Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

Secure and Efficient Post-Quantum Cryptosystem Realizations
Constant Weight Encoding and Vectorization in LEDACrypt

Relatore: Prof. Alessandro BARENGHI

Tesi di Laurea di:
Nicole GERVASONI Matr. 878439

Anno Accademico 2018–2019

Sommario

LEDAcrypt (Low-dEnsity parity-check coDe-bAsed cryptographic system) è tra i partecipanti al secondo round nel NIST Post-Quantum Contest. L'esistenza di questo concorso sottolinea come vi sia la necessità di identificare nuovi algoritmi di cifratura in grado di resistere alla futura diffusione di computer quantistici. LEDAcrypt è un crittositema basato su codici composto da due moduli: LEDAkem (Key Encapsulation Module) e LEDApkc (Public-Key Cryptosystem), entrambi costruiti attorno a problemi NP-Completi.

Questo elaborato si prefigge di studiare la funzione di codifica a peso costante (Constant Weight Encoding) utilizzata in LEDAcrypt, la quale gioca un ruolo chiave nel crittosistema sviluppato da Niederreteir, sul quale si basa LEDApkc, e in particolare, di migliorarne l'efficienza attraverso un'analisi della distribuzione dei vettori originati in relazione al numero di failure generate.

Secondo obiettivo di questa dissertazione è l'ottimizzazione della codebase di LEDAcrypt al fine di migliorarne le prestazioni generali attraverso una decisa ottimizzazione degli algoritmi per l'aritmetica polinomiale e l'utilizzo di istruzioni vettoriali per processori ARM.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Cryptography etymology derives from Greek: *kryptos* "hidden" plus *graphein* "to write" and thus refers to the "art of writing in secret characters", but its origin traced back to around 4000 years ago within the Egyptian population. Historically, cryptography has always been a fundamental tool in politics, its use in fact has been predominant in military and diplomatic communications. The diffusion of computers has widen its application from merely government services to the protection of any private data and communication. Nowadays cryptography is the study of mathematical techniques related to aspects of information security such as *confidentiality*, *data integrity*, *entity authentication* and *non-repudiation*.

- *Confidentiality* is a service used to keep the content of information from all but those authorized to have it. Possible approaches providing it range from physical protection to mathematical algorithms which render data unintelligible.

- *Data integrity* is a service which addresses the unauthorized alteration of data, i.e. insertion, deletion, and substitution. To assure data integrity, one must have the ability to detect data manipulation by unauthorized parties.

- *Authentication* is a service related to identification, which applies to both entities (*entity authentication*) and information itself (*data origin authentication*).

- *Non-repudiation* is a service which prevents an entity from denying previous commitments or actions.

A fundamental goal of cryptography is to adequately address these four areas in both theory and practice. The main difference between ancient and modern cryptography is due to the *Kerchoff's Principle*: a cryptosystem should be secure even if everything about the system is publicly known, except the value of a single parameter (the cryptographic key).

Let $E_e : e \in K$ denote the encryption transformation and $D_d : d \in K$ the decryption one, where K is the key space. In a *Symmetric key* encryption scheme for each associated (e, d) key pair it is computationally easy to determine d knowing e and vice versa. In most practical system $d = e$, thus $c = E_k(p)$ and $p = D_k(c)$ where p is the plaintext, c is the ciphertext and k is the key. Until 1976, when Diffie and Hellman introduced the revolutionary idea of public key cryptography, the only used cryptographic paradigm was the symmetric cryptography one. In a *public key* encryption scheme for each associate (e, d) key pair, given e it is computationally infeasible to determine the corresponding d. E_e is said to be a trapdoor one-way function (easy to compute in one direction and computationally hard to invert without the knowledge of a parameter) with d being the trapdoor information necessary to compute the inverse function and hence consent decryption. The main advantage over the previous encryption paradigm is that allows key exchanges over insecure channels. Each party of the communication owns a secret decryption key sk and a public encryption key pk. Being Alice and Bob the two parties, if Alice wants to send a message m to Bob she firstly needs to recover Bob's public key pk_B and then sends the ciphertext $c = E_{pk_B}(m)$. Bob can decipher the received c and obtain the original message by means of its secret key: $m = D_{sk_B}(c)$.

The main disadvantage of using a public key system instead of a symmetric one is the computational cost, hence the most used strategy is to leverage public key cryptosystem only for the protected transmission of a single use ephemeral key which is shared between the parties which communicates with symmetric encryption.

Encryption and decryption function are based on computationally hard problem, such as

- *integer factorization problem*: given a composite integer n, compute its factorization $\prod_i p_i^{e_i}, e_i \geq 1$;

- *discrete logarithm* extraction in a cyclic group: given $(\langle g \rangle, \cdot)$ and $a = g^x$, find $x \in 0, 1, ..., |g| - 1$;

- find the *shortest vector in a l-dimensional lattice*, which is a vector space with scalar coefficients over $\mathbb{Z}^l$ or $\mathbb{Q}^l$, given a basis for the space and a definition of distance;
- *decoding a general linear code.*

These problems fall under the Nondeterministic-Polynomial (NP) complexity class, which is characterized by problems for which is easy to verify the correctness of a solution but, given the general problem, it is conjectured that no polynomial time algorithm running on a deterministic machine is able to find a solution. In 1997, Shor proved that some NP problems, such as the integer factorization one on which is based RSA algorithm, could be solved in polynomial time by a quantum computer. Since Shor's publication [28], has indeed arisen the need for new algorithms based on NP-complete problem, which will gain only a polynomial speedup when approached with a quantum machine.

This necessity for new algorithms yielded to the NIST Post-Quantum Contest in which LEDAcrypt (Low-dEnsity parity-check coDe-bAsed cryptographic system) is a second round candidate. LEDAcrypt is the union of LEDAkem a Key Encapsulation Module and LEDApkc a Public-Key Cryptosystem, which are based on two NP-Complete problems, which are, respectively

- given a generic random linear code decode a codeword, i.e. find the error vector affecting a codeword;
- given a generic random parity-check matrix decode the syndrome, i.e. find the unique error vector corresponding to it.

At today state of the art, the fastest known method to solve these is exhaustive search.

The aims of this thesis are to analyse LEDAcrypt constant weight encoder, in order to determine how its approximated algorithm affects the output distribution and to achieve a possible mitigation, which improve the distribution without increasing the number of failures and to improve the system performances by leveraging optimized arithmetic algorithms and vectorized instructions specifically targeting an ARM architecture.

This manuscript is composed of six chapters:

- Chapter 1 *Background*: contains a introduction on coding theory and code-based cryptography paradigms, in particular McEliece's and Niederreiter's cryptosystem and of course LEDAcrypt. The last section of this chapter instead covers a brief introduction on vectorized instruction and a more detailed overview on the ARM NEON extension;
- Chapter 2 *State of the Art*: its first section describes in details both the literature exact and approximate approaches to obtain a constant weight encoding, respectively by Cover and Sendrier. The second half of the Chapter concerns suggested optimizations from the literature which are specific for code-based cryptosystems.
- Chapter 3 *Effectiveness and Efficiency Analysis of Constant Weight Encoding*: illustrates our proposed approaches to improve the constant weight encoder output distribution.
- Chapter 4 *Efficient Implementation of the LEDAcrypt Cryptosystem*: explains how to enhance LEDAcrypt performances exploiting sub-quadratic multiple precision multiplication approaches and code vectorization.
- Chapter 5 *Experimental Results*: reports the experimental results validating our improvements on the constant weight encoding technique and the performance gains on the LEDAcrypt primitives gained by exploiting the ARM Neon ISA extensions on a modern ARMv8a platform.
- Chapter 6 *Conclusions*: summarized the achieved results in the enhancement of both the constant weight encoder algorithm and the overall cyptosytem performances.

Chapter 1

Background

A code-based cryptographic system is a public key system exploiting the problem of decoding a general linear code which is NP-complete as demonstrated in [4]. Hence, the idea is to encode the message through a random public matrix, which is an obfuscated version of the code generator matrix. It follows that the correspondent private key, which allows efficient decryption, is the information about the code structure.

In 1978, McEliece in [20] presented an innovative public key cryptosystem based on Goppa codes, which never achieved much popularity due to the large size of the keys needed to obtain even the minimum 80-bit security level. Almost a decade later, in 1986, Niederreiter in [24] proposed an alternative, but security equivalent, public key cryptosystem based on Generalized Reed-Solomon codes which aimed to reduce the key sizes. In 2001, Kobara and Imai with [17] produced a significant step to improved the security of McEliece-based cryptosystems: they were in fact able to provide semantic security against adaptive chosen-ciphertext attacks while reducing the data overhead of $\frac{1}{4}$ compared with alternative generic conversions with practical parameters such as the one proposed in [11] and [25]. Nowadays code-based cryptography is becoming popular for its believed quantum computing resistance: since the publication in 1994 of Shor's factorization algorithm, which threatens systems based on the Discrete Logarithm Problem or on the Factorization Problem, the need for new type of cryptoschemes has indeed arisen. In [28], Shor proposes an innovative quantum algorithm to solve the factorization problem, whose computational time is polynomial

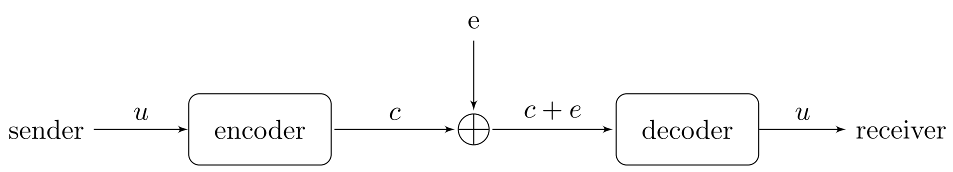


Figure 1.1: Basic error correcting code scenario. The information u is sent encoded (c) over a noisy channel which alters the original message thus the other party receives $c+e$, where e represents some errors. Despite the bitflips, the encoded information can still be decoded into the original u by leveraging its redundancy.

in $log(N)$, where N is the size of the input integer. Shor's algorithm relies on the quantum Fourier sampling attacks, which do not apply to the McEliece cryptosystems, as proven in [8], as long as the underlying q-ary linear code $\mathcal{C}(n, k)$, with $q^{k^2} \leq n^{0.2n}$:

- is "well-scrambled", i.e. it has a generator matrix with rank at least $k - o(\sqrt{n})$ and
- is "well-permuted", i.e. its automorphism group has minimal degree of at least $\Omega(n)$ and has size at most $e^{o(n)}$.

In conclusion, when facing the problem of decoding a generic linear code, using a quantum computer would only provide a polynomial speed-up with respect to the use of a classic computer.

1.1 Basics of Coding Theory

A brief introduction on coding theory is necessary in order to understand the core logic of primitives building code-based cryptosystems and thus the implementation details of LEDAcrypt. For this reason this section reviews a few relevant definitions and concepts.

In coding theory, a binary *error correcting code* (ECC) is used to detect and correct possible communication errors over a noisy channel, such as bits flipping, through the addition of some redundancy to the original information. Said redundancy, being dependent on the original bits, allows the receiver to recover up to a certain number of faulty bits. In Figure 1.1 is represented the common scenario in which ECC are adopted.

The number of asserted bits in a binary vector is defined as the *Hamming weight* of the vector $w_H(\cdot)$, while the *Hamming distance* $d_H(\cdot,\cdot)$ between two vectors is the number of different bits between the two.

A *linear code* is an error-correcting code for which any linear combinations of codewords $\mathbf{c_1}, \mathbf{c_2}$ is also a codeword $\mathbf{c}$.

The *minimum distance of a code* is defined as

$$d(\mathcal{C}) = \min_{\forall \mathbf{c_1},\mathbf{c_2} \in \mathcal{C}, \mathbf{c_1} \neq \mathbf{c_2}} d_H(\mathbf{c_1}, \mathbf{c_2}), \tag{1.1}$$

i.e. is the minimum hamming distance chosen among all the distances computed between any possible pairs of different codewords in $\mathcal{C}$. In a linear code, the minimum distance is equal to the minimum Hamming weight of a non-zero codeword. In fact, since $\mathcal{C}$ is a linear code then $\mathbf{0} \in \mathcal{C}$ and $\mathbf{c}_1 - \mathbf{c}_2 \in \mathcal{C} \forall \mathbf{c}_1, \mathbf{c}_2 \in \mathcal{C}$, it follows that $w_H(\mathbf{c}) = d_H(\mathbf{0}, \mathbf{c})$ and $d_H(\mathbf{c_1}, \mathbf{c_2}) = w_H(\mathbf{c_1} - \mathbf{c_2})$.

Given a message m, the problem of finding any codeword within a given Hamming distance from m is defined as *bounded-distance decoding* problem. A *bounded-distance decoder*, given a distance t, is always able to correct up to t errors in any codeword if $d_H(\mathcal{C}) > 2t + 1$.

A *binary code* is defined as the mapping between an information word u of length k (known as code dimension) and a codeword $\mathbf{c}$ of size n (known as code length):

$$\mathcal{C}(n,k) : \mathbb{F}_2^k \to \mathbb{F}_2^n, n, k \in \mathbb{N}, 0 < k < n \tag{1.2}$$

with $r = n - k$ redundancy symbols.

A *binary linear code* $\mathcal{C}(n,k)$ is a k-dimensional subspace of $\mathbb{F}_2^n$. It can be represented by several $k \times n$ generator matrix $\mathbf{G}$ whose rows form a basis of $\mathcal{C}(n,k)$, among these there is the *systematic representation* which is defined as $\mathbf{G} = [\mathbf{I_k}|\mathbf{P}]$ where $\mathbf{P}$ is a $k \times r$ binary matrix. Being $\mathbf{u} \in \mathbb{F}_2^k$ the information word to be encoded and $\mathbf{c} \in \mathbb{F}_2^n$ its codeword representation: $\mathbf{c} = \mathbf{uG}$. The $r \times n$ parity check matrix $\mathbf{H}$ is defined as a basis of the orthogonal complement of the code space, thus it spans the dual space of $\mathbf{G}$ and the following property holds $\mathbf{G} \cdot \mathbf{H} = \mathbf{0}_{k \times r}$. Moreover a vector $\mathbf{c}$ is a codeword of $\mathcal{C}(n,k)$ only if the *syndrome* vector $\mathbf{s}$ of $\mathbf{x}$ through the parity check matrix $\mathbf{H}$, which is defined as $\mathbf{s} = \mathbf{Hc}^T$, is null. Being $\mathbf{x}$ an incorrect codeword, i.e. $\mathbf{x} = \mathbf{c} + \mathbf{e}$ with $\mathbf{c} \in \mathcal{C}(n, k)$, the syndrome associated with $\mathbf{x}$ corresponds to the syndrome computed through $\mathbf{e}$:

$$\mathbf{s} = \mathbf{H}\mathbf{x}^T = \mathbf{H}(\mathbf{c} + \mathbf{e})^T = \cancel{\mathbf{H}\mathbf{c}^T} + \mathbf{H}\mathbf{e}^T = \mathbf{H}\mathbf{e}^T, \tag{1.3}$$

$$\mathbf{H}\mathbf{e}^T = \begin{bmatrix} h_{0,0} & h_{0,1} & \cdots & h_{0,n1} \\ h_{1,0} & h_{1,1} & \cdots & h_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ h_{r-1,0} & h_{r-1,1} & \cdots & h_{r-1,n-1} \end{bmatrix} \cdot \begin{bmatrix} e_0 \\ e_1 \\ \vdots \\ e_{n-1} \end{bmatrix} = \tag{1.4}$$

$$= \begin{bmatrix} h_{0,0}e_0 + h_{0,1}e_1 + \cdots + h_{0,n-1}e_{n-1} \\ h_{1,0}e_0 + h_{1,1}e_1 + \cdots + h_{1,n-1}e_{n-1} \\ \vdots \\ h_{r-1,0}e_0 + h_{r-1,1}e_1 + \cdots + h_{r-1,n-1}e_{n-1} \end{bmatrix} = \begin{bmatrix} s_0 \\ s_1 \\ \vdots \\ s_{r-1} \end{bmatrix}.$$

It is easy to see from Equation 1.4 that the syndrome vector is the vector of known terms of a system of equations with coefficients $h_{i,j}$ and unknowns e_i. This property can be exploited to extract the error vector $\mathbf{e}$ and recover the information $\mathbf{c}$. For example, in the *list decoding* approach the idea is to compute the syndrome vector for every possible $\mathbf{e}$ until the same value of $\mathbf{s}$ is found and then remove the identified errors from $\mathbf{x}$. Clearly the mentioned approach does not scale well with either the vector length or the number of errors. A more efficient alternative, considers $\mathbf{H}$ as a system of r equations in the n codeword bits, each one producing a syndrome bit. Consequently, each asserted bit in the i-th column of $\mathbf{H}$ can be seen as the indicator of which parity-check equations are involving the i-th bit of the codeword, or in other words, the i-th asserted bit in $\mathbf{s}$ implies that the i-th parity check equation is not satisfied and thus the vector $\mathbf{x}$ contains one or more incorrect bits.

Example 1. *In Equation 1.5 the asserted bits in the first column of* $\mathbf{H}$ *indicate that the first bit of* $\mathbf{e}$*,* e_0*, is involved in the first and third. These* *equations are indeed called parity-check equations.*

$$\mathbf{He}^T = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} e_0 \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \begin{bmatrix} e_0 + e_2 \\ e_1 + e_2 + e_3 \\ e_0 + e_1 + e_3 + e_4 \end{bmatrix} \tag{1.5}$$

For its characteristic, the syndrome is exploited in the so called *syndrome decoding procedure* of a codeword which iterates over three steps until a zero-valued syndrome is reached:

1. compute $\mathbf{s}$,
2. select candidate faulty positions as the ones involved in the highest amount of parity check equations with non null known term (i.e. corresponding to a non-null syndrome bit),
3. flip the candidate incorrect bits and update the value of the syndrome.

For a general system, the number of required iterations is worse than polynomial in the system parameters, for this reason this approach is feasible only when $\mathbf{H}$ is sparse.

A *low-density parity-check* (LDPC) *code* $\mathcal{C}(n, k)$, defined by Gallager in [12], is a particular linear code characterized by a sparse parity check matrix $\mathbf{H}$ and thus by the existence of an efficient syndrome decoding procedures known as Bit-Flipping (BF) algorithms. A BF-decoder, taking advantage of the sparsity of $\mathbf{H}$, iteratively learns which are the possible incorrect bits of the codeword and determines which to flip in order to obtain a zero-valued syndrome vector.

A *quasi-cyclic* (QC) *code* is a linear block code $\mathcal{C}(n, k)$ having information word size $k = pk_0$ and codeword size $n = pn_0$, where each cyclic shift of a codeword by n_0 symbols results in another valid codeword. The generator matrix $\mathbf{G}$ of said QC code, composed by $p \times p$ circulant block, has the following form

$$G = \begin{bmatrix} G_{0,0} & G_{0,1} & \cdots & G_{0,n_0-1} \\ G_{1,0} & G_{1,1} & \cdots & G_{1,n_0-1} \\ \vdots & \vdots & \ddots & \vdots \\ G_{k_0-1,0} & G_{k_0-1,1} & \cdots & G_{k_0-1,n_0-1} \end{bmatrix}$$

where each block $\mathbf{G}_{i,j}$ is a $p \times p$ circulant matrix, i.e a matrix having the following form

$$G_{i,j} = \begin{bmatrix} g_0 & g_1 & g_2 & \cdots & g_{p-1} \\ g_{p-1} & g_0 & g_1 & \cdots & g_{p-2} \\ g_{p-2} & g_{p-1} & g_0 & \cdots & g_{p-3} \\ \vdots & \vdots & \cdots & \ddots & \vdots \\ g_1 & g_2 & g_3 & \cdots & g_0 \end{bmatrix}.$$

1.2 McEliece Cryptosystem

In 1978, Robert McEliece proposed the first public key cryptosystem based on Goppa codes in [20]. The main advantages of this system are the fast encryption and decryption procedures and of course its quantum resistance. However to achieve the desired long term security it requires a large public key. In [5], the public key sizes for the minimum security level of 80-bit and for the long term one of 256-bit are set respectively around 60 KB and 958 KB. The primitives of the cryptosystem are described next.

Key generation After choosing a code family, the binary linear code $\mathcal{C}(n, k)$ is randomly picked from the set of codes which guarantee an efficient decoding. Such code is able to correct up to t errors by means of a generator matrix $\mathbf{G}$, which is part of the secret key. The public key instead contains a matrix aimed at the message encoding. This matrix is computed as $\mathbf{G}' = \mathbf{SGP}$, where $\mathbf{S}$ is a random dense $k \times k$ non-singular binary scrambling matrix and $\mathbf{P}$ is a random $n \times n$ permutation matrix. By obfuscating the generator matrix, it is possible to encrypt a plaintext without revealing any information about the code structure.

$$\text{Public key: } \langle \mathbf{G}' \rangle$$
$$\text{Secret key: } \langle \mathbf{G}, \mathbf{S}, \mathbf{P} \rangle$$

Encryption Given a plaintext message m, the associated ciphertext is computed as

$$\mathbf{c} = \mathbf{m}\mathbf{G}' + \mathbf{e} \tag{1.6}$$

where $\mathbf{e}$ is the random error vector with Hamming weight of t and the same length as $\mathbf{m}$.

Decryption When receiving a ciphertext $\mathbf{c}$, it can be decomposed into

$$\mathbf{c} = \mathbf{m}\mathbf{G}' + \mathbf{e} = \mathbf{m}\mathbf{G}\mathbf{S}\mathbf{P} + \mathbf{e} \tag{1.7}$$

and with the secret key knowledge it is possible to compute

$$\mathbf{c}' = \mathbf{c}\mathbf{P}^{-1} = \mathbf{m}\mathbf{G}\mathbf{S} + \mathbf{e}\mathbf{P}^{-1}. \tag{1.8}$$

$\mathbf{c}'$ is a codeword of the chosen code affected by t errors, thus it is possible to correct it obtaining $\mathbf{m}' = \mathbf{m}\mathbf{G}\mathbf{S}$ and thus $\mathbf{m} = \mathbf{m}'\mathbf{S}^{-1}\mathbf{G}^{-1}$.

1.3 Niederreiter Cryptosystem

In [24], Harald Niederreiter developed a code-based cryptosystem, exploiting the same trapdoor of the McEliece's, which used Generalized Reed-Solomon codes as private code. Generalized Reed-Solomon Codes were afterwards proven to make the system vulnerable, which instead remains as secure as McEliece when both use the same family of codes. Its primitive are described in the following paragraphs.

Key generation The private key is generated from two secret matrices:

- the $r \times n$ parity-check matrix $\mathbf{H}$,
- a random non-singular $r \times r$ scrambling matrix $\mathbf{S}$.

From these, the public matrix is computed as $\mathbf{H}' = \mathbf{S}\mathbf{H}$, which concides with the private one without allowing efficient decoding.

Public key: $\langle \mathbf{H}' \rangle$
Secret key: $\langle \mathbf{H}, \mathbf{S} \rangle$

Encryption Each message $\mathbf{m}$ is mapped into one or more n-bit strings $\mathbf{e}$ with weight t, which are encoded through the receiver public key:

$$\mathbf{x} = \mathbf{H}'\mathbf{e}^T = \mathbf{S}\mathbf{H}\mathbf{e}^T.$$

Differently from McEliece's, Niederreiter encryption is deterministic: while within McEliece cryptosystem some random errors are added to the encoded plaintext, in Niederreiter it is the error pattern deterministically generated from the plaintext to be converted in ciphertext.

Decryption Each received message $\mathbf{x}$ is efficiently decoded through the knowledge of the private key $\mathbf{H}$. The first step is to obtain the corresponding syndrome

$$\mathbf{s} = \mathbf{S}^{-1}\mathbf{x} = \mathbf{S}^{-1}\mathbf{S}\mathbf{H}\mathbf{e}^T = \mathbf{H}\mathbf{e}^T.$$

from which, by means of a syndrome decoding algorithm, is extracted the error vector. Reverse mapping from $\mathbf{e}$ reveals the message $\mathbf{m}$.

Both Niederreiter and McEliece were built in the original works with code families which are provided with bounded-distance decoders able to correct up to t faulty bits, thus when adding exactly t error bits the obtained decryption failure rate (DFR) is zero.

1.4 LEDAcrypt

LEDAcrypt (Low-dEnsity parity-check coDe-bAsed cryptographic system) is a public key system candidate to the second round of the NIST's contest for the *Post-Quantum Cryptography Standardization.* It was born from the combination of the first round proposals LEDApkc and LEDAkem, which provide, respectively, a Public-Key Cryptosystem (PKC) and a Key Encapsulation Module (KEM).

LEDAcrypt relies on a QC-LDPC code $\mathcal{C}(pn_0, p(n_0 - 1))$, with $r = pn_0 - p(n_0 - 1) = p$ redundancy symbols, meaning that

- its generator and parity-check matrices are composed by $p \times p$ circulant sub-matrices, where p is an odd prime integer, thus each of these can be easily represented as a polynomial of $\mathbb{F}_2[x]/\langle x^p + 1\rangle$ having coefficients equal to the first row entries;

- its parity-check matrix is sparse and can be efficiently decoded applying a BF-decoder.

Specifically $n_0 = \{2, 3, 4\}$ and p is a prime number such that $\texttt{ord}_p(2) = p-1$. Using non-algebraic QC-LDPC codes as private codes, this cryptosystem is able to overcome the fundamental public key size problem of its predecessors. It should be noted that the QC-LDPC private code cannot be coincident neither equivalent with the public one, as it was in McEliece proposal, otherwise an attacker could recover the secret key by exploiting the LDPC characteristic as illustrated in [23].

1.4.1 LEDAkem

The cryptographic primitives of LEDAcrypt KEM are presented in the next paragraphs.

Key Generation This process begins with the generation of the private key, by randomly selecting the asserted bits of the parity check matrix $\mathbf{H}$ and of the transformation matrix $\mathbf{Q}$ so that the following properties are respected:

- $\mathbf{H}$ is composed of $1 \times n_0$ circulant blocks of $p \times p$ elements,
- in order to guarantee the invertibility of each block of $\mathbf{H}$, the weight of each block d_v is fixed and odd,
- $\mathbf{Q}$ is composed of $n_0 \times n_0$ circulant blocks of $p \times p$ elements,
- the weights of each row/column of $\mathbf{Q}$ circulant blocks define a circulant matrix, thus each weight is fixed and the permanent of each circulant matrix is equal to m, which must be odd in order to guarantee the invertibility of $\mathbf{Q}$. The permanent of a $n \times n$ matrix $\mathbf{A}(a_{i,j})$ is defined as $\sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$, where S_n is the symmetric group over the matrix.

These characteristics are fundamental to ensure the non-singularity of the last block of $\mathbf{L}$, which is obtain as

$$\mathbf{L} = \mathbf{HQ} = [\mathbf{L_0}|\mathbf{L_1}|...|\mathbf{L_{n_0-1}}], \tag{1.9}$$

subsequently

$$\mathbf{M} = \mathbf{L_{n_0-1}^{-1}L} = [\mathbf{M_0}|\mathbf{M_1}|...|\mathbf{M_{n_0-2}}|\mathbf{I_p}] = [\mathbf{M_l}|\mathbf{I_p}] \tag{1.10}$$

from which originates the public key. Since the key dimensioning is a relevant issue in code based cryptosystems, it is convenient to discard the last identity block, which does not hold any information. Therefore the public key matrix is the concatenation of $(n_0 - 1)$ $p \times p$ blocks circulant matrices:

$$\mathbf{M_l} = [\mathbf{M_0}|\mathbf{M_1}|...|\mathbf{M_{n_0-2}}]. \tag{1.11}$$

With regard to the keys storage is sufficient to know the TRNG seed fed to the DRBG which is used to generate the private matrices H and Q, in order to recompute the private key matrices, and to represent the dense public key circulant matrix with its dual polynomial form.

Public key: $\langle \mathbf{M_l} \rangle$
Secret key: $\langle \mathbf{H}, \mathbf{Q} \rangle$

Encryption Being $\mathbf{e}$ a random error binary vector of length $n = n_0 p$ and weight t, the plaintext coincides with the random generated secret k_s = KDF($\mathbf{e}$) to be shared with the other party. The ciphertext, which is its associated $p \times 1$ syndrome vector $\mathbf{s}$, is computed as

$$\mathbf{s} = [\mathbf{M_l}|\mathbf{I}]\mathbf{e}^T, \tag{1.12}$$

where $\mathbf{M_l}$ is the public key of the receiver.

Decryption Inferring the secret error vector from the syndrome is possible in few steps only through the knowledge of the associated secret key, in fact

$$\mathbf{s} = \mathbf{M}\mathbf{e}^T = \mathbf{L_{n_0-1}^{-1}L}\mathbf{e}^T = \mathbf{L_{n_0-1}^{-1}HQ}\mathbf{e}^T. \tag{1.13}$$

The first step is to compute the syndrome associated to the expanded error vector $\mathbf{e}' = \mathbf{e}\mathbf{Q}^T$

$$\mathbf{s}' = \mathbf{L_{n_0-1}}\mathbf{s} = \mathbf{HQ}\mathbf{e}^T = \mathbf{H}\mathbf{e}'^T. \tag{1.14}$$

As a result of the LEDAkem efficient decoding procedure, it is not necessary to compute or store $\mathbf{Q}^{-1}$ to retrieve $\mathbf{e}$ from $\mathbf{e}$', which in fact is derived directly by syndrome decoding through $\mathbf{H}$ while taking into account the effects of $\mathbf{Q}$. Finally the shared secret can be simply obtained as $k_s = \mathrm{KDF}(\mathbf{e})$.

It is possible to observe some decoding failures, in general, since said decoder is not a bounded-distance one, which instead ensures to correct up to t errors by construction. Hence, the cryptosystem parameters should be chosen adequately: the decoding radius of the private code should be sufficiently larger than $w_H(\mathbf{e}\,) \approx mt$, due to the sparsity of $\mathbf{Q}$ and $\mathbf{e}$. In the unfortunate case of a decoding failure, the secret key is derived through hashing a secret value and the ciphertext. Within this strategy, an attacker in control of composing messages outside of the proper message space will not be able to distinguish whether the failure regarded the malformed plaintext or was due to the intrinsic behaviour of the underlining code, thus he will not be able to draw any conclusion about the decoding abilities of the QC-LDPC code.

1.4.2 LEDApkc

The primitives of the cryptosystem, which are based on McEliece's, are outlined in the following paragraphs along with the LEDApkc general functioning schemes (Figure 1.2). Since McEliece primitives do not guarantee semantic security, LEDApkc ones have been modified to incorporate the γ-conversion scheme proposed by Kobara and Imai, which guarantee IND-CCA2[1] as shown in Paragraph *Kobara-Imai γ-conversion scheme.*

Key generation LEDApkc has the same key generation process of LEDAkem, which is described in section. 1.4.1.

$$\text{Public key: } \langle \mathbf{M_l} \rangle$$
$$\text{Secret key: } \langle \mathbf{H}, \mathbf{Q} \rangle$$

Encryption Given any information word $\mathbf{u}$ as a $1 \times p(n_0 - 1)$ binary vector, the corresponding encrypted message is computed, as in the McEliece version, as

$$\mathbf{x} = \mathbf{u}\mathbf{G}' + \mathbf{e} = \mathbf{u}[\mathbf{I_k}|\mathbf{M_l^T}] + \mathbf{e} \tag{1.15}$$

[1]IND-CCA2: indistinguishability under adaptive chosen ciphertext attack.

where $\mathbf{G}'$ is the systematic generation matrix derived from the public key and $\mathbf{e}$ is a random $1 \times pn_0$ binary error vector with Hamming weight t (denoting the error correction capability of the code). The use of the systematic form of the generation matrix results in the original message to be exposed in $\mathbf{x}$ and thus to be easily recoverable for an observer.

Decryption Given an encrypted $1 \times pn_0$ binary vector $\mathbf{x}$ and the secret key $\langle \mathbf{H}, \mathbf{Q} \rangle$, the receiver can compute the binary $p \times 1$ syndrome vector $\mathbf{s}$ associated with the message:

$$\mathbf{s}^T = (\mathbf{HQ})\mathbf{x}^T = (\mathbf{HQ})(\mathbf{uG}' + \mathbf{e})^T = (\mathbf{HQ})\mathbf{e}^T. \tag{1.16}$$

The computed syndrome $\mathbf{s}$ can be seen as the syndrome of the expanded error vector $\mathbf{e}' = \mathbf{eQ}^T$ through $\mathbf{H}$, allowing to recover the original error vector $\mathbf{e}$ by means of an BF-decoding procedure. From 1.15 and knowing $\mathbf{e}$.

$$\mathbf{x} + \mathbf{e} = \mathbf{u}[\mathbf{I_k}|\mathbf{M_l}^T] \tag{1.17}$$

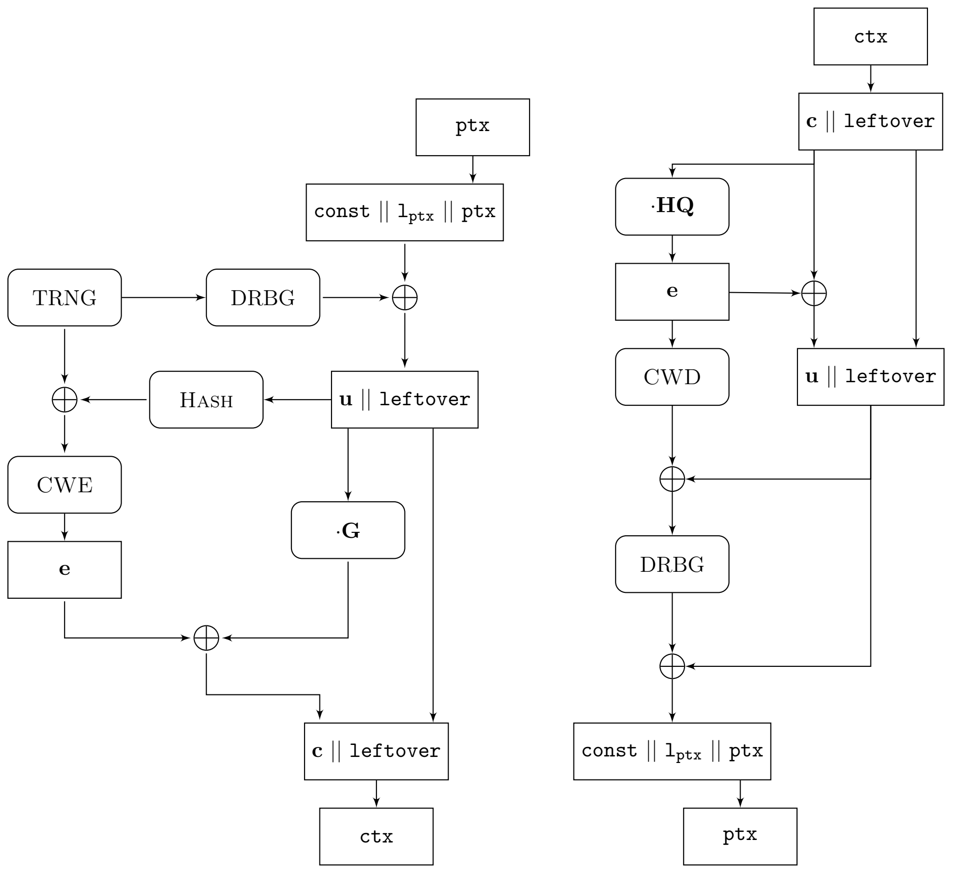
meaning that the information word $\mathbf{u}$ is represented by the first $k = p(n_0 - 1)$ bits of the previous vector.

Kobara-Imai γ-conversion scheme

By applying the Kobara-Imai conversion scheme to the previously described McEliece primitives in LEDApkc, the cryptosystem achieves IND-CCA2 security: the enhanced encryption and decryption algorithms are outlined in Algorithm 1.4.1 and 1.4.2. The improvement is based on the obfuscation of the information word before the encrypting procedure.

Within the γ-conversion the plaintext `ptx` is at first prefixed with a constant bitstring `const` of size l_{const} and the value of the plaintext length $l_{\texttt{ptx}}$. The expanded plaintext is then xored with the output of a DRBG seeded with a TRNG: the result is a perfectly random `obfuscatedPtx` ready to be encoded (lines 1-3). The computed `obfuscatedPtx` is then split into two bitstrings (line 5):

- the first $p(n_0 - 1)$ bits represent the information word `iword` to be encoded through the encryption primitive;



(a) LedaPKC encryption (b) LedaPKC decryption

Figure 1.2: Ledapkc encryption and decryption schemes.

- the remaining $l_{\texttt{obfuscatedPtx}} - p(n_0 - 1)$ bits form the `leftover` bitstring.

In order to obtain IND-CCA2 security a new seed from TRNG has to be used for each new message, so it also has to be sent to the receiver. Of course the seed needs to be unrecognisable hence, in line 4, it is zero-padded to reach the same length of the digest of the hash of the `obfuscatedPtx` and xored with it. The obtained `obfuscatedSeed` is then fed to the constant weight encoder to produce the error vector $\mathbf{e}$ (line 8). The goal of the Constant Weigth Encoding (CWE) function is to generate a vector of fixed length n within Hamming weight t, thus to allow the paired decryption process, this function should be bijective. At last the `iword` is encrypted as $c = \texttt{iword}[\mathbf{I_k}|\mathbf{M_l^T}] + \mathbf{e}$ and prefixed to the `leftover` bitstring (lines 6, 10 and 11).

The resulting enhanced decryption procedure operates in reversed order.

```
Algorithm 1.4.1: PKC encryption transformation
Input: ptx: plaintext bit-string, with $l_{\texttt{ptx}} \geq 0$
       $pk^{\texttt{McE}}$: QC-LDPC based McEliece public key

Output: ctx ciphertext bit-string
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 $l_{\texttt{hash}}$
      $l_{\texttt{obfuscatedPtx}} = \max\Big(p(n_0 - 1),\ \Big\lceil \frac{l_{\texttt{const}}+l_{\texttt{ptx}}}{8} \Big\rceil \cdot 8\Big)$
      $l_{\texttt{const}} = l_{\texttt{seed}}$
      $l_{\texttt{iword}} = p(n_0 - 1)$
      $l_{\texttt{eword}} = l_{\texttt{hash}}$
1  seed ← TRNG()                                        // bit-string with length $l_{\texttt{seed}}$
2  pad ← DRBG(seed)                              // bit-string with length $l_{\texttt{obfuscatedPtx}}$
3  obfuscatedPtx ← ZeroPadByteAligned(const||lenField||ptx) ⊕ pad
4  obfuscatedSeed ← ZeroExtendByteAligned(seed, $l_{\texttt{hash}}$) ⊕ Hash(obfuscatedPtx)
5  {iword, leftOver} ← Split(obfuscatedPtx, $l_{\texttt{iword}}$, $l_{\texttt{obfuscatedPtx}} - l_{\texttt{iword}}$)
6  $u$ ← ToVector(iword) // $1 \times p(n_0 - 1)$ information word vector
7  repeat
8      {$e$, encodingOk} ← ConstantWeightEncoder(obfuscatedSeed)
9  until encodingOk = true
10 $c$ ← Encrypt$^{\texttt{McE}}$($u$, $e$, $pk^{\texttt{McE}}$)                          // $1 \times pn_0$ codeword
11 ctx ← ToBitString($c$)||leftover                    // bit-string with $l_{\texttt{ctx}} = pn_0$
12 return ctx
```

Figure 1.3: Description of the KI-γ encryption function adopted to define the encryption primitive of LEDAcrypt PKC

First of all the ciphertext `ctx` is split into `leftOver` and codeword vector c, from which the error vector **e** and the `iword` are obtained (lines 1-5). Line 4 verifies the success of the decoding operation: in fact, there is a chance of decryption failure due to the use of a non-bounded distance decoder. Notwithstanding this, the IND-CCA2 property is maintained by making the decoding failure happen with negligible probability. Subsequently, the value of the ephemeral seed is retrieved by xoring the result of the constant weight decoding of **e** with the digest of the hashing of the concatenation of the `iword` and `leftOver` bitstrings (line 7). In lines 8-10, the `extendedPtx`, which correspond to the concatenation of a constant bitstring, the value of the plaintext length and the original plaintext, is extracted. Finally, if the decoded value of the first $l_{\texttt{const}}$ bits of the `extendedPtx` is different from the fixed `const` than the outcome is discarded, otherwise the `ptx` is returned.

Algorithm 1.4.2: PKC decryption transformation

```
Input: ctx: ciphertext bit-string.
       $sk^{\mathtt{McE}}$: PKC private key.
Output: ptx plaintext bit-string
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.
      $\mathtt{const} = 0^{l_{\mathtt{seed}}}$
      Hash: hash function with digest length in bits $l_{\mathtt{hash}}$
cword, leftOver ← Split(ctx, $pn_0$, $l_{\mathtt{ctx}} - pn_0$)
$c$ ← ToVector(cword)
$\{u, e, \mathtt{res}\}$ ← Decrypt$^{\mathtt{McE}}$($c$, $sk^{\mathtt{McE}}$)
if res = true and $wt(e) = t$ then
    iword ← ToBitString($u$)
    obfuscatedSeed ← ConstantWeightDecode($e$)
    seed ← ZeroTrim(obfuscatedSeed ⊕ Hash(iword||leftOver), $l_{\mathtt{seed}}$)
    pad ← DRBG(seed)
    extendedPtx ← obfuscatedPtx ⊕ pad // extendedPtx should equal
        const||lenField||ptx
    {retrievedConst, ptx} ← ZeroTrimAndSplit(extendedPtx, $l_{\mathtt{const}}$, $l_{\mathtt{lenField}}$)
    if retrievedConst = const then
        return ptx
return ⊥
```

Figure 1.4: Description of the KI-γ decryption function adopted to define the decryption primitive of LEDAcrypt PKC

1.4.3 Efficient Decoding

A noticeable characteristic of LEDAcrypt is its efficiency in decoding information due to its *Q-decoder*, which is an improved version of a BitFlipping decoder arranged specifically for it and presented in [2].

As explained before the BF-decoder is a fixed-point algorithm which selects the most probable incorrect bits and flips them at each round until a zero-valued syndrome is obtained. The Q-decoder mechanism, outlined in Algorithm 1.4.3, is the same as the previously described BF-decoder, but instead of recovering the expanded error vector **e**' it retrieves directly **e**. This is possible since the positions of the error to be corrected are not uniformly distributed, but depend on **Q**, being $\mathbf{e}' = \mathbf{e}\mathbf{Q}^T$, which is known to the decoder.

The first is step, as in BF-decoding, is the unsatisfied parity checks computation (lines 5-9). In lines 12-17, the Q-decoder takes into account the asserted positions of **Q** and computes the similarity between patterns of bits in the rows of **Q** and the unsatisfied parity checks vector. If the estimate is above a given threshold, which is retrieved from a precomputed table (lines

Algorithm 1.4.3: Q-decoding

```
   Input: $s'$: QC-LDPC syndrome, binary vector of size $p$
          Htr: transposed parity-check matrix, represented as an $n_0 \times d_v$ integer matrix
          containing the positions in $\{0, 1, \ldots, p-1\}$ of the set coefficients in the $n_0$ blocks
          of $H^T = [H_0^T \mid H_1^T \mid \ldots \mid H_{n_0-1}^T]$
          Qtr: private matrix, represented as an $n_0 \times m$, $m = \sum_{i=0}^{n_0-1} m_i$ integer matrix
          containing the positions in $\{0, \ldots, n_0 p - 1\}$ of the asserted coefficients in $Q^T$ rows
   Output: $e$: the decoded error vector with size $n_0 p$
           decodeOk: Boolean value denoting the successful outcome of the decoding action
   Data: imax: the maximum number of allowed iterations before reporting a decoding
         failure
         LutS: piecewise constant function yielding the value of the bit flipping threshold of
         similarity, given the syndrome weight. It is represented as an array of (weight,
         threshold) pairs for all the boundary values of the piecewise function.
 1 iter $\leftarrow 0$
 2 repeat
 3     unsat_pc $\leftarrow [0 \mid \ldots \mid 0]$  // array of $n_0 p$ counters of unsatisfied parity checks
 4     currSynd $\leftarrow s'$
 5     for i $= 0$ to $n_0 - 1$ do
 6         for exp $\leftarrow 0$ to $p - 1$ do
 7             for h $\leftarrow 0$ to $d_v - 1$ do
 8                 if GETBLOCKCOEFFICIENT(currSynd, (exp + Htr[i][h]) mod $p$) $= 1$ then
 9                     unsat_pc[i $\cdot p +$ exp] $\leftarrow$ unsat_pc[i $\cdot p +$ exp] $+ 1$
10     $\overline{\text{w}} \leftarrow$ MAX({w | (w, th) $\in$ LutS $\wedge$ w $<$ WEIGHT(currSynd)})
11     $\overline{\text{th}} \leftarrow$ th | ($\overline{\text{w}}$, th) $\in$ LutS
12     for i $\leftarrow 0$ to $n_0 - 1$ do
13         for exp $\leftarrow 0$ to $p - 1$ do
14             similarity $\leftarrow 0$
15             for k $\leftarrow 0$ to $m - 1$ do
                   // qrow contains the positions of the ones of a row of $Q$
                   rotated intra-block by exp
16                 qrow[k] $\leftarrow$ Qtr[i][k] $-$ (Htr[i][k] mod $p$) $+$ ((exp + Qtr[i][k]) mod $p$)
17                 similarity $\leftarrow$ similarity + unsat_pc[qrow[k]]
18             if similarity $\geq \overline{\text{th}}$ then
19                 $e$[i $\cdot p +$ j] $\leftarrow e$[i $\cdot p +$ j] $\oplus 1$
20                 for k $\leftarrow 0$ to $m - 1$ do
21                     for h $\leftarrow 0$ to $d_v - 1$ do
22                         idx $\leftarrow$ (Htr[qrow[k]/$p$][h] + (qrow[k] mod $p$)) mod $p$
23                         $s'$[idx] $\leftarrow s'$[idx] $\oplus 1$
24     iter $\leftarrow$ iter $+ 1$
25 until $s' \neq 0$  and  iter $<$ imax
26 if $s' = 0$ then
27     return $e$, $true$
28 return $e$, $false$
```

10 and 11), then both **e** and its related syndrome **s** are updated accordingly (lines 18-24) and the algorithm proceeds with the next iteration.

Notice that it is possible to leverage the sparsity of the matrices and thus store only the position of the asserted bits, reducing significantly the storage requirements.

A further optimization can be achieved by parallelizing parts of this sequential version. This is feasible since the instructions inside the Q-Decoder loops are independent from each other and only dependent on the loop indexes, which are fixed by the system. A straightforward way to accomplish so is by means of vector instructions, which are capable of performing the same operation over multiple elements.

1.5 Vector Architectures

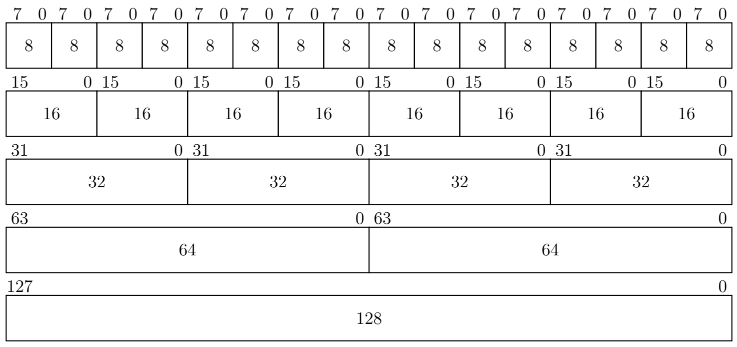
Vector architectures are able to operate over multiple data at the same time, that is why, following Flynn's taxonomy described in [10], they are also called SIMD (Single Instruction stream, Multiple Data stream) architectures. SIMD computers exploit data-level parallelism by applying the same operation to multiple items of data in parallel: each processor has its own data memory, but there are only a single instruction memory and a control processor, which fetches and dispatches instructions. SIMD instructions strength consists in the convenience of programming in a sequential way, while achieving parallel speed-up. Obviously, in order to work properly these architectures have specifically designed components, such as vector register, vector functional units and vector load and store units. This type of architecture retrieves data from memory, stores it in vector registers, performs the same operation on every element leveraging its vector functional units, and saves the results back into memory: in this way it increases the number of instruction per clock cycle.

Nowadays most of computation, especially scientific operations and image or sound processing ones, takes advantage of the so called *SIMD Extensions*. Differently from earlier vector architectures, SIMD extended processors are capable of operate on sub-parts of a vector element. In particular, *Arm Neon technology* is a SIMD architecture extension for the ARM Cortex-A series and Cortex-R52 processors.

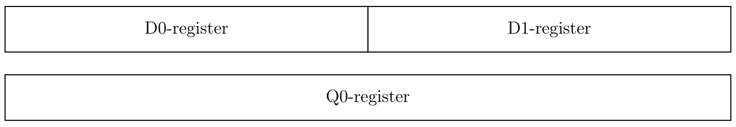
1.5.1 ARM Neon

Usually microprocessor instructions process data by

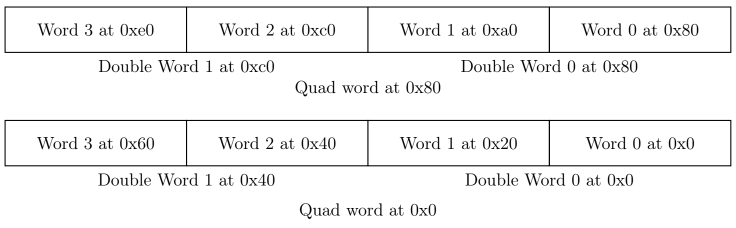
- *Single Instruction Single Data*: each operation specifies the single data source to process and hence processing multiple data items requires multiple instructions



(a) Available vector types



(b) Register aliasing



(c) Data storage convention in memory

Figure 1.5: ARM register and memory conventions.

- *Single Instruction Multiple Data*
  - *vector mode*: an operation can specify that the same processing occurs for multiple data sources, in ARM terminology this is called Vector Floating Point Extension and has been deprecated and replaced with Neon technology
  - *packed data mode*: an operation can specify that the same processing occurs for multiple data fields stored in one large register, thus a single instruction operates on all data values in the large register at the same time, in ARM terminology this is called Advanced SIMD technology or Neon technology.

Neon registers are considered as vector of elements of the same data type allowing to perform the same operation on every lane of the vector at the same time, thus there cannot be a carry or overflow from one lane to another. In particular, as shown in Figure 1.5, Neon instructions allow up to:

- 16x8-bit, 8x16-bit, 4x32-bit, 2x64-bit integer operations
- 8x16-bit, 4x32-bit, 2x64-bit floating-point operations.

In fact, in the Neon extension, the data is organized into very long registers which are composed of elements of 8, 16, 32 or 64 bits. It is possible to operate over 2 different kind of long registers, 64 bit D register and 128 bit Q register, which alias each other: D1 corresponds to the higher half of Q0 while D0 corresponds to its lower half, as represented in Figure 1.5. For this reason, each Neon unit register file can be seen as 16 128-bit Q (quadword) or 32 64-bit D (doubleword) registers.

A convenient way to leverage these vectorize processors is by means of the so called *Neon intrinsics*: function calls which are replaced by the compiler with the appropriate sequence of Neon assembly instructions. Their strength lies in offering as much control as writing assembly language, but leaving the allocation of registers to the compiler. The header file needed to use these functions with the `GCC` and `LLVM` compiler suites is `arm_neon.h`.

Neon vector data types, which are summarized in Table 1.1, are named according to the `<type><Block Size>x<Lanes Number>_t` pattern, for example `int32x4_t` describe a vector of four 32-bit element of type `int`. Some Neon instructions support the polynomial (`poly_t`) type, which represents a binary polynomial in x of the form $b_{n-1}x^{n-1} + \cdots + b_1x + b_0$ where b_k is k-th bit of the value. The same kind of nomenclature is applied to intrinsics instruction, which follow the pattern `<opname><flag>_<Scalar Type><Block Size>` (possible options are illustrated in Table 1.2), for example `vaddq_u64` is an add operation over two 64x2 vector of unsigned type (the `q` flag indicates a Q register hence 128-bit operand).

Below are explained the intrinsic functions employed in this thesis codebase, conveniently divided per instruction class.

Data movement between registers

`uint64x2_t vdupq_n_u64(uint64_t value)`: This instruction duplicates the

Table 1.1: This table illustrates the possible Neon vector data types which are defined following the pattern `<type><Block Size>x<Lane Number>_t`. For example `poly64x2_t` contains a polynomial variable of size 128 bit split in two 64 bit chunks.

Type	**Size**	**Block Size × Lanes Number**
`poly, int, uint, float`	64 bits	64x1, 32x2, 16x4, 8x8
	128 bits	64x2, 32x4, 16x8, 8x16

Table 1.2: This table illustrates the possible Neon vector operation flags and types which are defined following the pattern `<opname><flag>_<Scalar Type><Block Size>`. For example, `veorq_s16` performs a xor operation over two vectors of length 128 bits working in parallel on 8 lanes of 16 bit integer value, while `veor_s16` performs the same operation but on 4 16-bit integer lanes.

Flag	**Scalar Type**	**Block Size**
`q` iff 128-bit	`p (poly), s (int), u (uint), f (float)`	`8,16,32,64`

specified value into each element of a vector, and writes the result to the destination SIMD&FP register.

`uint64x2_t vcombine_u64 (uint64x1_t low, uint64x1_t high)`: This instruction joins two smaller vectors into a single larger vector.

`uint64_t vgetq_lane_u64 (uint64x2_t v, const int lane)`: This instruction reads the unsigned integer, in the specific lane of the source vector register and writes the result to the destination general-purpose register.

`uint64x2_t vcgeq_u64 (uint64x2_t a, uint64x2_t b)`: This instruction compares each vector element in the first source vector register with the corresponding vector element in the second source vector register and if the first unsigned integer value is greater than or equal to the second unsigned integer value sets every bit of the corresponding vector element in the destination vector register to one, otherwise sets every bit of the corresponding vector element in the destination vector register to zero.

`uint64x2_t vsetq_lane_u64 (uint64_t a, uint64x2_t v, const int lane)`:

This instruction copies the vector element of the source vector register to the specified vector element of the destination vector register.

`uint64x1_t vget_high_u64 (uint64x2_t a)`: This instruction duplicates the higher part of a vector register, and writes it to the destination vector register.

`uint64x1_t vget_low_u64 (uint64x2_t a)`: This instruction duplicates the lower part of a vector register, and writes it to the destination vector register.

Vector Arithmetic

`uint64x2_t vaddq_u64 (uint64x2_t a, uint64x2_t b)`: This instruction adds corresponding 64-bits elements in the two source vector registers, places the results into a vector, and writes the vector to the destination vector register.

`uint64x2_t vandq_u64 (uint64x2_t a, uint64x2_t b)`: This instruction performs a bitwise AND between the two source vector registers, and writes the result to the destination vector register.

`uint64x2_t vorrq_u64 (uint64x2_t a, uint64x2_t b)`: This instruction performs a bitwise OR operation between the two source vector registers, and places the result in the destination vector register.

`uint64x2_t veorq_u64 (uint64x2_t a, uint64x2_t b)`: This instruction performs a bitwise Exclusive OR operation between the two source vector registers, and places the result in the destination vector register.

`poly128_t vmull_p64 (poly64_t a, poly64_t b)` : This instruction multiplies corresponding elements in the lower half of the polynomials in the two source vector registers, places the results in a vector, and writes the vector to the destination vector register. The destination vector elements are twice as long as the elements that are multiplied.

`poly128_t vmull_high_p64 (poly64x2_t a, poly64x2_t b)`: Same as previous one but for lower halves.

Memory Load and Store

`uint64x2_t vld1q_u64 (uint64_t const * ptr)`: This instruction loads multiple single-element from memory and writes the result to one vector registers.

`void vst1q_u64 (uint64_t * ptr, uint64x2_t val)`: This instruction stores elements to memory from one vector register.

Shift

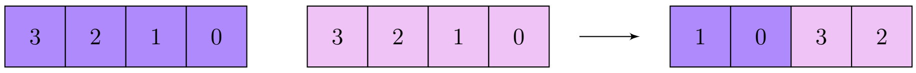
`uint64x2_t vshrq_n_u64 (uint64x2_t a, const int n)`: This instruction reads each vector element in the source vector register, right shifts each result by an immediate value, writes the final result to a vector, and writes the vector to the destination vector register. All the values in this instruction are unsigned integer values. The results are truncated.

`uint64x2_t vshlq_n_u64 (uint64x2_t a, const int n)`: Same as previous one but shifts left.

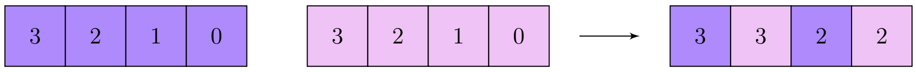
Permute

`uint64x2_t vextq_u64(uint64x2_t a, uint64x2_t b, const int n)`: This instruction extracts the lowest vector elements from the second source vector register and the highest vector elements from the first source vector register, concatenates the results into a vector, and writes the vector to the destination vector register vector. The index value specifies the lowest vector element to extract from the first source register, and consecutive elements are extracted from the first, then second, source registers until the destination vector is filled.

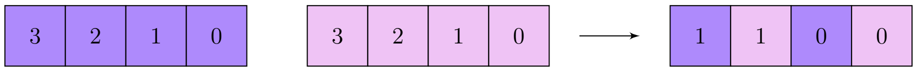
`uint64x2_t vzip1q_u64 (uint64x2_t a, uint64x2_t b)`: This instruction



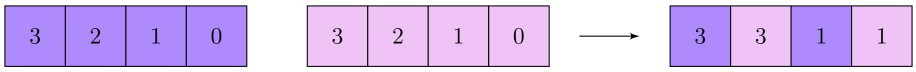
(a) `vextq_u32(A,B,2)`



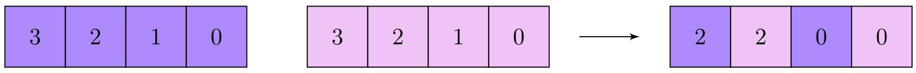
(b) `vzip1q_u32(A, B)`



(c) `vzip2q_u32(A, B)`



(d) `vtrn1q_u32(A, B)`



(e) `vtrn1q_u32(A, B)`

Figure 1.6: Permute Instructions over vectors of four 32-bit elements.

reads adjacent vector elements from the upper half of two source vector registers as pairs, interleaves the pairs and places them into a vector, and writes the vector to the destination vector register. The first pair from the first source register is placed into the two lowest vector elements, with subsequent pairs taken alternately from each source register.

`uint64x2_t vzip2q_u64 (uint64x2_t a, uint64x2_t b)`: Same as previous one but for lower halves.

`uint64x2_t vuzp1q_u64 (uint64x2_t a, uint64x2_t b)`: This instruction reads corresponding even-numbered vector elements from the two source vector registers, starting at zero, places the result from the first source register into consecutive elements in the lower half of a vector, and the result from the second source register into consecutive elements in the upper half of a vector, and writes the vector to the destination vector register.

`uint64x2_t vuzp2q_u64 (uint64x2_t a, uint64x2_t b)`: Same as previous one but for odd-numbered vector elements.

`uint64x2_t vtrn1q_u64 (uint64x2_t a, uint64x2_t b)`: This instruction reads corresponding even-numbered vector elements from the two source vector registers, starting at zero, places each result into consecutive elements of a vector, and writes the vector to the destination vector register. Vector elements from the first source register are placed into even-numbered elements of the destination vector, starting at zero, while vector elements from the second source register are placed into odd-numbered elements of the destination vector.

`uint64x2_t vtrn2q_u64 (uint64x2_t a, b)`: Same as previous one for odd-numbered vector elements.

Chapter 2

State of the Art

The first half of this chapter explains the details of the constant weight encoding problem and the different known strategies to approach it: from the exact but inefficient solution given by means of combinatorial number systems moving towards improved strategies such as the one implemented in LEDAcrypt. Relevant works have been done by Sendrier, which in [27] proposes an approximated but highly efficient solution and then by Heyse who, in [14], takes advantage of Sendrier's idea to obtain a constant weight encoder implementation which is feasible for a microcontroller.

The second half concerns the optimizations which can be apply specifically to the implementation of code-based cryptosystems employing low density parity check codes, as mainly suggested by the work [9] of Gueron and Drucker.

2.1 Constant Weight Encoding Techniques

The constant weight encoding problem concerns the conversion of an arbitrary binary string s into another binary string e characterized by length n and weight t, which in the LEDAcrypt case represent respectively the code length and the code error correction capacity. In other words, a constant weight encoder (CWE) is responsible for the generation of a bijective mapping from random binary strings to the set of weight t, n bit long strings, which can be thought of as elements of the set of $\binom{n}{t}$ combinations.

While in the source coding domain the challenge is how to obtain high computational efficiency, in the applied cryptography field a fundamental aspect to consider is also how the constant weight strings are distributed: since the bijective property should hold for the desired encoding, the output value distribution must match the input one, which is a uniform distribution over all the possible input binary strings. Hence, using the CWE algorithm for cryptographic purposes, it is critical to be able to produce a uniform distribution of the asserted bits over the output strings. Therefore, the main characteristics that the CWE should have are:

- efficient direct and inverse computations,
- uniform distributions of the t ones over all the n positions,
- low failure rate (ideally zero).

How to efficiently perform constant weight encoding is a problem of significant interest in the field of syndrome-based cryptosystems.

Considering $\mathtt{s}$ as the binary representation of a positive integer number, the straightforward approach is to create a mapping between the input and combinatorial number systems. Such conversion requires a computational effort almost equivalent to the computation of $\binom{n}{t}$, which severely impacts LEDAcrypt performance when calculated with its designated n and t. A second encoding technique trades off precision with performance by exploiting the Golomb's compression strategy. Both procedures are illustrated next.

2.1.1 Combinatorial Number System

The exact solution to the constant weight issue, proposed by Cover in [7], is combinatorial and consists in indexing each element of $W_{n,t}$, which is the set of binary words of length n and weight t, with an integer in the interval $[0, \binom{n}{t} - 1]$. In other words, within this approach each binary input string $\mathtt{s}$, interpreted as its natural binary encoding i, defines a unique combination of binomial coefficients $(c_0, c_1, ..., c_{t-1})$. Thus for every input exists a unique sequence of coefficients where each integer c_i indicates the position of one asserted bit out of t in the fixed weight output string within a little-endian convention. Distinct numbers correspond to distinct combinations which are produced in lexicographic order: the mapping from an integer i to its associated combination $(c_0, c_1, ..., c_{t-1})$ is defined as ranking, while its inverse procedure, $i = \binom{c_0}{1} + \binom{c_1}{2} + \cdots + \binom{c_{t-1}}{t}$, is called unranking, since the rank

Algorithm 2.1.1: ExactConstantWeightEncode($\mathtt{s}$)

```
Input: s: bitstring to be encoded, ToInteger(s) < $\binom{n}{t}$
Output: e output constant weight vector of length $n_0p$ and weight $t$
Data: $n = n_0p$: length of the constant weight output word,
      $t$: weight of the output word,
      c: vector storing the extracted coefficients
$v \leftarrow$ ToInteger(s)
if $v = 0$ then
    for $i \leftarrow 0$ to $t-1$ do
        $c_i \leftarrow i$
    return IdxSeqToConstantWeightWord(c)
if $v = 1$ then
    for $i \leftarrow 0$ to $t-2$ do
        $c_i \leftarrow i$
    $c_{t-1} \leftarrow t$
    return IdxSeqToConstantWeightWord(c)
$b \leftarrow 1$ // $b$ accumulates the binomial value, starts at $\binom{t}{t}$
$k \leftarrow t-1$
while $b < v$ do
    // Estimate the largest binomial-choose-$t$ smaller than $v$
    $k \leftarrow k+1$
    $b \leftarrow b \times \frac{k+1}{k+1-t}$ // $b$ is updated to $\binom{b+1}{t}$
$c_{t-1} \leftarrow k$
$i \leftarrow t$
$b \leftarrow b \times \frac{k+1-t}{k+1}$ // roll back $b$ to the largest $\binom{b}{t}$ smaller than $v$
$v \leftarrow v - b$
if $v \neq 0$ then
    $i \leftarrow i-1$
    while $i > 0$ do
        $b \leftarrow b \times \frac{i+1}{k+1-(i+1)}$ // $b$ is updated from $\binom{b}{k}$ to $\binom{b}{k-1}$
        while $b > v$ do
            $b \leftarrow b \times \frac{k-i}{k}$ // $b$ updated from $\binom{k}{i}$ to $\binom{k-1}{i}$
            $k \leftarrow k-1$
        $v \leftarrow v - b$
        $c_{i-1} \leftarrow k$
        $i \leftarrow i-1$
        if $v = 0$ then
            break
while $i > 0$ do
    $c_{i-1} \leftarrow i-1$
    $i \leftarrow i-1$
return IdxSeqToConstantWeightWord(c)
```

of a combination corresponds to the index i which uniquely identifies it in the set $W_{n,t}$.

Given a positive integer, the procedure outlined in Algorithm 2.1.1 determines which permutation is the one associated with the input over the lexicographically ordered set of all the possible $\binom{n}{k}$ ones. The algorithm starts by searching for the highest binomial coefficient c_{t-1} which represents the position of the last `1` in the output string, proceeds by iteratively identifying the other $t-1$ coefficients and finally generates a binary string of length n with t ones in the positions indicated by each coefficient. The detailed procedure is explained next.

The first step consists in considering the input bitstring $\mathtt{s}$ as the natural binary representation of an integer value v from which the unique combinations of t asserted bits will be generated. Since combinations are generated in lexicographic order, in lines 1-10 a value v equals to 0 or 1 is directly associated respectively with $(0,1,...,t-2,t-1)$ and $(0,1,...,t-2,t)$ combinations, which are in fact the first combinations in the ordered set. For input integer v greater than one, the computation of the corresponding combination starts from the estimation of the largest binomial b such that $b \leq v$ (lines 11-15). This is achieved by exploiting the following incremental relations: $\binom{n}{k+1} = \binom{n}{k}\frac{n+1-(k+1)}{k+1}, \binom{n+1}{k} = \binom{n}{k}\frac{n+1}{n+1-k}$. In the correlated parameter k, updated at each iteration, is accumulated the value of the first coefficient c_{t-1} which represent the farthest one position (lines 14 and 16). If $b \neq v$, the algorithm continues to search for the next coefficients through sequential updates of b and k, until the closest binomial to the decreased value of v is found (lines 17-29). Specifically, in lines 24-26 the binomial is decreased from b to $\binom{k}{i}$ to $\binom{k-1}{i}$ until the next coefficient c_{i-1} is derived. When found, the v value is decreased again by b and the number of remaining coefficients i decremented (lines 26-29). This process is iterated until i or v are null. Finally, the missing i coefficients are selected starting from $i-1$ in descendent order. The obtained coefficients represent the t positions of the asserted bits of the generated constant weight bitstring $\mathbf{e}$.

The inverse procedure, given the coefficients combination $(c_t,...,c_2,c_1)$, consists in the computation of the associated rank $i = \binom{c_0}{1}+\binom{c_1}{2}+\cdots+\binom{c_{t-1}}{t}$ and in its following conversion into the natural binary representation.

Although being an optimal source coding technique, this strategy has a quadratic complexity in the input length, which is $log_2\binom{n}{t}$. For this reason, as said earlier, this solution has a negative impact on LEDAcrypt performances for which n is in the range of ten of thousands and t in the hundreds range.

2.1.2 Golomb Coding

In [27], Sendrier presents a new algorithm for encoding information in words with fixed length and weight which has linear complexity in $log_2\binom{n}{t}$, which is the input length. This is obtained through a variable length encoding at the cost of a small loss of (information theoretic) efficiency.

The idea is to consider the arbitrary binary input string as the result of a very efficient run length encoding of a sequence of integers. In particular, it is thought as encoded with Golomb's method which is known for its very high efficiency. For this reason a large number of arbitrary binary strings can be considered as valid Golomb encoded strings.

A run length encoding algorithm achieves a lossless data compression by simply concatenating the efficient encodings of the lengths of the different symbols runs, of course longer the lengths higher the compression. For example, the string `AAAAAAAABBBAAABBBBBBBBBBBBBBA`, given that it is composed of `A` and `B` symbols, will be encoded into `8A3B3A14B1A`, allowing to store only 11 symbols instead of 29.

The output of the constant weight encoding, which is assumed as the input of the decoding method by Golomb, is a binary string characterized by the fact that a symbol is far more frequent than the other and thus it is encoded as the sequence of the lengths of the runs of the most common symbol. In particular, given the characteristics of the error vectors, the `0` symbol is by far more common than `1`. Denote with p the probability $\Pr(x = 0)$ where x is the random variable modelling the values of the bit string and with l the length of a run of zeroes. The probability of a run of length l is $p^l(p-1)$, i.e. the random variable modelling the bit values of the string at hand is modelled as a geometric distribution. The idea is to encode the length of these runs as the multiple of a value d, which depends on the symbols distribution. With the aim of achieving the best compression, by encoding as much information as possible in the minimum number of bits, it is fundamental to choose a proper value for the d parameter. For this reason, d can be approximated to the integer rounding of the median of the distribution of the zero runlengths. If we consider the input string as a sequence of samples from a Bernoulli process where the random variable x represents the value of each bit in the input sequence, e.g. $x = 0$ being a successful event and $x = 1$ the unsuccessful one, and set $P(x = 0) = p$, then d should verify the following inequality:

$$p^d + p^{d+1} \leq 1 < p^{d-1} + p^d.$$

Given an estimate for the value of d, the second step of Golomb's algorithm divides the length of each run of `0`s by d, obtaining a quotient q and a remainder r. In order to best compress the information, every run is encoded as the concatenation of the unary representation of q (which will require a few bits if d is a good estimate) and the truncated binary representation of r, which is the densest, prefix-free encoding for r. It has been arbitrarily chosen to encode the quotient with a sequence of `1` symbols terminated by a single `0` bit which is used as a separator between the two values of q and r. Note that the alternative convention, i.e. employ a sequence of zeroes with a `1` stopbit, is also a valid one. The value of r is instead encoded with the so-called *truncated binary encoding* which is a prefix-free encoding, meaning that there is not a whole output sequence which is a prefix of any other one. Given an integer x to be encoded such that $0 \leq x < n$, where n is the size of the source alphabet, let k be $\lfloor \log_2 n \rfloor$ (i.e. $2^k < n < 2^{k+1}$) and $u = 2^{k+1} - n$:

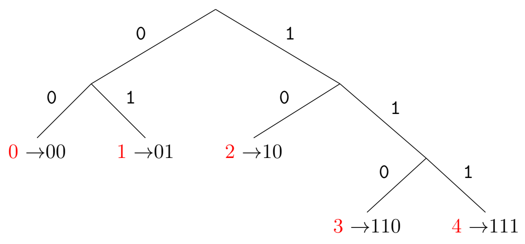
- if $x < u$ then x is encoded in its natural binary encoding using k bits;
- else if $x \geq u$ then x is encoded as the natural binary encoding of $x + u$ using $k + 1$ bits.

So truncated binary encoded words will have length of k or $k+1$ bits (variable length encoding) based on the starting value. An example of how each element can be represented when $n = 5$ can be found in Table 2.1a.

The optimal value for d, that is the one which guarantee the maximum compression and thus which allow achieving the highest efficiency, can be determined only by means of an exhaustive search on every possible d value for each possible run length, which is a highly computational demanding process. As stated in Golomb's paper, choosing d such that $p^d \approx \frac{1}{2}$ holds, where p^d is the probability for any string of zeros to have length greater

decimal value	natural binary	truncated binary
0	0	00
1	1	01
2	10	10
3	11	110
4	100	111

(a)



(b)

Figure 2.1: The left table shows how each element is represented in standard and truncated binary form, when the source alphabet is $\{0, 1, 2, 3, 4\}$ and the associated parameter are $n = 5, k = 2,$ and $u = 2^{2+1} - 5 = 3$. The right tree highlight the prefix-free property of the same code.

or equal to d, allows to achieve the string compression within a very little penalty with respect to the optimal d choice. Hence, within this choice of $d = \lfloor -\frac{1}{log_2(p)} \rfloor$, a run of length $l = d + n$ is only half as likely as a run of length $l = n$:

$$\Pr(l = n) = p^n(1 - p)$$
$$\Pr(l = d + n) = p^{d+n}(1 - p) = \tfrac{1}{2}p^n(1 - p)$$
$$\Pr(l = d + n) = \tfrac{1}{2}\Pr(l = n)$$

Thus it follows that the dense encoding for the $d + n$ run is expected to be just one bit longer than the codeword for runlength d.

For the sake of clarity, the following lines outline the steps of the encoding procedure of a single run with a running example.

1. First it is necessary to estimate d, a parameter representing the `0`'s density in the string;

2. then each length l of `0`s is divided by d, obtaining a quotient q and a remainder r;

3. finally the encoding is the concatenation of the unary representation of q, a single `0` digit and the truncated binary representation of r.

Example 2. *Given the bitstring 0000000000010000000 to be encoded with $d = 6, n = 11$, each run produces the following values:*

- $q_1 = \frac{l}{d} = \frac{11}{6} = 1$ *with* $r_1 = 5$

- $q_2 = \frac{l}{d} = \frac{7}{6} = 1$ *with* $r_2 = 1$.

The parameter for the truncated binary representation are $k = \lfloor log_2(d) \rfloor = 2$ *and* $u = 2^{k+1} - d = 2$*, being* $r_1 > u$ *it is encoded as the binary representation of* $r_1 + u = 7$*. Thus, the final Golomb encoded string is the concatenation of the encoding of* q_1, r_1 `10111` *and the encoding of* q_2, r_2 `1001`*, thus* `101111001`*.*

In [27] Sendrier exploits Golomb's efficient compression method which allows him to achieve a constant weight encoding using the minimum number of bits possible. We now describe the approach to the estimation of d proposed by Sendrier. The probability of a run of zeroes of length l, $0 \leq l \leq n-t-1$, of appearing in the CW string is equal to $P_l = \frac{\binom{n-l-1}{t-1}}{\binom{n}{t}}$ since there are $\binom{n}{t}$ possible constant weight strings and only $\binom{n-l-1}{t-1}$ ways to place the remaining $t-1$ ones in the places left free by the l-long run of zeroes. Therefore the probability of l exceeding the set threshold d is then $\Pr(l > d) = 1 - \sum_{l=0}^{d-1} P_l = \frac{\binom{n-d}{t}}{\binom{n}{t}}$. Sendrier states that choosing $\Pr(l > d) \approx \frac{1}{2}$ maximizes the amount of dense strings which given a length can be encoded into constant weight ones. The ideal minimum length of a compressed sequence requires the bits for the truncated binary representation of r plus zero or one bit for the unary representation q: this means that d should be set in a way that either every runlength is smaller than d itself, thus requiring zero bit for q, or not larger than $2d$, thus requiring exactly one bit for q. In other words, on average the length of the runs should be half of the time shorter than d and the remaining half just a bit longer.
Since encoding a run-length $l = q \cdot d + r$ will generate a string of length $\lfloor log_2(d) \rfloor + 1 + q$ bits if $r \leq 2^b - d$, otherwise $\lceil log_2(d) \rceil + 1 + q$, the expected length of an encoded generic run-length l as a function of d is

$$\mathbb{E}[L(d)] = \sum_{q=0}^{\lfloor l/d \rfloor} \left(\sum_{r=0}^{(2^b-d)-1} (\lfloor \log_2(d) \rfloor + 1 + q) \Pr(l = q \cdot d + r) + \sum_{r=(2^b-d)}^{d-1} (\lfloor \log_2(d) \rfloor + 2 + q) \Pr(l = q \cdot d + r) \right)$$

$$= \sum_{q=0}^{\lfloor l/d \rfloor} \Bigg((\lfloor \log_2(d) \rfloor + 1 + q) \sum_{r=0}^{(2^b-d)-1} \Pr(l = q \cdot d + r) +$$
$$(\lfloor \log_2(d) \rfloor + 2 + q) \sum_{x=0}^{d-1-(2^b-d)} \Pr(l = q \cdot d + (2^b - d) + x) \Bigg)$$

which can be simplified in the equivalent form

$$\mathbb{E}[L(d)] = \lfloor \log_2(d) \rfloor + 1 + q + 0 \cdot \sum_{q=0}^{\lfloor l/d \rfloor} \Pr(l \leq q \cdot d + (2^b - d)) + 1 \cdot \sum_{q=0}^{\lfloor l/d \rfloor} \Pr(l > q \cdot d + (2^b - d))$$

$$= \lfloor \log_2(d) \rfloor + 1 + q + \sum_{q=0}^{\lfloor l/d \rfloor} \Pr(l > q \cdot d + (2^b - d))$$

since the first $\lfloor \log_2(d) \rfloor + 1 + q$ bits will always be present and the additional last bit depends on the value of l. Assuming $\Pr(l > q \cdot d + (2^b - d)) \approx \frac{1}{2}$, the previous equation can be further reduce to $\mathbb{E}[L(d)] \approx \log_2(d) + \frac{1}{1-\Pr(l > qd + (2^b - d))}$, which is minimized by

$$d = \frac{n}{2t \ln(2)}, \tag{2.1}$$

as stated in Sendrier's publication [27]. This choice of the parameter reaches an encoding efficiency greater than 98% which is measured as the entropy of the constant weight string, considered as an uniform pick over $\binom{n}{t}$, divided by the average length of the dense encoding $\mathbb{E}[L(d)]$.

To measure the efficiency of the encoding, we consider the quantity η obtained dividing the length of the dense string by the entropy of the source of CW strings endowed with a random distribution over all the possible CW strings themselves: picking $d = \frac{n}{2t \ln(2)}$ results in $\eta \geq 98\%$.

In order to simplify the compression and decompression procedures, it is possible to choose d as the closest power of two to $\frac{n}{2t \ln(2)}$, i.e. $d = 2^{\lceil log_2 \frac{n}{2t \ln(2)} \rceil}$ or $d = 2^{\lfloor log_2 \frac{n}{2t \ln(2)} \rfloor}$. Notwithstanding this approximation the efficiency achievement is substantially unaltered ($\eta \geq 98\%$).
The computation of the d parameter can be performed just once at the beginning of the algorithm or can be re-estimated after the encoding of each run and thus adapted to the remaining coverable length and weight of the

Algorithm 2.1.2: ApproximateConstantWeightEncode(s)

```
Input: s: encoded bitstring of length $l$
Output: v: constant weight vector of length $n_0p$ and weight $t$
Data: x: vector containing the positions of the set bits,
      dist: vector containing the lengths of zero runs among set bits,
      $d$: density of zeros in s,
      $q$: value of unary-encoded quotient,
      $r$: value of truncated binary encoded remainder.
1  for $i \leftarrow 0$ to $t-1$ do
2      $(q, c) \leftarrow$ ExtractNextQ(s, $c$) // Counts the number of 1-bit until a 0-bit is
          found or $c = l$; if there are not enough unread bits left in s, pads it
          with 0-bits
3      $(r, c) \leftarrow$ ExtractNextRPadded(s, $c$) // Converts next $u$ or $u-1$ bits from
          truncated binary representation to the integer value $r$; if there are
          not enough unread bits left in s, pads it with 0-bits
4      dist[$i$] $\leftarrow q \cdot d + r$
5  if $c < l$ then
6      return ExcessInputFailure
7  x[0] $\leftarrow$ dist[0]
8  for $j \leftarrow 1$ to $t-1$ do
9      x[$j$] $\leftarrow$ x[$j-1$] + dist[$j$] + 1
10     if x[$j$] $\geq n_0p$ then
11         return OutBoundFailure
12 return v $\leftarrow$ SetCoefficients(x)
```

output. When choosing an adaptive computation of d, which makes sense only in case the run lengths are dependent one on another (as it is the case when they are part of a constant-length, constant-weight string), Sendrier suggests to set the new d equal to the largest power of two lesser of equal to $\lambda\frac{n}{t}$ where λ is a value representing the transformation coefficient between the natural and the base two logarithms. Depending on n and t, λ needs to be tuned, by trying different approximation, and according to Sendrier is best when between 0.6 and 0.7. Practical examples of high efficiency encodings, $\eta \geq 99\%$, are given with $\lambda = 0.67$.

LEDAcrypt implementation of the CWE exploits Golomb's decoding procedure to construct the t-weighted string of length n_0p from the input bistring `s`. In Algorithm 2.1.2 it is possible to identify the three main parts which are iterated for each of the t encoded runlength: 1. extract the unary value of the quotient q and convert it in its integer value (line 4); 2. extract and decode the value of the remainder r (line 5); 3. compute the position of the `1` bit which ends the `0`-run (lines 6).

As previously explained achieving an exact mapping between arbitrary binary string and constant weight ones significantly increases the execution time required by the function. Hence Algorithm 2.1.2 trades off precision

Algorithm 2.1.3: ApproximateConstantWeightDecode($\mathbf{v}$)

```
Input: v: constant weight vector of length $n_0 p$ and weight $t$.
Output: s: encoded bitstring.
Data: dist: vector containing the lengths of zero runs among set bits,
      $d$: density of zeros in s,
      $q$: value of the quotient,
      $r$: value of the remainder.
1 dist ← ExtractRuns(v) // Returns a vector of $t$ 0-runlegths
2 for $i \leftarrow 0$ to $t-1$ do
3     $q \leftarrow \frac{\mathbf{dist}[i]}{d}$
4     s ← WriteUnary(s, $q$) // WriteUnary(B,$x$) appends the unary
          representation of $x$ to the bitstring B
5     s ← Write(s, 0) // Write(B,$b$) appends the bit $b$ to the bitstring B
6     $r \leftarrow \mathbf{dist}[i] \bmod d$
7     s ← WriteTruncatedBinary(s, $r$) // WriteTruncatedBinary(B,$x$) appends
          the truncated binary representation of $x$ to the bitstring B
8 return s
```

with efficiency: not all the string received in input by the CWE can be seen as Golomb encoded and thus the process of CWE (which correspond of a Golomb decoding) might fail. The actual algorithm can fail for two different reasons: the approximated value of the d parameter cause the algorithm failure by decoding a length which would place a set bit out of the fixed length, constant weight string, `LengthOutOfBounds`, or a prefix of the input string can be decoded as t run lengths already, thus leaving some of the input bits of the dense string out of the information encoded in the constant weight one, `InsufficientInfoEncoded`. These are issues not solvable within the CWE function, hence the cryptosystem only solution is to restart the encryption process in order to obtain a new random seed and thus a new CWE input value (see Algorithm 1.4.1 for further details).

Algorithm 2.1.3 reports the Constant Weight decoding procedure which recovers the dense binary string starting from a constant weight one. The algorithm retrieves the length of the t `0`-runs and store them in the **dist** vector. Each of these is then Golomb-encoded into the corresponding quotient and remainder representations and appended to the output bitstring `s`.

Note that in both the description of Algorithm 2.1.2 and 2.1.3 the d parameter is estimated only once before the algorithm is run. In the CWE procedure is possible to adopt an adaptive d solution in which the estimate of d is recomputed before each iteration relying on the number of ones left to be placed *ones* and the remaining available positions *pos* in the constant weight string. Obviously, in this case, the same estimates of d have to be adopted also in the CWD algorithm: at the beginning of each i-th iteration d is recomputed based on *pos* and *ones* and at the end of the same iteration both these value should be updated as $pos = pos - \mathbf{dist}[i] + 1$ and $ones = ones - 1$.

In Chapter 3 we will propose a method to quantify the extent of the CWE failures both in terms of output uniformity and failure rate, and delineate possible mitigation strategies.

2.2 Low-Reiter

This section describes an implementation of the CWE algorithm which aims at further optimizations in order to run on a constrained device.

In [14], Heyse investigates the efficient implementation of the Niederreiter scheme on very constrained micro controllers and compares it with alternative public key schemes, such as RSA and ECC. The chosen micro controller is an 8-bit AVR ATxMega256, which is suitable for many embedded system applications. This particular choice is feasible since operations on binary codes do not require computationally expensive multi-precision integer arithmetic as in cryptosystems based on the discrete logarithm or the factorization problems.

On the other hand, it has to be taken into account that the main downside of code-based cryptography is the large keys size, which of course becomes a significant problem when working with limited memory as in the chosen micro controller. In fact, the parameters chosen for Low-Reiter implementation (Goppa code with $m = 11, n = 2048, k \geq 1751, t = 27$), which guarantee the same security level of an 80 bit key size in a symmetric cipher, generate a key size of 374 kB, which is too large to be efficiently stored inside the chosen hardware. In addition it is necessary to use secure on-chip key memories, whose cost increase with the number of bits to shield, to guarantee the secret key protection. By doing so, the key would be revealed only in case of an invasive attack on the chip itself.

To mitigate these issues, as explained in section 1.3, Niederreiter's scheme disguises the structure of the code by mean of an obfuscating matrix $\mathbf{S}$ which is multiplied by the public parity check matrix $\mathbf{H}$. A convenient solution is to use a PRNG to generate $\mathbf{S}$ at runtime and proceed to store only the PRNG

Algorithm 2.2.1: BIN2CW(n, t, δ, B)

```
Input: $n$, $t$: code parameters,
       $\delta$: $i$-th distance (0 in first call),
       B: binary stream.
Output: $t$-tuple of Integers.
if $t = 0$ then
    return
else if $n \leq t$ then
    return $\delta$, Bin2CW($n-1, t-1, 0$, B)
else
    $d \leftarrow$ BestD($n, t$)
    if Read(B, 1) $= 1$ then
        // Read(B, $i$) reads the next $i$ bits from B as integer
        return Bin2CW($n-d, t, \delta+d$, B)
    else
        $i \leftarrow$ decodeFd($d$, B) // decodeFd($d$, B) decodes from truncated binary
        return $\delta + i$, Bin2CW($n-i, t-1, 0$, B)
```

Algorithm 2.2.2: BIN2CWSMALL(n, t, B)

```
Input: $n$, $t$: code parameters,
       B: binary stream.
Output: $\mathbf{x}$: vector of distances between ones, which are also the length of the 0 runs.
Data: $\mathbf{u}_{tab}$: precomputed table containing for each pair of indexes the associated $u$ value.
for $i \leftarrow t-1$ to 0 do
    $\mathbf{x}[i] \leftarrow 0$
    if $n \leq i$ then
        $n \leftarrow n-1$
    else
        $u \leftarrow \mathbf{u}_{tab}[n - (n \bmod 2^5)][i]$
        $d \leftarrow 2^u$
        while Read(B, 1) $= 1$ do
            // Read(B, $i$) reads the next $i$ bits from B as integer
            $n \leftarrow n-d$
            $\mathbf{x}[i] \leftarrow \mathbf{x}[i] + d$
        $r \leftarrow$ Read(B, $u$)
        $\mathbf{x}[i] \leftarrow \mathbf{x}[i] + r$
        $n \leftarrow n-r$
```

seed and **H** in the flash memory of the AVR.

Using a Niederreiter cryptosystem, it is necessary to find a way to encode an arbitrary input string into the error vector, i.e. to perform a constant weight encoding. Heyse starts from Sendrier's approach (see Algorithm 2.2.1) proposed in [27] and improves it to speed up the encoding. Although its recursivity could hide it, this algorithm, essentially reads, from the Golomb encoded bitstring B, the unary representation of the quotient (lines 7-8), the truncated binary value of the remainder (lines 9-11) and compute the distance of the just selected bit from the previous one. The procedure BESTD estimates the d parameter following the methods described in Section 2.1.2.

This process continues until all asserted bits have been selected or until the number of remaining 1s is greater than the available bits in the n-length output string (lines 1-4).

Heyse is the first to approach the constant weight encoding problem with the aim of obtaining an efficient implementation for constrained devices. His improved algorithm is outlined in Algorithm 2.2.2. In order to simplify and speedup the computation of the d parameter, which contains a logarithm and a division (see Equation 2.1), he approximates the value of d itself with the closest power of two, $u = \lfloor log_2(d) \rceil$, which is memorized in the lookup table $\mathbf{u}_{tab}$ (lines 6 and 7). To precompute said table, Heyes run the original algorithm multiple times with random input strings and detected the most likely behaviour of n and t during the recursion and the minimal amount of bits that can be encoded in the error vector of length n. The table, which is indexed by the concatenation of the upper seven bits of n and all five bits of t, contains 4096 elements representing different values of u. This allows to compute $d = 2^u$ online (line 6), thus trading off computation time and memory space. It is possible to ignore the lower five bits of n since these bits are significant for the computation of the d parameter only in the case of small t and large n which happened rarely in the initial observation of the parameter distributions. In this way the table size is reduced to only 4KiB. Hence, Heyse's approach further approximates the encoding by working within a subsampling of Sendrier's original function. In lines 8-10, the value of the quotient, which is in unary representation, is computed and stored in δ as $quotient \cdot d$. In line 11, the DECODEFD procedure, which recovers the remainder part of the encoded distance, has been substituted with a simpler READ of exactly u bits, which is the expected length of the remainder. The final position of the asserted bit is given by the addition of r to the previously computed δ.

By adopting the previously explained CWE strategy to simplify the computation and storing already precomputed matrices and lookup tables, Heyse has been able to implement a Niederreiter cryptosystem on a AVR microcontroller. Furthermore, its performances outperforms comparable implementations of single core ECC cryptosystems in terms of data throughput.

2.3 Software Optimization

In [9], Drucker and Gueron list possible software optimizations on QC-MDPC cryptographic primitives and methods for side channel protection of the implementations. The considered attackers are traffic analysis eavesdroppers and spy program adversaries, both obtaining information with absolute accuracy. This means that the produced code should have a constant time implementation, i.e. the execution time of different steps, the timing and the memory access patterns should not reveal any secret information. By exploiting vectorized processor architectures they accomplish so and also obtained an improved algorithm with reference to the alternative open source libraries. A QC-MDPC code-based cryptosystem has four fundamental primitives: a constrained pseudorandom bits stream generator, an hash function, a polynomial arithmetic and a decoding algorithm. In the following paragraphs it will be explained how each of these can be improved.

Constrained Pseudorandom Bitstream Generator The authors suggest to resort to AES-CTR-PRF, which uses the block cipher AES-256: a 256 bit seed is used as the cipher key and CTR mode is exploited to populate an array of bytes, of given length, with pseudorandom values. This choice is very efficient on modern processors which have dedicated AES instructions, especially when computations can be pipelined and parallelized.

Generating a pseudorandom bitstream with a predefined weight w is essentially the same as generating w positions of asserted bits in the output, each one with value between 0 and the length l of the produced bitstring. This means that positions exceeding l will be simply discarded: notice that accepting any value by reducing it modulo l will not result in a uniform random distribution, since smaller values will be more frequent.

Using this method, the expected number of samples needed to fill w positions is $2w$, being the rejection probability $p = 1 - \frac{l}{2^{\lceil log_2(l) \rceil}} < \frac{1}{2}$. Note that starting from a zero valued string, flipping the bits in the relevant positions is not secure against someone observing the memory access pattern. A solution to this issue is outlined in Algorithm 2.3.1, where every bit is rewritten but only the selected ones flipped.

Algorithm 2.3.1: ApplyWList($\mathbf{w}_{list}$)

```
Input: $\mathbf{w}_{list}$: list of $w$ positions $\in [0, l-1]$ of the asserted bits.
Output: $\mathbf{a}$: vector of length $l$ and weight $w$.
1 $\mathbf{a} \leftarrow 0$
2 for $i \leftarrow 0$ to $len - 1$ do
3     for $j \leftarrow 0$ to $w$ do
4         $\mathbf{a}[i] \leftarrow \mathbf{a}[i]$ or isEqual$(i, \mathbf{w}_{list}[j])$
5 return $\mathbf{a}$
```

Efficient Hashing In code-based cryptosystems hash functions are used to compress a random bitstring into a fixed length one such as seeds or secret keys. The suggested strategy consists in converting a serial hash function h, with digest length of l_d bytes working on h_{bs} bytes each time, into a parallelizable process. h receives an array of l_a bytes as input which is split into s contiguous disjoint slices of length l_s and eventually in a remainder y. The length of each slice is computed as $l_s = \alpha h_{bs} + s_{rem}$, where s_{rem} is the pre padding length and $\alpha = \lfloor \frac{\lfloor \frac{l_a}{s} - s_{rem} \rfloor}{h_{bs}} \rfloor$. Then the s slices can be hashed through h in parallel obtaining

$$\mathbf{x}[j] = h(\mathbf{slices}[j]) \text{ with } j = 0, ..., s-1$$

and finally the output is

$$h(y||\mathbf{x}[s-1]||\mathbf{x}[s-2]||\cdots||\mathbf{x}[0]). \quad (2.2)$$

Polynomial Arithmetic Today general-purpose processors are equipped with a carry-less multiplication instruction which allows multiplication between two binary polynomials. For example, ARM processors equipped with advanced SIMD vector instruction can leverage the `PMULL` instruction which performs polynomial multiplications between the higher or lower halves of two 128-bit registers.

For particular high degree polynomials, as the one used in QC-MDPC code-based cryptosystems, the authors suggest exploiting the Karatsuba multiplication which computes products by working on halves of the original factors and thus recursively performing a sequence of smaller multiplication and addition.

Within the *Karatsuba-Ofman* multiplication [16], given 2^l-bits integer $x = x_1 2^l + x_0$ and $y = y_1 2^l + y_0$, $x \cdot y$ can be computed by performing

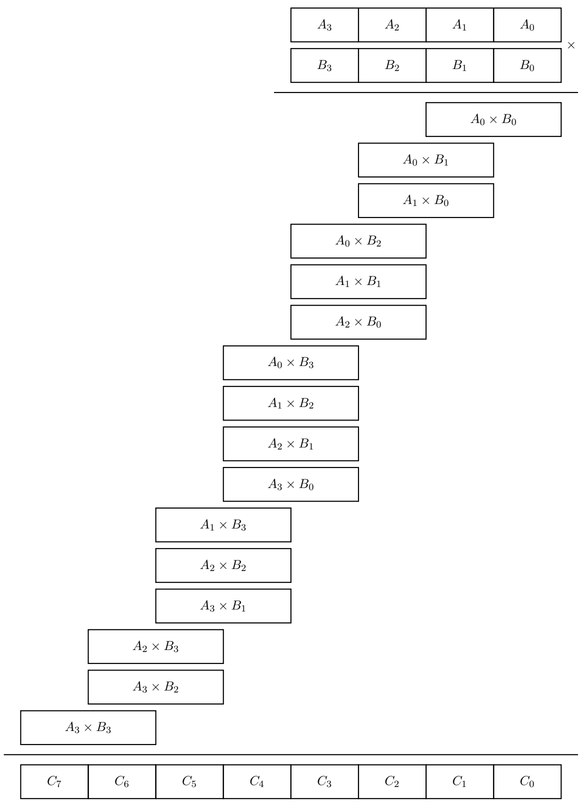


Figure 2.2: Execution of the vertical variant of a 4×4 schoolbook multiplication.

three multiplications of l-bit integers instead of one multiplication with $2l$-bit integers:

$$x \cdot y = (x_1 2^l + x_0)(y_1 2^l + y_0) = x_1 y_1 2^{2l} + [(x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0] 2^l + x_0 y_0$$

For very large values of l, the cost of performing several additions and subtractions is lower than the cost of performing a multiplication. Obviously, the final step of this recursive algorithm performs schoolbook multiplications and below a certain input size it remains the best choice.

Drucker and Gueron chose to exploit the formula proposed by Karatsuba recursively, up to the point where the operands are constituted by two architectural digits, and then use the vertical variant of the schoolbook multiplication, which is carried out in Figure 2.2, as last step.

Denote with $\mathbf{p} = p_{r-1} p_{r-2} \cdots p_0$, where p_i is the bit in position i of $\mathbf{p}$,

Algorithm 2.3.2: BITFLIP(**x**,**H**)

```
Input: $\mathbf{H} \in \mathbb{F}_2^{r\times n}$: parity check matrix,
       $\mathbf{x} \in \mathbb{F}_2^{n}$: codeword to be corrected.
Output: $\mathbf{e} \in \mathbb{F}_2^{n}$: error vector.
Data: $\tau$: unsatisfied parity check threshold value,
      $max$: maximal number of iterations.
1  $itr \leftarrow 0$, $\mathbf{e} \leftarrow 0$
2  $\mathbf{s} \leftarrow \mathbf{Hx}^T$
3  while $\mathbf{s} \neq 0$ and $itr < max$ do
4      for $i \leftarrow 0$ to $n-1$ do
5          $\mathbf{upc}[i] \leftarrow$ CountUPC($\mathbf{s}, \mathbf{w}_{list}$)
6          if $\mathbf{upc}[i] > \tau$ then
7              $\mathbf{e}[i] \leftarrow \mathbf{e}[i] \oplus 1$
8      $itr \leftarrow itr + 1$
9      $\mathbf{s} \leftarrow \mathbf{H}(\mathbf{x}+\mathbf{e})^T$
10 if $itr = max$ then
11     return Failure
12 return $\mathbf{e}$
```

the bitstring representing the polynomial $p(x) = p_{r-1}x^{r-1} + p_{r-2}x^{r-2} + \cdots + p_1x + p_0$. Let $\mathbf{p_1}, \mathbf{p_2}$ be two polynomial of degree $r-1$ padded into $q = \lceil \frac{r-1}{64} \rceil$ 64-bit containers, such that $\mathbf{p}[q-1] = \underbrace{0\cdots 0}_{64-\delta} p_{r-1}p_{r-2}\cdots p_{r-\delta}$, with $\delta = r \bmod 64$, and $\mathbf{p}[0] = p_{63}p_{62}\cdots p_1p_0$.

For each value of $k \in [0, q-1]$ it is possible to compute in parallel the intermediate values of $\mathbf{p_3}[2k+1:2k]$, due to their reciprocal independence, as

$$\mathbf{p_3}[2k+1:2k] = \textstyle\sum_{i+j=2k} \mathbf{p_1}[i] \times \mathbf{p_2}[j] \quad 0 \leq i,j,k \leq q-1$$

and then for $k = 0, ..., q-2$,

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

The other fundamental operation is the reduction modulo x^r+1 in $\mathbb{F}_2$ of a polynomial $\mathbf{p} \in \mathbb{F}_2[x]$ of degree r', $r \leq r' \leq 2r$. Exploiting modern SIMD processors this can be vectorized when computed as

$$\mathbf{p}[r-1:0] = \mathbf{p}[r-1:0] \oplus \mathbf{p}[r':r].$$

Efficient Decoding In code-based cryptosystems the decoding algorithm is responsible for the extraction of the error vector $\mathbf{e} \in \mathbb{F}_2^n$ from the syndrome $\mathbf{s} = \mathbf{Hx}^T$, where $\mathbf{H} \in \mathbb{F}_2^{r\times n}$ is the parity check matrix and $\mathbf{x} \in \mathbb{F}_2^n$ the input vector. The commonly chosen decoding algorithm is the so called BitFlip

Algorithm 2.3.3: CountUPC($\mathtt{s}$,$\mathtt{w_{list}}$)

```
   Input: $\mathbf{s} \in \mathbb{F}_2^r$: syndrome vector,
          $\mathbf{w}_{list}$: list of $w$ asserted bits positions of a column of $\mathbf{H}$.
   Output: $\mathbf{upc}$: array of $r$ bytes.
1  $\mathbf{upc} \leftarrow 0$
2  for $i \leftarrow 0$ to $r$ do
3      for $j \leftarrow 0$ to $w-1$ do
4          $\mathbf{y} \leftarrow$ RotL($\tilde{\mathtt{s}}, \mathtt{w_{list}}[j]$) // RotL($B, i$) performs a leftwise cyclic rotation
              of $B$ by $i$ bits
5          $\mathbf{upc}[i] \leftarrow \mathbf{upc}[i] + \mathbf{y}[i]$
6  return $\mathbf{upc}$
```

algorithm (Algorithm 2.3.2) which can be described by three main steps:

1. calculate the number of the n unsatisfied parity-checks $\mathbf{upc}_i$ (line 5);
2. if $\mathbf{upc}_i \geq \tau$, for some threshold τ, flip the i-th bit of the error vector $\mathbf{e}$ (lines 6-7);
3. add $\mathbf{e}$ to $\mathbf{x}$ and recompute $\mathbf{s}$ (line 9).

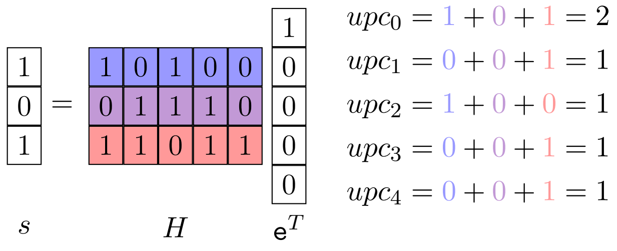
These should be repeated until $\mathbf{s}$ equals zero or in alternative for a chosen maximum number of iterations, which if reached causes the algorithm to stop, returning a decoding failure error. The specific choice of the threshold value τ leads to different Decoding Failure Rate (DFR): common options are setting it equal to a constant as in [12], to the maximum $\mathbf{upc}_i$ as in [15] or a small value below it as in [22].

As highlighted in [30], which compares several decoders that do not required polynomial multiplications, in order to update the syndrome two alternatives approaches are available: the out-of-place decoding, which performs BitFlipping by finding and updating all potential error bits in $\mathbf{e}$, i.e. those for which $\mathbf{upc}_j > \tau$, and then updating the syndrome as $\mathbf{s} = \mathbf{s} + \mathbf{h_j}$ for all the affected bits, where $\mathbf{h_j}$ is the j-th column of $\mathbf{H}$, or the in-place decoding, which updates the syndrome after each error bit is selected. Drucker and Gueron chose the out of place decoding approach.

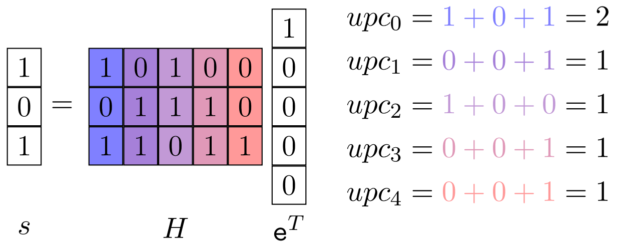
The first step of the decoding implementation is outlined in its sequential version in Algorithm 2.3.3 and in its parallelized one in Algorithm 2.3.4. The latter receives as input a list $\mathbf{w}_{list}$ representing the w asserted bits of the j-th column $\mathbf{h_j}$ of $\mathbf{H}$ and $\tilde{\mathtt{s}}$ a redundant representation of $\mathbf{s}$ expanding each bit of the syndrome into a byte, to allow the accumulation of up to 254 additions in the same syndrome ($w < 254$ always true for QC-MDPC cryptosystems).

Algorithm 2.3.4: CountUPCParallel($\tilde{\mathbf{s}}$,$\mathtt{w_{list}}$)

```
Input: $\tilde{\mathbf{s}}$: redundant representations of the $r$ bits syndrome $s$,
       $\mathbf{w}_{list}$: list of $w$ asserted bits positions of a column of $\mathbf{H}$.
Output: upc: array of $r$ bytes.
upc $\leftarrow 0$
for $i \leftarrow 0$ to $r$ by $M$ do
    for $j \leftarrow 0$ to $w-1$ do
        $\mathbf{y} \leftarrow$ RotL($\tilde{\mathbf{s}}^{\times 2}, \mathbf{w}_{list}[j]$) // RotL($B, i$) performs a leftwise cyclic
            rotation of $B$ by $i$ bits
        upc[$i : i+M$] $\leftarrow$ upc[$i : i+M$] $+ \mathbf{y}[i : i+M]$
return upc
```



(a) Horizontal Summation



(b) Vertical Summation

Figure 2.3: Example of vertical vs horizontal *upc* computation methods.

Let $\mathbf{a} = a_{r-1}a_{r-2}...a_0$ be a string of bits: the redundant representation of $\mathbf{a}$ is the array $\tilde{\mathbf{a}}$, of r bytes, where $\tilde{\mathbf{a}}[i] = \mathtt{01}$ if $a_i = 1$ and $\tilde{\mathbf{a}}[i] = \mathtt{00}$ otherwise, $0 \leq i \leq r-1$. In addition, to to speed up the rotation part in line 4, it is convenient to duplicate the vector in memory, $\tilde{\mathbf{s}}^{\times 2} = \tilde{\mathbf{s}}||\tilde{\mathbf{s}}$, this will in fact avoid any wraparound issue.

The sum of all $\mathbf{upc}_i$ is performed vertically, i.e. for each i separately as shown in Figure 2.3b, to guarantee higher efficiency (lines 3-5). In fact, being m the number of bytes which can be stored in the processor's wide-registers, when $r > m$ the syndrome $\tilde{\mathbf{s}}$ is too large to fit in these, and performing an horizontal summation, i.e. every i in parallel as shown in 2.3a, would

Algorithm 2.3.5: COUNTUPCCONSTWEIGHT($\tilde{\mathbf{s}}^{\times 2}, \mathbf{w}_{list}, \mathbf{b}$)

```
Input: $\tilde{\mathbf{s}}^{\times 2}$: redundant representations of the $r$ bits syndrome $s$,
       $\mathbf{w}'_{list}$: list of $w$ asserted bits positions of a column of $\mathbf{H}$,
       $\mathbf{b}$: flag list of length $w + w'$.
Output: upc: array of $r$ bytes.
1 upc $\leftarrow 0$
2 for $i \leftarrow 0$ to $r$ by $M$ do
3     for $j \leftarrow 0$ to $w - 1$ do
4         $b_m \leftarrow$ EXTEND($\mathbf{b}[\mathbf{w}'_{list}[j]]$) // EXTEND($x$) duplicates the single $x$-bit value
              to every bit of an $M$ bits word.
5         $\mathbf{y} \leftarrow$ ROTL($\tilde{\mathbf{s}}^{\times 2}, \mathbf{w}_{list}[j]$) // ROTL($B, i$) performs a leftwise cyclic
              rotation of $B$ by $i$ bits
6         $z \leftarrow \mathbf{y}[i : i + M]$ and $b_M$
7         upc$[i : i + M] \leftarrow$ upc$[i : i + M] + z$
8 return upc
```

required to read, accumalte and store intermediate results in some memory location for each of the w iteration. More precisely, an horizontal approach would require $2r$ memory reads plus r memory writes, thus a total of $3r$ memory operations, for each iteration. A vertical summation, instead, would accumulate the intermediate steps inside the registers and store them only in the end of each of the $\lceil \frac{r}{m} \rceil$ iterations, resulting in $w \times m$ memory read and m memory writes for each iteration and thus a total of $2w + m \times \lceil \frac{r}{m} \rceil \approx \frac{2wr}{m}$ memory operations. In the particular case of QC-MDPC, the gap between the two strategies is even wider due to the fact that $2r+m$ usually do not fit in the first level cache of the processor. Hence, adopting per bit approach allows leveraging modern CPU vector instructions which will operate in parallel on $\lfloor \frac{r}{m} \rfloor$ chunks and then handle the tail of $\lfloor \frac{r}{m} \rfloor \times m$ bytes separately. To avoid handling the tail, it is convenient to left-pad $\tilde{\mathbf{s}}^{\times 2}$ with $m - (\lfloor \frac{r}{m} \rfloor \times m)$ bytes and working on this longer array.

A second optimization which trade-off the number of iterations with DFR considers $\mathbf{w}_{list}$ with reduced weights. In other word, the algorithm cycles through only $w - \alpha$ values of $\mathbf{w}_{list}$ which are randomly chosen every time. This solution speeds up the algorithm by a factor of $\frac{w}{w-\alpha}$ but increases the DFR.

The last thing to consider for the decoding algorithm, which handles secret values and intermediate results, is the protection against side-channel attacks. To tackle this issue and prevent any information leakage is necessary to produce a constant time implementation. The authors' solution is illustrated in Algorithm 2.3.5. The first step is to choose uniformely at random

Table 2.1: Suggested w and w' values for different security level λ

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

$w' < r-w$ positions in $\mathbf{h_0}$ where the bits are not set (fake positions). The new $\mathbf{w}'_{list}$ vector will contain both position of the original w asserted bits and the w' fake ones, thus its length will be of $w + w'$, while the associated vector $\mathbf{b}$, indicates the non-fake bits positions in $\mathbf{h_0}$: $\mathbf{b}[i] = \mathbf{h_0}[\mathbf{w}'_{list}[i]], i = 0, ..., w+w'$. Provided that $\mathbf{b}$ remains confidential, $\mathbf{w}'_{list}$, unlike $\mathbf{w}_{list}$, does not have to be a secret. Thus it is possible to choose w' such as $\binom{w'+w}{w} > 2^{2\lambda}$ where λ is the chosen security bit-level (suggested values can be found in Table 2.1). With this approach a secure implementation needs to protect only operations which involve $\mathbf{b}$. Being $|\mathbf{b}| << r$, this secured implementation remains efficient: it is possible to keep in memory only the compact representation $\mathbf{w}'_{list}$ and perform only $|\mathbf{b}|$ iterations. Thus the final cost will be of $|\mathbf{b}| \times r$ memory accesses.

The straightforward approach for the third step execution recalculates the syndrome $\mathbf{s}$ of a $\mathcal{C}(n_0 r, r)$ code by means of n_0 polynomial multiplications (modulo $x^r + 1$) plus $n_0 - 1$ additions in $\mathbb{F}_2^n$. As said before, in order to avoid multiplications, it is possible to either add all the column $\mathbf{h_j}$ of $\mathbf{H}$ that correspond to error bits, or add all the columns while masking out the "unnecessary" one, to execute in constant time.

The authors have found that the latter strategy introduces significant overheads and thus it is best to adopt the straightforward implementation which consists in the syndrome recalculation by means of n_0 polynomial multiplications (modulo x^r+1) among $\mathbf{H}$ and $\mathbf{x}$ elements plus n_0-1 additions in $\mathbb{F}_2^n$.

Chapter 3

Effectiveness and Efficiency Analysis of Constant Weight Encoding

As described in Chapter 2, the constant weight encoder is a bijective function between arbitrary binary strings and strings of fixed length and weight. Due to its approximated implementation, LEDAcrypt constant weight encoder output distribution could be skewed, thus this chapter discusses this specific issue and considers possible mitigations.

3.1 LEDAcrypt Constant Weight Encoder

The implementation of the LEDAcrypt constant weight encoder which is based on Golomb's runlength encoding achieves fast computation at the price of a small amount of failures.

This procedure, briefly discussed in Section 2.1.2 and reported in greater details in Algorithm 3.1.1, in fact fails when the selected positions of the asserted bits in the decompressed string exceed the length of the output vector $\mathbf{v}$, i.e. if the sum of the length of the t generated runs of zeroes plus t is greater than n_0p (`LengthOutOfBounds`). Further failure situations arise when leveraging this algorithm inside a cryptosystem. In fact, within this in mind, it is critical for the function to be able to guarantee the encoding of a chosen amount of information, i.e. a number of bits l represented by the input string `s`, into the constant weight vector. There are two different reason for this requirement not to be met

- `InsufficientInfoEncoded`: the t `1`s positions are generated before reading all the information carried by `s`, i.e. a prefix of `s` is a valid encoding for t runs of zeroes.
- `LackOfEncodableInput`: the decompression process applied to the whole input string is not able to generate exactly t runs.

The `LengthOutOfBounds` failure is not improvable and neither is the `InsufficientInfoEncoded` one since the whole information has to be encoded and not just the first bits. However, there is a simple remedy, exploited in Algorithm 3.1.1, for the third failure situation (`LackOfEncodableInput`) which extends the input string with the amount of bits needed to complete the encoding process. In this way, when the whole input has been read, any following reads will return a zero-valued bitstring.

Algorithm Details Line 1 of Algorithm 3.1.1 sets the initial values of the variables *ones*, which will represent the number of ones still to be placed during the computation, and *pos*, keeping track of the remaining available positions in the constant weight vector as the `0`-runlengths are generated, respectively to t and n_0p.

In line 3, the EstimateD procedure returns the estimated d value which, as specified in Section 2.1.2, can be fixed or adaptively changed based on the updated values of *ones* and *pos*. Note that, if a non-adaptive estimation technique is employed to determine the value of d, the minimum length of an arbitrary bit string in input to the CWE which can be decoded as t runs can be obtained as follows: zero bits for the unary quotient representation, a single `0`-bit which behaves as stop bit for the quotient and $\lfloor \log_2(d) \rfloor$ bits necessary to represent the remainder in truncated binary, thus obtaining $l = (1 + \lfloor \log_2(d) \rfloor) \cdot t$. In other words, l is the minimum number of input bits required by the constant weight encoding whenever it's employing a fixed d estimate.

Lines 6-17, regard the proper Golomb decompression of each i-th `1` position through the decoding of the quotient q and the remainder r values. In particular, in lines 6-8 the unary representation of q is extracted by reading

Algorithm 3.1.1: ApproximateConstantWeightEncode($\mathtt{s}$)

```
Input: s encoded bitstring of length $l$.
Output: $\mathbf{v}$ constant weight vector of length $n_0 p$ and weight $t$.
Data: $\mathbf{x}$: vector containing ones position,
      $\mathbf{dist}$: vector containing the positions of the set bits,
      $d$: density of zeros in s,
      $u$: bits to encode $d$,
      $q$: integer value of the quotient,
      $r$: integer value of the remainder,
      f: read fragments of s,
      $c$: cursor (number of bits read from s).
1  $i \leftarrow 0$, $c \leftarrow 0$, $ones \leftarrow t$, $pos \leftarrow n_0 p$
2  for $i \leftarrow 0$ to $t-1$ do
3      $(d, u) \leftarrow$ EstimateD($pos, ones$)
4      $q \leftarrow 0$, f $\leftarrow 1$
5      // reading the unary encoded $q$
6      while f $= 1$ do
7          (f, $c$) $\leftarrow$ ReadPadZero(s, $c$, 1) // ReadPadZero after reading whole s
              pads with 0
8          $q \leftarrow q +$ f
9      if $u > 0$ then
10         (f, $c$) $\leftarrow$ ReadPadZero(s, $c$, $u-1$)
11         if BinToInt(f) $\geq 2^u - d$ then
12             $(y, c) \leftarrow$ ReadPadZero(s, $c$, 1)
13             $r \leftarrow 2 \cdot$ BinToInt(f) $+ y - (2^u - d)$
14         else
15             $r \leftarrow$ BinToInt(f)
16     else
17         $r \leftarrow 0$
18     $\mathbf{dist}[i] \leftarrow q \cdot d + r$
19     $pos \leftarrow pos - (\mathbf{dist}[i] + 1)$
20     $ones \leftarrow ones - 1$
21     if $pos < ones$ then
22         return LengthOutOfBounds
23 if $c < l$ then
24     return InsufficientInfoEncoded
25 $\mathbf{x}[0] \leftarrow \mathbf{dist}[0]$
26 // Converting run length into set bit positions
27 for $j \leftarrow 1$ to $t-1$ do
28     $\mathbf{x}[j] \leftarrow \mathbf{x}[j-1] + \mathbf{dist}[j] + 1$
29 return $\mathbf{v} \leftarrow$ SetCoefficients($\mathbf{x}$)
```

the input string s one bit at once until a `0` is found. In lines 9-17, the decoder reads the first $u-1$ bits of the remainder retrieving g: if its integer value is greater or equal than $2^u - d$ then it is necessary to read the additional last bit and compute r otherwise r is set to the integer value represented by the binary string g. The function ReadPadZero, exploited to obtain the values of q and r, concatenates additional `0` bits to the actual input bitstring when is employed to read out of the bounds of the input bitstring itself. In this way it possible to avoid a `LackOfEncodableInput` failure.

Then the integer value of $q \cdot d + r$, is stored in $\mathbf{dist}[i]$ which is the vector of distances between `1`s, i.e. the lengths of the `0` runs.

Lines 21 and 22, check the occurrence of a `LengthOutOfBounds` failure which arises if there are more `1`-bit left to be set than available positions in the output vector. Lines 23 and 24 instead address the `InsufficientInfo-Encoded` issue by verifying that the whole input bitstring has been taken into account in the output generation process. Finally, the variables *pos* and *ones* variables, are updated by subtracting the size of the decoded run to the former and by decrementing the latter (lines 18-20). The last step regards the conversion of the vector **dist** into the constant weight vector **v** of weight t and length n_0p (lines 25-29).

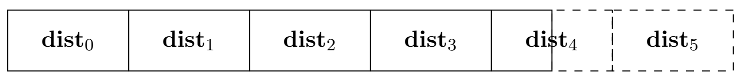
3.1.1 Tackling Output Uniformity and Failures

There are basically two features that are linked with the `LackOfEncodable-Input` and that indeed can be adjusted to mitigate this issue: the value of the d parameter and the padding strategy.

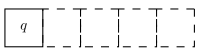
In the LEDAcrypt implementation of the CWE, d is estimated, as suggested by Equation 2.1, based on the length n and weight t of the constant weight string, thus

$$d = \frac{n-(t-1)}{2t\log(2)}. \tag{3.1}$$

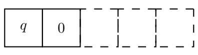
In the `fixed_d` version of the algorithm, this value, which is computed computed only once, is used as a constant during the whole computation. As said earlier, choosing this approach allow to know the exact minimum number of input bits required by the constant weight encoding, $l = (1 + \lfloor \log_2(d) \rfloor) \cdot t$, and thus to set the input string accordingly. There exists other approaches capable of achieving a better compression by tuning the d value based on the remaining coverable length and weight of the output after selecting each set bit. This is possible since the positions of the asserted bits are dependent one from the other. In particular, in the `adaptive_d` method d is recomputed before encoding each run as $d = \frac{pos-(ones-1)}{2pos\log(2)}$ where *pos* variable counts the number of available positions in the constant weight string and *ones* keeps track of the left to be set `1` bit; while in the similar `min_adaptive_d` version its value is computed as $d = \min\{\frac{n-(t-1)}{2t\log(2)}, \frac{n-t-1}{l}\}$ in the first call, where l is the minimum length of the input string to be encoded, and as the adaptive case in the following ones. The amount $\frac{n-t-1}{l}$ tunes the minimum value of



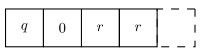
(a) Input bitstring s



(b)



(c)



(d)

Figure 3.1: Example of s which need some padding to generate the associate output for $t = 6$ (3.1a) and of different ways in which the last encoded sequence could terminate (3.1b, 3.1c, 3.1d).

d based on the input length l, decreasing in this way the number of failures compared to a fully adaptive one. Both these adaptive techniques guarantee an higher efficiency of the CWD, i.e. a better compression of CW strings into binary ones, with respect to the fixed one. On the other hand, in these situations the minimum length of the input string could be shorter than the one computed for the fixed case. In fact, the recomputation of the parameter could lead to a smaller values of d, and hence of $log_2(d)$, causing an increment in the number of `InsufficientInfoEncoded` failures. In the outlined algorithms the said d computation is performed by the ESTIMATED function.

While the choice of d is fundamental to maximize the number of arbitrary strings which can be encoded into constant weight ones, the value of the padding is responsible for both the conclusion of the last "in reading" encoded run and for the entire value of the runs still to be set after the whole input has been read. Consider the example illustrated in Figure 3.1 where $t = 6$, in this case after reading the whole string s only four runlength have been generated, therefore the last two need to be obtained through some kind of padding. Accordingly, the value of **dist**$_4$ needs to be padded using different approach based on the truncation point (represented in 3.1b, 3.1c and 3.1d) while **dist**$_5$ value can be simply decoded from a randomly generated padding.

A further issue to consider, when selecting the padding approach, is the randomness of the output distribution which will be generated. In fact, since the CWE function input space is uniformly distributed, its output space should also be uniformly distributed. Moreover, studying how the ones distribute over the positions allows us to choose the most suitable sizing for the system parameters, resulting in less failures thus gaining efficiency.

For this reason, the following paragraphs propose three different padding approaches for the CWE function:

1. pad with zeroes,

2. pad with arbitrary random bits,

3. pad with constrained random bits.

All the above mentioned strategies can be paired with any d computation choice with the purpose of selecting the best pair for the cryptosystem. In addition, to improve the efficiency of the algorithm, the last two strategies directly select at random the set bit position instead of decoding it from an arbitrary generated padding when dealing with non-truncated run length encoding ($\mathbf{dist}_5$ of the previous example).

Zero Padding With this approach, which is the one adopted in the round 1 submission codebase of LEDApkc CWE (Algorithm 3.1.1), when the whole input has been read: any following reads on it returns a zero-valued bitstring. In this way, the missing runs that are generated through padding will all have null length. In fact, each length will be represented by a sequence of zeroes: the first bit will act as quotient stop-bit, hence setting $q = 0$, then the decoding will proceed to read the following $\log_2(d)$ null bits representing the remainder, thus the computed length of the run will be $q+d\cdot r = 0+d\cdot 0 = 0$. This is the most conservative approach with the aim of avoiding a possible `LengthOutOfBounds` due to the padding strategy. It is expected that padding with zeroes will generate some skew in the distribution of the ones of the constant weight string.

Random Padding This solution attempts to improve the output distribution by incorporating some randomness into the padding step. Instead of just padding with `0` bits, the simpler way to introduce some arbitrariness is to pad the input string within an arbitrary number of random bits, generated from a DBRG seeded with a TRNG, and proceed with the decoding as

Algorithm 3.1.2: BinaryToConstantWeightSmartRnd(s)

```
Input: s: encoded bitstring of length $l$.
Output: v: constant weight vector of length $n_0p$ and weight $t$.
Data: x: vector containing ones position,
      dist: vector containing intra-ones distances,
      $d$: density of zeros in s,
      $u$: bits to represent $d$,
      $q$: value of unary-encoded quotient,
      $r$: value of truncated binary encoded remainder,
      f: read fragment,
      $c$: cursor (number of bits read from s. )
$i \leftarrow 0$, $c \leftarrow 0$, $ones \leftarrow t$, $pos \leftarrow n_0p$
for $i \leftarrow 0$ to $t-1$ do
    $(d,u) \leftarrow$ EstimateDU($pos, ones$)
    $q \leftarrow 0$, f $\leftarrow 1$
    // reading the unary encoded $q$
    while $f = 1$ do
        if $c+1 \leq l$ then
            $(\mathtt{f}, c) \leftarrow$ BitstreamRead(s, 1)  // BitstreamRead(s, $i$) reads exactly $i$
            bits from s and returns them while updating the cursor $c$ to $c+i$
            $q \leftarrow q + \mathtt{f}$
        else
            return CompleteRandomQuotient($pos, ones$, dist, $i, q$)
    if $u > 0$ then
        if $c+u-1 \leq l$ then
            $(\mathtt{f}, c) \leftarrow$ BitstreamRead(s, $u-1$)
            if BinToInt(f) $\geq 2^u - d$ then
                if $c+1 \leq l$ then
                    $(y, c) \leftarrow$ BitstreamRead(s, 1)
                    $r \leftarrow 2 \cdot$ BinToInt(f) $+ y - (2^u - d)$
                else
                    return CompleteRemainderLastBit($pos, ones$, dist, $q$, f)
        else
            return CompleteRemainderPrefix($pos, ones$, dist, s, $c, q$)
    else
        $r \leftarrow 0$
    dist[$i$] $\leftarrow q \cdot d + r$
    $pos \leftarrow pos -$ (dist[$i$] $+ 1$)
    $ones \leftarrow ones - 1$
    if $pos < ones$ then
        return LengthOutOfBounds
if $c < l$ then
    return InsufficientInfoEncoded
return RndAndUpd($pos, ones$, dist)
```

usual until all t bit are set. Note that this does not provide an impairment to the decoding of the input string from the constant weight one, as the entire information of the input is present in the constant weight output string.

Constrained Random Padding The last proposed method, outlined in Algorithm 3.1.2, is an additional improvement of the previous one, based on a "smart" randomization of the padding bits which aims at decreasing the number of `LengthOutOfBounds` failures. This time, as soon as all the l bits of $\mathbf{s}$ are read:

1. the decoded value of the truncated run length sequence ($\mathbf{dist}_4$ in Figure 3.1a) is randomly chosen only between the positions which are compatible with the last bits read while guaranteeing a successful procedure conclusion,
2. the positions of the remaining bits to be asserted ($\mathbf{dist}_5$ in Figure 3.1a) are randomly chosen among the available positions.

For readability reasons, the CWE code which deals with the padding strategy has been divided into four different sub-functions:

- CompleteRandomQuotient, which considers the case when the string terminates while reading the unary encoded quotient q (Algorithm 3.1.4);
- CompleteRemainderPrefix, which considers the case when the string terminates while reading the first u bits of the remainder (Algorithm 3.1.5);
- CompleteRemainderLastBit, which considers the case when the string terminates during the determination of the last bit of the remainder (Algorithm 3.1.6),
- RndAndUpd, which directly generates the position of the missing asserted bits of the constant weight string, if any, and converts the vector of distances into the final CW vector $\mathbf{v}$ (Algorithm 3.1.3).

It is convenient to start from the explanation of the latter procedure since it is called at the end of all the others.

The function RndAndUpd, outlined in Algorithm 3.1.3, arbitrarily selects the position of the remaining set bits into the output vector and returns the CW vector. At first, lines 1 and 2 check the value of *pos* and *ones* for an eventual LengthOutOfBounds failure. Then, having verified that there are enough positions available, if there are still bits to be set. Each `1` position is randomly chosen between the available positions (lines 3-12). In the final step, lines 13-17, the distances between `1`s obtained before the padding stage are converted to asserted bits in the output vector $\mathbf{v}$ which is then returned.

Algorithm 3.1.3: RndAndUpd($pos, ones, \mathbf{dist}$)

```
Input: $pos$: remaining available positions,
       $ones$: ones to place,
       $\mathbf{dist}$: vector containing intra-ones distances.
Output: $\mathbf{v}$: constant weight vector of length $n_0 p$ and weight $t$.
Data: $\mathbf{x}$: vector containing ones position,
      $o$: number of randomly placed ones,
      $dup$: checks if the generated value is already present in $\mathbf{x}$.
1  if $pos < ones$ then
2      return LengthOutOfBounds
3  $i = t - ones$, $o \leftarrow 0$
4  while $o < ones$ do
5      $p \leftarrow$ Rand($pos, n_0 p \cdot t$)
6      $dup \leftarrow 0$
7      for $j \leftarrow 0$ to $o - 1$ do
8          if $\mathbf{x}[i + j] = p$ then
9              $dup = 1$
10     if $dup = 0$ then
11         $\mathbf{x}[i + o] = p$
12         $o \leftarrow o + 1$
13 // Converting run length into set bit positions
14 $\mathbf{x}[0] \leftarrow \mathbf{dist}[0]$
15 for $j \leftarrow 1$ to $i - 1$ do
16     $\mathbf{x}[j] \leftarrow \mathbf{x}[j - 1] + \mathbf{dist}[j] + 1$
17 return $\mathbf{v} \leftarrow$ SetCoefficients($\mathbf{x}$)
```

Algorithm 3.1.4: CompleteRandomQuotient($pos, ones, \mathbf{dist}, q$)

```
Input: $pos$: remaining available positions,
       $ones$: ones to place,
       $\mathbf{dist}$: vector containing intra-ones distances,
       $q$: partially read quotient.
Data: $r$: value of truncated binary encoded remainder,
      $\mathbf{dist}$: vector containing intra-ones distances,
      $d$: density of zeros in s,
      $t$: weight of the constant weight output vector.
1  $v \leftarrow q \cdot d$
2  $a \leftarrow pos - v - (ones - 1)$
3  if $a < 0$ then
4      return OutBoundFailure
5  else if $a > 0$ then
6      $v \leftarrow v +$ Rand($a$)
7  $\mathbf{dist}[t - ones] \leftarrow v$
8  $pos \leftarrow pos - (\mathbf{dist}[t - ones] + 1)$
9  $ones \leftarrow ones - 1$
10 return RandAndUpd($pos, ones, \mathbf{dist}$)
```

The CompleteRandomQuotient procedure handles the situation represented in Figure 3.1b. It computes the length of the new run of `0`s as a random integer picked from the interval between the already read value $v = q \cdot d$ and the remaining positions, which are computed as $a = pos - v - (ones - 1)$. In this way the computation takes into account also the still to be placed `1` bits of the CW output in order to prevent an `LengthOutOfBounds` failure,

Algorithm 3.1.5: CompleteRemainderPrefix($pos, ones, \mathbf{dist}, \mathtt{s}, c, q$)

```
Input: pos: remaining available positions,
       ones: ones to place,
       dist: vector containing intra-ones distances,
       s: encoded bitstring of length l,
       c: cursor (number of bits read from s)
       q: unary quotient value.
Data: b: remaining readable bits in s,
      r: value of the remainder,
      t_h: truncated binary representation offset,
      d: density of zeros in s,
      u: bits to encode d,
      rem: table of possible remainders to choose from,
      t: weight of the constant weight output vector,
      f: read fragment.
1  $b \leftarrow l - c$
2  if $b$ then
3      $\mathtt{f} \leftarrow$ BitstreamRead($\mathtt{s}, b$)
4      $r \leftarrow$ BinToInt($\mathtt{f}$)
5      $t_h \leftarrow 2^u - d$
6      $j \leftarrow 0$
7      for $i \leftarrow 0$ to $d - 1$ do
8          if ($i < t_h$ and $\lfloor r = \frac{i}{2^{u-1-b}} \rfloor$) or
9          ($i \geq t_h$ and $\lfloor r = \frac{i}{2^{u-b}} \rfloor$) then
10             $\mathbf{rem}[j] \leftarrow i$
11             $j \leftarrow j + 1$
12     if $j = 0$ then
13         return OutBoundFailure
14     $\mathbf{dist}[t - ones] \leftarrow \mathbf{rem}[$Rand($j$)$] + q \cdot d$
15 else
       // $b = 0$
16     $\mathbf{dist}[t - ones] \leftarrow$ Rand($d$) $+ q \cdot d$
17 $pos \leftarrow pos - (\mathbf{dist}[t - ones] + 1)$
18 $ones \leftarrow ones - 1$
19 return RandAndUpd($pos, ones, \mathbf{dist}$)
```

which can happen if the last read bits of $\mathtt{s}$ already generated an integer value v greater than a. Then it decrements the value of *pos* and *one* and calls the RndAndUpd function which will proceed to randomly set the leftover ones and generated the constant weight output vector.

The last procedures, CompleteRemainderPrefix and CompleteRemainderLastBit handle the situations represented in Figures 3.1c and 3.1d. Given the implementation of Algorithm 3.1.2, CompleteRemainderLastBit function is called when the check on having at least u bits available to be read fails. For this reason, the called function starts by computing the actually remaining readable bits b (line 1). If any, it reads them and computes a table **rem** which contains the integers associated with the truncated binary representations starting with the given b bits (lines 2-11).

Algorithm 3.1.6: COMPLETEREMAINDERLASTBIT($pos, ones, \mathbf{dist}, q, r$)

```
Input: $pos$: remaining available positions,
       $ones$: ones to place,
       $\mathbf{dist}$: vector containing intra-ones distances,
       $q$: decompressed quotient,
       $\mathbf{g}$: read truncated binary representation of the remainder.
Data: $d$: density of zeros in $\mathbf{s}$,
      $u$: bits to encode $d$,
      $t$: weight of the constant weight output vector.
1 $r \leftarrow 2 \cdot$ BINTOINT($\mathbf{g}$) + RAND(2) $- (2^u - d)$
2 $\mathbf{dist}[t - ones] \leftarrow r + q \cdot d$
3 $pos \leftarrow pos - (\mathbf{dist}[t - ones] + 1)$
4 $ones \leftarrow ones - 1$
5 return RANDANDUPD($pos, ones, \mathbf{dist}$)
```

Then, at line 14, the remainder value is randomly extracted from the values of **rem**. Note that if the b read bits cannot be extended into a truncated binary representation corresponding to an admissible value, the algorithm returns a decoding failure (lines 12-13). Lines 15 and 16 address the eventuality of the last bits of $\mathbf{s}$ being just enough to extract the whole quotient part plus the additional `0` bit separator, i.e. $b = 0$, in this case the remainder value can be simply set to any integer smaller than d. The last lines, 17-19, compute the new distance from the last positioned `1` bit, decrement the remaining number of ones to place and the remaining available positions and call RNDANDUPD.

The last padding function, COMPLETEREMAINDERLASTBIT, has to randomly generate only the last bit, thus the procedure simply computes the integer value of r and add zero or one to it. Then, as in the previous functions, the parameters are updated and the RNDANDUPD procedure called.

Note that only COMPLETERANDOMQUOTIENT function picks the new position in order to guarantee a successful encoding, COMPLETEREMAINDERPREFIX in fact selects the new asserted bit position only based on the last read bits while COMPLETEREMAINDERLASTBIT is essentially a coin toss over the last bit value. This choice is due to the fact that "forcing" the encoding in every situation leads to an accumulation of asserted bits in the last bits of the output string thus worsening the distribution.

Chapter 4

Efficient Implementation of the LEDAcrypt Cryptosystem

This chapter focuses on LEDAcrypt performance improvements, which leverage both sub-quadratic multiple precision multiplication approaches and code vectorization specifically optimized for the target architecture, which is ARMv8a in this case.

4.1 Code Optimization

By Amdahls Law, the speed up given from the improvement of a part of an algorithm is limited to the fraction of time in which that part takes place. In other words, the idea behind the optimization process is to first make the common case fast. Thus, being the functions that are called the most through the whole codebase, the first thing to optimize by means of vector instructions are of course the addition and shifting operations. In fact, these improved functions will be then leveraged by any other more complex function.

A common optimization technique is *loop unrolling*: loops can be unrolled by a factor f, meaning that the same instruction is performed over f data items at each iteration, reducing the number of iterations by $\frac{1}{f}$.

As explained in 1.5, the most expensive instruction are loads and stores which require an exchange of data with the memory and for this reason they need more time to fetch data from it, than operations working over

Algorithm 4.1.1: Right to left shift and add multiplication in $\mathbb{F}_{2^m}$

```
Input: $\mathbf{a}, \mathbf{b}$ binary polynomials of degree at most $m-1$
Output: $\mathbf{c} = \mathbf{ab}$ binary polynomial of degree at most $2m-2$
Data:
1 if $a_0 = 1$ then
2     $\mathbf{c} \leftarrow \mathbf{b}$
3 else
4     $\mathbf{c} \leftarrow 0$
5 for $i \leftarrow 0$ to $m-1$ do
6     $\mathbf{b} \leftarrow \mathbf{b} \ll 1$
7     if $a_i = 1$ then
8         $\mathbf{c} \leftarrow \mathbf{c} \oplus \mathbf{b}$
9 return $\mathbf{c}$
```

data which is already stored in registers. For this reason it is fundamental to reduce as much as possible this kind of instructions by loading the data when needed and storing it only at the end of the computation, which is exactly what is pursued in the shown algorithms.

Dealing with performances, another thing to consider while working with arbitrary length numbers is that some arithmetic operations, especially multiplication, can require a significant amount of time. When thinking of a multiplication, the common algorithm is the schoolbook multiplication, represented in Algorithm 4.1.1, which can be though as a shift-and-add operation when working over binary fields. Of course exist better alternatives which are based on the "divide et impera" principle, i.e. the factors are decomposed into smaller parts which are then multiplied and added, achieving in this way a significant speed up. In particular, the schoolbook multiplication has quadratic complexity over the number of bits of the input, while the other approaches complexity is smaller than $O(n^2)$.

The algorithm proposed in [16] by Karatsuba exploits this approach: given two 2^l-bits integers $x = x_1 2^l + x_0$ and $y = y_1 2^l + y_0$, xy can be computed by performing three multiplications of l-bit integers instead of one multiplication with $2l$-bit integers

$$x \cdot y = (x_1 2^l + x_0)(y_1 2^l + y_0) = x_1 y_1 2^{2l} + ((x_0 + x_1)(y_0 + y_1) - x_1 y_1 - x_0 y_0)\, 2^l + x_0 y_0.$$

Example 3. *For example, the multiplication between* $1234 = 12 \cdot 10^2 + 34$ *and* $5678 = 56 \cdot 10^2 + 78$ *can be computed as:*

$$x = 12 \cdot 56 = 672 \tag{4.1}$$

Algorithm 4.1.2: Karatsuba($\mathbf{a}, \mathbf{b}, n$)

```
Input: $\mathbf{a}, \mathbf{b}$ binary polynomials of degree at most $n-1$
Output: Binary polynomial of degree at most $2n-2$
Data: $t_h$: threshold under which is more convenient a schoolbook multiplication
1  if $n \leq t_h$ then
2      SchoolbookMul($\mathbf{a}, \mathbf{b}$)
3      return
4  if $n \bmod 2 = 0$ then
5      $\mathbf{p} \leftarrow$ Karatsuba($\mathbf{a}[n-1:\frac{n}{2}], \mathbf{b}[n-1:\frac{n}{2}], \frac{n}{2}$)
6      $\mathbf{s} \leftarrow$ Karatsuba($\mathbf{a}[n-1:\frac{n}{2}] \oplus \mathbf{a}[\frac{n}{2}-1:0], \mathbf{b}[n-1:\frac{n}{2}] \oplus \mathbf{b}[\frac{n}{2}-1:0], \frac{n}{2}$)
7      $\mathbf{t} \leftarrow$ Karatsuba($\mathbf{a}[\frac{n}{2}-1:0], \mathbf{b}[\frac{n}{2}-1:0], \frac{n}{2}$)
8      return $\mathbf{p} \ll \mathbf{n} \oplus (\mathbf{s} \oplus \mathbf{p} \oplus \mathbf{t}) \ll \frac{n}{2} \oplus \mathbf{t}$
9  else
10     $\mathbf{p} \leftarrow$ Karatsuba($\mathbf{a}[n-1:\lceil\frac{n}{2}\rceil], \mathbf{b}[n-1:\lceil\frac{n}{2}\rceil], \lfloor\frac{n}{2}\rfloor - 1$)
11     $\mathbf{s} \leftarrow$ Karatsuba($\mathbf{a}[n-1:\lceil\frac{n}{2}\rceil] \oplus \mathbf{a}[\lfloor\frac{n}{2}\rfloor:0], \mathbf{b}[n-1:\lceil\frac{n}{2}\rceil] \oplus \mathbf{b}[\lfloor\frac{n}{2}\rfloor:0], \lceil\frac{n}{2}\rceil$)
12     $\mathbf{t} \leftarrow$ Karatsuba($\mathbf{a}[\lfloor\frac{n}{2}\rfloor:0], \mathbf{b}[\lfloor\frac{n}{2}\rfloor:0], \lceil\frac{n}{2}\rceil$)
13     return $\mathbf{p} \ll (2\lceil\frac{n}{2}\rceil) \oplus (\mathbf{s} \oplus \mathbf{p} \oplus \mathbf{t}) \ll \lceil\frac{n}{2}\rceil \oplus \mathbf{t}$
```

$$y = 34 \cdot 78 = 2652 \tag{4.2}$$

$$z = (12+34)(56+78) - x - y = 2840 \tag{4.3}$$

$$x \cdot 10^{2 \cdot 2} + y + z \cdot 10^2 = 7006652 = 1234 \cdot 5678. \tag{4.4}$$

This approach can indeed be easily applied also to polynomials, as shown in Algorithm 4.1.2, if we consider each bit as a polynomial coefficient. In particular, a the byte `00001111` = `0x0F` represents the polynomial $\mathbf{a} = x^3 + x^2 + x + 1$ written in Big Endian format, i.e. the leftmost bit $\mathbf{a}[3]$ is the most significant one while $\mathbf{a}[0]$ is the least significant one. With this representation of binary polynomials, every add operation become a a Boolean eXclusive-OR (xor) one while multiplication and division by x become shift operations.

An additional speed up can be achieved by leveraging Toom-k algorithms, first proposed in [19], where k is the number of parts in which each source operand is divided. Karatsuba multiplication can be seen as an implementation of a Toom-2 algorithm. Toom-k complexity is $O(c(k)n^{log(2k-1)/log(k)})$, where $c(k)$ depends on the number and cost of simpler operations such as addition and shifting. Each Toom-k algorithm requires the polynomial evaluation of the two operands and a polynomial interpolation problem, with base points not specified a priori, giving rise to many possible Toom-k algorithms, even for a fixed size of the operands. Thus the exact sequence of smaller multiplication, addition and shifting operation determines the real efficiency of the algorithm. Algorithm 4.1.3 outlines a highly efficient Toom-3

Algorithm 4.1.3: TOOM3($\mathbf{a}, \mathbf{b}$, n)

```
**Input**: $\mathbf{u}, \mathbf{v}$ binary polynomials of degree at most $n-1$
**Output**: Binary polynomial of degree at most $2n-2$
**Data**: $th$: threshold under which is more convenient a schoolbook multiplication
if $n \leq th$ then
    SCHOOLBOOKMUL($\mathbf{a}, \mathbf{b}$)
    return
$l = \lceil \frac{n}{3} \rceil$
$u_2 \leftarrow \mathbf{u}[n-1:2l], u_1 \leftarrow \mathbf{u}[2l-1:l], u_0 \leftarrow \mathbf{u}[l-1:0]$
$v_2 \leftarrow \mathbf{v}[n-1:2l], v_1 \leftarrow \mathbf{v}[2l-1:l], v_0 \leftarrow \mathbf{v}[l-1:0]$
$w_3 \leftarrow u_2 \oplus u_1 \oplus u_0$
$w_2 \leftarrow v_2 \oplus v_1 \oplus v_0$
$w_1 \leftarrow$ TOOM3($w_3, w_2, l$)
$w_0 \leftarrow u_2 \cdot x^2 \oplus u_1 \cdot x$
$w_4 \leftarrow v_2 \cdot x^2 \oplus v_1 \cdot x$
$w_3 \leftarrow w_3 \oplus w_0$
$w_2 \leftarrow w_2 \oplus w_4$
$w_0 \leftarrow w_0 \oplus u_0$
$w_4 \leftarrow w_4 \oplus v_0$
$w_3 \leftarrow$ TOOM3($w_3, w_2, l$)
$w_2 \leftarrow$ TOOM3($w_0, w_4, l$)
$w_4 \leftarrow$ TOOM3($u_2, v_2, l$)
$w_0 \leftarrow$ TOOM3($u_0, v_0, l$)
$w_3 \leftarrow w_3 \oplus w_2$
$w_2 \leftarrow \frac{\frac{w_2 \oplus w_0}{x} \oplus w_3 \oplus w_4 \cdot (x^3+1)}{x+1}$
$w_1 \leftarrow w_1 \oplus w_0$
$w_3 \leftarrow \frac{(w_3 \oplus w_1)}{x \cdot (x+1)}$
$w_1 \leftarrow w_1 \oplus w_4 \oplus w_2$
$w_2 \leftarrow w_2 \oplus w_3$
$\mathbf{w}[5l-1:4l] \leftarrow w_4, \mathbf{w}[4l-1:3l] \leftarrow w_3$
$\mathbf{w}[3l-1:2l] \leftarrow w_2, \mathbf{w}[2l-1:l] \leftarrow w_1$
$\mathbf{w}[l-1:0] \leftarrow w_0$
return $\mathbf{w}$
```

Algorithm 4.1.4: SUM($\mathbf{a}, \mathbf{b}$, n)

```
**Input**: $\mathbf{a}, \mathbf{b}$: binary polynomials of degree at most $n$
**Output**: $\mathbf{c}$: binary polynomial of degree $n$
**Data**: $l_{word}$: machine word bit size
for $i \leftarrow 0$ to $n - l_{word}$ by $l_{word}$ do
    $\mathbf{c}[l_{word} + i : i] \leftarrow \mathbf{a}[l_{word} + i : i] \oplus \mathbf{b}[l_{word} + i : i]$
$r \leftarrow n \bmod l_{word}$
if $r \neq 0$ then
    $\mathbf{c}[n : n-r] \leftarrow \mathbf{a}[n : n-r] \oplus \mathbf{b}[n : n-r]$
return $\mathbf{c}$
```

implementation proposed by Bodrato in [6].

Each single operation within these algorithms can be vectorized as shown in Algorithm 4.1.4 which outlined a vectorized sum. The KARATSUBA and TOOM-3 recursion limits, i.e. the minimum number of required architectural words, are chosen empirically to minimize the multiplications running times, specifically for each possible size of the operands.

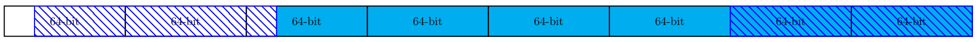


Figure 4.1: Syndrome extension to facilitate 128-bit vector extraction. The original input $\mathbf{s}_{priv}$, represented in cyan, is copied in $\mathbf{s}_{curr}$ extended with the blue lined 128-bit piece.

As a final optimization, we explicitly employed the optimal Karatsuba expansions of all the multiplications with a number of input limbs between 1 and 9 reported in Su and Fan work on the optimal Karatsuba expansions for binary polynomial multiplication [29].

4.2 Decoder Optimization through SIMD instructions

The decoder, which extracts the errors from a compressed arbitrary bit-string, is a critical part of code-based cryptosystems. Here are outlined the original sequential algorithm and the improved vectorized one.

It's easy to see from Algorithm 4.2.1 that there is plenty of room for vectorization. In sequence, it is possible to improve the first part, where the unsatisfied parity checks are computed (lines 3-10), the correlation computation (lines 11-23) and also the bitflips on the syndrome (lines 24-31). For ease of reading, the three optimizations have been split into different algorithms.

The NEON intrinsics instructions, which are capable of working on 128-bit at the time, have been used to parallelize said parts. Notice that in order to avoid interrupting the execution with wrap around issues, it is convenient to prepend to the leftmost bits the rightmost ones, as shown in Figure 4.1, since the chosen code is cyclic: the position of the bits on the syndrome is computed modulo $n_0 p$ and thus the positions exceeding the left bound of syndrome correspond to the rightmost ones.

Regarding the unsatisfied parity checks computation, it can be chosen to iterate either through the $n_0 p$ bits of the syndrome or through the d_v and m indexes of the $\mathbf{H}$ and $\mathbf{Q}$ matrices. The chosen strategy, illustrated by Algorithm 4.2.2 which vectorize lines 5-10 of Algorithm 4.2.1, is the first one, since this will produce adjacent memory reads and writes, leveraging the locality principle. On the contrary, the latter approach, will produce scattered

Algorithm 4.2.1: QDecoder

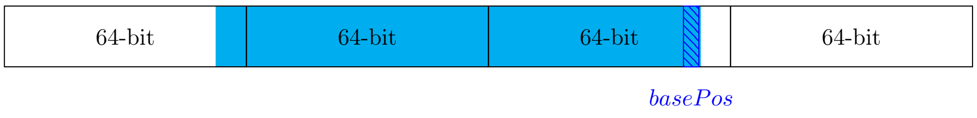
```
Input: out: $n_0$ binary polynomials of errors,
       $\mathbf{H}_{tr}$: transpose parity-check matrix, represented as $n_0 \times d_v$ integer matrix
       containing the positions in $\{0, 1, ...p-1\}$ of the set coefficients in the $n_0$ blocks of
       $\mathbf{H} = [\mathbf{H}_0^T | \mathbf{H}_1^T | \cdots | \mathbf{H}_{n_0-1}^T]$,
       $\mathbf{Q}_{tr}$: private matrix, represented as an $n_0 \times m$, $m = \sum_{i=0}^{n_0-1} m_i$ integer matrix
       containing the positions in $\{0, \dots, n_0 p - 1\}$ of the asserted coefficients in $\mathbf{Q}^T$ rows,
       $\mathbf{s}_{priv}$: already computed syndrome
Output: $out$: $n_0$ binary polynomials of errors
Data: $it_{max}$: maximum number of iterations,
      $\mathbf{W}_{bw}$: $n_0 \times n_0$ matrix representing the block weights of the Q matrix,
      $\mathbf{bit}_Q$, $\mathbf{blk}_Q$: vectors of length $m$ $num$: number of 64-bit word containing $\mathbf{s}_{priv}$
1  $iteration \leftarrow 0$
2  repeat
3      $\mathbf{s}_{curr} \leftarrow \mathbf{s}_{priv}$
4      upc $\leftarrow 0$
5      for $i \leftarrow 0$ to $n_0 - 1$ do
6          for $valueIdx \leftarrow 0$ to $p - 1$ do
7              for $h \leftarrow 0$ to $d_v - 1$ do
8                  $tmp \leftarrow \mathbf{H}_{tr}[i, h] + valueIdx)$ mod $p$
9                  if gf2xGetCoeff($\mathbf{s}_{curr}, tmp$) then
10                     $\mathbf{upc}[i \cdot p + valueIdx] \leftarrow \mathbf{upc}[i \cdot p + valueIdx] + 1$
11     for $i \leftarrow 0$ to $n_0 - 1$ do
12         for $j \leftarrow 0$ to $p - 1$ do
13             $idx_Q \leftarrow 0$, $end_Q \leftarrow 0$, $corr \leftarrow 0$
14             for $blockIdx \leftarrow 0$ to $n_0 - 1$ do
15                 $end_Q \leftarrow end_Q + \mathbf{W}_{bw}[blockIdx][i]$
16                 while $idx_Q \leq end_Q$ do
17                     $tmp \leftarrow \mathbf{Q}_{tr}[i][idx_Q] + j$
18                     if $tmp \geq p$ then
19                         $tmp \leftarrow tmp - p$
20                     $\mathbf{bit}_Q[idx_Q] \leftarrow tmp$
21                     $\mathbf{blk}_Q[idx_Q] \leftarrow blockIdx$
22                     $corr \leftarrow corr + \mathbf{upc}[tmp + blockIdx \cdot p]$
23                     $idx_Q \leftarrow idx_Q + 1$
24             if $corr \geq thresholds[iteration]$ then
25                 gf2xToggleCoeff($\mathbf{out}[i], j$)
26                 for $v \leftarrow 0$ to $m - 1$ do
27                     for $h \leftarrow 0$ to $dv - 1$ do
28                         $posToFlips_s = \mathbf{H}_{tr}[\mathbf{blk}_Q[v]][h] + \mathbf{bit}_Q[v])$
29                         if $posToFlips_s \geq p$ then
30                             $posToFlips_s \leftarrow posToFlips_s - p$
31                         gf2xToggleCoeff($\mathbf{s}_{priv}, posToFlips_s$)
32     $iteration \leftarrow iteration + 1$
33     $check \leftarrow 0$
34 until $iteration \geq it_{max}$ and $\mathbf{s}_{priv} \neq 0$
35 if $\mathbf{s}_{priv} = 0$ then
36     return True, out
37 else
38     return False, out
```

memory requests, increasing the latency of the algorithm. Differently from the approach of Drucker and Gueron in [9], this one adopts an "horizontal summation" method (see Figure 2.3 over 128 *upc*s at once: i.e. the value of several *upc*s is kept into registers until their computation is terminated.

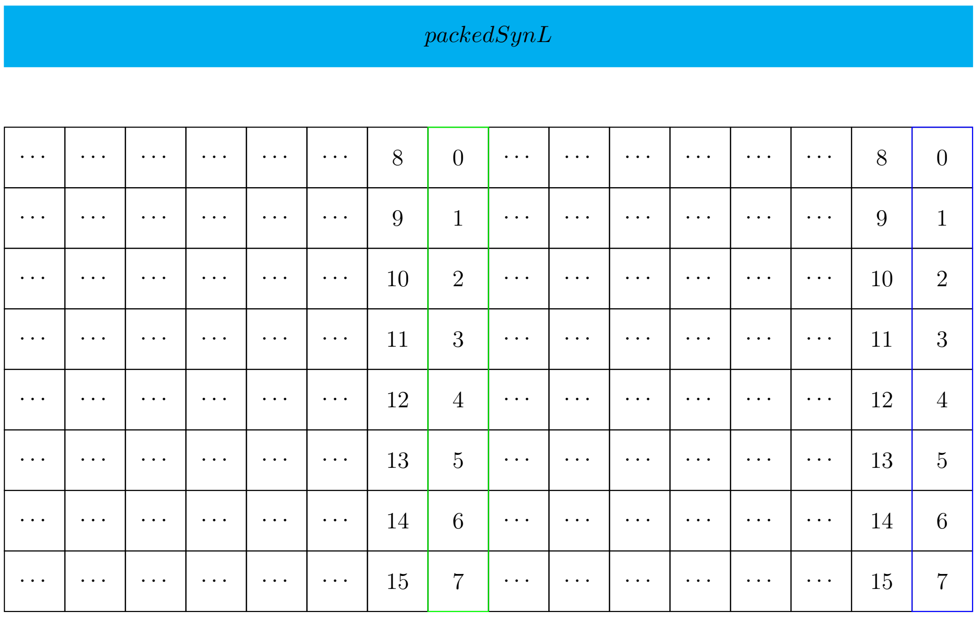
Algorithm 4.2.2: UPCVECT()

```
Data: $l_{word}$: bits in chosen machine word,
      $\mathbf{H}_{tr}$: transpose parity-check matrix, represented as $n_0 \times d_v$ integer matrix
      containing the positions in $\{0, 1, ...p-1\}$ of the set coefficients in the $n_0$ blocks of
      $\mathbf{H} = [\mathbf{H}_0^T|\mathbf{H}_1^T| \cdots |\mathbf{H}_{n_0-1}^T]$,
      $\mathbf{upc}_M$: vector of length $8l_{word}$ representing a $16 \times 8$ matrix of unsatisfied parity
      checks,
      GETVECTORBOUNDLESS($\mathbf{s}, i$): extracts exactly $l_{word}$ bits from $\mathbf{s}$ starting from $i$-th
      bit position.
mask = $0x01 \cdots 010101$
for $i \leftarrow 0$ to $n_0 - 1$ do
    $valueIdx \leftarrow 0$
    while $valueIdx < p$ do
        for $x \leftarrow 0$ to $7l_{word}$ by $l_{word}$ do
            $\mathbf{upc}_M[x + l_{word} - 1 : x] \leftarrow 0$
        for $h \leftarrow 0$ to $d_v - 1$ do
            $basePos \leftarrow \mathbf{H}_{tr}[i, h] + valueIdx$
            $packedSynBits \leftarrow$ GETVECTORBOUNDLESS($\mathbf{s}_{curr}, basePos$)
            for $x \leftarrow 0$ to $7l_{word}$ by $l_{word}$ do
                $\mathbf{upc}_M[x + l_{word} - 1 : x] \leftarrow$
                $\mathbf{upc}_M[x + l_{word} - 1 : x] \oplus (packedSynBits\&$mask$)$
                $packedSynBits = packedSynBits \gg 1$
        $\mathbf{upc}_M \leftarrow$ TRANSPOSE8 $\times$ LWORD($\mathbf{upc}_M$)
        for $x \leftarrow 0$ to $7l_{word}$ by $l_{word}$ do
            $\mathbf{upc}[valueIdx + x + l_{word} - 1 : valueIdx + x] \leftarrow \mathbf{upc}_M[x + l_{word} - 1 : x]$
        $valueIdx \leftarrow valueIdx + l_{word}$
```

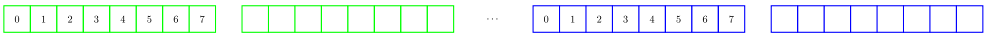
The vectorize algorithm fetches $l_{word} = 128$ packed bits from each position indicated in $\mathbf{H}_{tr}$, which contains the asserted bit position of $\mathbf{H}^T$, and adds them to the 128 *upc* counters in the 16×8 matrix $\mathbf{upc}_M$ (lines 2-12). For every asserted bit of the parity check matrix $\mathbf{H}$, the correspondent bit of the syndrome is identified by the variable *basePos* and leverage by the GETBITVECTORBOUNDLESS function to extract a 128-bit vector *packedSynBits* from the syndrome, with *basePos* bit in the least significant position (Figure 4.2a). Note that process exploits the sparsity and quasi-cyclic properties of $\mathbf{H}$ by iterating only over the positions specified in $\mathbf{H}_{tr}$. Then, in lines 10-12 is performed the computation of the *upc*s by means of the bitstring mask `mask` which is applied to the *packedSynBits* bits: the rightmost bit of each byte of *packedSynBits*, which is shifted at each iteration, is extracted and added to the associated *upc*, i.e. $upc_0, upc_8, upc_{16} \cdots upc_{56}$ at the beginning and then $upc_{0+x}, upc_{8+x}, upc_{16+x} \cdots upc_{56+x}$ after x shift (see Figure 4.2b, thus incrementing 16 *upc* at once. In this way is possible to linked the asserted bits *packedSynBits* with $\mathbf{H}$ coefficients and thus to compute the unsatisfied parity checks. After the computation, the 128 *upc* need to be stored in proper order in the *upc* vector: it is thus necessary



(a) GETBITVECTORBOUNDLESS



(b) $\mathbf{upc}_M$



(c) Trasposition and linearization

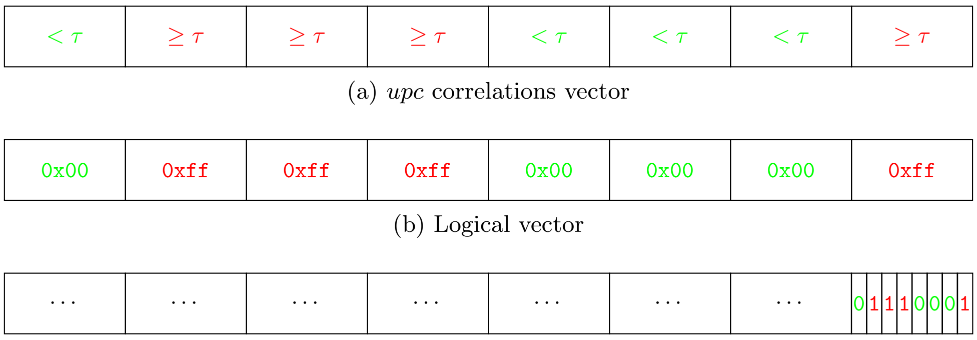
to transpose the $\mathbf{upc}_M$ and linearise it by rows, as illustrated in 4.2c (lines 26-34).Obviously, a specific intrinsic function which performs the $8 \times l_{word}/8$ transposition does not exist, thus this specific operation is achieved leveraging `zip`, `unzip` and `transpose` intrinsics as shown in the Appendix A.

The second optimization concerns the computation of the correlation values (lines 11-23 of Algorithm 4.2.1). The best speed-up, leveraging NEON intrinsics, can be achieved through Algorithm 4.2.3 within $l_{word} = 128$ and $vecBlocks = 4$. Since a single correlation value is the sum of m "block circulant positioned" values of *upc*s, in line 5 the proper position *tmp* is computed and, in lines 6-8, for each *tmp* value $vecBlocks \cdot \frac{l_{word}}{8} = 4 \cdot \frac{128}{8} = 64$ consecutive *upc* correlations are considered and stored in the correlations vector $\mathbf{c}$.

The last part of the original algorithm, which regards the commit on the

Algorithm 4.2.3: CorrelationVec

```
Data: $l_{word}$: bits in chosen machine word,
      $\mathbf{c}$: vector of computed correlations,
      $vecBlocks$: number of $l_{word}$ bits block within $\mathbf{c}$,
      $\mathbf{Q}_{tr}$: private matrix, represented as an $n_0 \times m$, $m = \sum_{i=0}^{n_0-1} m_i$ integer matrix
      containing the positions in $\{0, \dots, n_0 p - 1\}$ of the asserted coefficients in $\mathbf{Q}^T$ rows,
      $m$: permanent of each circulant matrix defined from the weights of each
      row/column of Q circulant blocks.
for $j \leftarrow 0$ to $p - 1$ by $l_{word}/8 \cdot vecBlocks$ do
    for $k \leftarrow l_{word}$ to $l_{word} \cdot vecBlocks$ by $l_{word}$ do
        $\mathbf{c}[k-1 : k - l_{word}] \leftarrow 0$
    for $idx_Q \leftarrow 0$ to $m - 1$ do
        $tmp \leftarrow \mathbf{Q}_{tr}[i][idx_Q] + j$
        $index \leftarrow p \cdot qBlockOffsets[idx_Q] + tmp$
        for $k \leftarrow 0$ to $l_{word}(vecBlocks - 1)$ by $l_{word}$ do
            $\mathbf{c}[k + l_{word} - 1 : k] \leftarrow$
            $\mathbf{upc}[index + k + l_{word} - 1 : index + k] \oplus \mathbf{c}[k + l_{word} - 1 : k]$
return c
```



(a) *upc* correlations vector

(b) Logical vector

(c) Commit vector

Figure 4.2: Vectorize commit of bitflips.

syndrome **s** and on the **out** bitstring, can be vectorized by extracting the last bit from each byte in the **c** into the designated position and flip 128-bit word at the time, as shown in Figure 4.2. Each byte of Figure 4.2a contains the correlation value of a *upc*, if this value is greater then τ the corresponding syndrome bit has to be flipped. The vector in Figure 4.2b is thus used as a selector of *upc* yielding bitflips. For each byte of the logical vector a bit is extracted and copied into the rightmost unset bit of the commit vector represented in Figure 4.2c. In this way, instead of flipping one bit at the time it is possible to performs multiple bitflips at once by adding the full commit vector to the syndrome one. This method has revealed itself to not be suitable in this specific case, in fact when working with 128-bit words this approach generates more accesses to memory than the actual bitflips to perform. This happens since the number of bitflips to perform is low and their position sparse. In particular, if on average there is less than one bit to be flipped per single architectural word

$$\frac{t}{\frac{n_0 p}{l_{word}}} = \frac{t \cdot l_{word}}{n_0 p} < 1, \tag{4.5}$$

it is more convenient to perform a load-flip-store instruction sequence for every single bitflip Using an Intel processor instead, with instruction over 256 or even 512 bits may allow to gain a speedup from the vectorization of this last part of the decoder.

Chapter 5

Experimental Results

In this Chapter are gathered the results regarding the ones distribution analysis and the cryptosystem vectorized implementation benchmarks. The relevant data are displayed by means of tables and graphs.

The CWE analysis develops within three parts: benchmark suite choice, analysis of the length of the dense bitstrings and analysis of the CWE outputs distributions associated with the amount of failures.

The cryptosystem optimization section compares the performances of the reference implementation with the ones achieved by the vectorized implementation for each LEDAcrypt version: KEM, KEM-LT and PKC.

5.1 Constant Weight Encoder

This section focuses on the evaluation of the goodness of the CWE proposed alternatives compared to the LEDApkc round 1 implementation. In particular, we observed the efficiency of the CWE over CW vectors with a variable number of asserted bits compared over the same set of input strings.

5.1.1 Benchmark Suite

The observations have been performed over smaller n and t parameters than the real ones of LEDAcrypt, since the adoption of the exact same values is computationally infeasible. The four pairs of parameters, which are outlined in Table 5.1, have been chosen in increasing length n and decreasing density $\frac{n}{t}$ of the CW vector, maintaining a CWE target of about 2^{30}

Figure 5.1: The first table outlines the parameters used during the CWE analysis with their densities and the associate dense string length able to successfully generated a vector of length n and weight t. The second one shows the parameters of the first round implementation of LEDApkc for different security x levels.

$\mathbf{n}$	$\mathbf{t}$	$\frac{\mathbf{n}}{\mathbf{t}}$ (bits)	$\mathbf{min_l}$ (bits)	$\mathbf{max_l}$ (bits)	$\lceil \mathbf{log_2}\binom{\mathbf{n}}{\mathbf{t}} \rceil$ (bits)
30	15	0.5	15	30	28
100	6	0.06	18	35	31
746	4	0.005	16	37	34
1000	3	0.003	15	30	28

(b) LEDApkc

SL	**n** (bits)	**t** (bits)	**n/t**
	30026	143	0.005
1	28929	90	0.003
	33868	72	0.002
	49070	208	0.004
3	53481	129	0.002
	58868	104	0.002
	75238	272	0.004
5	85431	172	0.002
	91412	135	0.001

CW vectors, i.e. $log_2\binom{n}{t} \approx 2^{30}$. This amount in fact allows a good trade off between the amount of computation time required and the number of considered vectors which is large enough to trust the derived statistics.

5.1.2 Acceptable Input Lengths for CWE

The first phase of the analysis focuses on the determination of the acceptable input length for every pair of n and t and for every approach for the d estimation.

When dealing with a `fixed_d` approach the minimum length of the input

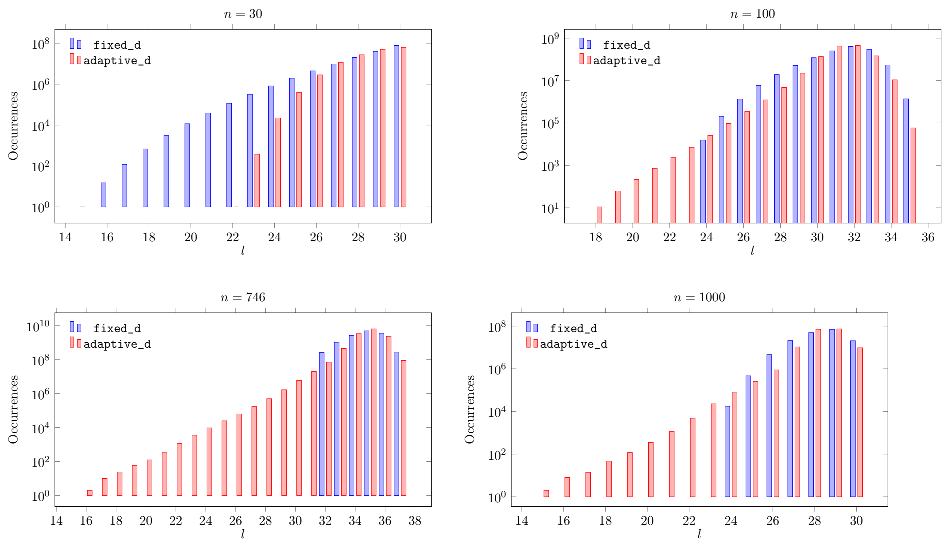


Figure 5.2: Number of CWD output dense bitstring of length l for every pair (n, t).

can be easily computed with the aforementioned formula:

$$l = (1 + \lfloor \log_2(d) \rfloor) \cdot t. \tag{5.1}$$

This formula cannot be used instead when leveraging the adaptive estimation of d. It thus necessary to generated all possible bitstrings with weight t and length n, feed them to the CWD function, i.e. the function which given a constant weight bitstring recovers the associate binary string and to record the length l of the returned binary strings. With this approach we enumerate any possible input CW string and thus we determine which lengths are accepted by the CWE, i.e. recognized as valid compressions. Figure 5.1 shows the number of successful decoding if the CW vectors into dense bitstrings for each recorded output length l. Note how in general an adaptive approach allow picking smaller input size compared to a fixed one. Table 5.1 reports the minimum and maximum bitstring lengths which can be successfully encoded for each pair (n, t): note that the maximum amount of padding bits needed is $max_l - min_l$, a longer input is not CW-decodable by construction.

5.1.3 CWE Efficiency

Knowing the boundaries of the encodable dense string, we generated the distributions of their produced CW vectors for each input length l by applying enumeratively the inverse method. In particular, for each pair (n, t), padding strategy and d estimation approach all possible binary strings of length l were generated and fed to the CWE. If the encoding process resulted in a success, i.e. a valid CW string, then the positions of the asserted bits of the output vector have been recorded. We computed the entropy of the random variable x modelling the value of the bits of the CW string, for each set of CWE parameters $(n, t, d,$ padding strategy$)$, adopting the `1` frequencies as probability values. Note that this is feasible since we explored the entire input domain.

The zero padding strategy and constrained random one output distributions have been reported for each of the three different computational choices for the d parameter; in particular

- fixed d results can be found in Figures 5.3, 5.4, 5.5,
- min-adaptive d results can be found in Figures 5.6, 5.7, 5.8,
- adaptive d results can be found in Figures 5.9, 5.10, 5.11.

For readability reasons the data is displayed by means of Cartesian plots instead of histograms and the frequencies of `1` positions when n equals 100, 746, 1000 are condensed in bins in order to maintain around 30 points in each plot. The graphs show for each triple $((n, t), d, padding)$ the asserted bits distribution, the value of the entropy and the amount of failures over different input lengths.

5.1.4 CWE Results Comments

When choosing the CWE approach, i.e. how to deal with the estimation of the d parameter and with the `LackOfEncodableInput` failures, it is important to consider the density of the CW string and the desired trade off of failure rates and uniformity of the output.

The simplest CWE implementation consists in the adoption of a zero padding method paired with the `fixed_d` computation. As illustrated in the plots (Figure 5.3), the distribution of the set bits in this case is highly skewed due to the accumulation of the null runlengths at the final positions of the output vector.

Generally, observing the obtained data it is possible to say that the adoption of an `adaptive_d` solution paired with a constrained random padding yields a more uniform distributions of the asserted bits over the CW vector. This is confirmed by the fact that this situation presents the higher entropy values (Figure 5.4) compared to the other ones (Figure 5.7 and 5.10). This combination in fact allows to decrease the minimum input length, which correspond to the minimum bits of information that have to be encoded, below the `fixed_d` limit while achieving a better uniformity over the set bits, due to its randomness. So, as a rule of thumb, choosing an `adaptive_d` computation plus a constrained random padding approach will reduce the amount of `LengthOutOfBounds` failures, while improving the skewness of the output distribution. However, when the CW vector is particularly dense ($n = 30, t = 15$) the graphs clearly show that a `fixed_d` approach yields substantially the same entropy and failures results as the adaptive ones. Moreover, the `min_adaptive_d` strategy, which take into consideration the input length of the string during the first estimation of said parameter and thus guarantees higher efficiency with respect to the fixed one, results in worse entropy value for any considered benchmark compared to the other approaches.

Notice that, in general, selecting a smaller value as minimum input length for the CWE will lead to a smaller amount of failures. In fact, if the constant weight vector is generated before reading the entire string the algorithm returns an `InsufficientInfoEncoded` failure, while if the input string terminates before finding all the t values then the input is padded and the computation continues. In this situation the choice over the padding strategy is fundamental to achieve a uniform output distribution. On the other hand, up to a certain point increasing l improves the distributions, then longer lengths lead to a larger amount of failures. For this reason, when the input margin is large the choice over the padding strategy doesn't really matter any more and the two solutions yield the same results. This can be also observed in the "first increasing then decreasing" trend of the entropy values within the increasing l bits.

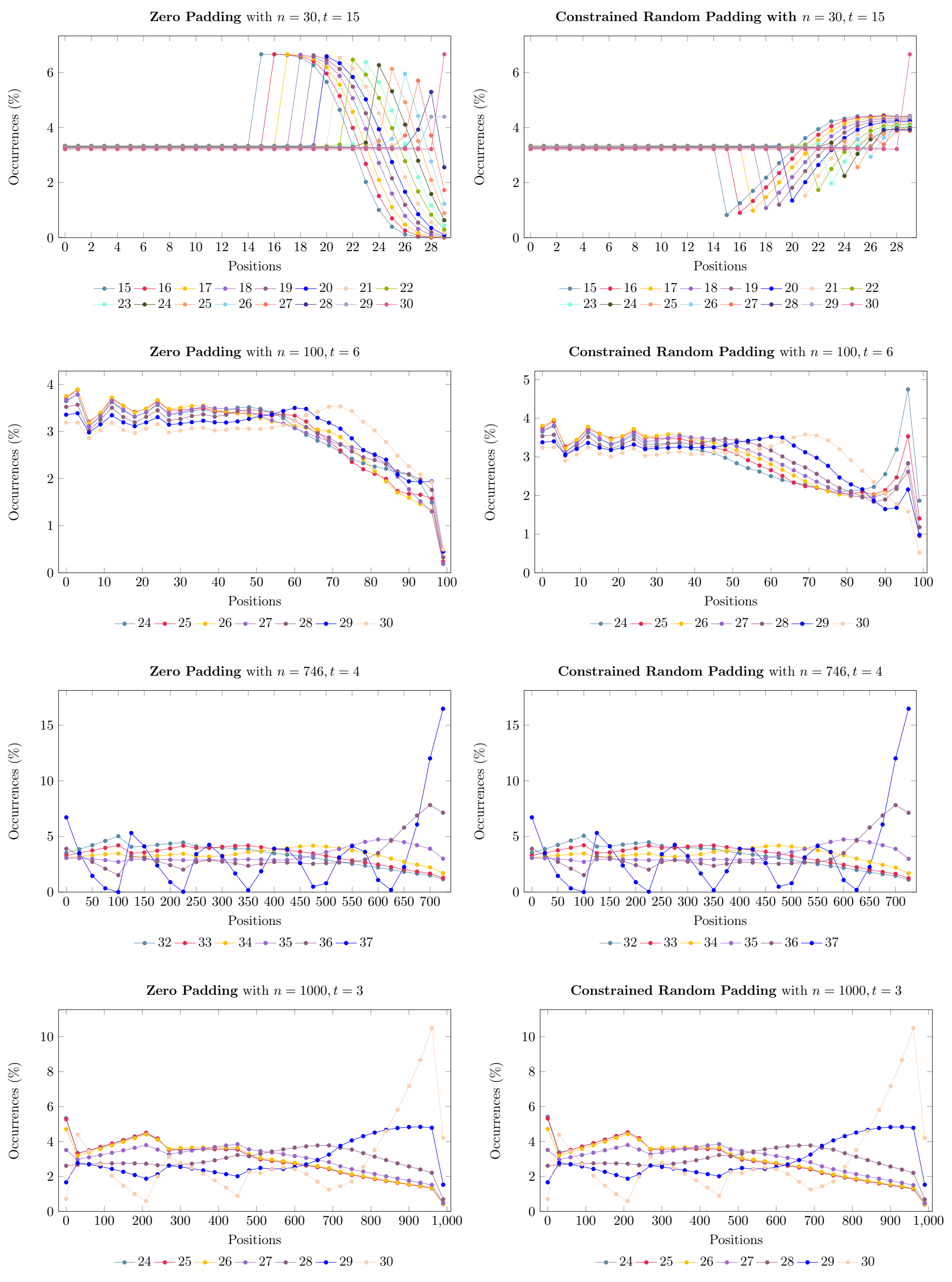


Figure 5.3: Asserted bit distributions of the CWE output, with `fixed_d` over alternatives padding strategies and (n, t) pairs.

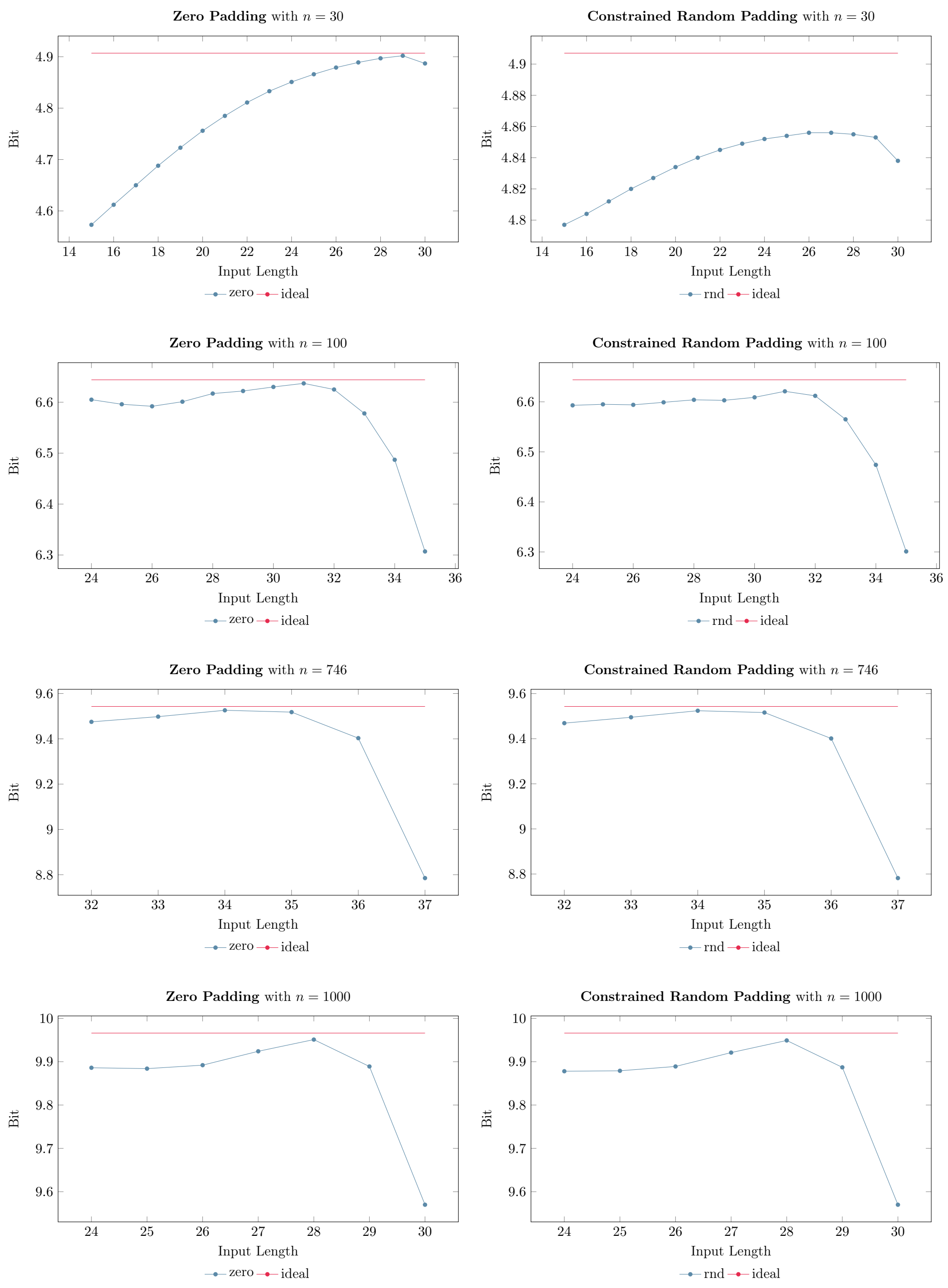


Figure 5.4: Entropy of the CWE output vector, for different input lengths computed with `fixed_d` approach.

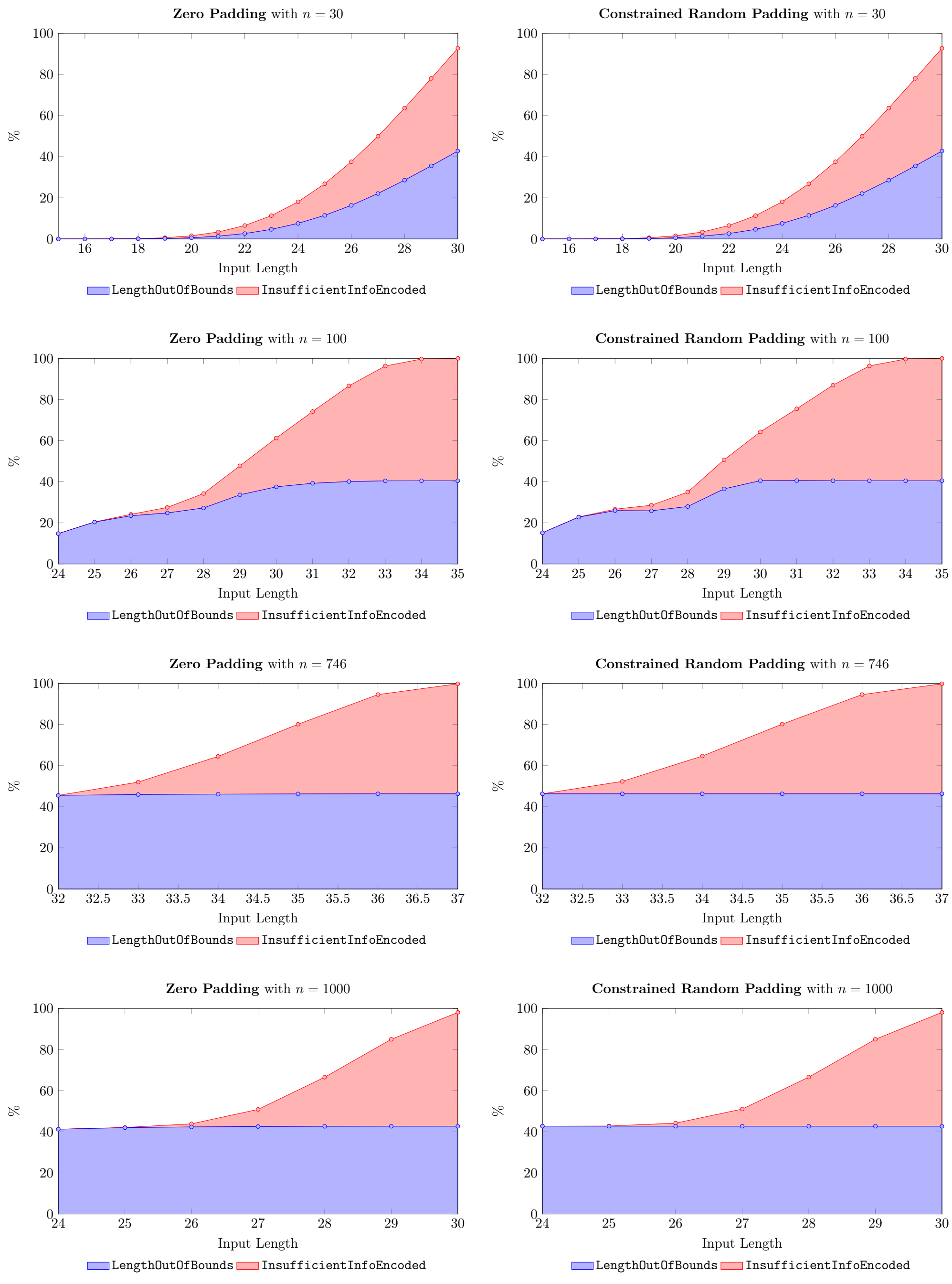


Figure 5.5: Total number of CWE failures for different input lengths computed with `fixed_d` approach.

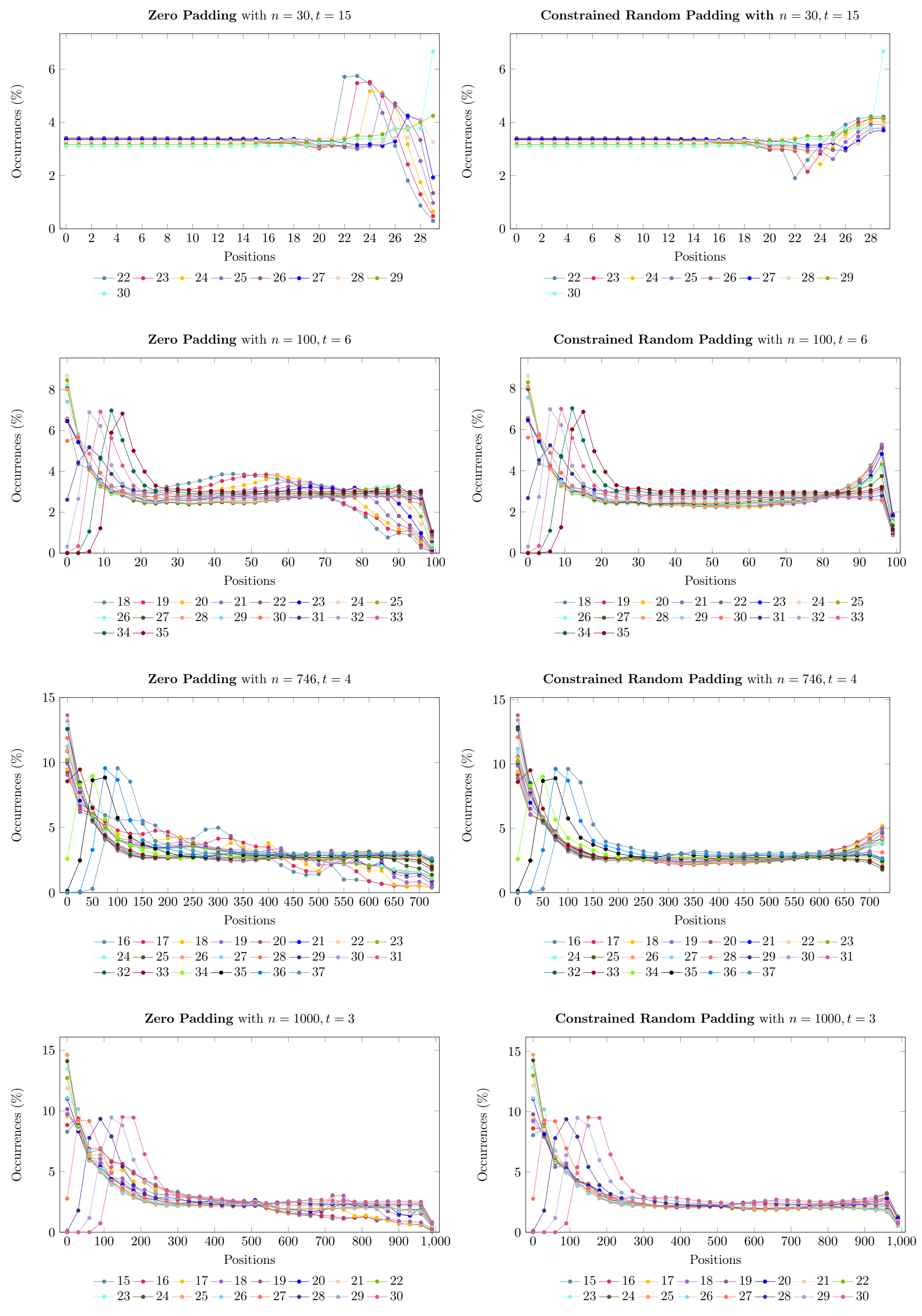


Figure 5.6: Asserted bit distributions of the CWE output, with `min_-adaptive_d` over alternatives padding strategies and (n, t) pairs.

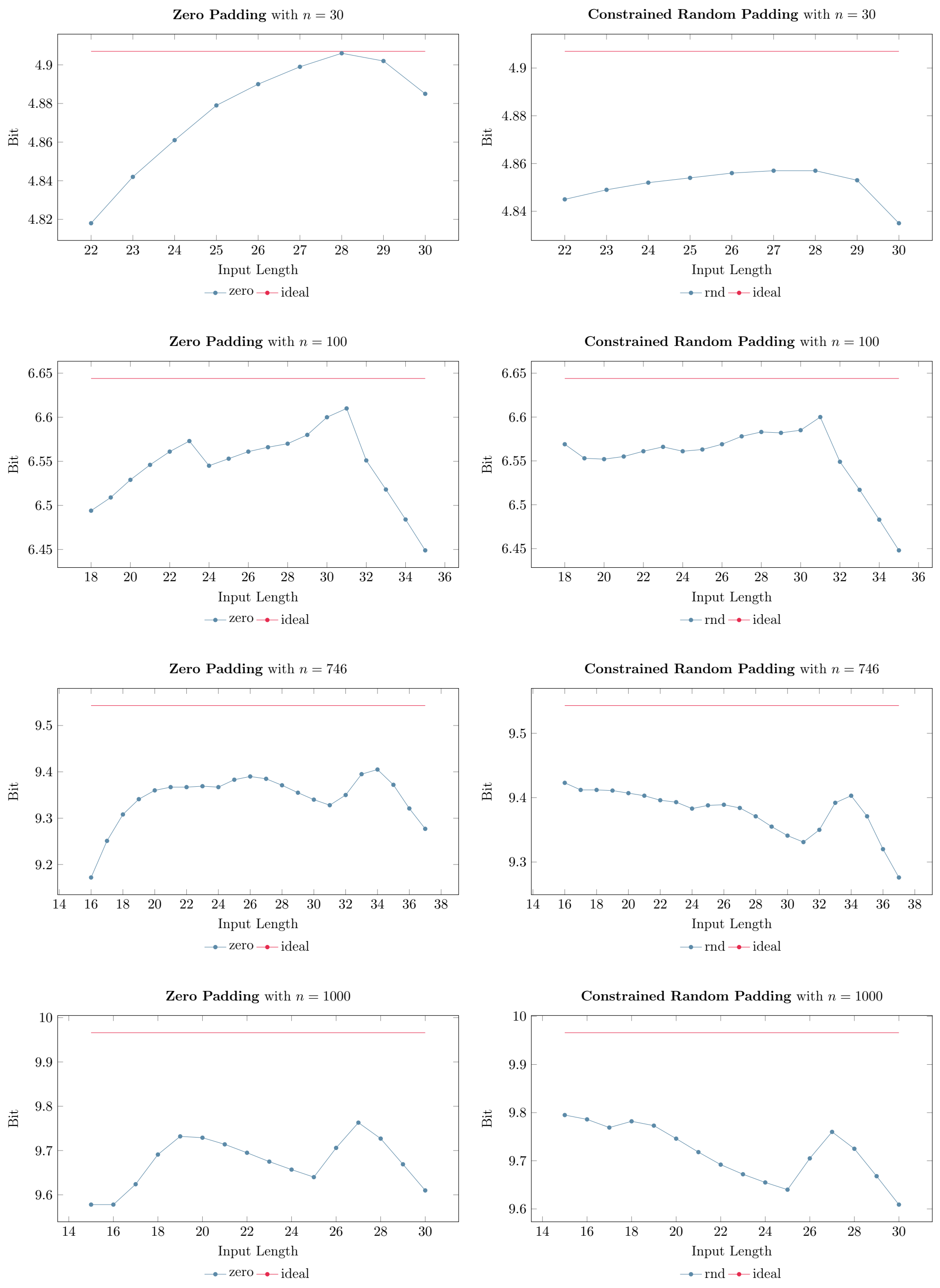


Figure 5.7: Entropy of the CWE output vector, for different input lengths computed with `min_adaptive_d` approach.

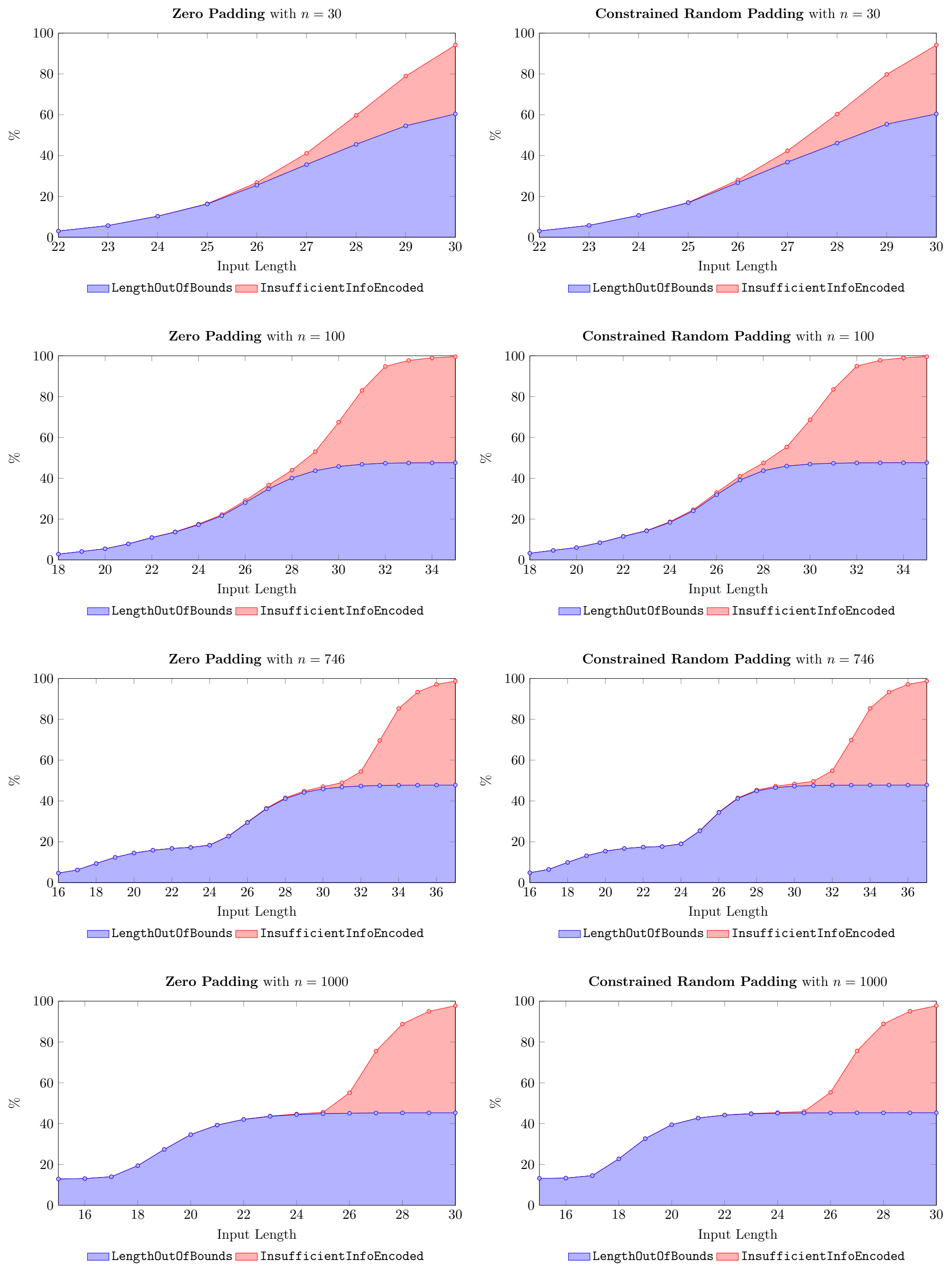


Figure 5.8: Total number of CWE failures for different input lengths computed with `min_adaptive_d` approach.

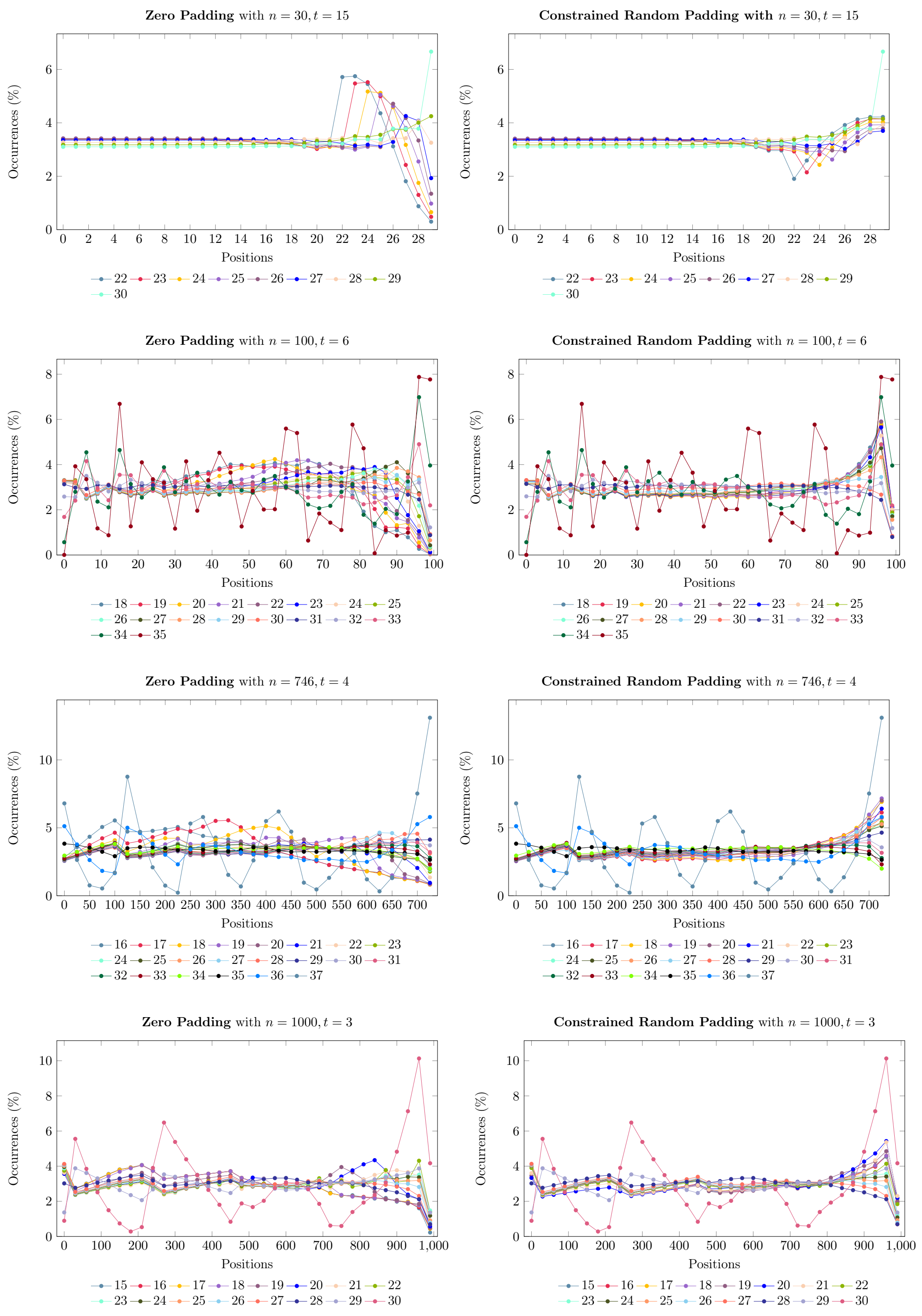


Figure 5.9: Asserted bit distributions of the CWE output, with `adaptive_d` over alternatives padding strategies and (n, t) pairs.

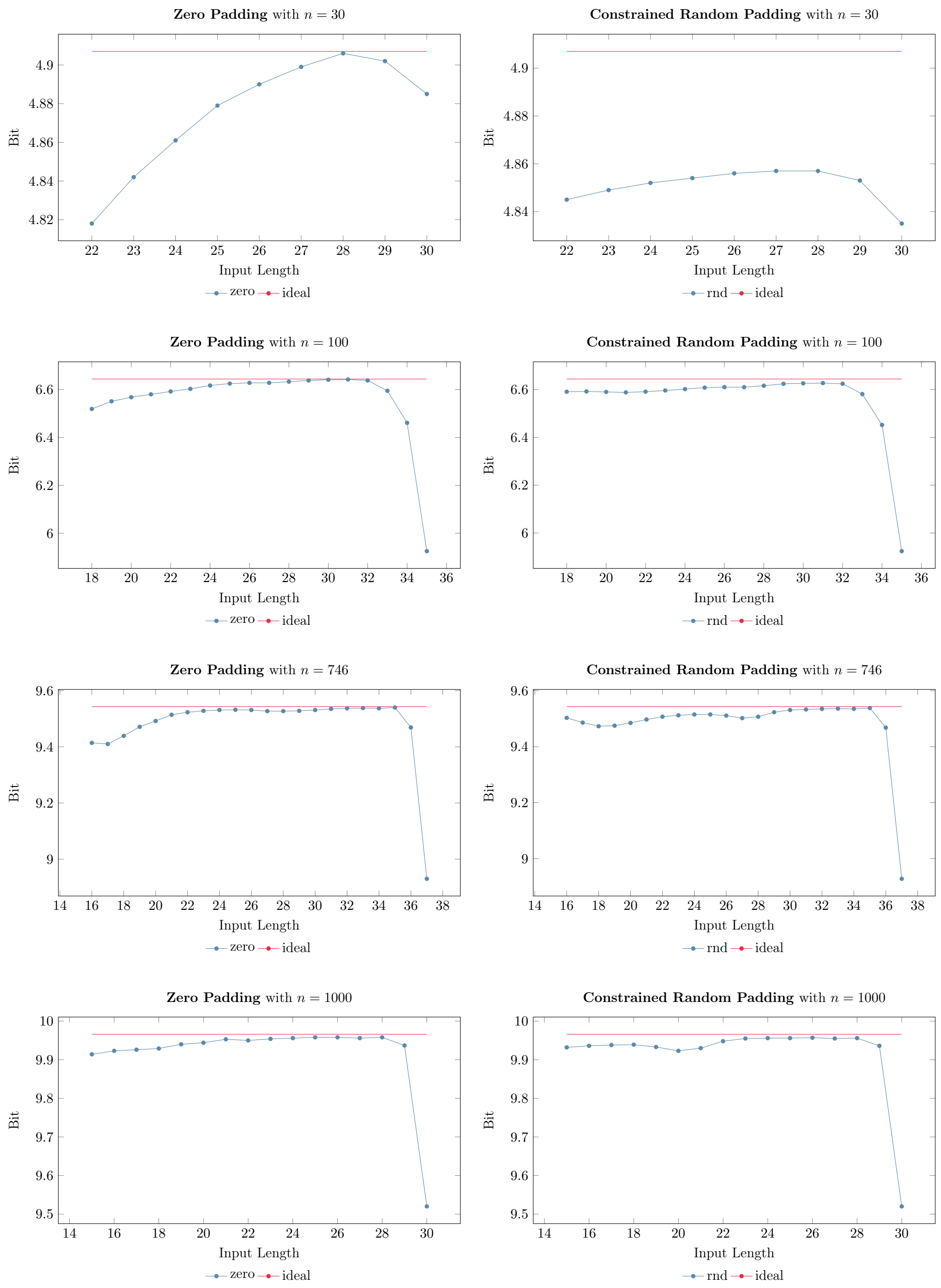


Figure 5.10: Entropy of the CWE output vector, for different input lengths computed with `adaptive_d` approach.

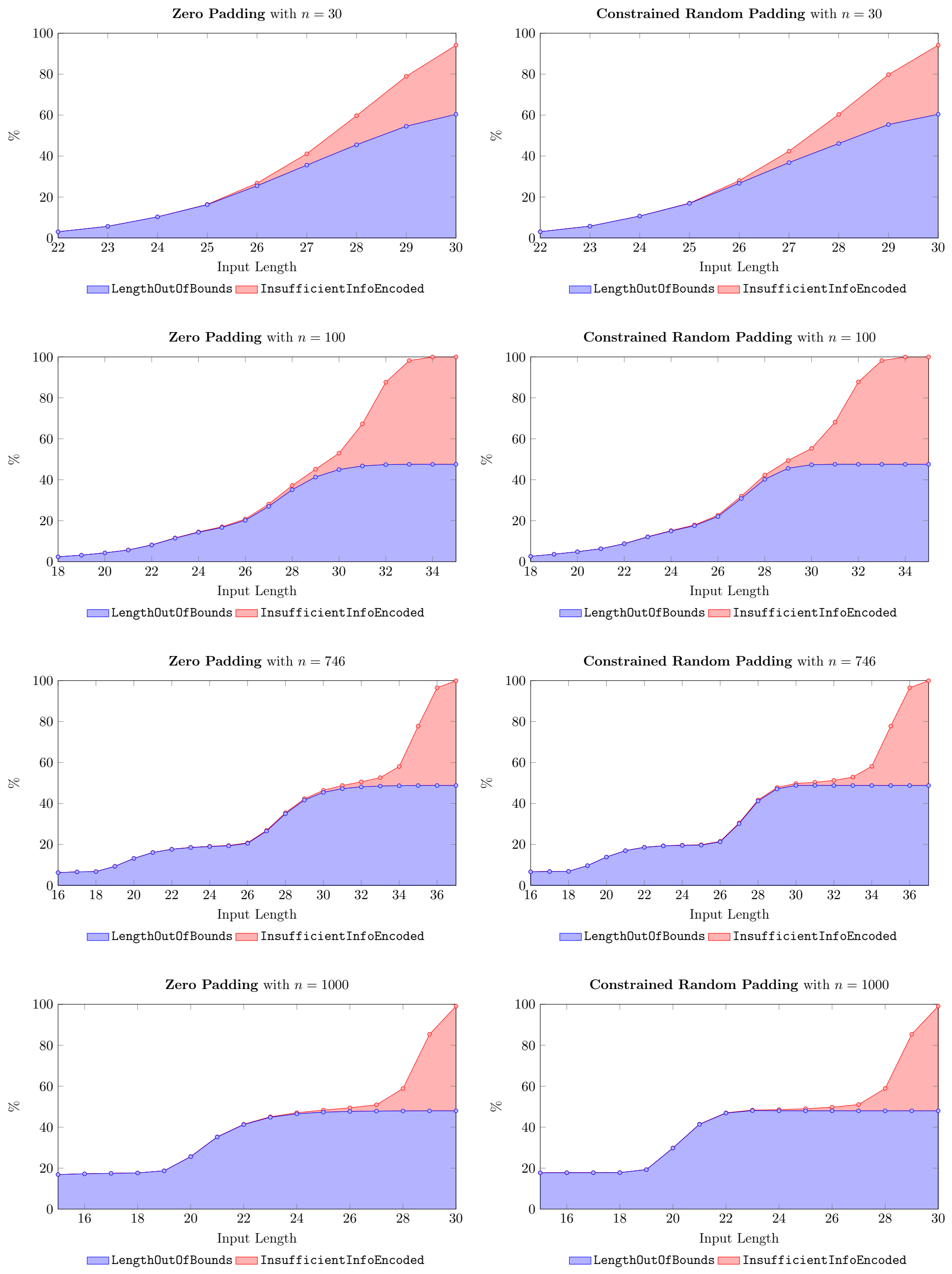


Figure 5.11: Total number of CWE failures for different input lengths computed with `adaptive_d` approach.

5.2 LEDAcrypt benchmark

This section collects LEDAcrypt benchmarks results of its three version, KEM, KEM-LT and PKC, compiled using `gcc 8.3.0` with `-O3` flag. The benchmarks results have been obtained by running both the original and optimized implementations on an Softiron Overdrive 1000 machine with a 64-bit ARM v8 architecture. It features an AMD OpteronTM A1100 Series processor with 4 ARM Cortex-A57 cores with 60 Hz frequency and 8 GB DDR4-1600 RAM.

The minimum number of operands limbs allowing to exploit Toom-3 and Karatsuba algorithms, have been tuned, by exhaustive search over the number of limbs by recording the number of cycles needed for each multiplication. For input of shorter lengths, in general, the adoption of these algorithms increment the computational time compared to the schoolbook multiplication. The results are outlined in Table 5.1 for every security level and n_0 value of LEDAcrypt. In LEDAcrypt specific case, the schoolbook multiplication is never used since the Karatsuba minimum number of limb is set to 9 and every multiplication with an operands size of 1 to 9 limbs has been specifically optimized following the work of Su and Fan [29].

For each version, key generation, encoding and decoding processes timings were taken by exploiting the `clock_gettime(clockid_t clock_id, struct timespec *tp)` function from `time.h` with `CLOCK_PROCESS_CPUTIME_ID`, which represents the CPU-time clock of the calling process, as chosen clock. The amount of required clock cycles has been computed by directly reading the `cntvct_el0` counter register value before and after the timed process. Specifically, Tables 5.2a, 5.3a and 5.4a contain data from the original implementation while Tables 5.2b, 5.3b and 5.4b have been generated from the optimized and vectorized implementation.

For ease of reading the variances for each the recorded timings have not been reported directly in the tables but summarized by their maximum percentages in the captions. Note that higher variances, specifically in the key generation process, are due to the inverse computation based on the work [18] of Kobayashi, N. Takagi and K. Takagi. With this method in fact, the verification of the generate key could required longer or shorter time depending on the position of the first set bit in the polynomial to be

Table 5.1: Minimum number of limbs of the factors for each security level of LEDAcrypt system.

(a) LEDAkem

Security Level	**P**	**N0**	**Karatsuba**	**ToomCook-3**
1 (128-bit)	14,939	2	9	9
	8,269	3	9	10
	7,547	4	9	12
3 (192-bit)	25,693	2	9	9
	16,067	3	10	11
	14,341	4	9	9
5 (256-bit)	36877	2	9	9
	27437	3	11	14
	22691	4	10	11

(b) LEDApkc and LEDAkem-LT

Security Level	**DFR**	**P**	**Karatsuba**	**ToomCook-3**
1 (128-bit)	2^{-64}	35,899	9	9
	2^{-128}	52,147	9	10
3 (192-bit)	2^{-64}	57,899	9	9
	2^{-128}	96,221	10	11
5 (256-bit)	2^{-64}	89,051	9	9
	2^{-128}	152,267	11	14

inverted.

Table 5.2: LEDAkem reference benchmarks.

(a) LEDAkem reference benchmarks, variance up to 25% (KG), 3% (E), 10%(D)

SL	**N0**	**KeyGeneration**		**Encoding**		**Decoding**		**KG+E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2	24.20	6,049	1.91	478	7.00	1,750	33.11
	3	8.09	2,022	1.52	380	9.24	2,310	18.85
	4	7.97	1,993	2.00	499	14.42	3,604	24.39
3	2	69.15	17,287	3.76	939	20.44	5,110	93.35
	3	29.42	7,356	4.08	1,019	20.62	5,154	54.12
	4	28.00	7,001	5.18	1,294	33.26	8,315	66.45
5	2	142.63	35,659	7.16	1,790	34.72	8,680	184.51
	3	86.31	21,578	7.77	1,941	41.78	10,444	135.85
	4	63.56	15,890	9.77	2,441	43.88	10,969	117.21

(b) LEDAkem optimized benchmarks, variance up to 25% (KG), 3% (E), 10%(D)

SL	**N0**	**KeyGeneration**		**Encoding**		**Decoding**		**KG+E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2	5.61	1,402	0.15	38	0.72	181	6.48
	3	1.96	490	0.09	23	0.99	247	3.04
	4	2.37	591	0.09	22	1.72	428	4.17
3	2	15.90	3,974	0.32	80	1.82	454	18.04
	3	6.67	1,668	0.21	53	2.06	516	8.95
	4	7.28	1,820	0.21	51	5.14	1,284	12.62
5	2	32.56	8,139	0.54	133	2.91	727	36.00
	3	18.68	4,671	0.48	119	3.33	831	22.49
	4	14.79	3,697	0.43	108	6.65	1,663	21.87

Table 5.3: LEDAkem Long Term optimized benchmarks for $N0 = 2$.

(a) LEDAkem Long Term reference benchmarks, variance up to 45% (KG), 0.3% (E), 30%(D)

SL	**DFR**	**KeyGeneration**		**Encoding**		**Decoding**		**E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2^{-64}	1119.60	279,902	6.72	1,679	9.87	2,468	16.59
	2^{-128}	1741.86	435,475	10.98	2,746	10.86	2,714	21.84
3	2^{-64}	3212.03	803,012	12.49	3,122	20.38	5,095	32.87
	2^{-192}	5907.30	1,476,837	26.61	6,653	25.27	6,316	51.88
5	2^{-64}	16422.29	4,105,601	22.40	5,599	43.90	10,976	66.30
	2^{-256}	16695.99	4,174,005	51.30	12,825	48.85	12,212	100.15

(b) LEDAkem Long Term optimized benchmarks, variance up to 40% (KG), 6% (E), 1%(D)

SL	**DFR**	**KeyGeneration**		**Encoding**		**Decoding**		**E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2^{-64}	1111.21	277,805	0.25	62	0.96	239	1.21
	2^{-128}	1213.07	303,268	0.43	108	1.31	327	1.74
3	2^{-64}	2176.18	544,056	0.69	172	1.84	459	2.53
	2^{-192}	3649.29	912,326	1.02	254	2.78	695	3.80
5	2^{-64}	6935.10	1,733,791	1.25	312	3.37	841	4.62
	2^{-256}	14800.58	3,700,152	1.93	482	5.37	1,343	7.30

Table 5.4: LEDApkc benchmark for $N0 = 2$ and message size of 1 kB.

(a) LEDApkc reference implementation benchmarks, variance up to 64% (KG), 1% (E), 20%(D)

SL	**DFR**	**KeyGeneration**		**Encoding**		**Decoding**		**E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2^{-64}	1382.86	345,716	7.32	1,830	12.88	3,219	20.20
	2^{-128}	1769.86	442,469	11.72	2,929	12.82	3,204	24.54
3	2^{-64}	5157.71	1,289,431	13.44	3,361	24.79	6,198	38.23
	2^{-192}	5917.38	1,479,348	27.98	6,996	27.67	6,916	55.65
5	2^{-64}	13749.40	3,437,379	23.86	5,965	50.48	12,620	74.34
	2^{-256}	16210.57	4,052,651	53.85	13,462	49.49	12,371	103.34

(b) LEDApkc optimized benchmarks, variance up to 40% (KG), 2% (E), 1%(D).

SL	**DFR**	**KeyGeneration**		**Encoding**		**Decoding**		**E+D**
		(ms)	(kcycles)	(ms)	(kcycles)	(ms)	(kcycles)	(ms)
1	2^{-64}	1114.47	278,620	1.16	291	1.79	447	2.95
	2^{-128}	1262.54	315,635	2.05	512	2.55	636	4.60
3	2^{-64}	2203.91	550,978	2.39	598	3.16	789	5.55
	2^{-192}	3663.11	915,791	3.58	894	5.06	1,264	8.64
5	2^{-64}	5308.29	1,327,076	3.40	851	5.55	1,387	8.95
	2^{-256}	12124.38	3,031,135	8.67	2,168	9.05	2,262	17.72

5.2.1 Performance Results Comment

It is evident from the results outlined in Tables 5.2a, 5.3a, 5.4a, 5.2b, 5.3b and 5.4b that the vectorization of the codebase achieved significant performances improvement. These results have been obtained mainly through the optimization and vectorization of the polynomial arithmetic operations, such as addition, shifting, inversion and multiplication, and of the Q-Decoder function as explained in Section 4.2, which was a bottleneck in the reference implementation. In particular, the Karatsuba multiplications have been expanded and vectorized for all the multiplication with a number of limbs between 1 and 9 based on [29].

The following tables, Figure 5.5, summarized the speedup gained for the three LEDAcrypt versions. Among the system primitives, the best improvement is achieved by the encoding process which in LEDAkem Long Term reference implementation took 6.72 ms while in the optimized one just 0.25 ms, thus gaining a speedup of 26.88.

Table 5.5: LEDAcrypt speedup achieved through optimization and vectorization of the codebase.

(a) LEDAkem

SL	**N0**	**KeyGeneration**	**Encoding**	**Decoding**	**KG+E+D**
1	2	4.30	12.73	9.76	5.10
	3	4.15	16.89	9.38	6.23
	4	3.43	22.11	6.80	5.23
3	2	4.34	11.78	11.35	5.18
	3	4.35	19.38	10.01	6.00
	4	3.58	24.81	5.59	4.75
5	2	4.44	13.35	11.76	5.16
	3	4.62	16.21	12.66	6.06
	4	4.18	22.72	7.09	5.43

(b) LEDAkem Long Term with N0=2

SL	**DFR**	**KeyGeneration**	**Encoding**	**Decoding**	**E+D**
1	2^{-64}	1.01	26.88	10.28	13.71
	2^{-128}	1.44	25.53	8.29	12.55
3	2^{-64}	1.48	18.10	11.08	12.99
	2^{-192}	1.62	26.09	9.09	13.65
5	2^{-64}	2.37	17.92	13.03	14.35
	2^{-256}	1.13	26.58	9.10	13.72

(c) LEDApkc with N0=2

SL	**DFR**	**KeyGeneration**	**Encoding**	**Decoding**	**E+D**
1	2^{-64}	1.24	6.31	7.20	6.85
	2^{-128}	1.40	5.72	5.03	5.33
3	2^{-64}	2.34	5.62	7.84	6.89
	2^{-192}	1.62	7.82	5.47	6.44
5	2^{-64}	2.59	7.02	9.10	8.31
	2^{-256}	1.34	6.21	5.47	5.83

Chapter 6

Conclusions

The first aim of this thesis is to analyse LEDAcrypt constant weight encoder, i.e. the algorithm which bijectively associates any arbitrary input bitstring with a vector of length n and weight t.

We started by choosing a proper set of n and t parameters: small enough to allow computations and picked to generate a vast set of CW vector in order to extract valid statistics. We observed the CWE state of the art function with the intention of determining if the CW vector distribution was uniform, as it was expected being the input uniform. After collecting the data, the conclusion was that the distribution of the `1` positions over the output space was highly skewed. In fact, when the whole input bistring has been read, the standard CWE algorithm adopts a zero padding strategy, i.e. concatenates `0` bits to the original input, which indeed translates in computing a zero distance between the 1 bits positions that are being generated and results in an accumulations of asserted bits at specific positions of the output strings. In order to mitigate the aforementioned issue, two alternative constant weight encoders were considered and their distribution analysed. These solutions attempt to improve their output distribution by incorporating some randomness into the padding step. The results, outlined in Chapter 5, confirmed, as theorized, that the constrained random padding strategy, which arbitrarily chooses the position of the missing bits among a pool of positions which guarantee a successful encoding, generates the most uniform output distribution, especially when paired with an adaptive estimation of the d parameter.

The second goal of this thesis is to improve the LEDAcrypt performances.

We choose to exploit Arm Neon technology, a SIMD architecture extension for the ARM Cortex-A series and Cortex-R52 processors in order to pursue this goal. SIMD instructions in fact allow to write code in a sequential way while gaining a parallel speed up. Since NEON extension ISA support polynomial arithmetic and especially carryless multiplications, leveraging it is particular convenient in the LEDAcrypt case.

We rewrote the most called functions in the codebase in a vectorized and optimized way using NEON intrinsics, i.e. function calls which are replaced by the compiler with the appropriate sequence of Neon assembly instructions. This means vectorizing all the polynomial arithmetic and in addition exploiting sub-quadratic multiple precision approaches, such as the Karatsuba [16] and ToomCook [6] algorithms, to perform the multiplication. The computation of multiplications between large polynomial, as the one exploited in LEDAcrypt, is in fact a time consuming process. A second point of interest was the Q-Decoder function which represents a significant bottleneck in the reference implementation and which has been completely vectorized in the optimized version.

By applying wise optimization and vectorizing techniques, such as the common loop unrolling one, we were able to reach relevant speed up of the computation for every LEDAcrypt version and primitives, respectively PKC, KEM, KEM-LT and key generation, encoding and decoding.

Appendix A

Matrix Transposition with NEON

```
uint32x4_t transpose_128(uint32x4_t matrix) {

    uint8_t C[16] = {0, 4,  8, 12,
                     1, 5,  9, 13,
                     2, 6, 10, 14,
                     3, 7, 11, 15};

    uint8x16_t transpose_vec = vld1q_u8(C);

    return vreinterpretq_u32_u8(vqtbl1q_u8(
          vreinterpretq_u8_u32(matrix), transpose_vec));
}
```

```
void transpose_128x4( uint64x2_t w0, uint64x2_t w1,
                      uint64x2_t w2, uint64x2_t w3,
                      uint64x2_t *r0, uint64x2_t *r1,
                      uint64x2_t *r2, uint64x2_t *r3 ) {

    uint32x4_t t0 = (uint32x4_t) vuzp1q_u64(w0, w2);
    uint32x4_t t1 = (uint32x4_t) vuzp1q_u64(w1, w3);
    uint32x4_t t2 = (uint32x4_t) vuzp2q_u64(w0, w2);
    uint32x4_t t3 = (uint32x4_t) vuzp2q_u64(w1, w3);

    *r0 = vreinterpretq_u64_u32(vtrn1q_u32(t0, t1));
    *r1 = vreinterpretq_u64_u32(vtrn2q_u32(t0, t1));
    *r2 = vreinterpretq_u64_u32(vtrn1q_u32(t2, t3));
    *r3 = vreinterpretq_u64_u32(vtrn2q_u32(t2, t3));
}
```

```
void transpose_8x16( uint64x2_t *x0, uint64x2_t *x1, uint64x2_t *x2,
                      uint64x2_t *x3, uint64x2_t *x4, uint64x2_t *x5,
                      uint64x2_t *x6, uint64x2_t *x7 ) {

    uint32x4_t w00, w01, w02, w03;
    uint32x4_t w10, w11, w12, w13;

    transpose_128x4(*x0, *x1, *x2, *x3, &w00, &w01, &w02, &w03);
    transpose_128x4(*x4, *x5, *x6, *x7, &w10, &w11, &w12, &w13);

    w00 = transpose_128(w00);
    w01 = transpose_128(w01);
    w02 = transpose_128(w02);
    w03 = transpose_128(w03);
    w10 = transpose_128(w10);
    w11 = transpose_128(w11);
    w12 = transpose_128(w12);
    w13 = transpose_128(w13);

    *x0 = vreinterpretq_u64_u32(vzip1q_u32(w00, w10));
    *x1 = vreinterpretq_u64_u32(vzip2q_u32(w00, w10));
    *x2 = vreinterpretq_u64_u32(vzip1q_u32(w01, w11));
    *x3 = vreinterpretq_u64_u32(vzip2q_u32(w01, w11));
    *x4 = vreinterpretq_u64_u32(vzip1q_u32(w02, w12));
    *x5 = vreinterpretq_u64_u32(vzip2q_u32(w02, w12));
    *x6 = vreinterpretq_u64_u32(vzip1q_u32(w03, w13));
    *x7 = vreinterpretq_u64_u32(vzip2q_u32(w03, w13));
}
```

Bibliography

[1] Marco Baldi. *QC-LDPC Code-Based Cryptography*. Springer Publishing Company, Incorporated, 2014.

[2] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAkem: A Post-quantum Key Encapsulation Mechanism Based on QC-LDPC Codes. In *Post-Quantum Cryptography - 9th International Conference, PQCrypto 2018, Fort Lauderdale, FL, USA, April 9-11, 2018, Proceedings*, pages 3–24, 2018.

[3] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. LEDAcrypt website. `https://www.ledacrypt.org/`, 2019.

[4] Elwyn R. Berlekamp, Robert J. McEliece, and Henk C. A. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Trans. Information Theory*, 24(3):384–386, 1978.

[5] Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. In Johannes A. Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography, Second International Workshop, PQCrypto 2008, Cincinnati, OH, USA, October 17-19, 2008, Proceedings*, volume 5299 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2008.

[6] Marco Bodrato. Towards optimal toom-cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*, pages 116–133, 2007.

[7] Thomas M. Cover. Enumerative Source Encoding. *IEEE Trans. Information Theory*, 19(1):73–77, 1973.

[8] Hang Dinh, Cristopher Moore, and Alexander Russell. Mceliece and niederreiter cryptosystems that resist quantum fourier sampling attacks. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 761–779, 2011.

[9] Nir Drucker and Shay Gueron. A toolbox for software optimization of QC-MDPC code-based cryptosystems. *IACR Cryptology ePrint Archive*, 2017:1251, 2017.

[10] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, Dec 1966.

[11] Eiichiro Fujisaki and Tatsuaki 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*, pages 537–554, 1999.

[12] Robert G. Gallager. Low-density parity-check codes. *IRE Trans. Information Theory*, 8(1):21–28, 1962.

[13] Solomon W. Golomb. Run-length encodings (Corresp.). *IEEE Trans. Information Theory*, 12(3):399–401, 1966.

[14] Stefan Heyse. Low-reiter: Niederreiter encryption scheme for embedded microcontrollers. In *Post-Quantum Cryptography, Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings*, pages 165–181, 2010.

[15] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003.

[16] Anatolii Karatsuba and Yu Ofman. Multiplication of multidigit numbers on automata. *Soviet Physics Doklady*, 7:595, 12 1962.

[17] Kazukuni Kobara and Hideki Imai. Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece PKC. In Kwangjo Kim, editor, *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 19–35. Springer, 2001.

[18] 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*, 6(3):180–185, May 2012.

[19] A L. Toom. The complexity of a scheme of functional elements realizing the multiplication of integers. *Doklady Akademii Nauk SSSR*, 3, 01 1963.

[20] Robert J. McEliece. A Public-Key Cryptosystem Based on Algebraic Coding Theory. *JPL DSN Progress Report*, 44, 1978.

[21] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[22] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *Proceedings of the 2013 IEEE International Symposium on Information Theory, Istanbul, Turkey, July 7-12, 2013*, pages 2069–2073, 2013.

[23] C. Monico, J. Rosenthal, and A. Shokrollahi. Using low density parity check codes in the McEliece cryptosystem. In *Proc. IEEE International Symposium on Information Theory (ISIT 2000)*, 2000.

[24] H. Niederreiter. Knapsack-type cryptosystems and algebraic coding theory. *Probl. Contr. and Inf. Theory*, 15:159–166, 1986.

[25] David Pointcheval. Chosen-ciphertext security for any one-way cryptosystem. In *Public Key Cryptography, Third International Workshop on Practice and Theory in Public Key Cryptography, PKC 2000, Melbourne, Victoria, Australia, January 18-20, 2000, Proceedings*, pages 129–146, 2000.

[26] Nicolas Sendrier. *Codes Correcteurs d'Erreurs à Haut Pouvoir de Correction.* Thèse de doctorat, Université Paris 6, December 1991.

[27] Nicolas Sendrier. Encoding Information into Constant Weight Words. In *Proceedings of the 2005 IEEE International Symposium on Information Theory, ISIT 2005, Adelaide, South Australia, Australia, 4-9 September 2005*, pages 435–438. IEEE, 2005.

[28] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, 1997.

[29] Chen Su and Haining Fan. Impact of intel's new instruction sets on software implementation of gf(2)[x] multiplication. *Inf. Process. Lett.*, 112(12):497–502, 2012.

[30] Ingo von Maurich, Tobias Oder, and Tim Güneysu. Implementing QC-MDPC mceliece encryption. *ACM Trans. Embedded Comput. Syst.*, 14(3):44:1–44:27, 2015.