n_0	p	v	t
1	Transmitted data	2,760 × (B)	2	10,883	71	133
	Computation Speed	1,915.7 × (10^3 clock cycles)	4	7,187	83	67
	Combined Metric	4,895.9 × (10^3 equiv.clock cycles)	2	10,883	71	133
3	Transmitted data	5,296 × (B)	2	21,011	103	198
	Computation Speed	5,260.4 × (10^3 clock cycles)	4	13,109	123	99
	Combined Metric	11,141.9 × (10^3 equiv.clock cycles)	2	21,011	103	198
5	Transmitted data	8,888 × (B)	2	35,339	137	263
	Computation Speed	10,294.2 × (10^3 clock cycles)	3	25,603	155	166
	Combined Metric	19,958.2 × (10^3 equiv.clock cycles)	3	25,603	155	166

6.4 Recommended parameters

In this section we report our recommendation on the choices for the QC-MDPC code configurations of all the LEDAcrypt cryptosystems. In doing this, we consider three metrics: the amount of transmitted data, the computation time required and the combined metric made by the number of clock cycles taken by the computation plus the number of transmitted bytes multiplied by 1000.

We do not directly take as an optimization metric the size of the public key alone: our motivation to do so is twofold. First of all, in the scenarios where the public key needs to be communicated often, such as the ephemeral key use in LEDAcrypt-KEM-CPA, its size is already taken into account in the communicated data size. Second, in all three LEDAcrypt systems (LEDAcrypt-KEM, LEDAcrypt-PKC, and LEDAcrypt-KEM-CPA) the concerns on the public key size regarding the impossibility to fit in practical use constraints are fulfilled for all the public key sizes we propose, as we analyzed in Section 6.1. Indeed, our public key sizes are always smaller than the $2^{16} - 1$ bound mandated by the public key size field in TLS 1.3, and fit rather comfortably even in the Flash memory of modern (e.g., ARM Cortex-M based), low end microcontrollers.

6.4.1 Recommended parameters for LEDAcrypt-KEM-CPA

Table 6.10 reports our recommendation on the optimal choices for the QC-MDPC code configurations of the LEDAcrypt-KEM-CPA instances. If the transmitted data size alone is preferred, the choice of $n_0 = 2$ fits all the three NIST categories, while if the computation speed alone is taken into account, picking $n_0 > 2$ is always the most efficient choice. An analysis of the combined metric, allows us to observe that picking $n_0 = 2$ for NIST category 1 and 3, and $n_0 = 3$ for NIST category 3 yields the best results when balancing computation and transmitted data. Indeed, if the transmission of both the public key and the ciphertext of the LEDAcrypt-KEM-CPA is required, as it is in an ephemeral key use scenario, the increase in the transmitted data is compensated in a combined metric by the reduction of the prime size only for NIST category 5.

Table 6.11: **LEDAcrypt-KEM and LEDAcrypt-PKC recommended parameter choices for IND-CCA2 guarantees**, i.e. providing DFR= $2^{-\lambda}, \lambda \in \{128, 192, 256\}$, for Category 1, 3, 5, respectively (extracted from Table 4.1, Table 6.2, and Table 6.8)

NIST Category	Figure of Merit	Value of Figure of Merit	$\mathbf{n_0}$	p	v	t
1	Transmitted data	$2,056 \times$ (B)	4	16,229	83	65
	Computation Speed	$3,100.5 \times$ (10^3 clock cycles)	3	19,709	79	82
	Combined Metric	$5,377.2 \times$ (10^3 equiv.clock cycles)	4	16,229	83	65
3	Transmitted data	$3,888 \times$ (B)	4	30,803	123	98
	Computation Speed	$7,999.3 \times$ (10^3 clock cycles)	2	52,667	103	195
	Combined Metric	14,026.6 $\times$ (10^3 equiv.clock cycles)	3	36,629	115	123
5	Transmitted data	$6,088 \times$ (B)	4	48,371	161	131
	Computation Speed	17,560.0 $\times$ (10^3 clock cycles)	2	83,579	135	260
	Combined Metric	26,470.5 $\times$ (10^3 equiv.clock cycles)	4	48,371	161	131

6.4.2 Recommended parameters for the IND-CCA2 LEDAcrypt-KEM and LEDAcrypt-PKC systems

Table 6.11 reports our recommendation on the optimal choices for the QC-MDPC code configurations of the IND-CCA2 version of the LEDAcrypt-KEM, and, since the parameter choice acts only on the asymmetric components of the PKC system, also for the code configurations of the LEDAcrypt-PKC instances.

Examining the results, we have that our recommendation is to pick either $n_0 = 4$, or $n_0 = 3$ in all cases where the transmitted data or the combined computation-transmission metric is to be optimized. Indeed, the reduction in transmitted data which is achievable raising the code rate $(\frac{n_0-1}{n_0})$ is enough to compensate the increase in computational power required during encryption and decryption. Contrariwise, if computation speed alone is the criterion for optimization, $n_0 = 3$ should be chosen for NIST category 1 parameters, and $n_0 = 2$ for NIST category 3 and 5 parameters.

Finally, in our recommended parameter choices, we point out also a parameter selection when a DFR= 2^{-64} only is guaranteed. These parameter sets, while not matching the formal requirement mandated by [39] to attain IND-CCA2 security assurances, provide a practical data-point for scenarios where a concrete DFR= 2^{-64} is not likely to provide to an attacker the opportunity of observing a decoding failure. This includes, for instance, low-end embedded devices, which are unlikely to exceed 2^{24} encryptions during their entire lifetime (e.g., in smartcards the number of encryptions is typically capped via a hardware counter), thus bringing the probability of an attacker observing a decoding failure to be practically negligible.

In this scenario, the summary of the results, reported in Table 6.12, recommends to pick $n_0 = 4$, if minimizing the transmitted data is the end goal, $n_0 = 3$ or $n_0 = 4$ depending on the NIST category if the combined metric is to be optimized, and $n_0 = 2$ or $n_0 = 3$ if the computation speed is the only aim of the optimization.

Table 6.12: **LEDAcrypt-KEM and LEDAcrypt-PKC recommended parameter choices to provide DFR= 2^{-64} (also the number of ciphertexts available to a CCA opponent is** $\leq 2^{64}$), for Category 1, 3, 5, respectively (extracted from Table 4.1, Table 6.2, and Table 6.8)

NIST Category	Figure of Merit	Value of Figure of Merit	n_0	p	v	t
1	Transmitted data	$1,704 \times$ (B)	4	13,397	83	66
	Computation Speed	$2,028.6 \times$ (10^3 clock cycles)	2	23,371	71	130
	Combined Metric	$4,119.1 \times$ (10^3 equiv.clock cycles)	4	13,397	83	66
3	Transmitted data	$2,896 \times$ (B)	4	22,901	123	98
	Computation Speed	$5,975.9 \times$ (10^3 clock cycles)	2	40,787	103	195
	Combined Metric	$9,594.4 \times$ (10^3 equiv.clock cycles)	3	28,411	117	124
5	Transmitted data	$4,480 \times$ (B)	4	35,507	163	131
	Computation Speed	12,895.3 $\times$ (10^3 clock cycles)	3	42,677	153	165
	Combined Metric	18,271.3 $\times$ (10^3 equiv.clock cycles)	3	42,677	153	165

Bibliography

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, D. Smith-Tone, and Y.-K. Liu, "Status report on the first round of the NIST post-quantum cryptography standardization process," National Institute of Standards and Technology, Tech. Rep. NISTIR 8240, Jan. 2019.

[2] D. Apon, R. Perlner, A. Robinson, and P. Santini, "Cryptanalysis of ledacrypt," Cryptology ePrint Archive, Report 2020/455, 2020, https://eprint.iacr.org/2020/455.

[3] S. Arora and B. Barak, *Computational Complexity - A Modern Approach*. Cambridge University Press, 2009. [Online]. Available: http://www.cambridge.org/catalogue/catalogue.asp?isbn=9780521424264

[4] M. Baldi and F. Chiaraluce, "Cryptanalysis of a new instance of McEliece cryptosystem based on QC-LDPC codes," in *Proc. IEEE International Symposium on Information Theory (ISIT 2007)*, Nice, France, Jun. 2007, pp. 2591–2595.

[5] M. Baldi, *QC-LDPC Code-Based Cryptography*, ser. SpringerBriefs in Electrical and Computer Engineering. Springer International Publishing, 2014.

[6] M. Baldi, A. Barenghi, F. Chiaraluce, G. Pelosi, and P. Santini, "LEDAcrypt-KEM and LEDAcrypt-PKC website," https://www.ledacrypt.org/.

[7] A. Barenghi and G. Pelosi, "Constant Weight Strings in Constant Time: a Building Block for Code-based Post-quantum Cryptosystems," in *Proceedings of the 17th ACM International Conference on Computing Frontiers (CF'20), Catania, Sicily, Italy, May 11 - 13, 2020*, M. Palesi, G. Palermo, C. Graves, and E. Arima, Eds. New York, NY, USA: ACM, May 2020, pp. 1–6, iSBN: 978-1-4503-7956-4/20/05. [Online]. Available: http://dx.doi.org/10.1145/3387902.3392630

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," U.S.A. National Institute of Standards and Technology (NIST) Special Publication 800-90A (NIST SP 800-90A). Revision 1. Available online at: http://dx.doi.org/10.6028/NIST.SP.800-90Ar1, 2015.

[9] A. Becker, A. Joux, A. May, and A. Meurer, "Decoding random binary linear codes in $2^n/20$: How $1 + 1 = 0$ improves information set decoding," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, 2012, pp. 520–536.

[10] M. Bellare, H. Davis, and F. Günther, "Separate Your Domains: NIST PQC KEMs, Oracle Cloning and Read-Only Indifferentiability," Cryptology ePrint Archive, Report 2020/241, 2020, https://eprint.iacr.org/2020/241.

[11] T. P. Berger, P. Cayrel, P. Gaborit, and A. Otmani, "Reducing key length of the McEliece cryptosystem," in *Progress in Cryptology - AFRICACRYPT 2009, Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings*, ser. Lecture Notes in Computer Science, B. Preneel, Ed., vol. 5580. Springer, 2009, pp. 77–97. [Online]. Available: https://doi.org/10.1007/978-3-642-02384-2_6

[12] E. Berlekamp, R. McEliece, and H. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Trans. Information Theory*, vol. 24, no. 3, pp. 384–386, May 1978.

[13] D. J. Bernstein, "Grover vs. mceliece," in *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, ser. Lecture Notes in Computer Science, N. Sendrier, Ed., vol. 6061. Springer, 2010, pp. 73–80. [Online]. Available: https://doi.org/10.1007/978-3-642-12929-2_6

[14] D. J. Bernstein and B. Yang, "Fast constant-time gcd computation and modular inversion," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 3, pp. 340–398, 2019. [Online]. Available: https://doi.org/10.13154/tches.v2019.i3.340-398

[15] E. Bernstein and U. V. Vazirani, "Quantum complexity theory," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1411–1473, Oct. 1997.

[16] M. Bodrato, "Towards Optimal Toom-Cook Multiplication for Univariate and Multivariate Polynomials in Characteristic 2 and 0," in *WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, ser. Lecture Notes in Computer Science, C. Carlet and B. Sunar, Eds., vol. 4547. Springer, 2007, pp. 116–133. [Online]. Available: https://doi.org/10.1007/978-3-540-73074-3_10

[17] H. Brunner, A. Curiger, and M. Hofstetter, "On Computing Multiplicative Inverses in GF(2ˆm)," *IEEE Trans. Computers*, vol. 42, no. 8, pp. 1010–1015, 1993.

[18] T. Chou, "Qcbits: Constant-time small-key code-based cryptography," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, ser. Lecture Notes in Computer Science, B. Gierlichs and A. Y. Poschmann, Eds., vol. 9813. Springer, 2016, pp. 280–300. [Online]. Available: https://doi.org/10.1007/978-3-662-53140-2_14

[19] J. W. Chung, S. G. Sim, and P. J. Lee, "Fast Implementation of Elliptic Curve Defined over GF(p^{m}) on CalmRISC with MAC2424 Coprocessor," in *Cryptographic Hardware and Embedded Systems - CHES 2000, Second International Workshop, Worcester, MA, USA, August 17-18, 2000, Proceedings*, ser. Lecture Notes in Computer Science, Ç. K. Koç and C. Paar, Eds., vol. 1965. Springer, 2000, pp. 57–70. [Online]. Available: https://doi.org/10.1007/3-540-44499-8_4

[20] I. Corp., "Intel intrinsics guide," https://software.intel.com/sites/landingpage/IntrinsicsGuide/, 2020.

[21] E. Crockett, C. Paquin, and D. Stebila, "Prototyping post-quantum and hybrid key exchange and authentication in TLS and SSH," Cryptology ePrint Archive, Report 2019/858, 2019, https://eprint.iacr.org/2019/858.

[22] S. de Vries, "Achieving 128-bit security against quantum attacks in OpenVPN," Master's thesis, University of Twente, Aug. 2016. [Online]. Available: http://essay.utwente.nl/70677/

[23] N. Drucker, S. Gueron, and D. Kostic, "QC-MDPC decoders with several shades of gray," *IACR Cryptology ePrint Archive*, vol. 2019, p. 1423, 2019. [Online]. Available: https://eprint.iacr.org/2019/1423

[24] ——, "Fast polynomial inversion for post quantum QC-MDPC cryptography," *IACR Cryptology ePrint Archive*, vol. 2020, p. 298, 2020. [Online]. Available: https://eprint.iacr.org/2020/298

[25] M. J. Dworkin, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," U.S.A. Federal Information Processing Standard. (NIST FIPS). Report number: 202. Available online at: https://doi.org/10.6028/NIST.FIPS.202, 2015.

[26] E. Eaton, M. Lequesne, A. Parent, and N. Sendrier, "QC-MDPC: A timing attack and a CCA2 KEM," in *PQCrypto*, T. Lange and R. Steinwandt, Eds. Fort Lauderdale, FL, USA: Springer International Publishing, Apr. 2018, pp. 47–76.

[27] T. Fabšič, V. Hromada, P. Stankovski, P. Zajac, Q. Guo, and T. Johansson, "A reaction attack on the QC-LDPC McEliece cryptosystem," in *Post-Quantum Cryptography: 8th International Workshop, PQCrypto 2017*, T. Lange and T. Takagi, Eds. Utrecht, The Netherlands: Springer International Publishing, Jun. 2017, pp. 51–68.

[28] T. Fabšič, V. Hromada, and P. Zajac, "A reaction attack on LEDApkc," Cryptology ePrint Archive, Report 2018/140, 2018, https://eprint.iacr.org/2018/140.

[29] M. Finiasz and N. Sendrier, "Security bounds for the design of code-based cryptosystems," in *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, 2009, pp. 88–105.

[30] E. Fujisaki and T. Okamoto, "Secure Integration of Asymmetric and Symmetric Encryption Schemes," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, ser. Lecture Notes in Computer Science, M. J. Wiener, Ed., vol. 1666. Springer, 1999, pp. 537–554. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_34

[31] ——, "Secure integration of asymmetric and symmetric encryption schemes," *J. Cryptology*, vol. 26, no. 1, pp. 80–101, 2013. [Online]. Available: https://doi.org/10.1007/s00145-011-9114-1

[32] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21–28, Jan. 1962.

[33] ——, *Low-Density Parity-Check Codes*. M.I.T. Press, 1963.

[34] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt, "Applying Grover's algorithm to AES: quantum resource estimates," in *Post-Quantum Cryptography - 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings*, 2016, pp. 29–43.

[35] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. 28th Annual ACM Symposium on the Theory of Computing*, Philadephia, PA, May 1996, pp. 212–219.

[36] A. Guimarães, D. de Freitas Aranha, and E. Borin, "Optimized implementation of QC-MDPC code-based cryptography," *Concurrency and Computation: Practice and Experience*, vol. 31, no. 18, 2019. [Online]. Available: https://doi.org/10.1002/cpe.5089

[37] Q. Guo, T. Johansson, and P. Stankovski Wagner, "A key recovery reaction attack on QC-MDPC," *IEEE Trans. Information Theory*, vol. 65, no. 3, pp. 1845–1861, Mar. 2019.

[38] Q. Guo, T. Johansson, and P. Stankovski, "A key recovery attack on MDPC with CCA security using decoding errors," in *Advances in Cryptology { ASIACRYPT 2016*, ser. Lecture Notes in

Computer Science, J. H. Cheon and T. Takagi, Eds. Springer Berlin Heidelberg, 2016, vol. 10031, pp. 789–815.

[39] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation," in *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, ser. Lecture Notes in Computer Science, Y. Kalai and L. Reyzin, Eds., vol. 10677. Springer, 2017, pp. 341–371. [Online]. Available: https://doi.org/10.1007/978-3-319-70500-2_12

[40] S. Jaques, M. Naehrig, M. Roetteler, and F. Virdia, "Implementing Grover oracles for quantum key search on AES and LowMC," *CoRR*, vol. abs/1910.01700, 2019. [Online]. Available: http://arxiv.org/abs/1910.01700

[41] H. Jiang, Z. Zhang, L. Chen, H. Wang, and Z. Ma, "IND-CCA-secure key encapsulation mechanism in the quantum random oracle model, revisited," in *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part III*, ser. Lecture Notes in Computer Science, H. Shacham and A. Boldyreva, Eds., vol. 10993. Springer, 2018, pp. 96–125. [Online]. Available: https://doi.org/10.1007/978-3-319-96878-0_4

[42] H. Jiang, Z. Zhang, and Z. Ma, "Tighter security proofs for generic key encapsulation mechanism in the quantum random oracle model," in *Post-Quantum Cryptography*, J. Ding and R. Steinwandt, Eds. Cham: Springer International Publishing, 2019, pp. 227–248.

[43] G. Kachigar and J. Tillich, "Quantum information set decoding algorithms," in *Post-Quantum Cryptography - 8th International Workshop, PQCrypto 2017, Utrecht, The Netherlands, June 26-28, 2017, Proceedings*, 2017, pp. 69–89.

[44] D. E. Knuth, *The Art of Computer Programming: Generating All Combinations and Partitions*, 1st ed. Addison-Wesley, 2005, vol. 4.

[45] K. Kobara and H. Imai, "Semantically secure McEliece public-key cryptosystems — conversions for McEliece PKC," *Lecture Notes in Computer Science*, vol. 1992, pp. 19–35, 2001. [Online]. Available: citeseer.ist.psu.edu/kobara01semantically.html

[46] K. Kobayashi, N. Takagi, and K. Takagi, "Fast inversion algorithm in GF(2^m) suitable for implementation with a polynomial multiply instruction on GF(2)," *IET Computers & Digital Techniques*, vol. 6, no. 3, pp. 180–185, 2012. [Online]. Available: https://doi.org/10.1049/iet-cdt.2010.0006

[47] P. J. Lee and E. F. Brickell, "An observation on the security of McEliece's public-key cryptosystem," in *Advances in Cryptology - EUROCRYPT '88, Workshop on the Theory and Application of of Cryptographic Techniques, Davos, Switzerland, May 25-27, 1988, Proceedings*, 1988, pp. 275–280.

[48] J. S. Leon, "A probabilistic algorithm for computing minimum weights of large error-correcting codes," *IEEE Trans. Information Theory*, vol. 34, no. 5, pp. 1354–1359, Sep. 1988.

[49] Y. X. Li, R. Deng, and X. M. Wang, "On the equivalence of McEliece's and Niederreiter's public-key cryptosystems," *IEEE Trans. Information Theory*, vol. 40, no. 1, pp. 271–273, Jan. 1994.

[50] M. Luby, M. Mitzenmacher, M. Shokrollahi, and D. Spielman, "Improved low-density parity-check codes using irregular graphs," *IEEE Trans. Information Theory*, vol. 47, no. 2, pp. 585–598, Feb. 2001.

[51] A. May, A. Meurer, and E. Thomae, "Decoding random linear codes in $\tilde{O}(2^{0.054n})$," in *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, 2011, pp. 107–124.

[52] R. J. McEliece, "A public-key cryptosystem based on algebraic coding theory." *DSN Progress Report*, pp. 114–116, 1978.

[53] National Institute of Standards and Technology. (2016, Dec.) Post-quantum crypto project. [Online]. Available: http://csrc.nist.gov/groups/ST/post-quantum-crypto/

[54] ——. (2018) Post-quantum cryptography - round 1 submissions. [Online]. Available: https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-1-Submissions

[55] H. Niederreiter, "Knapsack-type cryptosystems and algebraic coding theory," *Probl. Contr. and Inform. Theory*, vol. 15, pp. 159–166, 1986.

[56] A. Nilsson, T. Johansson, and P. Stankovski Wagner, "Error amplification in code-based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2019, no. 1, pp. 238–258, Nov. 2018.

[57] R. Perlner, "Attack on LEDA's product structure," private communication, Oct. 2019.

[58] E. Prange, "The use of information sets in decoding cyclic codes," *IRE Trans. Information Theory*, vol. 8, no. 5, pp. 5–9, Sep. 1962.

[59] M. Rossi, M. Hamburg, M. Hutter, and M. E. Marson, "A Side-Channel Assisted Cryptanalytic Attack Against QcBits," in *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, ser. Lecture Notes in Computer Science, W. Fischer and N. Homma, Eds., vol. 10529. Springer, 2017, pp. 3–23. [Online]. Available: https://doi.org/10.1007/978-3-319-66787-4_1

[60] A. Salomaa, "Chapter II - Finite Non-deterministic and Probabilistic Automata," in *Theory of Automata*, ser. International Series of Monographs on Pure and Applied Mathematics, A. Salomaa, Ed. Pergamon, 1969, vol. 100, pp. 71 – 113.

[61] P. Santini, M. Battaglioni, M. Baldi, and F. Chiaraluce, "Hard-decision iterative decoding of ldpc codes with bounded error rate," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, May 2019, pp. 1–6.

[62] P. Santini, M. Baldi, and F. Chiaraluce, "Assessing and countering reaction attacks against post-quantum public-key cryptosystems based on qc-ldpc codes," in *Cryptology and Network Security*, J. Camenisch and P. Papadimitratos, Eds. Cham: Springer International Publishing, 2018, pp. 323–343.

[63] P. Santini, M. Battaglioni, F. Chiaraluce, and M. Baldi, "Analysis of reaction and timing attacks against cryptosystems based on sparse parity-check codes," in *Code-Based Cryptography*, M. Baldi, E. Persichetti, and P. Santini, Eds. Cham: Springer International Publishing, 2019, pp. 115–136.

[64] Sean Eron Anderson, "Bit Twiddling Hacks," https://graphics.stanford.edu/~seander/bithacks.html, last accessed 7 April 2020, 2020.

[65] N. Sendrier, "Decoding one out of many," in *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29 - December 2, 2011. Proceedings*, 2011, pp. 51–67.

[66] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, Oct. 1997.

[67] V. Sidelnikov and S. Shestakov, "On cryptosystems based on generalized Reed-Solomon codes," *Diskretnaya Math*, vol. 4, pp. 57–63, 1992.

[68] B. Sim, J. Kwon, K. Y. Choi, J. Cho, A. Park, and D. Han, "Novel Side-Channel Attacks on Quasi-Cyclic Code-Based Cryptography," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2019, no. 4, pp. 180–212, 2019. [Online]. Available: https://doi.org/10.13154/tches.v2019.i4.180-212

[69] J. Stern, "A method for finding codewords of small weight," in *Coding Theory and Applications, 3rd International Colloquium, Toulon, France, November 2-4, 1988, Proceedings*, 1988, pp. 106–113.

[70] C. Su and H. Fan, "Impact of Intel's new instruction sets on software implementation of GF(2)[x] multiplication," *Inf. Process. Lett.*, vol. 112, no. 12, pp. 497–502, 2012. [Online]. Available: https://doi.org/10.1016/j.ipl.2012.03.012

[71] J. Tillich, "The decoding failure probability of MDPC codes," in *2018 IEEE International Symposium on Information Theory, ISIT 2018, Vail, CO, USA, June 17-22, 2018*, 2018, pp. 941–945.

[72] R. Townsend and E. J. Weldon, "Self-orthogonal quasi-cyclic codes," *IEEE Trans. Information Theory*, vol. 13, no. 2, pp. 183–195, Apr. 1967.

[73] R. Ueno, S. Morioka, N. Homma, and T. Aoki, "A high throughput/gate AES hardware architecture by compressing encryption and decryption datapaths - Toward efficient CBC-mode implementation," in *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, 2016, pp. 538–558.

[74] A. Vardy, "The intractability of computing the minimum distance of a code," *IEEE Trans. Information Theory*, vol. 43, no. 6, pp. 1757–1766, 1997. [Online]. Available: https://doi.org/10.1109/18.641542

[75] C. Xing and S. Ling, *Coding Theory: A First Course*. New York, NY, USA: Cambridge University Press, 2003.

Appendix A

Deriving the bit-flipping probabilities for the in-place iteration function of the LEDAdecoder

We consider one call of RandomizedInPlaceIteration, with input error vector estimate $\hat{e}$ and actual error vector e. We denote with E_0 the set of positions in which $\hat{e}$ and e match, and with E_1 the set of positions in which they differ. We define $\hat{t}_0 = |\{\text{Supp}(e \oplus \hat{e}) \cap E_0\}|$, and $\hat{t}_1 = |\{\text{Supp}(e \oplus \hat{e}) \cap E_1\}|$; clearly, $\hat{t} = wt\,(e \oplus \hat{e}) = \hat{t}_0 + \hat{t}_1$.

We now characterize the statistical distribution of $\hat{t}_0$ and $\hat{t}_1$ after n bit estimates, processed in the order pointed out by $\pi^* \in S^*_n$, i.e., the permutation which places at the end all the positions j where $\hat{e}_j \neq e_j$.

We characterize distribution of $\hat{t}_0$ during the computation of the iteration as a function of $\mathsf{P}^{\mathsf{th}}_{f|0}(\hat{t})$ and $\mathsf{P}^{\mathsf{th}}_{m|0}(\hat{t})$, i.e. the probability that an error estimate bit will be flipped or maintained (Assumption 3.1.1).

To model the statistical distribution of $\hat{t}_0$ we employ the framework of Probabilistic Finite State Automata (PFSA) [60]. Informally, a PFSA is a Finite State Automaton (FSA) characterized by transition probabilities for each of the transitions of the FSA. The state of a PFSA is a discrete probability distribution over the set of FSA states and the probabilities of the transitions starting from the same FSA state, reading a the same symbol, must add up to one.

We model the evolution of the statistical distribution of $\hat{t}_0$ during the iteration computation as the state of a PFSA having $n - \hat{t}$ FSA states, each one mapped onto a specific value for $\hat{t}_0$, as depicted in Figure A.1. We consider the underlying FSA to be accepting the input language constituted by binary strings obtained as the sequences $\pi^*(\hat{t} \oplus e)$.

We therefore have that, for the PFSA modeling the evolution of $\hat{t}_0$ while the R-IP BF decoder acts on the first $n - \hat{t}$ positions specified by π^*, all the read bits will be equal to 0, as π^* sorts the positions of $\hat{t}$ so that the ($n - \hat{t}$ at the first iteration) positions with no discrepancy between $\hat{e}$ and e come first.

The transition probability for the PFSA transition from a state $\hat{t}_0 = i$ to $\hat{t}_0 = i + 1$ requires the

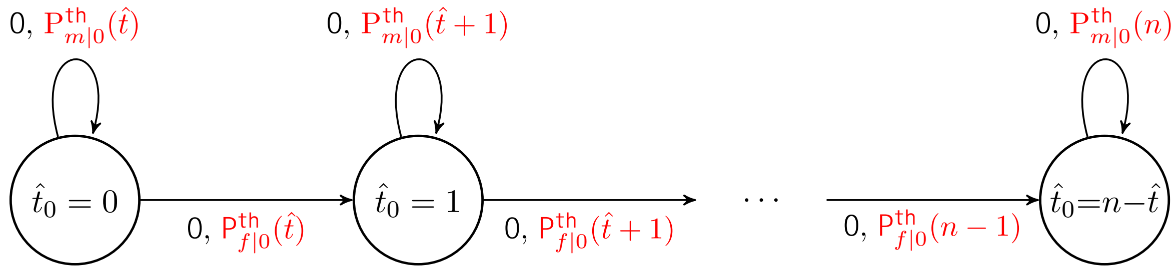


Figure A.1: Structure of the probabilistic FSA modeling the evolution of the distribution of the $\hat{t}_0$ variable. Read characters are reported in black, transition probabilities in red.

RIP decoder to flip a bit of $\hat{e}$ equal to zero and matching the one in the same position of e, causing a discrepancy. Because of Assumption 3.1.1, the probability of such an event is $\mathsf{P}^{\mathsf{th}}_{f|0}(\hat{t}+i)$, while the probability of not altering the bit, i.e., the self-loop transition from $\hat{t}_0 = i$ to $\hat{t}_0 = i$ itself, is $\mathsf{P}^{\mathsf{th}}_{m|0}(\hat{t}+i)$.

Note that, during the inner loop iterations of the R-IP BF-decoder acting on positions of $\hat{t}$ which have no discrepancies it is not possible to decrease the value $\hat{t}_0$, as no reduction on the number of discrepancies between $\hat{t}$ and e can be done changing values of $\hat{t}$ which are already equal to the ones in e. Hence, the probability of transitioning from $\hat{t}_0 = i$ to $\hat{t}_0 = i-1$ is zero.

The evolution of a PFSA can be computed simply taking the current state, represented as the vector y of $n-\hat{t}$ elements, with the i-th element containing $\Pr\left[\hat{t}_0 = i\right]$ and multiplying it by an appropriate matrix which characterizes the transitions in the PFSA. Such a matrix is derived as the adjacency matrix of the PFSA graph representation, keeping only the edges for which the read character matches the edge label, and substituting the one-values in the adjacency matrix with the probability labelling the corresponding edge.

We obtain the transition matrix for the PFSA in Figure A.1 as the $(n-\hat{t}+1)\times(n-\hat{t}+1)$ matrix:

$$K_0 = \begin{bmatrix} \mathsf{P}^{\mathsf{th}}_{m|0}(\hat{t}) & \mathsf{P}^{\mathsf{th}}_{f|0}(\hat{t}) & 0 & 0 & 0 & 0 \\ 0 & \mathsf{P}^{\mathsf{th}}_{m|0}(\hat{t}+1) & \mathsf{P}^{\mathsf{th}}_{f|0}(\hat{t}+1) & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \mathsf{P}^{\mathsf{th}}_{m|0}(n-1) & \mathsf{P}^{\mathsf{th}}_{f|0}(n-1) \\ 0 & 0 & 0 & 0 & 0 & \mathsf{P}^{\mathsf{th}}_{m|0}(n) \end{bmatrix}$$

Since we want to compute the effect on the distribution of $\hat{t}_0$ after the first $n-\hat{t}$ rounds of the loop in the randomized in place iteration function, we obtain it as $yK_0^{n-\hat{t}}$. Note that the subsequent t iterations of the RIP decoder will not alter the value of $\hat{t}_0$ as they act on positions j such that $e_j = 1$. Since we know that, at the beginning of the first iteration $y = [\Pr\left[\hat{t}_0 = 0\right] = 1, \Pr\left[\hat{t}_0 = 1\right] = 0, \Pr\left[\hat{t}_0 = 2\right] = 0, \cdots, \Pr\left[\hat{t}_0 = n-\hat{t}\right] = 0]$, we are able to compute $\Pr_{S_n^*}\left[\omega \xrightarrow{E_0} x\right]$ as the $(x+1)$-th element of $yK_0^{n-\hat{t}}$.

We now model the distribution of $\hat{t}_1$, during the last $\hat{t}$ rounds of the loop in the randomized in place iteration function. Note that, to this end, the first $n-\hat{t}$ iterations of the inner loop have no effect on $\hat{t}_1$. Denote with $\tilde{t}$ the incorrectly estimated bits in $wt\,(\tilde{e})$, which corresponds to $wt\,(e \oplus \hat{e})$ when the iteration function is about to evaluate the first of the positions j where $\hat{e}_j \neq e_j$. Note

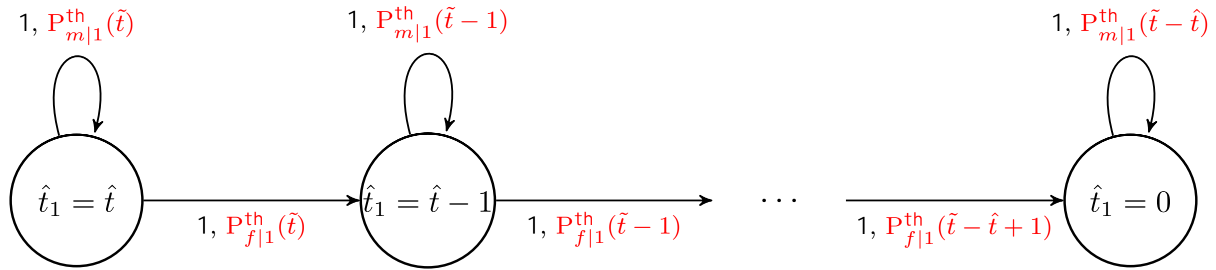


Figure A.2: Structure of the probabilistic FSA modeling the evolution of the distribution of the $\hat{t}_1$ variable. Read characters are reported in black, transition probabilites in red-

that, at the beginning of the iteration function computation, we have $\tilde{t} = \tilde{t}_0 + \hat{t}_1$, where $\tilde{t}_0$ is the number of discrepancies in the first $n - \hat{t}$ positions when the iteration is about to analyze the first position of $\hat{e}$ for which $wt\,(e \oplus \hat{e})$. Following arguments analogous to the ones employed to model the PFSA describing the evolution for $\hat{t}_0$, we obtain the one modeling the evolution of $\hat{t}_1$, reported in Figure A.2. The initial distribution of the values of $\hat{t}_1$, constituting the initial state of the PFSA in Figure A.2 is such that $\Pr\left[\hat{t}_1 = \hat{t}\right] = 1$, corresponding to the $\hat{t}_1$. element vector $z = [0, 0, \ldots, 0, 1]$. Computing the transition matrix of the PFSA in Figure A.2 we obtain:

$$K_1 = \begin{bmatrix} \mathrm{P}^{\mathsf{th}}_{m|1}(\tilde{t}-\hat{t}) & 0 & 0 & 0 & 0 & 0 \\ \mathrm{P}^{\mathsf{th}}_{f|1}(\tilde{t}-\hat{t}+1) & \mathrm{P}^{\mathsf{th}}_{m|1}(\tilde{t}-\hat{t}+1) & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \mathrm{P}^{\mathsf{th}}_{f|1}(\tilde{t}-1) & \mathrm{P}^{\mathsf{th}}_{m|1}(\tilde{t}-1) & 0 \\ 0 & 0 & 0 & 0 & \mathrm{P}^{\mathsf{th}}_{f|1}(\tilde{t}) & \mathrm{P}^{\mathsf{th}}_{m|1}(\tilde{t}) \end{bmatrix}$$

We are thus able to obtain the $\Pr_{S_n^*}\left[\omega \xrightarrow{E_1} x\right]$ computing $zK_1^{\hat{t}}$ and taking the $(x+1)$-th element of it.

Statements

Statement by Each Submitter

I, Marco Baldi, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt.

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive nancial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's speci cations (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identi ed to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

Statement by Reference/Optimized Implementations' Owner(s)

I, Marco Baldi, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, am one of the owners of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

Statement by Each Submitter

I, Alessandro Barenghi, of Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via G. Ponzio 34/5, I-20133, Milano, Italy, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt.

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive financial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's specifications (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identified to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

Statement by Reference/Optimized Implementations' Owner(s)

I, Alessandro Barenghi, of Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via G. Ponzio 34/5, I-20133, Milano, Italy, am one of the owners of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

Statement by Each Submitter

I, Franco Chiaraluce, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt.

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive nancial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's speci cations (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identi ed to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

Statement by Reference/Optimized Implementations' Owner(s)

I, Franco Chiaraluce, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, am one of the owners of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

Statement by Each Submitter

I, Gerardo Pelosi, of Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via G. Ponzio 34/5, I-20133, Milano, Italy, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt.

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive nancial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's speci cations (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identi ed to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

Statement by Reference/Optimized Implementations' Owner(s)

I, Gerardo Pelosi, of Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB), Via G. Ponzio 34/5, I-20133, Milano, Italy, am one of the owners of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place:

Statement by Each Submitter

I, Paolo Santini, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, do hereby declare that the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt, is my own original work, or if submitted jointly with others, is the original work of the joint submitters. I further declare that I do not hold and do not intend to hold any patent or patent application with a claim which may cover the cryptosystem, reference implementation, or optimized implementations that I have submitted, known as LEDAcrypt.

I do hereby acknowledge and agree that my submitted cryptosystem will be provided to the public for review and will be evaluated by NIST, and that it might not be selected for standardization by NIST. I further acknowledge that I will not receive nancial or other compensation from the U.S. Government for my submission. I certify that, to the best of my knowledge, I have fully disclosed all patents and patent applications which may cover my cryptosystem, reference implementation or optimized implementations. I also acknowledge and agree that the U.S. Government may, during the public review and the evaluation process, and, if my submitted cryptosystem is selected for standardization, during the lifetime of the standard, modify my submitted cryptosystem's speci cations (e.g., to protect against a newly discovered vulnerability).

I acknowledge that NIST will announce any selected cryptosystem(s) and proceed to publish the draft standards for public comment.

I do hereby agree to provide the statements required by Sections 2.D.2 and 2.D.3, below, for any patent or patent application identi ed to cover the practice of my cryptosystem, reference implementation or optimized implementations and the right to use such implementations for the purposes of the public review and evaluation process.

I acknowledge that, during the post-quantum algorithm evaluation process, NIST may remove my cryptosystem from consideration for standardization. If my cryptosystem (or the derived cryptosystem) is removed from consideration for standardization or withdrawn from consideration by all submitter(s) and owner(s), I understand that rights granted and assurances made under Sections 2.D.1, 2.D.2 and 2.D.3, including use rights of the reference and optimized implementations, may be withdrawn by the submitter(s) and owner(s), as appropriate.

Signed:

Title:

Date:

Place:

Statement by Reference/Optimized Implementations' Owner(s)

I, Paolo Santini, of Universita Politecnica delle Marche, Dipartimento di Ingegneria dell' Informazione (DII), Via Brecce Bianche 12, I-60131, Ancona, Italy, am one of the owners of the submitted reference implementation and optimized implementations and hereby grant the U.S. Government and any interested party the right to reproduce, prepare derivative works based upon, distribute copies of, and display such implementations for the purposes of the post-quantum algorithm public review and evaluation process, and implementation if the corresponding cryptosystem is selected for standardization and as a standard, notwithstanding that the implementations may be copyrighted or copyrightable.

Signed:

Title:

Date:

Place: