On Optimal k-Deletion Correcting Codes

Jin Sima and Jehoshua Bruck, *Fellow, IEEE*

***Abstract*—Levenshtein introduced the problem of constructing k-deletion correcting codes in 1966, proved that the optimal redundancy of those codes is $O(k \log N)$ for constant k, and proposed an optimal redundancy single-deletion correcting code (using the so-called VT construction). However, the problem of constructing optimal redundancy k-deletion correcting codes remained open. Our key contribution is a major step towards a complete solution to this longstanding open problem for constant k. We present a k-deletion correcting code that has redundancy $8k \log N + o(\log N)$ when $k = o(\sqrt{\log \log N})$ and encoding/decoding algorithms of complexity $O(N^{2k+1})$.**

***Index Terms*—Deletion codes, Varshamov-Tenengoltz code.**

I. Introduction

A set of binary vectors of length N is a k-deletion correcting code (denoted by $\mathcal{C}$) iff any two vectors in $\mathcal{C}$ do not share a subsequence of length $N-k$. The problem of constructing a k-deletion correcting code was introduced by Levenshtein [1]. He proved that the optimal redundancy (defined as $N - \log |\mathcal{C}|$) is $O(k \log N)$ for constant k. Specifically, it is in the range $k \log N + o(\log N)$ to $2k \log N + o(\log N)$ when k is a constant. In general, the redundancy $O(k \log N)$ is orderwise optimal when $k \leq O(N^{\epsilon})$ for some $\epsilon < 1$. The optimal redundancy becomes $O(k \log(N/k))$ when $k = O(N)$ and k is small, e.g., $k = n/4$ [9]. When $k \geq n/2$, a k-deletion correcting code has cardinality at most two. In addition, Levenshtein proposed the following optimal construction (the well-known Varshamov-Tenengolts (VT) code [2]):

$$\left\{ (c_1, \ldots, c_N) : \sum_{i=1}^{N} i c_i \equiv 0 \bmod (N+1) \right\}, \qquad (1)$$

that is capable of correcting a single deletion with redundancy not more than $\log(N+1)$ [1]. The encoding/decoding complexity of VT codes is linear in N. Generalizing the VT construction to correct more than a single deletion was elusive for more than 50 years. In particular, the past approaches [4], [5], [6] result in asymptotic code rates that are bounded away from 1.

A recent breakthrough paper [7] proposed a k-deletion correcting code construction with $O(k^2 \log k \log N)$ redundancy and $O_k(N \log^4 N)$[1] encoding/decoding complexity, where k is a constant. For the case $k = 2$ deletions, the redundancy was improved in [12], [13]. Specifically, the code in [13] has redundancy of $7 \log N$ and linear encoding/decoding complexity. The work in [14] considered correction with high probability and proposed a k-deletion correcting code construction with redundancy $(k+1)(2k+1) \log N + o(\log N)$ and encoding/decoding complexity $O(N^{k+1}/\log^{k-1} N)$. The result for this randomized coding setting was improved in [8], where redundancy $O(k \log(N/k))$ and complexity $poly(N, k)$ were achieved. However, finding a deterministic k-deletion correcting code construction that achieves the optimal order redundancy $O(k \log N)$ remained elusive.

Our key contribution is a solution to this longstanding open problem for constant k: We present a code construction that achieves $O(k \log N)$ redundancy when $k = o(\sqrt{\log \log N})$ and $O(N^{2k+1})$ encoding/decoding computational complexity. Note that the complexity is polynomial in N when k is a constant. The following theorem summarizes our main result. We note that in this paper, the optimality of the code is redundancy-wise rather than cardinality-wise, namely, the focus is on the asymptotic rather than exact size of the code. The problem of finding optimal cardinality k-deletion correcting code appears highly nontrivial even for $k = 1$.

Theorem 1. *Let k and n be two integers satisfying $k = o(\sqrt{\log \log n})$. For integer $N = n + 8k \log n + o(\log n)$, there exists an encoding function $\mathcal{E} : \{0,1\}^n \to \{0,1\}^N$, computed in $O(n^{2k+1})$ time, and a decoding function $\mathcal{D} : \{0,1\}^{N-k} \to \{0,1\}^n$, computed in $O(n^{k+1}) = O(N^{k+1})$ time, such that for any $\mathbf{c} \in \{0,1\}^n$ and subsequence $\mathbf{d} \in \{0,1\}^{N-k}$ of $\mathcal{E}(\mathbf{c})$, we have that $\mathcal{D}(\mathbf{d}) = \mathbf{c}$.*

Recently, an independent work [9] proposed a k-deletion correcting code with $O(k \log N)$ redundancy and better complexity of $poly(N, k)$. In contrast to a redundancy of $8k \log N$ in this paper, the redundancy in [9] only specified asymptotically, and it is estimated to be at least $200k \log n$. Moreover, the techniques in [9] and this paper are different.

Here are the key building blocks in our code construction: (i) *generalizing the VT construction* to k deletions by considering constrained sequences, (ii) separating the encoded vector to blocks and using *concatenated codes* and (iii) a novel strategy to *separate the vector to blocks by a single pattern.* The following gives a more specific description of these ideas.

In our previous work for 2-deletions codes [13], we *generalized the VT construction*. In particular, we proved that while the higher order parity checks $\sum_{i=1}^{N} i^j c_i \bmod (N^j + 1)$, $j = 0, 1, \ldots, t$ do not work in general, those parity checks work in the two deletions case when the sequences are constrained to have no consecutive 1's. In this paper we generalize this idea, specifically, the higher order parity checks can be used to correct $k = t/2$ deletions in sequences that satisfy the

Manuscript received October 15, 2019; revised June 4, 2020; accepted September 8, 2020. This work was supported in part by NSF grants CCF-1717884 and CCF-1816965. This paper was presented in part at the 2019 IEEE International Syposium on Information Theory.

J. Sima and J. Bruck are with the Electrical Engineering Department, California Institute of Technology, Pasadena, CA, 91125 USA (e-mail: jsima@caltech.edu ;bruck@caltech.edu).

[1]The notion O_k denotes *parameterized complexity*, i.e., $O_k(N \log^4 N) = f(k) O(N \log^4 N)$ for some function f.

following constraint: The index distance between any two 1's is at least k, i.e., there is a 0 run of length at least $k-1$ between any two 1's.

The fact that we can correct k deletions using the generalization of the VT construction on constrained sequences, enables a *concatenated code construction*, which is the underlying structure of our k-deletion correcting codes. In concatenated codes, each codeword $\mathbf{c}$ is split into blocks of small length, by certain markers at the boundaries between adjacent blocks. Each block is protected by an inner deletion code that can be efficiently computed when the block length is small. The block boundary markers are chosen such that k deletions in a codeword result in at most $O(k)$ block errors. Then, it suffices to use an outer code, such as a Reed-Solomon code, to correct the block errors.

Concatenated codes were used in a number of k-deletion correcting code constructions, [7], [11], [10]. One of the main differences among these constructions is the choice of the block boundary markers that separate the codewords. The constructions in [10], [11] insert extra bits as markers between blocks, which introduces $O(N)$ bits of redundancy. In [7], occurrences of short subsequences in the codeword, called patterns, were used as markers. The construction in [7] requires $O(k \log k)$ different patterns where each pattern introduces $O(k \log N)$ bits of redundancy.

We improve the redundancy in [7] by using *a single type of pattern* for block boundary markers. Specifically, we choose a pattern such that its indicator vector, a binary vector in which the 1 entries indicate the occurrences of the pattern in $\mathbf{c}$, is a *constrained sequence* that is immune to deletions. Then, the generalized VT construction can be used to protect the indicator vector and thus the locations of the block boundary markers. Knowing the boundary between blocks, we can recover the blocks with at most $2k$ block errors, which then can be corrected using a combination of short deletion correcting codes and Reed-Solomon codes.

The concatenated code construction consists of short blocks, hence, the pattern has to occur frequently in the codeword such that every consecutive bits of certain length contains at least one occurrence of the pattern. Namely, the first step of the encoding is a mapping of an arbitrary binary sequence to a sequence with frequent synchronization patterns. The constructions in [7], [9] compute such mappings using randomized algorithms. In this paper, we provide a deterministic algorithm to generate sequences with frequent synchronization patterns.

We now formally define the synchronization pattern and its indicator vector.

Definition 1. *A* synchronization pattern*, is a length* $3k + \lceil \log k \rceil + 4$ *sequence* $\mathbf{a} = (a_1, \ldots, a_{3k+\lceil \log k \rceil + 4})$ *satisfying*

- $a_{3k+i} = 1$ *for* $i \in [0, \lceil \log k \rceil + 4]$*, where* $[0, \lceil \log k \rceil + 4] = \{0, \ldots, \lceil \log k \rceil + 4\}$.
- *There does not exist a* $j \in [1, 3k-1]$*, such that* $a_{j+i} = 1$ *for* $i \in [0, \lceil \log k \rceil + 4]$.

Namely, a synchronization pattern is a length $3k + \lceil \log k \rceil + 4$ *sequence that ends with* $\lceil \log k \rceil + 5$ *consecutive* 1*'s and no other 1-run with length* $\lceil \log k \rceil + 5$ *exists.*

For a sequence $\mathbf{c} = (c_1, \ldots, c_n)$, the indicator vector of the synchronization pattern, referred to as *synchronization vector* $\mathbb{1}_{sync}(\mathbf{c}) \in \{0,1\}^n$, is defined by

$$\mathbb{1}_{sync}(\mathbf{c})_i = \begin{cases} 1, & \text{if } (c_{i-3k+1}, c_{i-3k+2}, \ldots, c_{i+\lceil \log k \rceil + 4}) \\ & \text{is a synchronization pattern,} \\ 0, & \text{else.} \end{cases} \tag{2}$$

Note that $\mathbb{1}_{sync}(\mathbf{c})_i = 0$ for $i \in [1, 3k-1]$ and for $i \in [n - \lceil \log k \rceil - 3, n]$. The entry $\mathbb{1}_{sync}(\mathbf{c}) = 1$ if and only if c_i is the first bit of the final 1 run in a synchronization pattern. It can be seen from the definition that any two 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ have index distance at least $3k$, i.e., any two 1 entries are separated by a 0-run of length at least $3k-1$. Hence, $\mathbb{1}_{sync}(\mathbf{c})$ is a constrained sequence described above.

Example 1. *Let integers* $k = 2$ *and* $n = 35$*. Then the sequence* $(1,0,0,1,0,1,1,1,1,1,1)$ *is a synchronization pattern. Let the length* n *sequence*

$$\begin{aligned} \mathbf{c} = (&1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,\\ &1,0,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1). \end{aligned}$$

Then the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ *is given by*

$$\begin{aligned} \mathbb{1}_{sync}(\mathbf{c}) = (&0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,\\ &0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0). \end{aligned}$$

Next, we present the definition of the generalized VT code. Define the integer vectors

$$\mathbf{m}^{(e)} \triangleq (1^e, 1^e + 2^e, \ldots, \sum_{j=1}^{n} j^e) \tag{3}$$

for $e \in [0, 6k]$, where the i-th entry $\mathbf{m}_i^{(e)}$ of $\mathbf{m}^{(e)}$ is the sum of the e-th powers of the first i positive integers. Given a sequence $\mathbf{c} \in \{0,1\}^n$, we compute the generalized VT redundancy $f(\mathbf{c})$ of dimension $6k+1$ as follows:

$$\begin{aligned} f(\mathbf{c})_e &\triangleq \mathbf{c} \cdot \mathbf{m}^{(e)} \bmod 3kn^{e+1} \\ &= \sum_{i=1}^{n} c_i \mathbf{m}_i^{(e)} \bmod 3kn^{e+1}, \end{aligned} \tag{4}$$

for $e \in [0, 6k]$, where $f(\mathbf{c})_e$ is the e-th component of $f(\mathbf{c})$ and $\cdot$ denotes inner product over integers. Our generalization of the VT code construction shows that the vector $f(\mathbb{1}_{sync}(\mathbf{c}))$ helps protect the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ from k deletions in $\mathbf{c}$.

The rest of the paper is organized as follows. Section II provides an outline of our construction and some of the basic lemmas. Based on results of the basic lemmas, Section III presents the encoding and decoding procedures of our code. Section IV presents our VT generalization for recovering the synchronization vector. Section V explains how to correct k deletions based on the synchronization vector, when the synchronization patterns appear frequently. Section VI describes an algorithm to transform a sequence into one with frequently occurred synchronization patterns. Section VII concludes the paper.

II. Outline and Preliminaries

In this section, we outline the ingredients that constitute our code construction and present notations that will be used throughout the paper. A summary of these notations is provided in Table I. We begin with an overview of the code construction, which has a concatenated code structure as described in Section I. In our construction, each codeword $\mathbf{c}$ is splitted into blocks, with the boundaries between adjacent blocks given by synchronization patterns. Specifically, let $t_1, \ldots, t_J$ be the indices of the synchronization patterns in $\mathbf{c}$, i.e., the indices of the 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$. Then the blocks are given by $(c_{t_{j-1}+1}, \ldots, c_{t_j-1})$ for $j \in [0, J+1]$, where $t_j = 0$ if $j = 0$ and $t_j = n+1$ if $j = J+1$. The key idea of our construction is to use the VT generalization f (see Eq. (4) for definition) as parity checks to protect the synchronization vector $\mathbb{1}_{\mathbf{c}}$, and thus identify the indices of block boundaries. The proof details will be given in Lemma 1. Note that the parity $f(\mathbf{c})$ in (4) has size $O(k^2 \log n)$. To compress the size of $f(\mathbf{c})$ to the targeted $O(k \log n)$, in Lemma 2 we apply a modulo operation on the function f. Given the block boundaries, Lemma 3 provides an algorithm to protect the codewords. Specifically, we show that given the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$, it is possible to recover most of the blocks in $\mathbf{c}$, with up to $2k$ block errors. These block errors can be corrected with the help of their deletion correcting hashes, which can be exhaustively computed and will be presented in Lemma 5. Hence, to correct the block errors, it suffices to protect the sequence of deletion correcting hashes using, for example, Reed-Solomon codes.

TABLE I
Summary of Notations

$\mathcal{B}_k(\mathbf{c})$	$\triangleq$	The set of sequences that share a length $n-k$ subsequence with $\mathbf{c} \in \{0,1\}^n$.
$\mathcal{R}_m$	$\triangleq$	The set of sequences where any two 1 entries are separated by at least $m-1$ zeros.
$\mathbb{1}_{sync}(\mathbf{c})$	$\triangleq$	Synchronization vector defined in (2).
$\mathbf{m}^{(e)}$	$\triangleq$	Weights of order e in the VT generalization, defined in (3).
$f(\mathbf{c})$	$\triangleq$	Generalized VT redundancy defined in (4).
L	$\triangleq$	Maximal length of zero runs in the synchronization vector of a *k-dense* sequence, defined in (5)
$p(\mathbf{c})$	$\triangleq$	The function used to compute the redundancy protecting the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$, defined in Lemma 2.
$Hash_k(\mathbf{c})$	$\triangleq$	The deletion correcting hash function for k-dense sequences, defined in Lemma 3.
$T(\mathbf{c})$	$\triangleq$	The function that generates k-dense sequences, defined in Lemma 4.
$H(\mathbf{c})$	$\triangleq$	Deletion correcting hash function for any sequence, defined in Lemma 5.

Lemma 2 and Lemma 3 together define a hash function that protects $\mathbf{c}$ from k deletions. However, in order to exhaustively compute the block hash given in Lemma 5 and obtain the desired size of the redundancy, the block length has to be of order $O(poly(k) \log n)$. Hence the synchronization pattern must appear frequently enough in $\mathbf{c}$. Such sequence $\mathbf{c}$ will be defined in the following as a k-dense sequence. To encode for any given sequence $\mathbf{c} \in \{0,1\}^n$, in Lemma 4 we present an invertible mapping that takes any sequence $\mathbf{c} \in \{0,1\}^n$ as input and outputs a k-dense sequence. Then, any binary sequence can be encoded into a k-dense sequence and protected.

Finally, the hash function defined by Lemma 2 and Lemma 3 is subject to deletion errors and has to be protected. To this end, we use an additional hash that encodes the deletion correcting hash of the hash function defined in Lemma 2 and Lemma 3. The additional hash is protected by a $(k+1)$-fold repetition code. Similar additional hash technique was also given in [7]. The final encoding/decoding algorithm of our code, which combines the results in Lemma 2, Lemma 3, and Lemma 4, is provided in Section III. The encoding is illustrated in Fig. 1.

Before presenting the lemmas, we give necessary definitions and notations. For a sequence $\mathbf{c} \in \{0,1\}^n$, define its deletion ball $\mathcal{B}_k(\mathbf{c})$ as the collection of sequences that share a length $n-k$ subsequence with $\mathbf{c}$.

Definition 2. *A sequence $\mathbf{c} \in \{0,1\}^n$ is said to be* k-dense *if the lengths of the 0 runs in $\mathbb{1}_{sync}(\mathbf{c})$ is at most*

$$L \triangleq (\lceil \log k \rceil + 5) 2^{\lceil \log k \rceil + 9} \lceil \log n \rceil + (3k + \lceil \log k \rceil + 4)(\lceil \log n \rceil + 9 + \lceil \log k \rceil). \quad (5)$$

For k-dense $\mathbf{c}$, the index distance between any two 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ is at most $L+1$, i.e., the 0-runs between two 1 entries have length at most $L+1$.

Note that $f(\mathbf{c})$ is an integer vector that can be presented by $\log(3k)^{6k+1} n^{(3k+1)(6k+1)}$ bits or by an integer in the range $[0, (3k)^{6k+1} n^{(3k+1)(6k+1)} - 1]$. In this paper, we interchangeably use $f(\mathbf{c})$ to denote its binary presentation or integer presentation.

The following lemma shows that the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ can be recovered from k deletions with the help of $f(\mathbb{1}_{sync}(\mathbf{c}))$. Its proof will be given in Section IV-A.

Lemma 1. *For integers n and k and sequences $\mathbf{c}, \mathbf{c}'$, if $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$ and $f(\mathbb{1}_{sync}(\mathbf{c})) = f(\mathbb{1}_{sync}(\mathbf{c}'))$, then $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$.*

By virtue of Lemma 1, the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ can be recovered from k deletions in $\mathbf{c}$, with the help of a hash $f(\mathbb{1}_{sync}(\mathbf{c}))$ of size $O(k^2 \log n)$ bits. To further reduce the size of the hash to $O(k \log n)$ bits, we apply modulo operations on $f(\mathbb{1}_{sync}(\mathbf{c}))$ in the following lemma, the proof of which will be proved in Section IV-B.

Lemma 2. *For integers n and $k = o(\sqrt{\log \log n})$, there exists a function $p : \{0,1\}^n \to [1, 2^{2k \log n + o(\log n)}]$, such that if $f(\mathbb{1}_{sync}(\mathbf{c})) \equiv f(\mathbb{1}_{sync}(\mathbf{c}')) \bmod p(\mathbf{c})$ for two sequences $\mathbf{c} \in \{0,1\}^n$ and $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$, then $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$. Hence if*

$$(f(\mathbb{1}_{sync}(\mathbf{c})) \bmod p(\mathbf{c}), p(\mathbf{c})) = (f(\mathbb{1}_{sync}(\mathbf{c}')) \bmod p(\mathbf{c}'), p(\mathbf{c}'))$$

and $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$, we have that $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$.

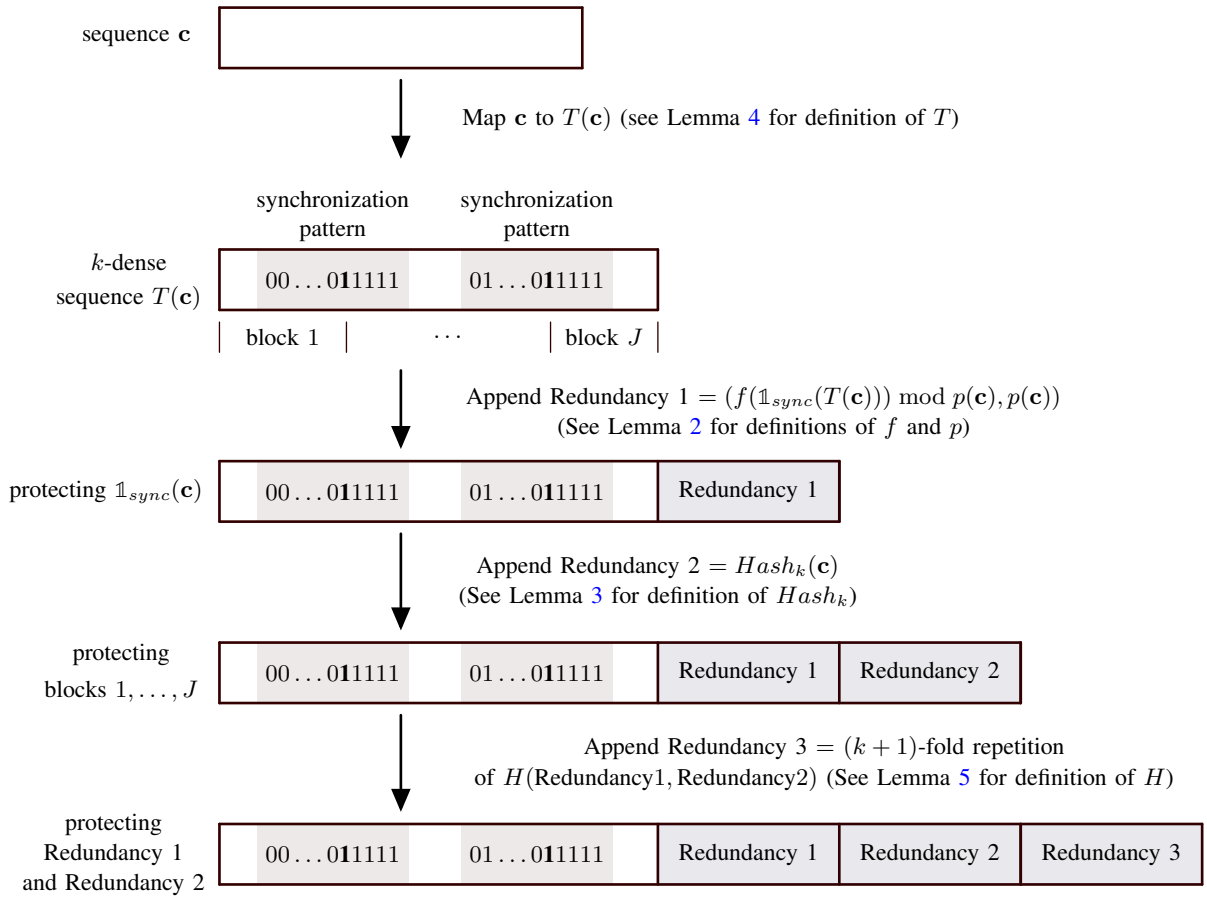

Fig. 1. Illustrating the encoding procedure of our k-deletion correcting code construction, based on results of Lemma 2, Lemma 3, Lemma 4, and Lemma 5. The indices of the bold 1 entries are the indices of the 1 entries in the synchronization vector $\mathbb{1}_{sync}(T(\mathbf{c}))$.

Lemma 2 presents a hash of size $4k \log n + o(\log n)$ bits for correcting $\mathbb{1}_{sync}(\mathbf{c})$. With the knowledge of the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$, the next lemma shows that the sequence $\mathbf{c}$ can be further recovered using another $4k \log + o(\log n)$ bit hash, when $\mathbf{c}$ is k-dense, i.e., when the synchronization patter occurs frequently enough in $\mathbf{c}$. The proof of Lemma 3 will be given in Section V.

Lemma 3. *For integers n and $k = o(\sqrt{\log \log n})$, there exists a function $Hash_k : \{0,1\}^n \to \{0,1\}^{4k \log n + o(\log n)}$, such that every k-dense sequence $\mathbf{c} \in \{0,1\}^n$ can be recovered, given its synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$, its length $n-k$ subsequence $\mathbf{d}$, and $Hash_k(\mathbf{c})$.*

Combining Lemma 2 and Lemma 3, we obtain size $O(k \log n)$ hash function to correct deletions for a k-dense sequence. To encode for arbitrary sequence $\mathbf{c} \in \{0,1\}^n$, a mapping that transforms any sequence to a k-dense sequence is given in the following lemma. The details will be given in Section VI-B.

Lemma 4. *For integers k and $n > k$, there exists a map $T : \{0,1\}^n \to \{0,1\}^{n+3k+3\lceil \log k \rceil + 15}$, computable in $poly(n,k)$ time, such that $T(\mathbf{c})$ is a k-dense sequence for $\mathbf{c} \in \{0,1\}^n$. Moreover, the sequence $\mathbf{c}$ can be recovered from $T(\mathbf{c})$.*

The next three lemmas (Lemma 5, Lemma 6, and Lemma 7) present existence results that are necessary to prove Lemma 2 and Lemma 3. Lemma 5 gives a k-deletion correcting hash function for short sequences, which is the block hash described above in proving Lemma 3. It is an extension of the result in [7]. Lemma 6 is a slight variation of the result in [1]. It shows the equivalence between correcting deletions and correcting deletions and insertions. Lemma 6 will be used in the proof of Lemma 1, where we need an upper bound on the number of deletions/insertions in $\mathbb{1}_{sync}(\mathbf{c})$ caused by a deletion in $\mathbf{c}$. Lemma 7 (see [15]) gives an upper bound on the number of divisors of a positive integer n. With Lemma 7, we show that the VT generalization in Lemma 1 can be compressed by taking modulo operations. The details will be given in the proof of Lemma 2.

Lemma 5. *For any integers w, n, and k, there exists a hash function $H : \{0,1\}^w \to \{0,1\}^{\lceil (w/\lceil \log n \rceil) \rceil (2k \log \log n + O(1))}$, computable in $O_k((w/\log n) n \log^{2k} n)$ time, such that any sequence $\mathbf{c} \in \{0,1\}^w$ can be recovered from its length $w-k$ subsequence $\mathbf{d}$ and the hash $H(\mathbf{c})$.*

Proof. We first show by counting arguments the existence of a hash function $H' : \{0,1\}^{\lceil \log n \rceil} \to \{0,1\}^{2k \log \log n + O(1)}$, exhaustively computable in $O_k(n \log^{2k} n)$ time, such that

$H'(\mathbf{s}) \neq H'(\mathbf{s}')$ for all $\mathbf{s} \in \{0,1\}^{\lceil \log n \rceil}$ and $\mathbf{s}' \in \mathcal{B}_k(\mathbf{s}) \backslash \{\mathbf{s}\}$. The hash $H'(\mathbf{c}')$ protects the sequence $\mathbf{s} \in \{0,1\}^{\lceil \log n \rceil}$ from k deletions. Note that $|\mathcal{B}_k(\mathbf{c}')| \leq \binom{\lceil \log n \rceil}{k}^2 2^k \leq 2\lceil \log n \rceil^{2k}$. Hence it suffices to use brute force and greedily assign a hash value for each sequence $\mathbf{s} \in \{0,1\}^{\lceil \log n \rceil}$ such that $H'(s) \neq H'(\mathbf{s}')$ for all $\mathbf{s}' \in \mathcal{B}_k(\mathbf{s}) \backslash \{\mathbf{s}\}$. Since the size of $\mathcal{B}_k(\mathbf{s})$ is upper bounded by $2\lceil \log n \rceil^{2k}$, there always exists such a hash $H'(\mathbf{s}) \in \{0,1\}^{\log(2\lceil \log n \rceil^{2k}+1)}$, that has no conflict with the hash values of sequences in $\mathcal{B}_k(\mathbf{s})$. The total complexity is $O_k(n \log^{2k} n)$ and the size of the hash value $H'(\mathbf{s})$ is $2k \log\log n + O(1)$.

Now split $\mathbf{c}$ into $\lceil (w/\lceil \log n \rceil) \rceil$ blocks $c_{(i-1)\lceil \log n \rceil + 1}, \ldots, c_{i\lceil \log n \rceil}$, $i \in [1, \lceil (w/\lceil \log n \rceil) \rceil]$ of length $\lceil \log n \rceil$. If the length of the last block is less than $\lceil \log n \rceil$, add zeros to the end of the last block such that its length is $\lceil \log n \rceil$. Assign a hash value $\mathbf{h}_i = H'((c_{(i-1)\lceil \log n \rceil + 1}, \ldots, c_{i\lceil \log n \rceil}))$, $i \in [1, \lceil (w/\lceil \log n \rceil) \rceil]$ for each block. Let $H(\mathbf{c}) = (\mathbf{h}_1, \ldots, \mathbf{h}_{\lceil (w/\lceil \log n \rceil) \rceil})$ be the concatenation of $\mathbf{h}_i$ for $i \in [1, \lceil (w/\lceil \log n \rceil) \rceil]$.

We show that $H(\mathbf{c})$ protects $\mathbf{c}$ from k deletions. Let $\mathbf{d}$ be a length $n-k$ subsequence of $\mathbf{c}$. Note that $d_{(i-1)\lceil \log n \rceil + 1}$ and $d_{i\lceil \log n \rceil - k}$ come from bits $c_{(i-1)\lceil \log n \rceil + 1 + x}$ and $c_{i\lceil \log n \rceil - k + y}$ respectively after deletions in $\mathbf{c}$, where the integers $x, y \in [0, k]$. Therefore, $(d_{(i-1)\lceil \log n \rceil + 1}, \ldots, d_{i\lceil \log n \rceil - k})$ is a length $\lceil \log n \rceil - k$ subsequence of $(c_{(i-1)\lceil \log n \rceil + 1 + x}, \ldots, c_{i\lceil \log n \rceil - k + y})$, and thus a subsequence of the block $(c_{(i-1)\lceil \log n \rceil + 1}, \ldots, c_{i\lceil \log n \rceil})$. Hence the i-th block $(c_{(i-1)\lceil \log n \rceil + 1}, \ldots, c_{i\lceil \log n \rceil})$ can be recovered from $\mathbf{h}_i = H'(c_{(i-1)\lceil \log n \rceil + 1}, \ldots, c_{i\lceil \log n \rceil})$ and $(d_{(i-1)\lceil \log n \rceil + 1}, \ldots, d_{i\lceil \log n \rceil - k})$. Therefore, $\mathbf{c}$ can be recovered given $\mathbf{d}$ and $H(\mathbf{c})$. The length of $H(\mathbf{c})$ is $\lceil (w/\lceil \log n \rceil) \rceil (2k \log\log n + O(1))$ and the complexity of $H(\mathbf{c})$ is $O_k((w/\log n) n \log^{2k} n)$. □

Lemma 6. *Let r, s, and k be integers satisfying $r + s \leq k$. For sequences $\mathbf{c}, \mathbf{c}' \in \{0,1\}^n$, if $\mathbf{c}'$ and $\mathbf{c}$ share a common resulting sequence after r deletions and s insertions in both, then $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$.*

Lemma 7. *For a positive integer $n \geq 3$, the number of divisors of n is upper bounded by $2^{1.6 \ln n/(\ln\ln n)}$.*

The proofs of Lemma 1, Lemma 2, Lemma 3, and Lemma 4 rely on several propositions, the details of which will be presented in the next sections. For convenience, a dependency graph for the theorem, lemmas and propositions is given in Fig. 2.

III. Proof of Theorem 1

Based on the lemmas stated in Section II, in this section we present the encoding function $\mathcal{E}$ and the decoding function $\mathcal{D}$ of our k-deletion correcting code, and prove Theorem 1. Given any sequence $\mathbf{c} \in \{0,1\}^n$, let the function $\mathcal{E} : \{0,1\}^n \to \{0,1\}^{n+8k\log n + o(\log n)}$ be given by

$$\mathcal{E}(\mathbf{c}) = (T(\mathbf{c}), R'(\mathbf{c}), R''(\mathbf{c})),$$

where

$$R'(\mathbf{c}) = (f(\mathbb{1}_{sync}(T(\mathbf{c}))) \bmod p(T(\mathbf{c})), p(T(\mathbf{c}))), \text{ and}$$
$$R''(\mathbf{c}) = Rep_{k+1}(H(R'(\mathbf{c}))).$$

Function $Rep_{k+1}(H(R'(\mathbf{c})))$ is the $(k+1)$-fold repetition of the bits in $H(R'(\mathbf{c}))$, where function $H(R'(\mathbf{c}))$ is defined Lemma 5 and protects $R'(\mathbf{c})$ from k deletions. Function $T(\mathbf{c})$ is defined in Lemma 4 and transforms $\mathbf{c}$ into a k-dense sequence (see Definition 2 for definition of a k-dense sequence). Function $f(\mathbb{1}_{sync}(T(\mathbf{c})))$ in $R'(\mathbf{c})$ is represented by an integer and protects the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ by Lemma 1. Function $p(T(\mathbf{c}))$ is defined in Lemma 2, which compresses the hash $f(\mathbb{1}_{sync}(T(\mathbf{c})))$. Function $Hash_k(T(\mathbf{c}))$ is defined in Lemma 3 and protects a k-dense sequence from k deletions.

Note that $k = o(\sqrt{\log\log n})$. Hence, according to Lemma 2, Lemma 3, and Lemma 4, the length of $R'(\mathbf{c})$ is $N_1 = 8k\log(n + 3k + 3\lceil \log k \rceil + 15) + o(\log(n + 3k + 3\lceil \log k \rceil + 15)) = 8k \log n + o(\log n)$. The length of $R''(\mathbf{c})$ is $N_2 = 2k(k+1)(N_1/\lceil \log n \rceil)\log\log n = o(\log n)$. The length of $T(\mathbf{c})$ is $n + N_0 = n + 3k + 3\lceil \log k \rceil + 15$. Therefore, the length of $\mathcal{E}(\mathbf{c})$ is $n + N_0 + N_1 + N_2 = n + 8k\log n + o(\log n)$. The redundancy of the code is $8k \log n + o(\log n)$.

To show how the sequence $\mathbf{c}$ can be recovered from a length $N-k$ subsequence $\mathbf{d}$ of $\mathcal{E}(\mathbf{c})$ and implement the computation of the decoding function $\mathcal{D}(\mathbf{d})$, we prove that

1) **Statement 1:** The redundancy $R'(\mathbf{c})$ can be recovered given $(d_{n+N_0+N_1+1}, \ldots, d_{n+N_0+N_1+N_2-k})$.
2) **Statement 2:** The sequence $\mathbf{c}$ can be recovered given $(d_1, \ldots, d_{n+N_0-k})$ and $R'(\mathbf{c})$.

We first prove Statement 1. Note that $d_{n++N_0+N_1+1}$ and $d_{n+N_0+N_1+N_2-k}$ come from bits $\mathcal{E}(\mathbf{c})_{n+N_0+N_1+1+x}$ and $\mathcal{E}(\mathbf{c})_{n+N_0+N_1+N_2-k+y}$ respectively after deletions in $\mathcal{E}(\mathbf{c})$, where $x, y \in [0,k]$. Hence $(d_{n+N_0+N_1+1}, \ldots, d_{n+N_0+N_1+N_2-k})$ is a length $N_2 - k$ subsequence of $(\mathcal{E}(\mathbf{c})_{n+N_0+N_1+1+x}, \ldots, \mathcal{E}(\mathbf{c})_{n+N_0+N_1+N_2-k+y})$, and thus a subsequence of $(\mathcal{E}(\mathbf{c})_{n+N_0+N_1+1}, \ldots, \mathcal{E}(\mathbf{c})_{n+N_0+N_1+N_2}) = R''(\mathbf{c})$. Since $R''(\mathbf{c})$ is $k+1$-fold repetition code and thus a k-deletion correcting code that protects $H(R'(\mathbf{c}))$, the hash function $H(R'(\mathbf{c}))$ can be recovered from $(d_{n+N_0+N_1+1}, \ldots, d_{n+N_0+N_1+N_2-k})$.

Similarly, $(d_{n+N_0+1}, \ldots, d_{n+N_0+N_1-k})$ is a length $N_1 - k$ subsequence of $(\mathcal{E}(\mathbf{c})_{n+N_0+1}, \ldots, \mathcal{E}(\mathbf{c})_{n+N_0+N_1}) = R'(\mathbf{c})$. From Lemma 5, the function $R'(\mathbf{c})$ can be recovered from $H(R'(\mathbf{c}))$ and $(d_{n+N_0+1}, \ldots, d_{n+N_0+N_1-k})$. Hence Statement 1 holds.

We now prove Statement 2. Note that $R'(\mathbf{c})$ contains hashes $(f(\mathbb{1}_{sync}(T(\mathbf{c}))) \bmod p(T(\mathbf{c})), p(T(\mathbf{c})))$ and $Hash_k(T(\mathbf{c}))$. Moreover, $(d_1, \ldots, d_{n+N_0-k})$ is a length $n + N_0 - k$ subsequence of $(\mathcal{E}(\mathbf{c})_1, \ldots, \mathcal{E}(\mathbf{c})_{n+N_0}) = T(\mathbf{c})$. According to Lemma 2, the synchronization vector $\mathbb{1}_{sync}(T(\mathbf{c}))$ can be recovered from hash $(f(\mathbb{1}_{sync}(T(\mathbf{c}))) \bmod p(T(\mathbf{c})), p(T(\mathbf{c})))$ and $(d_1, \ldots, d_{n+N_0-k})$, by exhaustively searching over all length $n + N_0$ supersequence $\mathbf{c}'$ of $(d_1, \ldots, d_{n+N_0-k})$ such that $f(\mathbb{1}_{sync}(\mathbf{c}')) = f(\mathbb{1}_{sync}(T(\mathbf{c})))$. Then we have that $\mathbb{1}_{sync}(T(\mathbf{c})) = \mathbb{1} + sync(\mathbf{c}')$. Since by Lemma 4, $T(\mathbf{c})$ is a k-dense sequence, by Lemma 3, it can be recovered from $\mathbb{1}_{sync}(T(\mathbf{c}))$, $Hash_k(T(\mathbf{c}))$, and the length $n + N_0 - k$ subsequence $(d_1, \ldots, d_{n+N_0-k})$ of $T(\mathbf{c})$. Finally, the

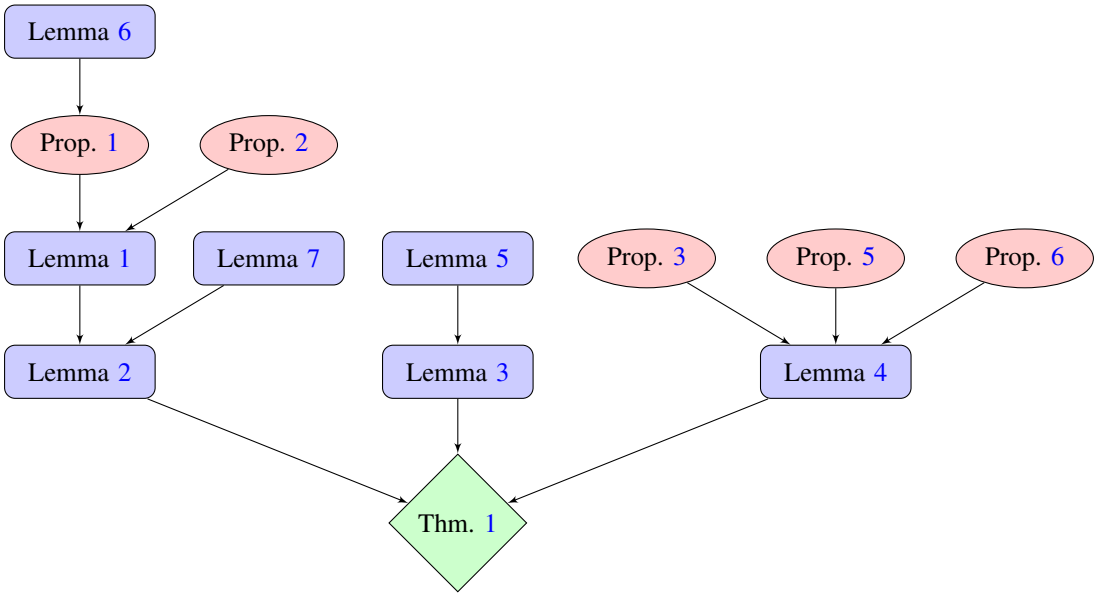

Fig. 2. Dependencies of the claims in the paper.

sequence $\mathbf{c}$ can be recovered from $T(\mathbf{c})$ by Lemma 4. Hence Statement 2 holds and $\mathbf{c}$ can be recovered.

The encoding complexity of $\mathcal{E}(\mathbf{c})$ is $O(n^{2k+1})$, which comes from brute force search for integer $p(T(\mathbf{c}))$. The decoding compliexity is $O(n^k + 1)$, which comes from brute force search for the correct $\mathbb{1}_{sync}(T(\mathbf{c}))$, given $f(\mathbb{1}_{sync}(T(\mathbf{c}))) \bmod p(T(\mathbf{c}))$ and $p(T(\mathbf{c}))$.

IV. Protecting the Synchronization Vectors

In this section we present a hash function with size $4k \log n + o(\log n)$ to protect the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ from k deletions in $\mathbf{c}$ and prove Lemma 2. We first prove Lemma 1, which is decomposed to Proposition 1 and Proposition 2. In Proposition 1 we present an upper bound on the radius of the deletion ball for the synchronization vector. In Proposition 2, we prove that the higher order parity check helps correct multiple deletions for sequences in which the there is a 0-run of length at least $3k-1$ between any two 1's. Since $\mathbb{1}_{sync}(\mathbf{c})$ is such a sequence, we conclude that the higher order parity check helps recover $\mathbb{1}_{sync}(\mathbf{c})$. After obtaining a bound on the difference between the higher order parity checks of two ambiguous sequence, we then apply Proposition 2 on the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ to prove Lemma 1, which replaces the higher order parity checks in Proposition 2 by the higher parity checks modulo a numbers. After proving Lemma 1, we use Lemma 7 to further compress the size of the higher order parity check that protects $\mathbb{1}_{sync}(\mathbf{c})$ and then prove Lemma 2.

Proposition 1. *For* $\mathbf{c}, \mathbf{c}' \in \{0,1\}^n$, *if* $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$, *then* $\mathbb{1}_{sync}(\mathbf{c}') \in \mathcal{B}_{3k}(\mathbb{1}_{sync}(\mathbf{c}))$.

Proof. Since $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$, the sequences $\mathbf{c}'$ and $\mathbf{c}$ share a common subsequence after k deletions in both. We now show that a single deletion in $\mathbf{c}$ causes at most two deletions and one insertion in its synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$. We first show that a deletion in $\mathbf{c}$ can destroy and generate at most 1 synchronization pattern. This is because for any synchronization pattern that is destroyed or generated, there must be a deletion that occurs within the synchronization pattern. Hence any two destroyed or generated synchronization patterns cannot be caused by the same deletion. Therefore, we need to consider four cases in total. Let $\mathbf{d}'$ be the subsequence of $\mathbf{c}$ after a single deletion.

1) The deletion destroys a synchronization pattern $(c_{i+1}, \ldots, c_{i+3k+\lceil \log k \rceil + 4})$ for some i and no synchronization pattern is generated. Then the sequence $\mathbb{1}_{sync}(\mathbf{d}')$ can be obtained by deleting the 1 entry $\mathbb{1}_{sync}(\mathbf{c})_{i+3k}$ in $\mathbb{1}_{sync}(\mathbf{c})$.
2) The deletion generates a new synchronization pattern $(c'_{i'+1}, \ldots, c'_{i'+3k+\lceil \log k \rceil + 4})$ for some i' and destroys a synchronization pattern $(c_{i+1}, \ldots, c_{i+3k+\lceil \log k \rceil + 4})$. The sequence $\mathbb{1}_{sync}(\mathbf{d}')$ can be obtained by deleting the 1 entry $\mathbb{1}_{sync}(\mathbf{c})_{i+3k}$ and the 0 entry $\mathbb{1}_{sync}(\mathbf{c})_{i+3k-1}$ in $\mathbb{1}_{sync}(\mathbf{c})$ and inserting a 1 entry at $\mathbb{1}_{sync}(\mathbf{c})_{i'+3k}$.
3) The deletion generates a new synchronization pattern $(c'_{i'+1}, \ldots, c'_{i'+3k+\lceil \log k \rceil + 4})$ for some i' and no synchronization pattern is destroyed. Then the $\mathbb{1}_{sync}(\mathbf{d}')$ can be obtained by deleting two 0 entries $\mathbb{1}_{sync}(\mathbf{c})_{i'+3k}$ and $\mathbb{1}_{sync}(\mathbf{c})_{i'+3k+1}$ in $\mathbb{1}_{sync}(\mathbf{c})$ and inserting a 1 entry at $\mathbb{1}_{sync}(\mathbf{c})_{i'+3k}$.
4) No synchronization pattern is generated or destroyed. Then $\mathbb{1}_{sync}(\mathbf{d}')$ can be obtained by deleting a 0 entry $\mathbb{1}_{sync}(\mathbf{c})_j$, where j is the location of the deletion.

In summary, in each of the above cases, a single deletion in $\mathbf{c}$ causes at most two deletions and one insertion in $\mathbb{1}_{sync}(\mathbf{c})$. Hence k deletions in $\mathbf{c}$ and $\mathbf{c}'$ cause at most $2k$ deletions and k insertions in $\mathbb{1}_{sync}(\mathbf{c})$ and $\mathbb{1}_{sync}(\mathbf{c}')$ respectively. According to Lemma 6, we have that $\mathbb{1}_{sync}(\mathbf{c}') \in \mathcal{B}_{3k}(\mathbb{1}_{sync}(\mathbf{c}))$ when $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$. Hence, Proposition 1 is proved. □

Let $\mathcal{R}_m$ be the set of length n sequences in which there

is a 0 run of length at least $m-1$ between any two 1's. Any two 1's in a sequence $\mathbf{c} \in \mathcal{R}_m$ have index distance at least m. The following lemma shows that the sequences in $\mathcal{R}_{3k}$ can be protected using higher order parity checks. Note that compared to the higher order parity checks $f(\mathbf{c})$, the higher order parity checks in the following proposition do not have modulo operations.

Proposition 2. *For sequences $\mathbf{c}, \mathbf{c}' \in \mathcal{R}_{3k}$, if $\mathbf{c}' \in \mathcal{B}_{3k}(\mathbf{c})$ and $\mathbf{c} \cdot \mathbf{m}^{(e)} = \mathbf{c}' \cdot \mathbf{m}^{(e)}$ for $e \in [0, 6k]$, then $\mathbf{c} = \mathbf{c}'$.*

Proof. We first compute the difference $\mathbf{c} \cdot \mathbf{m}^{(e)} - \mathbf{c}' \cdot \mathbf{m}^{(e)}$, $e \in [0, 6k]$. Since $\mathbf{c}' \in \mathcal{B}_{3k}(\mathbf{c})$, there exist two subsets $\boldsymbol{\delta} = \{\delta_1, \ldots, \delta_{3k}\} \subset [1, n]$ and $\boldsymbol{\delta}' = \{\delta'_1, \ldots, \delta'_{3k}\} \subset [1, n]$ such that deleting bits with indices $\boldsymbol{\delta}$ and $\boldsymbol{\delta}'$ respectively from $\mathbf{c}$ and $\mathbf{c}'$ results in the same length $n-3k$ subsequence, i.e., $(c_i : i \notin \boldsymbol{\delta}) = (c'_i : i \notin \boldsymbol{\delta}')$. Let $\boldsymbol{\Delta} = \{i : c_i = 1\}$ and $\boldsymbol{\Delta}' = \{i : c'_i = 1\}$ be the indices of 1 entries in $\mathbf{c}$ and $\mathbf{c}'$ respectively. Let $S_1 = \boldsymbol{\Delta} \cap \boldsymbol{\delta}$ be the indices of 1 entries that are deleted in $\mathbf{c}$. Then $S_1^c = \boldsymbol{\Delta} \cap ([1, n] \backslash \boldsymbol{\delta})$ denotes the indices of 1 entries that are not deleted. Similarly, let $S_2 = \boldsymbol{\Delta}' \cap \boldsymbol{\delta}'$ and $S_2^c = \boldsymbol{\Delta}' \cap ([1, n] \backslash \boldsymbol{\delta}')$ be the indices of 1 entries that are deleted and not in $\mathbf{c}'$ respectively. Let the elements in $\boldsymbol{\delta} \cup \boldsymbol{\delta}'$ be ordered by $1 \le p_1 \le p_2 \le \ldots \le p_{6k} \le n$. Denote $p_0 = 0$ and $p_{6k+1} = n$. Then we have that

$$
\begin{aligned}
&\mathbf{c} \cdot \mathbf{m}^{(e)} - \mathbf{c}' \cdot \mathbf{m}^{(e)} \\
=&\sum_{\ell \in \boldsymbol{\Delta}} \mathbf{m}_\ell^{(e)} - \sum_{\ell \in \boldsymbol{\Delta}'} \mathbf{m}_\ell^{(e)} \\
=&\sum_{\ell \in \boldsymbol{\Delta}} (\sum_{i=1}^{\ell} i^e) - \sum_{\ell \in \boldsymbol{\Delta}'} (\sum_{i=1}^{\ell} i^e) \\
=&\sum_{i=1}^{n} (\sum_{\ell \in \boldsymbol{\Delta} \cap [i,n]} i^e) - \sum_{i=1}^{n} (\sum_{\ell \in \boldsymbol{\Delta}' \cap [i,n]} i^e) \\
=&\sum_{i=1}^{n} (|\boldsymbol{\Delta} \cap [i, n]| - |\boldsymbol{\Delta}' \cap [i, n]|) i^e \\
=&\sum_{i=1}^{n} (|S_1 \cap [i, n]| + |S_1^c \cap [i, n]| - |S_2 \cap [i, n]| \\
&- |S_2^c \cap [i, n]|) i^e \\
=&\sum_{j=0}^{6k} \sum_{i=p_j+1}^{p_{j+1}} (|S_1 \cap [i, n]| - |S_2 \cap [i, n]| + |S_1^c \cap [i, n]| \\
&- |S_2^c \cap [i, n]|) i^e \\
\overset{(a)}{=}&\sum_{j=0}^{6k} \sum_{i=p_j+1}^{p_{j+1}} (|S_1 \cap [p_{j+1}, n]| \\
&- |S_2 \cap [p_{j+1}, n]| + |S_1^c \cap [i, n]| - |S_2^c \cap [i, n]|) i^e, \quad (6)
\end{aligned}
$$

where (a) holds since by definition of p_j, there is no deleted 1 entry in interval $(p_j, p_{j+1}) = \{p_j + 1, \ldots, p_{j+1} - 1\}$, $j \in [0, 6k]$. In the following we show

Statement 1: $-1 \le |S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \le 1$ for $i \in [1, n]$.

Statement 2: For each interval $(p_j, p_{j+1}] = \{p_j + 1, \ldots, p_{j+1}\}$, $j = 0, \ldots, 6k$, we have either $|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \le 0$ for all $i \in (p_j, p_{j+1}]$ or $|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \ge 0$ for all $i \in (p_j, p_{j+1}]$.

We first prove **Statement** 1. Note that deleting bits with indices $\boldsymbol{\delta}$ in $\mathbf{c}$ and deleting bits with indices $\boldsymbol{\delta}'$ in $\mathbf{c}'$ result in the same subsequence. Hence, for every $i \in S_1^c$, there is a unique corresponding index $i' \in S_2^c$ such that the two 1 entries c_i and $c'_{i'}$ end in the same location after deletions, i.e., $i - |\boldsymbol{\delta} \cap [1, i-1]| = i' - |\boldsymbol{\delta}' \cap [1, i'-1]|$. This implies that $|i' - i| \le 3k$. Fix integers i and i'. Then by definition of i and i', for every $x \in S_1^c \cap [i+1, n]$, there is a unique corresponding $y \in S_2^c \cap [i'+1, n]$ such that the two 1 entries c_x and c'_y end in the same location after deletions. Therefore, we have that $|S_1^c \cap [i+1, n]| = |S_2^c \cap [i'+1, n]|$, and thus that $|S_1^c \cap [i, n]| = |S_2^c \cap [i', n]|$. If $i' \ge i$, then

$$
\begin{aligned}
&|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \\
=&|S_1^c \cap [i, n]| - |S_2^c \cap [i', n]| - |S_2^c \cap [i, i'-1]| \\
=&-|S_2^c \cap [i, i'-1]| \\
\overset{(a)}{\ge}&-|S_2^c \cap [i, i+3k-1]| \\
\overset{(b)}{\ge}&-1,
\end{aligned}
$$

where (a) follows from the fact that $i' \le i+3k$ and (b) follows from the fact that $\mathbf{c}, \mathbf{c}' \in \mathcal{R}_{3k}$. Also we have that $|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| = -|S_2^c \cap [i, i'-1]| \le 0$. Hence when $i' \le i$, we have that $-1 \le |S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \le 0$. Similarly, when $i' < i$, we have that

$$
\begin{aligned}
&|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \\
=&|S_1^c \cap [i, n]| - |S_2^c \cap [i', n]| + |S_2^c \cap [i', i-1]| \\
=&|S_2^c \cap [i', i-1]| \\
\le&|S_2^c \cap [i', i'+3k-1]| \\
\le&1,
\end{aligned}
$$

and that $|S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| = |S_2^c \cap [i', i-1]| \ge 0$. Therefore, we have that $0 \le |S_1^c \cap [i, n]| - |S_2^c \cap [i, n]| \le 1$ when $i' < i$. Thus **Statement** 1 is proved.

We now prove **Statement** 2 by contradiction. Suppose on the contrary, there exist $i_1, i_2 \in (p_j, p_{j+1}]$ such that $i_1 < i_2$ and

$$
\begin{aligned}
&(|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]|)(|S_1^c \cap [i_2, n]| - |S_2^c \cap [i_2, n]|) \\
&< 0
\end{aligned}
$$

From **Statement 1** we have that $|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]| \in [-1, 1]$ and that $|S_1^c \cap [i_2, n]| - |S_2^c \cap [i_2, n]| \in [-1, 1]$. Hence by symmetry it can be assumed that $|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]| = -1$ and $|S_1^c \cap [i_2, n]| - |S_2^c \cap [i_2, n]| = 1$. As shown in proof of **Statement 1**, for every element $i \in S_1^c$, there is a corresponding element $i' \in S_2^c$ such that the two 1 entries c_i and $c'_{i'}$ end in the same location after deletions. Hence, for $y = \min_{i \in S_2^c \cap [i_1, n]} i$, there exists an integer $x \in S_1^c$ such that the two 1 entries c_x and c'_y are in the same location after deletions, i.e., $x - |\boldsymbol{\delta} \cap [1, x-1]| = y - |\boldsymbol{\delta}' \cap [1, y-1]|$. Since $|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]| = -1$, we have that $x \in S_1^c \cap [1, i_1 - 1]$. Otherwise, we have that $x \in S_1^c \cap [i_1, n]$ and for every integer $i' \in S_2^c \cap (y, n]$, there exists an integer $i \in S_1^c \cap (x, n]$ such that c_i and $c_{i'}$ end up in the same location after deletions. This implies that $|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]| \ge 0$,

contradicting the fact that $|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]| = -1$. Therefore,

$$\begin{aligned} i_1 - |\boldsymbol{\delta} \cap [1, i_1 - 1]| >& i_1 - 1 - |\boldsymbol{\delta} \cap [1, i_1 - 1]| \\ \geq& x - |\boldsymbol{\delta} \cap [1, x - 1]| \\ =& y - |\boldsymbol{\delta}' \cap [1, y - 1]| \\ \geq& i_1 - |\boldsymbol{\delta}' \cap [1, i_1 - 1]|, \end{aligned}$$

which implies that

$$|\boldsymbol{\delta} \cap [1, i_1 - 1]| < |\boldsymbol{\delta}' \cap [1, i_1 - 1]|. \tag{7}$$

Similarly, from $|S_1^c \cap [i_2, n]| - |S_2^c \cap [i_2, n]| = 1$ we have that

$$|\boldsymbol{\delta} \cap [1, i_2 - 1]| > |\boldsymbol{\delta}' \cap [1, i_2 - 1]|. \tag{8}$$

Eq. (7) and Eq. (8) implies that

$$\begin{aligned} & |\boldsymbol{\delta} \cap [1, i_2 - 1]| - |\boldsymbol{\delta} \cap [1, i_1 - 1]| \\ \geq& |\boldsymbol{\delta}' \cap [1, i_2 - 1]| + 1 - |\boldsymbol{\delta}' \cap [1, i_1 - 1]| + 1 \\ \geq& 2. \end{aligned} \tag{9}$$

However, since $i_1, i_2 \in (p_j, p_{j+1}]$ and no deletion occurs in the interval $(p_j, p_{j+1}]$, we have that $|\boldsymbol{\delta} \cap [1, i_1]| = |\boldsymbol{\delta} \cap [1, i_2 - 1]|$ and $|\boldsymbol{\delta}' \cap [1, i_1]| = |\boldsymbol{\delta}' \cap [1, i_2 - 1]|$, which implies that

$$\begin{aligned} & |\boldsymbol{\delta} \cap [1, i_2 - 1]| - |\boldsymbol{\delta} \cap [1, i_1 - 1]| \\ \leq& |\boldsymbol{\delta} \cap [1, i_2 - 1]| - |\boldsymbol{\delta} \cap [1, i_1]| + 1 \\ =& 1, \end{aligned}$$

contradicting Eq. (9). Hence there do not exist different integers $i_1, i_2 \in (p_j, p_{j+1}]$ such that

$$\begin{aligned} & (|S_1^c \cap [i_1, n]| - |S_2^c \cap [i_1, n]|)(|S_1^c \cap [i_2, n]| - |S_2^c \cap [i_2, n]|) \\ & < 0. \end{aligned}$$

Hence **Statement** 2 is proved.

Now we continue to prove Proposition 2. Denote

$$s_i \triangleq |S_1 \cap [i, n]| - |S_2 \cap [i, n]| + |S_1^c \cap [i, n]| - |S_2^c \cap [i, n]|. \tag{10}$$

Note that no deletion occurs in the interval $(p_j, p_{j+1}]$, it follows that

$$\begin{aligned} & |S_1 \cap [i, n]| - |S_2 \cap [i, n]| \\ =& |S_1 \cap [p_{j+1}, n]| - |S_2 \cap [p_{j+1}, n]| \end{aligned} \tag{11}$$

for $i \in (p_j, p_{j+1}]$. Combining (11) with **Statement 1** and **Statement 2**, we conclude that for each interval $(p_j, p_{j+1}]$, $j \in \{0, \ldots, 6k\}$, either $s_i \geq 0$ for all $i \in (p_j, p_{j+1}]$ or $s_i \leq 0$ for all $i \in (p_j, p_{j+1}]$. Let $\mathbf{x} = (x_0, \ldots, x_{6k}) \in \{-1, 1\}^{6k+1}$ be a vector defined by

$$x_i = \begin{cases} -1, & \text{if } s_j < 0 \text{ for some } j \in (p_i, p_{i+1}] \\ 1, & \text{else.} \end{cases}.$$

Then from Eq. (6) and Eq. (10), the difference $\mathbf{c} \cdot \mathbf{m}^{(e)} - \mathbf{c}' \cdot \mathbf{m}^{(e)}$ is given by

$$\mathbf{c} \cdot \mathbf{m}^{(e)} - \mathbf{c}' \cdot \mathbf{m}^{(e)} = \sum_{j=0}^{6k} \Big(\sum_{i=p_j+1}^{p_{j+1}} |s_i| i^e \Big) x_j \tag{12}$$

Let A be a $6k+1 \times 6k+1$ matrix with entries defined by $A_{e,j} = \sum_{i=p_{j-1}+1}^{p_j} |s_i| i^{e-1}$ for $e, j \in [1, 6k+1]$. If $\mathbf{c} \cdot \mathbf{m}^{(e)} = \mathbf{c}' \cdot \mathbf{m}^{(e)}$ for $e \in [0, 6k]$, we have the following linear equation

$$\begin{aligned} A\boldsymbol{x} =& \begin{bmatrix} \sum_{i=p_0+1}^{p_1} |s_i| i^0 & \cdots & \sum_{i=p_{6k}+1}^{p_{6k+1}} |s_i| i^0 \\ \vdots & \ddots & \vdots \\ \sum_{i=p_0+1}^{p_1} |s_i| i^{6k} & \cdots & \sum_{i=p_{6k}+1}^{p_{6k+1}} |s_i| i^{6k} \end{bmatrix} \begin{bmatrix} x_0 \\ \vdots \\ x_{6k} \end{bmatrix} \\ =& 0, \end{aligned} \tag{13}$$

with a solution $x_i \in \{-1, 1\}$ for $i \in [0, 6k]$. We show that this is impossible unless A is a zero matrix. Suppose on the contrary that A is nonzero, let $j_1 < \ldots < j_Q$ be the indices of all nonzero columns of A. Let A^* be a submatrix of A, obtained by choosing the intersection of the first Q rows and columns with indices $j_1, \ldots, j_Q$. Then taking the first Q linear equations from the equation set (14) and noting that the nonzero columns in A are the $j_1, \ldots, j_Q$-th columns, we have that

$$\begin{aligned} & A^* \boldsymbol{x}' \\ =& \begin{bmatrix} \sum_{i=p_{j_1-1}+1}^{p_{j_1}} |s_i| i^0 & \cdots & \sum_{i=p_{j_{Q-1}}+1}^{p_{j_Q}} |s_i| i^0 \\ \vdots & \ddots & \vdots \\ \sum_{i=p_{j_1-1}+1}^{p_{j_1}} |s_i| i^{Q-1} & \cdots & \sum_{i=p_{j_{Q-1}}+1}^{p_{j_Q}} |s_i| i^{Q-1} \end{bmatrix} \begin{bmatrix} x_{j_1} \\ \vdots \\ x_{j_Q} \end{bmatrix} \\ =& 0 \end{aligned} \tag{14}$$

The determinant of A^* is given by

$$\begin{aligned} & \det(A^*) \\ =& \det \begin{pmatrix} \sum_{i=p_{j_1-1}+1}^{p_{j_1}} |s_i| i^0 & \cdots & \sum_{i=p_{j_{Q-1}}+1}^{p_{j_Q}} |s_i| i^0 \\ \vdots & \ddots & \vdots \\ \sum_{i=p_{j_1-1}+1}^{p_{j_1}} |s_i| i^{Q-1} & \cdots & \sum_{i=p_{j_{Q-1}}+1}^{p_{j_Q}} |s_i| i^{Q-1} \end{pmatrix} \\ \overset{(a)}{=}& \sum_{\substack{i_1 \in (p_{j_1-1}, p_{j_1}], \ldots, \\ i_Q \in (p_{j_Q-1}, p_{j_Q}]}} \det \begin{pmatrix} |s_{i_1}| i_1^0 & \cdots & |s_{i_Q}| i_Q^0 \\ \vdots & \ddots & \vdots \\ |s_{i_1}| i_1^{Q-1} & \cdots & |s_{i_Q}| i_Q^{Q-1} \end{pmatrix} \\ \overset{(b)}{=}& \sum_{\substack{i_1 \in (p_{j_1-1}, p_{j_1}], \ldots, \\ i_Q \in (p_{j_Q-1}, p_{j_Q}]}} \Big[\prod_{q=1}^{Q} |s_{i_q}| \det \begin{pmatrix} i_1^0 & \cdots & i_Q^0 \\ \vdots & \ddots & \vdots \\ i_1^{Q-1} & \cdots & i_Q^{Q-1} \end{pmatrix} \Big] \\ \overset{(c)}{=}& \sum_{\substack{i_1 \in (p_{j_1-1}, p_{j_1}], \ldots, \\ i_Q \in (p_{j_Q-1}, p_{j_Q}]}} [\prod_{q=1}^{Q} |s_{i_q}| \prod_{1 \leq m < \ell \leq Q} (i_\ell - i_m)], \end{aligned} \tag{15}$$

where equality (a) follows from the multi-linearity of the determinant

$$\begin{aligned} & \det([\mathbf{v}_1 \ \ldots \ a\mathbf{v}_i + b\mathbf{v} \ \ldots \ \mathbf{v}_q]) \\ =& a \det([\mathbf{v}_1 \ \ldots \ \mathbf{v}_i \ \ldots \ \mathbf{v}_q]) \\ & + b \det([\mathbf{v}_1 \ \ldots \ \mathbf{v}_{i-1} \ \mathbf{v} \ \mathbf{v}_{i+1} \ \ldots \ \mathbf{v}_q]) \end{aligned}$$

for any integers q and i and q-dimensional vectors $\mathbf{v}_1, \ldots, \mathbf{v}_q$, $\mathbf{v}$. Equality (b) follows from the linearity of the determinant

$$\det([\mathbf{v}_1 \ \ldots \ a\mathbf{v}_i \ \ldots \ \mathbf{v}_q]) = a \det([\mathbf{v}_1 \ \ldots \ \mathbf{v}_i \ \ldots \ \mathbf{v}_q])$$

for any integers q and i and q-dimensional vectors $\mathbf{v}_1, \ldots, \mathbf{v}_q$. Equality (c) follows from the determinant of Vandermonde matrix

$$\det \begin{pmatrix} i_1^0 & \ldots & i_q^0 \\ \vdots & \ddots & \vdots \\ i_1^{q-1} & \ldots & i_q^{q-1} \end{pmatrix} = \prod_{1 \le m < \ell \le q} (i_\ell - i_m)$$

for any integers q, $i_1, \ldots, i_q$. . The determinant $det(A^*)$ is positive since $i_\ell > i_m$ for $\ell > m$. and for $i_1 \in (p_{j_1-1}, p_{j_1}], \ldots, i_Q \in (p_{j_Q-1}, p_{j_Q}]$. Note that all the columns of A^* are nonzero. Therefore, the linear equation $A^* \boldsymbol{x}' = 0$ does not have nonzero solutions, contradicting the fact that $\boldsymbol{x}' = (x_{j_1}, \ldots, x_{j_Q}) \in \{-1, 1\}^Q$. Hence A is a zero matrix, meaning that

$$\begin{aligned} &|S_1 \cap [i,n]| - |S_2 \cap [i,n]| + |S_1^c \cap [i,n]| - |S_2^c \cap [i,n]| \\ =&|\boldsymbol{\Delta} \cap [i,n]| - |\boldsymbol{\Delta}' \cap [i,n]| = 0 \end{aligned}$$

for $i \in \{1, \ldots, n\}$. This implies $\boldsymbol{\Delta} = \boldsymbol{\Delta}'$ and thus $\mathbf{c} = \mathbf{c}'$. Hence Proposition 2 is proved. □

A. Proof of Lemma 1

We are now ready to prove Lemma 1, which states that $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$ for sequences $\mathbf{c}$ and $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$ satisfying $f(\mathbb{1}_{sync}(\mathbf{c})) = f(\mathbb{1}_{sync}(\mathbf{c}))$. From Proposition 1 we have that $\mathbb{1}_{sync}(\mathbf{c}') \in \mathcal{B}_{3k}(\mathbb{1}_{sync}(\mathbf{c}))$. Then, it is not hard to see that $(\mathbb{1}_{sync}(\mathbf{c}')_i, \ldots, \mathbb{1}_{sync}(\mathbf{c}')_n) \in \mathcal{B}_{3k}((\mathbb{1}_{sync}(\mathbf{c})_i, \ldots, \mathbb{1}_{sync}(\mathbf{c})_n))$. This implies that $||\boldsymbol{\Delta} \cap [i,n]| - |\boldsymbol{\Delta}' \cap [i,n]|| \le 3k$, where $\boldsymbol{\Delta} = \{i : \mathbb{1}_{sync}(\mathbf{c})_i = 1\}$ and $\boldsymbol{\Delta}' = \{i : \mathbb{1}_{sync}(\mathbf{c}')_i = 1\}$. According to the forth line in Eq. (6), we have that

$$\begin{aligned} &|\mathbb{1}_{sync}(\mathbf{c}) \cdot \mathbf{m}^{(e)} - \mathbb{1}_{sync}(\mathbf{c}') \cdot \mathbf{m}^{(e)}| \\ =&|\sum_{i=1}^{n} (|\boldsymbol{\Delta} \cap [i,n]| - |\boldsymbol{\Delta}' \cap [i,n]|) i^e|, \\ \le&\sum_{i=1}^{n} 3k i^e \\ <&3kn^{e+1}. \end{aligned} \tag{16}$$

If $f(\mathbb{1}_{sync}(\mathbf{c})) = f(\mathbb{1}_{sync}(\mathbf{c}'))$, then

$$\mathbb{1}_{sync}(\mathbf{c}) \cdot \mathbf{m}^{(e)} \equiv \mathbb{1}_{sync}(\mathbf{c}') \cdot \mathbf{m}^{(e)} \bmod 3kn^{e+1} \tag{17}$$

for $e \in [0, 6k]$. Equations (17) and (16) imply that $\mathbb{1}_{sync}(\mathbf{c}) \cdot \mathbf{m}^{(e)} = \mathbb{1}_{sync}(\mathbf{c}') \cdot \mathbf{m}^{(e)}$ for $e \in [0, 6k]$. Since $\mathbb{1}_{sync}(\mathbf{c}') \in \mathcal{B}_{3k}(\mathbb{1}_{sync}(\mathbf{c}))$ and $\mathbb{1}_{sync}(\mathbf{c}), \mathbb{1}_{sync}(\mathbf{c}') \in \mathcal{R}_{3k}$, from Proposition 2 we conclude that $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$. Hence Lemma 1 is proved.

B. Proof of Lemma 2

Based on Lemma 1, we now show Lemma 2. Specifically, we show that there exists a function $p : \{0,1\}^n \to [1, 2^{2k \log n + o(\log n)}]$ such that $\mathbb{1}_{sync}(\mathbf{c}) = \mathbb{1}_{sync}(\mathbf{c}')$ for sequences $\mathbf{c}$ and $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$ satisfying $(f(\mathbb{1}_{sync}(\mathbf{c})) \bmod p(\mathbf{c}), p(\mathbf{c})) = (f(\mathbb{1}_{sync}(\mathbf{c}')) \bmod p(\mathbf{c}'), p(\mathbf{c}'))$.

Lemma 1 implies that $f(\mathbb{1}_{sync}(\mathbf{c})) \neq f(\mathbb{1}_{sync}(\mathbf{c}'))$ for $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c}) \backslash \{\mathbf{c}\}$, if $\mathbb{1}_{sync}(\mathbf{c}) \neq \mathbb{1}_{sync}(\mathbf{c}')$. Hence $|f(\mathbb{1}_{sync}(\mathbf{c})) - f(\mathbb{1}_{sync}(\mathbf{c}'))| \neq 0$ for $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c}) \backslash \{\mathbf{c}\}$, where $f(\mathbb{1}_{sync}(\mathbf{c}))$ and $f(\mathbb{1}_{s}ync(\mathbf{c}'))$ denote the integer presentation of their vector form. The integers are in the range $[0, (3k)^{6k+1} n^{(3k+1)(6k+1)} - 1]$. According to Lemma 7, the number of divisors of $|f(\mathbb{1}_{sync}(\mathbf{c})) - f(\mathbb{1}_{sync}(\mathbf{c}'))|$ is upper bounded by

$$\begin{aligned} &2^{2[(3k+1)(6k+1)\ln n + (6k+1)\ln 3k] / \ln((3k+1)(6k+1)\ln n + (6k+1)\ln 3k)} \\ &= 2^{o(\log n)}, \end{aligned}$$

where the equality holds since $k = o(\sqrt{\log \log n})$. For any sequence $\mathbf{c} \in \{0,1\}^n$, let

$$\begin{aligned} &\mathcal{P}(\mathbf{c}) = \{p : p \text{ divides } |f(\mathbb{1}_{sync}(\mathbf{c}')) - f(\mathbb{1}_{sync}(\mathbf{c}))| \\ &\text{for some } \mathbf{c}' \in \mathcal{B}_k(\mathbf{c}) \backslash \{\mathbf{c}\} \text{ such that } \mathbb{1}_{sync}(\mathbf{c}) \neq \mathbb{1}_{sync}(\mathbf{c}')\} \end{aligned}$$

be the set of all divisors of the numbers $\{|f(\mathbb{1}_{sync}(\mathbf{c}')) - f(\mathbb{1}_{sync}(\mathbf{c}))| : \mathbf{c}' \in \mathcal{B}_k(\mathbf{c}) \backslash \{\mathbf{c}\} \text{ and } \mathbb{1}_{sync}(\mathbf{c}) \neq \mathbb{1}_{sync}(\mathbf{c}')\}$. Since $|\mathcal{B}_k(\mathbf{c})| \le \binom{n}{k}^2 2^k \le 2n^{2k}$, we have that

$$\begin{aligned} |\mathcal{P}(\mathbf{c})| \le& 2n^{2k} 2^{o(\log n)} \\ =& 2^{2k \log n + o(\log n)}. \end{aligned}$$

Therefore, there exists a number $p(\mathbf{c}) \in [1, 2^{2k \log n + o(\log n)}]$ such that $p(\mathbf{c})$ does not divide $|f(\mathbb{1}_{sync}(\mathbf{c}')) - f(\mathbb{1}_{sync}(\mathbf{c}))|$ for all $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c}) \backslash \{\mathbf{c}\}$ satisfying $\mathbb{1}_{sync}(\mathbf{c}) \neq \mathbb{1}_{sync}(\mathbf{c}')$. Hence, if $f(\mathbb{1}_{sync}(\mathbf{c}')) \equiv f(\mathbb{1}_{sync}(\mathbf{c})) \bmod p(\mathbf{c})$ and $\mathbf{c}' \in \mathcal{B}_k(\mathbf{c})$, we have that $p(\mathbf{c})$ divides $|f(\mathbb{1}_{sync}(\mathbf{c}')) - f(\mathbb{1}_{sync}(\mathbf{c}))|$, and thus that $\mathbb{1}_{sync}(\mathbf{c}') = \mathbb{1}_{sync}(\mathbf{c})$. This completes the proof of Lemma 2.

V. Hash for k-dense sequences

In this section, we present a hash function of size $4k \log n + o(\log n)$ bits for correcting k deletions in a k-dense sequence $\mathbf{c}$, when the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$ is known. This proves Lemma 3. Recall that a sequence $\mathbf{c}$ is k-dense if there is a 0 run of length at most L between any two 1's in the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$, where L is given in (5).

Let the indices of the 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ be $t_1 < t_2 < \ldots < t_J$, where $J = \sum_{i=1}^{n} \mathbb{1}_{sync}(\mathbf{c})_i$ is the number of 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$. For notation convenience, let $t_0 = 0$ and $t_{J+1} = n + 1$. Split $\mathbf{c}$ into blocks $\mathbf{a}_0, \ldots, \mathbf{a}_J$, where

$$\mathbf{a}_j = (c_{t_j+1}, c_{t_j+2}, \ldots, c_{t_{j+1}-1}) \tag{18}$$

for $j \in [0, J]$. The blocks are separated by the synchronization patterns. Since $\mathbf{c}$ is k-dense, the length $|\mathbf{a}_j|$ of $\mathbf{a}_j$ is at most L. The goal is to protect all blocks $\mathbf{a}_j$, $j \in [0, J]$ and then $\mathbf{c}$.

We will show that given $\mathbb{1}_{sync}(\mathbf{c})$ and a length $n - k$ subsequence $\mathbf{d}$ of $\mathbf{c}$, most of the blocks $\mathbf{a}_j$ can be recovered with up to $2k$ block errors. This is done by noticing that no deletion occurs in most of the blocks and their boundary, which are marked by the synchronization patterns in $\mathbf{c}$. These blocks with no deletions inside are not destructed and appear in $\mathbf{d}$ with bits indices decrease by an integer at most k. They can be identified by looking at the synchronization patterns in $\mathbf{d}$.

For blocks that are not recovered, we show that they can be recovered with up to k deletion errors. Then we use the k-deletion correcting hash function $H(\mathbf{a}_j)$, $j \in [0, J]$ defined

in Lemma 5 to correct the block errors. The size of the hash $H(\mathbf{a}_j)$ is at most $L/\lceil \log n \rceil (2k \log\log n + O(1))$ bits, since the length of $\mathbf{a}_j$ is at most L. Note that the recovered blocks $\mathbf{a}_j$ result in the right hash $H(\mathbf{a}_j)$. It suffices to protect the hashes $H(\mathbf{a}_j)$, $j \in [0, J]$ using a Reed-Solomon code. Define the hash function $Hash_k$ as follows.

$$Hash_k(\mathbf{c}) = RS_{2k}((H(\mathbf{a}_0), \ldots, H(\mathbf{a}_J))), \tag{19}$$

where $RS_{2k}(\mathbf{c})$ is the redundancy of a systematic Reed-Solomon code (see e.g., [16] for an introduction to the Reed-Solomon code) protecting the length $J+1$ sequence $(H(\mathbf{a}_0), \ldots, H(\mathbf{a}_J))$ from $2k$ symbol substitution errors. Note that the redundancy of a k-error correcting Reed-Solomon code of length n and alphabet size $q \geq n-1$ is $2k \log q$ bits [16]. The symbols $H(\mathbf{a}_j)$,$j \in [0, J]$ have alphabet size at most $2^{\lceil (L/\lceil \log n \rceil) \rceil (2k \log\log n + O(1))}$ and can be represented using $\lceil (L/\lceil \log n \rceil) \rceil (2k \log\log n + O(1))$ bits. The length of $Hash_k(\mathbf{c})$ is $\max\{4k \log(J+1), 4k \lceil (L/\lceil \log n \rceil) \rceil (2k \log\log n + O(1))\}$, which equals $4k \log n + o(\log n)$ when $k = o(\sqrt{\log\log n})$.

We now present the following procedure that recovers $\mathbf{c}$ from its length $n-k$ subsequence $\mathbf{d}$, given the hash function $Hash_k(\mathbf{c})$ and the synchronization vector $\mathbb{1}_{sync}(\mathbf{c})$.

1) **Step 1:** Let $\mathbb{1}_{sync}(\mathbf{d}) \in \{0,1\}^{n-k}$ be the synchronization vector of $\mathbf{d}$. The indices of 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ are known and given by $1 \leq t_1 < \ldots < t_J \leq n$. Let $t_0 = 0$ and $t_{J+1} = n+1$.
2) **Step 2:** Let $\mathbb{1}_{sync}(\mathbf{d})_0 = \mathbb{1}_{sync}(\mathbf{d})_{n+1-k} = 1$. For each $j \in [0, J]$, if there exist two numbers $t'_j \in [t_j - k, t_j]$ and $t'_{j+1} \in [t_{j+1} - k, t_{j+1}]$ such that $\mathbb{1}_{sync}(\mathbf{d})_{t'_j} = \mathbb{1}_{sync}(\mathbf{d})_{t'_{j+1}} = 1$ and $t'_{j+1} - t'_j = t_{j+1} - t_j$, let $\mathbf{a}'_j = (d_{t'_j+1}, d_{t'_j+2}, \ldots, d_{t'_{j+1}-1})$. Else let $\mathbf{a}'_j = 0$.
3) **Step 3:** Apply the Reed-Solomon decoder to recover $H(\mathbf{a}_j)$ from $(H(\mathbf{a}'_0), \ldots, H(\mathbf{a}'_J), Hash_k(\mathbf{c}))$, where $\mathbf{a}_j$ is defined in (20), $j \in [0, J]$.
4) **Step 4:** Let $\mathbf{b}_j = (d_{t_j+1}, d_{t_j+2}, \ldots, d_{t_{j+1}-k-1})$ and recover $\mathbf{a}_j$ by using $\mathbf{b}_j$ and $H(\mathbf{a}_j)$. Then

$$\mathbf{c} = (\mathbf{a}_1, 1, \mathbf{a}_2, 1, \ldots, \mathbf{a}_{J-1}, 1, \mathbf{a}_J)$$

.

The following example illustrates how to recover the blocks with at most $2k$ block errors.

Example 2. *Let $k = 2$, $n = 6$, and sequence $\mathbf{c}$ the same as in Example 1, i.e.,*

$$\mathbf{c} = (1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,1,1,1,1,1,1).$$

The indices of the 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ are $\{6, 14, 23, 30\}$. Then the tuple $(t_0, t_1, t_2, t_3, t_4, t_5) = (0, 6, 14, 23, 30, 36)$ is known at the decoder. Suppose the 2 deletions occurs at the first and the last bits, resulting in a subsequence

$$\mathbf{d} = (1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,0,1,1,1,1,1).$$

Then the synchronization vector of $\mathbf{d}$ is given by

$$\mathbb{1}_{sync}(\mathbf{d}) = (0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0),$$

with indices of the 1 entries given by $\{0, 6, 13, 22, 34\}$, where it is assumed in the decoding procedure that $\mathbb{1}_{sync}(\mathbf{d})_0 = \mathbb{1}_{sync}(\mathbf{d})_{34} = 1$. Then we have that $t'_0 = 0$, $t'_1 = 6$, $t'_2 = 13$, and $t'_3 = 22$, which implies that the intervals $[t_0, t_1]$ and $[t_2, t_3]$ are good. Hence, we can recover 2 blocks $\mathbf{a}'_0 = (1,1,1,1,1)$ and $\mathbf{a}'_2 = (1,1,1,1,1,0,1,0)$. The number of block errors is $3 \leq 2k$.

We now prove that the decoding procedure obtains the correct $\mathbf{c}$. Since $\mathbf{c}_{t_j} = \mathbb{1}_{sync}(\mathbf{c})_{t_j} = 1$ for $j \in [1, J]$, it suffices to show that $\mathbf{a}_j$, $j \in [0, J]$ can be recovered correctly. Note that $(d_{t_j+1}, \ldots, d_{t_{j+1}-k-1})$ is a length $|\mathbf{a}_j| - k$ subsequence of $(c_{t_j+1}, \ldots, c_{t_{j+1}-1}) = \mathbf{a}_j$. Hence $\mathbf{a}_j$ can be correctly decoded given $\mathbf{b}_j$ and $H(\mathbf{a}_j)$, $j \in [0.J]$. It is then left to recover $H(\mathbf{a}_j)$ for $j \in [0, J]$. To this end, we show that there are at most $2k$ indices j, such that $\mathbf{a}'_j \neq \mathbf{a}_j$. Since $H(\mathbf{a}_j)$ can be computed for correct blocks $\mathbf{a}_j$, there are at most $2k$ symbol errors in the sequence $(H(\mathbf{a}_0), \ldots, H(\mathbf{a}_J))$, which can be corrected given the Reed-Solomon code redundancy $Hash_k(\mathbf{c})$.

Let t''_j, $j \in [1, J]$ be the index of $\mathbf{c}_{t_j}$ in $\mathbf{d}$ after deletions in $\mathbf{c}$, where $t''_j = -1$ if $\mathbf{c}_{t_j}$ is deleted. Let $t''_0 = 0$ and $t''_{J+1} = n+1-k$. The interval $[t_j, t_{j+1}]$, $j \in [0.J]$ is called *good* if $\mathbb{1}_{sync}(\mathbf{d})_{t''_j} = \mathbb{1}_{sync}(\mathbf{d})_{t''_{j+1}} = 1$ and $t''_{j+1} - t''_j = t_{j+1} - t_j$. We now show that $\mathbf{a}'_j = \mathbf{a}_j$ if the interval $[t_j, t_{j+1}]$ is *good*. Note that the bits $d_{t''_j}$ and $d_{t''_{j+1}}$ come from c_{t_j} and $c_{t_{j+1}}$ respectively after deletions in $\mathbf{c}$. Hence if $[t_j, t_{j+1}]$ is good, we have that $t''_{j+1} - t''_j = t_{j+1} - t_j$, and thus that

$$(d_{t''_j+1}, \ldots, d_{t''_{j+1}-1}) = (c_{t_j+1}, \ldots, c_{t_{j+1}-1}) = \mathbf{a}_j. \tag{20}$$

Moreover, we have that $\mathbb{1}_{sync(\mathbf{d})_{t''_j}} = \mathbb{1}_{sync}(\mathbf{d})_{t''_{j+1}} = 1$. Since $t_j - t''_j \leq k$ and $t_{j+1} - t''_{j+1} \leq k$ by definition of t''_j and t''_{j+1}, it follows from Step 2 in the decoding procedure that $t'_j = t''_j$, $t'_{j+1} = t''_{j+1}$, and $\mathbf{a}'_j = (d_{t''_j+1}, \ldots, d_{t''_{j+1}-1})$. Hence from (20), we conclude that $\mathbf{a}'_j = \mathbf{a}_j$ when the interval $[t_j, t_{j+1}]$ is good.

Next, we show that a deletion can destroy at most 2 *good* intervals. Notice that if no deletion occurs in $\mathbf{c}$, then all intervals $[t_j, t_{j+1}]$, $j \in [0, J]$ are good. If no deletions occur in the interval $[t_{j-1}, t_{j+2}]$, then $\mathbb{1}_{sync}(\mathbf{d})_{t''_j} = \mathbb{1}_{sync}(\mathbf{d})_{t''_{j+1}} = 1$ and $t''_{j+1} - t''_j = t_{j+1} - t_j$, where t''_j is the index of c_{t_j} in $\mathbf{d}$ after deletions. Hence the interval $[t_j, t_{j+1}]$ is good when no deletion occurs in $[t_{j-1}, t_{j+2}]$. It follows that a deletion that occurs in interval $[t_j, t_{j+1}]$ can destroy at most 3 good intervals $[t_{j-1}, t_j]$, $[t_j, t_{j+1}]$, and $[t_{j+1}, t_{j+2}]$. We prove that the deletion in $[t_j, t_{j+1}]$ can destroy at most two of them. If the deletion in $[t_j, t_{j+1}]$ does not destroy the synchronization pattern $(c_{t_j-3k+1}, \ldots, c_{t_j+\lceil \log k \rceil+4})$, then it destroys at most two good intervals $[t_j, t_{j+1}]$ and $[t_{j+1}, t_{j+2}]$. If the deletion in $[t_j, t_{j+1}]$ destroys the synchronization pattern $(c_{t_j-3k+1}, \ldots, c_{t_j+\lceil \log k \rceil+4})$, then it destroys the pattern $(c_{t_j}, \ldots, c_{t_j+\lceil \log k \rceil+4}) = \mathbf{1}^{\lceil \log k \rceil+5}$ and the synchronization pattern $(c_{t_{j+1}-3k+1}, \ldots, c_{t_{j+1}+\lceil \log k \rceil+4})$ is not affected. As a result, the good interval $[t_{j+1}, t_{j+2}]$ is not destroyed by the deletion in $[t_j, t_{j+1}]$. At most two good intervals $[t_{j-1}, t_j]$ and $[t_j, t_{j+1}]$ are destroyed. Therefore, a deletion affects at most two *good* intervals. This implies that k deletions result

in at most $2k$ block errors $\mathbf{a}'_j \neq \mathbf{a}_j$. Therefore, the sequence $\mathbf{c}$ can be recovered.

VI. Generating k-dense sequences

In this section we present an algorithm to compute the map $T(\mathbf{c})$, which transforms any sequence $\mathbf{c} \in \{0,1\}^n$ into a *k-dense* sequence, and thus proves Lemma 4. Let $\mathbf{1}^x$ and $\mathbf{0}^y$ denote sequences of consecutive x 1's and consecutive y 0's, respectively. We first show in Proposition 3 that any sequence $\mathbf{c}$ satisfying the following two properties is a k-*dense* sequence. is k-*dense*. Then, the algorithm for computing computing $T(\mathbf{c})$ can be decomposed into two parts. In the first part, we generate a sequence $T_1(\mathbf{c})$ that satisfies Property 1. In the second part, we use $T_1(\mathbf{c})$ to compute $T(\mathbf{c})$ that satisfies both properties.

Property 1. *Every length* $B \triangleq (\lceil \log k \rceil + 5)2^{\lceil \log k \rceil + 9}\lceil \log n \rceil$ *interval of* $\mathbf{c}$ *contains the pattern* $\mathbf{1}^{\lceil \log k \rceil + 5}$*, i.e., for any integer* $i \in [1, n-B+1]$*, there exists an integer* $j \in [i, i+B-\lceil \log k \rceil - 5]$ *such that* $(c_j, c_{j+1}, \ldots, c_{j+\lceil \log k \rceil + 4}) = \mathbf{1}^{\lceil \log k \rceil + 5}$.

Property 2. *Every length* $R \triangleq (3k + \lceil \log k \rceil + 4)(\lceil \log n \rceil + 9 + \lceil \log k \rceil)$ *interval of* $\mathbf{c}$ *contains a length* $3k+\lceil \log k \rceil + 4$ *subinterval that does not contain the pattern* $\mathbf{1}^{\lceil \log k \rceil + 5}$*, i.e., for any integer* $i \in [1, n-R+1]$*, there exists an integer* $j \in [i, i+R-3k-\lceil \log k \rceil - 4]$*, such that* $(c_m, c_{m+1}, \ldots, c_{m+\lceil \log k \rceil + 4}) \neq \mathbf{1}^{\lceil \log k \rceil + 5}$ *for every* $m \in [j, j+3k-1]$.

Proposition 3. *If a sequence* $\mathbf{c}$ *satisfies Property 1 and Property 2, then it is a* k-dense *sequence.*

Proof. Let the locations of the 1 entries in $\mathbb{1}_{sync}(\mathbf{c})$ be $t_1 < \ldots < t_J$. Let $t_0 = 0$ and $t_{J+1} = n+1$. From Definition 2, it suffices to show that $t_{i+1} - t_i \leq B + R + 1 = L + 1$ for any $i \in [0, J]$.

According to Property 2, there exists an index $j^* \in [t_i, t_i + R - 3k - \lceil \log k \rceil - 4]$, such that $(c_m, c_{m+1}, \ldots, c_{m+\lceil \log k \rceil + 4}) \neq \mathbf{1}^{\lceil \log k \rceil + 5}$ for every $m \in [j^*, j^* + 3k - 1]$. According to Property 1, there exists an integer $x \in [j^* + 1, j^* + B]$ such that $(c_x, c_{x+1}, \ldots, c_{x+\lceil \log k \rceil + 4}) = \mathbf{1}^{\lceil \log k \rceil + 5}$. Let $\ell = \min\{x \geq j^* : (c_x, c_{x+1}, \ldots, c_{x+\lceil \log k \rceil + 4}) = \mathbf{1}^{\lceil \log k \rceil + 5}\}$. Then we have that $\ell \neq j^*$, $\ell \leq x \leq j^* + B$, and thus that $\ell \in [j^* + 1, j^* + B]$. In addition, $(c_m, c_{m+1}, \ldots, c_{m+\lceil \log k \rceil + 4}) \neq \mathbf{1}^{\lceil \log k \rceil + 5}$ for every $m \in [j^*, \ell) = \{j^*, \ldots, \ell - 1\}$. By definition of j^*, we have that $\ell - j^* \geq 3k$. Since $(c_\ell, c_{\ell+1}, \ldots, c_{\ell+\lceil \log k \rceil + 4}) = \mathbf{1}^{\lceil \log k \rceil + 5}$, we have that $\mathbb{1}_{sync}(\mathbf{c})_\ell = 1$. Therefore, we conclude that

$$\begin{aligned} t_{i+1} - t_i &\leq \ell - t_i \\ &\leq j^* + B - t_i \\ &\leq R + B + 1 \\ &= L + 1 \end{aligned}$$

□

A. Generating sequences satisfying Property 1

Given a sequence $\mathbf{c} \in \{0,1\}^n$, we now generate $T_1(\mathbf{c}) \in \{0,1\}^{n+2\lceil \log k \rceil + 10}$ that satisfies Property 1. The idea is to repeatedly delete the length B subsequences of $\mathbf{c}$ that do not contain the pattern $\mathbf{1}^{\lceil \log k \rceil + 5}$, and append length B subsequences containing $\mathbf{1}^{\lceil \log k \rceil + 5}$ to the end, without losing the information of the deleted subsequences. The deleting and appending procedure repeats until no length B subsequence with no $\mathbf{1}^{\lceil \log k \rceil + 5}$ pattern is found. Notice that any binary sequence containing no pattern $\mathbf{1}^{\lceil \log k \rceil + 5}$ can be regarded as a sequence of symbols with alphabet size $2^{\lceil \log k \rceil + 5} - 1$. Hence, such binary sequence can be compressed to a shorter binary sequence. In this way, we can add the index of the deleted subsequence and the pattern $\mathbf{1}^{\lceil \log k \rceil + 5}$ to the compressed sequence. The sequence keeps the same size after the deleting and appending procedure.

Note that the above procedure keeps appending length B subsequences to the end. Hence the suffix of $T_1(\mathbf{c})$ are appended bits. To guarantee that these appended bits are not deleted in the procedure, we keep track of the end index of the non-appended bits n' and always delete the bits with indices at most n'. To deal with cases when a length B subsequence to be deleted overlaps with the appended bits, we delete only non-appended bits from it. Then, we append shorter subsequences such that the total length does not change. The decoder detects the shorter appended subsequences by looking at the length of the 1 run in it.

Before presenting the details of encoding and decoding, we need the following proposition, which states that the length B binary sequences containing no $\mathbf{1}^{\lceil \log k \rceil + 5}$ pattern can be compressed to shorter binary sequences.

Proposition 4. *Let* $\mathcal{S}$ *be the set of sequences* $\mathbf{b} \in \{0,1\}^B$ *such that* $(b_i, \ldots, b_{i+\lceil \log k \rceil + 4}) \neq \mathbf{1}^{\lceil \log k \rceil + 5}$ *for every* $i \in [1, B - \lceil \log k \rceil - 4]$*. There exists an invertible map* $\phi : \mathcal{S} \to \{0,1\}^{B - \lceil \log n \rceil - 2\lceil \log k \rceil - 12}$*, such that both* ϕ *and its inverse* ϕ^{-1} *can be computed in* $O(B)$ *time.*

Proof. For any $\mathbf{b} \in \mathcal{S}$, the procedure for computing $\phi(\mathbf{b})$ is as follows. Split $\mathbf{b}$ into $2^{\lceil \log k \rceil + 9}\lceil \log n \rceil$ blocks of length $(\lceil \log k \rceil + 5)$. Since each block is not $\mathbf{1}^{\lceil \log k \rceil + 5}$, it can be represented by a symbol of alphabet size $2^{\lceil \log k \rceil + 5} - 1$. Therefore, the sequence $\mathbf{b}$ can be uniquely represented by a sequence $\mathbf{v}$ of $2^{\lceil \log k \rceil + 9}\lceil \log n \rceil$ symbols, each having alphabet size $2^{\lceil \log k \rceil + 5} - 1$. Convert $\mathbf{v}$ into a binary sequence $\phi(\mathbf{b})$. Then $\phi(\mathbf{b})$ can be represented by a binary sequence with length

$$\begin{aligned} &\lceil \log_2[(2^{\lceil \log k \rceil + 5} - 1)^{2^{\lceil \log k \rceil + 9}\lceil \log n \rceil}] \rceil \\ =&\lceil \log_2[(1 - 1/2^{\lceil \log k \rceil + 5})^{2^{\lceil \log k \rceil + 9}\lceil \log n \rceil}]] \\ &+ (\lceil \log k \rceil + 5)2^{\lceil \log k \rceil + 9}\lceil \log n \rceil \\ \leq& 16\lceil \log n \rceil \log_2[(1 - 1/2^{\lceil \log k \rceil + 5})^{2^{\lceil \log k \rceil + 5}}] + B + 1 \\ \overset{(a)}{\leq}& -16\lceil \log n \rceil \log_2 e + B + 1 \\ \leq& B - 16\lceil \log n \rceil + 1 \end{aligned}$$

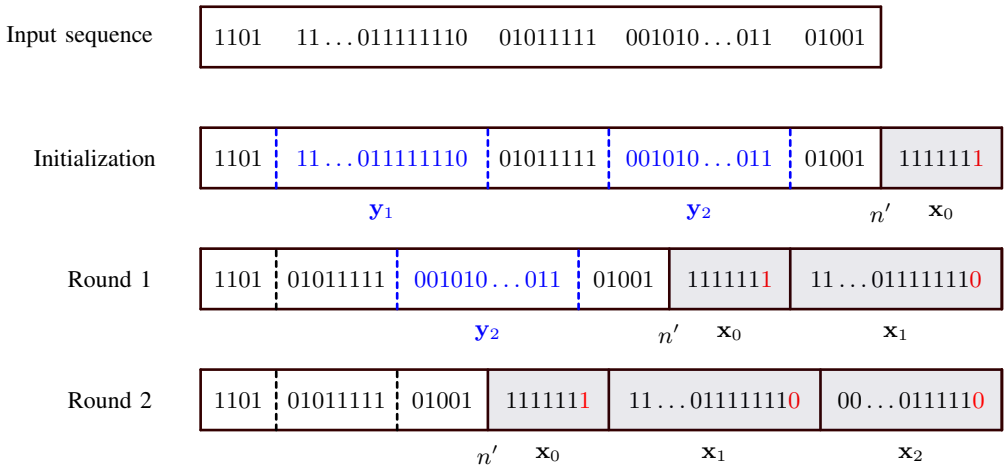

Fig. 3. This figure shows an example of how the encoding in Proposition 5 proceeds. The subsequences $\mathbf{x}_0$, $\mathbf{x}_1$, and $\mathbf{x}_2$ are the subsequences appended in the Initialization step, the first, and the second round of delete and insert procedure, respectively. The subsequence $\mathbf{y}_1$ and $\mathbf{y}_2$ are the subsequences deleted in the first and the second round of delete and insert procedure, respectively. The integer n' indicates the end point of the non-appended bits.

$$\leq B - \lceil \log n \rceil - 2\lceil \log k \rceil - 12,$$

where (a) follows from the fact that the function $(1-1/x)^x$ is increasing in x for $x > 1$ and that $\lim_{x\to\infty}(1-1/x)^x = 1/e$. Therefore, $\phi(\mathbf{b})$ can be represented by $B - \lceil \log n \rceil - 2\lceil \log k \rceil - 12$ bits. The inverse map ϕ^{-1} can be computed by converting $\phi(\mathbf{b})$ back to a length $2^{\lceil \log k \rceil + 9}\lceil \log n \rceil$ sequence $\mathbf{v}$ of alphabet size $2^{\lceil \log k \rceil + 5} - 1$. Then, concatenate the binary representation of symbols in $\mathbf{v}$, we obtain $\mathbf{b}$.

The complexity for computing ϕ or ϕ^{-1} is that of converting binary sequences to sequences of alphabet size $2^{\lceil \log k \rceil + 5} - 1$ or vice versa, which is $O(B)$. □

With the function ϕ defined in Proposition 4, the following proposition provides details of the encoding/decoding for computing $T_1(\mathbf{c})$. An example illustrating the encoding in Proposition 5 is presented in Fig. 3.

Proposition 5. *For integers k and $n > k$, there exists an invertible map $T_1 : \{0,1\}^n \to \{0,1\}^{n+2\lceil \log k \rceil + 10}$, computable in $O(n^2 k \log n \log^2 k)$ time, such that $T_1(\mathbf{c})$ satisfies Property 1. Moreover, either $(T_1(\mathbf{c})_{n+\lceil \log k \rceil + 6}, \ldots, T_1(\mathbf{c})_{n+2\lceil \log k \rceil + 10}) = \mathbf{1}^{\lceil \log k \rceil + 5}$ or $(T_1(\mathbf{c})_{n+\lceil \log k \rceil + 5}, \ldots, T_1(\mathbf{c})_{n+2\lceil \log k \rceil + 9}) = \mathbf{1}^{\lceil \log k \rceil + 5}$.*

Proof. For a sequence $\mathbf{c} \in \{0,1\}^n$, the encoding procedure for computing $T_1(\mathbf{c})$ is as follows.

1) **Initialization:** Let $T_1(\mathbf{c}) = \mathbf{c}$. Append $\mathbf{1}^{2\lceil \log k \rceil + 10}$ to the end of the sequence $T_1(\mathbf{c})$. Let $n' = n$. Go to Step 1.
2) **Step 1:** If there exists an integer $i \in [1, n']$ such that $(T_1(\mathbf{c})_j, T_1(\mathbf{c})_{j+1}, \ldots, T_1(\mathbf{c})_{j+\lceil \log k \rceil + 4}) \neq \mathbf{1}^{\lceil \log k \rceil + 5}$ for every $j \in [i, i + B - \lceil \log k \rceil - 5]$, go to Step 2. Else go to Step 4.
3) **Step 2:** If $i > n' - B + 1$, go to Step 3. Else, delete $(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{i+B-1})$ from $T_1(\mathbf{c})$ and append $(i, \phi(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{i+B-1}), 0, \mathbf{1}^{2\lceil \log k \rceil + 10}, 0)$ to the end of $T_1(\mathbf{c})$, where the appended i is encoded by $\lceil \log n \rceil$ binary bits. Let $n' = n' - B$. Go to Step 1.
4) **Step 3:** Delete $(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{n'})$ from $T_1(\mathbf{c})$ and append $(i, \phi(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{n'}, \mathbf{0}^{i+B-n'-1}), 0, \mathbf{1}^{2\lceil \log k \rceil + 10 - (i+B-n'-1)}, 0)$ to the end of $T_1(\mathbf{c})$. Let $n' = i - 1$, where the appended i is encoded by $\lceil \log n \rceil$ binary bits. Go to Step 1.
5) **Step 4:** Output $T_1(\mathbf{c})$.

In the encoding procedure, Step 2 and Step 3 are the deleting and appending operation described above, while Step 3 deals with the case when the deleted subsequence overlaps with the appended bits. Note that the index of the deleted subsequence is provided in the appended subsequence, so that the decoder can recover the deleted subsequence, given the appended subsequence. According to Proposition 4 and the fact that the index i has size $\lceil \log n \rceil$ bits, the lengths of the deleted and appended subsequences in Step 2 or Step 3 are equal. Hence, the sequence $T_1(\mathbf{c})$ keeps constant and is $n + 2\lceil \log k \rceil + 10$.

We first show that the integer n' is the split index of appended bits and non-appended bits. Specifically, $(T_1(\mathbf{c})_{n'+1}, \ldots, T_1(\mathbf{c})_{n+2\lceil \log k \rceil + 10})$ are the appended bits and $(T_1(\mathbf{c})_1, \ldots, T_1(\mathbf{c})_{n'})$ are non-appended bits that remain after deleting operations in Step 2 and Step 3. In addition, $(T_1(\mathbf{c})_{n'+1}, \ldots, T_1(\mathbf{c})_{n'+2\lceil \log k \rceil + 10}) = \mathbf{1}^{2\lceil \log k \rceil + 10}$ are the bits appended in the Initialization step. The claim holds in the Initialization step. Note that in each round of Step 2 or Step 3, the deleted bits have indices at most n' and the integer n' decreases by the length of deleted subsequence. Hence, the claim always holds.

We now show that the output sequence $T_1(\mathbf{c})$ satisfies Property 1.

We have shown that $(T_1(\mathbf{c})_{n'+1}, \ldots, T_1(\mathbf{c})_{n'+\lceil \log k \rceil + 5}) = \mathbf{1}^{\lceil \log k \rceil + 5}$. Then according to the if conditions in Step 1 and Step 2 that lead to Step 3, the integer i in Step 3 satisfies $1 \leq i + B - n' - 1 \leq \lceil \log k \rceil + 4$. Otherwise the subsequence $(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{i+B-1})$ contains $(T_1(\mathbf{c})_{n'+1}, \ldots, T_1(\mathbf{c})_{n'+\lceil \log k \rceil + 5}) = \mathbf{1}^{\lceil \log k \rceil + 5}$, which does not satisfy the if condition in Step 1. Therefore, the 1

run $\mathbf{1}^{2\lceil \log k\rceil+10-(i+B-n'-1)}$ in the appended subsequence in Step 3 has length at least $\lceil \log k\rceil + 6$. Thus, the $\mathbf{1}^{\lceil \log k\rceil+5}$ pattern appears in the subsequence appended in each round of Step 2 or Step 3. Moreover, the index distance between the two $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns in two consecutively appended subsequences is at most $B-\lceil \log k\rceil-5$. We conclude that for $i' > n'$, any length B subsequence $(T_1(\mathbf{c})_i, \ldots, T_1(\mathbf{c})_{i+B-1})$ contains the $\mathbf{1}^{\lceil \log k\rceil+5}$ pattern. Furthermore, note that for any $i \in [1, n']$, there exists some $j \in [i, i+B-\lceil \log k\rceil-5]$ such that $T_1(\mathbf{c})_j = T_1(\mathbf{c})_{j+1} = \ldots = T_1(\mathbf{c})_{j+\lceil \log k\rceil+4} = 1$. Otherwise $T_1(\mathbf{c})_i$ is deleted in Step 2 or Step 3. Hence, the sequence $T_1(\mathbf{c})$ satisfies Property 1.

Note that the integer n' decreases in each round. Hence, the algorithm terminates within $O(n)$ rounds of Step 1, Step 2 and Step 3. Therefore, the search for the i satisfying the if condition in Step 1 takes $O(nB\log k)$ time. In addition, the deleting and appending operation in Step 2 or Step 3 takes at most $O(n+B)$ time. Hence, the total complexity is $O(n^2 k \log n \log^2 k)$.

Next, we show that either $(T_1(\mathbf{c})_{n+\lceil \log k\rceil+6}, \ldots, T_1(\mathbf{c})_{n+2\lceil \log k\rceil+10}) = \mathbf{1}^{\lceil \log k\rceil+5}$ or $(T_1(\mathbf{c})_{n+\lceil \log k\rceil+5}, \ldots, T_1(\mathbf{c})_{n+2\lceil \log k\rceil+9}) = \mathbf{1}^{\lceil \log k\rceil+5}$. The former holds when the appending operation only occurs in the Initialization step. Note that all subsequence appended in Step 2 or Step 3 ends with a 1 run with length at least $\lceil \log k\rceil + 6$ followed by a 0 bit. Hence the latter holds when the appending operation in Step 2 or Step 3 occurs.

The decoding follows a reverse procedure of the encoding, by repeatedly removing the appended subsequences and use them to recover the deleted subsequences. Since the appended subsequences contain the ϕ function and position of the deleted subsequences, the deleted subsequences can be recovered. The decoding stops when the end of the appended sequence becomes a 1 bit. The decoder determines the length of the appended subsequence by looking at the 1 run before the ending 0 bit. The decoding procedure that recovers $\mathbf{c}$ from $T_1(\mathbf{c})$ is given as follows.

1) **Initialization:** Let $\mathbf{c} = T_1(\mathbf{c})$ and go to Step 1.
2) **Step 1:** If $c_{n+2\lceil \log k\rceil+10} = 0$, find the length ℓ of the 1 run that ends with $c_{n+2\lceil \log k\rceil+9}$. Let i be the integer representation of $(c_{n+4\lceil \log k\rceil+21-B-\ell}, c_{n+4\lceil \log k\rceil+22-B-\ell}, \ldots, c_{n+4\lceil \log k\rceil+20-B-\ell+\lceil \log n\rceil})$. Let $\mathbf{b}$ be the sequence obtained by computing $\phi^{-1}(c_{n+4\lceil \log k\rceil+21-B-\ell+\lceil \log n\rceil}, c_{n+4\lceil \log k\rceil+22-B-\ell+\lceil \log n\rceil}, \ldots, c_{n+2\lceil \log k\rceil+8-\ell})$, where the function ϕ is defined in Proposition 4, Delete $(c_{n+4\lceil \log k\rceil+21-B-\ell}, c_{n+4\lceil \log k\rceil+22-B-\ell}, \ldots, c_{n+2\lceil \log k\rceil+10})$ from $\mathbf{c}$ and insert $(\mathbf{b}_1, \ldots, \mathbf{b}_{B-2\lceil \log k\rceil-10+\ell})$ at location i of $\mathbf{c}$. Repeat. Else delete $c_{n+1}, \ldots, c_{n+2\lceil \log k\rceil+10}$ and go to Step 2.
3) **Step 2:** Output $\mathbf{c}$.

In each round of Step 1, the decoder processes an appended subsequence of length B or less. It can be seen that the appended subsequences in the encoding procedure end with a 0 bit except for the one appended in the Initialization step. Hence if the end of an appended sequence is a 1 bit, the subsequence is the $\mathbf{1}^{2\lceil \log k\rceil+10}$ appended in the Initialization step of the encoding procedure. Moreover, since $\mathbf{1}^{2\lceil \log k\rceil+10}$ is the appended subsequence, the decoding procedure ends up in $\mathbf{1}^{2\lceil \log k\rceil+10}$ after processing all other appended subsequences.

Note that the encoding procedure consists of a series of deleting and appending operations. The decoding procedure exactly reverses the series of operations in the encoding procedure. Let $T_{1,i}(\mathbf{c})$, $i \in [0, I]$ be the sequence $T_1(\mathbf{c})$ obtained after the i-th deleting and appending operation in the encoding procedure, where I is the number of deleting and appending operations in total in the encoding procedure. We have that $T_{1,0}(\mathbf{c}) = \mathbf{c}$ and that $T_{1,I}(\mathbf{c})$ is the final output $T_1(\mathbf{c})$. Then the decoding procedure obtains $T_{1,I-i}(\mathbf{c})$, $i \in [0, I]$ after the i-th deleting and inserting operation. Hence we get the output $T_{1,I-I}(\mathbf{c}) = \mathbf{c}$ in the decoding procedure. □

B. Proof of Lemma 4

After having the sequence $T_1(\mathbf{c}) \in \{0,1\}^{n+2\lceil \log k\rceil+10}$ satisfying Property 1, we now use $T_1(\mathbf{c})$ to generate a sequence $T(\mathbf{c}) \in \{0,1\}^{n+3k+3\lceil \log k\rceil+15}$ that satisfies both Property 1 and Property 2. According to Proposition 3, $T(\mathbf{c})$ is a k-dense sequence and Lemma 4 is proved. The encoding of $T(\mathbf{c})$ follows a similar repetitive deleting and appending procedure to the encoding in Proposition 5. To satisfy Property 2, we repeatedly look for length R subsequences, every length $3k+\lceil \log k\rceil+4$ subsequence of which contains $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns, delete most part of it, and then append a subsequence that contains a length $3k+\lceil \log k\rceil+4$ subsequence containing no $\mathbf{1}^{\lceil \log k\rceil+5}$ pattern. The appended subsequence has all information about the deleted sequence, including the index and the bits, and has the same length as that of the deleted subsequence. To this end, we need to construct a map that compresses a length $3k+\lceil \log k\rceil+4$ sequence containing $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns to a shorter sequence. Moreover, the compressed sequence does not contain $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns. Then we can compress the deleted subsequence and add indices and markers as we did in Proposition 5. The end of an appended subsequence is marked by a 0 bit. We also keep track of the integer n', which is the end of non-appended bits, to guarantee that the appended bits are not deleted.

Note that we do not delete the whole length R subsequences, every length $3k+\lceil \log k\rceil+4$ subsequence of which contains $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns. Instead, we split the length R subsequence into blocks of length $3k+\lceil \log k\rceil+4$, each containing $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns. Then the first and the last blocks remain and the blocks in the middle are deleted. This is to keep the sequence $T(\mathbf{c})$ satisfying Property 1 and protect the appended bits from being deleted.

The following proposition presents the compression map described above, which encodes a sequence containing $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns into a shorter sequence without the $\mathbf{1}^{\lceil \log k\rceil+5}$ pattern. Similar to the encoding procedures that compute $T(\mathbf{c})$ and $T_1(\mathbf{c})$, the algorithm for computing the compression map follows a delete and append process, which repeatedly deletes $\mathbf{1}^{\lceil \log k\rceil+5}$ patterns and appends their indices and 0 runs to the end.

Proposition 6. *For an integer k, let $\mathbf{c} \in \{0,1\}^{3k+\lceil \log k \rceil +4}$ be a sequence such that $c_i = c_{i+1} = \ldots = c_{i+\lceil \log k \rceil +4} = 1$ for some $i \in [1, 3k]$. There exists an invertible mapping $T_2 : \{0,1\}^{3k+\lceil \log k \rceil +4} \to \{0,1\}^{3k+\lceil \log k \rceil +3}$, such that $T_2(\mathbf{c})$ contains no $\lceil \log k \rceil + 5$ consecutive 1 bits. Both T_2 and its inverse T_2^{-1} are computable in $O(k^2 \log k)$ time.*

Proof. Given $\mathbf{c} \in \{0,1\}^{3k+\lceil \log k \rceil +4}$, the encoding procedure for computing $T_2(\mathbf{c})$ is as follows.

1) **Initialization:** Let $T_2(\mathbf{c}) = \mathbf{c}$. Append 0 to the end of the sequence $T_2(\mathbf{c})$. Find the smallest $i \in [1, 3k]$ such that $T_2(\mathbf{c})_i = T_2(\mathbf{c})_{i+1} = \ldots = T_2(\mathbf{c})_{i+\lceil \log k \rceil +4} = 1$. Delete $(T_2(\mathbf{c})_i, \ldots, T_2(\mathbf{c})_{i+\lceil \log k \rceil +4})$ from $T_2(\mathbf{c})$ and append $(i, \mathbf{0}^{\lceil \log k \rceil +3-\lceil \log(3k) \rceil})$ to the end of $T_2(\mathbf{c})$, where the appended i is encoded by $\lceil \log 3k \rceil$ binary bits. . Let $n' = 3k-1$ and $i = 1$. Go to Step 1.
2) **Step 1:** If there exists an integer $i \leq n'$ such that $T_2(\mathbf{c})_i = T_2(\mathbf{c})_{i+1} = \ldots = T_2(\mathbf{c})_{i+\lceil \log k \rceil +4} = 1$, delete $(T_2(\mathbf{c})_i, \ldots, T_2(\mathbf{c})_{i+\lceil \log k \rceil +4})$ from $T_2(\mathbf{c})$ and append $(i, \mathbf{0}^{\lceil \log k \rceil +4-\lceil \log(3k) \rceil}, 1)$ to the end of $T_2(\mathbf{c})$. Let $n' = n' - \lceil \log k \rceil - 5$ and $i = 1$. Repeat. Else go to Step 2.
3) **Step 2:** Output $T_2(\mathbf{c})$.

There are deleting and appending operations in both the Initialization step and Step 1. In the Initialization step, the length of the deleted subsequence is larger than the length of the appended subsequence by 2. Hence after appending the 0 bit in the beginning, the length of $T_2(\mathbf{c})$ decreases by 1 after the Initialization step. The lengths of the deleted and appended subsequence in Step 1 are equal. Hence, the length of the sequence $T_2(\mathbf{c})$ keeps constant after the Initialization step and is

$$\begin{aligned} &3k + \lceil \log k \rceil + 4 + 1 - \lceil \log k \rceil - 5 + \lceil \log k \rceil + 3 \\ =&3k + \lceil \log k \rceil + 3. \end{aligned}$$

We show that $(T_2(\mathbf{c})_{n'+1}, \ldots, T_2(\mathbf{c})_{3k+\lceil \log k \rceil +3})$ are appended bits and $(T_2(\mathbf{c})_1, \ldots, T_2(\mathbf{c})_{n'})$ are non-appended bits remain in $T_2(\mathbf{c})$ after deletions. In particular, $T_2(\mathbf{c})_{n'+1} = 0$ is the bit appended in the beginning of the Initialization step. The claims hold after the Initialization step. Note that when $T_2(\mathbf{c})_{n'} = 0$, the integer i satisfying the if condition in Step i is at most $n' - \lceil \log k \rceil - 4$. Otherwise $(T_2(\mathbf{c})_i, \ldots, T_2(\mathbf{c})_{i+\lceil \log k \rceil +4}) \neq \mathbf{1}$. Therefore, the deleted bits in Step 1 have indices at most n'. Moreover, the integer n' decreases by the length of the deleted bits. Therefore, we conclude that $(T_2(\mathbf{c})_{n'+1}, \ldots, T_2(\mathbf{c})_{3k+\lceil \log k \rceil +3})$ consists of appended bits and that $T_2(\mathbf{c})_{n'+1} = 0$ is the bit appended at the beginning of the Initialization step. The appended bits $(T_2(\mathbf{c})_{n'+1}, \ldots, T_2(\mathbf{c})_{3k+\lceil \log k \rceil +3})$ are not deleted in the procedure.

We now show that $T_2(\mathbf{c})$ contains no $\mathbf{1}^{\lceil \log k \rceil +5}$ patterns. Note that the $\mathbf{1}^{\lceil \log k \rceil +5}$ patterns with indices at most n' are deleted in the encoding procedure. Since $T_2(\mathbf{c})_{n'+1} = 0$, there is no $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern containing the bit $T_2(\mathbf{c})_{n'+1}$. Hence a $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern has indices at least $n'+1$. Moreover, since the $\lceil \log k \rceil + 4$-th bit in each appended subsequence is a 0 bit, there is no $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern in the appended bits. Hence, no $\mathbf{1}^{\lceil \log k \rceil +5}$ with indices at least $n'+1$ exists. This implies that $T_2(\mathbf{c})$ does not contain $\mathbf{1}^{\lceil \log k \rceil +5}$ patterns.

Since n' decreases in each step, the algorithm terminates within $O(k)$ iterations of Step 1. Since it takes $O(k \log k)$ to look for the integer i, the total complexity is $O(k^2 \log k)$.

The decoding $T_2^{-1}(\mathbf{c})$, which recovers $\mathbf{c}$ from $T_2(\mathbf{c})$, follows a reverse procedure of the encoding and is presented in the following.

1) **Initialization:** Let $\mathbf{c} = T_2(\mathbf{c})$ and go to Step 1.
2) **Step 1:** If $c_{3k+\lceil \log k \rceil +3} = 1$, let i be the integer representation of $(c_{3k-1}, c_{3k}, \ldots, c_{3k+\lceil \log 3k \rceil -2})$. Delete $(c_{3k-1}, c_{3k}, \ldots, c_{3k+\lceil \log k \rceil +3})$ from $\mathbf{c}$ and insert $\mathbf{1}^{\lceil \log k \rceil +5}$ at location i of $\mathbf{c}$. Repeat. Else go to Step 2.
3) **Step 2:** Let i be the decimal representation of $(c_{3k+1}, c_{3k+2}, \ldots, c_{3k+\lceil \log(3k) \rceil})$. Delete $(c_{3k}, c_{3k+2}, \ldots, c_{3k+\lceil \log k \rceil +3})$ from $\mathbf{c}$ and insert $\mathbf{1}^{\lceil \log k \rceil +5}$ at location i of $\mathbf{c}$. Output $\mathbf{c}$.

Note that in the encoding procedure, the appended subsequence in the Initialization Step ends with a 0. The appended subsequence in Step 1 ends with a 1. Hence the algorithm stops when $\mathbf{c}_{3k+\lceil \log k \rceil +3} = 0$ and all subsequences appended in Step 1 of the encoding have be processed.

Similarly to the proof of correctness of decoding in Proposition 5, the decoding procedure exactly reverses the series of operations in the encoding procedure. Note that the appended subsequences contain the index of the deleted $\mathbf{1}^{\lceil \log k \rceil +5}$ patterns. Let $T_{2,i}(\mathbf{c})$, $i \in [0, I]$ be the sequence obtained after the i-th deleting and appending operation in the encoding procedure, where I is the number of deleting and appending operations in total in the encoding procedure. Then $T_{2,i}(\mathbf{c})$ is the sequence obtained after the $I-i$-th deleting and inserting operation in the decoding procedure. Therefore, the decoding procedure recovers the sequence $\mathbf{c}$ after the I-th operation.

The complexity of the decoding has the same order $O(k^2 \log k)$ as that of the encoding. □

We are now ready to present the encoding and decoding procedures for computing $T(\mathbf{c})$, which generates k-dense sequences that satisfy Property 1 and Property 2. The encoding procedure is as follows.

1) **Initialization:** Let $T(\mathbf{c}) = T_1(\mathbf{c})$. Append $(\mathbf{0}^{3k}, \mathbf{1}^{\lceil \log k \rceil +5})$ to the end of the sequence $T(\mathbf{c})$. Let $n' = n + 2\lceil \log k \rceil + 10$ (the length of $T_1(\mathbf{c})$). Go to Step 1.
2) **Step 1:** If there exists an integer $i \leq \min\{n', n + 3k + 3\lceil \log k \rceil + 16 - R\}$, such that for every $j \in [i, i + R - 3k - \lceil \log k \rceil - 4]$, there exists an integer $m \in [j, j + 3k - 1]$ satisfying $(T(\mathbf{c})_m, T(\mathbf{c})_{m+1}, \ldots, T(\mathbf{c})_{m+\lceil \log k \rceil +4}) = \mathbf{1}^{\lceil \log k \rceil +5}$, split $(T(\mathbf{c})_i, T(\mathbf{c})_{i+1}, \ldots, T(\mathbf{c})_{i+R-1})$ into $(\lceil \log n \rceil + 9 + \lceil \log k \rceil)$ blocks $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_{\lceil \log n \rceil +9+\lceil \log k \rceil}$ of length $3k + \lceil \log k \rceil + 4$. Delete $(\mathbf{b}_2, \ldots, \mathbf{b}_{\lceil \log n \rceil +8+\lceil \log k \rceil})$ from $T(\mathbf{c})$ and append $(0, T_2(\mathbf{b}_2), T_2(\mathbf{b}_3), \ldots, T_2(\mathbf{b}_{\lceil \log n \rceil +8+\lceil \log k \rceil}), i + 3k + \lceil \log k \rceil + 4, \mathbf{1}^{\lceil \log k \rceil +5}, 0)$ to the end of $T(\mathbf{c})$, where the appended $i + 3k + \lceil \log k \rceil + 4$ encoded by $\lceil \log n \rceil$ binary bits. Let $n' = n' - R + 6k + 2\lceil \log k \rceil + 8$. Repeat. Else go to Step 2.

3) **Step 2:** Output $T(\mathbf{c})$.

Note that the index $i+3k+\lceil \log k \rceil + 4$ has size $\log n$ bits. Then from Proposition 6, it can be verified that the lengths of the deleted and appended subsequences in Step 1 are equal. Hence $T(\mathbf{c})$ keeps constant and is $n+3k+3\lceil \log k \rceil + 15$.

Similar to the encoding in Proposition 5 and Proposition 6, we show that the integer n' marks the end of the non-appended bits, i.e., the subsequence $(T(\mathbf{c})_{n'+1}, \ldots, (\mathbf{c})_{n+3k+3\lceil \log k \rceil +15})$ consists of appended bits and $(T(\mathbf{c})_1, \ldots, T(\mathbf{c})_{n'})$ are non-appended bits that remain after deletions. In addition, we show that $T(\mathbf{c})_{n'+1}, \ldots, T(\mathbf{c})_{n'+3k} = \mathbf{0}^{3k}$ and that the appended bits are not deleted. The claims hold in the Initialization step. Suppose the claims hold in the r-th round of Step 1. Then in the $r+1$-th round of Step 1, the integer i satisfying the if condition in Step 1 must be in the range $[1, n' - R + 3k + \lceil \log k \rceil + 4]$. Otherwise, since $T(\mathbf{c})_{n'+1}, \ldots, T(\mathbf{c})_{n'+3k} = \mathbf{0}^{3k}$ in the r-th round, we have an integer $n' \in [i, i+R-3k-\lceil \log k \rceil -4]$ such that $(T(\mathbf{c})_m, T(\mathbf{c})_{m+1}, \ldots, T(\mathbf{c})_{m+\lceil \log k \rceil +4}) \neq \mathbf{1}^{\lceil \log k \rceil +5}$ for every $m \in [n', n'+3k-1]$. This contradicts to the fact that i satisfies the if condition. Moreover, note that the block $\mathbf{b}_{\lceil \log n \rceil +9+\lceil \log k \rceil}$ in $(T(\mathbf{c})_i, \ldots, T(\mathbf{c})_{i+R-1})$ is not deleted in Step 1, which implies that the bits with indices at least $i+R-3k-\lceil \log k \rceil -4 \leq n'$ are not deleted. Note that n' decreases by the length of the deleted sequence in Step 1. We conclude that $(T(\mathbf{c})_{n'+1}, \ldots, (\mathbf{c})_{n+3k+3\lceil \log k \rceil +15})$ consists of appended bits and these bits are not deleted. Specifically, $T(\mathbf{c})_{n'+1}, \ldots, T(\mathbf{c})_{n'+3k}$ are the bits appended in the Initialization step. By induction, the claims hold.

We now show by induction on the number of rounds r that $T(\mathbf{c})$ satisfies Property 1. From Proposition 5, the initial sequence $T(\mathbf{c}) = (T_1(\mathbf{c}), \mathbf{0}^{3k}, \mathbf{1}^{\lceil \log k \rceil +5})$ satisfies Property 1. Hence the claim holds for $r = 0$. Suppose after r-th round of Step 1, $T(\mathbf{c})$ satisfies Property 1. In the $r+1$-th round, the deleting operation leaves blocks $\mathbf{b}_1$ and $\mathbf{b}_{\lceil \log n \rceil +9+\lceil \log k \rceil}$, which both contain $\mathbf{1}^{\lceil \log k \rceil +5}$ as a subsequence. Hence $T(\mathbf{c})$ satisfies Property 1 after the deletion. In addition, all appended subsequences end with a $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern or a $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern followed by a 0 bit. Note that these appended subsequences are not deleted. Hence the index distance between two $\mathbf{1}^{\lceil \log k \rceil +5}$ patterns in the appended bits $(T(\mathbf{c})_{n'+1}, \ldots, T(\mathbf{c})_{n+3k+3\lceil \log k \rceil +15}$ is at most $R-6k-2\lceil \log k \rceil -8 \leq B - \lceil \log k \rceil -5$. Therefore, The sequence $T(\mathbf{c})$ satisfies Property 1 after the appending operation.

Next, we prove that $T(\mathbf{c})$ satisfies Property 2. According to the encoding procedure, for any $i \in [1, \min\{n', n+3k+3\lceil \log k \rceil + 16 - R\}]$, there exists some $j \in [i, i+R-3k-\lceil \log k \rceil -4]$, such that $(T(\mathbf{c})_m, T(\mathbf{c})_{m+1}, \ldots, T(\mathbf{c})_{m+\lceil \log k \rceil +4}) \neq \mathbf{1}^{\lceil \log k \rceil +5}$ for every $m \in [j, j+3k-1]$. Otherwise, the encoding does not stop. Note that the appended bits $(T(\mathbf{c})_{n'+1}, \ldots, T(\mathbf{c})_{n+3k+3\lceil \log k \rceil +15})$ are not deleted. Hence for $i \in [n'+1, n+3k+3\lceil \log k \rceil +16-R]$, the interval $[i, i+R-1]$ contains the first $3k+\lceil \log k \rceil +4$ bits $(0, T_2(\mathbf{b}_2))$ of some appended subsequence, where $\mathbf{b}_2 \in \{0,1\}^{3k+\lceil \log k \rceil +4}$ contains the $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern. Let $[j, j+3k+\lceil \log k \rceil +3]$ be the indices of the $3k+\lceil \log k \rceil +4$ bits $(0, T_2(\mathbf{b}_2))$ in $T(\mathbf{c})$. According to Proposition 6, the function $T_2(\mathbf{b}_2)$ does not contain the $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern. Hence $(T(\mathbf{c})_m, T(\mathbf{c})_{m+1}, \ldots, T(\mathbf{c})_{m+\lceil \log k \rceil +4}) \neq \mathbf{1}^{\lceil \log k \rceil +5}$ for every $m \in [j, j+3k-1]$. Therefore, any interval of length R contains a length $3k+\lceil \log k \rceil +4$ subsequence with no $\mathbf{1}^{\lceil \log k \rceil +5}$ pattern. Hence, the sequence $T(\mathbf{c})$ satisfies Property 2. According to Proposition 3, we conclude that $T(\mathbf{c})$ satisfies Property 1 and Property 2 and is k-*dense*.

Since n' decreases in the encoding procedure, the procedure terminates within $O(n)$ iterations. Each iteration takes $O(nk \log kR)$ time to search for the integer i satisfying the if condition and $O(\log nk^2 \log k)$ to compute the T_2 functions of the blocks. Hence the complexity is at most $O(n^2k^2 \log k(\log n))$. Therefore, the total complexity is $poly(n,k)$.

Finally we present the following decoding procedure that recovers $\mathbf{c}$ from $T(\mathbf{c})$, which follows a reverse procedure of the encoding.

1) **Initialization:** Let $\mathbf{c} = T(\mathbf{c})$ and go to Step 1.
2) **Step 1:** If $c_{n+3k+3\lceil \log k \rceil +15} = 0$, let i be the integer representation of $(c_{n+3k+2\lceil \log k \rceil +10-\lceil \log n \rceil}, c_{n+3k+2\lceil \log k \rceil +11-\lceil \log n \rceil}, \ldots, c_{n+3k+2\lceil \log k \rceil +9})$. Split $(c_{n+9k+5\lceil \log k \rceil +25-R}, \ldots, c_{n+3k+2\lceil \log k \rceil +9-\lceil \log n \rceil})$ into $\lceil \log n \rceil + 7 + \lceil \log k \rceil$ blocks $(\mathbf{b}'_1, \ldots, \mathbf{b}'_{\lceil \log n \rceil +7+\lceil \log k \rceil})$ of length $3k+\lceil \log k \rceil +3$. Compute $\mathbf{b}_j = T_2^{-1}(\mathbf{b}'_j)$ for $j \in [1, \lceil \log n \rceil +7+\lceil \log k \rceil]$, where $T_2^{-1}(\mathbf{b}'_j)$ is defined in Proposition 6. Delete $(c_{n+9k+5\lceil \log k \rceil +24-R}, \ldots, c_{n+3k+3\lceil \log k \rceil +15})$ from $\mathbf{c}$ and insert $\mathbf{b}_1, \ldots, \mathbf{b}_{\lceil \log n \rceil +7+\lceil \log k \rceil}$ at location i of $\mathbf{c}$. Repeat. Else delete $(c_{n+2\lceil \log k \rceil +11}, \ldots, c_{n+3k+3\lceil \log k \rceil +15})$ and go to Step 2.
3) **Step 2:** Output $T_1^{-1}(\mathbf{c})$.

According to the encoding procedure, the inserted bits end with a 1 entry in the Initialization Step and with a 0 entry in Step 1. Note that the inserted bits are not deleted in the encoding procedure. Hence the decoding algorithm stops when an ending 1 entry is detected.

Similar to the proof of correctness of decoding in Proposition 5 and Proposition 6, the decoding procedure exactly reverses the series of operations in the encoding procedure. Therefore, the decoding procedure decodes the sequence $\mathbf{c}$ correctly.

VII. Conclusion and Future Work

We construct a k-deletion correcting code with optimal order redundancy. Interesting open problems include finding complexity $O(N^{O(1)})$ encoding/decoding algorithms for our code, as well as constructing a systematic k-deletion correcting code with optimal redundancy.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable suggestions that greatly improved the clarity and the organization of the paper.

References

[1] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet physics doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[2] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Autom. Remote Control*, vol. 26, no. 2, pp. 286–290, 1965.
[3] M. Mitzenmacher, "A survey of results for deletion channels and related synchronization channels," *Probability Surveys*, vol. 6, pp. 1–33, 2009.
[4] A. S. Helberg and H. C. Ferreira, "On multiple insertion/deletion correcting codes," *IEEE Trans. on Inf. Th.*, vol. 48, no. 1, pp. 305–308, 2002.
[5] K. A. Abdel-Ghaffar, F. Paluncic, H. C. Ferreira, and W. A. Clarke, "On Helberg's generalization of the Levenshtein code for multiple deletion/insertion error correction," *IEEE Trans. on Inf. Th.*, vol. 58, no. 3, pp. 1804–1808, 2012.
[6] F. Paluncic, K. A. Abdel-Ghaffar, H. C. Ferreira, and W. A. Clarke, "A multiple insertion/deletion correcting code for run-length limited sequences," *IEEE Trans. on Inf. Th.*, vol. 58, no. 3, pp. 1809–1824, 2012.
[7] J. Brakensiek, V. Guruswami, and S. Zbarsky, "Efficient low-redundancy codes for correcting multiple deletions," *IEEE Trans. on Inf. Th.*, vol. 64, no. 5, pp. 3403–3410, 2018.
[8] B. Haeupler, "Optimal document exchange and new codes for small number of insertions and deletions," *IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 334–347, 2019.
[9] K. Cheng, Z. Jin, X. Li and K. Wu, "Deterministic document exchange protocols, and almost optimal binary codes for edit errors," *IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pp. 200–211, 2018.
[10] L. J. Schulman and D. Zuckerman, "Asymptotically good codes correcting insertions, deletions, and transpositions," *IEEE Trans. on Inf. Th.*, vol. 45, no. 7, pp. 2552–2557, 1999.
[11] V. Guruswami and C. Wang, "Deletion codes in the high-noise and high-rate regimes," *IEEE Trans. on Inf. Th.*, vol. 63, no. 4, pp. 1961–1970, 2017.
[12] R. Gabrys and F. Sala, "Codes correcting two deletions," *IEEE Trans. on Inf. Th.*, vol. 65, no. 2, pp. 965–974, Feb. 2019.
[13] J. Sima, N. Raviv, and J. Bruck, "Two Deletion Correcting Codes from Indicator Vectors," in *IEEE Trans. on Inf. Th.*, vol. 66, no. 4, pp. 2375-2391, 2020.
[14] S. K. Hanna and S. El Rouayheb, "Guess & check codes for deletions, insertions, and synchronization," *IEEE Trans. on Inf. Th.*, vol. 65, no. 1, pp. 3–15, Jan. 2019.
[15] J. L. Nicolas, "On highly composite numbers," in *Ramanujan revisited*, Urbana-Champaign, Ill., pp. 215-244, 1987.
[16] R. Roth, Introduction to coding theory, Cambridge University Press, 2006.

Jin Sima is a PhD candidate in the Department of Electrical Engineering at the California Institute of Technology. He received a B.Eng. and a M.Sc. in Electronic Engineering from Tsinghua University, China, in 2013 and 2016 respectively. His research interests include information and coding theory, with its applications in data storage systems. He is a recipient of the 2019 IEEE Jack Keil Wolf ISIT Student Paper Award.

Jehoshua Bruck (S'86-M'89-SM'93-F'01) is the Gordon and Betty Moore Professor of computation and neural systems and electrical engineering at the California Institute of Technology (Caltech). His current research interests include information theory and systems and the theory of computation in nature.

Dr. Bruck received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion-Israel Institute of Technology, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University, in 1989. His industrial and entrepreneurial experiences include working with IBM Research where he participated in the design and implementation of the first IBM parallel computer; cofounding and serving as Chairman of Rainfinity (acquired in 2005 by EMC), a spin-off company from Caltech that created the first virtualization solution for Network Attached Storage; as well as cofounding and serving as Chairman of XtremIO (acquired in 2012 by EMC), a start-up company that created the first scalable all-flash enterprise storage system.

Dr. Bruck is a recipient of the Feynman Prize for Excellence in Teaching, the Sloan Research Fellowship, the National Science Foundation Young Investigator Award, the IBM Outstanding Innovation Award and the IBM Outstanding Technical Achievement Award.