# Multilayer Codes for Synchronization From Deletions and Insertions

Mahed Abroshan, Ramji Venkataramanan, *Senior Member, IEEE*, and Albert Guillén i Fàbregas, *Senior Member, IEEE*

***Abstract*— Consider two remote nodes (encoder and decoder), each with a binary sequence. The encoder's sequence $X$ differs from the decoder's sequence $Y$ by a small number of edits (deletions and insertions). The goal is to construct a message $M$, to be sent via a one-way error free link, such that the decoder can reconstruct $X$ using $M$ and $Y$. In this paper, we devise a coding scheme for this one-way synchronization model. The scheme is based on multiple layers of Varshamov-Tenengolts (VT) codes combined with off-the-shelf linear error-correcting codes, and uses a list decoder. We bound the expected list size of the decoder under certain assumptions, and validate its performance via numerical simulations. We also consider an alternative decoder that uses only the constraints from the VT codes (i.e., does not require a linear code), and has a smaller redundancy at the expense of a slightly larger average list size.**

***Index Terms*— File synchronization, document exchange, edit channel, Varshomov-Tenengolts (VT) codes.**

## I. Introduction

CONSIDER two remote nodes with binary sequences $X$ and $Y$, respectively. The sequence $Y$ is an *edited* version of $X$, where the edits consist of deletions and insertions. In the synchronization model shown in Fig. 1, the node with $X$ (the encoder) sends a message $M$ via an error-free link to the other node (the decoder), which attempts to reconstruct $X$ using $M$ and $Y$. The goal is to design a scheme so that the decoder can reconstruct $X$ with minimal communication, i.e., we want to minimize the number of bits used to represent the message $M$.

This synchronization model is relevant in a number of applications including distributed file editing, and systems for file backup and sharing (e.g., Dropbox). The synchronization problem has been studied in several previous works, both in the one-way setting [1]–[8], and in the two-way setting where the encoder and decoder can exchange multiple rounds of messages [9]–[13]. Some practical file synchronization tools such as rsync [14] also use multiple rounds of information exchange. We discuss the prior work on one-way synchronization in more detail in Section I-B.

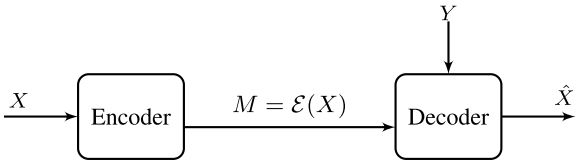


Fig. 1. Synchronization model.

We seek codes for one-way synchronization: in Fig. 1, the message $M$ is produced by the encoder using only $X$, with no knowledge of $Y$, except that the number of edits is at most $k$. We assume that the decoder knows the length of $X$, which is denoted by $n$. The message $M$ belongs to a finite set $\mathcal{M}$ with cardinality $|\mathcal{M}|$. The synchronization rate (or redundancy per symbol) is defined as $R = \frac{\log|\mathcal{M}|}{n}$. (Throughout the paper, $\log$ denotes logarithm with base 2.) We would like to design a code for reliable synchronization with $R$ as small as possible, noting that $R = 1$ is equivalent to the encoder sending the entire string $X$.

In this paper, we construct a code based on multiple layers of Varshomov-Tenengolts (VT) codes [15], for synchronization when the number of edits $k$ is small compared to $n$. The output of the decoder is a small list of sequences that is guaranteed to contain the correct sequence $X$. We observe from simulations that with a careful choice of the code parameters, the list size rarely exceeds 2 or 3; for reasonably large $n$, the list size can be made 1, i.e., $X$ is exactly reconstructed. For example, we construct a code of length $n = 378$ that can synchronize from $k = 7$ edits with $R = 0.365$, and a length $n = 2800$ code which can synchronize from $k = 10$ edits with $R = 0.135$. (Details in Section IV and VIII.)

To explain the main ideas in the code construction and the decoding algorithm, we largely focus on the case where the edits are all deletions. Section VIII describes how to modify the decoder (keeping the same encoding scheme) to reconstruct a combination of insertions and deletions.

### A. Overview of the Code Construction

The starting point for our code construction is the family of Varshamov-Tenengolts (VT) codes [15], [16]. Each VT code

Manuscript received October 14, 2019; revised June 26, 2020; accepted October 20, 2020. Date of publication November 5, 2020; date of current version May 20, 2021. This work was supported in part by the European Research Council under Grant 725411 and in part by the Spanish Ministry of Economy and Competitiveness under Grant TEC2016-78434-C3-1-R. This article was presented in part at the 2017 IEEE Information Theory Workshop. *(Corresponding author: Mahed Abroshan.)*

Mahed Abroshan was with the Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, U.K. He is now with The Alan Turing Institute, London NW1 2DB, U.K. (e-mail: mabroshan@turing.ac.uk).

Ramji Venkataramanan is with the Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, U.K. (e-mail: ramji.v@eng.cam.ac.uk).

Albert Guillén i Fàbregas is with the Department of Information and Communication Technologies, Universitat Pompeu Fabra, 08018 Barcelona, Spain, also with the Institució Catalana de Recerca i Estudis Avançats (ICREA), 08010 Barcelona, Spain, and also with the Department of Engineering, University of Cambridge, Cambridge CB2 1PZ, U.K. (e-mail: guillen@ieee.org).

Communicated by L. Dolecek, Guest Editor for the Special Issue: "From Deletion-Correction to Graph Reconstruction: In Memory of Vladimir I. Levenshtein."

Digital Object Identifier 10.1109/TIT.2020.3036284

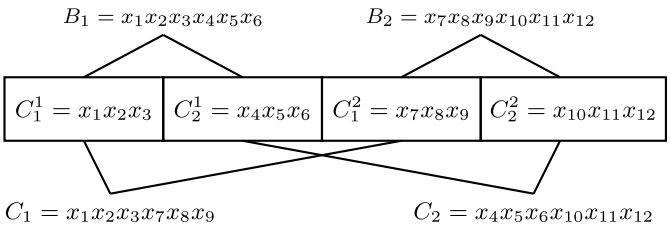


Fig. 2. Blocks and chunk-strings structure for the example where $l_1 = l_2 = 2$.

is a single edit correcting code, where the edit can be either an insertion or a deletion. A construction based on modifications of VT codes is also used for recovering a burst of consecutive deletions or insertions [17]. As observed in [1], the single edit correcting VT code provides an elegant scheme for synchronizing from a single edit: the encoder simply sends the VT syndrome of the sequence $X$. The VT syndrome (defined in the next section) indicates which VT code $X$ belongs to. The decoder then uses the single edit correcting property of the VT code to recover $X$.

In our case, the code needs to synchronize from $k > 1$ edits, assumed for now to be all deletions. The encoder sends the VT syndromes of various substrings of $X$ to the decoder. Specifically, the length $n$ sequence $X$ is divided into smaller chunks of $n_c$ bits each. The encoder then computes VT syndromes for two kinds of substrings: *blocks* which are composed of adjacent chunks, and *chunk-strings* which are composed of well-separated chunks. Fig. 2 shows an example where $X$ of length 12 is divided into 4 length-3 chunks. The blocks $B_1$ and $B_2$ are each formed by combining two adjacent chunks, while the chunk-strings $C_1$ and $C_2$ are each formed by combining two alternate chunks. In this case, the encoder sends the VT syndromes of $B_1, B_2, C_1$, and $C_2$.

The intersecting VT constraints of blocks and chunk-strings help the decoder to estimate locations of the edits. The VT syndromes serve a dual purpose: i) they are used to recover deleted bits in blocks or chunk-strings inferred to have a single deletion, and this recovery may result in new blocks and chunk-strings with a single deletion; ii) the VT syndromes also act as hashes that eliminate a large number of potential deletion patterns, allowing the decoder to localize the deletions to a relatively small set of chunks.

The final part of the message is the syndrome of $X$ with respect to the parity-check constraints of a linear code. The linear parity-check constraints are used to recover the deletions in chunks that still remain unresolved at the decoder after processing the intersecting VT constraints. We call this code construction a two-layer code as the chunks are combined to form two kinds of intersecting substrings. The construction can be generalized to combine chunks in multiple ways to form many layers of intersecting substrings. (A three-layer construction is briefly discussed in Section IX.) Increasing the number of constraints in the code improves its synchronization capability at the cost of increasing the redundancy.

For decoding, we use a list decoder. The output of the decoder is the list of all length $n$ sequences that can be obtained by inserting $k$ bits into sequence $Y$, and satisfy the VT constraints and the parity-check constraints that are imposed via message $M$. The correct sequence $X$ is always in the list.

### *B. Related Work*

In [2], Irmak et al. propose a one-way randomized scheme that synchronizes with a message of length $O(k \log^2 n)$, where $k$ is the number of edits and $n$ is the length of $X$. The scheme in [2] uses a multi-level message formed by splitting $X$ into successively smaller blocks. The message at each level is computed by applying a hash function to the blocks at that level. In a series of recent papers [4]–[7], variants of the construction in [2] have been used to achieve synchronization with message lengths of smaller order. The deterministic scheme proposed in [5] uses a message of length $O(k \log^2 \frac{n}{k})$, and the randomized scheme in [6] achieves synchronization with high probability with a message of length $O(k \log \frac{n}{k})$, which is the optimal order [1].

The goal in these works is to obtain a synchronization scheme that is *order-optimal*, i.e., a scheme with message length (redundancy) of order close to $k \log \frac{n}{k}$ and polynomial-time encoding and decoding. The constants in these results are not explicitly specified and can be quite large. For example, the message length for the scheme in [5] is at least $200\ k \log n$ [18], which implies that we need $n$ to be at least a few tens of thousands before the per-symbol redundancy is less than 1. In contrast, we are interested in practical codes to synchronize from a few edits in sequences that are a few hundred to a few thousand bits long.

From this perspective, the most relevant work to our setup is the "guess-and-check" (GC) code recently proposed by Hanna and El Rouayheb in [8]. In the GC code, the length $n$ sequence $X$ is divided into chunks of $\log n$ bits each. Assume that $n$ is a power of 2, so that each chunk can be considered as a symbol over the field $GF(n)$. The encoder's message consists of $c$ parity-check symbols of a systematic MDS code over $GF(n)$, computed with the information sequence $X$. Here $c > 2k$, where $k$ is the number of deletions. The decoder considers each pattern of $k$ deletions, and checks whether the pattern is consistent with the parity-check symbols. Decoding is successful if there is a unique sequence consistent with all the linear parity-check constraints. It is shown in [8] that the probability of successful decoding is $O(n^{-(c-2k)}/(\log n)^k)$. A list decoder for the GC code was recently considered in [19].

Our construction can be viewed as a generalization of GC code. Like the GC code, we divide the sequences into chunks and use parity-check symbols as part of the message. However, the set of syndromes of intersecting VT constraints is an essential ingredient in our construction that is not present in the GC code. The VT constraints significantly reduce the decoding complexity by localizing and correcting a large number of deletions, and reduces the number of parity-check symbols required. The parity-check symbols help to recover a small number of chunks in the original string, with the large majority of chunks being resolved using the intersecting VT constraints. In fact, in Section VII we discuss a variant of the code that does not use any parity-check constraints. The decoding

complexity of our scheme is compared in detail with the GC code in Section VI-D.

List decoding of codes for insertions and deletions was recently analyzed in [20]. Specifically, that paper obtains a lower bound for the maximum list size when the code consists of a single VT constraint, and shows that the list size can grow exponentially with the number of deletions. In contrast, our construction uses multiple intersecting VT constraints, and is therefore challenging to analyze rigorously.

The problem of one-way synchronization from $k$ deletions is closely related to the problem of communicating over a deletion channel that deletes $k$ bits from a length $n$ codeword [21]. Constructing efficient codes for the deletion channel is known to be a challenging problem, see e.g., [22]–[24]. Any one way synchronization scheme directly yields a deletion correcting code. Indeed, for a fixed message $M$ from the synchronization scheme, one can take the codebook to be the set of all sequences for which the synchronization scheme produces $M$. Using this method, we obtain a deletion correcting code corresponding to each message of the synchronization scheme. However, it may not be possible to translate a deletion code directly to a synchronization scheme. For example, an efficient $k$-deletion correcting channel code with near-optimal redundancy was recently proposed in [18]. This code has redundancy of $8k \log n + o(\log n)$ and its decoding complexity is $O(n^{2k+1})$. However, this code cannot be directly translated to the one-way synchronization model since the VT-like syndrome used in the code only works for sequences with no consecutive ones. Similarly, other practical codes for deletion channel such as watermark codes [22] use codewords with a special structure designed to aid decoding. These codes cannot be directly applied to the one-way synchronization model where the sequence available at the encoder is arbitrary and will not have the desired structure in general. Our construction is based on VT codes because they can be translated to a synchronization scheme for one-deletion via the VT syndromes [1]. Designing synchronization schemes based on multiple deletion correcting channel codes is an interesting direction for future work.

### C. Contributions

The organization and the contributions of the paper are as follows.

- The construction of the two layer code and the encoding are described in Section II, and the list decoding algorithm in Section III. The performance of the list decoder is evaluated using numerical simulations in Section IV.
- In Section V, we obtain a bound on the expected list size under certain assumptions. Though not tight, the bound gives insight into how the various code parameters affect the list size.
- In Section VI-B, we analyze the complexity of encoding and decoding. The list decoder consists of multiple steps, and the complexity of each step depends on the list size at the end of the previous step. For a fixed number of edits, the encoding complexity is linear and the decoding complexity is $O(n^3)$. However, this is based on a worst-case analysis that does not consider the effect of the VT constraints in reducing the list size. Our numerical experiments indicate that the decoding complexity is typically much lower. In Section VI-D, we compare the decoding complexity with that of the Guess and Check code via a numerical example.
- In Section VII, we discuss an alternative decoder that does not require the parity-check constraints. Eliminating these constraints reduces per-symbol redundancy at the expense of a slightly larger average list size.
- In Section VIII, we extend the decoding algorithm to handle a combination of deletions and insertions. Section IX concludes the paper with a discussion of how the two-layer construction can be generalized to multiple layers.

Before we proceed, we emphasize that the code construction and its analysis throughout the paper is for the case where the number of edits $k$ is constant as $n$ grows.

*Notation*: We denote scalars by lower-case letters and sequences by capital letters. We denote the subsequence of $X$, from index $i$ to index $j$, with $i < j$ by $X(i:j) = x_i x_{i+1} \cdots x_j$. Matrices are denoted by boldfaced capitals. We use brackets for merging sequences, so $X = [X_1, \cdots, X_u]$ is a super-sequence defined by concatenating sequences $X_1, \cdots, X_u$. Logarithms with base 2 unless otherwise mentioned.

## II. Code Construction and Encoding

We begin with a brief review of VT codes. For a detailed discussion on properties of VT codes the reader is referred to [25]. The VT syndrome of a binary sequence $W = (w_1, \ldots, w_n)$ is defined as

$$\mathsf{syn}(W) = \sum_{j=1}^{n} j\, w_j \pmod{(n+1)}. \tag{1}$$

For positive integers $n$ and $0 \le s \le n$, we define the VT code of length $n$ and syndrome $s$, denoted by

$$\mathcal{VT}_s(n) = \big\{W \in \{0,1\}^n : \mathsf{syn}(W) = s\big\}, \tag{2}$$

as the set of sequences $W$ of length $n$ for which $\mathsf{syn}(W) = s$.

The $(n+1)$ sets $\mathcal{VT}_s(n) \subset \{0,1\}^n$, for $0 \le s \le n$, partition the set of all sequences of length $n$. Each of these sets $\mathcal{VT}_s(n)$ is a single-deletion correcting code. The VT encoding and decoding complexity is linear in the code length $n$ [25], [26].

### A. Constructing the Message $M$

The message $M$ generated by the encoder consists of three parts, denoted by $M_1, M_2$, and $M_3$. The first part comprises the VT syndromes of the blocks, the second part comprises the VT syndromes of the chunk-strings, and the third part is the parity-check syndrome of $X$ with respect to a linear code.

The sequence $X = x_1 x_2 \cdots x_n$ is divided into $l_1$ equal-sized blocks (assume that $n$ is divisible by $l_1$). We denote the length of each block by $n_b = \frac{n}{l_1}$. For $1 \le i \le l_1$, the $i$th block is denoted by $B_i = X((i-1)n_b + 1 : in_b)$, and its VT syndrome is $s_{B_i} = \mathsf{syn}(B_i)$. The first part of the

message is the collection of VT syndromes for the $l_1$ blocks, i.e., $M_1 = \{s_{B_1}, s_{B_2}, \cdots, s_{B_{l_1}}\}$. Since each $s_{B_i}$ is an integer between 0 and $n_b$, the number of bits required to represent the VT syndromes of the $l_1$ blocks is $l_1\lceil\log(n_b+1)\rceil$.

Each of the $l_1$ blocks is further divided into $l_2$ chunks, each of length $n_c$ bits. Since the length of each block is $n/l_1$, we have $n/l_1 = n_c l_2$, and therefore the length of $X$ satisfies $n = n_c l_1\ l_2$. (We assume that $\frac{n}{l_1}$ is divisible by $l_2$.) For $1 \le j \le l_2$, the $j$th chunk within the $i$th block is denoted by

$$C_j^i = X((i-1)n_b + (j-1)n_c + 1 : (i-1)n_b + jn_c).$$

The $j$th *chunk-string* is then formed by concatenating the $j$th chunk from each of the $l_1$ blocks. That is, the $j$th chunk string $C_j = [C_j^1, C_j^2, \cdots, C_j^{l_1}]$, for $1 \le j \le l_2$. Fig. 2 shows the blocks and the chunk-strings in an example where $X$ of length $n = 12$ is divided into $l_1 = 2$ blocks, each of which is divided into $l_2 = 2$ chunks of $n_c = 3$ bits.

The second part of the message is the collection of VT syndromes for the $l_2$ chunk-strings, i.e., $M_2 = \{s_{C_1}, s_{C_2}, \cdots, s_{C_{l_2}}\}$, where $s_{C_j}$ denotes the VT syndrome of the $j$th chunk string. Since the length of each chunk-string is $n_c l_1$, each $s_{C_j}$ is an integer between 0 and $n_c l_1$. Therefore the number of bits required to represent the VT syndromes of the $l_2$ chunk-strings is is $l_2\lceil\log(n_c l_1 + 1)\rceil$.

The final part of the message is the parity-check syndrome of $X$ with respect to a linear code. Consider a linear code of length $n$ with parity-check matrix $\boldsymbol{H} \in \{0,1\}^{z\times n}$. Then $M_3 = \boldsymbol{H}X$ is the third component of $M$. The coset of the linear code containing $X$ will be used as an erasure correcting code. In our experiments in Section IV, the linear code is chosen to be either a Reed-Solomon code over $GF(2^{n_c})$ or a bianry linear code defined by parity-check constraints drawn uniformly at random. The number of bits for $M_3$ is equal to the number of rows of $\boldsymbol{H}$, i.e., number of binary parity checks in the code, $z$. If a non-binary linear code with an $m \times n$ parity check matrix over $GF(2^{n_c})$ is used, the number of bits for $M_3$ is $z = mn_c$.

The total redundancy, or the overall number of bits required to represent the message $M = [M_1, M_2, M_3]$, is $l_1\lceil\log(n_b + 1)\rceil + l_2\lceil\log(n_c l_1 + 1)\rceil + z$.

Since $n_b = n_c l_2$, normalizing by $n = n_c l_1 l_2$ gives the *synchronization rate* (or per-symbol redundancy) $R_{\text{sync}}$ of our scheme:

$$R_{\text{sync}} = \frac{z}{n} + \frac{\lceil\log(n_c l_2 + 1)\rceil}{n_c l_2} + \frac{\lceil\log(n_c l_1 + 1)\rceil}{n_c l_1}. \quad (3)$$

*Remark:* To compute the per-symbol redundancy in (3), we assumed that each of the $(l_1 + l_2)$ VT syndromes is separately converted to a binary sequence. The binary strings are then concatenated to construct the message. This can be done more efficiently: for instance, we can list all $(n_c l_2 + 1)^{l_1}(n_c l_1 + 1)^{l_2}$ possible syndromes, and use a look-up table to map these syndromes into binary sequences. This gives the following per-symbol redundancy:

$$R_{\text{sync}} = \frac{z}{n} + \frac{\lceil\log\left((n_c l_2 + 1)^{l_1}(n_c l_1 + 1)^{l_2}\right)\rceil}{n} \quad (4)$$

$$\le \frac{z}{n} + \frac{\log(n_c l_2 + 1)}{n_c l_2} + \frac{\log(n_c l_1 + 1)}{n_c l_1} + \frac{1}{n}. \quad (5)$$

For the rest of the paper, unless specified, we use the expression in (3) for the per-symbol redundancy.

To illustrate the effect of the code parameters on the redundancy, consider the following choice for synchronizing from $k$ deletions: $l_1 = l_2 = \alpha k$, for $\alpha > 0$, so that $n_c = n/(\alpha^2\ k^2)$. Let the number of bits used to convey the parity check symbols be $z = \beta(kn_c)$, for $\beta \ge 0$. With these parameters, the per-symbol redundancy in (3) becomes

$$R_{\text{sync}} = \frac{\beta}{\alpha^2\ k} + \frac{2\alpha k\lceil\log(1 + n/(\alpha k))\rceil}{n}. \quad (6)$$

The simulation results in Section IV show that for $n$ in the range of a few hundreds to a few thousands, taking $\alpha$ close to 1 gives a good tradeoff between redundancy, synchronization performance, and decoding complexity.

Though we are primarily interested in constructing practical synchronization schemes for small and moderate values of $n$, it is interesting to examine how the redundancy scales with $n$ (with $k$ fixed). To achieve per-symbol redundancy of the optimal order $O((k\log n)/n)$, we need to choose a constant $\alpha$ and set $\beta = 0$, i.e., no parity check constraints. We discuss this setting in Section VII, where we use a guess-based decoder that allows us to achieve the order-optimal redundancy at the expense of a slightly larger list size. Thus $\beta$ can be viewed as a tuning parameter that allows us to tradeoff between list size and redundancy. In Section IX, we briefly discuss how the two-layer construction described above can be generalized to $L = \Theta(\log n)$ layers to achieve near-optimal redundancy even with the parity check constraints.

*Example 1:* Suppose that we want to design a code for synchronizing a binary sequence of length $n = 60$ from $k = 4$ deletions. Choose the chunk length $n_c = 4$, so that the sequence consists of 15 chunks. Divide the sequence into $l_1 = 5$ blocks, each comprising $l_2 = 3$ chunks. Thus there are 5 blocks each consisting of 3 adjacent chunks, and 3 chunk-strings each consisting of 5 separated chunks.

Noting that each chunk of $n_c = 4$ bits corresponds to a symbol in $GF(2^4)$, we use a Reed-Solomon code defined over $GF(2^4)$ with length $2^4 - 1 = 15$. We also choose the parity-check matrix to have 4 parity-check equations in $GF(2^4)$, so we can recover 4 erased chunks using this Reed-Solomon code.

Assume that the sequence $X$ represented in $GF(2^4)$ is

$$X = [4\ 10\ 5\ \ 0\ 3\ 14\ \ 7\ 7\ 1\ \ 0\ 2\ 4\ \ 4\ 6\ 8]^T. \quad (7)$$

Each symbol above represents a chunk of $n_c = 4$ bits. The first block [4 10 5] in binary is $B_1 = 0100\ 1010\ 0101$. The VT syndrome of this sequence is $s_{B_1} = \mathsf{syn}(B_1) = 10$. The VT syndromes of the other four blocks are $6, 3, 4$, and 11, respectively. The first part of the message is therefore $M_1 = \{10, 6, 3, 4, 11\}$.

We similarly compute $M_2$. The first chunk-string [4 0 7 0 4] in binary is

$$C_1 = 0100\ 0000\ 0111\ 0000\ 0100,$$

with VT syndrome $s_{C_1} = 11$. Computing the VT syndromes of the other chunk-strings in a similar manner, we get $M_2 = \{11, 20, 4\}$.

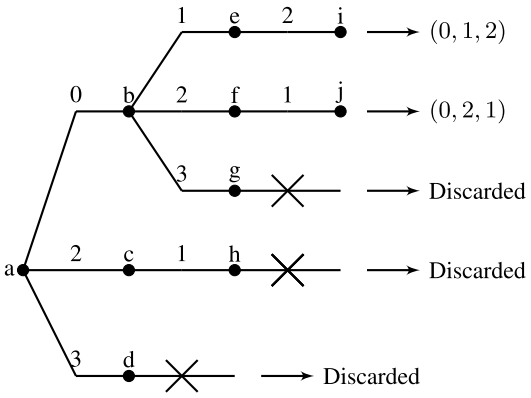


Fig. 3. Tree representing the valid block vectors for Example 2.

The final part of the message is the syndrome of $X$ with respect to the Reed-Solomon parity-check matrix. We use the following parity-check matrix $\boldsymbol{H}$ in $GF(2^4)$, with the generator 2:

$$\boldsymbol{H} = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & 2 & 4 & 8 & \cdots & 2^{14} \\ 1 & 4 & 3 & 12 & \cdots & (2^2)^{14} \\ 1 & 8 & 12 & 10 & \cdots & (2^3)^{14} \end{bmatrix}$$

to compute $M_3 = \boldsymbol{H}X = [11, 6, 13, 2]^T$. (In the representation of $GF(2^4)$ elements as degree-three polynomials with coefficients in $GF(2)$ with polynomial multiplication defined modulo $1 + x + x^4$, the generator 2 corresponds to $x$.) Since $\boldsymbol{H}$ has four rows each representing one constraint in $GF(2^4)$, $z = 16$ bits are needed to represent the parity-check syndrome in binary. The total number of bits to convey the message is $5\lceil\log(13)\rceil + 3\lceil\log(21)\rceil + 16 = 51$ bits.

## III. Decoding Algorithm

The goal of the decoder is to recover $X$ given $Y$, $n$ and the message $M = [M_1, M_2, M_3]$. From $M_1, M_2$, the decoder knows the VT syndrome of each block and each chunk-string. Using this, the decoder first finds all possible configurations of deletions across blocks, and then for each of these configurations, it finds all possible chunk deletion patterns. Since each chunk is the intersection of a block and a chunk-string, each chunk plays a role in determining exactly two VT syndromes. The intersecting construction of blocks and chunk-strings enables the decoder to iteratively recover the deletions in a large number of cases. The decoder is then able to localize the positions of the remaining deletions to within a few chunks. These chunks are considered erased, and are finally recovered by the erasure-correcting code.

The decoding algorithm consists of six steps, as described below.

### *Step 1: Block Boundaries*

In the first step, the decoder produces a list of candidate *block-deletion patterns* $V = (a_1, \cdots, a_{l_1})$ compatible with $Y$, where $a_i$ is the number of deletions in the $i$th block. Each pattern in the list should satisfy $\sum_{i=1}^{l_1} a_i = k$ with $0 \leq a_i \leq k$. The list of candidates always includes the true block-deletion pattern. It is convenient to represent the candidate block-deletion patterns as branches on a tree with $l_1$ levels, as shown in Fig. 3. At every level (block) $i = 1, \ldots, l_1$, branches are added and labeled with all possible values of $a_i$. Specifically, the tree is constructed as follows.

*Level 1 of the tree*: Consider the first $n_b$ received bits $Y(1 : n_b)$, compute its VT syndrome $u = \mathsf{syn}(Y(1 : n_b))$ and compare it with $s_{B_1}$, the correct syndrome of the first block. There are two alternatives for the $k$ branches of the first level.

1) $u = s_{B_1}$: First, the decoder adds a branch with $a_1 = 0$, corresponding to the case that the first $n_b$ bits are deletion-free. The first block cannot have just one deletion, because in this case the single-deletion correcting property of the VT code would imply that $u \neq s_{B_1}$. However, it is possible that two or more than two deletions happened in block one, and by considering additional bits from the next block, the VT-syndrome of first $n_b$ bits accidentally matches with $s_{B_1}$. For example, consider blocks of length $n_b = 4$, and let the first two blocks of $X$ be 0100 1111 ..., with the underlined bits deleted we get $Y = 001111 \ldots$. In this case $u = s_{B_1} = 2$. The decoder thus adds a branch for $a_1 = 0, 2, \ldots, k$.
2) $u \neq s_{B_1}$: Block one contains one or more deletions and the decoder adds a branch for $a_1 = 1, 2, \ldots, k$.

*Level $i + 1$, $1 \leq i < l_1$*: Assume that we have constructed the tree up to level $i$. Consider a branch of the tree at level $i$ with the number of deletions in blocks 1 through $i$ given by $a_1, a_2, \cdots, a_i$, respectively. This gives us the starting position of block $(i + 1)$ in $Y$. Denote this starting position by

$$p_{i+1} = n_b i - d_i + 1. \tag{8}$$

where $d_i = \sum_{j=1}^{i} a_j$ is the number of deletions on the branch up to block $i$. Compute the VT syndrome of next $n_b$ bits $u = \mathsf{syn}\left(Y\left(p_{i+1} : p_{i+1} + n_b - 1\right)\right)$. There are two alternatives:

1) $u = s_{B_{i+1}}$: If $(k - d_i) < 2$ then the only possibility is that $a_{i+1} = 0$. If $(k - d_i) \geq 2$, $k - d_i - 1$ branches are added for $a_{i+1} = 0, 2, \ldots, k - d_i$.
2) $u \neq s_{B_{i+1}}$: If $(k - d_i) > 0$ then there are $(k - d_i)$ possibilities at this branch: the $i$th block can have $1, 2, \cdots, (k - d_i)$ deletions. If $(k - d_i) = 0$, it is assumed this is an invalid branch, and the path is discarded.

*Example 2:* Assume $k = 3$ deletions, $l_1 = 3$ blocks, and that the true deletion pattern is (0,2,1), i.e., there are zero deletions in the first block, two deletions in second block, and one deletion in third block. The tree constructed by the decoder depends on the underlying sequences $X$ and $Y$. In Fig. 3, we illustrate one possible tree constructed for this scenario without explicitly specifying $X$ and $Y$.

Assume that in the first step, the syndrome matches with $s_{B_1}$, so we have $a_1 = 0, 2$, or 3. At node b (corresponding to $a_1 = 0$), suppose that the syndrome does not match with $s_{B_2}$, so we have $a_2 = 1, 2$, or 3. Now suppose that at nodes c and d, the syndrome does not match with $s_{B_2}$. At node d, $a_1 = 3$, so there are no more deletions available for the second block; so this branch is discarded. At node c, $a_1 = 2$, so the only possibility is one deletion in the second block. Then if

the syndrome at node h does not match $s_{B_3}$, the branch is discarded. At nodes e and f, we assign the remaining deletions to the last block. At node g, the syndrome does not match with $a_3$, and the branch is discarded.

*Step 2: Primary Fixing of Blocks*

Denote by $\mathcal{L}_1$ the list of the block-deletion pattern candidates after the first step and denote the corresponding block-deletion patterns by $V_1, \cdots, V_{|\mathcal{L}_1|}$. In this second step, for each of the block-deletion patterns, we restore the deleted bit in blocks containing a single deletion by using the VT decoder. Specifically, for a block-deletion pattern $V = (a_1, \cdots, a_{l_1})$, let the $i$th block of $Y$ with respect to $V$ be $S = Y\,(p_i : p_i + n_b - 1)$ where $p_i$ is the starting position of the $i$th block in $Y$, defined analogously to (8). If $a_i = 1$, feed the sequence $S$ to the VT decoder and in $Y$, replace $S$ with the decoded sequence. After this, the $i$th block in $Y$ is deletion free, thus, the decoder updates the block-deletion pattern $V$ by setting $a_i = 0$. We carry out this procedure for all blocks with one deletion in $V$. This results in a sequence $\hat{Y}$, which is obtained from $Y$ by recovering the single-deletion blocks corresponding to block-deletion pattern $V$. Denote the updated version of block-deletion pattern $V$ by $\hat{V}$. Thus at the end of this step, we have $|\mathcal{L}_1|$ updated candidate sequences $\hat{Y}_1, \cdots, \hat{Y}_{|\mathcal{L}_1|}$ with corresponding block-deletion patterns $\hat{V}_1, \cdots, \hat{V}_{|\mathcal{L}_1|}$.

*Example 3:* Consider the code of Example 1 with $l_1 = 5$ blocks, and $k = 4$ deleted bits. If the list of block-deletion patterns at the end of the first step is

$$V_1 = (1,1,1,1,0),\ \ V_2 = (1,1,2,0,0),$$
$$V_3 = (1,2,1,0,0),\ \ V_4 = (2,0,2,0,0),$$

then the updated list of block-deletion patterns is

$$\hat{V}_1 = (0,0,0,0,0),\ \ \hat{V}_2 = (0,0,2,0,0),$$
$$\hat{V}_3 = (0,2,0,0,0),\ \ \hat{V}_4 = (2,0,2,0,0).$$

*Step 3: Chunk Boundaries*

In this step, for each updated block-deletion pattern $\hat{V}$ and the corresponding $\hat{Y}$, we list all possible allocations of deletions across chunks. More precisely, for each pair $(\hat{V}, \hat{Y})$ we list all possible $l_1 \times l_2$ matrices $\boldsymbol{A} = (a_{ij})$, where $a_{ij}$ is the number of deletions in the $j$th chunk of the $i$th block, such that $\sum_{j=1}^{l_2} a_{ij} = a_i$, the $i$th entry of $\hat{V}$. The $j$th column of matrix $\boldsymbol{A}$, specifies the number of deletions in the $l_1$ chunks of the $j$th chunk-string. For example, some of the possible matrices for $\hat{V}_4 = (2,0,2,0,0)$ in Example 3 are

$$\boldsymbol{A}_1 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},\ \boldsymbol{A}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix},\ \boldsymbol{A}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{9}$$

The algorithm that lists all chunk-deletion matrices $\boldsymbol{A}$ compatible with a given block-deletion pattern $\hat{V} = (a_1, \ldots, a_{l_1})$ is very similar to the tree construction described in Step 1. In this case, for each block-deletion pattern $\hat{V}$, another tree will be constructed, with each path in the tree representing a valid chunk-deletion matrix $\boldsymbol{A}$.

*Level* 1 *of the tree*: Construct a sequence $S$ by concatenating the first $n_c$ bits of each block in $\hat{Y}$ and compute its VT syndrome $u = \mathsf{syn}(S)$. There are two possibilities:

1) $\underline{u = s_{C_1}}$: For the first chunk-string, list all valid chunk-deletion patterns of the form $(a_{11}, \ldots, a_{l_1 1})$, where $0 \le a_{i1} \le a_i$, and $\sum_{i=1}^{l_1} a_{i1} \ne 1$, since a single deletion in the chunk-string would result in $u \ne s_{C_1}$.
2) $\underline{u \ne s_{C_1}}$: List all valid chunk-vectors for the first chunk-string of the form $(a_{11}, \ldots, a_{l_1 1})$, where $0 \le a_{i1} \le a_i$, and $\sum_{i=1}^{l_1} a_{i1} \ge 1$.

*Level* $j$, $1 < j \le l_2$: Assume that we have constructed the tree up to level $(j-1)$. Thus, we know the number of deletions in each chunk of the first $(j-1)$ chunk-strings. From this, we can determine the total number of deletions in the first $(j-1)$ chunks of each block. Let $d_{i,j-1}$ denote the number of deletions in the first $(j-1)$ chunks of block $i$. Then along this path, the $j$th chunk of $i$th block in $\hat{Y}$ is

$$S_{ij} = \hat{Y}\Big(p_i + (j-1)n_c - d_{i,j-1} : p_i + jn_c - d_{i,j-1} - 1\Big). \tag{10}$$

Form the $j$th chunk-string, $S_j = [S_{1j}, \cdots, S_{l_1 j}]$, compute its VT syndrome $u = \mathsf{syn}(S_j)$, and compare it with the correct syndrome $s_{C_j}$. There are two possibilities.

1) $\underline{u = s_{C_j}}$: List all valid chunk-deletion patterns for the $j$th chunk-string of the form $(a_{1j}, \ldots, a_{l_1 j})$, where $0 \le a_{ij} \le a_i - d_{i,j-1}$, and $\sum_{i=1}^{l_2} a_{ij} \ne 1$.
2) $\underline{u \ne s_{C_j}}$: List all valid chunk-deletion patterns for the $j$th chunk-string of the form $(a_{1j}, \ldots, a_{l_1 j})$, where $0 \le a_{ij} \le a_i - d_{i,j-1}$, and $\sum_{i=1}^{l_2} a_{ij} \ge 1$. If the list is empty, discard the branch. The list will be empty when there are no more deletions to assign to $j$th chunk-string.

At the end of step 3, the decoder provides a list of pairs $(\hat{Y}, \boldsymbol{A})$, where $\hat{Y}$ is a candidate sequence to be decoded using the chunk-deletion matrix $\boldsymbol{A}$, with $a_{ij}$ being the number of deletions in the $j$th chunk of the $i$th block. Denote the number of such pairs in the list by $|\mathcal{L}_3|$.

*Step 4: Iterative Correction of Blocks and Chunk-Strings*

Similar to step 2, in step 4 we use the VT syndromes (known from the message sent by the encoder) to recover deletions in blocks and chunk-strings for which the matrix $\boldsymbol{A}$ indicates a single deletion. Whenever a deletion recovered using a VT decoder lies in a chunk different from the one indicated by $\boldsymbol{A}$, the candidate is discarded. As discussed in Section IV and VI, this is an effective way of discarding several invalid candidates. The iterative algorithm is described below. For each pair $(\hat{Y}, \boldsymbol{A})$:

i) For each column of $\boldsymbol{A}$ containing a single 1 (indicating a single deletion in the corresponding chunk-string), recover the deleted bit in the chunk-string using its VT syndrome. With some abuse of notation we still refer to the restored sequence as $\hat{Y}$. If the restored bit does not lie in the expected chunk indicated by the 1,

discard the pair $(\hat{Y}, \boldsymbol{A})$ and move to the next candidate pair. Otherwise, update the matrix $\boldsymbol{A}$ by replacing the 1s corresponding to the restored chunks by 0s. If there is a row in the updated matrix $\boldsymbol{A}$ with a single 1, proceed to step 4.ii).

ii) For each row of $\boldsymbol{A}$ containing a single 1 (indicating a single deletion in the corresponding block), recover the deleted bit in the block using its VT syndrome. Again, with some abuse of notation we still refer to the restored sequence as $\hat{Y}$. If the restored bit does not lie in the expected chunk indicated by the 1, discard the pair $(\hat{Y}, \boldsymbol{A})$ and move to the next pair. Otherwise, update the chunk-deletion matrix by replacing the 1s corresponding to the restored chunks to 0s. If there is a column in the updated matrix $\boldsymbol{A}$ with a single 1, go to step 4.i).

Denote the updated candidate pairs at the end of this procedure by $(\tilde{Y}, \tilde{\boldsymbol{A}})$, and assume there are $|\mathcal{L}_4|$ of them.

As an illustrative example, consider the three chunk-matrices given in (9). In $\boldsymbol{A}_1$, we can successfully recover all the deletions. In $\boldsymbol{A}_2$, we can only fix two deletions in the third block. However, for $\boldsymbol{A}_3$, we cannot recover any of the deletions. Thus, the updated $\tilde{\boldsymbol{A}}$ matrices are

$$\tilde{\boldsymbol{A}}_1 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \tilde{\boldsymbol{A}}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \ \tilde{\boldsymbol{A}}_3 = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}. \tag{11}$$

In Section VII, we discuss a method to recover remaining deletions using VT constraints and bypassing the fifth step (where we use linear codes).

*Step 5: Replacing Deletions With Erasures*

In this step, for each of the $|\mathcal{L}_4|$ surviving pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$, we replace each chunk of $\tilde{Y}$ that still contains deletions with $n_c$ erasures. Hence, if there are $\nu$ chunks with deletions (where $1 \leq \nu \leq k$), the resulting sequence will have length $n$, with $n_c\nu$ erasures and no deletions. Notice that this operation of replacing with erasures can be performed without ambiguity since $\tilde{\boldsymbol{A}}$ precisely indicates the starting position of each chunk and also the number of deletions within that chunk.

The purpose of the linear code is to recover the erased bits. The minimum distance of the linear code should be large enough to guarantee that we can resolve all the $\nu n_c$ erased bits. In Example 1, as there are four deletions, we will have at most $\nu = 4$ erased chunks, so we choose a Reed-Solomon code with 4 parity-check equations in $GF(2^4)$. The chunk-matrix $\tilde{\boldsymbol{A}}_3$ in (11) shows that a smaller number of parity-check symbols will not suffice if we want to correct all deletion patterns.

Some invalid candidates may be discarded in the process of correcting the erasures as we may find that the parity-check equations are inconsistent, i.e. there is no solution for the erased chunks. We denote the number of remaining candidates at the end of this step by $|\mathcal{L}_5|$.

TABLE I

NUMBER OF DELETIONS $k$, CODE LENGTH $n$, AND CODE PARAMETERS FOR EACH SETUP

| | $k$ | $n$ | $l_1$ | $l_2$ | $n_c$ | $z$ | $R_{\text{sync}}$ |
|---|---|---|---|---|---|---|---|
| Setup 1 | 3 | 60 | 5 | 3 | 4 | 4 | 0.650 |
| Setup 2 | 3 | 60 | 5 | 3 | 4 | 8 | 0.717 |
| Setup 3 | 3 | 60 | 5 | 3 | 4 | 12 | 0.783 |
| Setup 4 | 4 | 60 | 5 | 3 | 4 | 16 | 0.850 |
| Setup 5 | 7 | 378 | 9 | 7 | 6 | 42 | 0.365 |
| Setup 6 | 7 | 486 | 9 | 9 | 6 | 50* | 0.325 |
| Setup 7 | 9 | 1080 | 15 | 12 | 6 | 55* | 0.225 |
| Setup 8 | 10 | 2800 | 20 | 20 | 7 | 60* | 0.135 |

*Step 6: Discarding Invalid/Identical Candidates*

The reconstructed sequences at the end of Step 5, denoted by $\hat{X}$, all have length $n$ and are deletion free. For each of the $|\mathcal{L}_5|$ sequences $\hat{X}$, we check the VT and parity-check constraints for each of the block and chunk-strings and discard those not meeting any of the constraints. At the end of Step 5 it is possible to have multiple copies of the same sequence. This is due to a deletion occurring in a run that intersects two chunks (or more); this deletion can be interpreted as a deletion in either chunk, and each interpretation leads to seemingly different candidates which will turn out to be the same at the end of the process. The surviving $|\mathcal{L}_6|$ distinct sequences comprise the final list produced by the decoder.

The final list of reconstructed sequences consist of all length-$n$ sequences that can be obtained by adding $k$ bits to $Y$ and also satisfy all the VT and parity-check constraints. The correct sequence is always among the $|\mathcal{L}_6|$ candidates. The synchronization algorithm is said to be zero-error if and only if $|\mathcal{L}_6| = 1$ for all sequences and deletion patterns. When $|\mathcal{L}_6| > 1$, the list size can be further reduced if additional hash functions or cyclic redundancy checks are available from the encoder.

## IV. NUMERICAL EXPERIMENTS

In this section, we present numerical results illustrating the performance of the synchronization code for various choices of the system parameters. The different setups that were simulated are shown in Table I. For each setup, the performance was recorded over $10^6$ trials. In each trial, the sequence $X$ and the locations of the $k$ deletions were chosen independently and uniformly at random. For the first five setups, we used parity-check constraints from a Reed-Solomon code over $GF(2^{n_c})$ with code length $(2^{n_c} - 1)$. For example, in setup 5 we used 7 parity-check constraints from a Reed-Solomon code over $GF(2^6)$, hence $z = 42$ bits are needed to represent the parity-check syndrome. In the last three setups, where the $z$ entry is denoted with an asterisk, we used $z$ binary parity-check constraints (for a length $n$ seqeunce) drawn uniformly at random.

Table II shows the list sizes of the number of candidates at the end of various steps of the decoding process. Recall that $|\mathcal{L}_1|$ is the number of candidate block-deletion patterns at the end of step 1, $|\mathcal{L}_3|$ is the number of pairs $(\hat{Y}, \boldsymbol{A})$ at the end of step 3, $|\mathcal{L}_4|$ is the number of pairs $(\tilde{Y}, \tilde{\boldsymbol{A}})$ at the end of

TABLE II
LIST SIZE AFTER EACH STEP, AVERAGED OVER $10^6$ TRIALS

| | $\mathbb{E}\|\mathcal{L}_1\|$ | $\mathbb{E}\|\mathcal{L}_3\|$ | $\mathbb{E}\|\mathcal{L}_4\|$ | $\mathbb{E}\|\mathcal{L}_6\|$ | $\max\|\mathcal{L}_6\|$ | $\mathbb{P}[\|\mathcal{L}_6\|>1]$ |
|---|---|---|---|---|---|---|
| Setup 1 | 1.87 | 1.92 | 1.42 | 1.003 | 3 | 0.003 |
| Setup 2 | 1.87 | 1.92 | 1.42 | 1.000 | 2 | $2.5\times10^{-5}$ |
| Setup 3 | 1.87 | 1.92 | 1.42 | 1 | 1 | 0 |
| Setup 4 | 3.39 | 6.18 | 2.53 | 1 | 1 | 0 |
| Setup 5 | 11.51 | 74.43 | 3.42 | 1 | 1 | 0 |
| Setup 6 | 11.20 | 28.64 | 2.55 | 1 | 1 | 0 |
| Setup 7 | 14.45 | 94.38 | 2.41 | 1 | 1 | 0 |
| Setup 8 | 12.76 | 26.16 | 1.57 | 1 | 1 | 0 |

step 4, and $|\mathcal{L}_6|$ is the number of sequences $\hat{X}$ in the final list. The average of $|\mathcal{L}_i|$ over the $10^6$ trials is denoted by $\mathbb{E}|\mathcal{L}_i|$. The column $\max|\mathcal{L}_6|$ shows the maximum size of the final list across the $10^6$ trials. The column $\mathbb{P}[|\mathcal{L}_6|>1]$ shows the fraction of trials for which $|\mathcal{L}_6|>1$.

The first three setups have identical parameters, except for the number of Reed-Solomon parity checks. This shows the effect of adding parity-check constraints on the list size and the redundancy. Adding more parity-check constraints improves the decoder performance by reducing the number of trials with list size greater than one, at the expense of increased redundancy.

The fourth setup is precisely the code described in Example 1. It has the same values of $(n_c, l_1, l_2)$ as the first three setups but with a larger number of deletions and parity-check constraints. We observe that increasing the number of deletions (with $n_c, l_1, l_2$ unchanged) increases the average number of candidates in the different decoding steps. In general, choosing $l_1 \geq k$ ensures that the average list size after step 1 is small.

The fifth setup is a larger code with length $n = 378$ and can handle a larger number of deletions ($k = 7$). Though the final list size is always one, the number of candidate chunk-deletion matrices at the end of the third step is large, which increases the decoding complexity. The only difference between setups five and six is that the latter has a larger value of $l_2$. Comparing $\mathbb{E}|\mathcal{L}_3|$ for these setups, we observe that increasing $l_2$ significantly reduces the number of candidate chunk-deletion matrices at the end of the third step. This is because increasing $l_2$ increases the number of chunk-string VT constraints, which allows the decoder to eliminate more candidates while determining chunk boundaries.

The last setup is a relatively long code. Although the average number of candidates in each of the decoding steps is not very high, we found that a small fraction of trials have a very large number of candidates, resulting in considerably slower decoding for these trials.

## V. LIST SIZE ANALYSIS

The final list produced by the decoder consists of all sequences that satisfy the $l_1$ block VT constraints, the $l_2$ chunk-string VT constraints, and the parity-check constraints. Recall that at the end of step 3 of decoding we have a set of candidate chunk deletion patterns, each of which is of the form $\{a_{ij}\}_{1\leq i\leq l_1,\, 1\leq j\leq l_2}$, where $a_{ij}$ is the number of deletions in chunk $j$ within block $i$. A number of candidate patterns are then discarded in Step 4 as they fail to satisfy the intersecting VT constraints.

As evident from Table II, the VT constraints play a key role in reducing the list size. However, the non-linearity of the VT constraints and the intersecting construction makes it challenging to obtain theoretical bounds on the list size. We will therefore bound the expected list size by considering only the effect of the parity-check constraints. Though the bound is loose, it gives us insight into how the code parameters affect the list size.

In step 5 of decoding, the parity-check constraints are used to recover the unresolved chunks for each of the surviving chunk deletion patterns at the end of step 4. Since there are a total of $k$ deletions, we consider all chunk-deletion patterns $\{a_{ij}\}_{1\leq i\leq l_1,\, 1\leq j\leq l_2}$ that satisfy

$$\sum_{i=1}^{l_1}\sum_{j=1}^{l_2} a_{ij} = k, \quad a_{ij}\geq 0. \tag{12}$$

Furthermore, assume that any pattern of upto $k$ erased chunks can be recovered using the $z$ binary parity-check constraints. This can be ensured by using $z = kn_c$ linearly independent parity-check equations from a binary linear code with minimum distance at least $kn_c + 1$. (For example, we can use $k$ parity-check constraints of an $(n-k, n)$ MDS code over GF($2^{n_c}$).) This implies that the parity-check constraints can be used to recover any pattern of up to $k$ erased chunks. For each chunk-deletion pattern considered, the bits in the unresolved chunks are erased (according to the pattern), and the parity-check constraints are used to recover these erased chunks. Note that the recovered bits should be a supersequence of the bits erased in the unresolved chunks, otherwise the decoder can discard the deletion pattern.

We will bound the the probability that an incorrect deletion pattern satisfying (12) satisfies all the parity-check constraints and is a supersequence of the erased bits. Since there are $\binom{k+l_1l_2-1}{k}$ deletion patterns satisfying (12), this will give a bound on the expected list size. We make two assumptions on any sequence reconstructed using an incorrect chunk deletion pattern. To motivate these assumptions, consider the following example.

*Example 4:* Assume that $n_c = 3$ and $l_1 = l_2 = 2$, and that are $k = 3$ deletions. Let $X = \underline{1}01\ 100\ 011\ 1\underline{00}$ and $Y = 011000111$, with the underlined bits being deleted from $X$ to produce $Y$. The correct chunk deletion pattern is $(1, 0, 0, 2)$. According to this pattern, the erased sequence $Y' = xxx\ 100\ 011\ xxx$ where $x$ denotes an erased bit (from which $X$ is recovered using the parity-check constraints.

Consider an incorrect deletion pattern, say $(2, 0, 0, 1)$. The erased sequence according to this pattern is $Y'' = xxx\ 110\ 001\ xxx$. Note that for recovered sequence based on this deletion pattern, the decoder requires that the first chunk contains a 0, and the last chunk contains 11. We will assume that the bits recovered in first and fourth chunks of $Y''$ (using the parity-check constraints) are uniformly random and hence independent of the erased bits (0 in the first chunk and 11 in the fourth chunk).

*Assumption 1:* If the set of parity-check constraints has a solution for the erased chunks corresponding to an incorrect deletion pattern, then the recovered bits will be uniformly random and independent of the bits erased from the chunks.

Given an incorrect deletion pattern which corresponds to $k' < k$ erased chunks, the chunks can be recovered using any $k'$ parity-check constraints of the MDS code.

*Assumption 2:* Given an incorrect deletion pattern for which the erased chunks are recovered using $z' < z$ binary parity-check constraints then the recovered sequence satisfies each of the remaining $(z - z')$ constraints independently with probability $\frac{1}{2}$. Therefore the probability that the recovered sequence satisfies all the remaining parity-check constraints is $\left(\frac{1}{2}\right)^{z-z'}$.

Assumption 2 is motivated by the observation that when for an incorrect chunk deletion pattern, the bits in the $i$th unerased chunk in the sequence do not represent the actual $i$th chunk of the sequence. Furthermore, using Assumption 1, the recovered bits in the erased chunks are uniformly random. Hence evaluating a new parity-check constraint on the recovered sequence is equally likely to result in 0 or 1.

*Proposition 1:* Assume that the binary string $X$ and the locations of the $k$ deletions to produce $Y$ are both chosen uniformly at random. Let the $z \geq kn_c$ linearly independent binary parity-check constraints be chosen from a linear code with minimum distance at least $kn_c + 1$ bits. Then under Assumptions 1 and 2, the probability that the final list size exceeds 1 satisfies

$$\mathbb{P}[|\mathcal{L}_6| > 1] \leq \left(e\left(1 + \frac{l_1\ l_2}{k}\right)\left(\frac{n_c+1}{2^{n_c}}\right)\right)^k. \tag{13}$$

The expected size of the final list satisfies

$$\mathbb{E}|\mathcal{L}_6| \leq 1 + \left(e\left(1 + \frac{l_1\ l_2}{k}\right)\left(\frac{n_c+1}{2^{n_c}}\right)\right)^k. \tag{14}$$

*Proof:* Consider an incorrect deletion pattern $(a_{11}, \cdots, a_{l_1 l_2})$, where the deletions are in $k' \leq k$ chunks. Let $a_{ij} > 0$ in this pattern. Using Assumption 1, the probability that the $n_c$ bits recovered in this chunk (using the parity-check constraints) are a supersequence of the $(n_c - a_{ij})$ bits erased in this chunk is:

$$\frac{\sum_{m=0}^{a_{ij}} \binom{n_c}{m}}{2^{n_c}} \leq \frac{(n_c+1)^{a_{ij}}}{2^{n_c}}. \tag{15}$$

The numerator in the LHS is the number of supersequences of length $n_c$ for the erased chunk (see [27] for a proof), and the denominator is the total number of length $n_c$ binary sequences. Therefore, the probability that the recovered sequence is a supersequence of the erased bits in all the chunks with deletions can be bounded by

$$\prod_{i,j:a_{ij}\geq 1} \left(\frac{(n_c+1)^{a_{ij}}}{2^{n_c}}\right) = \frac{(n_c+1)^k}{2^{k'n_c}}, \tag{16}$$

where we have used $\Sigma_{i,j} a_{i,j} = k$ and the fact that the deletion pattern has $k'$ chunks with deletions, i.e., $k'$ pairs $(i, j)$ with $a_{i,j} > 0$.

Since $k'$ chunks are erased, $k'n_c$ linearly independent parity constraints suffice to recover the $k'n_c$ bits in these chunks. Furthermore, using Assumption 2, the probability that the recovered sequence satisfies the $(z - k'n_c)$ remaining parity-check equations is $\left(\frac{1}{2}\right)^{z-k'n_c} \leq \left(\frac{1}{2}\right)^{kn_c - k'n_c}$. Combining this with (16), we have the following upper bound on the probability that the sequence recovered from the incorrect chunk deletion pattern is in the final list:

$$\left(\frac{1}{2}\right)^{kn_c - k'n_c} \frac{(n_c+1)^k}{2^{k'n_c}} = \left(\frac{n_c+1}{2^{n_c}}\right)^k. \tag{17}$$

Now using union bound for each of the possible chunk deletion patterns, the probability that the final list size exceeds 1 satisfies

$$\mathbb{P}[|\mathcal{L}_6| > 1] \leq \binom{k + l_1 l_2 - 1}{k}\left(\frac{n_c+1}{2^{n_c}}\right)^k \tag{18}$$

$$= \left(e\left(1 + \frac{l_1 l_2}{k}\right)\left(\frac{n_c+1}{2^{n_c}}\right)\right)^k, \tag{19}$$

We can also use the upper bound in (17) for each of the possible chunk deletion patterns. Noting that the correct pattern will be on the list with probability 1, we have

$$\mathbb{E}|\mathcal{L}_6| \leq 1 + \binom{k + l_1 l_2 - 1}{k}\left(\frac{n_c+1}{2^{n_c}}\right)^k \tag{20}$$

$$\leq 1 + \left(\frac{e(k + l_1 l_2)}{k}\right)^k \left(\frac{n_c+1}{2^{n_c}}\right)^k. \tag{21}$$

□

According to Proposition 1, we will have $\mathbb{E}|\mathcal{L}_6| < 2$ if the parameters are chosen such that

$$n_c \log 2 - \log(n_c + 1) > \log e + \log(1 + l_1 l_2/k). \tag{22}$$

For example, we can choose $l_1 = l_2 = k$, and $n_c > \log(3(1+k))$ to ensure that $\mathbb{E}|\mathcal{L}_6| < 2$ as the number of deletions $k$ grows. This choice is similar to the parameters used for the numerical simulations in Section IV.

Since we have not taken into account the effect of the VT constraints, the bound in (14) is loose. Indeed, the bound suggests that increasing $l_1, l_2$ will increase the final list size. However, we will see in the next section that increasing $l_1, l_2$ reduces the average list size in the intermediate steps of the decoder. This is because increasing the number of VT constraints allows the decoder to reject a larger of number of incorrect deletion patterns as being inconsistent with the VT constraints. An interesting direction for future work is to tighten the bound of Proposition 1 by taking into account the effect of the VT constraints.

## VI. Encoding and Decoding Complexity

In this section we discuss the number of operations required for constructing the encoded message $M = [M_1, M_2, M_3]$ and for the decoding algorithm.

### A. Encoding Complexity

Computing the VT syndrome of a length $m$ sequence needs $O(m)$ arithmetic operations. For $M_1$ we need to compute VT syndrome of $l_1$ blocks, each of length $n_b = n_c l_2$. This can be done with $O(n_c l_1 l_2)$ operations. Similarly, for $M_2$ we need $O(n_c l_2\ l_1)$ operations. Recalling that $n = n_c l_1 l_2$, the complexity of computing $M_1$ and $M_2$ is $O(n)$. Recall that $M_3 = \boldsymbol{H}X$ is constructed via multiplication of a $z \times n$ matrix with a length $n$ vector, which requires $O(zn)$ operations. This is the dominant term in the encoding complexity, therefore, the overall complexity of the encoder is $O(zn)$.

### B. Decoding Complexity

In the following, we analyze the decoding complexity by finding an upper bound for the complexity of each of the six decoding steps.

#### *Step 1: Block Boundaries*

In the first step, we construct a tree for finding all the candidates for block boundaries. At each node of the tree, a VT syndrome of a length $n_b$ sequence is computed and compared with the syndrome known from $M_1$. Since there are a total of $l_1$ blocks, the maximum number of valid block deletion patterns at the end of Step 1 is the number of non-negative integer solutions of

$$a_1 + \cdots + a_{l_1} = k, \tag{23}$$

which is $\binom{k+l_1-1}{k}$. This number is only an upper bound on the number of valid block deletion patterns as we do not take into account the effect of the VT syndrome in discarding patterns. (Recall that for a block deletion pattern to be valid, all the blocks with zero deletions in the pattern should be consistent with their VT syndromes.) Each valid block deletion pattern is a leaf of the tree. Since there are $l_1$ levels, the total number of nodes in the tree is at most $l_1\binom{k+l_1-1}{k}$. (Note from Fig. 3 that not every block deletion pattern corresponds to a branch with $l_1$ nodes.)

Therefore, an upper bound for the number of required operations in step 1 is:

$$\binom{k+l_1-1}{k} \times l_1 \times O(n_b) = O\left(n\binom{k+l_1-1}{k}\right). \tag{24}$$

As explained in the Section III, many of the branches of the tree will get discarded because of the block deletion pattern being inconsistent with the VT constraints. The average number of nodes at the final level of the tree (denoted by $\mathbb{E}|\mathcal{L}_1|$) is particularly important since it will determine the average complexity of the next steps of the decoding. In Table III, we compare the empirical value of $\mathbb{E}|\mathcal{L}_1|$ (from Table II) with $\binom{k+l_1-1}{k}$, the upper bound for $|\mathcal{L}_1|$ obtained from (23). The considerable difference between these two numbers shows the importance of using VT codes — in addition to recovering single deletions, they act as hashes and allow the decoder to discard a larger number of incorrect block deletion patterns. This significantly decreases the decoding complexity by reducing the number of candidates that need to be considered in subsequent steps. This lower complexity allows us to efficiently decode relatively long codes like the code in setup 7 of Table I.

TABLE III

COMPARISON OF AVERAGE NUMBER OF SURVIVING PATHS AFTER STEP 1

| | Setup 1 to 3 | Setup 4 | Setup 5 | Setup 6 | Setup 7 |
|---|---|---|---|---|---|
| $\mathbb{E}\|\mathcal{L}_1\|$ | 1.87 | 3.39 | 11.51 | 11.20 | 12.76 |
| $\binom{k+l_1-1}{k}$ | 35 | 70 | $6.4 \times 10^3$ | $6.4 \times 10^3$ | $2.0 \times 10^7$ |

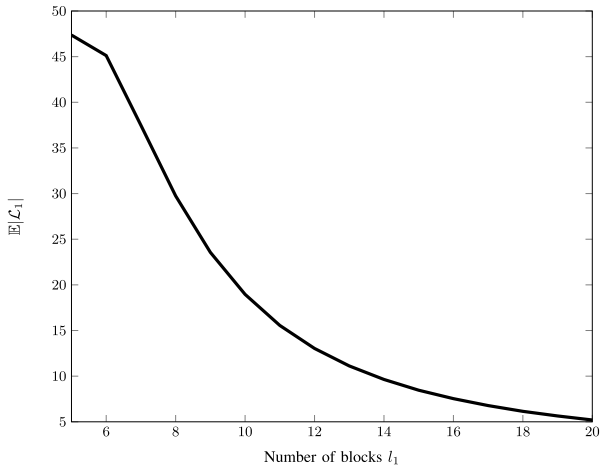


Fig. 4. Empirical average $\mathbb{E}|\mathcal{L}_1|$ for different values of $l_1$ when $k = 8$, and block length $n_b = 42$.

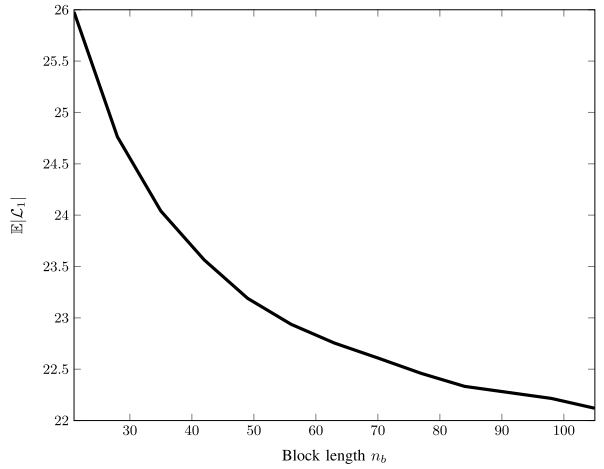


Fig. 5. Empirical average $\mathbb{E}|\mathcal{L}_1|$ for different values of $n_b$ when $k = 8$, and the number of blocks $l_1 = 9$.

In Figures 4 and 5, we show how the empirical average $\mathbb{E}|\mathcal{L}_1|$ changes with the parameters of the code. The list size $|\mathcal{L}_1|$ depends on the number of deletions $k$, the number of block constraints $l_1$ and the number of bits per block $n_b$. In Figure 4, where $k$ and $n_b$ are fixed, we see that the empirical average $\mathbb{E}|\mathcal{L}_1|$ decreases with $l_1$ (although $\binom{k+l_1-1}{k}$, the upper bound on $|\mathcal{L}_1|$ increases). This is because a given solution of equation (23) will not be in the list $\mathcal{L}_1$ when it is not consistent with a block VT constraint. In particular, if $a_i = 0$, i.e., the block is considered deletion free according to the deletion pattern, the VT syndrome of the sequence corresponding to

$i$th block should match with the correct syndrome known from $M_1$.

Figure 5 shows that $\mathbb{E}|\mathcal{L}_1|$ also decreases with $n_b$ when $k$ and $l_1$ are fixed. This is because the probability that the VT syndrome of a length $n_b$ sequence accidentally matches the correct block VT syndrome decreases with $n_b$. (Recall that that the VT syndrome is a number between 0 and $n_b$.) Such accidental matches, if not detected in a subsequent level of the tree, will increase the number of incorrect deletion patterns in $\mathcal{L}_1$.

*Step 2: Primary Fixing of Blocks*

In the second step, we use block VT syndromes to recover deletions in blocks with a single deletion. There are at most $l_1$ such blocks. Since the VT decoding complexity is linear in $n_b$ (the length of each block), the complexity for the second step is

$$|\mathcal{L}_1| \times l_1 \times O(n_b) = O\left(n|\mathcal{L}_1|\right). \tag{25}$$

*Step 3: Chunk Boundaries*

In this step, we find all possibilities for the number of deletions in each chunk by performing the tree search on each of the block deletion patterns produced in the first step. Let $V = (a_1, \cdots, a_{l_1})$ to be one of the block deletion patterns at the end of the first step. Without loss of generality, assume that $a_1, a_2, \ldots, a_s$ are non-zero, for some $s \leq l_1$. Since these $s$ blocks are not recovered in the second step of the decoding we know that $a_1, \ldots, a_s$ are each greater than 1. Furthermore, $\sum_{i=1}^{s} a_i \leq k$. Recalling that $a_{ij}$ represents the number of deletions in the $j$th chunk of the $i$th block, we have

$$\begin{aligned} a_{11} + a_{12} + \cdots + a_{1l_2} &= a_1 \\ a_{21} + a_{22} + \cdots + a_{2l_2} &= a_2 \\ &\vdots \\ a_{sl_2} + a_{sl_2} + \cdots + a_{sl_2} &= a_s. \end{aligned} \tag{26}$$

Similar to the first step, the number of non-negative integer solutions of the above set of equations is an upper bound for the number of nodes in the last level of the tree which can also serve as an upper bound for the other levels. To bound the complexity of this step we need the following lemma.

*Lemma 1:* The number of non-negative integer solutions of the set of equations in (26) when $\sum_{i=1}^{s} a_i = k$, $a_i \geq 0$, and $s$ and $l_2$ are positive integers, is bounded by

$$\binom{k/s + l_2 - 1}{l_2 - 1}^s. \tag{27}$$

Here, for a real number $x$ and integer $a$,

$$\binom{x}{a} \triangleq \frac{x(x-1)\cdots(x-a+1)}{a!}. \tag{28}$$

*Proof:* See Appendix A. □

Lemma 1 shows that (27) is an upper bound for the number of nodes in each level of the tree corresponding to the block deletion pattern $(a_1, \cdots, a_{l_1})$. Since $\sum_{i=1}^{l_1} a_i = k$ and $a_i \geq 2$ for $1 \leq i \leq s$, $s$ is a number between 1 and $\frac{k}{2}$. It is shown in Appendix B that the derivative of (27) with respect to $s$ is positive when $s > 1$. Therefore, $s = \frac{k}{2}$ maximizes (27). Thus an upper bound for the number of nodes in each level of the tree is

$$\max_{1 \leq s \leq \frac{k}{2}} \binom{k/s + l_2 - 1}{l_2 - 1}^s = \binom{l_2 + 1}{l_2 - 1}^{\frac{k}{2}} = \binom{l_2 + 1}{2}^{\frac{k}{2}}. \tag{29}$$

We therefore have

$$|\mathcal{L}_3| \leq |\mathcal{L}_1| \binom{l_2 + 1}{2}^{\frac{k}{2}}. \tag{30}$$

At each node of the tree, we compute the VT syndrome of a length $n_c l_1$ sequence and compare it with the syndrome known from $M_2$. Therefore, the complexity of this step is

$$|\mathcal{L}_3| \times l_2 \times O(n_c l_1) \leq O\left(n|\mathcal{L}_1| \binom{l_2 + 1}{2}^{\frac{k}{2}}\right) = O\left(n|\mathcal{L}_1| l_2^k\right). \tag{31}$$

Similar to the first step, many of the solutions of (26) are not compatible with VT syndromes of chunk-strings. Table IV compares the empirical value of $\mathbb{E}|\mathcal{L}_3|$ with the upper bound in (30), and shows the importance of the VT constraints in reducing the number of compatible chunk deletion patterns in Step 3.

TABLE IV
COMPARISON OF AVERAGE NUMBER OF SURVIVING PATHS AFTER STEP 3

| | Setup 1 to 3 | Setup 4 | Setup 5 | Setup 6 | Setup 7 |
|---|---|---|---|---|---|
| $\mathbb{E}\lvert\mathcal{L}_3\rvert$ | 1.92 | 6.18 | 74.43 | 28.64 | 26.16 |
| $\mathbb{E}\lvert\mathcal{L}_1\rvert\binom{l_2+1}{2}^{k/2}$ | 27.48 | 122.04 | $1.3 \times 10^6$ | $6.8 \times 10^6$ | $5.2 \times 10^{12}$ |

*Step 4: Iterative Correction of Blocks and Chunk-Strings*

In this step, we iteratively use the VT decoder for blocks and chunk strings to recover deletions. Each of the VT checks will be used at most once. Since there are $l_1$ blocks and $l_2$ chunk-strings an upper bound for the complexity is

$$|\mathcal{L}_3| \times (l_1 \times O(n_c l_2) + l_2 \times O(n_c l_1)) = O\left(n|\mathcal{L}_3|\right). \tag{32}$$

Recall from the decoding algorithm that some of the candidates will be discarded in this step, therefore, $|\mathcal{L}_4| \leq |\mathcal{L}_3|$.

*Step 5: Replacing Deletions With Erasures*

In this step, we use the linear equations for recovering the erased chunks. There are at most $k$ erased chunks and hence $kn_c$ bits erased. Hence, the complexity of finding solutions for the set of linear equations can be bounded by $O\left(n^3|\mathcal{L}_4|\right)$. We discard a candidate if there is no solution for the linear equations; therefore $|\mathcal{L}_5| \leq |\mathcal{L}_4|$.

*Step 6: Discarding Invalid/Identical Candidates*

In this step, we compute the VT syndrome of blocks and chunk-strings for all the candidates on the list and compare them with the known syndromes. Hence the complexity is $O\left(n|\mathcal{L}_5|\right)$.

We have computed the complexity of each step of the decoding in terms of the the list size at the end of the previous step.

An upper bound for the decoding complexity (not considering the effect of VT codes in eliminating incompatible deletion patterns) is $O\left(n^3\binom{k+l_1-1}{k}l_2^k\right)$. If one assumes that $k$, $l_2$, and $l_1$ are fixed and the length of the code is increased by increasing $n_c$, then the complexity of the decoding is $O(n^3)$ while the complexity of the encoding is $O(n)$. We remark again that this bound on decoding complexity is loose: as illustrated in Tables III and IV, the the VT constraints allow the decoder to discard a large number of incorrect deletion patterns in Steps and 1 and 3.

As expected, the third step of decoding (determining chunk boundaries) is the most time consuming one in practice. For example, the average wall times for the six decoding steps for setup 6 (for a Matlab implementation on a personal computer) were observed to be: 0.55ms, 0.78ms, 5.5ms, 0.41ms, 0.43ms, 0.10ms.

### *C. Tradeoffs Between Redundancy, List Sizes, and Decoding Complexity*

To get some insight into how the redundancy and the intermediate list sizes decrease with increasing $n$ (for a fixed $k$), consider the choice of parameters $l_1 = \alpha_1\ k, l_2 = \alpha_2\ k$ for $\alpha_1, \alpha_2 > 0$, and $z = \beta k n_c$ for $\beta \geq 0$. The bound on the per-symbol redundancy from (5) is

$$R_{\text{sync}} \leq \frac{\beta}{\alpha_1\alpha_2 k} + \frac{\log(1+n_c\alpha_1 k)}{\alpha_1 k n_c} + \frac{\log(1+n_c\alpha_2 k)}{\alpha_2 k n_c} + \frac{1}{n}. \tag{33}$$

Let us now consider increasing $n$ and $l_1$ by increasing $\alpha_1$, with $k, n_c, \alpha_2$ fixed. Since $n = n_c l_1 l_2 = n_c\alpha_1\alpha_2\ k^2$, $n$ increases linearly with $\alpha_1$. The first two terms of the per-symbol redundancy in (33) decrease with $\alpha_1$. Furthermore, the simulation results in Fig. 4 show that the average value of $|\mathcal{L}_1|$ (list size at the end of Step 1) decreases as $\alpha_1$ increases. We therefore expect the average list sizes at the end of Steps 1-4 to decrease with $\alpha_1$ (despite the upper bound $\binom{k+\alpha_1\ k-1}{k}$ on $|\mathcal{L}_1|$ increasing). This in turn allows us to use fewer parity check constraints (smaller value of $\beta$) which helps further reduce the redundancy as well as decoding complexity.

We now examine via an example how the choice of the parameters $(n_c, l_1, l_2)$ influence the decoding performance and complexity for fixed $(k, n)$, recalling that $n = n_c l_1\ l_2$. We let $n = 400$, $k = 5$ and $\beta = 0.5$. The code uses $z = \beta k n_c$ binary parity check constraints chosen uniformly at random. Table V shows the effect of increasing $l_1$, with $l_2$ and $n$ fixed. The decoding time decreases with increasing $l_1$ due to fewer valid deletion patterns at the end of the first step. [1] However, the rate increases with $l_1$ beyond a small value (see Figure 6). Therefore there is a tradeoff between rate and complexity when changing $l_1$. Table VI shows the effect of increasing $l_2$ with $(l_1, n)$ fixed. We observe that the effect of $l_2$ on decoding time is not as dominant as $l_1$. This is because of the importance of $l_1$ in reducing number of deletion patterns in the first step. Since the rate is symmetric with respect to $l_1$ and $l_2$, one approach to choose the parameters could be tuning $l_2$ such that it minimizes the rate for the chosen $l_1$. Our experiments shows typically choosing $l_1 > k$ and $l_2 \approx k$ gives a good tradeoff for values of $n$ and $k$ that we consider in this paper.

[1] In our current implementation, the chunks are all of equal size $n_c$, and therefore $n$ should be divisible by $l_1$ and $l_2$. However, if the binary parity check constraints are drawn uniformly at random (not from a Reed Solomon code), then we do not need equal-sized chunks or blocks. In this case, $n$ does not need to be divisible by $l_1$ and $l_2$.

TABLE V

COMPLEXITY, REDUNDANCY AND ERROR PROBABILITY CHANGES WITH $l_1$ WHEN $l_2 = 4, k = 5, n = 400$

| $(l_1, n_c)$ | $R_{\text{sync}}$ | $\mathbb{P}[\lvert\mathcal{L}_6\rvert > 1]$ | $\mathbb{E}\lvert\mathcal{L}_1\rvert$ | $\mathbb{E}\lvert\mathcal{L}_3\rvert$ | Decoding Time (s) |
|---|---|---|---|---|---|
| (4,25) | 0.1644 | 0 | 6.8664 | 24.7716 | 0.0378 |
| (5,20) | 0.1708 | 0 | 5.8459 | 15.7365 | 0.0196 |
| (10,10) | 0.2130 | 0 | 3.0266 | 4.7102 | 0.0081 |
| (20,5) | 0.2924 | 0 | 1.8694 | 2.1681 | 0.0053 |

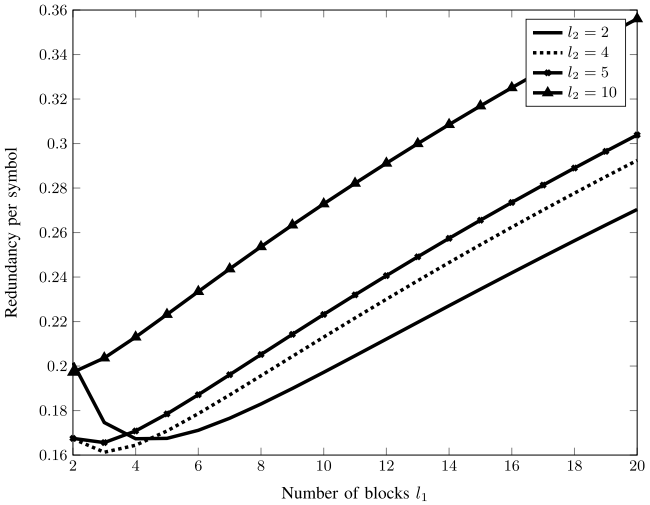


Fig. 6. Redundancy per symbol for different values of $l_1$ and $l_2$.

TABLE VI

COMPLEXITY, REDUNDANCY AND ERROR PROBABILITY CHANGES WITH $l_2$ WHEN $l_1 = 10, k = 5, n = 400$

| $(l_2, n_c)$ | $R_{\text{sync}}$ | $\mathbb{P}[\lvert\mathcal{L}_6\rvert > 1]$ | $\mathbb{E}\lvert\mathcal{L}_1\rvert$ | $\mathbb{E}\lvert\mathcal{L}_3\rvert$ | Decoding Time (s) |
|---|---|---|---|---|---|
| (2,20) | 0.1972 | 0 | 2.9710 | 4.6581 | 0.0256 |
| (4,10) | 0.2130 | 0 | 2.9706 | 4.5335 | 0.0094 |
| (8,5) | 0.2536 | 0 | 2.9880 | 2.2901 | 0.0101 |
| (10,4) | 0.2729 | $1 \times 10^{-6}$ | 2.9804 | 2.0346 | 0.0090 |

### *D. Comparison With Guess and Check (GC) Codes*

In the GC code, the sequence $X$ of length $n$ (assumed to be a power of 2) is divided into chunks of $\log n$ bits. The encoder's message consists of $c$ parity-check symbols of a systematic MDS code over $GF(n)$, computed with the information sequence $X$. The decoder considers each possible pattern of $k$ deletions, erases the chunks corresponding to the deletion pattern, and recovers the erased chunks using the MDS decoder. Decoding is successful when the recovered sequence is consistent with each of the $c$ parity symbols received from the encoder. The number of deletion patterns

tested by the decoder is $\binom{n/\log n+k-1}{k}$, and the MDS decoder run for each deletion pattern has complexity $O(k^3\ n\log n)$ (assuming a Reed-Solomon code). Therefore the decoding complexity of the GC code is $O\Big((n/\log n)^k\, k^3\ \ n\log n\Big)$. In particular, the complexity increases exponentially with the number of deletions $k$. As discussed above, the upper bound on decoding complexity of the multilayer scheme also scales exponentially with $k$. However the empirical results in Tables III and IV demonstrate that the 'typical' decoding complexity is much lower, due to a large number of deletion patterns being eliminated by the intersecting VT constraints.

*Example:* Let us compare the complexity of synchronizing a sequence of length $n = 1024$ from $k = 8$ deletions with a multilayer code and a GC code.

*Multilayer code*: The code parameters were chosen to be $l_1 = 16$, $l_2 = 8$, $n_c = 8$, and $z = 60$ random binary linear constraints. The corresponding per-symbol redundancy is $R = 0.230$. Over $10^6$ independent simulation trials, all sequences were successfully recovered by the multilayer decoder. From VI-B, the upper bound for the list size $|\mathcal{L}_1|$ at the end of Step 1 is $\binom{k+l_1-1}{k} = 490,314$. However, the average list size was observed to be $\mathbb{E}|\mathcal{L}_1| = 7.27$, which means that on average only $1.4\times 10^{-5}$ of the possible block deletion patterns were forwarded to the subsequent steps. The average list sizes at the end of Steps 3 and 4 were $\mathbb{E}|\mathcal{L}_3| = 58.16$ and $\mathbb{E}|\mathcal{L}_4| = 2.15$, again much smaller than the upper bounds. The average decoding time per trial (Matlab implementation on a personal computer) was 0.0614 seconds.

*GC Code*: In the standard construction of the GC code, the sequence is divided into chunks of $\log n = 10$ bits. To synchronize from $k = 8$ deletions, $\binom{n/\log n+k-1}{k} > 4\times 10^{11}$. For each of these patterns, the GC decoder has to run an MDS decoder. For this sequence length $n = 1024$, the authors report in [8] that the GC decoding time is of the order of seconds for $k = 3$ deletions, and of the order of minutes for $k = 4$ deletions. Due to the prohibitively large number of deletion patterns, decoding is infeasible for $k = 8$ with the default choice of chunk length $\log n = 10$.

As suggested in [8], the number of deletion patterns to be checked in the GC scheme can be reduced by increasing the chunk length, at the expense of increased redundancy. Consider a chunk length of $n_c = 30$, with $c = (k+1) = 9$ parity symbols (which is the minimum required by the decoder). The per-symbol redundancy of the GC code with these parameters is $R = \frac{cn_c}{n} = 0.263$, which is slightly higher than that of the multilayer code above. The number of deletion patterns to be checked by the GC decoder is $\binom{\lceil n/n_c\rceil+k-1}{k} = \binom{42}{8} > 10^8$. Since an MDS decoder of complexity $O(k^3\ n\log n)$ has to be run for each of these deletion patterns, GC decoding is still too complex to run on a personal computer.

To summarize, the VT constraints in the multilayer play a crucial role in eliminating a large number of deletion patterns, which makes decoding feasible for a larger number of deletions than the GC scheme. On the other hand, due to the non-linearity of the VT constraints it is difficult to obtain sharp analytical bounds on the probability of decoding failure and the typical decoding complexity.

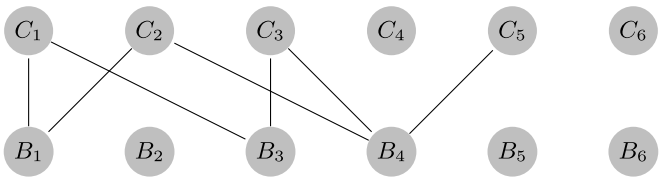


Fig. 7. Example of a chunk deletion matrix $\boldsymbol{A}_0$ and the corresponding bipartite graph.

## VII. Guess-Based VT Decoding

In this section, we consider an alternative decoder which does not use the parity-check constraints. Recall that the parity-check constraints are used in Step 5 of the decoding algorithm to recover deletions that cannot be directly recovered using the intersecting VT constraints. Here we first characterize such deletions, and then show how they can be often be recovered using only the VT constraints. Eliminating the parity-check constraints decreases the redundancy, but this comes at the expense of an increased list size.

### A. Unresolved Deletions in Step Four

Here we characterize the deletions that cannot be recovered by the iterative algorithm in the step 4 of decoding, for a given chunk-deletion matrix produced in step 3. We use a graph representation for the chunk-deletion matrix to illustrate this. Recall that the chunk deletion matrix $\boldsymbol{A}$ consists of entries $\{a_{ij}\}_{1\le i\le l_1, 1\le j\le l_2}$, where $a_{ij}$ specifies the number of deletions in the $j$th chunk of the $i$th block.

*Definition 1:* Define a bipartite graph $\mathcal{G}$ associated with each chunk-deletion matrix $\boldsymbol{A}$ with vertex sets $\mathcal{B}$ and $\mathcal{C}$. Each vertex in $\mathcal{B}$ corresponds to a block (row of $\boldsymbol{A}$), and each vertex in $\mathcal{C}$ corresponds to a chunk-string (column of $\boldsymbol{A}$). For any non-zero entry $a_{ij}$ of $\boldsymbol{A}$, there are $a_{ij}$ edges between the $i$th vertex in $\mathcal{B}$ and $j$th vertex in $\mathcal{C}$.

Figure 7 shows an example of a chunk-deletion matrix and the corresponding bipartite graph. Here a vertex $C_j$ represents the $j$th column (chunk-string) of the matrix and $B_i$ represents $i$th row (block). In the following, we will adopt usual definitions of paths and cycles from graph theory. In particular, if there are two edges between two vertices, it is considered a cycle of length 2.

In step 4, the decoder iteratively corrects deletions by identifying a row or column in $\boldsymbol{A}$ with a single one. This corresponds to finding a degree one vertex in the bipartite graph. When such a vertex is identified, the deletion is corrected and the bipartite graph updated by removing the edge connected to the degree one vertex. This process is iterated until there are no more degree one edges. In the example in Figure 7, $C_5$ is a degree one vertex, indicating that the fifth chunk-string has only one deletion. The deletion corresponding to edge between $B_4$ and $C_5$ is recovered, and this edge is then removed from the graph. The other deletions remain unresolved as there are no more degree one vertices. The following result determines the graph configurations that result in unrecovered deletions at the end of Step 4.

*Proposition 2:* A deletion occurring in the $j$th chunk of the $i$th block will not be recovered by means of the iterative

algorithm if and only if the corresponding edge, $B_iC_j$, in the graph $\mathcal{G}$ belongs to a cycle, or belongs to a path between two cycles.

*Proof:* Consider an unrecovered edge $B_iC_j$ which does not belong to a cycle. As the degree of $B_i$ is greater than one, we can find a vertex other than $C_j$ connected to $B_i$. Similarly, the degree of that vertex is greater than one, hence we can continue this procedure. Since the graph is finite we revisit a vertex which means there is a path from $B_i$ to a cycle. By repeating this argument for $C_j$, we conclude that $B_iC_j$ is in a path which connects two cycles. □

### *B. Guess-Based Decoding*

Here we show how to recover the remaining deletions at the end of step 4 of the decoding by guessing bits to break cycles in the bipartite graph. Since there are no parity-check constraints, the per-symbol redundancy is now:

$$R = \frac{\lceil \log(n_c l_2 + 1) \rceil}{n_c l_2} + \frac{\lceil \log(n_c l_1 + 1) \rceil}{n_c l_1}, \tag{34}$$

which is a saving of $z/n$ over the redundancy in (3).

To motivate the guess-based decoder, consider the matrix $\boldsymbol{A}_0$ and its corresponding graph in Figure 7. After correcting the deletion corresponding to $B_4 - C_5$, the remaining deletions form a cycle of length 6 in the graph. If we recover one of the deletions in the cycle, then we can immediately recover all the other deletions using the iterative algorithm (as there is no other cycle in the graph). We therefore guess the deleted bit (both location and value) in one of the chunks in the cycle. For instance, we can guess the deleted bit in $C_1$. Since we already know $(n_c - 1)$ bits of $C_1$, there are $(n_c + 1)$ distinct sequences that can be obtained by inserting one bit into this chunk. The decoder runs the iterative deletion correction algorithm Step 4 for each of these $n_c + 1$ obtained sequences. Since there are no other cycles in the graph, the iterative algorithm will either successfully find all the remaining deletions, or discard the sequence due to the position of the recovered bits being incompatible with the chunk they are expected to be in (known from $\boldsymbol{A}_0$). The decoder then forwards the remaining sequences, which are now of length $n$, to the sixth step of the decoding algorithm (bypassing the fifth step).

In general, for each unresolved chunk-deletion matrix at the end of Step 4, Proposition 2 identifies a minimal set of deletions that need to be guessed in order to resolve all the deletions corresponding to the chunk-deletion matrix. The proposition tells us that it is necessary and sufficient to remove a set of edges such that the remaining graph has no cycles. Hence, the minimum number of edges that need to be removed to make the graph acyclic is equal to the minimum number of bits that need to be guessed. Denote this number by $a^*$. If there are $c$ connected components in the graph with $\alpha_1, \cdots, \alpha_c$ vertices, respectively, then $a^* = e - (\sum_{i=1}^{c} \alpha_i) + c$, where $e$ is the total number of edges in the graph (total number of deletions). Since the number of distinct supersequences that can be obtained by inserting $a^*$ bits in a length $(n_c - a^*)$ binary sequence is [27]

$$\sum_{j=0}^{a^*} \binom{n_c}{j} \leq (n_c + 1)^{a^*}. \tag{35}$$

Using (35), $(n_c + 1)^{a^*}$ is an upper bound for the number sequences that need to be guessed. Note that $a^*$ is determined by the specific chunk-deletion matrix (or its bipartite graph).

In our implementation of the algorithm, the decoder chooses one of the edges in a cycle uniformly at random, removes the edge it by guessing a bit in the corresponding chunk, and then performs the iterative algorithm on the updated graph (discarding inconsistent candidates). If any unresolved deletions remain, it chooses another edge from a cycle uniformly at random, and repeats the algorithm until there are no more edges in the graph.

*Numerical simulations*: We present simulation results for the guess-based decoder, for the setups listed in Table VII. Setup 8 and 10 are similar to setups 1 and 4 in Section IV respectively, with the only difference being that there is no linear code in setups 8 and 10. The quantity $R'_{\text{sync}}$ in brackets is the higher overall redundancy per symbol when linear codes were used. The performance was recorded over $10^6$ simulation trials. The first three columns in Table VIII show the fraction of trials in which the final list size was exactly 1, 2, and 3 respectively. The fourth column shows the fraction of trials with more than 3 candidates on the final list, and the last column shows the largest list size over all $10^6$ trials.

Comparing the performance of setups 8 and 10 in Table VIII with setups 1 and 4 in Table II shows that the guess-based iterative decoder allows for smaller rates, but has a much larger probability of having more than one candidate on the final list. Guess-based decoding is effective if we are willing to tolerate list sizes greater than one with non-negligible probability.

Comparing setups 8 and 9 shows that increasing $n_c$ decreases the redundancy (according to (34)) but increases the average list size. The reason for this is that when a chunk deletion pattern contains cycles, the number of possible guesses increases with $n_c$. The same effect can be observed by comparing setups 10 and 11. Furthermore, as expected, the list

TABLE VII
NUMBER OF DELETIONS AND CODE PARAMETERS FOR EACH SETUP

| | $k$ | $n$ | $l_1$ | $l_2$ | $n_c$ | $z$ | $R_{\text{sync}}$ $(R'_{\text{sync}})$ |
|---|---|---|---|---|---|---|---|
| Setup 8 | 3 | 60 | 5 | 3 | 4 | 0 (4) | 0.583 (0.650) |
| Setup 9 | 3 | 60 | 5 | 3 | 6 | 0 (6) | 0.444 (0.511) |
| Setup 10 | 4 | 60 | 5 | 3 | 4 | 0 (16) | 0.583 (0.850) |
| Setup 11 | 4 | 60 | 5 | 3 | 6 | 0 (24) | 0.444 (0.711) |

TABLE VIII
LIST SIZE DISTRIBUTION FOR GUESS-BASED DECODER

| | $\lvert\mathcal{L}_6\rvert = 1$ | $\lvert\mathcal{L}_6\rvert = 2$ | $\lvert\mathcal{L}_6\rvert = 3$ | $\lvert\mathcal{L}_6\rvert > 3$ | $\max \lvert\mathcal{L}_6\rvert$ |
|---|---|---|---|---|---|
| Setup 8 | 92.7% | 6.3% | 0.91% | 0.09% | 6 |
| Setup 9 | 85.9% | 10.2% | 2.8% | 1.1% | 10 |
| Setup 10 | 84.3% | 12.8% | 2.3% | 0.6% | 13 |
| Setup 11 | 71.9% | 18.2% | 6.1% | 3.8% | 25 |

size increases with the number of deletions $k$ as can been seen by comparing setups 8 and 10 (and also setups 9 and 11).

## VIII. Synchronizing From a Combination of Deletions and Insertions

In this section, we use the multilayer code for synchronization when the edits are a combination of insertions and deletions. The code construction and the encoding are unchanged, and as described in Section II, the message sent by the encoder is of the form $M = [M_1, M_2, M_3]$. We describe the modifications required in the decoding algorithm to recover a combination of up to $k$ deletions and insertions. First notice that for the case where we have only insertions we can use nearly the same decoding algorithm used for the deletion only case. (Recall that VT codes can recover either a single insertion or deletion in a sequence.)

For the case where the edits are a combination of insertions and deletions, assume that the sequence $Y$ is of length $m$ can be obtained from $X$ by $a$ deletions and $b$ insertions where $a + b \leq k$. (Thus $m = n - a + b$.) We will use a similar six-step decoder for reconstructing $X$.

*1) Step 1:* In this step, we perform a tree search to find the number of insertions and deletions in each block. The output of this step is a list of block edit patterns of the form

$$V = \big((a_1, b_1),\, (a_2, b_2),\, \cdots, (a_{l_1}, b_{l_1})\big), \tag{36}$$

where $a_i$ and $b_i$ are the number of deletions and insertions, respectively, in block $i$ according to the edit pattern. Each valid edit pattern should satisfy

$$\sum_{i=1}^{l_1}(a_i + b_i) \leq k, \quad \sum_{i=1}^{l_1}(a_i - b_i) = n - m. \tag{37}$$

The tree search to construct the list of valid block edit patterns proceeds sequentially as follows. Assume that for a given node at level $j$ of the tree corresponds to a total of $d_j$ deletions and $\iota_j$ insertions in the previous $(j-1)$ blocks. The starting point of the $j$th block is then

$$p_j = (j-1)n_b - d_j + \iota_j + 1. \tag{38}$$

Note that given (37), we know that $a_j$ and $b_j$ should satisfy

$$a_j \leq \frac{k+n-m}{2} - d_j, \quad b_j \leq \frac{k+m-n}{2} - \iota_j. \tag{39}$$

The decoder computes the VT syndrome of the $j$th block, $\mathsf{syn}(Y(p_j : p_j + n_b - 1))$. If it does not match with the correct VT syndrome of block $j$ (which is known from the message sent by the encoder), the possible values for $a_j$ and $b_j$ are all the pairs which satisfy (39) and also $a_j + b_j \neq 0$. If $d_j + \iota_j = k$, then we discard the correspond branch of the tree.

If the VT syndrome of the block matches with the correct VT syndrome, then the possible values for $(a_j, b_j)$ are all the pairs which satisfy (39) as well as $a_j + b_j \neq 1$.

*2) Step 2:* For each valid block edit pattern from step 1, the decoder recovers the edits in the blocks with a single insertion or deletion, i.e. when $a_i + b_i = 1$. After recovering the edit in a block, the edit pattern is updated.

*3) Step 3:* The goal of this step is to create a list of chunk-edit matrices, each of dimension $l_1 \times l_2$, the $(i, j)$ entry of the matrix is a pair $(a_{ij}, b_{ij})$. Here $a_{ij}, b_{ij}$ denote the number of deletions and insertions, respectively, in the $j$th chunk of $i$th block. Similar to step 3 of decoding in the deletion-only case, we construct these chunk-edit matrices via a tree search for each block edit pattern of the form $((a_1, b_1),\, (a_2, b_2),\, \cdots, (a_{l_1}, b_{l_1}))$. This is done sequentially as follows.

For each node at level $j$ of the tree, the decoder knows $a_{ih}$ and $b_{ih}$ for all $i$ and $h < j$. Thus it knows the starting position of the $j$th chunk of each block, and can therefore form the $j$th chunk-string and compute its VT syndrome. This computed VT syndrome is compared with the correct syndrome of $j$th chunk-string (known from the encoder's message). There are two possibilities:

1) If the VT syndrome of the $j$th chunk-string matches the correct syndrome, the possible values for $a_{ij}$ and $b_{ij}$ are all non-negative integers that satisfy $\sum_{i=1}^{l_1}(a_{ij} + b_{ij}) \neq 1$ and also:
$$a_{ij} \leq a_i - \sum_{h=1}^{j-1} a_{ih} \quad \text{and} \quad b_{ij} \leq b_i - \sum_{h=1}^{j-1} b_{ih}. \tag{40}$$
2) If the VT syndrome of the $j$th chunk-string does not match the correct syndrome, the possible values for $a_{ij}$ and $b_{ij}$ are all non-negative integers that satisfy (40), and also $\sum_{i=1}^{l_1}(a_{ij} + b_{ij}) \neq 0$. The node will be discarded if
$$a_i = \sum_{h=1}^{j-1} a_{ih} \quad \text{and} \quad b_i = \sum_{h=1}^{j-1} b_{ih}, \quad \text{for} \quad 1 \leq i \leq l_1. \tag{41}$$

*4) Steps 4 to 6:* The last three steps are very similar to the deletion only case. In step 4, we iteratively use the VT decoder to recover any single deletion in blocks or chunk-strings. Similarly to the deletion only case, we will discard a candidate if the recovered bit lies in a wrong chunk. In step 5, we replace any chunk which still contains edits with $n_c$ erasures, and use the parity-check constraints to recover erasures; any inconsistent candidate will be discarded. Finally, at the sixth step we check all constraints for the remaining candidates and output all compatible sequences.

*Numerical simulations*: We evaluated the performance of the decoder for the seven setups in Table VII. For each $k$, the number of deletions was chosen to be an integer $d$ between 0 and $k$ uniformly at random. The number of insertions was then $(k - d)$. Table IX shows the average list size for each each of the setups, in the different steps of the decoding. As expected, with a combination of insertions and deletions, the number of valid block edit patterns (in step 1) and chunk edit matrices (in Step 3) are larger than the deletion-only case. As the decoding complexity of each step depends on the list size at the end of the previous step, the average decoding complexity is also higher than the deletion-only case. However, we observe that the increase in the final list size compared to the deletion-only case (Table II) is negligible. This is because

TABLE IX
LIST SIZE AFTER EACH STEP WHEN THERE ARE BOTH INSERTIONS AND DELETIONS

| | $\mathbb{E}\|\mathcal{L}_1\|$ | $\mathbb{E}\|\mathcal{L}_3\|$ | $\mathbb{E}\|\mathcal{L}_4\|$ | $\mathbb{E}\|\mathcal{L}_6\|$ | $\max\|\mathcal{L}_6\|$ | $\mathbb{P}[\|\mathcal{L}_6\|>1]$ |
|---|---|---|---|---|---|---|
| Setup 1 | 2.96 | 3.44 | 2.12 | 1.004 | 7 | $4.215\times10^{-4}$ |
| Setup 2 | 2.96 | 3.44 | 2.12 | 1.000 | 2 | $1.3\times10^{-5}$ |
| Setup 3 | 2.96 | 3.44 | 2.12 | 1.000 | 2 | $5\times10^{-6}$ |
| Setup 4 | 7.78 | 17.66 | 5.95 | 1.000 | 2 | $4\times10^{-6}$ |
| Setup 5 | 86.29 | 782.38 | 22.5 | 1 | 1 | 0 |
| Setup 6 | 82.73 | 254.06 | 15.08 | 1 | 1 | 0 |
| Setup 7 | 210.74 | 1523.0 | 34.41 | 1 | 1 | 0 |

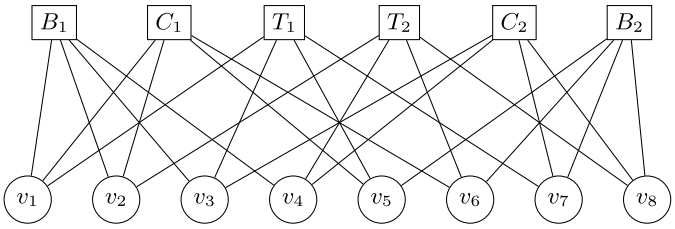


Fig. 8. Factor graph representation of a three-layer code.

a large number of the edit patterns are inconsistent with the intersecting VT constraints and the parity-check constraints.

## IX. DISCUSSION AND FUTURE WORK

In this work we introduced a new method for one-way synchronization of binary sequences based on a combination of intersecting VT constraints and linear parity-check constraints. We showed that the intersecting VT constraints enable a iterative decoding procedure which alternates between identifying compatible edit patterns, and correcting subsequences indicated by these patterns as having a single edit.

*Generalizing the two-layer construction*: The code construction based on two layers of intersecting VT constraints can be generalized in many ways. First, it can be extended to sequences over non-binary alphabets with size $q > 2$ by using the $q$-ary VT codes proposed by Tenengoltz [28]. The construction can also be generalized to include multiple layers of intersecting VT constraints. We illustrate the idea with an example of a three-layer construction. Consider a sequence $X = [v_1, v_2, \cdots, v_8]$ consisting of eight chunks of length $n_c$ each. The message consists of syndromes corresponding to three kinds of intersecting VT constraints, defined as follows. The two block constraints, $B_1$ and $B_2$, are the VT syndromes of $[v_1, v_2, v_3, v_4]$ and $[v_5, v_6, v_7, v_8]$. The two chunk-string constraints $C_1$ and $C_2$, are the VT syndromes of $[v_1, v_2, v_5, v_6]$ and $[v_3, v_4, v_7, v_8]$. The third set of constraints, $T_1$ and $T_2$, are the VT syndromes of $[v_1, v_3, v_5, v_7]$ and $[v_2, v_4, v_6, v_8]$. Figure 8 illustrates the three sets of constraints using a factor graph, with the circles and squares representing the chunks and constraints, respectively. The decoding algorithm for a such a construction is a straightforward extension of that in Section III: we identify the compatible chunk edit patterns via a tree search, and then using the VT constraints to iteratively solve for sub-sequences with a single deletion.

Extending this idea further, we could consider an $L$-layer construction with $L = \Theta(\log n)$ layers, $l_1 = l_2 = \ldots = l_L = 2$, $n_c = \log n$, and $z = kn_c$ binary parity check constraints. Such a construction would have an overall redundancy of $k \log n + 2L \log(\frac{n}{2}+1)$, which is near-optimal. Moreover, since each layer has only two VT constraints the number of sequences compatible with each layer is at most $k$. Developing an iterative decoding algorithm to recover the chunks from these constraints, and investigating the trade-offs between redundancy, list-size, and decoding complexity is an interesting direction for future research.

Another direction for future work is to use the multilayer code construction for communication over the deletion channel. Such a channel code will consist of all sequences with a specified set of values for the intersecting VT and parity-check constraints. The decoding algorithm is essentially the same as that described in Section III, however constructing an efficient encoder for this channel code is an open question.

## APPENDIX

### A. Proof of Lemma 1

*Proof:* Define the function $p(x) = \binom{x+l_2-1}{l_2-1}$. We note that $p$ is a polynomial of degree $(l_2-1)$, and $p(a_i)$ is the number of non-negative integer solutions to the equation $\sum_{j=1}^{l_2} a_{ij} = a_i$.

We first show that $p(x)p(y) \le p(\frac{x+y}{2})^2$ for any two positive real numbers $x, y$. To show this, we need to prove

$$\prod_{i=1}^{l_2-1}(y+i)(x+i) \le \prod_{i=1}^{l_2-1}\left(\frac{x+y}{2}+i\right)^2, \tag{42}$$

which clearly follows from $xy \le (\frac{x+y}{2})^2$. Now if we define $g(x) \triangleq \ln(p(x))$, we have $g(x)+g(y) \le 2g(\frac{x+y}{2})$ which means $g$ is mid-point concave, and since it is continuous, it is generally concave. Hence, we have $g(x_1)+g(x_2)+\cdots+g(x_n) \le ng(\sum_{i=1}^n x_i/n)$ for any integer $n$ and positive $x_i$'s. Therefore, we have

$$p(x_1)p(x_2)\cdots p(x_n) \le p\left(\frac{\sum_{i=1}^n x_i}{n}\right)^n. \tag{43}$$

Choosing $n = s$ and $x_i = a_i$ yields the result. □

### B. Derivative of (27)

Here we show that the derivative of $f(s) = \binom{k/s+l_2-1}{l_2-1}^s$ with respect to $s$ is positive for $s > 0$. We write $f(s) = h(s; l_2)^s$, where

$$h(s; l_2) = \binom{k/s+l_2-1}{l_2-1}. \tag{44}$$

Hence, we have:

$$\frac{d\ln f(s)}{ds} = \frac{1}{f(s)}f'(s) = \ln(h(s;l_2)) + \frac{s}{h(s;l_2)}h'(s;l_2). \tag{45}$$

Therefore, to prove $df/ds > 0$ for $s > 0$ we need to show that

$$\ln(h(s;l_2)) + \frac{s}{h(s;l_2)}h'(s;l_2) > 0. \tag{46}$$

From the definition in (28), we can write $h(s;l_2) = p(s^{-1};l_2)/(l_2-1)!$, where

$$p(s;l_2) = (ks+1)(ks+2)\cdots(ks+l_2-1). \tag{47}$$

Using this in (46), we need to show that

$$\ln(h(s;l_2)) > \frac{p'(s^{-1};l_2)}{sp(s^{-1};l_2)}. \tag{48}$$

We prove (48) by induction on $l_2$. For $l_2 = 2$, we need to show that

$$\ln(ks^{-1}+1) > \frac{k}{(k+s)} \tag{49}$$

Letting $x = ks^{-1}$, then we can rewrite (49) as $\ln(x+1) > \frac{x}{(x+1)}$, which holds for all $x > 0$. Assuming that (48) holds for $l_2$, we prove it for $l_2+1$. We have

$$\ln\left(h(s;l_2+1)\right) = \left(\ln(ks^{-1}+l_2) - \ln(l_2)\right) + \ln\left(h(s;l_2)\right) \tag{50}$$

$$> \left(\ln(ks^{-1}+l_2) - \ln\left(l_2\right)\right) + \frac{p'(s^{-1};l_2)}{sp(s^{-1};l_2)}, \tag{51}$$

where the inequality holds by the induction hypothesis.

Using (47), we have

$$\frac{p'(s;l_2+1)}{p(s;l_2+1)} = \frac{d\ln p(s;l_2+1)}{ds} = \sum_{i=1}^{l_2}\frac{k}{i+ks}. \tag{52}$$

Therefore,

$$\frac{p'(s^{-1};l_2+1)}{sp(s^{-1};l_2+1)} = \frac{k}{k+l_2\ s} + \frac{p'(s^{-1};l_2)}{sp(s^{-1};l_2)}. \tag{53}$$

Comparing the RHS of (51) and (53) we have to prove that

$$\ln\left(\frac{ks^{-1}+l_2}{l_2}\right) > \frac{k}{k+l_2\ s}. \tag{54}$$

Letting $x = ks^{-1}/l_2$, this is equivalent to showing that $\ln(x+1) > \frac{x}{x+1}$. This holds for all $x > 0$, which completes the proof.

## REFERENCES


[1] A. Orlitsky, "Interactive communication of balanced distributions and of correlated files," *SIAM J. Discrete Math.*, vol. 6, no. 4, pp. 548–564, Nov. 1993.

[2] U. Irmak, S. Mihaylov, and T. Suel, "Improved single-round protocols for remote file synchronization," in *Proc. IEEE 24th Annu. Joint Conf. IEEE Comput. Commun. Societies*, vol. 3, Mar. 2005, pp. 1665–1676.

[3] N. Ma, K. Ramchandran, and D. Tse, "Efficient file synchronization: A distributed source coding approach," in *Proc. IEEE Int. Symp. Inf. Theory Proc.*, Jul. 2011, pp. 583–587.

[4] D. Belazzougui and Q. Zhang, "Edit distance: Sketching, streaming, and document exchange," in *Proc. IEEE 57th Annu. Symp. Found. Comput. Sci. (FOCS)*, Oct. 2016, pp. 51–60.

[5] K. Cheng, Z. Jin, X. Li, and K. Wu, "Deterministic document exchange protocols, and almost optimal binary codes for edit errors," in *Proc. IEEE 59th Annu. Symp. Found. Comput. Sci. (FOCS)*, Oct. 2018, pp. 200–211.

[6] B. Haeupler, "Optimal document exchange and new codes for insertions and deletions," 2018, *arXiv:1804.03604*. [Online]. Available: http://arxiv.org/abs/1804.03604

[7] H. Jowhari, "Efficient communication protocols for deciding edit distance," in *Eur. Symp. Algorithms*. Berlin, Germany: Springer, 2012, pp. 648–658.

[8] S. K. Hanna and S. El Rouayheb, "Guess & check codes for deletions, insertions, and synchronization," *IEEE Trans. Inf. Theory*, vol. 65, no. 1, pp. 3–15, Jan. 2019.

[9] G. Cormode, M. Paterson, S. C. Sahinalp, and U. Vishkin, "Communication complexity of document exchange," in *Proc. ACM-SIAM Symp. Discrete Algorithms*, pp. 197–206, 2000.

[10] A. Orlitsky and K. Viswanathan, "Practical protocols for interactive communication," in *Proc. IEEE Int. Symp. Inf. Theory*, Jun. 2001, p. 115.

[11] R. Venkataramanan, V. Narasimha Swamy, and K. Ramchandran, "Low-complexity interactive algorithms for synchronization from deletions, insertions, and substitutions," *IEEE Trans. Inf. Theory*, vol. 61, no. 10, pp. 5670–5689, Oct. 2015.

[12] S. M. S. Tabatabaei Yazdi and L. Dolecek, "Synchronization from deletions through interactive communication," *IEEE Trans. Inf. Theory*, vol. 60, no. 1, pp. 397–409, Aug. 2014.

[13] F. Sala, C. Schoeny, N. Bitouze, and L. Dolecek, "Synchronizing files from a large number of insertions and deletions," *IEEE Trans. Commun.*, vol. 64, no. 6, pp. 2258–2273, Jun. 2016.

[14] A. Trigdell, "Efficient algorithms for sorting synchronization," Ph.D. dissertation, Dept. Comput. Sci., Austral. Nat. Univ., Canberra ACT, Australia, Feb. 1999.

[15] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Automatika i Telemkhanika*, vol. 26, no. 2, pp. 288–292, 1965.

[16] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady Akademii Nauk SSSR*, vol. 163, no. 4, pp. 845–848, 1965.

[17] C. Schoeny, A. Wachter-Zeh, R. Gabrys, and E. Yaakobi, "Codes correcting a burst of deletions or insertions," *IEEE Trans. Inf. Theory*, vol. 63, no. 4, pp. 1971–1985, Apr. 2017.

[18] J. Sima and J. Bruck, "Optimal $k$-deletion correcting codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 847–851. https://arxiv.org/abs/1910.12247.

[19] S. K. Hanna and S. El Rouayheb, "List decoding of deletions using guess & check codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 2374–2378.

[20] A. Wachter-Zeh, "List decoding of insertions and deletions," *IEEE Trans. Inf. Theory*, vol. 64, no. 9, pp. 6297–6304, Sep. 2018.

[21] A. Orlitsky and K. Viswanathan, "One-way communication and error-correcting codes," *IEEE Trans. Inf. Theory*, vol. 49, no. 7, pp. 1781–1788, Jul. 2003.

[22] M. C. Davey and D. J. C. Mackay, "Reliable communication over channels with insertions, deletions, and substitutions," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 687–698, 2001.

[23] K. A. S. 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. Inf. Theory*, vol. 58, no. 3, pp. 1804–1808, Mar. 2012.

[24] J. Brakensiek, V. Guruswami, and A. Zbarsky, "Efficient low-redundancy codes for correcting multiple deletions," *IEEE Trans. Inf. Theory*, vol. 64, no. 5, pp. 3403–3410, May 2018.

[25] N. J. A. Sloane, "On single-deletion-correcting codes," in *Designs, Ohio State University (Ray-Chaudhuri Festschrift)*. Columbus, OH, USA: Ohio State Univ., 2000, pp. 273–291. [Online]. Available: https://arxiv.org/abs/math/0207197.

[26] K. A. S. Abdel-Ghaffar and H. C. Ferreira, "Systematic encoding of the Varshamov-Tenengol'ts codes and the Constantin-Rao codes," *IEEE Trans. Inf. Theory*, vol. 44, no. 1, pp. 340–345, Jan. 1998.

[27] V. I. Levenshtein, "Efficient reconstruction of sequences from their subsequences or supersequences," *J. Combinat. Theory A*, vol. 93, no. 2, pp. 310–332, 2001.

[28] G. Tenengolts, "Nonbinary codes, correcting single deletion or insertion (Corresp.)," *IEEE Trans. Inf. Theory*, vol. 30, no. 5, pp. 766–769, Sep. 1984.


**Mahed Abroshan** received the B.Sc. degree in mathematics, the B.Sc. degree in electrical engineering, and the M.Sc. degree in communication system from Sharif University of Technology, Iran, in 2013, 2014, and 2015, respectively, and the Ph.D. degree from the Department of Engineering, University of Cambridge, in 2019. He held a post-doctoral position with the Applied Mathematics and Theoretical Physics Department, University of Cambridge. He is currently a Post-Doctoral Research Associate with The Alan Turing Institute. His research interests include deep learning, information theory, and coding for communication.

**Ramji Venkataramanan** (Senior Member, IEEE) received the B.Tech. degree from the Indian Institute of Technology Madras in 2002, and the Ph.D. degree in electrical engineering (Systems) from the University of Michigan, Ann Arbor, in 2008. He held post-doctoral positions at Stanford University and Yale University, before joining the University of Cambridge in 2013, where he is currently a Reader in information engineering. He is also a Turing Fellow at The Alan Turing Institute. His research interests include statistical learning, information theory, and communication theory. He is an Associate Editor of the IEEE TRANSACTIONS ON INFORMATION THEORY and an Editor of the IEEE TRANSACTIONS ON COMMUNICATIONS.

**Albert Guillén i Fàbregas** (Senior Member, IEEE) received the Telecommunication Engineering degree from the Universitat Politècnica de Catalunya, the Electronics Engineering degree from the Politecnico di Torino in 1999, and the Ph.D. degree in communication systems from the École Polytechnique Fédérale de Lausanne (EPFL) in 2004.

Since 2020, he has been a Faculty Member of the University of Cambridge. Since 2011, he has been an ICREA Research Professor at Universitat Pompeu Fabra. He has held appointments at the New Jersey Institute of Technology, Telecom Italia, European Space Agency (ESA), Institut Eurécom, and University of South Australia, as well as visiting appointments at EPFL, École Nationale des Télécommunications (Paris), Universitat Pompeu Fabra, University of Cambridge, University of South Australia, Centrum Wiskunde & Informatica, and Texas A&M University at Qatar. His specific research interests include the areas of information theory, communication theory, coding theory, digital modulation, and signal processing techniques.

Dr. Guillén i Fàbregas is a member of the Young Academy of Europe, and received the Starting and Consolidator Grants from the European Research Council, the Young Authors Award of the 2004 European Signal Processing Conference, the 2004 Best Doctoral Thesis Award from the Spanish Institution of Telecommunications Engineers, and the Research Fellowship of the Spanish Government, to join ESA. He is also an Associate Editor of the IEEE TRANSACTIONS ON INFORMATION THEORY and an Editor of the *Foundations and Trends in Communications and Information Theory* (Now Publishers), and was an Editor of the IEEE TRANSACTIONS ON WIRELESS COMMUNICATIONS.