Constant Weight Strings in Constant Time: a Building Block for Code-based Post-quantum Cryptosystems

Alessandro Barenghi[*]
Politecnico di Milano
Milano, Italy
alessandro.barenghi@polimi.it

Gerardo Pelosi
Politecnico di Milano
Milano, Italy
gerardo.pelosi@polimi.it

ABSTRACT

Code based cryptosystems often need to encode either a message or a random bitstring into one of fixed length and fixed (Hamming) weight. The lack of an efficient and reliable bijective map presents a problem in building constructions around the said cryptosystems to attain security against active attackers. We present an efficiently computable, bijective function which yields the desired mapping. Furthermore, we delineate how the said function can be computed in constant time. We experimentally validate the effectiveness and efficiency of our approach, comparing it against the current state of the art solutions, achieving three to four orders of magnitude improvements in computation time, and validate its constant runtime.

CCS CONCEPTS

• **Security and privacy** → **Public key encryption**; • **Theory of computation** → *Cryptographic primitives*.

KEYWORDS

Post-quantum cryptosystems, code-based cryptography, constant time algorithms.

ACM Reference Format:
Alessandro Barenghi and Gerardo Pelosi. 2020. Constant Weight Strings in Constant Time: a Building Block for Code-based Post-quantum Cryptosystems. In *17th ACM International Conference on Computing Frontiers (CF '20), May 11–13, 2020, Catania, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387902.3392630

1 INTRODUCTION

The strong push toward improving the current capabilities of quantum computers, with the aim of tackling significant computational problems in operating research and computational chemistry, carries the drawback of opening an avenue to breaking the integer factoring and discrete logarithm trapdoors on which most of the widely adopted asymmetric cryptographic algorithms rely. The need to design and efficiently engineer post-quantum cryptographic primitives and the related building blocks is witnessed by the activities of the US NIST with its post-quantum cryptography standardization process, started on November 2017 [13], and the EU ETSI with its working group for quantum safe cryptography, started in March 2017 [1]. Among the mathematical trapdoors resistant to attacks with quantum computers, decoding an error-affected codeword of a general linear error correcting code occupies a prominent place. Since the first use of such a trapdoor to build a public-key cryptosystems by Robert McEliece in 1978 [12], a significant amount of research developments has culminated in code-based schemes being roughly half of the encryption algorithms selected by the US NIST for the second round of its post-quantum cryptography standardization process [13].

The trapdoor function in code-based cryptosystems is built picking a linear block code admitting efficient decoding (i.e., non random) and providing an obfuscated representation of the said code as the public key. To encrypt a message, the sender either encodes it in a codeword of the obfuscated code, and intentionally adds as many errors as the code corrects, or it encodes the message as an error vector, i.e., a binary vector as long as a codeword and with a fixed Hamming weight equal to the correction power of the code. The former approach is the one proposed by McEliece in its work [12], while the latter variant was proposed by Niederreiter in 1986 [14], and proven to be as secure, provided the same obfuscated code family is employed.

These code-based trapdoors can be effectively wrapped within formal constructions to obtain either a key encapsulation mechanism (KEM) or a public key encryption scheme (PKC) with the strong guarantee of indistinguishability (of the ciphertext content) against an adaptive chosen-ciphertext attack (IND-CCA2). The most bandwidth efficient among such constructions [11] requires the PKC encryption of a McEliece cryptosystem to convert any binary stream of plaintext into the form of a predetermined Hamming weight codeword. Therefore, the McEliece cryptosystems employing such a construction share the need for a building block performing the encoding of an arbitrary, fixed length, bitstring onto a bitstring which has predetermined length and Hamming weight. The current state of the art comprises two approaches. The first solution allows a bijective mapping, which relies on the computation of large binomial coefficients [10, 15], at a cost quadratic in the input string length. The second solution, proposed by Sendrier in [17] relies on a run-length compression algorithm proposed by Golomb [9], obtaining a procedure which has linear complexity in the input string length. However, the proposal by Sendrier is affected by hard-to-predict encoding failures which force the sender to change the message being encrypted to successfully encode it in a constant weight string.

[*]Both authors contributed equally to the paper

CF '20, May 11–13, 2020, Catania, Italy
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7956-4/20/05…$15.00
https://doi.org/10.1145/3387902.3392630

Contributions. We define a new pair of maps, computing the encoding of a random binary string in a fixed-length, constant weight one, and vice-versa, both having linear computational complexity in the size of the input string length, and free from encoding failures. We prove our constructions functional, and define constraints for the parameters of the said maps stemming from the length and weight of the constant weight output string. Tackling the concerns on timing side channels present in other building blocks for post-quantum cryptosystems, we describe a constant-time implementation of the proposed maps.
Finally, we experimentally validate the constant time computation property of our maps and compare their efficiency against the failure-free alternative present in the state of the art.

2 PRELIMINARIES AND RELATED WORK

A partial function is a map between two sets where some elements of the domain may not have an image, or some elements of the codomain may not have a preimage. By contrast, a total function is a map between two sets where each element of the domain is mapped to at least an element of the codomain and each element of the codomain is mapped back to at least one element in the domain. These definitions are employed in computability theory to frame the problem of stating if the result of computing a map (resp., its inverse) on an arbitrary element of the domain (resp., of the codomain) yields a result or not. Thus, the notions of injectivity and bijectivity for an arbitrary map (partial or total) are meant to hold only over the subsets of the domain and the codomain for which the map is computable.

Definition 2.1 (String to Combination). Consider a set of binary strings S, two integers $n{>}t{\geq}1$, and a set C of combinations of t distinct elements out of n possible ones. Each element $c{\in}C$ is represented as an ordered sequence of t integers $c{=}[\gamma_0, \ldots, \gamma_{t-1}]$, each of which in $\{0, 1, \ldots, n{-}1\}$, with $\gamma_{i-1} < \gamma_i$ and $1{\leq}i{\leq}t{-}1$. A string to combination map (StC map) $\varphi{:}S{\to}C$ is defined as a (partial) $\underline{\text{bijective}}$ function from S to C.

Definition 2.2 (Ideal String to Combination). An ideal StC map is a StC map $\varphi(\cdot)$ having the following four properties:

- ***i)*** it is total over the input string set S, i.e., $\forall\, s{\in}S,\ \varphi(s){\neq}\bot\ \wedge\ \exists\,!\, c{\in}C\ s.t.\ c{=}\varphi(s)$, and $\forall\, c{\in}C,\ \exists\,!\, \bar{s}{\in}S\ s.t.\ \bar{s} = \varphi^{-1}(c)$;
- ***ii)*** it has a complexity linear in the length of the string $s{\in}S$;
- ***iii)*** it is possible to perform a uniform random sampling of the domain S with a computation complexity $O(log(|S|))$;
- ***iv)*** the entropy of a random variable $\mathcal{Z}$ over the set of bitstrings in S, is lesser or equal to the entropy of the random variable $\mathcal{W} = \varphi(\mathcal{Z})$, i.e., $\mathrm{H}(\mathcal{Z}) \leq \mathrm{H}(\mathcal{W})$.

As a consequence of bijectivity and property *i)*, an ideal StC map should have one and only one image for each element $s{\in}S$, thus yielding $|S|{=}|C|{=}\binom{n}{t}$. Moreover, since any combination $c{\in}C$, with $c{=}[\gamma_0, \ldots, \gamma_{t-1}]$, can be put in one-to-one correspondence with an n-bit binary string with t asserted bits in positions $\gamma_i{\in}\{0, 1, \ldots, n{-}1\}$, $0{\leq}i{\leq}t{-}1$, an ideal StC map can be employed to derive a *constant weight encoding* (CWE) procedure and, analogously, its inverse can be employed as a *constant weight decoding* (CWD) procedure; where the "weight" refers to the number of asserted bits $t \geq 1$, which is decided a-priori (and hence "constant").

A total string to combination map. Given $n,\ t$, with $n{>}t{\geq}1$, consider a set of binary strings S (of variable length) with $|S|{=}\binom{n}{t}$ endowed with the total order relation induced by the lexicographic order of their encoding in natural binary with the minimum number of bits. Moreover, consider the set of combinations C of t elements out of n, $|C|{=}\binom{n}{t}$, with the lexicographic order relation.

Definition 2.3 (Rank and Unrank functions). Given a finite set X with a total order relation, the *rank* function $\mathcal{R}_X{:}X{\to}\{0,\ldots,|X|{-}1\}$ is defined as a total bijective function between an element and its index (ordinal position) as established by the order relation. Conversely, the *unrank* function of the finite set X, $\mathcal{R}_X^{-1}{:}\{0,\ldots,|X|{-}1\}{\to}X$ is defined as the total function that is the inverse of $\mathcal{R}_X(\cdot)$ and maps each index back to its corresponding element in the set X.

The CWE of an element $s{\in}S$ evaluates the index of s, i.e., the integer $i{\in}\{0,\ldots,\binom{n}{t}{-}1\}$ obtained as the natural number encoded by s: $i{=}\mathcal{R}_S(s)$, and subsequently computes the combination $c{\in}C$ such that $c{=}\mathcal{R}_C^{-1}(i)$. Thus, the StC map $\varphi(\cdot)$ implemented by the CWE procedure is such that: $\forall\, s{\in}S,\ \varphi(s){=}\mathcal{R}_C^{-1}(\mathcal{R}_S(s))$. Viceversa, the CWD of an element $c{\in}C$ evaluates the index of the combination c, i.e., $j{=}\mathcal{R}_C(c)$, $j{\in}\{0,\ldots,\binom{n}{t}{-}1\}$, and computes the string $s{\in}S$ representing the integer j in binary encoding: $s{=}\mathcal{R}_S^{-1}(j)$. Thus, the inverse of the StC map $\varphi^{-1}(\cdot)$ implemented by the CWD procedure is such that: $\forall\, c{\in}C,\ \varphi^{-1}(c){=}\mathcal{R}_S^{-1}(\mathcal{R}_C(c))$.

This strategy requires a way to compute the rank and unrank functions. To this end, while $\mathcal{R}_S(\cdot)$ and $\mathcal{R}_S^{-1}(\cdot)$ are trivial to compute, the computation of $\mathcal{R}_C(\cdot)$ and $\mathcal{R}_C^{-1}(\cdot)$ was first described in [15], where it is shown that the so-called *combinatorial number system* in positional notation, is obtained from the sequence of binomial coefficients, $\binom{\mathsf{base}}{\mathsf{choose}}$, with their bases sorted in increasing order and their chooses taken as consecutive natural numbers [10].

Using the elements of $c{=}[\gamma_0, \ldots, \gamma_{t-1}]$ as the bases of a sequence of binomials with the numbers $\{1, \ldots, t\}$ as their chooses, we can compute the rank of c as $\mathcal{R}_C(c){=}\sum_{a=1}^{t}\binom{\gamma_{a-1}}{a}$. Viceversa, given a natural number $j{\in}\{0,\ldots,\binom{n}{t}{-}1\}$, the computation of the unrank function $\mathcal{R}_C^{-1}(\cdot)$, derives the values $\gamma_0, \ldots, \gamma_{t-1}$, for which the equality $j{=}\sum_{a=1}^{t}\binom{\gamma_{a-1}}{a}$ holds as described in Alg. 1.

Alg. 1 finds the largest binomial base for which $\binom{\mathsf{base}}{t} \leq j$ holds (lines 2–4), and employs it as the value of the last integer γ_{t-1} to be found (line 5). Then, Alg. 1 computes the sequence of the largest integers γ_{t-a}, for all $1{<}a{\leq}t$, such that $\binom{\gamma_{t-a-1}}{t-a} \leq a - \sum_{l=1}^{a-1}\binom{\gamma_{t-l-1}}{t-l}$ holds, deriving $c{=}[\gamma_{t-1}, \ldots, \gamma_0]$. The aforementioned unrank function $\mathcal{R}_C^{-1}(\cdot)$, yields a StC map $\varphi(s){=}\mathcal{R}_C^{-1}(\mathcal{R}_S(s))$ having only three of the four properties desired for an ideal StC map (Def. 2.2). Indeed, the said function is total (prop. *i)* in Def. 2.2) over the set S of binary strings encoding in natural binary the integer values $0{\leq}j{<}\binom{n}{t}$. This, in turn, allows to efficiently sample elements in S (prop. *iii)* in Def. 2.2), as it suffices to uniformly draw an integer in $\{0,\ldots,\binom{n}{t}{-}1\}$. Finally, the said $\varphi(\cdot)$ map also provides prop. *iv)* in Def. 2.2, as it is a bijection between S and C.

As far as Def. 2.2, prop. *ii)* is concerned, Alg. 1 shows that the computational cost of the unrank function $\mathcal{R}_C^{-1}(\cdot)$ is $O\left(t \cdot \log^2\binom{n}{t}\right)$, thus not providing the required efficiency. Indeed, while such computational complexity does not make Alg. 1 unusable to compute

Algorithm 1: $\mathcal{R}_C^{-1}:\{0,\ldots,\binom{n}{t}-1\}\to C$

```
Input: $n$, $t$: positive integers $n>t\geq 1$
       $j$: an integer number such that $0 \leq j < \binom{n}{t}$
Output: a combination of $t$ items out of $n$ denoted as
        $c=[\gamma_0,\ldots,\gamma_{t-1}]$, where $\gamma_a \in \{0,1,\ldots,n-1\}$,
        $\gamma_{a-1} < \gamma_a$ and $1\leq a\leq t-1$
1  base $\leftarrow t-1$
2  while $\binom{\text{base}}{t} \leq j$ do
3      base $\leftarrow$ base $+ 1$
4  base $\leftarrow$ base $- 1$; $\gamma_{t-1} \leftarrow$ base
5  $j \leftarrow j - \binom{\text{base}}{t}$
6  for $a \leftarrow t-1$ to $1$ do
7      while $\binom{\text{base}}{a} \geq j$ do
8          base $\leftarrow$ base $- 1$
9      $\gamma_{a-1} \leftarrow$ base; $j \leftarrow j - \binom{\text{base}}{a}$
10 return $[\gamma_0,\ldots,\gamma_{t-1}]$
```

Algorithm 2: Non ideal $\varphi^{-1}: C \to S$ as in [17]

```
Input: $c=[\gamma_0,\ldots,\gamma_{t-1}]$: combination in $C$, $\gamma_i \in \{0,\ldots,n-1\}$,
       $0\leq i\leq t-1$.
Output: s: binary string in $S=\{0,1\}^*$
1  offset $\leftarrow 0$
2  for $i \leftarrow 0$ to $t-1$ do
3      $d \leftarrow \left\lfloor \frac{n-\text{offset}}{2(t-i)\log(2)} \right\rceil$, $u \leftarrow \lceil \log_2(d) \rceil$
4      $\lambda_i \leftarrow \gamma_i -$ offset
5      $q \leftarrow \left\lfloor \frac{\lambda_i}{d} \right\rfloor$, $r \leftarrow \lambda_i \bmod d$
6      s $\leftarrow$ Concat($s$, $1^q0$)
7      if $r < 2^u$ then
8          $s \leftarrow$ Concat($s$, IntegerToNaturalBinary($r$, $u-1$))
9      else
10         $s \leftarrow$ Concat($s$, IntegerToNaturalBinary($r+2^u$, $u$))
11     offset $\leftarrow$ offset $+ \lambda_i + 1$
12 return s
```

the map $\mathcal{R}_C^{-1}(\cdot)$, it represents a significant performance hog for values of n and t employed for cryptographic purposes on both general purpose and embedded platforms, e.g., $10^3<n<10^5$, $50<t<250$.

Approximated bitstring to combination map. In [17], Sendrier proposes an approximated StC map between the set of all binary strings $S=\{0,1\}^*$, and the finite set C of combinations of t elements chosen out of n ones (with $n>t\geq 1$). The map described in [17], while being a partial function from S to C, is also bijective (on the elements which do have an image). However, its definition does not match the one of an ideal StC map. In [17], the only way to determine an element $s\in S$ for which $\varphi(s)\neq\perp$ is to enumerate all the elements in $\{\forall\, c \in C,\ \varphi^{-1}(c)\}$, i.e., computing the inverse StC map on all combinations.

The algorithm reported in [17] to compute $\varphi^{-1}(c)$, $c\in C$ relies on representing $c=[\gamma_0,\ldots,\gamma_{t-1}]$ with $\gamma_i\in\{0,1,\ldots,n-1\}$, $\gamma_{i-1}<\gamma_i$, $1\leq i\leq t-1$, as an n-bit string, $\mathtt{str}(c)$, with t asserted bits in the positions specified by $[\gamma_0,\ldots,\gamma_{t-1}]$, and maintains that, in practical cases, $1<t\ll n$. In these cases, $\mathtt{str}(c)$ contains long sequences (a.k.a. *runs*) of null bits, thus the run-length encoding technique by Golomb [9] can be used to represent $\mathtt{str}(c)$ in a more compact way. Since the technique in [9] is optimal in terms of compression ratio, the work in [17] assumes that the compact representation of the elements $\mathtt{str}(c)$, $c\in C$ will allow the map $\varphi^{-1}(\cdot)$ to somehow densely cover a subset of $S=\{0,1\}^*$ containing binary strings up to some length. This intuition is used to justify the fact that drawing a random binary string, $\widehat{s}\in S$, shorter than the said length will likely yield $\varphi(\widehat{s})\neq\perp$.

To compute the inverse StC map $\varphi^{-1}(\cdot)$, Golomb in [9] puts in one-to-one correspondence $\mathtt{str}(c)$ with the sequence of integers, $[\lambda_0,\ldots,\lambda_{t-1}]$, $\lambda_i\in\{0,\ldots,n-t\}$, representing the counting of null bits preceding each asserted bit, in a scan of $\mathtt{str}(c)$ from left to right. Each integer value in $[\lambda_0,\ldots,\lambda_{t-1}]$ is subsequently encoded with the aim of obtaining a bitstring shorter than $\mathtt{str}(c)$.

For example, consider the combination $c=[0,4,7,8]$ of $t=4$ elements chosen among the ones in $\{0,1,2,3,\ldots,8\}$ ($n=9$).

The combination c is associated to the binary string `100010011`, for which the sequence of the zero-run-lengths to be efficiently encoded is $[0,3,2,0]$.

To achieve the best possible compression, Golomb in [9] suggests to choose a positive integer d and to encode each zero-run-length λ_i as a string $\mathtt{Gol}(\lambda_i)$ obtained as the concatenation of two binary strings corresponding to the quotient and the remainder of the zero-run-length value divided by d, i.e., given λ_i with $0\leq\lambda_i\leq n-t$, the quotient and remainder are: $q_i=\lfloor\frac{\lambda_i}{d}\rfloor$, $r_i=\lambda_i-q_i\cdot d$.

The quotient q_i is encoded in *unary* with a trailing zero as a delimiter (i.e., $\mathtt{unary}(q_i)$=111 . . . 110, with q_i 1s), while the remainder r_i (with $0\leq r_i<d$) is encoded employing the prefix-free *truncated binary* notation, i.e., assuming $u=\lceil\log_2(d)\rceil$; if $r_i<2^u-d$ then the string $\mathtt{str}(r_i)$ is computed as the natural binary encoding of r_i with $u-1$ bits; otherwise, being $r\geq 2^u-d$, the string $\mathtt{str}(r_i)$ is obtained as the natural binary encoding of r_i+2^u-d with u bits. Note that the truncated binary encoding, in case the value of d is a power of two coincides with the natural binary encoding over $\log_2(d)$ bits.

In [9] Golomb states that the value of d, should be chosen as the median value of the random variable modeling the lengths of the zero-runs. In [17], the author picks $d=\frac{n}{2t\log(2)}$, showing that it minimizes the expected length of the binary string computed by $\varphi^{-1}(\cdot)$. Indeed, [17] assesses the compression efficiency of the approach considering the ratio between the entropy of a uniformly distributed variable modeling the choice of a combination in C and the entropy of the random variable $\mathcal{L}=\mathtt{length}(\varphi^{-1}(C))$, defined over the lengths of elements in a subset of $S=\{0,1\}^*$.

In a further attempt to improve this figure of merit, i.e., bringing it closer to one, [17] suggests to recompute $d=\frac{n}{2t\log(2)}$ after encoding each run-length λ_i in $[\lambda_0,\ldots,\lambda_{t-1}]$ – replacing n with the number of remaining bits and t with the remaining asserted bits. An operative description of the inverse StC map, $\varphi(\cdot)^{-1}$, is showed by Alg. 2.

The StC map $\varphi: S\to C$, with $S=\{0,1\}^*$ proposed in [17] is thus derived as Golomb's zero-run-length decoding of an element $s\in S$ and showed by Alg. 3, where the input binary string s is scanned as the sequence of t Golomb-encoded zero-run-lengths.

Algorithm 3: Non ideal $\varphi : S \to C$ as in [17]

```
Input: s: binary string in $S$={0, 1}$^*$.
Output: $c=[\gamma_0, \ldots, \gamma_{t-1}]$: combination in $C$, or $\perp$ (error)
Data: SplitAndPad(arg1, $x$) takes a binary string arg1 and a
      number $x$ and splits arg1 in a $x$ long prefix and the
      remaining suffix, returning them. If the length of arg1 is
      insufficient, it adds arbitrary padding to arg1.
offset ← 0
for $i \leftarrow 0$ to $t-1$ do
    $q \leftarrow 0$ // decode of the unary encoded part of $i$-th zero-run-length
    do
        (f, s) ← SplitAndPad(s, 1)
        $q \leftarrow q$ + NaturalBinaryToInteger(f)
    while f = 1
    $d \leftarrow \left\lfloor \frac{n-\text{offset}}{2(t-i)\log(2)} \right\rceil$, $u \leftarrow \lceil \log_2(d) \rceil$
    $r \leftarrow 0$ // decode of the trunc. bin. part of $i$-th zero-run-length
    if $u - 1 > 0$ then
        (f, s) ← SplitAndPad(s, $u-1$)
        if NaturalBinaryToInteger(f) < $2^u - d$ then
            $r$ ← NaturalBinaryToInteger(f)
        else
            (f′, s) ← SplitAndPad(s, 1)
            f ← Concat(f, f′)
            $r$ ← NaturalBinaryToInteger(f′) − $(2^u - d)$
    $\lambda \leftarrow q \cdot d + r$
    $\gamma_i$ ← offset + $\lambda$
    if $\gamma_i \geq n$ then return $\perp$
    offset ← $\gamma_i + 1$
return $[\gamma_0, \ldots, \gamma_{t-1}]$
```

To this end, Alg. 3 iteratively decodes the unary representation of a quotient q (lines 3–7), estimates the value of d according to the said improved criterion (line 8) and decodes the truncated binary representation of the following remainder r (lines 9–17). Finally, the value of the zero-run-length λ is reconstructed (line 18) and employed to derive the element γ_i of the combination c at the i-th iteration (line 19). If $\gamma_i \geq n$ the computation of the combination fails (lines 20–21).

As shown by Alg. 3, the StC map proposed $\varphi(\cdot)$ in [17] is efficiently computable, as it has a linear computation complexity in the lenght of the input binary string (prop. *ii)* in Def. 2.2).

However, we mantain that such a map is characterized by two issues providing a hindrance to its practical use.

Issue a) Note that Alg. 3 does not need to read the entire input string before a combination is returned. Indeed, it is possible that only a proper prefix p of a binary input string $s = \text{Concat}(p, u), u \in \{0, 1\}^+$ is a valid encoding of t zero-run-lengths. In this case, all the binary input strings sharing the same prefix p will be mapped to the same combination, effectively breaking the injectivity of the map. In other words, $\varphi(\cdot)$ in [17] is non computable for all the strings s=$\text{Concat}(p, u)$, with $p \in \{0, 1\}^*$, $u = \{0, 1\}^+$ for which one of their proper prefixes, p, is computable: $\varphi(p) \neq \perp, p \in S = \{0, 1\}^* \Rightarrow \forall u \in \{0, 1\}^+, \varphi(\text{Concat}(p, u)) = \perp$.
This, in turn, increases the difficulty of efficiently and uniformly sampling from the domain S (prop. *iii)* in Def. 2.2), since it is hard to test, at sampling time, for the potential encodability of the prefixes of the string that would lead to a non-encodability of the string itself. Note that, if this injectivity preserving measure is not applied, and the result of $\varphi(\text{Concat}(p, u))$ is defined as $\varphi(\text{Concat}(p, u)) = \varphi(p)$, a violation of prop. *iv)* in Def. 2.2 will happen, as all $\text{Concat}(p, u)$ strings have the same image.

Issue b) Picking an arbitrary bitstring $s \in \{0, 1\}^*$ as input to $\varphi(\cdot)$, may result in the values γ_i, $0 \leq i \leq t-1$, of the output combination $c = [\gamma_0, \ldots, \gamma_{t-1}]$ exceeding $n-1$ (lines 20–21, in Alg. 3). This, in turn, makes the map non ideal due to the lack of prop. *i)* in Def. 2.2. We note that, even if the map is defined to be total over a subset of $S \subset \{0, 1\}^*$ by rejecting the elements for which it is not computable, there will still be no efficient method to randomly and uniformly sample an element in S, implying the lack of prop. *iii)* in Def. 2.2.

3 IDEAL STRING TO COMBINATION MAP CONSTRUCTION

We define our StC map $\chi : S' \rightsquigarrow C$ as a randomized total bijective function from the set of binary strings $S' = \{0, 1\}^l$ having length $l \geq 1$, to the set of combinations, C, of t elements out of n, with $n > t \geq 1$.

To reconcile the bijectivity property with the fact that $|S'| = 2^l$, $|C| = \binom{n}{t}$, and $|S'| \neq |C|$ for most choices of the n, t parameters, the definition of $\chi(\cdot)$ is obtained composing two maps ψ and φ, and keeping $2^l < \binom{n}{t}$.

Specifically, the StC map χ and its inverse χ^{-1} are obtained as: $\chi = \psi \circ \varphi$, that is $\chi(\cdot) = \varphi(\psi(\cdot))$, and $\chi^{-1} = \varphi^{-1} \circ \psi^{-1}$, that is $\chi^{-1}(\cdot) = \psi^{-1}(\varphi^{-1}(\cdot))$.

The total bijective map $\varphi : S^{\not\perp} \to C$, $S^{\not\perp} = \{s \in \{0, 1\}^* \text{ s.t. } \varphi(s) \neq \perp\}$, is obtained as the Golomb's zero-run-length decoding of an element $s \in S^{\not\perp}$, i.e., feeding Alg. 3 with an input binary string interpreted as a sequence of t Golomb-encoded zero-run-lengths.

The total and invertible map $\psi : S' \rightsquigarrow S^{\not\perp}$, with $S' \subset S^{\not\perp}$, from the set S' of l-bit strings ($l \geq 1$) to the set $S^{\not\perp}$ is a randomized function yielding a different element of $S^{\not\perp}$ each time it is computed. Each $s' \in S'$ is mapped to an element of $S^{\not\perp}$ concatenating a randomly chosen binary suffix to it. The invertibility of ψ is obtained by mapping any $s \in S^{\not\perp}$ onto an $s' \in S'$, by truncating s to its first l bits.

As far as the properties of an ideal StC map stated in Def. 2.2 are concerned, we note that property *i)* is achieved with our definition by ensuring that ϕ and φ are total functions and by devising $\chi : S' \rightsquigarrow S^{\not\perp} \to C$ as $\chi = \psi \circ \varphi$, with $\psi : S' \rightsquigarrow S^{\not\perp}$ and $\varphi : S^{\not\perp} \to C$. Furthermore, it is easy to recognize that, the proposed definition satisfies also property *iii)* as the domain S' of $\chi(\cdot)$ and $\psi(\cdot)$ is the set of constant-length strings with $l \geq 1$ bits. Indeed, a simple method to draw a string in S' with uniform distribution can be easily implemented with a computational complexity linear in l. Finally, properties *ii)* and *iv)* will be proven after the operative definitions of ψ and φ reported below.

3.1 Designing $\psi(\cdot)$ and $\varphi(\cdot)$

The key issue in designing the total map $\psi : S' \rightsquigarrow S^{\not\perp}$ is to guarantee that its codomain matches $S^{\not\perp} = \{s \in \{0, 1\}^* \text{ s.t. } \varphi(s) \neq \perp\}$. To this end, we note that Alg. 3 may fail to compute a valid combination starting from randomly drawn l-bit string in $\{0, 1\}^*$ as highlighted at the end of Section 2 namely when *a)*, a proper prefix p of $s = \text{Concat}(p, u)$ admits an image under φ (i.e., $\varphi(s) \neq \perp$), and *b)*,

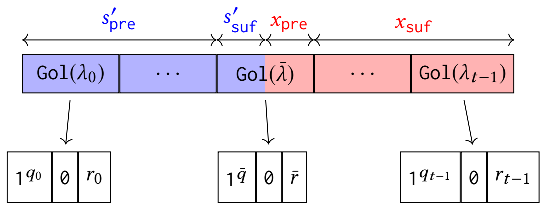
s'_{pre}
s'_{suf}
x_{pre}
x_{suf}
$\texttt{Gol}(\lambda_0)$
$\cdots$
$\texttt{Gol}(\bar\lambda)$
$\cdots$
$\texttt{Gol}(\lambda_{t-1})$
1^{q_0} | 0 | r_0
$1^{\bar q}$ | 0 | $\bar r$
$1^{q_{t-1}}$ | 0 | r_{t-1}

Figure 1: Diagram of an element of $\textsc{Concat}(s', x) \in S^{\not\perp}$ and the corresponding interpretation by φ, with $\lambda_i = q_i d + r_i$

when the sum of the zero-run-lengths λ_i, $0{\le}i{\le}t{-}1$, in the string $\texttt{str}(c)$, $c{=}\varphi(s)$, exceeds $\sum_{i=0}^{t-1} \lambda_i{>}n{-}t$.
We design the randomized map $\psi{:}S'{\rightsquigarrow}S^{\not\perp}$ as $s = \psi(s'){=}\textsc{Concat}(s', x)$, where $s'{\in}S'{=}\{0, 1\}^l$ and $x{\in}\bigcup_{a=0}^{m}\{0, 1\}^a$ is a randomly picked binary string with length at most m, with $m{\ge}0$. For such a ψ to exist we need to determine an appropriate range for the values of l, such that it is possible, for any $s'{\in}S'$ to find at least one suffix x such that $s{=}\textsc{Concat}(s', x){\in}S^{\not\perp}$, where $S^{\not\perp}{=}\{s{\in}\{0, 1\}^*$ s.t. $\varphi(s){\ne}\perp\}$.
To this end, we consider the string s as the concatenation of the strings $\texttt{Gol}(\lambda_i)$, $0{\le}i{\le}t{-}1$ as depicted in Fig. 1. Each $\texttt{Gol}(\lambda_i)$ is the binary string containing the Golomb encoding for the i-th zero-run length of $\texttt{str}(c)$, $c{=}\varphi(s)$. We denote with $\Lambda(s), s{\in}S^{\not\perp}$ the sum $\sum_{i=0}^{t-1} \lambda_i$, obtained performing the Golomb decoding of each binary sequence $\texttt{Gol}(\lambda_i)$, $0{\le}i{\le}t{-}1$ composing s (i.e., the zero-run-lengths obtained during the computation of $\varphi(s)$).

Our constraint to fix the first issue, named *constraint-a* requires to pick a value l lesser or equal to the minimum length of an element in $S^{\not\perp}$, while we remedy the second issue imposing *constraint-b*, i.e., requiring that, for any $s' \in S'$ there must be at least a suffix such that $\textsc{Concat}(s', x) \in S^{\not\perp}$, therefore mandating $\Lambda(\textsc{Concat}(s', x)){\le}n{-}t$.
To determine the effect of *constraint-a* on the value of l, given a choice of n, t and d (a whished median run-length value), we prove:

Lemma 3.1. *The length of the shortest element in $S^{\not\perp}$ as a function of d is $t \cdot (1 + \lfloor\log_2(d)\rfloor)$ bits long.*

Proof. Consider one of the t binary sequences $\texttt{Gol}(\lambda_i)$ composing $s \in S^{\not\perp}$. Such a sequence can be split into three portions: the quotient q_i encoded in unary as $1^{\lfloor \frac{\lambda_i}{d} \rfloor}$, the $\texttt{0}$ stopbit and the remainder r_0 encoded in truncated binary over either $\lfloor\log_2(d)\rfloor$ or $\lfloor\log_2(d)\rfloor{+}1$ bits. Therefore, the shortest possible valid binary sequence representing $\texttt{Gol}(\lambda_i)$ is $1 + \lfloor\log_2(d)\rfloor$ bits long and encodes a null q_0 and $r_0{<}2^{\lfloor\log_2(d)\rfloor}$. Since s is obtained as the concatenation of t $\texttt{Gol}(\lambda_i)$ sequences, a lower bound on the length of $s \in S^{\not\perp}$ is $t \cdot (1 + \lfloor\log_2(d)\rfloor)$. To show that at least an $s \in S^{\not\perp}$ matches the lower bound on the length, we observe that $s{=}\texttt{0}^{1+\lfloor\log_2(d)\rfloor}$ complies with the constraint $\Lambda(s){\le}n{-}t$, since $\Lambda(s){=}0$ and is the sequence of Golomb encodings of all null zero-run-lengths λ_i, $0{\le}i{\le}t{-}1$. □

As a consequence of Lemma 3.1 the following relation must hold:

$$l \le t \cdot (1 + \lfloor\log_2(d)\rfloor) \qquad (1)$$

From now on, we denote with σ the number of zero-run-lengths which have their Golomb encodings entirely contained in s' (in blue in Fig. 1), and with $\bar\lambda$ the value of the zero-run-length which has its Golomb encoding, $\texttt{Gol}(\bar\lambda)$, split across a suffix of s' and a prefix of x. As shown in Fig. 1, considering $s'{=}\textsc{Concat}(s'_{\text{pre}}, s'_{\text{suf}})$ and $x{=}\textsc{Concat}(x_{\text{pre}}, x_{\text{suf}})$, $\texttt{Gol}(\bar\lambda) = \textsc{Concat}(s'_{\text{suf}}, x_{\text{pre}})$. Note that it may be possible for such a $\bar\lambda$ to be missing, in case the Golomb encoding, $\texttt{Gol}(\lambda_i)$, of each one of the zero-run-lengths, λ_i, $0{\le}i{\le}t{-}1$, is fully contained in either s' or x.
To determine the effect of *constraint-b* on the value of l, given a choice of n, t, d, we prove that for all $s'{\in}S'$ there is at least a suffix x such that the inequality $\Lambda(\textsc{Concat}(s', x)){\le}n{-}t$ holds. To this end, we consider the suffixes x entirely constituted of null bits, $x{=}\texttt{0}^m$, $m{\ge}0$, noting that $\Lambda(\textsc{Concat}(s', x)){\le}n{-}t$ can be rewritten as $\Lambda(\textsc{Concat}(s', \texttt{0}^m)){=}\sum_{i=0}^{\sigma-1} \lambda_i{+}\bar\lambda{\le}n{-}t$ since all the Golomb encodings $\texttt{Gol}(\lambda_i)$, $i{>}\sigma$ (or $\texttt{Gol}(\lambda_i)$ following $\texttt{Gol}(\bar\lambda)$) are filled with null bits (thus encoding $\lambda_i{=}0$). Therefore, the only non null contribution to $\Lambda(\textsc{Concat}(s', x))$ comes from $\texttt{Gol}(\lambda_0), \ldots, \texttt{Gol}(\lambda_{\sigma-1})$ and $\texttt{Gol}(\bar\lambda)$. We now find the element $s' \in S'{=}\{0, 1\}^l$ with the maximum value of $\Lambda(\textsc{Concat}(s', \texttt{0}^m))$. Thus,
$\Lambda(\textsc{Concat}(s'', \texttt{0}^m)){\le}\Lambda(\textsc{Concat}(s', \texttt{0}^m)){\le}n{-}t$ for any other $s''{\in}S'$.

Lemma 3.2. *Assuming $s'{\in}S'{=}\{0, 1\}^l$, $l{\ge}1$, the value of s' maximizing $\Lambda(\textsc{Concat}(s', \texttt{0}^m))$ is $s'{=}1^l$.*

Proof. Picking $s' \in S' = \{0, 1\}^l$ so that $\textsc{Concat}(s', \texttt{0}^m) = \textsc{Concat}(\texttt{Gol}(\lambda_0), \ldots, \texttt{Gol}(\lambda_{\sigma-1}), \texttt{Gol}(\bar\lambda), \ldots, \texttt{Gol}(\lambda_{t-1}))$ has $\sigma{=}0$ leads to consider $\textsc{Concat}(s', \texttt{0}^m) = \texttt{Gol}(\bar\lambda) = \texttt{Gol}(\lambda_0)$ and consequentially $\Lambda(\textsc{Concat}(s', \texttt{0}^m)){=}\bar\lambda{=}\bar q{\cdot}d{+}\bar r$.
To show that $\Lambda(\textsc{Concat}(1^l, \texttt{0}^m)) \ge \Lambda(\textsc{Concat}(s', \texttt{0}^m))$, we distinguish three possible cases: s' contains only part of the unary encoding of $\bar q$, s' contains the full encoding of $\bar q$ including the stopbit and s' contains the full unary encoding of $\bar q$ plus part of the encoding of $\bar r$. In the first case, 1^l coincides with s', hence trivally $\Lambda(\textsc{Concat}(1^l, \texttt{0}^m)){=}\Lambda(\textsc{Concat}(s', \texttt{0}^m)){=}l{\cdot}d$. In the second case, $s'{=}1^{l-1}\texttt{0}$, thus $\Lambda(\textsc{Concat}(s', \texttt{0}^m)){=}(l{-}1){\cdot}d{<}l{\cdot}d$. In the third case, $s'{=}\textsc{Concat}(1^{\bar q}, 0, \{0, 1\}^a)$, $0{<}a{\le}\lfloor\log_2(d)\rfloor$, therefore we can rewrite $l{=}\bar q{+}1{+}a$. We thus have $\Lambda(\textsc{Concat}(1^l, \texttt{0}^m)){=}d{\cdot}(\bar q{+}1{+}a){>}\bar q{\cdot}d{+}(d{-}1){\ge}\bar q{\cdot}d{+}r$.

Now, picking $s' \in S' = \{0, 1\}^l$ such that $\textsc{Concat}(s', \texttt{0}^m) = \textsc{Concat}(\ldots, \texttt{Gol}(\lambda_{\sigma-1}), \texttt{Gol}(\bar\lambda), \texttt{Gol}(\lambda_{\sigma+1}), \ldots, \texttt{Gol}(\lambda_{t-1}))$ exhibits $\sigma{>}0$ allows to prove that $\Lambda(\textsc{Concat}(1^l, \texttt{0}^m)) \ge \Lambda(\textsc{Concat}(s', \texttt{0}^m))$ by showing that $\Lambda(\textsc{Concat}(s'', \texttt{0}^m)) \ge \Lambda(\textsc{Concat}(s', \texttt{0}^m))$ for any s'' obtained replacing one or more portions of s' identified as $\texttt{Gol}(\lambda_i)$ $0{\le}i{\le}\sigma{-}1$ with a corresponding string $1^{\texttt{length}(\texttt{Gol}(\lambda_i))}$, or in the case of $\texttt{Gol}(\bar\lambda)$ which is split across s' and $\texttt{0}^m$, s'' is equal to s' up to $\texttt{Gol}(\lambda_{\sigma-1})$ and differs from it in the value of s'_{suf} (see Fig. 1), which is replaced $1^{\texttt{length}(s'_{\text{suf}})}$.

The latter case is a straightforward consequence of the case $\sigma{=}0$ since s'_{suf} with $1^{\texttt{length}(s'_{\text{suf}})}$ has no impact on the values λ_i, $0{\le}i{\le}\sigma{-}1$ therefore $\Lambda(\textsc{Concat}(s'', \texttt{0}^m)) =$
$\sum_{i=0}^{\sigma-1} \lambda_i + d \cdot \texttt{length}(s'_{\text{suf}}){\ge}\sum_{i=0}^{\sigma-1} \lambda_i{+}\bar\lambda = \Lambda(\textsc{Concat}(s', \texttt{0}^m))$.
The former case is proven analyzing the value $\Lambda(\textsc{Concat}(s'', \texttt{0}^m))$. Consider the case where the Golomb encoding being replaced with a sequence of ones is $\texttt{Gol}(\lambda_j)$, $0{\le}j{\le}\sigma{-}1$. Replacing $\texttt{Gol}(\lambda_j)$ with $1^{\texttt{length}(\texttt{Gol}(\lambda_j))}$ results in an alteration of the value of λ_{j+1}, which is increased by $d \cdot \texttt{length}(\texttt{Gol}(\lambda_j))$ since the sequence of ones replacing $\texttt{Gol}(\lambda_j)$ is interpreted as part of the unary encoded quotient q_{j+1}. Besides incrementing the value of λ_{j+1}, the string replacement has also the effect of reducing σ by one, since replacing $\textsc{Concat}(\texttt{Gol}(\lambda_j), \texttt{Gol}(\lambda_{j+1}))$ with the string obtained as

$\textsc{Concat}(1^{\texttt{length}(\texttt{Gol}(\lambda_i))}, \texttt{Gol}(\lambda_{j+1}))$ replaces a string interpreted as two Golomb encodings by one interpreted as a single one, thus $\Lambda(\textsc{Concat}(s'', 0^m))=\sum_{i=0}^{j-1}\lambda_i+d\cdot\texttt{length}(\texttt{Gol}(\lambda_j))+\sum_{i=j+1}^{\sigma-1}\lambda_i$. Noting that $\Lambda(\textsc{Concat}(s'', 0^m)) - \Lambda(\textsc{Concat}(s', 0^m)) = d\cdot\texttt{length}(\texttt{Gol}(\lambda_j))-\lambda_j$, the statement would be proven if the previous quantity is greater or equal to zero. To this end, it is sufficient to observe that $\lambda_j=q_j\cdot d+r_j$ (with $r_j<d$) and that $\texttt{length}(\texttt{Gol}(\lambda_j))$ is at least $q_j+1+\lfloor\log_2(d)\rfloor$ to conclude that $d\cdot q_j+d+d\cdot\lfloor\log_2(d)\rfloor-d\cdot q_j+r_j\geq 0$. □

Since $s'=1^l$ implies $\Lambda(\textsc{Concat}(s', 0^m))=l\cdot d$, the following holds:

$$l\cdot d\leq n-t \tag{2}$$

Given the integers $n>t\geq 1$, keeping into account Eq. 1 and Eq. 2 the choice of the integer parameter d, employed in the computation of the map φ, must satisfy the following relation:

$$2^{l/t-1}-1\leq d\leq\frac{n-t}{l} \tag{3}$$

Therefore, given the n, t, l parameters chosen by the use case, employing the aforementioned constraint, we obtain a range of values for d, guaranteeing that at least a suffix x exists such that $\varphi(\textsc{Concat}(s', x))$ is computable. This fact jointly with the bijectivity of $\varphi : S^{\measuredangle} \to C$ guarantees that it is total. Furthermore, the existence of a suffix x for any string s' in $S'=\{0,1\}^l$ such that $\varphi(\textsc{Concat}(s', x))\in S^{\measuredangle}$, proves that building a total $\psi : S' \rightsquigarrow S^{\measuredangle}$ is possible. As a consequence, the map $\chi : S' \rightsquigarrow C$, $\chi=\psi\circ\varphi$ is total, since it is defined as composition of total functions (prop. *i)* in Def. 2.2), and it is also efficiently samplable given the constant-length nature of strings in S' (prop. *iii)* in Def. 2.2.

To complete the definition of $\psi : S' \rightsquigarrow S^{\measuredangle}$, we detail how to pick uniformly at random a suffix x for a string s' among the ones such that $\textsc{Concat}(s', x)\in S^{\measuredangle}$. To this end, the only requirement which x must satisfy is that $\Lambda(\textsc{Concat}(s', x))\leq n-t$. Given s', we compute the value of x one $\texttt{Gol}(\lambda_i)$ at a time, starting from $\texttt{Gol}(\bar{\lambda})$.

For all $\texttt{Gol}(\lambda_j)$, $\sigma+1\leq j\leq t-1$, that are fully contained in x, the value of each λ_j can simply be randomly drawn over $\{0,\ldots,(n-t)-\sum_{i=0}^{j-1}\lambda_i\}$, as this complies with the constraint $\Lambda(\textsc{Concat}(s', x))\leq n-t$. The value for $\bar{\lambda}$ cannot be freely drawn over $\{0,\ldots,(n-t)-\sum_{i=0}^{\sigma-1}\lambda_i\}$, since part of the value of $\bar{\lambda}$ is determined by the trailing part of s', i.e., s'_{suf}.
We draw a value uniformly over the interval $\{0,\ldots,(n-t)-\left(\sum_{i=0}^{\sigma-1}\lambda_i\right)-y$, where y is the value Golomb encoded as $\textsc{Concat}(s'_{\mathsf{suf}}, 0^{|x_{\mathsf{pre}}|})=\texttt{Gol}(y)$ and add it to y to obtain $\bar{\lambda}$.
Depending on the value of s'_{suf}, we derive the upper bound of the former interval in three different ways: if $s'_{\mathsf{suf}}=1^b$, then $y=bd$; if $s'_{\mathsf{suf}}=\textsc{Concat}(1^b, 0)$, then the upper bound of the interval is $\min(d-1, n-t-(\sum_{i=0}^{\sigma-1}\lambda_i)-bd)$ since s'_{suf} contains the unary encoding of the entire $\bar{q}$ and thus the randomly drawn value must be a valid remainder modulo d. Finally, if $s'_{\mathsf{suf}}=\textsc{Concat}(1^b0z)$, where $z\in\{0,1\}^c, c\in\{0,\ldots,\lfloor\log_2(d)\rfloor\}$, then the upper bound is determined considering that the c most significant bits of the remainder are contained in s'_{suf}, therefore the bound is $\min(2^c-1, n-t-(\sum_{i=0}^{\sigma-1}\lambda_i)-bd-\textsc{BinToInt}(z)2^c)$.

The suffix x required for $\psi(s')=\textsc{Concat}(s', x)$ is obtained as $\texttt{Gol}(\bar{\lambda})$ concatenated with $\texttt{Gol}(\lambda_j)$, $\sigma+1\leq j\leq t-1$, in turn making ψ pick randomly the suffix x among the ones for which $\textsc{Concat}(s', x)\in S^{\measuredangle}$ holds. As a consequence of the operative description to compute the map ψ, it is now possible to prove also prop. *iv)* in Def. 2.2. Let $\mathcal{Z}$ be a random variable over the set S', and $\mathcal{W}=\chi(\mathcal{Z})$ a random variable over the combinations C (or equivalently, over the strings in $S^{\measuredangle}$, given the bijectivity of $\varphi(\cdot)$). The design of the ψ procedure allows to observe that the conditional entropy $\mathrm{H}(\mathcal{W}|\mathcal{Z}=s')$ is always greater than or equal to zero (it is zero, only if a single admissible suffix exists). From this, recalling that the entropies of two random variables are bound by the following relation: $\mathrm{H}(\mathcal{W})=\mathrm{H}(\mathcal{Z})+\mathrm{H}(\mathcal{W}|\mathcal{Z}=s')$, it is easy to conclude that the proposed StC map $\chi(\cdot)$ is entropy preserving as $\mathrm{H}(\mathcal{W})\geq\mathrm{H}(\mathcal{Z})$. From a practical standpoint, we will not materialize $\texttt{Gol}(\bar{\lambda})$, nor of $\texttt{Gol}(\lambda_j)$, $\sigma+1\leq j\leq t-1\}$; instead we will simply draw them according to the rules described above, and directly employ them to derive the output combination $c\in C$.

3.2 Constant Time Algorithm for $\chi(\cdot)$, $\chi^{-1}(\cdot)$

We report in Alg. 4 a constant time algorithm computing $\lambda_i, 0\leq i\leq t-1$ from a binary string $s'\in S'$. We consider, for the sake of simplicity and efficiency, the choice of d as a power of two, and thus encode the remainders of the divisions $r_i=\lambda_i \bmod d$ in natural binary. In particular, the algorithm details the entire process which outputs the sequence of λ_i, since this sequence can easily be transformed in the required combination $c=[\gamma_0,\ldots,\gamma_{t-1}]$ through a simple fixed-iteration loop, represented by the opaque function at line 38. The algorithm is logically split in three phases.

The first phase of the algorithm (lines 4–18) computes all the value λ_i preceding $\bar{\lambda}$ in constant time updating the current value (λ in the algorithm) overwriting the location in the destination vector `lambdaVec` at position `idx` (line 11) with a constant-time store primitive at each iteration. The computation of the quotient value q is performed employing a boolean mask value, `qdone`, which is updated inclusive-or-ing the complement of the read bit value. This in turn causes the value of `qdone` to transition to one as soon as the `0` bit which delimits the unary representation of q is read. We thus rely (at line 6) on the value of `qdone` to perform a predicated addition of the read bit onto q. The same predicated addition approach is employed to count how many of the remainder bits have been read, (line 7), and determine if we are done reading an entire remainder value (line 8). Whenever the correct value for λ_i has been read from the input bitstream, computed and stored, the two boolean variables `qdone` and `rdone` will both be set, allowing the algorithm to update the position of the destination vector in which the data store is performed (line 13), exploiting again a predicated addition. In a similar fashion, the values of q and r are reset, and the counter of the decoded positions is updated (lines 14–18).

The second phase of the algoritm is dedicated to the computation of $\bar{\lambda}$ (lines 19–31). This portion of the algorithm, due to the need of being performed in constant time, employs the current, partial values stored in q and, possibly, r compute the appropriate value y, randomly drawing it in the correct interval, depending on whether part of the quotient (lines 19–21), the whole quotient (lines 22–25), or the whole quotient plus part of the remainder (lines 26–28) was read out from s'. All the three possible values are randomly drawn in their appropriate intervals (lines 20, 24 and 27 for the

Algorithm 4: Constant time computation of $\chi : S' \rightarrow C$

```
Input: $s'$: an $l$-bit binary string
       $d$: positive integer employed to derive the encodings of the zero-run
            lengths present in $s'$
Output: $c$=[$\gamma_0, \ldots, \gamma_i, \ldots, \gamma_{t-1}$]: combination in $C$, $\gamma_i \in \{0, \ldots, n-1\}$
Data: Extend($v$) : replicates the l.s.b. in all other bits of register $v$
      CTCond(cond, $t$, $f$) computes (Extend($c$)$\wedge t$)$\vee$(Extend($\neg c$)$\wedge f$),
      returns the register values $t$ or $f$ if cond is true or false, respectively.
      CTStore($array$, $pos$, $v$): constant-time store of the register $v$ in the
      $array$ cell with index $0 \leq pos \leq t-1$
      CTLoad($array$, $pos$): returns the value in the $array$ cell with index
      $0 \leq pos \leq t-1$ in constant-time
      CTRand($v$): returns a randomly and uniformily picked integer in
      $\{0, \ldots, v\}$ in constant-time
1  qdone ← 0, rdone ← 0, lambdadone ← 0                     // Bit variables
2  $q \leftarrow 0, r \leftarrow 0, \lambda \leftarrow 0$                          // register variables
3  idx←0, rbitctr←0, remainingPos←$n-t$
                                          // Golomb decoding of $\lambda_0, \ldots, \lambda_{\sigma-1}$
4  foreach bit in $s'$ do
5      qdone ← qdone $\vee$ $\neg$bit
6      $q \leftarrow q$ + (bit $\wedge$ qdone)
7      rbitctr ← rbitctr + qdone                 // count read bits in $r$
8      rdone ← (rbitctr = $d + 1$)               // stopbit in remainder
9      $r \leftarrow 2r$ + (bit $\wedge$ qdone)
10     $\lambda \leftarrow qd + r$
11     CTStore(lambdaVec, idx, $\lambda$)
12     lambdadone ← qdone $\wedge$ rdone
13     idx ← idx + CTCond(lambdadone, 1, 0)
14     remainingPos←remainingPos−CTCond(lambdadone, $\lambda$, 0)
15     $q \leftarrow$ CTCond(lambdadone, 0, $q$)
16     qdone ← CTCond(lambdadone, 0, qdone)
17     $r \leftarrow$ CTCond(lambdadone, 0, $r$)
18     rbitctr ← CTCond(lambdadone, 0, rbitctr)
                                                   // Recomputation of $\bar{\lambda}$
19 casePQ ← ¬qdone                                         // incomplete $q$
20 r-pq ← CTRand(remainingPos − $\lambda$)
21 $\lambda \leftarrow \lambda$ + CTCond(casePQ, r-pq, 0)
22 caseCQ ← qdone $\wedge$ (rbitctr = 1)               // comp. $q$, missing $r$
23 r-cq ← ((r-pq > ($d-1$)) $\wedge$ ($d-1$)) $\vee$ ($\neg$(r-pq > ($d-1$)) $\wedge$ (r-pq)
24 r-cq ← CTRand(r-cq)
25 $\lambda \leftarrow \lambda$ + CTCond(caseCQ, r-cq, 0)
26 casePR ← qdone $\wedge$ ¬rdone $\wedge$ (rbitctr > 1)             // incomp. $r$
27 r-pr ← $qd + r2^{\log_2(d)-\text{rbitctr}}$ + CTRand($2^{\log_2(d)-\text{rbitctr}} - 1$)
28 $\lambda \leftarrow$ CTCond(casePR, r-pr, 0)
29 CTStore(lambdaVec, idx, $\lambda$)
30 idx ← idx + CTCond(lambdadone, 0, 1)
31 remainingPos ← remainingPos − CTCond(lambdadone, $\lambda$, 0)
                                         // Random drawing of $\lambda_\sigma \ldots \lambda_{t-1}$
32 for $i \leftarrow 0$ to $t-1$ do
33     $r \leftarrow$ CTRand(remainingPos)
34     $\lambda \leftarrow$ CTLoad(lambdaVec, $i$)
35     $\lambda \leftarrow$ CTCond(($i$ > idx), $r$, $\lambda$)
36     CTStore(lambdaVec, $i$, $\lambda$)
37     remainingPos←remainingPos−CTCond(($i$ > idx), $\lambda$, 0)
38 return CTRunLengthsToCombination(lambdaVec)
```

three aforementioned cases), and conditionally added via boolean predication to the current value of λ (lines 21, 25 and 28, respectively). Having reached line 31 the algorithm has computed all the run-lengths up to $\bar{\lambda}$ included, and the value of `remainingPos` is updated accordingly.

In the third phase, (lines 32–37), the algorithm needs to draw completely at random the remaining (if any) values of run-lengths. In order to avoid a computation time depending on the number of remaining run-lengths to be drawn at random, a random value, in the range of the remaining sum of run-lengths is always drawn (line 33), and conditionally replaces the value in an entry of the vector `lambdaVec` holding the run-lengths themselves (lines 33-35). Finally, the sum of the remaining run-lengths, `remainingPos`, is updated with a constant time predicated subtraction.

The reported algorithm runs in $O(l + t)$ constant time as it is constituted of two countable loops with fixed trip count, and simple arithmetic-logic operations, achieving (prop. *ii)* in Def. 2.2).

The computation of χ^{-1} in constant time is significantly more straightforward. Indeed, the first step is to Golomb-encode all the zero runs with a fixed trip-count loop, performing exactly t iterations, and encoding a value of λ_i at each iteration. The Golomb encoding of a single λ_i can be performed in constant time knowing that the value of q and r are naturally upper bounded by $n - t$. This allows to perform a constant time writeout of q, simply performing a predicated substitution of its value in memory, with one having an extra set bit or not. The value of r, instead, encoded in natural binary can be computed in constant time by means of a boolean mask, recalling that d is a power of two. To perform the aforementioned computation, Golomb-encoding the zero-run-lengths present in $\mathtt{str}(c)$, a memory amount equal to the to the maximum admissible length for an element $s \in S^{\perp}$ should be reserved. Such a length is $\lfloor \frac{n-t}{d} \rfloor + t(1 + \log_2(d))$ and is matched by the string having λ_0 with $q_0 = \lfloor \frac{n-t}{d} \rfloor$, $r_0 = (n - t) \bmod d$ and all $\lambda_i = 0$, $1 \leq i \leq t-1$.

4 EXPERIMENTAL EVALUATION

We report the results of the experimental evaluation of our constant-time `C99` implementation of $\chi(\cdot)$. We compare it with the one relying the combinadics number system, as it is the only other one guaranteeing that a given bitstring of length l will be encoded in a constant weight bitstring. We realized both a straightforward, variable time version of our proposed χ, where branches in the algorithm are actual branching constructs, and the constant time version following Alg. 4 in Section 3. We also implemented the method by [17], which cannot run in constant time due to the encoding failures, and report that its performances match the ones of the variable-time implementation of our χ, when it is able to constant-weight encode the input. All timing results were collected on an Intel Core i5-6500 CPU clocked at 3.2 GHz, with 32 GiB DDR-4 at 2133 MHz. Frequency scaling was disabled, the frequency locked to 3.2 GHz, and the Turbo Boost feature was disabled. All benchmarks were compiled and run on Debian GNU/Linux 10.2 with GCC 8.3.0, compilation options `-march=native -O3`. We employed Intel's `rdrand` instruction as a random number source, and Intel's `movnti` instruction to obtain a constant time store, as it forces a writeback to memory regardless of the state of the caches. The clock cycle count was obtained with of Intel's `rtdscp` instruction, measuring the number of cycles taken by $100k$ runs of the CWE and CWD over random l-bit strings for our approach, and by 100 runs of the combinadics based CWE/CWD on random integers in $\{0, \ldots, \binom{n}{t}\}$.

To validate experimentally the fact that our implementation is actually running in constant time, we employ a well established statistical testing methodology proposed in [7] and applied to the timing side channel leakage in [16] relying on Student's t test. This testing methodology collects two set of measurements from the running time of the implementation acting on input sets having

Table 1: Number of required clock cycles$\times 10^3$ to compute CWE and CWD, with $d = 2^8$. Parameters for Category 1: $(n; t; l) = (71,798; 136; 256)$**; Category 3:** $(n; t; l) = (115,798; 199; 384)$**; Category 5:** $(n; t; l) = (178,102; 267; 512)$

StC map	Category 1				Category 3				Category 5			
	CWE (kcycles)		CWD (kcycles)		CWE (kcycles)		CWD (kcycles)		CWE (kcycles)		CWD (kcycles)	
	avg.	std.dev.	avg.	std.dev.	avg.	std.dev.	avg.	std.dev.	avg.	std.dev.	avg.	std.dev.
Combinadics	51×10^4	1.3×10^4	18×10^4	0.6×10^4	144×10^4	2×10^4	49×10^4	1.7×10^4	366×10^4	5×10^4	119×10^4	4.8×10^4
Ours (CTime)	114.5	1.3	28.2	0.1	166.5	1.6	43.5	0.2	224.0	1.8	61.8	0.2
Ours (VTime)	50.3	0.6	3.0	0.2	72.6	0.7	4.6	0.2	98.2	1.0	5.9	0.2

different distribution. If the implementation at hand is actually constant time, no choice of the input set distribution should influence its running time. To detect whether or not the inputs influence the running time, a t test, having as null hypothesis the fact that the average running time of the implementation on the first input set is different from the average running time on the second is performed. The choice of the null hypothesis minimizes the probability of stating that the average running time of the implementation on the two input sets is the same, when it actually is not.

We collected 10^5 measurements of the number of clock cycles taken to compute χ (χ^{-1}), taking as a first set of inputs, a set of randomly generated bitstring of the appropriate length to guarantee the encoding. We took as the second set of inputs a set of all-zero bit strings: such a choice is meant to elicit any timing difference due to the fact that the all-zero bit string is both one among the shortest which can be encoded (eliciting potential timing differences due to the input length), and the one yielding the minimum value of the run lengths (eliciting potential differences due to the arithmetic computations). The obtained t-statistic is < 4.5 corresponding to a confidence in rejecting the null hypothesis of the running times differing between the two input sets of $> 99.999\%$. By contrast the same test run on the variable time implementation of χ and χ^{-1} yields a t-statistic in the hundreds range, corresponding to a negligible confidence ($\approx 10^{-7}\%$) in the means being the same.

Table 1 reports the results collected choosing values for n, t and l from the specification of LEDACrypt [2–5], a current round 2 candidate in the NIST post-quantum standardization process, employing the IND-CCA2 conversion of [11] which needs a constant weight encoding primitive. The parameters match security levels equivalent to AES-128 (Category 1), AES-192 (Category 3), and AES-256 (Category 5). The results in Table 1 highlight how our approach is between 3 and 4 orders of magnitude faster than the one based on the combinatorial number system. While our constant time implementation of the χ and χ^{-1} maps are between 1.5$\times$ and 2.2$\times$ slower than their variable time counterparts, that the absolute values of the clock cycles correspond to timings in the tenths of μs range. To further put things in perspective, we compare our implementation of the χ function with the constant time techniques to directly generate a random, constant weight binary string reported in [8]. The results in [8] report the number of clock cycles taken to to generate a 20, 326-bit string with weight 134 on an Intel i7-7700 clocked at 3.6 GHz with three different approaches. The figures reported show that 45–390 kcycles are needed, depending on which of the algorithms proposed by the authors is chosen, while our constant time implementation of χ takes 128.3 kcycles to generate a constant weight string of the same length and weight (we obtain $l = 256, d = 2^6$), further showing the practicality of the approach.

5 CONCLUSION

We introduced a novel technique to reliably perform the constant weight encoding of a random binary string. The proposed technique is an enabler for the use of the currently most bandwidth efficient IND-CCA2 construction that can be applied to post-quantum code-based cryptosystems. We detailed how our technique can be efficiently implemented in constant time, allowing a timing side-channel secure realization of the proposed primitive.

REFERENCES

[1] European Telecommunications Standards Institute (ETSI). 2017. Quantum-Safe Cryptography (QSC). (March 2017). https://www.etsi.org/technologies/quantum-safe-cryptography?jjj=1581466327931

[2] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2018. 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 (Lecture Notes in Computer Science)*, Vol. 10786. Springer, 3–24.

[3] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2019. LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate. In *Code-Based Cryptography - 7th International Workshop, CBC 2019, Darmstadt, Germany, May 18-19, 2019, Revised Selected Papers (Lecture Notes in Computer Science)*, Vol. 11666. Springer, 11–43.

[4] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. 2019. LEDACrypt Specification v2.0. https://www.ledacrypt.org/documents/LEDAcrypt_spec_latest.pdf, last accessed Nov. 2019. (2019).

[5] Alessandro Barenghi, William Fornaciari, Andrea Galimberti, Gerardo Pelosi, and Davide Zoni. 2019. Evaluating the Trade-offs in the Hardware Design of the LEDAcrypt Encryption Functions. In *26th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2019, Genoa, Italy, Nov. 27-29, 2019.* IEEE.

[6] Alessandro Barenghi and Gerardo Pelosi. 2020. Constant time, constant weight encoding library and results reproduction framework. https://doi.org/10.5281/zenodo.3747545. (2020).

[7] Jean-Sébastien Coron, David Naccache, and Paul C. Kocher. 2004. Statistics and secret leakage. *ACM Trans. Embedded Comput. Syst.* 3, 3 (2004), 492–508.

[8] Nir Drucker and Shay Gueron. 2019. Generating a Random String with a Fixed Weight. In *CSCML 2019 (LNCS)*, Vol. 11527. Springer.

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

[10] Donald Erwin Knuth. 1887. Generating All Combinations and Partitions. *The Art of Computer Programming* 4, 3 (1887), 45–49.

[11] Kazukuni Kobara and Hideki Imai. 2001. Semantically Secure McEliece Public-Key Cryptosystems-Conversions for McEliece PKC. In *4th Int.'l Workshop on Practice and Theory in Public Key Cryptography, PKC 2001 (LNCS)*, Vol. 1992. Springer.

[12] R. J. McEliece. 1978. A public-key cryptosystem based on algebraic coding theory. *The Deep Space Network Progress Report, DSN PR 42-44* (1978), 114–116.

[13] National Institute of Standards and Technology. 2017. Post-Quantum Crypto Project. (Nov. 2017). http://csrc.nist.gov/groups/ST/post-quantum-crypto/

[14] H. Niederreiter. 1986. Knapsack-type cryptosystems and algebraic coding theory. *Probl. Contr. and Inform. Theory* 15 (1986), 159–166.

[15] Ernesto Pascal. 1887. Sopra una formola numerica. *Giornale di matematiche* 25, 1 (1887), 45–49.

[16] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. 2017. Dude, is my code constant time?. In *DATE 2017.* IEEE, 1697–1702.

[17] Nicolas Sendrier. 2005. 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*. IEEE, 435–438.

A SOFTWARE ARTIFACT EVALUATION AND RESULT REPLICATION

This is a short appendix on the evaluation and use of the software artifact related to the paper "Constant Weight Strings in Constant Time: a Building Block for Code-based Post-quantum Cryptosystems" accepted for publication at the 17th ACM International Conference on Computing Frontiers. The provided software artifact is a C library implementing both the proposed, constant time software library to perform constant weight encoding, and the implementation of the state of the art methods to allow both the reuse of the software artifact, and the reproduction of the results contained in the paper.

A.1 Source Code Organization

The software artifact is a standalone C library, implementing the proposed constant time, constant weight encoding and decoding techniques, plus the non constant time constant weight encoding technique, and the combinadic based encoding technique from the state of the art. The library is organized in two C compilation units, `combinadics_library.c` and `constant_weight_codec.c`. The former contains the entire combinadics constant weight encoding technique, together with a self-contained, portable C99 arbitrary precision arithmetic library. The latter contains both the constant time implementation of our proposed constant weight encoding technique, and the variable time implementation of the current state of the art to be used for result reproducibility.

In addition to the library, the software artifact package also contains a boilerplate main program performing functional tests on the encoding and decoding techniques of all three methods, and timing benchmarks on them.

The user, willing to reuse our provided software artifact, will include `constant_weight_codec.h` in her own codebase, and employ the following interface:

- The `constant_time_bin_to_cw` takes three parameters: i) an unsigned character vector, containing the dense binary string to be encoded, with bit ordering going from left to right, i.e. from the most significant bit of the first element of the vector, to the least significant bit of the last; ii) an integer specifying the maximum guaranteed encoding length, denoted as l in the paper, and iii) a vector of t unsigned integers, which will be filled with the positions of the set bits in the constant weight vector.
- The `constant_time_cw_to_bin` function takes the same three parameters as the `constant_time_bin_to_cw`, acting on them in a dual fashion, i.e. it reads from the t element integer vector the positions of the asserted terms in the constant weight vector, and decodes them, writing the corresponing dense binary string in the character array provided.

The codebase is available under Public Domain license at [6].

A.2 System Requirements and Reproducibility of Results

The software library only relies on the the C standard library, and the availability of a C99 supporting compiler, together with the headers required to compile the `_mm_mfence` `_mm_stream_si32` and `_rdrand32_step` intrinsics (both GCC and Clang/LLVM distribution packages are equipped to do so). The building system relies on CMake, version 3.9 or greater. The software pacakage was tested on Debian GNU/Linux 10.3 and Gentoo Linux 17.1.

The hardware requirement is an `x86_64` CPU supporting the `movnti`, `mfence`, and `rdrand` instructions to be able to run the library, plus the and `rtdscp` instruction to perform the timing measurements. Any Intel CPU starting from the Ivy Bridge generation fulfills these requirements, and so does any AMD CPU starting from the Excavator generation, including all the Zen and Zen2 CPUs.

To obtain the binaries to reproduce all the results contained in the paper, it is sufficient to i) uncompress the code archive, ii) enter the `build` subdirectory in the `constant_weight_library` directory obtained decompressing the archive, iii) run `cmake ../ && make`. This will create four binaries performing the benchmarks recreating the results in Table 1 of the paper, and the t-statistics reported in the evaluation section. It is advisable, although not mandatory, to disable any frequency scaling/boosting mechanism on the test machine, if possible. The expected running times (dominated by the combinadics procedure) for the four executables are:

- `reproduce_paper_results_nvalue_20326` - 13 s.
- `reproduce_paper_results_nvalue_71798` - 47 s.
- `reproduce_paper_results_nvalue_115798` - 2 min 7s.
- `reproduce_paper_results_nvalue_178102` - 5 min 16 s.