Criss-Cross Insertion and Deletion Correcting Codes

Rawad Bitar, *Member, IEEE*, Lorenz Welter, *Graduate Student Member, IEEE*, Ilia Smagloy, *Student Member, IEEE*, Antonia Wachter-Zeh, *Senior Member, IEEE*, and Eitan Yaakobi, *Senior Member, IEEE*

***Abstract*—This paper studies the problem of constructing codes correcting deletions in arrays. Under this model, it is assumed that an $n \times n$ array can experience deletions of rows and columns. These deletion errors are referred to as $(t_{\mathrm{r}}, t_{\mathrm{c}})$-*criss-cross deletions* if t_{r} rows and t_{c} columns are deleted, while a code correcting these deletion patterns is called a $(t_{\mathrm{r}}, t_{\mathrm{c}})$-*criss-cross deletion correction code*. The definitions for *criss-cross insertions* are similar. It is first shown that when $t_r = t_c$ the problems of correcting criss-cross deletions and criss-cross insertions are equivalent. The focus of this paper lies on the case of $(1, 1)$-criss-cross deletions. A non-asymptotic upper bound on the cardinality of $(1, 1)$-criss-cross deletion correction codes is shown which assures that the redundancy is at least $2n - 3 + 2\log n$ bits. A code construction with an existential encoding and an explicit decoding algorithm is presented. The redundancy of the construction is at most $2n + 4\log n + 7 + 2\log e$. A construction with explicit encoder and decoder is presented. The explicit encoder adds an extra $5\log n + 5$ bits of redundancy to the construction.**

***Index Terms*—Insertion/deletion correcting codes, array codes, criss-cross deletion errors.**

I. Introduction

RECENTLY, codes correcting insertions/deletions attracted a lot of attention due to their relevance in many applications such as DNA-based data storage systems [2], communication systems [3] and file synchronization [4]–[7]. Due to the loss of synchronization and working over vector spaces of different dimension, correcting deletions and insertions is seen as a harder problem than correcting substitution errors.

The problem of coding for the deletion channel was introduced by Levenshtein [8] in the 1960s. A set $\mathcal{C}$ of binary vectors of length n is a k-deletion correcting code if and only if any two vectors in $\mathcal{C}$ do not share a common subsequence of length $n-k$. Levenshtein showed [8] that a code can correct any combination of k insertions and deletions if and only if it can correct k deletions. The main property of the codes being optimized is the redundancy defined as $R \triangleq n - \log|\mathcal{C}|$ where n is the length of the codewords in $\mathcal{C}$ and $|\mathcal{C}|$ is the cardinality of the code. The optimal redundancy $n - \log|\mathcal{C}|$ of any k-deletion correcting code $\mathcal{C}$ is $\mathcal{O}(k\log n)$ [8]. The Varshamov-Tenengolts (VT) code [9] is a nearly optimal single insertion correcting code with redundancy $\log(n+1)$ bits. Constructing k-deletion/insertion correcting codes with small redundancy was the focus of several recent works, e.g., [10]–[16]. In [17] and [18], the authors construct codes that can correct bursts of deletions. The main idea of the papers is to imagine the codeword as a binary array and to use the structure of that array to detect and correct bursts of deletions that happen in the one-dimensional codeword.

In this paper we extend the one-dimensional study of deletion and insertion correction to two-dimensional arrays. A $(t_{\mathrm{r}}, t_{\mathrm{c}})$-*criss-cross deletion* is the event in which an $n \times n$ array experiences a deletion of t_{r} rows and t_{c} columns. A code capable of correcting all $(t_{\mathrm{r}}, t_{\mathrm{c}})$-criss-cross deletions is referred to as $(t_{\mathrm{r}}, t_{\mathrm{c}})$-*criss-cross deletion correcting code* and $(t_{\mathrm{r}}, t_{\mathrm{c}})$-*criss-cross insertion correcting codes* are defined similarly. Coding in the two dimensional space has proved profitable for data storage and wireless communications [19]–[26]. Therefore, we find it important to understand the generalization of the well-studied one-dimensional insertion- and deletion-correcting codes to the two-dimensional space. The main advantage of coding in the two-dimensional space is to leverage the structure of the code arrays rather than applying one dimensional deletion/insertion correcting codes on each dimension of the array. Along this line of thought, [27] studies the problem of correcting a predetermined number of row and column deletions in two-dimensional arrays. Furthermore, the trace-reconstruction problem, which is a variant of the deletion channel, is investigated in the two-dimensional space in [28].

It is well-known that in the one-dimensional case the size of the single-deletion ball equals the number of runs in the word. However, the characterization of the arrays that can be obtained from a $(1, 1)$-criss-cross deletion is more complicated. Nonetheless, we derive a non-asymptotic lower bound on the redundancy of these codes. Second, we propose a code construction which heavily depends on the construction of non-binary single-insertion/deletion correcting codes by

Manuscript received September 14, 2020; revised May 10, 2021; accepted August 25, 2021. Date of publication September 9, 2021; date of current version November 22, 2021. This work was supported in part by the European Research Council (ERC) through the European Union's Horizon 2020 Research and Innovation Programme under Grant 801434 and in part by the Technical University of Munich—Institute for Advanced Studies, funded by the German Excellence Initiative and European Union Seventh Framework Programme under Grant 291763. An earlier version of this paper was presented in part at ISITA 2020 [1]. *(Corresponding author: Rawad Bitar.)*

Rawad Bitar, Lorenz Welter, and Antonia Wachter-Zeh are with the Institute for Communications Engineering, Technical University of Munich (TUM), 80333 Munich, Germany (e-mail: rawad.bitar@tum.de; lorenz.welter@tum.de; antonia.wachter-zeh@tum.de).

Ilia Smagloy and Eitan Yaakobi are with the Computer Science Department, Technion—Israel Institute of Technology, Haifa 3200003, Israel (e-mail: ilia.smagloy@cs.technion.ac.il; yaakobi@cs.technion.ac.il).

Communicated by L. Dolecek, Associate Editor for Coding Techniques.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TIT.2021.3111450.

Digital Object Identifier 10.1109/TIT.2021.3111450

Tenengolts [29], which can be seen as the extension of the q-ary alphabet of [8]. In the one-dimensional case, successful decoding from deletions in the transmitted word does not necessarily guarantee that the indices of the deleted symbols are known since the deletion of symbols from the same run results in the same output. While this does not impose a constraint in the one-dimensional case, we had to take this constraint into account when using non-binary single-deletion correcting codes as our component codes.

The rest of the paper is organized as follows. In Section II, we formally define the codes and notations that we use throughout the paper. We give a high level summary of the presented results in Section III. We prove in Section IV that the correction of (t, t)-criss-cross deletions and insertions is equivalent. In Section V, we give a non-asymptotic upper bound on the cardinality of $(1, 1)$-criss-cross deletion correcting codes. This bound shows that the minimum redundancy of these codes is $2n - 3 + 2\log n$ bits. In Section VI, we construct $(1, 1)$-criss-cross deletion correcting codes that we call CrissCross codes. The correctness of this family of codes is given by an explicit decoding algorithm. The redundancy of the proposed CrissCross codes is at most $2n + 4\log n + 7 + 2\log e$. We present in Section VII CrissCross codes with explicit encoder and decoder. We show that the explicit encoder comes at the expense of increasing the redundancy by $5\log n + 5$ bits compared to the existence result. We conclude the paper in Section VIII.

II. Definitions and Preliminaries

This section formally defines the codes and notations that we use throughout this paper. Let $\Sigma_q \triangleq \{0, \dots, q-1\}$ be the q-ary alphabet. We denote by $\Sigma_q^{n\times n}$ the set of all q-ary arrays of dimension $n \times n$. All logarithms are base 2 unless otherwise indicated. For two integers $i, j \in \mathbb{N}$, $i \leqslant j$, the set $\{i, \dots, j\}$ is denoted by $[i, j]$ and the set $\{1, \dots, j\}$ is denoted by $[j]$. For an array $\mathbf{X} \in \Sigma_q^{n\times n}$, we denote by $X_{i,j}$ the entry of $\mathbf{X}$ positioned at the i-th row and j-th column. We denote the i-th row and j-th column of $\mathbf{X}$ by $\mathbf{X}_{i,[n]}$ and $\mathbf{X}_{[n],j}$, respectively. Similarly, we denote by $\mathbf{X}_{[i_1,i_2],[j_1,j_2]}$ the sub array of $\mathbf{X}$ formed by rows i_1 to i_2 and their corresponding entries from columns j_1 to j_2.

For two positive integers $t_\mathrm{r}, t_\mathrm{c} \leqslant n$, we define a $(t_\mathrm{r}, t_\mathrm{c})$-*criss-cross deletion* in an array $\mathbf{X} \in \Sigma_q^{n\times n}$ to be the deletion of any t_r rows and t_c columns of $\mathbf{X}$. We denote by $\mathbb{D}_{t_\mathrm{r},t_\mathrm{c}}(\mathbf{X})$ the set of all arrays that result from $\mathbf{X}$ after a $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross deletion (i.e., the two-dimensional deletion ball[1]). In a similar way we define $(t_\mathrm{r}, t_\mathrm{c})$-*criss-cross insertion* and the set $\mathbb{I}_{t_\mathrm{r},t_\mathrm{c}}(\mathbf{X})$ for the insertion case. If $t_\mathrm{r} = t_\mathrm{c} = t$, we will use the notation of $\mathbb{D}_t(\mathbf{X})$, (t)-*criss-cross deletion*, (t)-*criss-cross insertion*, and $\mathbb{I}_t(\mathbf{X})$. Note that the order between the row and column deletions/insertions does not matter.

Definition 1 ($(t_\mathrm{r}, t_\mathrm{c})$-Criss-Cross Deletion Correction Code): A $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross deletion correcting code $\mathcal{C}$ is a code that can correct any $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross deletion. A $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross insertion correcting code is defined similarly.

For clarity of presentation, we will refer to a $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross deletion as a (t)-*criss-cross deletion* when $t = t_\mathrm{r} = t_\mathrm{c}$ and $(t_\mathrm{r}, t_\mathrm{c})$-criss-cross deletion correcting code as (t)-*criss-cross deletion correcting code*. The corresponding definitions for the insertion case are similar. Notice that throughout this paper, we do not consider combinations of insertions and deletions (c.f. Section III).

In our code construction we use Varshamov-Tenengolts (VT) single-deletion correcting codes [9]. A VT code was proven by Levenshtein [8] to correct a single deletion in a binary string of length n, with redundancy not more than $\log(n+1)$ bits. In fact, we use Tenengolt's extension [29] for the q-ary alphabet, which is briefly explained next. For a q-ary vector $\mathbf{x} = (x_1, \dots, x_n)$ we associate its *binary signature* $\boldsymbol{s} = (s_1, \dots, s_n)$. The entries of $\mathbf{s}$ are calculated such that $s_1 = 1$ and $s_i = 1$ if $x_i \geqslant x_{i-1}$ or $s_i = 0$ otherwise for $i > 1$. Thus, all q-ary vectors of length n can be split into disjoint cosets $\mathcal{VT}_{n,q}(a, b)$ defined as the set of all $\mathbf{x}$ with signature $\mathbf{s}$ satisfying

$$\sum_{i=1}^{n}(i-1)s_i \equiv a \mod n, \quad \sum_{i=1}^{n} x_i \equiv b \mod q,$$

where $0 \leqslant a \leqslant n-1, 0 \leqslant b \leqslant q-1$. Each coset is a single q-ary insertion/deletion correcting code. Note that the qn disjoint cosets form a partition of Σ_q^n. Therefore, by the pigeon-hole principal, there exists a set (or a VT code) $\mathcal{VT}_{n,q}(a^\star, b^\star)$ such that

$$|\mathcal{VT}_{n,q}(a^\star, b^\star)| \geqslant \frac{q^n}{qn}.$$

III. Main Results

Our main results can be summarized as follows. In Theorem 1, we extend the equivalence between insertion correcting codes and deletion correcting to the 2-dimensional codes considered in this setting. Namely, we show that a given code $\mathcal{C}$ is a (t)-criss-cross deletion correcting code if and only if $\mathcal{C}$ is a (t)-criss-cross insertion correcting code. As a consequence, all our results proven for the (1)-criss-cross deletion case hold for the insertion case as well. To evaluate how good a given (1)-criss-cross deletion correcting code is, we derive a non-asymptotic upper bound on the cardinality of any criss-cross deletion correcting code as follows.

A. Lower Bound

(Theorem 11) The non-asymptotic redundancy of a q-ary (1)-criss-cross deletion correcting code $\mathcal{C}$ is bounded from below by $R \geqslant 2n - 3 + 2\log_q n$.

We show that there exist (1)-criss-cross deletion/insertion correction codes that have redundancy $2\log n + o(1)$ bits far from our lower bound. We do so by constructing an existential (1)-criss-cross deletion correction code called CrissCross code that has redundancy $2n + 4\log n + o(1)$. We extend the existential construction to a (1)-criss-cross code with explicit encoder and decoder at the expense of increasing the redundancy by $5\log n + 5$ bits.

[1] Strictly speaking, the set $\mathbb{D}_{t_\mathrm{r},t_\mathrm{c}}(\mathbf{X})$ must be called the two-dimensional deletion *sphere* of $\mathbf{X}$. However, we abuse terminology and refer to this set as the deletion ball to follow the nomenclature used by the literature on deletion-correcting codes. The same holds for the set $\mathbb{I}_{t_\mathrm{r},t_\mathrm{c}}(\mathbf{X})$.

B. Code Constructions

The CrissCross code constructed in Section VI is a (1)-criss-cross deletion/insertion correcting code (Theorem 13). The redundancy of the code is upper bounded by $2n + 4\log n + o(1)$ bits (Corollary 14). The encoder of the code can be made systematic at the expense of increasing the redundancy to at most $2n + 9\log n + o(1)$ bits (Theorem 18).

C. Extensions

In this work we restrict our attention to (t)-criss-cross deletions and insertions. However, our (t)-criss-cross deletion-correcting code construction can correct a mixed (1)-criss-cross error defined as a row/column deletion and a column/row insertion. In addition our codes can correct a single row insertion/deletion or a single column insertion/deletion. Nevertheless, the bound on the redundancy does not necessarily hold for those general problems. In fact, we show in [30] that, under the generalized model, a code correcting a (1)-criss-cross deletion is able to correct two row deletions (no column deletions) or two column deletions (no row deletions). We leave those general problems as an interesting direction for future research.

IV. Equivalence Between Insertion and Deletion Correction

In this section we first show an equivalence between a (1)-criss-cross deletion correcting code and a (1)-criss-cross insertion correcting code (Theorem 1). Then we use the result of Theorem 1 to prove the more general equivalence between (t)-criss-cross deletion correcting codes and (t)-criss-cross insertion correcting codes for all $t \in [n-1]$ (Corollary 2).

Theorem 1: A code $\mathcal{C} \subset \Sigma_q^{n\times n}$ is a (1)-criss-cross deletion correcting code if and only if it is a (1)-criss-cross insertion correcting code.

Corollary 2: For any integer $t \in [n-1]$, a code $\mathcal{C} \subset \Sigma_q^{n\times n}$ is a (t)-criss-cross deletion correcting code if and only if it is a (t)-criss-cross insertion correcting code.

Note that in the one-dimensional case Theorem 1 holds since the intersection of the deletion balls of two vectors is not trivial if and only if the intersection of their insertion balls is not trivial [8]. Since this property holds over any alphabet, the following lemma can be derived by considering the arrays as one dimensional vectors where each element is a row/column.

Lemma 3: For a positive integer m and two arrays $\mathbf{X} \in \Sigma_q^{m\times m}, \mathbf{Y} \in \Sigma_q^{m\times m}$,

$$\mathbb{D}_{1,0}(\mathbf{X}) \cap \mathbb{D}_{1,0}(\mathbf{Y}) \neq \emptyset \text{ if and only if } \mathbb{I}_{1,0}(\mathbf{X}) \cap \mathbb{I}_{1,0}(\mathbf{Y}) \neq \emptyset$$
$$\mathbb{D}_{0,1}(\mathbf{X}) \cap \mathbb{D}_{0,1}(\mathbf{Y}) \neq \emptyset \text{ if and only if } \mathbb{I}_{0,1}(\mathbf{X}) \cap \mathbb{I}_{0,1}(\mathbf{Y}) \neq \emptyset.$$

While the last lemma is derived from properties of vectors, the next one, albeit similar, requires a complete proof.

Lemma 4: For a positive integer m and two arrays $\mathbf{X} \in \Sigma_q^{(m+1)\times m}, \mathbf{Y} \in \Sigma_q^{m\times (m+1)}$,

$$\mathbb{D}_{1,0}(\mathbf{X}) \cap \mathbb{D}_{0,1}(\mathbf{Y}) \neq \emptyset \text{ if and only if } \mathbb{I}_{0,1}(\mathbf{X}) \cap \mathbb{I}_{1,0}(\mathbf{Y}) \neq \emptyset.$$

Proof: We show the "if" direction while the "only if" part is proved similarly. That is, we prove that if $\mathbb{D}_{1,0}(\mathbf{X}) \cap \mathbb{D}_{0,1}(\mathbf{Y}) \neq \emptyset$ then $\mathbb{I}_{0,1}(\mathbf{X}) \cap \mathbb{I}_{1,0}(\mathbf{Y}) \neq \emptyset$. Assume that there exists $\mathbf{D} \in \Sigma_q^{m\times m}$ such that $\mathbf{D} \in \mathbb{D}_{1,0}(\mathbf{X}) \cap \mathbb{D}_{0,1}(\mathbf{Y})$ and by contradiction assume that $\mathbb{I}_{0,1}(\mathbf{X}) \cap \mathbb{I}_{1,0}(\mathbf{Y}) = \emptyset$. Let i_R, i_C be the indices of the row and column deleted in $\mathbf{X}$ and $\mathbf{Y}$, respectively, to obtain $\mathbf{D}$. Let $\boldsymbol{r}$ denote row i_R of $\mathbf{X}$, i.e., $\mathbf{X}_{i_R,[m]}$, after an insertion of 0 in position i_C. Similarly, let $\boldsymbol{c}$ be the column $\mathbf{Y}_{[m],i_C}$ after an insertion of 0 in position i_R. Notice that it is also possible to insert 1 in both of the words, as long as the symbol inserted is the same. The following relations hold from the definition of $\mathbf{D}$.

$$\begin{aligned}
X_{i,j} &= D_{i,j} = Y_{i,j} \text{ for } 1 \leqslant i < i_R, 1 \leqslant j < i_C,\\
X_{i+1,j} &= D_{i,j} = Y_{i,j} \text{ for } i_R \leqslant i \leqslant m, 1 \leqslant j < i_C,\\
X_{i,j} &= D_{i,j} = Y_{i,j+1} \text{ for } 1 \leqslant i < i_R, i_C \leqslant j \leqslant m,\\
X_{i+1,j} &= D_{i,j} = Y_{i,j+1} \text{ for } i_R \leqslant i \leqslant m, i_C \leqslant j \leqslant m. \quad (1)
\end{aligned}$$

Let $\mathbf{I}^x$ be the result of inserting column $\boldsymbol{c}$ at index i_C into $\mathbf{X}$. The array $\mathbf{I}^y$ is defined similarly by inserting row $\boldsymbol{r}$ at index i_R in $\mathbf{Y}$. Notice that $\mathbf{I}^x$ is a result of inserting a column to $\mathbf{X}$ and thus $\mathbf{I}^x \in \mathbb{I}_{0,1}(\mathbf{X})$. For the same reasons it holds that $\mathbf{I}^y \in \mathbb{I}_{1,0}(\mathbf{Y})$. We conclude the proof by showing that $\mathbf{I}^x = \mathbf{I}^y$. This will be done by considering the following cases.

- For $i < i_R, j < i_C$, both $I^x_{i,j}, I^y_{i,j}$ are not affected by the insertions or deletions. Hence, it follows that
$$I^x_{i,j} = X_{i,j} = Y_{i,j} = I^y_{i,j}$$
- For $i = i_R$ and for $j < i_C$, the symbols $I^x_{i,j} = r_j$ remain unaffected by the insertion. On the other hand, $I^y_{i,j}$ is exactly an inserted symbol into $\mathbf{Y}$ that is defined to be r_j which results in $I^y_{i,j} = r_j = I^x_{i,j}$.
- For $i < i_R$ and for $j = i_C$, the symbols $I^y_{i,j} = c_i$ remain unaffected by the insertion. On the other hand, $I^x_{i,j}$ is exactly an inserted symbol into $\mathbf{X}$ that is defined to be c_i which results in $I^x_{i,j} = c_i = I^y_{i,j}$.
- For $i = i_R$ and for $j = i_C$, it holds that $I^x_{i,j} = c_i$ and $I^y_{i,j} = r_j$. By definition, both of these symbols are 0, which results in $I^x_{i,j} = I^y_{i,j}$.
- For $i > i_R$ and for $j < i_C$, by (1) it holds that
$$I^x_{i,j} = X_{i,j} = D_{i-1,j} = Y_{i-1,j}.$$
On the other hand, after a row insertion in index i_R, it holds that $I^y_{i,j} = Y_{i-1,j}$ which results in $I^x_{i,j} = I^y_{i,j}$.
- For $i > i_R$ and for $j = i_C$, by definition $I^x_{i,j} = c_i = Y_{i-1,j}$. On the other hand, $\mathbf{I}^y$ had a row insertion in index i_R, which means that $I^y_{i,j} = Y_{i-1,j}$ and results in $I^x_{i,j} = I^y_{i,j}$.
- For $i < i_R$ and for $j > i_C$, by (1) it holds that
$$I^y_{i,j} = Y_{i,j} = D_{i,j-1} = X_{i,j-1}.$$
On the other hand, after a column insertion in index i_C, it holds that $I^x_{i,j} = X_{i,j-1}$ which results in $I^x_{i,j} = I^y_{i,j}$.
- For $i = i_R$ and for $j > i_C$, by definition $I^y_{i,j} = r_j = X_{i,j-1}$. On the other hand, $\mathbf{I}^x$ had a column insertion in index i_C, which means that $I^x_{i,j} = X_{i,j-1}$ and results in $I^x_{i,j} = I^y_{i,j}$.
- For $i > i_R$ and for $j > i_C$, $\mathbf{I}^x$ had a column insertion in index i_C, which means that $I^x_{i,j} = X_{i,j-1}$. On the other hand, $\mathbf{I}^y$ had a row insertion in index i_R, which

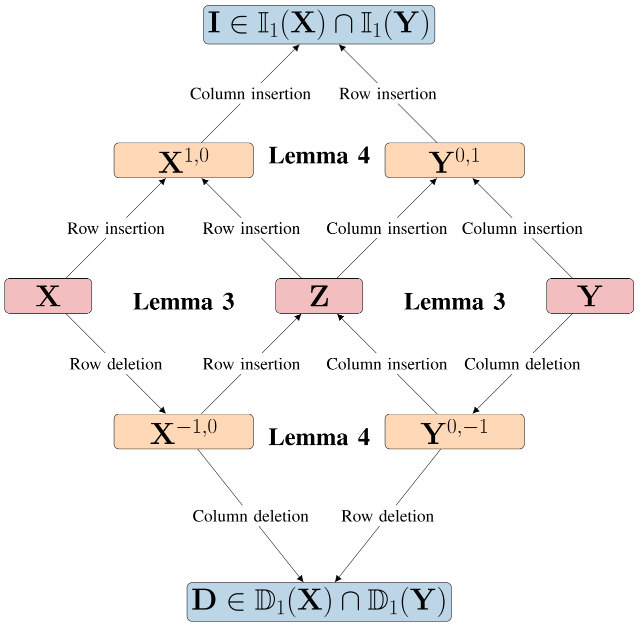

Fig. 1. A flowchart of the proof of Theorem 1.

means that $I^y_{i,j} = Y_{i-1,j}$. From (1) it holds that $X_{i,j-1} = D_{i-1,j-1}$ and $Y_{i-1,j} = D_{i-1,j-1}$. This results in

$$I^x_{i,j} = X_{i,j-1} = D_{i-1,j-1} = Y_{i-1,j} = I^y_{i,j}.$$

This concludes that for all $i, j \in [m+1]$, $I^x_{i,j} = I^y_{i,j}$, which assures that $\mathbf{I}^x = \mathbf{I}^y$, and hence $\mathbb{I}_{0,1}(\mathbf{X}) \cap \mathbb{I}_{1,0}(\mathbf{Y}) \neq \emptyset$, that contradicts our assumption. ■

We now use the results of Lemma 3 and Lemma 4 to prove Theorem 1.

Proof of Theorem 1: The proof follows by showing that for any $\mathbf{X}, \mathbf{Y} \in \Sigma_q^{n\times n}$, $\mathbb{D}_1(\mathbf{X}) \cap \mathbb{D}_1(\mathbf{Y}) = \emptyset$ if and only if $\mathbb{I}_1(\mathbf{X}) \cap \mathbb{I}_1(\mathbf{Y}) = \emptyset$. For the reader's convenience, a flowchart of the proof is presented in Figure 1. We only show the "only if" part as the "if" part follows similarly.

Assume that there exists an array $\mathbf{D} \in \Sigma_q^{(n-1)\times(n-1)}$ such that $\mathbf{D} \in \mathbb{D}_{1,1}(\mathbf{X}) \cap \mathbb{D}_{1,1}(\mathbf{Y})$. Hence, $\mathbf{D}$ can be obtained by deleting a row and then a column from $\mathbf{X}$ and by deleting a column and then a row from $\mathbf{Y}$ (note that the order of the row and column deletions does not matter and can be chosen arbitrarily).

Denote the intermediate arrays by $\mathbf{X}^{-1,0}, \mathbf{Y}^{0,-1}$, so the following relation holds.

$$\mathbf{X} \xrightarrow{\text{Row Deletion}} \mathbf{X}^{-1,0} \xrightarrow{\text{Col Deletion}} \mathbf{D},$$
$$\mathbf{Y} \xrightarrow{\text{Col Deletion}} \mathbf{Y}^{0,-1} \xrightarrow{\text{Row Deletion}} \mathbf{D}.$$

Hence, it holds that

$$\mathbf{D} \in \mathbb{D}_{1,0}(\mathbf{Y}^{0,-1}) \cap \mathbb{D}_{0,1}(\mathbf{X}^{-1,0}),$$

and thus, from Lemma 4 there exists an array $\mathbf{Z} \in \Sigma_q^{n\times n}$, such that $\mathbf{Z} \in \mathbb{I}_{0,1}(\mathbf{Y}^{0,-1}) \cap \mathbb{I}_{1,0}(\mathbf{X}^{-1,0})$. By definition, $\mathbf{Z} \in \mathbb{I}_{1,0}(\mathbf{X}^{-1,0})$ is equivalent to $\mathbf{X}^{-1,0} \in \mathbb{D}_{1,0}(\mathbf{Z})$. But, it is also known that $\mathbf{X}^{-1,0} \in \mathbb{D}_{1,0}(\mathbf{X})$, which means that

$$\mathbf{X}^{-1,0} \in \mathbb{D}_{1,0}(\mathbf{Z}) \cap \mathbb{D}_{1,0}(\mathbf{X}).$$

From Lemma 3 it follows that there exists some $\mathbf{X}^{1,0} \in \mathbb{I}_{1,0}(\mathbf{Z}) \cap \mathbb{I}_{1,0}(\mathbf{X})$. The same argument can be done for $\mathbf{Y}$, and define its result by $\mathbf{Y}^{0,1}$. Next, notice that we can also conclude that

$$\mathbf{Z} \in \mathbb{D}_{1,0}(\mathbf{X}^{1,0}) \cap \mathbb{D}_{0,1}(\mathbf{Y}^{0,1}),$$

so from Lemma 4 it is deduced that there exists an array $\mathbf{I} \in \mathbb{I}_{0,1}(\mathbf{X}^{1,0}) \cap \mathbb{I}_{1,0}(\mathbf{Y}^{0,1})$. Note that $\mathbf{I} \in \mathbb{I}_{0,1}(\mathbf{X}^{1,0})$ and $\mathbf{X}^{1,0} \in \mathbb{I}_{1,0}(\mathbf{X})$, which means $\mathbf{I}$ is obtained by inserting a row and a column in $\mathbf{X}$, i.e., $\mathbf{I} \in \mathbb{I}_1(\mathbf{X})$. A symmetrical argument holds for $\mathbf{Y}$, which assures that $\mathbf{I} \in \mathbb{I}_1(\mathbf{X}) \cap \mathbb{I}_1(\mathbf{Y})$. ■

We now prove Corollary 2 by using the result of Theorem 1.

Proof of Corollary 2: The proof follows by showing that for any $\mathbf{X}_1, \mathbf{X}_{t+1} \in \Sigma_q^{n\times n}$, $\mathbb{D}_t(\mathbf{X}) \cap \mathbb{D}_t(\mathbf{X}_{t+1}) \neq \emptyset$ if and only if $\mathbb{I}_t(\mathbf{X}) \cap \mathbb{I}_t(\mathbf{X}_{t+1}) \neq \emptyset$. We first prove the following claim.

Claim 5: For any two arrays $\mathbf{X}_1, \mathbf{X}_{t+1} \in \Sigma_q^{n\times n}$, $\mathbb{D}_t(\mathbf{X}_1) \cap \mathbb{D}_t(\mathbf{X}_{t+1}) \neq \emptyset$ if and only if there exist $t-1$ arrays $\mathbf{X}_2, \ldots, \mathbf{X}_t$ such that $\mathbb{D}_1(\mathbf{X}_i) \cap \mathbb{D}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$.

Proof: We prove the "if" part by induction. The proof of the "only if" part follows similarly and is omitted.

A. Base Case

We need to show that if $\mathbb{D}_1(\mathbf{X}_1) \cap \mathbb{D}_1(\mathbf{X}_2) \neq \emptyset$ then $\mathbb{D}_1(\mathbf{X}_i) \cap \mathbb{D}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $i = 1$ which follows from the assumption.

B. Induction Step

Assume the property holds for $t \in [n-2]$ and we show that the property holds for $t+1$. Let $\mathbf{X}_1, \mathbf{X}_{t+2}$ be such that $\mathbb{D}_{t+1}(\mathbf{X}_1) \cap \mathbb{D}_{t+1}(\mathbf{X}_{t+2}) \neq \emptyset$. Then, there exists $\mathbf{X}_1^{(1)}, \mathbf{X}_{t+1}^{(1)}$ resulting from a criss-cross deletion of $\mathbf{X}_1$ and $\mathbf{X}_{t+2}$, respectively, such that $\mathbb{D}_t(\mathbf{X}_1^{(1)} \cap \mathbb{D}_t(\mathbf{X}_{t+1}^{(1)}) \neq \emptyset$. Thus, according to the induction hypothesis, there exist $t-2$ arrays $\mathbf{X}_1^{(1)}, \ldots, \mathbf{X}_t^{(1)}$ that satisfy $\mathbb{D}_1(\mathbf{X}_i^{(1)}) \cap \mathbb{D}_1(\mathbf{X}_{i+1}^{(1)}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$.

According to Theorem 1, there exist t arrays $\mathbf{X}_2, \ldots, \mathbf{X}_{t+1}$ such that for all $2 \leqslant i \leqslant t+1$, $\mathbf{X}_i \in \mathbb{I}(\mathbf{X}_{i-1}^{(1)}) \cap \mathbb{I}(\mathbf{X}_i^{(1)})$. Therefore, it holds that for $1 \leqslant 1 \leqslant t+1$,

$$\mathbf{X}_i^{(1)} \in \mathbb{D}_1(\mathbf{X}_i) \cap \mathbb{D}_1(\mathbf{X}_{i+1}).$$

This completes the "if" part of the proof. ■

Next we prove a similar claim for the insertion case.

Claim 6: For any two arrays $\mathbf{X}_1, \mathbf{X}_{t+1} \in \Sigma_q^{n\times n}$, $\mathbb{I}_t(\mathbf{X}_1) \cap \mathbb{I}_t(\mathbf{X}_{t+1}) \neq \emptyset$ if and only if there exist $t-1$ arrays $\mathbf{X}_2, \ldots, \mathbf{X}_t$ such that $\mathbb{I}_1(\mathbf{X}_i) \cap \mathbb{I}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$.

The proof of Claim 6 is similar to the proof of Claim 5 and is thus given in Appendix A.

Having the results of Claim 5 and Claim 6, we can now prove Corollary 2 as follows. For any $\mathbf{X}_t, \mathbf{X}_{t+1} \in \Sigma_q^{n\times n}$, if $\mathbb{D}_t(\mathbf{X}_1) \cap \mathbb{D}_t(\mathbf{X}_{t+1}) \neq \emptyset$, then from Claim 5 we know that there exist $t-1$ arrays $\mathbf{X}_2, \ldots, \mathbf{X}_t$ such that $\mathbb{D}_1(\mathbf{X}_i) \cap \mathbb{D}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$. Then, according to Theorem 1, there exist t arrays $\mathbf{X}_1^{(1)}, \ldots, \mathbf{X}_t^{(1)}$ such that for all $1 \leqslant i \leqslant t$,

$$\mathbf{X}_i^{(1)} \in \mathbb{I}_1(\mathbf{X}_i) \cap \mathbb{I}_1(\mathbf{X}_{i+1}).$$

Finally, we can now apply Claim 6 to conclude that $\mathbb{I}_t(\mathbf{X}_1) \cap \mathbb{I}_t(\mathbf{X}_{t+1}) \neq \emptyset$. The "only if" part follows similarly. ■

V. Upper Bound on the Cardinality

In this section we prove a non-asymptotic upper bound on the cardinality of a (1)-criss-cross deletion correcting code. For an array $\mathbf{X} \in \Sigma_q^{n\times n}$, we denote by $\mathbf{X}^{i,j}$ the array obtained from $\mathbf{X}$ after deleting the i-th row and the j-th column. Let $\mathbf{X} \in \Sigma_q^{n\times n}$ and let $i_1, i_2, j_1, j_2, \delta \in [n]$ be such that $i_1 \leqslant i_2$ and $j_1 \leqslant j_2$. We define a *column run* of length $i_2 - i_1 + 1$ as a sequence of identical consecutive symbols in a column j_1, i.e., $X_{i_1,j_1} = X_{i_1+1,j_1} = \cdots = X_{i_2,j_1}$. We define a *row run* similarly. A *diagonal run* of length $\delta+1$ is a sequence of identical symbols situated on a diagonal of $\mathbf{X}$, i.e., $X_{i_1,j_1} = X_{i_1+1,j_1+1} = \cdots = X_{i_1+\delta,j_1+\delta}$. An *anti-diagonal run* of length $\delta+1$ is a sequence of identical symbols situated on an anti-diagonal of $\mathbf{X}$, i.e., $X_{i_1,j_1} = X_{i_1+1,j_1-1} = \cdots = X_{i_1+\delta,j_1-\delta}$. In Lemma 7 we give a necessary and sufficient condition that two different (1)-criss-cross deletions applied on an array $\mathbf{X}$ must satisfy to result in the same array $\mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$ for $(i_1,j_1) \neq (i_2,j_2)$. We start with an example that illustrates the idea of Lemma 7.

Example 1: Consider the following binary 9×9 array divided into nine 3×3 arrays structured as in Figure 2.

$$\mathbf{X} = \begin{bmatrix} \mathbf{X}_{\mathrm{I}} & \mathbf{X}_{\mathrm{T}} & \mathbf{X}_{\mathrm{II}} \\ \mathbf{X}_{\mathrm{L}} & \mathbf{X}_{\mathrm{C}} & \mathbf{X}_{\mathrm{R}} \\ \mathbf{X}_{\mathrm{III}} & \mathbf{X}_{\mathrm{B}} & \mathbf{X}_{\mathrm{IV}} \end{bmatrix} = \begin{bmatrix} 0&1&0&1&1&1&0&1&0\\ 1&1&1&0&0&0&1&0&0\\ 1&0&1&0&0&0&0&1&0\\ 0&1&1&1&0&1&0&1&0\\ 0&1&1&0&1&0&0&1&0\\ 0&1&1&0&0&1&0&1&0\\ 0&1&0&0&0&0&0&1&0\\ 1&1&1&1&1&1&1&0&0\\ 1&0&1&1&1&1&0&1&0 \end{bmatrix}.$$

It is easy to verify that deleting column 4 and row 4 or deleting column 6 and row 6 results in the same array, i.e., $\mathbf{X}^{4,4} = \mathbf{X}^{6,6}$. Let $(i_1,j_1) = (4,4)$ and $(i_2,j_2) = (6,6)$. The equality $\mathbf{X}^{4,4} = \mathbf{X}^{6,6}$ happens because: all rows of the arrays $\mathbf{X}_{\mathrm{T}} = \mathbf{X}_{[1,i_1-1],[j_1,j_2]} = \mathbf{X}_{[1,3],[4,6]}$ and $\mathbf{X}_{\mathrm{B}} = \mathbf{X}_{[i_2+1,n],[j_1,j_2]} = \mathbf{X}_{[7,9],[4,6]}$ are row runs; all the columns of the arrays $\mathbf{X}_{\mathrm{L}} = \mathbf{X}_{[i_1,i_2],[1,j_1-1]} = \mathbf{X}_{[4,6],[1,3]}$ and $\mathbf{X}_{\mathrm{R}} = \mathbf{X}_{[i_1,i_2],[j_2+1,n]} = \mathbf{X}_{[4,6],[7,9]}$ are column runs; and all the diagonals of $\mathbf{X}_{\mathrm{C}} = \mathbf{X}_{[i_1,i_2],[j_1,j_2]}$ are diagonal runs. Lemma 7 generalises this example to show that given an array $\mathbf{X}$ and two (1)-criss-cross deletions applied on $\mathbf{X}$, the equality $\mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$ for $(i_1,j_1) \neq (i_2,j_2) \in [n]\times[n]$ holds if and only if $\mathbf{X}$ has the structure described in this example.

Lemma 7: For $i_1, i_2, j_1, j_2 \in [n]$ such that $(i_1,j_1) \neq (i_2,j_2)$, we define $i_{\min} \triangleq \mathit{min}(i_1,i_2)$ and $i_{\max} \triangleq \mathit{max}(i_1,i_2)$ and assume w.l.o.g. that $j_1 \leqslant j_2$. For all $n \geqslant 3$ and $\mathbf{X} \in \Sigma_q^{n\times n}$, the equality $\mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$ holds if and only if the entries $X_{i,j}$ of $\mathbf{X}$ satisfy the following structure (illustrated in Figure 2 for the case $i_1 \leqslant i_2$).

$$\begin{array}{ll} i \in [1, i_{\min}-1], j \in [1, j_1-1]: & X_{i,j} \text{ is arbitrary,} \\ i \in [1, i_{\min}-1], j \in [j_2+1, n]: & X_{i,j} \text{ is arbitrary,} \\ i \in [i_{\max}+1, n], j \in [1, j_1-1]: & X_{i,j} \text{ is arbitrary,} \\ i \in [i_{\max}+1, n], j \in [j_2+1, n]: & X_{i,j} \text{ is arbitrary,} \\ i \in [1, i_{\min}], j \in [j_1, j_2-1]: & X_{i,j} = X_{i,j+1}, \\ i \in [i_{\max}+1, n], j \in [j_1, j_2-1]: & X_{i,j} = X_{i,j+1}, \\ i \in [i_{\min}, i_{\max}-1], j \in [1, j_1-1]: & X_{i,j} = X_{i+1,j}, \\ i \in [i_{\min}, i_{\max}-1], j \in [j_2+1, n]: & X_{i,j} = X_{i+1,j}, \end{array}$$
$$\begin{cases} i \in [i_{\min}, i_{\max}-1], j \in [j_1+1, j_2]: & X_{i,j} = X_{i+1,j+1} \text{ for } i_1 \leqslant i_2, \\ i \in [i_{\min}, i_{\max}-1], j \in [j_1, j_2-1]: & X_{i,j} = X_{i+1,j-1} \text{ for } i_1 > i_2. \end{cases}$$

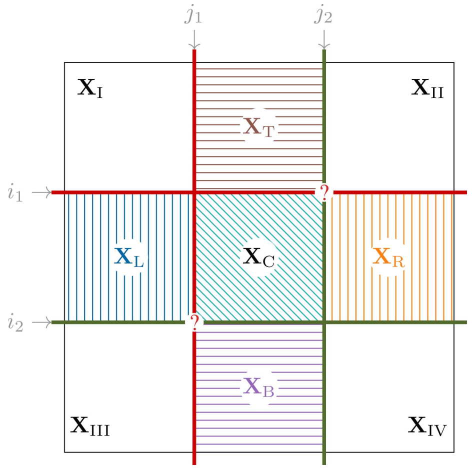
j_1
j_2
$\mathbf{X}_{\mathrm{I}}$
$\mathbf{X}_{\mathrm{II}}$
$\mathbf{X}_{\mathrm{T}}$
i_1
?
$\mathbf{X}_{\mathrm{L}}$
$\mathbf{X}_{\mathrm{C}}$
$\mathbf{X}_{\mathrm{R}}$
i_2
?
$\mathbf{X}_{\mathrm{B}}$
$\mathbf{X}_{\mathrm{III}}$
$\mathbf{X}_{\mathrm{IV}}$

Fig. 2. Required pattern for $\mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$. Let $(i_1,j_1) \neq (i_2,j_2)$ be the indices of the deleted row and column in two different criss-cross deletions on the array $\mathbf{X}$. W.l.o.g $j_1 < j_2$ and for case $i_1 < i_2$ the constraints are: *1)* Each row of the sub arrays $\mathbf{X}_{\mathrm{T}}$ and $\mathbf{X}_{\mathrm{B}}$ must be a row run of length $j_2 - j_1 + 1$ and each column of $\mathbf{X}_{\mathrm{L}}$ and $\mathbf{X}_{\mathrm{R}}$ must be a column run of length $i_2 - i_1 + 1$. *2)* Each diagonal of the sub array $\mathbf{X}_{\mathrm{C}}$ must be a diagonal run. *3)* The corner sub arrays $\mathbf{X}_{\mathrm{I}}$, $\mathbf{X}_{\mathrm{II}}$, $\mathbf{X}_{\mathrm{III}}$ and $\mathbf{X}_{\mathrm{IV}}$ are outside of the region affected by the criss-cross deletions. Therefore, no constraints are imposed on those sub arrays. The same holds for X_{i_1,j_2} and X_{i_2,j_1} since they are both deleted by criss-cross deletions. Note that for $i_1 > i_2$, all the requirements remain the same except for $\mathbf{X}_{\mathrm{C}}$. In this case, the bottom-left to top-right diagonals are diagonal runs.

Proof: For a better grasp of the proof we use the notation of Figure 2 for the sub arrays and the aforementioned notation of runs. Furthermore, for an array $\mathbf{X}$ we write $\mathbf{X}_{[:-1]}$ if all the elements of that array are shifted by one column to the left, i.e., $\mathbf{X}_{[:-1]} = \left[\mathbf{X}_{[n],2} | \mathbf{X}_{[n],3} | \cdots | \mathbf{X}_{[n],n}\right]$ where $[\cdot|\cdot]$ denotes a concatenation of arrays. Similarly, we write $\mathbf{X}_{[-1:]}$ for a row shift by one to the top and $\mathbf{X}_{[-1:-1]}$ for a simultaneous row and column shift.

We now assume that for $(i_1,j_1) \neq (i_2,j_2) \in [n]\times[n]$ there exists an array $\widetilde{\mathbf{X}} \in \Sigma_q^{(n-1)\times(n-1)}$ such that $\widetilde{\mathbf{X}} = \mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$. Note that w.l.o.g. we assume that $j_1 \leqslant j_2$.

Due to the assumption that $\mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$ we have $\mathbf{X}_{\mathrm{I}}^{i_1,j_1} = \mathbf{X}_{\mathrm{I}}^{i_2,j_2}$. The indices of the columns and rows of $\mathbf{X}_{\mathrm{I}}^{i_1,j_1}$ and $\mathbf{X}_{\mathrm{I}}^{i_2,j_2}$ satisfy $i < i_{\min}$ and $j < j_1$. Therefore the entries of those sub arrays are not affected by the criss-cross deletion. Hence, $\mathbf{X}_{\mathrm{I}}^{i_1,j_1} = \mathbf{X}_{\mathrm{I}}^{i_2,j_2} = \mathbf{X}_{\mathrm{I}}$ irrespective of the values of the entries of $\mathbf{X}$. For $i < i_{\min}$ and $j > j_2$, we have $\mathbf{X}_{\mathrm{II}}^{i_1,j_1} = \mathbf{X}_{\mathrm{II}}^{i_2,j_2} = \mathbf{X}_{\mathrm{II}[:-1]}$ irrespective of the values of the entries of $\mathbf{X}$. This equality holds because in both cases the columns of $\mathbf{X}_{\mathrm{II}}$ are shifted by one column to the left. Using a similar argument, one can show that $\mathbf{X}_{\mathrm{III}}^{i_1,j_1} = \mathbf{X}_{\mathrm{III}}^{i_2,j_2} = \mathbf{X}_{\mathrm{III}[-1:]}$ and $\mathbf{X}_{\mathrm{IV}}^{i_1,j_1} = \mathbf{X}_{\mathrm{IV}}^{i_2,j_2} = \mathbf{X}_{\mathrm{IV}[-1:-1]}$.

In contrast, from $\mathbf{X}_{\mathrm{T}}^{i_1,j_1} = \mathbf{X}_{\mathrm{T}}^{i_2,j_2}$ we get $\mathbf{X}_{\mathrm{T}}^{i_1,j_1} = \mathbf{X}_{\mathrm{T}[:-1]}$ and $\mathbf{X}_{\mathrm{T}}^{i_2,j_2} = \mathbf{X}_{\mathrm{T}}$ which produces the row run constraint $\mathbf{X}_{\mathrm{T}[:-1]} = \mathbf{X}_{\mathrm{T}}$, i.e., $X_{i,j} = X_{i,j+1}$ for all corresponding values of i and j. The same constraints hold for the equalities $\mathbf{X}_{\mathrm{B}}^{i_1,j_1} = \mathbf{X}_{\mathrm{B}[-1:-1]}$ and $\mathbf{X}_{\mathrm{B}}^{i_2,j_2} = \mathbf{X}_{\mathrm{B}[-1:]}$ following from the existence of $\widetilde{\mathbf{X}}$.

Furthermore, we observe that $\mathbf{X}_{\mathrm{L}}^{i_1,j_1} = \mathbf{X}_{\mathrm{L}[-1:]}$ and $\mathbf{X}_{\mathrm{L}}^{i_2,j_2} = \mathbf{X}_{\mathrm{L}}$ which produces the column run constraint $\mathbf{X}_{\mathrm{L}[-1:]} = \mathbf{X}_{\mathrm{L}}$, i.e., $X_{i,j} = X_{i+1,j}$ for all corresponding values of i and j. Once more, due to the existence of $\widetilde{\mathbf{X}}$ the same constraints holds due to $\mathbf{X}_{\mathrm{R}}^{i_1,j_1} = \mathbf{X}_{\mathrm{R}[-1:-1]}$ and $\mathbf{X}_{\mathrm{R}}^{i_2,j_2} = \mathbf{X}_{\mathrm{R}[:-1]}$.

In the center sub array $\mathbf{X}_{\mathrm{C}}$ we need to distinguish whether $i_1 \leqslant i_2$ or $i_2 > i_1$, since this imposes different constraints on $\mathbf{X}_{\mathrm{C}}$. For the first case we notice that $\mathbf{X}_{\mathrm{C}}^{i_1,j_1} = \mathbf{X}_{\mathrm{C}[-1:-1]}$ and $\mathbf{X}_{\mathrm{C}}^{i_2,j_2} = \mathbf{X}_{\mathrm{C}}$ which leads to a diagonal run constraint of $\mathbf{X}_{\mathrm{C}[-1:-1]} = \mathbf{X}_{\mathrm{C}}$, i.e., $X_{i,j} = X_{i+1,j+1}$ for all corresponding values of i and j. In the case where $i_2 > i_1$, we see that $\mathbf{X}_{\mathrm{C}}^{i_1,j_1} = \mathbf{X}_{\mathrm{C}[:-1]}$ and $\mathbf{X}_{\mathrm{C}}{}^{i_2,j_2} = \mathbf{X}_{\mathrm{C}[-1:]}$. Therefore we need the anti-diagonal run constraint, i.e., $X_{i,j} = X_{i+1,j-1}$ for all corresponding values of i and j.

The constraints imposed on the sub arrays are exactly the same as the structure imposed on the array $\mathbf{X}$ which concludes the first part of the proof.

The reverse statement follows by observing that an array $\mathbf{X}$ satisfying the structure described in the claim will result in $\widetilde{\mathbf{X}} = \mathbf{X}^{i_1,j_1} = \mathbf{X}^{i_2,j_2}$. The reason is that this structure makes the sub arrays invariant to the different shifts in $\mathbf{X}$ resulting from both (i_1, j_1) and (i_2, j_2) (1)-criss-cross deletions. ■

We use the following nomenclature throughout this section.

A. Good and Bad Arrays

An array $\mathbf{X} \in \Sigma_q^{n\times n}$ is called *good* if its deletion ball is larger than $\frac{2n^2}{5}$, i.e., $|\mathbb{D}_1(\mathbf{X})| \geqslant \frac{2n^2}{5}$ and $\mathbf{X}$ is called *bad* otherwise. Denote by $\mathcal{G}_n, \mathcal{B}_n$ the set of all good and bad arrays in $\Sigma_q^{n\times n}$, respectively.

B. Bad Columns and Rows

A column $\mathbf{X}_{[n],j}$, $j \in [2, n]$, is called *bad* if for any pair of row indices $i_1, i_2 \in [n]$ with $i_1 < i_2$ the columns $\mathbf{X}_{[n],j}$ and $\mathbf{X}_{[n],j-1}$ satisfy the following constraints:

1) They are identical in the intervals $[1, i_1 - 1]$ and $[i_2 + 1, n]$, i.e., $\mathbf{X}_{[i_1-1],j} = \mathbf{X}_{[i_1-1],j-1}$ and $\mathbf{X}_{[i_2+1,n],j} = \mathbf{X}_{[i_2+1,n],j-1}$.
2) The column $\mathbf{X}_{[n],j}$ is either identical to $\mathbf{X}_{[n],j-1}$ up to a single down shift in the interval $[i_1, i_2]$, i.e., $\mathbf{X}_{i+1,j} = \mathbf{X}_{i,j-1}$ for all $i \in [i_1, i_2 - 1]$; or identical to $\mathbf{X}_{[n],j-1}$ up to a single up shift in the interval $[i_1, i_2]$, i.e., $\mathbf{X}_{i-1,j} = \mathbf{X}_{i,j-1}$ for all $i \in [i_1 + 1, i_2]$.

For the case $i_1 = i_2$ a column $\mathbf{X}_{[n],j}$ is bad if it is identical to the column $\mathbf{X}_{[n],j-1}$, except for the bit i_1 which can have an arbitrary value, i.e., $\mathbf{X}_{[i_1-1],j} = \mathbf{X}_{[i_1-1],j-1}$ and $\mathbf{X}_{[i_1+1,n],j} = \mathbf{X}_{[i_1+1,n],j-1}$. Columns that do not satisfy the aforementioned constraints are referred to as *good* columns. Bad rows and good rows are defined similarly.

Claim 8: The deletion ball size of an array $\mathbf{X} \in \Sigma_q^{n\times n}$ is bounded from below by

$$|\mathbb{D}(\mathbf{X})| \geqslant |\mathcal{I}_{\mathbf{X}}^c||\mathcal{I}_{\mathbf{X}}^r|.$$

Thus, $\mathbf{X}$ is good if the number of good columns and the number of good rows are at least $\sqrt{\frac{2}{5}}n$, i.e., $|\mathcal{I}_{\mathbf{X}}^c| \geqslant \sqrt{\frac{2}{5}}n$ and $|\mathcal{I}_{\mathbf{X}}^r| \geqslant \sqrt{\frac{2}{5}}n$.

Proof: Let $c_g = |\mathcal{I}_{\mathbf{X}}^c|$ and $r_g = |\mathcal{I}_{\mathbf{X}}^r|$ be the number of good columns and good rows, respectively. For a column $\mathbf{X}_{[n],j}$, $j \in [n]$, the number of distinct arrays resulting from deleting the j-th column and any row $\mathbf{X}_{i,[n]}, i \in [n]$ is greater than or equal to r_g. In other words, deleting column $\mathbf{X}_{[n],j}$ and any good row $\mathbf{X}_{i_g,[n]}$, $i_g \in \mathcal{I}_{\mathbf{X}}^r$ gives a new distinct array. To see this, assume by contradiction that there exists a pair $i_1, i_2 \in \mathcal{I}_{\mathbf{X}}^r$ such that $\mathbf{X}^{i_1,j} = \mathbf{X}^{i_2,j}$. Then, according to Lemma 7, all rows $\mathbf{X}_{i_1,[n]}$ up to $\mathbf{X}_{i_2,[n]}$ must be identical. Thus we have a contradiction since $\mathbf{X}_{i_2,[n]} \neq \mathbf{X}_{i_2-1,[n]}$ by the definition of a good row.

We now turn our attention to good columns. For any $j_g \in \mathcal{I}_{\mathbf{X}}^c$, the arrays resulting from deleting column $\mathbf{X}_{[n],j_g}$ and any row are distinct. Assume by contradiction that there exist $j_2 \in \mathcal{I}_{\mathbf{X}}^c, j_2 \neq j_g$, and $i_1, i_2 \in [n]$ such that $\mathbf{X}^{i_1,j_g} = \mathbf{X}^{i_2,j_2}$. Let $j_m = \max\{j_g, j_2\}$. Then according to Lemma 7, the columns $\mathbf{X}_{[n],j_m-1}$ and $\mathbf{X}_{[n],j_m}$ must satisfy $X_{i,j_m-1} = X_{i,j_m}$ for all $i \in [1, \min\{i_1, i_2\} - 1] \cup [\max\{i_1, i_2\} + 1, n]$ and $\mathbf{X}_{[n],j_2}$ must be identical to $\mathbf{X}_{[n],j_2}$ up to a single up or down shift in the interval $[\min\{i_1, i_2\}, \max\{i_1, i_2\}]$ depending whether $i_1 > i_2$ or $i_2 > i_1$. However, this contradicts the definition of a good column.

Hence, $|\mathbb{D}(\mathbf{X})| \geqslant c_g r_g$. Thus, if $c_g \geqslant \sqrt{\frac{2}{5}}n$ and $r_g \geqslant \sqrt{\frac{2}{5}}n$, then $|\mathbb{D}(\mathbf{X})| \geqslant 2n^2/5$ and $\mathbf{X}$ is a good array. ■

Claim 9: For $n \geqslant 5$ the number of possible choices of a good column (or row) is equal to $\left(2^n - 2n^2\right)$.

Proof: For a column $\mathbf{X}_{[n],j_b}$ to be bad, for any $i \in [n]$ it could be identical to column $\mathbf{X}_{[n],j_b-1}$ on all entries except the i-th entry which can still be arbitrary, i.e., for all $i \in [n]$ it must hold that $\mathbf{X}_{[i-1],j_b} = \mathbf{X}_{[i-1],j_b-1}$ and $\mathbf{X}_{[i+1,n],j_b} = \mathbf{X}_{[i+1,n],j_b-1}$. There are $2n$ such columns. In addition, for two integers $i_1, i_2 \in [n]$ such that $i_1 < i_2$, a bad column $\mathbf{X}_{[n],j_b}$ could also be identical to $\mathbf{X}_{[n],j_b-1}$ on the intervals $[1, i_i - 1]$ and $[i_2 + 1, n]$, i.e., $\mathbf{X}_{[i_1-1],j_b} = \mathbf{X}_{[i_1-1],j_b}$ and $\mathbf{X}_{[i_2+1,n],j_b} = \mathbf{X}_{[i_2+1],j_b}$; and identical to column $\mathbf{X}_{[n],j_b-1}$ up to a single down shift on the interval $[i_1, i_2]$, i.e., $\mathbf{X}_{[i+1,j_b} = \mathbf{X}_{i,j_b-1}$ for all $i \in [i_1, i_2 - 1]$. We have $2\binom{n}{2}$ such columns. Similarly there are $2\binom{n}{2}$ bad columns resulting from being identical to $\mathbf{X}_{[n],j_b-1}$ up to a single up shift on the interval $[i_1, i_2]$. Therefore, the total number of bad columns is $b_n \triangleq 2n + 4\binom{n}{2} = 2n^2$ and the total number of good columns is equal to $2^{n^2} - 2n^2$. The same calculation holds for good and bad rows. ■

We are ready to give an upper bound on $|\mathcal{B}_n|$, the number of bad arrays.

Lemma 10: For $n \geqslant 41$ and $q \geqslant 2$ the number of bad arrays is bounded from above by

$$|\mathcal{B}_n| \leqslant \sqrt{\frac{8}{5}} \cdot q^{n^2-3n}.$$

Proof: If an array $\mathbf{X} \in \Sigma_q^{n\times n}$ satisfies the conditions of Claim 8, then it is a good array. Otherwise, $\mathbf{X}$ can be either a good array or a bad array. Therefore, we can compute an upper bound on the number of bad arrays as the number of arrays that do not satisfy the conditions of Claim 8, i.e., have either less than $\sqrt{\frac{2}{5}}n$ good columns or less than $\sqrt{\frac{2}{5}}n$ good rows. Thus, we can write

$$\begin{aligned}|\mathcal{B}_n| &\leqslant 2\sum_{j=n-\sqrt{\frac{2}{5}}n+1}^{n}\binom{n}{j}(b_n)^j q^{n\cdot(n-j)}\\ &\leqslant 2\sqrt{\frac{2}{5}}n2^n(2\,n^2)^n q^{\sqrt{\frac{2}{5}}n^2-n} = \sqrt{\frac{8}{5}}n(4\,n^2)^n q^{\sqrt{\frac{2}{5}}n^2-n}\\ &= \sqrt{\frac{8}{5}}q^{\sqrt{\frac{2}{5}}n^2-n+\log_q(n)+n\log_q(4n^2)}\\ &\leqslant \sqrt{\frac{8}{5}}q^{\sqrt{\frac{2}{5}}n^2-n+\log_2(n)+n\log_2(4n^2)}\\ &\leqslant \sqrt{\frac{8}{5}}\cdot q^{n^2-3n},\end{aligned}$$

where $b_n = 2n^2$ results from the observation of Claim 9. The upper bound can be interpreted as summing over all arrays with at least $n-\sqrt{\frac{2}{5}}n+1$ bad columns (or bad rows) which can be located at $\binom{n}{j}$ different positions. The other columns (rows) can be chosen arbitrarily. The second inequality is obtained by bounding $\binom{n}{j}$ with 2^n and the last inequality holds for $n \geqslant 41$. ■

We now use the upper bound on the number of bad arrays to prove the following lower bound on the redundancy of a criss-cross deletion correcting code.

Theorem 11: The cardinality of any q-ary (1)-criss-cross deletion correcting code $\mathcal{C}$ for $n \geqslant 41$ and $q \geqslant 2$ is bounded by

$$|\mathcal{C}| \leqslant (1+\varepsilon)\frac{q^{n^2}}{q^{2n-1}\cdot\frac{2n^2}{5}},$$

with $\varepsilon = 0.29$, and thus its redundancy is lower bounded by

$$R \geqslant 2n-3+2\log_q(n).$$

Proof: Let $\mathcal{C}_\mathcal{B} \triangleq \mathcal{C}\cap\mathcal{B}_n$ and $\mathcal{C}_\mathcal{G} \triangleq \mathcal{C}\cap\mathcal{G}_n$. Consider the following sphere packing argument

$$\frac{2n^2}{5}|\mathcal{C}_\mathcal{G}| \leqslant \sum_{\mathbf{X}\in\mathcal{C}_\mathcal{G}}|\mathbb{D}_1(\mathbf{X})| \leqslant \sum_{\mathbf{X}\in\mathcal{C}}|\mathbb{D}_1(\mathbf{X})| \leqslant q^{(n-1)^2}.$$

Hence, $|\mathcal{C}_\mathcal{G}| \leqslant \frac{q^{(n-1)^2}}{\frac{2n^2}{5}}$. From Lemma 10, for $n \geqslant 41$ the number of bad arrays is bounded by $|\mathcal{B}_n| \leqslant \sqrt{\frac{8}{5}}\cdot q^{n^2-3n}$. Thus,

$$\begin{aligned}|\mathcal{C}| &= |\mathcal{C}_\mathcal{G}|+|\mathcal{C}_\mathcal{B}|\\ &\leqslant |\mathcal{C}_\mathcal{G}|+|\mathcal{B}_n|\\ &\leqslant \frac{q^{(n-1)^2}}{\frac{2n^2}{5}}+\sqrt{\frac{8}{5}}\cdot q^{n^2-3n}\\ &= \frac{q^{n^2}}{q^{2n-1}\cdot\frac{2n^2}{5}}\left(1+\sqrt{\frac{32}{125}}\frac{n^2}{q^{n+1}}\right).\end{aligned} \tag{2}$$

We show next that for any $n \geqslant 5$ and $q = 2$ we have

$$f(n) \triangleq \sqrt{\frac{32}{125}}\frac{n^2}{q^{n+1}} < \varepsilon = 0.29.$$

Then we plug this result in (2) to obtain the bound on the cardinality of $\mathcal{C}$.

First, the only root of $f(n)$ is at $n = 0$. Second, we can find the only maximum at $f(\frac{2}{\log_e(2)}) = 0.2850$ and a saddle point at $f(0) = 0$. Therefore, for $n \geqslant 3$ the function is monotonically decreasing and is always greater than zero. Moreover, it holds for any $n \geqslant 3$ and $q > 2$

$$\sqrt{\frac{32}{125}}\frac{n^2}{q^{n+1}} < \sqrt{\frac{32}{125}}\frac{n^2}{2^{n+1}} \leqslant f\left(\frac{2}{\log_e(2)}\right) < \varepsilon = 0.29,$$

since $2^{n+1} < q^{n+1}$.

Consequently, we can re-write the upper bound in (2) with $n \geqslant 41$ and $\varepsilon = 0.29$ as

$$|\mathcal{C}| \leqslant (1+\varepsilon)\frac{q^{n^2}}{q^{2n-1}\cdot\frac{2n^2}{5}},$$

which coincides with the expression in the theorem.

For the lower bound on the redundancy of a criss-cross code, we calculate a bound on $R = n^2-\log_q(|\mathcal{C}|)$ as

$$\begin{aligned}R &\geqslant n^2-\log_q\left((1+\varepsilon)\frac{q^{n^2}}{q^{2n-1}\cdot\frac{2n^2}{5}}\right)\\ &= n^2-\log_q(1+\varepsilon)\\ &\quad -n^2+(2n-1)+2\log_q(n)-\log_q\left(\frac{5}{2}\right)\\ &\geqslant 2n-3+2\log_q(n),\end{aligned}$$

since $\log_q(1+\varepsilon)+\log_q(5/2) < \log_2(1.29)+\log_2(5/2) < 2$. ■

Corollary 12: For a (1)-criss-cross deletion correcting code $\mathcal{C}$ the following holds:

$$|\mathcal{C}| \leqslant \frac{q^{n^2}}{q^{2n-1}\cdot\frac{n^2}{2}}\left(1+\frac{n^2}{\sqrt{2}\cdot q^{n+1}}\right)$$

thus its asymptotic redundancy (in n) is at least $2n-2+2\log_q n$.

Proof: The derivations are similar to the ones in Theorem 11. We only change the definition of good arrays to have deletion balls greater than or equal to $n^2/2$. A complete proof is given in Appendix B. ■

VI. Construction

In this section we present our *CrissCross codes* that can correct a (1)-criss-cross deletion or insertion and state their main properties. Throughout the rest of the paper, we only consider binary arrays for the ease of presentation. The same construction can be extended for q-ary arrays. We denote the set of all binary arrays $\{0,1\}^{n\times n}$ by $\Sigma^{n\times n}$. Moreover, we assume that a,b,c,d are non-negative integers such that $0 \leqslant a,b,d \leqslant n-1$ and $0 \leqslant c \leqslant n-2$. We also assume that n is a power of 2 so that $\log n$ is an integer, while the extension for other values of n will be clear from the context. The main results of this section are summarized in the following theorem and corollary.

Theorem 13: The CrissCross code $\mathcal{C}_n(a,b,c,d)$ (defined in Construction 1) is a (1)-criss-cross deletion and insertion correcting code that has an explicit decoder.

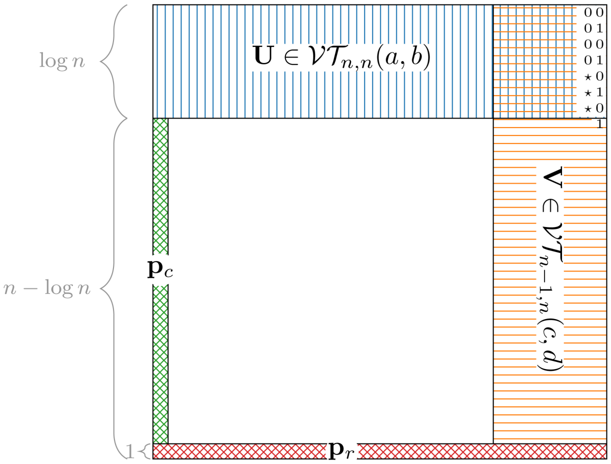
$\log n$
$n - \log n$
1
$\mathbf{U} \in \mathcal{VT}_{n,n}(a,b)$
$\mathbf{V} \in \mathcal{VT}_{n-1,n}(c,d)$
$\mathbf{p}_c$
$\mathbf{p}_r$
0 0
0 1
0 0
0 1
⋆ 0
⋆ 1
⋆ 0
1

Fig. 3. The structure of the codewords of our CrissCross code. $\mathbf{U}$ is the binary representation of a q-ary vector $\mathbf{u} \in \mathcal{VT}_{n,q}(a,b)$ with $q = n$. Each column is viewed as a symbol of the VT coded vector $\mathbf{u}$. The last column of $\mathbf{U}$ is an alternating sequence and the second to last column must start with four consecutive 0's. $\mathbf{V}$ is defined similarly to $\mathbf{U}$ where each row is a symbol of a VT coded vector $\mathbf{v} \in \mathcal{VT}_{n-1,n}(c,d)$. The alternating sequence of $\mathbf{U}$ is extended by one bit in $\mathbf{V}$. For the encoding of $\mathbf{V}$, we replace the alternating sequence by the all 0 sequence. The column $\mathbf{p}_c$ is a parity column consisting of the sum of all columns of its size (and position). The row $\mathbf{p}_r$ is a parity row defined similarly to $\mathbf{p}_c$. We denote by $\mathbf{X} \in \mathcal{VT}_{n,q}(a,b)$ the binary representation of a q-ary vector $\mathbf{x} \in \Sigma_q^n$, such that $\mathbf{x} \in \mathcal{VT}_{n,q}(a,b)$.

Corollary 14: There exist integers a, b, c, d for which the redundancy of the CrissCross code $\mathcal{C}_n(a,b,c,d)$ is at most

$$2n + \log n + 7 + 2\log e$$

bits and is therefore at most $2\log n + 10 + 2\log e$ bits away from the lower bound.

We prove Theorem 13 through a detailed explanation of the code construction that will be given in Section VI-A. In Section VI-B, we show how the decoding works for a (1)-criss-cross deletion and insertion. Afterwards, we compute an upper bound on the redundancy in Section VI-C, and thus prove Corollary 14.

A. The Construction

The CrissCross code $\mathcal{C}$ is an existential code whose codewords are $n \times n$ binary arrays structured as shown in Figure 3 and as explained next. The code consists of two main components. The columns and the rows are indexed using the binary expansion $\mathbf{U}$ and $\mathbf{V}$ of two n-ary VT coded vectors to recover the positions of the inserted/deleted row and column. The parity bits $\mathbf{p}_c$ and $\mathbf{p}_r$ are used to recover the deleted information in case of a deletion and to help detect the location of an inserted column or row in case of an insertion.

1) Indexing the Columns: The first $\log n$ rows of a codeword $\mathbf{C} \in \mathcal{C}$ are the binary representation of a q-ary vector $\mathbf{u}$ encoded using a VT code $\mathcal{VT}_{n,q}(a,b)$ that can correct one insertion/deletion, where $q = n$. The $\log n \times n$ binary array $\mathbf{U}$ satisfies the following requirements: *i)* every column of $\mathbf{U}$ is the binary representation of a symbol of the VT coded vector $\mathbf{u} \in \mathcal{VT}_{n,n}(a,b)$; *ii)* any two consecutive columns are different; *iii)* the last column is the alternating sequence that starts with 0; and *iv)* the first 4 bits of the second to last column are 0's. As we shall see in the decoding section, this array serves as an index of the columns. That is, it allows the decoder to exactly recover the position of the inserted/deleted column.

2) Indexing the Rows: The $(n-1) \times \log n$ array formed of the last $\log n$ bits of rows 1 to $n-1$ (situated at the right of the array $\mathbf{C}$) is the binary representation of a q-ary vector $\mathbf{v}$ encoded using a VT code $\mathcal{VT}_{n-1,q}(c,d)$ that can correct one insertion/deletion, with $q = n$. The $(n-1) \times \log n$ binary array $\mathbf{V}$ satisfies the following requirements: *i)* each row of $\mathbf{V}$ is the binary representation of a symbol of the VT coded vector $\mathbf{v} \in \mathcal{VT}_{n-1,n}(c,d)$; *ii)* any two consecutive rows are different; *iii)* the first $\log n$ rows also satisfy the requirements imposed on $\mathbf{U}$, with the exception of replacing the alternating sequence by the all 0 sequence[2]; and *iv)* the first bit below the alternating sequence is the opposite of the last bit of the alternating sequence. In other words, the alternating sequence is of length $\log n + 1$. Again here we assume that the stored bit belongs to the alternating sequence, but for the encoding of $\mathbf{v}$ we assume that the first $\log n + 1$ bits of the last column are all 0's. This array serves as an index of the rows that allows the decoder to recover the position of the inserted/deleted row.

3) Parities: The part of the first column of $\mathbf{C}$ that is not included in $\mathbf{U}$ is a parity of the same part of all corresponding columns, i.e., each entry of that column is the sum of all bits corresponding to its same row. This column is denoted by $\mathbf{p}_c$ and is shown on the left in Figure 3. Moreover, the last row of $\mathbf{C}$ is a parity of all the rows and is denoted by $\mathbf{p}_r$. In case of deletion, the parities allow the decoder to recover the information in the deleted column and row. In case of an insertion, the parities help the decoder to exactly recover the index of the inserted row and column in case the arrays $\mathbf{U}$ and $\mathbf{V}$ failed to do so, as explained in more details in the next section. We start with an example to illustrate the idea before going into the formal definition of the construction. The example also illustrates the deletion decoder.

Example 2: We construct a 9×9 codeword of our code to illustrate the construction and the decoding algorithm. We chose n to be 9 (not a multiple of 2) for convenience and to make the example simpler.

Assume that we want the columns and the rows to be indexed by codewords of q-ary VT codes with $q = 2^{(\lceil \log n \rceil)} = 16$ with $a = 2$, $b = 0$, $c = 7$ and $d = 0$. The first 4 rows of the codeword should then be the binary representation of a q-ary vector $\mathbf{u} \in \mathcal{VT}_{9,16}(2,0)$. For clarity of presentation, we represent a symbol $\mathbf{x} = (x_1, x_2, x_3, x_4)^T \in \Sigma_{2^4}$ as the decimal representation $x = \sum_{i=1}^{4} x_i 2^{i-1}$. Our construction requires the last symbol of $\mathbf{u}$ to be 10, i.e., its binary representation is the alternating sequence. In addition, the second to last symbol of $\mathbf{u}$ must be 0. Moreover, every two consecutive symbols of $\mathbf{u}$ must be different. An example is $\mathbf{u} = (0, 1, 2, 3, 4, 5, 11, 0, 10) \in \Sigma_{2^4}^9$. The binary representation $\mathbf{U}$ of $\mathbf{u}$ is the first four rows of $\mathbf{X}$ given in (3). Given $\mathbf{u}$,

[2] The array $\mathbf{U}$ can still store the alternating sequence. When checking the constraints on the rows of $\mathbf{V}$ we assume the last column of $\mathbf{U}$ is the all 0 sequence. Similarly, when encoding $\mathbf{V}$ we also assume that the last column is all 0. Since this information is known by the decoder, the all 0 sequence need not be stored in the array.

we now index the columns with a vector $\mathbf{v} \in \mathcal{VT}_{8,16}(7,0)$ such that the first four symbols of $\mathbf{v}$ are predetermined to be $3, 10, 1$ and 10, respectively. Our construction requires the last bit of the binary representation of the fifth symbol of $\mathbf{v}$ to be set to 0 as an extension of the alternating sequence of the last symbol of $\mathbf{u}$ (c.f. (3)). Similarly to $\mathbf{u}$, any two consecutive symbols of $\mathbf{v}$ must be different. An example is $\mathbf{v} = (3, 10, 1, 10, 6, 8, 9, 7) \in \Sigma_{2^4}^8$. The binary representation $\mathbf{V}$ of $\mathbf{v}$ is the transpose of the last four columns and first eight rows of the array $\mathbf{X}$ given in (3). The remaining entries of the array $\mathbf{X}$ not belonging to the first column nor the last row (marked in black) are arbitrary. The entries of the last row (marked in red) are the column-wise parity bits. The remaining entries of the first column (in green) are the row-wise parity bits. The constructed codeword $\mathbf{X}$ is shown in (3).

$$\mathbf{X} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad (3)$$

$$\mathbf{X}^{2,7} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}. \quad (4)$$

To illustrate the decoding strategy assume that column 7 and row 2 of $\mathbf{X}$ are deleted. The resulting array $\mathbf{X}^{2,7}$ is illustrated in (4). The decoder looks at the last non-deleted column and knows that it should be the alternating sequence because it is not the all zero column in the first four rows. From the last column, the decoder knows that the second row is deleted. Using the row-wise parity bits of the last row, the decoder can recover the second row. Now the decoder has all the rows of $\mathbf{U}$ with one deleted column and can thus use the VT decoder to recover the value and position of the lost column. Note that since every two consecutive columns in $\mathbf{U}$ are different, the decoder recovers the exact location of the deleted column. Having the index of the deleted column, the decoder recovers the values of the bits outside of $\mathbf{U}$ (rows 5 to 8) using the column-wise parity bits. The last bit of the deleted column is the sum of all other bits.

Formally, the CrissCross code $\mathcal{C}$ can be seen as an intersection of four codes over $\Sigma^{n\times n}$ that define the constraints imposed on the codewords of $\mathcal{C}$. Let $\ell \triangleq \log n$, we define $\mathbf{W}$ to be the all zero array except for the first $\ell + 1$ bits of the last column to be the alternating sequence, i.e., $\mathbf{W}_{[\ell+1],n} = [01010101\cdots]^T$ and $W_{i,j} = 0$ otherwise. We denote by $\mathbf{X} \in \mathcal{VT}_{n,q}(a,b)$ the binary representation of a q-ary vector $\mathbf{x} \in \Sigma_q^n$, such that $\mathbf{x} \in \mathcal{VT}_{n,q}(a,b)$.

$$\mathcal{U}(a,b) \triangleq \left\{ \mathbf{X} : \begin{array}{l} \mathbf{X}_{[\ell],j} \neq \mathbf{X}_{[\ell],j+1}, \quad j \in [n-1] \\ \mathbf{X}_{[4],n-1} = [0000]^T, \\ \mathbf{X}_{[\ell+1],n} = [010101\cdots]^T, \\ \mathbf{X}_{[\ell],[n]} \in \mathcal{VT}_{n,2^\ell}(a,b) \end{array} \right\},$$

$$\mathcal{V}(c,d) \triangleq \left\{ \mathbf{X} : \begin{array}{l} \mathbf{X}_{i,[n-\ell+1,n]} \neq \mathbf{X}_{i+1,[n-\ell+1,n]}, \ i \in [n-1] \\ \mathbf{X}_{[\ell+1],n} = [0000\cdots]^T, \\ \mathbf{X}^T_{[n-\ell+1,n],n-1} \in \mathcal{VT}_{n-1,2^\ell}(c,d) \end{array} \right\},$$

$$\mathcal{V}'(c,d) \triangleq \left\{ \mathbf{Y} : \begin{array}{l} \mathbf{Y}_{[\ell+1],n} = [010101\cdots]^T, \\ \mathbf{Y} \oplus \mathbf{W} \in \mathcal{V}(c,d) \end{array} \right\},$$

$$\mathcal{P}_c \triangleq \left\{ \mathbf{X} : x_{i,1} = \sum_{j=2}^{n} x_{i,j}, \quad i = \ell+1, \ldots, n-1 \right\},$$

$$\mathcal{P}_r \triangleq \left\{ \mathbf{X} : x_{n,j} = \sum_{i=1}^{n-1} x_{i,j}, \quad j \in [n] \right\}.$$

Construction 1: The CrissCross code $\mathcal{C}_n(a,b,c,d)$ is the set of arrays $\mathbf{C} \in \Sigma^{n\times n}$ that belong to

$$\mathcal{C}_n(a,b,c,d) \triangleq \mathcal{U}(a,b) \cap \mathcal{V}'(c,d) \cap \mathcal{P}_c \cap \mathcal{P}_r.$$

B. Decoder

In this section, we show how the CrissCross code construction leverages the structure of a codeword $\mathbf{C} \in \mathcal{C}_n(a,b,c,d)$ to correct a criss-cross deletion or insertion. Formally, we prove Theorem 13. We assume that the decoder knows the dimension of received array. In other words, the decoder knows whether a criss-cross deletion or a criss-cross insertion has happened and only needs to correct it.

Intuition: The goal of the decoder is to use the insertion/deletion correction capability of $\mathcal{VT}_{n,n}(a,b)$ and $\mathcal{VT}_{n-1,n}(c,d)$ to recover the positions of the inserted/deleted column and row. The decoder first uses the alternating sequence to check if a row of $\mathbf{U}$ is inserted/deleted and therefore corrects it before proceeding to the VT code decoder. In case of a deletion, the second to last column allows the decoder to detect whether the alternating sequence was deleted or not. If the alternating sequence is deleted, the decoder cannot use $\mathbf{U}$ and has to start using $\mathbf{V}$ to detect the position of the deleted row, recover it using the parities and then obtain the position of the deleted column from $\mathbf{U}$. The parties are used to recover the deleted information once the decoder has the position of the deleted row and/or column. In case of an insertion, the inserted vector may be equal to another consecutive vector in either $\mathbf{U}$ or $\mathbf{V}$. In this case, the decoder uses the parity bits over the remaining part of the vector to exactly recover the position of the inserted row or column. We are now ready to present the proof of Theorom 13.

Proof of Theorem 13: We split the proof into two parts: *i)* an explicit decoder for a (1)-criss-cross deletion; and *ii)* an explicit decoder for a (1)-criss-cross insertion.

1) Deletion Correcting Decoder: The decoder for $\mathcal{C}_n(a,b,c,d)$ receives as input an $(n-1)\times(n-1)$ array $\widetilde{\mathbf{C}}$ resulting from a (1)-criss-cross deletion in an array $\mathbf{C}$ of $\mathcal{C}_n(a,b,c,d)$ and works as follows. The decoder starts by looking at the first $\ell \times (n-1)$ subarray of $\widetilde{\mathbf{C}}$ and examining the last column.

Case 1: Assume the last column of $\mathbf{C}$ is not deleted. Using the alternating sequence, the decoder can detect whether or not there was a row deletion in $\mathbf{U}$ and locate its index. This is done by locating a run of length 2 in the alternating sequence. The last bit of the alternating sequence falling in $\mathbf{V}$ and not in $\mathbf{U}$ ensures that the decoder can detect whether the last row of $\mathbf{U}$ is deleted or not.

Case 1 (a): If there was a row deletion in $\mathbf{U}$, the decoder uses the non deleted part of $\mathbf{p}_r$ to recover the deleted row except for the bit in the deleted column. The decoder can now use the properties of $\mathcal{VT}_{n,n}(a,b)$ to decode the column deletion in $\mathbf{U}$. Since any two consecutive columns in $\mathbf{U}$ are different, the decoder can locate the exact position of the deleted column and recover its value. The position of the deleted column in $\mathbf{U}$ is the same as the deleted column in the whole array. Using $\mathbf{p}_c$, the decoder can now recover the remaining part of the deleted column.

Case 1 (b): If the deleted row was not in $\mathbf{U}$, the decoder uses $\mathcal{VT}_{n,n}(a,b)$ to recover the index of the deleted column and its value within $\mathbf{U}$ and uses $\mathbf{p}_c$ to recover the value of the deleted column outside of $\mathbf{U}$ (except for the bit in the intersection of the deleted row and column). Then, the decoder uses $\mathcal{VT}_{n-1,n}(c,d)$ to recover the index of the deleted row. Again, since any two consecutive rows in $\mathbf{V}$ are different the decoder can recover the exact position of the deleted row. Using $\mathbf{p}_r$, the decoder recovers the value of the bits of the deleted row.

Case 2: Now assume that the last column of $\mathbf{C}$ is deleted. By looking at the last column of $\widetilde{\mathbf{C}}$, the decoder knows that the alternating sequence is missing thanks to the run of 0's inserted in the beginning of the second to last column of $\mathbf{C}$. Note that irrespective of the location of the row deletion, the last column will have a run of at least three 0's which cannot happen in the alternating sequence. Therefore, the decoder knows that the last column is deleted and starts by looking at $\mathbf{V}$. Using the parity $\mathbf{p}_c$, the decoder recovers the missing part of the deleted column that is in $\mathbf{V}$ but not in $\mathbf{U}$. By construction, the first ℓ bits of the last column of $\mathbf{V}$ are set to 0 when encoding $\mathbf{V}$ using a VT code. Thus, the decoder recovers the whole missing column. By using the property of $\mathcal{VT}_{n-1,n}(c,d)$, the decoder recovers the index of the missing row and uses $\mathbf{p}_r$ to recover the value of the bits of this row. After recovering the deleted row the decoder adds the alternating sequence to $\mathbf{U}$ and recovers the whole array $\mathbf{C}$.

2) Insertion Correcting Decoder: The decoder for $\mathcal{C}_n(a,b,c,d)$ receives as input an $(n+1)\times(n+1)$ array $\widetilde{\mathbf{C}}$ resulting from a (1)-criss-cross insertion of an array $\mathbf{C}$ of $\mathcal{C}_n(a,b,c,d)$ and works as follows. The decoder starts by looking at the first $(\ell+1)\times(n+1)$ subarray of $\widetilde{\mathbf{C}}$ (recall that $\ell = \log n$) and examines the last two columns.

Case 1: Assume the second to last column is not an alternating sequence (special insertions that we consider in cases 2 and 3) and the last column is the alternating sequence. Using the alternating sequence, the decoder can detect whether or not there was a row insertion in $\mathbf{U}$. This is done by locating a run of length 2 in the alternating sequence.

Case 1 (a): If there was a row insertion in $\mathbf{U}$, the decoder has two candidates for the inserted row: the ones that cause the run of length 2 in the alternating sequence. Recall that the bit-wise sum of the n rows of $\mathbf{C}$ is known to the decoder (parity check constraint). The decoder verifies which of the two candidate rows does not satisfy the parity constraints, i.e., the decoder sums the $n-1$ remaining rows together with each of the candidate rows and checks the Hamming weight of the resulting vector. The row that results in a vector with Hamming weight more than 1 is the inserted row.[3] If both resulting vectors are different and result in Hamming weight 1, then the decoder is confused between two candidates for the inserted row and two candidates for the inserted column. In this case, the decoder deletes both candidate rows and both candidate columns where a "1" appears in the resulting vectors and uses the deletion correction capability of the code to recover the original message. This works since the inserted row and column were removed, i.e., the array is now affected by *one* row deletion and *one* column deletion.

Otherwise, the decoder removes the inserted row and uses the properties of $\mathcal{VT}_{n,n}(a,b)$ to decode the column insertion in $\mathbf{U}$. Since any two consecutive columns in $\mathbf{U}$ are different, the inserted column is either different from both adjacent columns or equal to only one of them. In the former case, the decoder recovers the exact position of the inserted column and removes it. The position of the inserted column in $\mathbf{U}$ is the same as the inserted column in the whole array. In the latter case, the decoder has two candidates of inserted columns. The decoder uses the column parity check to verify which column is the inserted one and removes it. Note that since the inserted row is removed, the decoder will have at most one column that does not satisfy the column parity check constraints. If both columns verify the parity constraints, then they are identical.

Case 1 (b): If the inserted row was not in $\mathbf{U}$, i.e., the alternating sequence is intact, the decoder uses $\mathcal{VT}_{n,n}(a,b)$ to recover the index and value of the inserted column in $\mathbf{U}$. If this column in $\mathbf{U}$ is different from both of its adjacent columns in $\mathbf{U}$, then the decoder removes the whole column and proceeds to correcting the inserted row. However, if the inserted column in $\mathbf{U}$ is equal to one of its adjacent columns (since any two consecutive columns are different), then the decoder has two candidates of inserted columns. In a similar way to Case 1 (a), the decoder uses the column parity check constraints to verify which column is the inserted one. After removing the inserted column, the decoder uses $\mathcal{VT}_{n-1,n}(c,d)$ to recover the index of the inserted row. Again, if the inserted row in $\mathbf{V}$ is different from both adjacent rows in $\mathbf{V}$, the decoder removes the whole row. Otherwise, the decoder has two candidates for the inserted rows; therefore the decoder uses the row parity check to recover the exact position of the inserted row.

Case 2: Now assume that the two last columns of $\mathbf{U}$ are identical. Due to the 4 zeros in the second to last column of $\mathbf{C}$ (now third to last column in $\widetilde{\mathbf{C}}$), the decoder detects that a column insertion happened in one of the last two columns of $\mathbf{U}$. The decoder uses the column parity check to verify

[3] The original row can only result in at most one 1 located in the position of the inserted column.

which column is the inserted one. In case both columns satisfy the parity check constraints, then they are identical. If both columns violate the parity check constraints in one position, similarly to Case 1 (a), the decoder deletes both columns and both rows where the columns do not satisfy the parity check constraint and uses the deletion correction capability of the code. After removing the inserted column, the decoder examines the alternating sequence to check if the inserted row is in $\mathbf{U}$. If this is the case, the decoder uses the row parity check to verify which row is inserted and removes it. If the inserted row is not in $\mathbf{U}$, the decoder uses $\mathcal{VT}_{n-1,n}(c,d)$ and the column parity check to recover the exact index of the inserted row.

Case 3: Assume that the last column of $\mathbf{U}$ is not the alternating sequence. Thus, the last column is an inserted column. The decoder removes this column and proceeds to detect which row is inserted as explained in the previous case. ■

C. Redundancy of the Code

The redundancy $R_{\mathcal{C}_n(a,b,c,d)}$ of $\mathcal{C}_n(a,b,c,d)$ is given by

$$\begin{aligned} R_{\mathcal{C}_n(a,b,c,d)} &= \log(2^{n^2}) - \log|\mathcal{C}_n(a,b,c,d)| \\ &= n^2 - \log|\mathcal{U}(a,b)\cap\mathcal{V}'(c,d)\cap\mathcal{P}_c\cap\mathcal{P}_r|\,. \end{aligned}$$

In this section we show that there exist a,b,c,d for which

$$R_{\mathcal{C}_n(a,b,c,d)} \leqslant 2n + 4\log n + 7 + 2\log e.$$

We do so by computing a lower bound on $\log|\mathcal{C}_n(a,b,c,d)|$. To that end we count the number of $n \times n$ binary arrays that satisfy all the requirements imposed on the codewords $\mathbf{C}$ in $\mathcal{C}_n(a,b,c,d)$.

Since the constraints imposed on the codes $\mathcal{U}(a,b)\cap\mathcal{V}'(c,d)$, $\mathcal{P}_c$, and $\mathcal{P}_r$ are disjoint, we have that

$$\begin{aligned} R_{\mathcal{C}_n(a,b,c,d)} &= R_{\mathcal{U}(a,b)\cap\mathcal{V}'(c,d)} + R_{\mathcal{P}_c} + R_{\mathcal{P}_r} \\ &= R_{\mathcal{U}(a,b)\cap\mathcal{V}'(c,d)} + 2n - \log n - 1. \qquad (5) \end{aligned}$$

Equation (5) follows from the fact that the $n-\log n-1$ bits of $\mathbf{p}_c$ and the n bits of $\mathbf{p}_r$ are fixed to predetermined values.

We now compute an upper bound on the redundancy of the set $\mathcal{U}(a,b)\cap\mathcal{V}'(c,d)$.

Proposition 15: There exists four values $a^\star, b^\star, c^\star$ and $d^\star$ for which the redundancy R_1 of $\mathcal{U}(a^\star,b^\star)\cap\mathcal{V}'(c^\star,d^\star)$ is bounded from above by

$$R_1 < (2n - 2\log n - 3)\log\left(\frac{n}{n-1}\right) + 5\log n + 6. \qquad (6)$$

From Equations (5) and (6) we obtain,

$$\begin{aligned} R_{\mathcal{C}_n(a,b,c,d)} &< (2n-2\log n-3)\log\left(\frac{n}{n-1}\right) \\ &\quad + 2n + 4\log n + 5 \\ &< 2n\log\left(\frac{n}{n-1}\right) + 2n + 4\log n + 5 \\ &< 2n + 4\log n + 5 + 2\log 2e \qquad (7) \\ &= 2n + 4\log n + 7 + 2\log e. \end{aligned}$$

In (7) we use the inequality $2n\log\left(\frac{n}{n-1}\right) \leqslant 2\log 2e$. This inequality follows from noting that $\left(1-\frac{1}{n}\right)^n$ is an increasing function of n that converges to $\frac{1}{e}$ and is always greater than or equal to $0.25 > \frac{1}{2e} = 0.1852$ for $n \geqslant 2$. The proof of Corollary 14 is now complete. We conclude this section with the proof of Proposition 15.

Proof of Proposition 15: We start with counting the number of arrays that satisfy all the imposed constraints except for the VT constraints in the codes $\mathcal{U}(a^\star,b^\star)$ and $\mathcal{V}'(c^\star,d^\star)$. To that end, we define the following three sets over $\Sigma^{n\times n}$.

$$\mathcal{U}_\perp \triangleq \left\{\mathbf{X} : \mathbf{X}_{[\ell],j} \neq \mathbf{X}_{[\ell],j+1},\quad j \in [n-\ell-1]\right\},$$

$$\mathcal{V}_\perp \triangleq \left\{\mathbf{X} : \begin{array}{l} \mathbf{X}_{i,[n-\ell+1,n]} \neq \mathbf{X}_{i+1,[n-\ell+1,n]},\quad \ell<i<n-1 \\ X_{\ell+1,n} \equiv \ell \mod 2 \end{array}\right\},$$

$$\mathcal{S}_\cap \triangleq \left\{\mathbf{X} : \begin{array}{l} \mathbf{X}_{[\ell],j} \neq \mathbf{X}_{[\ell],j+1},\quad n-\ell \leqslant j < n, \\ \mathbf{X}_{i,[n-\ell+1,n]} \neq \mathbf{X}_{i+1,[n-\ell+1,n]}, i \in [\ell], \\ \mathbf{X}_{[4],n-1} = [0000]^T, \\ \mathbf{X}_{[\ell],n} = [010101\cdots]^T \end{array}\right\}.$$

$\mathcal{U}_\perp$ is the set of all $n \times n$ arrays in which any two consecutive columns, from column 1 to $n-\ell$, are different when restricted to the first ℓ entries; $\mathcal{V}_\perp$ is the set of all $n \times n$ arrays in which the entry $X_{\ell+1,n}$ is fixed to a predetermined value and any two consecutive rows, from row $\ell+1$ to $n-1$, are different when restricted to the last ℓ entries; and $\mathcal{S}_\cap$ is the set of $n \times n$ arrays in which the $\ell \times \ell$ sub array ending at the last bit of the first row of the original array has distinct consecutive columns, distinct consecutive rows, the last row fixed to a predetermined value and the first 4 bits of the second to last column are also predetermined. $\mathcal{S}_\cap$ is also defined to guarantee that the first column of the $\ell \times \ell$ sub array is different from the ℓ entries of column $n-\ell$ and similarly to the last row.

Claim 16: The redundancies of $\mathcal{U}_\perp$ and $\mathcal{V}_\perp$ are respectively given by

$$\begin{aligned} R_{\mathcal{U}_\perp} &= (n-\log n-1)\log\left(\frac{n}{n-1}\right), \\ R_{\mathcal{V}_\perp} &= (n-\log n-2)\log\left(\frac{n}{n-1}\right) + 1. \end{aligned}$$

The intuition behind Claim 16 is that the first $\log n$ bits of any two consecutive columns of $\mathbf{U}$ (last $\log n$ bits of any two consecutive rows of $\mathbf{V}$) must be different. The proof of Claim 16 is given in Appendix C.

Claim 17: The redundancy of $\mathcal{S}_\cap$ is upper bounded by

$$R_{\mathcal{S}_\cap} < \log n + 5. \qquad (8)$$

The intuition behind Claim 17 is that with at most one bit of redundancy we can guarantee that every two consecutive rows and every two consecutive columns of the $\log n \times \log n$ square are different. The remaining $\log n + 4$ bits are due to the use of the alternating sequence and fixing four bits of the second to last column of the square. The proof of Claim 17 is given in Appendix C.

The remaining part of the proof is to count the number of arrays that satisfy the above requirements and have $\mathbf{U} \in$

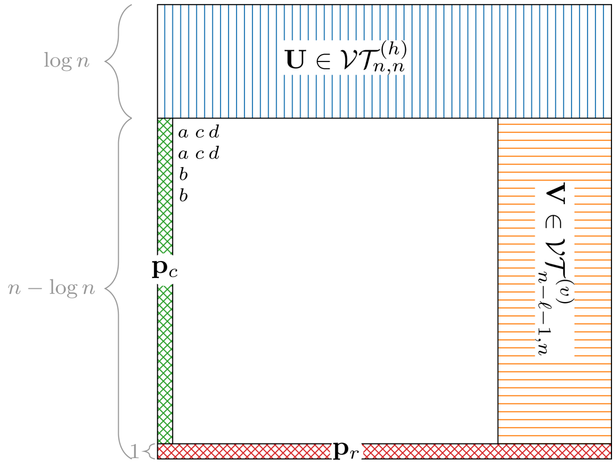
$\log n$
$\mathbf{U} \in \mathcal{VT}^{(h)}_{n,n}$
$a\ c\ d$
$a\ c\ d$
b
b
$\mathbf{p}_c$
$n - \log n$
$\mathbf{V} \in \mathcal{VT}^{(v)}_{n-\ell-1,n}$
1
$\mathbf{p}_r$

Fig. 4. The structure of the codewords of our CrissCross code with explicit encoder and decoder. $\mathbf{U}$ is the binary representation of a q-ary vector $\mathbf{u}$ encoded using an explicit VT code $\mathcal{VT}^{(h)}_{n,q}$ with $q = n$. Each column is viewed as a symbol of the VT coded vector $\mathbf{u}$. $\mathbf{V}$ is defined similarly to $\mathbf{U}$ where each row is a symbol of a VT coded vector $\mathbf{v}$ encoded using an explicit VT code $\mathcal{VT}^{(v)}_{n-\ell-1,n}$, where $\ell = \log n$. $\mathbf{p}_c$ is a parity column consisting of the sum of all columns of its size (and position). $\mathbf{p}_r$ is a parity row consisting of the sum of all rows. The first four bits of the second column below $\mathbf{U}$ (shown as a, a, b, b) are reserved to help the decoders of $\mathcal{VT}^{(h)}_{n,n}$ and $\mathcal{VT}^{(v)}_{n-\ell-1,n}$. The first two bits of the third and fourth columns below $\mathbf{U}$ (i.e., c, c, d, d) are reserved to help the decoder of $\mathcal{VT}^{(h)}_{n,n}$. Choosing the values of a, b, c and d will be clarified.

$\mathcal{VT}_{n,n}(a,b)$ and $\mathbf{V} \in \mathcal{VT}_{n-1,n}(c,d)$. Using the same arguments explained in Section II, we note that the VT constraints partition the set $\mathcal{U}_\perp \cap \mathcal{V}_\perp \cap \mathcal{S}_\cap$ into $(n^3)(n-1)$ disjoint cosets. Therefore, there exist $a^\star, d^\star, c^\star, d^\star$ for which

$$|\mathcal{U}(a^\star, b^\star) \cap \mathcal{V}(c^\star, d^\star)| \geqslant \frac{|\mathcal{U}_\perp \cap \mathcal{V}_\perp \cap \mathcal{S}_\cap|}{(n^3)(n-1)}.$$

In other words, the redundancy R_1 of $\mathcal{U}(a^\star, b^\star) \cap \mathcal{V}(c^\star, d^\star)$ is bounded from above by

$$R_1 \leqslant R_{\mathcal{U}_\perp \cap \mathcal{V}_\perp \cap \mathcal{S}_\cap} + \log\left((n^3)(n-1)\right).$$

Since all the constraints in $\mathcal{U}_\perp$, $\mathcal{V}_\perp$, $\mathcal{S}_\cap$ are disjoint by construction, we can rewrite the previous equation as

$$\begin{aligned} R_1 &\leqslant R_{\mathcal{U}_\perp} + R_{\mathcal{V}_\perp} + R_{\mathcal{S}_\cap} + \log\left(n^3(n-1)\right) \\ &< R_{\mathcal{U}_\perp} + R_{\mathcal{V}_\perp} + R_{\mathcal{S}_\cap} + 4\log n \qquad (9) \\ &\leqslant (2n - 2\log n - 3)\log\left(\frac{n}{n-1}\right) + 5\log n + 6. \qquad (10) \end{aligned}$$

In (10) we substituted the results from Claim 16 and Claim 17. ∎

VII. Construction With Explicit Encoder

In this section we show how to construct a CrissCross code with explicit encoder and decoder at the expense of increasing the redundancy by $5\log n + 5$ bits. The main idea is to change the arrays $\mathbf{U}$ and $\mathbf{V}$ so that they are the binary representations of two q-ary vectors $\mathbf{u}$ and $\mathbf{v}$, which are encoded using two variations of the explicit systematic non-binary VT codes from [29] and will be introduced in the sequel. The new structure of the codewords is depicted in Figure 4. In the remaining of this section we also take $\log n$ to be an integer. The main result of this section is stated in the next theorem.

Theorem 18: The CrissCross code defined below, constructed by modifying Construction 1, is a (1)-criss-cross deletion and insertion correcting code that has explicit encoder and decoder. The redundancy of this code is bounded from above by

$$R_{\text{explicit}} < 2n + 9\log n + 12 + 2\log e.$$

A. Construction

We first review the non-binary systematic VT construction from [29].

1) Systematic VT Code: In [29] Tenengolts presented two VT code constructions: an existential construction as in Section II, i.e., one defines the constraints on the codewords and shows that such a code exists; and a systematic construction that takes as input a message and only adds redundancy to it such that the resulting codeword satisfies some imposed constraints. Before going into the details of our construction, we explain the construction of the systematic VT code as presented in [29].

The systematic VT code, denoted by $\mathcal{VT}^\star_{n,q}$, takes as input a message $\mathbf{a} = (a_1, \dots, a_k) \in \Sigma_q^k$ and encodes it into a vector $\mathbf{c} \in \Sigma_q^n$ where[4] $n = k + 3 + \lceil \log_q k \rceil$. Since in our case we take $q = n$, we explain here the construction of $\mathcal{VT}^\star_{n,n}$, and $n = k + 4$. Given the message $\mathbf{a}$, the encoded vector $\mathbf{c} = (c_1, \dots, c_n) \in \Sigma_n^n$ of $\mathcal{VT}^\star_{n,n}$ is constructed as follows.

1) The first k symbols of $\mathbf{c}$, referred to as the *systematic data part*, are the same as the first k symbols of $\mathbf{a}$, i.e., $c_i = a_i$ for $i = 1, \dots, k$.
2) The symbols c_{k+1} and c_{k+2} satisfy $c_{k+1} = c_{k+2} = a_k + 1 \mod n$.
3) To compute c_{k+3}, the signature vector $\mathbf{s} = (s_1, \dots, s_k)$ is computed as $s_1 = 1$ and
$$s_i = \begin{cases} 1 & \text{if } a_i \geqslant a_{i-1} \\ 0 & \text{otherwise.} \end{cases}$$
The symbol[5] c_{k+3} is then equal to $\sum_{i=1}^k (i-1)s_i \mod k$.
4) The symbol c_{k+4} is computed as $c_{k+4} = \sum_{i=1}^k c_i \mod n$.

The symbols c_{k+3} and c_{k+4} are referred to as the *parity symbols*. The symbols c_{k+1} and c_{k+2} are used as separators between the data part and the parity part so that the decoder can localize the insertion/deletion. Note that they can have other values besides $a_k + 1 \mod n$ as long as they are different from a_k. We will use this variation in our construction. If the insertion/deletion happens in the data part, the decoder uses c_{k+3} and c_{k+4} together with the same VT decoder explained in [29] to decode the insertion/deletion. Otherwise, the data part is intact and no decoding is needed.

[4] In the construction by Tenengolts, three extra symbols are added at the end of the sequence to account for the case of sending several concatenated codewords. We do not need those symbols here as only one array is sent through the channel.

[5] In the general $\mathcal{VT}^\star_{n,q}$ where $q < n$, one needs $r = \lceil \log_q k \rceil$ symbols $c_{k+3}, \dots, c_{k+3+r}$ to be the q-ary representation of the equal to $\sum_{i=1}^k (i-1)s_i \mod k$.

2) Encoding of $\mathbf{U}$ *and* $\mathbf{V}$*:* We slightly modify the systematic VT code to fit our setting. Namely, for the vector $\mathbf{u}$ (used to compute the array $\mathbf{U}$) we put the systematic part of the data in the end of the sequence and the parity part in the beginning. For the vector $\mathbf{v}$, we maintain the structure of the systematic VT code. For both vectors $\mathbf{u}$ and $\mathbf{v}$ we pre-encode the message so that every two consecutive symbols of the vectors $\mathbf{u}$ and $\mathbf{v}$ are different. Thus, the construction will not be systematic, but explicit. Furthermore, we also change the separator symbols to better fit our setting. We require any two consecutive symbols to be different to detect a row deletion within $\mathbf{U}$. These modifications require also adding one more redundancy symbol.

a) Horizontal VT encoder $\mathcal{VT}^{(h)}_{n,n}$*:* Consider the message $\mathbf{a} = (a_1, \ldots, a_k) \in \{1, \ldots, n-1\}^k$ to be encoded into the vector $\mathbf{u} \in \Sigma_n^n$. Here we take $n = k + 5$. For notational convenience we number the indices of $\mathbf{u}$ from -4 to $n-5$, i.e., $\mathbf{u} = (u_{-4}, u_{-3}, \ldots, u_{n-5})$. The vector $\mathbf{u}$ is constructed as follows.

1) To guarantee that every two consecutive symbols are different, the symbol u_1 is made equal to a_1 and the symbols u_2 to u_{n-5} are computed as $u_i = u_{i-1} + a_i \mod n$.
2) The symbol u_0 can take an arbitrary value up to the restrictions explained in the sequel.
3) The symbol u_{-4} is computed as $u_{-4} = \sum_{i=0}^{n-5} u_i \mod n$.
4) To compute u_{-3}, we compute the signature vector $\mathbf{s} = (s_1, \ldots, s_{k+1})$ as $s_1 = 1$ and for $i = 2, \ldots, n$
$$s_i = \begin{cases} 1 & \text{if } u_{i-1} \geqslant u_{i-2} \\ 0 & \text{otherwise.} \end{cases}$$
The symbol u_{-3} is then equal to $\sum_{i=1}^{k+1}(i-1)s_i \mod (k+1)$.
5) The symbol u_{-2} is the n-ary value of a length $\log n$ alternating sequence and u_{-1} is chosen as the complement alternating sequence of the binary representation of u_{-2}.
6) Since for our CrissCross code we need any two consecutive symbols to be different, $u_i \neq u_{i\pm 1}$ for all i, we choose the value of u_0 to be different from u_{-1} and u_1 such that it ensures that $u_{-4} \neq u_{-3}$ and $u_{-3} \neq u_{-2}$. This is proved in the next claim.

Claim 19: A value for u_0 that satisfies $u_0 \neq u_{-1}$, $u_0 \neq u_1$ and makes $u_{-4} \neq u_{-3}$ and $u_{-3} \neq u_{-2}$ always exists.

The intuition behind Claim 19 is that u_{-3} changes if u_0 is smaller or greater than u_1 (2 choices), whereas u_{-4} changes with the value of u_0 ($n-2$ choices). A detailed proof is given in Appendix D.

We refer to this encoding procedure as the *horizontal VT encoder* and is denoted by $\mathcal{VT}^{(h)}_{n,n}$.

b) Vertical VT encoder $\mathcal{VT}^{(v)}_{n-\ell-1,n}$*:* Consider the message $\mathbf{b} = (b_1, \ldots, b_{k'}) \in \{1, \ldots, n-1\}^{k'}$ to be encoded into the vector $\mathbf{v}$ of length $n' = n - \ell - 1$. Here $n' = n - \ell - 1 = k' + 5$. The vector $\mathbf{v} \in \Sigma_n^{n'}$ is encoded similarly to $\mathbf{u}$ except for the ordering of the data part and the parity part.

1) Let $v_1 = b_1$ and $v_i = v_{i-1} + b_i \mod n$ for $i = 1, \ldots, k'$.
2) The symbol $v_{k'+1}$ can take an arbitrary value up to the constraints explained next.
3) We let $v_{k'+2}$ and $v_{k'+3}$ be the n-ary values of two length $\log n$ complement alternating sequences.
4) Computing the signature $\mathbf{s}'$ of the vector $(v_1, \ldots, v_{k'+1})$, we let $v_{k'+4}$ be equal to $\sum_{i=1}^{k'+1}(i-1)s'_i \mod (k'+1)$. and $v_{k'+5} = \sum_{i=1}^{k'+1} v_i \mod n$.
5) We choose $v_{k'+1}$ to be different from $v_{k'}$ and $v_{k'+2}$ such that $v_{k'+3} \neq v_{k'+4}$ and $v_{k'+4} \neq v_{k'+5}$. By Claim 19, such a value of $v_{k'+1}$ always exists.

We refer to this encoding procedure as the *vertical VT encoder* and is denoted by $\mathcal{VT}^{(v)}_{n-\ell-1,n}$.

B. Encoder

We are now ready to explain our explicit encoder for the CrissCross code construction. The encoder takes as input $n_1 + n_2 + n_3$ bits and encodes them as follows, where

$$n_1 = n^2 - 2n - 9\log n - (2n - \log n - 11)\log(n) - 8,$$
$$n_2 = \lfloor (n-5)\log(n-1) \rfloor,$$
$$n_3 = \lfloor (n - \log n - 6)\log(n-1) \rfloor.$$

1) The first $\lfloor (n-5)\log(n-1) \rfloor$ bits are encoded using the horizontal VT encoder $\mathcal{VT}^{(h)}_{n,n}$ and we let $\mathbf{U}$ be the binary array representation of the resulting vector.
2) The next $\lfloor (n - \log n - 6)\log(n-1) \rfloor$ bits are encoded using the vertical VT encoder $\mathcal{VT}^{(v)}_{n-\ell-1,n}$ and let $\mathbf{V}$ be the binary array representation of the transpose of the resulting vector.
3) The first bit of the alternating sequence representing u_{-2} is repeated in the second column in the first and second row below $\mathbf{U}$. This bit is shown as a in Figure 4. The first bit of the alternating sequence representing $v_{n-\ell+1}$ is repeated in the second column in the third and fourth row below $\mathbf{U}$. This bit is shown as b in Figure 4.
4) The alternating sequences representing u_{-2} and u_{-1} are extended by 1 bit each. This bit is then repeated in the row below. Those bits are shown in Figure 4 as c and d, respectively.
5) The remaining $n^2 - 2n - 9\log n - (2n - \log n - 11)\log(n) - 8$ bits are systematically distributed in the $n \times n$ array outside of $\mathbf{U}$, $\mathbf{V}$, the positions of the parity check bits, and the eight reserved bits (shown in Figure 4).
6) The parity check bits are then computed as the respective column-wise and row-wise sums of all the bits.

C. Decoder

The decoder works exactly the same as explained in Section VI-B where the alternating sequence is now the third column of $\mathbf{U}$ rather than the last column of $\mathbf{U}$ (even after either u_{-2} or u_{-1} is deleted). For completeness, we explain the subtle details of decoding deletions in $\mathbf{U}$ and $\mathbf{V}$. The insertion case follows similarly.

The decoder first examines the received version of $\mathbf{U}$. To check whether a row of $\mathbf{U}$ is deleted, the decoder checks the alternating sequences (or one of them if the other is deleted). If a row is deleted, the alternating sequence must have a run of length 2, unless the first row is deleted. If no run of length 2 exist, the decoder simply counts the length of the alternating sequence to check if the first row is deleted. The decoder is guaranteed to count the exact length of the alternating sequence thanks to extending the sequence by one bit and repeating that bit. If a row of $\mathbf{U}$ is deleted, the decoder uses the row parity check to recover the value of the deleted row.

After checking for (and correcting) deleted rows in $\mathbf{U}$, the decoder checks for deleted columns. If both alternating sequences are not deleted and are in their correct positions, then the deleted column is in the systematic data part. The decoder uses the detailed decoding of [29] to recover the value and the position of the deleted column. If both alternating sequences are not deleted and are not in their correct positions, then the deleted column happened in the first two columns that are function of the systematic data part. The decoder can then recompute the parity part from the systematic data part and recover the index of the deleted column. In case one of the alternating sequences is deleted, then the decoder needs to know whether u_{-2} or u_{-1} is deleted. To that end, the decoder verifies the first bit of the non deleted alternating sequence with the bit in the second column and first row below $\mathbf{U}$. Thus, the decoder recovers the index of the deleted column.

$\mathbf{V}$ is decoded similarly, except that the decoder would have recovered the index and value of the deleted column.

D. Redundancy

The redundancy of the explicit code is given by

$$\begin{aligned} R_{\text{explicit}} &= n^2 - (n_1 + n_2 + n_3) \\ &= 2n + 9\log n + (2n - \log n - 11)\log(n) + 8 \\ &\quad - \lfloor (n-5)\log(n-1) \rfloor \\ &\quad - \lfloor (n - \log n - 6)\log(n-1) \rfloor \\ &\overset{(a)}{<} 2n + 9\log n + (2n - \log n - 11)\log(n) + 8 \\ &\quad - (n-5)\log(n-1) + 1 \\ &\quad - (n - \log n - 6)\log(n-1) + 1 \end{aligned}$$

In inequality (a) we used the inequality $\lfloor a \rfloor > a - 1$.

Using the fact that $(2n - \log n - 11)\log\left(\frac{n}{n-1}\right)$ is less than $2n\log\left(\frac{n}{n-1}\right)$ which is less than or equal to $2\log 2e = 2 + 2\log e$, we can write

$$\begin{aligned} R_{\text{explicit}} &= 2n + 9\log n + (2n - \log n - 10)\log\left(\frac{n}{n-1}\right) + 10 \\ &< 2n + 9\log n + 12 + 2\log e. \end{aligned}$$

VIII. Conclusion

This paper considers the problem of criss-cross insertion/deletion in an $n \times n$ array. We have shown that every (t)-criss-cross deletion correcting code is a (t)-criss-cross insertion correcting code by extending the equivalence between insertion and deletion correcting codes from the one-dimensional case to the considered two-dimensional case.

We derived a bound which shows that the redundancy of any (1)-criss-cross deletion/insertion correcting code is bounded from below by $2n - 3 + 2\log n$ for $n \geqslant 41$. We then constructed CrissCross code. This code can correct a single row and single column deletion in an $n \times n$ array. The redundancy of the CrissCross code is bounded from above by $2n + 4\log n + 7 + 2\log e$ bits. We have presented an explicit decoder for correcting deletions and insertions with this CrissCross code. We also modified this code construction to an explicit construction that has an explicit *encoder* and an explicit decoder. The explicit encoder is based on systematic VT codes and comes at the expense of increasing the redundancy of the code by $5\log n + 5$ bits.

In this work, we have considered deletions of one row *and* one column. Although our CrissCross code can correct a more general type of deletions, our bound on the redundancy and the equivalence proof do not directly hold in the more general model. Thus, as a future research direction, we are interested in investigating the case where any combination of t_r rows and t_c columns, such that $t_r + t_c$ is equal to a predetermined constant t, can be deleted or inserted. Another open problem of interest is also the case of mixed errors in which any t_c columns may be deleted or inserted and any t_r rows may be inserted or deleted. We expect the techniques presented in this work to provide valuable insights on solving the more general problem. Preliminary results can be found in [30].

Appendix A
Proof of Claim 6

We prove that for any two arrays $\mathbf{X}_1, \mathbf{X}_{t+1} \in \Sigma_q^{n\times n}$, $\mathbb{I}_t(\mathbf{X}_1) \cap \mathbb{I}_t(\mathbf{X}_{t+1}) \neq \emptyset$ if and only if there exist $t-1$ arrays $\mathbf{X}_2, \ldots, \mathbf{X}_t$ such that $\mathbb{I}_1(\mathbf{X}_i) \cap \mathbb{I}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$.

We prove the "if" part by induction. The proof of the "only if" part follows similarly and is omitted.

A. Base Case

We need to show that if $\mathbb{I}_1(\mathbf{X}_1) \cap \mathbb{I}_1(\mathbf{X}_2) \neq \emptyset$ then $\mathbb{I}_1(\mathbf{X}_i) \cap \mathbb{I}_1(\mathbf{X}_{i+1}) \neq \emptyset$ for all $i = 1$ which follows from the assumption.

B. Induction Step

Assume the property holds for $t \in [n-2]$ and we show that the property holds for $t+1$. Let $\mathbf{X}_1, \mathbf{X}_{t+2}$ be such that $\mathbb{I}_{t+1}(\mathbf{X}_1) \cap \mathbb{I}_{t+1}(\mathbf{X}_{t+2}) \neq \emptyset$. Then, there exists $\mathbf{X}_1^{(1)}, \mathbf{X}_{t+1}^{(1)}$ resulting from a criss-cross insertion of $\mathbf{X}_1$ and $\mathbf{X}_{t+2}$, respectively, such that $\mathbb{I}_t(\mathbf{X}_1^{(1)}) \cap \mathbb{I}_t(\mathbf{X}_{t+1}^{(1)}) \neq \emptyset$. Thus, according to the induction hypothesis, there exist $t-2$ arrays $\mathbf{X}_1^{(1)}, \ldots, \mathbf{X}_t^{(1)}$ that satisfy $\mathbb{I}_1(\mathbf{X}_i^{(1)}) \cap \mathbb{I}_1(\mathbf{X}_{i+1}^{(1)}) \neq \emptyset$ for all $1 \leqslant i \leqslant t$.

According to Theorem 1, there exist t arrays $\mathbf{X}_2, \ldots, \mathbf{X}_{t+1}$ such that for all $2 \leqslant i \leqslant t+1$, $\mathbf{X}_i \in \mathbb{D}(\mathbf{X}_{i-1}^{(1)}) \cap \mathbb{D}(\mathbf{X}_i^{(1)})$. Therefore, it holds that for $1 \leqslant i \leqslant t+1$,

$$\mathbf{X}_i^{(1)} \in \mathbb{I}_1(\mathbf{X}_i) \cap \mathbb{I}_1(\mathbf{X}_{i+1}).$$

This completes the "if" part of the proof. ■

APPENDIX B
PROOF OF COROLLARY 12

We want to prove that when n goes to infinity the redundancy of a criss-cross deletion correcting code is bounded from below by $2n-2+2\log_q n$.

To that end, we redefine a good array $\mathbf{X}$ to have a deletion ball greater than or equal to $n^2/2$. From Claim 8 we know that if an array $\mathbf{X}$ has more than $n/\sqrt{2}$ good rows and $n/\sqrt{2}$ good columns, then $\mathbf{X}$ is good.

Following the same steps of Lemma 10 we can bound the number of bad arrays (following this new definition) as

$$\begin{aligned}|\mathcal{B}_n| &\leqslant 2 \sum_{j=n-\frac{n}{\sqrt{2}}+1}^{n} \binom{n}{j} (b_n)^j q^{n(n-j)} \leqslant \sqrt{2} n 2^n (2\ n^2)^n q^{\frac{n^2}{\sqrt{2}}-n}\\ &\leqslant \sqrt{2} q^{\frac{1}{\sqrt{2}}n^2 - n + \log_q(n) + n\log_q(4n^2)}\\ &\leqslant \sqrt{2} q^{\frac{1}{\sqrt{2}}n^2 - n + \log_2(n) + n\log_2(4n^2)}\\ &\leqslant \sqrt{2} q^{n^2-3n},\end{aligned}$$

where $b_n \triangleq 3\binom{n}{2}$ and the last inequality holds for $n \geqslant 54$.

Following the same steps of Theorem 11, we can bound the number of good arrays as $|\mathcal{C}_{\mathcal{G}}| \leqslant \frac{q^{n^2-1}}{\frac{n^2}{2}}$. We can now write

$$\begin{aligned}|\mathcal{C}| &= |\mathcal{C}_{\mathcal{G}}| + |\mathcal{C}_{\mathcal{B}}|\\ &\leqslant |\mathcal{C}_{\mathcal{G}}| + |\mathcal{B}_n|\\ &\leqslant \frac{q^{(n-1)^2}}{\frac{n^2}{2}} + \sqrt{2} q^{n^2-3n}\\ &= \frac{q^{n^2}}{q^{2n-1}\cdot\frac{n^2}{2}} \left(1 + \frac{n^2}{\sqrt{2}\cdot q^{n+1}}\right)\\ &\approx \frac{q^{n^2}}{q^{2n-1}\cdot\frac{n^2}{2}},\end{aligned}$$

where the last equality is an asymptotic statement in n. This concludes the proof. ■

APPENDIX C
PROOFS OF CLAIM 16 AND CLAIM 17

Proof of Claim 16: Remember that we have defined $\ell = \log n$. We first show that the redundancy of $\mathcal{U}_\perp$ is given by

$$R_{\mathcal{U}_\perp} = (n - \log n - 1)\log\left(\frac{n}{n-1}\right).$$

Recall that $\mathcal{U}_\perp$ is defined as the set of all $n \times n$ arrays in which any two consecutive columns, from column 1 to $n-\ell$, are different when restricted to the first ℓ entries, i.e.,

$$\mathcal{U}_\perp \triangleq \left\{\mathbf{X} : \mathbf{X}_{[\ell],j} \neq \mathbf{X}_{[\ell],j+1}, \quad j \in [n-\ell-1]\right\}.$$

We count the number of arrays that satisfy those constraints. The first ℓ entries of the first column can take 2^ℓ different values. For every other column from 2 to $n-\ell$, the first ℓ entries can take $2^\ell - 1$ different values because they have to be different from the entries of the column before. All other entries have no constraints and can take $2^{n^2-\ell(n-\ell)}$ values. We can then write,

$$\begin{aligned}|\mathcal{U}_\perp| &= 2^\ell (2^\ell - 1)^{n-\ell-1} 2^{n^2-\ell(n-\ell)}\\ &= 2^{n^2} 2^{-(n-\ell-1)\ell} (2^\ell - 1)^{n-\ell-1}\\ &= 2^{n^2}(1-2^{-\ell})^{n-\ell-1}.\end{aligned}$$

Thus, the redundancy can be computed as

$$\begin{aligned}R_{\mathcal{U}_\perp} &= n^2 - \log|\mathcal{U}_\perp|\\ &= -(n-\log n - 1)\log\left(1 - \frac{1}{n}\right)\\ &= (n - \log n - 1)\log\left(\frac{n}{n-1}\right).\end{aligned}$$

To complete the proof we need to show that

$$R_{\mathcal{V}_\perp} = (n - \log n - 2)\log\left(\frac{n}{n-1}\right) + 1.$$

Recall that $\mathcal{V}_\perp$ is defined as the set of all $n \times n$ arrays in which the entry $X_{\ell+1,n}$ is fixed to a predetermined value and any two consecutive rows, from row $\ell+1$ to $n-1$, are different when restricted to the last ℓ entries, i.e.,

$$\mathcal{V}_\perp \triangleq \left\{\mathbf{X} : \begin{array}{ll}\mathbf{X}_{i,[n-\ell+1,n]} \neq \mathbf{X}_{i+1,[n-\ell+1,n]}, & \ell < i < n-1\\ X_{\ell+1,n} \equiv \ell \mod 2 & \end{array}\right\}.$$

We count the number of arrays that satisfy those constraints. The last ℓ entries of row $\ell+1$ can take $2^{\ell-1}$ different values, because $X_{\ell+1,n}$ is predetermined. For every other row from $\ell+2$ to $n-1$, the last ℓ entries can take $2^\ell - 1$ different values because they have to be different from the entries of the row before. All other bits have no constraints and can take $2^{n^2-\ell(n-\ell-1)}$ values. We can then write,

$$\begin{aligned}|\mathcal{U}_\perp| &= 2^{\ell-1}(2^\ell-1)^{n-\ell-2} 2^{n^2-\ell(n-\ell-1)}\\ &= 2^{n^2} 2^{-(n-\ell-2)\ell}(2^\ell-1)^{n-\ell-2} 2^{-1}\\ &= 2^{n^2}(1-2^{-\ell})^{n-\ell-2}2^{-1}.\end{aligned}$$

The redundancy can then be computed as

$$\begin{aligned}R_{\mathcal{U}_\perp} &= n^2 - \log|\mathcal{U}_\perp|\\ &= -(n-\log n-2)\log\left(1-\frac{1}{n}\right) + 1\\ &= (n-\log n - 2)\log\left(\frac{n}{n-1}\right) + 1.\end{aligned}$$

■

Next we prove Claim 17, i.e. we show that the redundancy of $\mathcal{S}_\cap$ is upper bounded by

$$R_{\mathcal{S}_\cap} < \log n + 5.$$

Proof of Claim 17: Recall that $\mathcal{S}_\cap$ is defined as the set of $n \times n$ arrays in which the $\ell \times \ell$ sub array ending at the last bit of the first row of the original array has distinct consecutive columns, distinct consecutive rows, the last row fixed to a predetermined value and the first 4 bits of the second to last column are also predetermined. $\mathcal{S}_\cap$ also guarantees that the first column of the $\ell \times \ell$ sub array is different from the ℓ entries of column $n-\ell$ and similarly to the last row., i.e.,

$$\mathcal{S}_{\cap} \triangleq \left\{ \mathbf{X} : \begin{array}{l} \mathbf{X}_{[\ell],j} \neq \mathbf{X}_{[\ell],j+1}, \quad n-\ell \leqslant j < n, \\ \mathbf{X}_{i,[n-\ell+1,n]} \neq \mathbf{X}_{i+1,[n-\ell+1,n]}, i \in [\ell], \\ \mathbf{X}_{[4],n-1} = [0000]^T, \\ \mathbf{X}_{[\ell],n} = [010101\cdots]^T \end{array} \right\}.$$

Let $\mathcal{S}_{c,r}$ be the set of arrays that have different consecutive columns and different consecutive rows. $\mathcal{S}_{\cap}$ is the intersection between $\mathcal{S}_{c,r}$ and the set of all arrays that have the first ℓ entries of the last column for an alternating sequence and the first 4 entries of the second to last columns fixed to 0. We shall prove in the sequel that $|\mathcal{S}_{c,r}| > 2^{\ell^2-1}$. Once we have this bound, we can write

$$|\mathcal{S}_{\cap}| \geqslant \frac{|\mathcal{S}_{c,r}|}{2^{\ell}2^4} > \frac{2^{\ell^2}}{2^{\ell}2^5}. \tag{11}$$

The first inequality follows from the fact that fixing the last column to a predetermined value reduces the number of arrays in $\mathcal{S}_{\cap}$ by at most 2^{ℓ} arrays and fixing 4 bits of the second to last column reduces the number of arrays by at most 2^4.

Therefore, using (11) we have

$$R_{\mathcal{S}_{\cap}} = \ell^2 - |\mathcal{S}_{\cap}| \leqslant \ell + 5 < \log n + 5.$$

The remainder of the proof is to show that $|\mathcal{S}_{c,r}| \geqslant 2^{\ell^2-1}$. We start by showing that the number of $\ell \times \ell$ arrays is lower bounded by 2^{ℓ^2-1}. This means that with one bit of redundancy we can guarantee the constraints on the rows and columns.

To that end, we count the number of arrays that have at least two identical consecutive columns. Let j and $j+1$, $j = n-\ell, \dots, n-1$, be the indices of two identical consecutive columns. Column j can take $2^{\ell}-1$ possible values and column $j+1$ can only take one value. Not imposing any constraints on the other $(\ell-2)$ columns, each column can have 2^{ℓ} values and we have $(\ell-1)$ possible values for j. Therefore, the number of arrays having at least two identical consecutive columns is $(\ell-1)(2^{\ell}-1)(2^{\ell})^{\ell-2}$.

Following the same counting argument, the number of $\ell \times \ell$ arrays that have at least two identical consecutive rows is $(\ell-1)2^{\ell}(2^{\ell})^{\ell-2}$.

The number of arrays in $\mathcal{S}_{c,r}$ is lower bounded by the total number of $\ell \times \ell$ arrays minus the number of arrays that have at least two identical consecutive columns and minus the number of arrays that have at least two identical consecutive rows. Thus, we can write

$$\begin{aligned} |\mathcal{S}_{c,r}| &\geqslant 2^{\ell^2} - 2(\ell-1)(2^{\ell}-1)(2^{\ell})^{\ell-2} \\ &> 2^{\ell^2} - 2(\ell-1)2^{\ell}(2^{\ell})^{\ell-2} \qquad (12) \\ &\geqslant 2^{\ell^2-1}. \qquad (13) \end{aligned}$$

The inequality in (12) follows from

$$2(\ell-1)(2^{\ell}-1)(2^{\ell})^{\ell-2} < 2(\ell-1)2^{\ell}(2^{\ell})^{\ell-2}, \tag{14}$$

which is true because $2^{\ell}-1 < 2^{\ell}$. The inequality in (13) follows from

$$2(\ell-1)2^{-\ell} \leqslant \frac{1}{2}. \tag{15}$$

which is equivalent to $4(\ell-1) \leqslant 2^{\ell}$ and is true for all $\ell \geqslant 3$. ■

Appendix D
Proof of Claim 19

We prove that for a vector $\mathbf{u} = (u_{-4}, \dots, u_{n-5}) \in \Sigma_n^n$, there exists a value of u_0 such that the following holds:

1) $u_1, \dots, u_{n-5}$ can take arbitrary values such that any two consecutive symbols are different.
2) u_{-2} and u_{-1} are the n-ary representation of two complement binary alternating sequences i.e., $u_{-2} = -u_{-1}$, of length $\log n$ each.
3) $u_{-4} = \sum_{i=0}^{n-5} u_i \mod n$.
4) u_{-3} is equal to

$$\sum_{i=1}^{n-4} (i-1)s_i \mod (n-4)$$

where, $\mathbf{s} = (s_1, \dots, s_{n-4})$ is the signature vector computed as $s_1 = 1$ and

$$s_i = \begin{cases} 1 & \text{if } u_{i-1} \geqslant u_{i-2} \\ 0 & \text{otherwise.} \end{cases}$$

5) u_0 is chosen such that $u_{-4} \neq u_{-3}$, $u_{-3} \neq u_{-2}$, $u_{-1} \neq u_0$ and $u_0 \neq u_1$.

Let u_{-2}, u_{-1} and $u_1, \dots, u_{n-5}$ be fixed. The symbol u_{-3} can take two different values depending whether the chosen u_0 is less than or equal to u_1 or not. The value of u_{-4} can take n different values depending on the value of u_0. Therefore, we start by ensuring that u_{-3} is different than u_{-2}, i.e., we choose if $u_0 \leqslant u_1$ or $u_0 > u_1$. Assume that $u_0 \leqslant u_1$. Once u_{-3} is fixed, we must choose a given value of u_0 such that $u_0 \neq u_{-1}$ that makes u_{-4} different than u_{-3}. Notice that there is a one-to-one mapping between the value of u_0 and the value of u_{-4}. Thus, since u_{-1} is a large number, as long as $u_{-3} \geqslant 2$, u_0 has at least two options (0 and 1) out of which at least one satisfies all the aforementioned requirements. However, if $u_{-3} = 1$, u_0 must be equal to 0. In this case, if u_{-4} is equal to u_{-3} (then u_0 must be non zero) we switch the symbols u_{-2} and u_{-1} so that u_0 can now be greater than u_1 and has more than two different options that satisfy the aforementioned requirements. A similar argument holds for the case where $u_0 > u_1$. ■

Acknowledgment

The authors would like to thank an Associate Editor and the anonymous reviewers for their valuable comments that contributed to the improvement of the quality of this work.

References

[1] R. Bitar, L. Welter, I. Smagloy, A. Wachter-Zeh, and E. Yaakobi, "Criss-cross insertion and deletion correcting codes," 2020, *arXiv:2004.14740*. [Online]. Available: http://arxiv.org/abs/2004.14740

[2] R. Heckel, G. Mikutis, and R. N. Grass, "A characterization of the DNA data storage channel," *Sci. Rep.*, vol. 9, no. 1, p. 9663, Dec. 2019.

[3] L. Dolecek and V. Anantharam, "Using Reed–Muller RM (1,m) codes over channels with synchronization and substitution errors," *IEEE Trans. Inf. Theory*, vol. 53, no. 4, pp. 1430–1443, Apr. 2007.

[4] F. Sala, C. Schoeny, N. Bitouzé, 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.

[5] R. Venkataramanan, H. Zhang, and K. Ramchandran, "Interactive low-complexity codes for synchronization from deletions and insertions," in *Proc. 48th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Sep. 2010, pp. 1412–1419.

[6] S. M. S. T. Yazdi and L. Dolecek, "A deterministic polynomial-time protocol for synchronizing from deletions," *IEEE Trans. Inf. Theory*, vol. 60, no. 1, pp. 397–409, Jan. 2013.

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

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals (in Russian)," *Doklady Akademii Nauk SSR*, vol. 163, no. 4, pp. 845–848, 1965.

[9] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Automatika i Telemkhanika*, vol. 161, no. 3, pp. 288–292, 1965.

[10] V. Guruswami and C. Wang, "Deletion codes in the high-noise and high-rate regimes," *IEEE Trans. Inf. Theory*, vol. 63, no. 4, pp. 1961–1970, Apr. 2017.

[11] 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 2017.

[12] S. K. Hanna and S. E. Rouayheb, "Guess & check codes for deletions, insertions, and synchronization," *IEEE Trans. Inf. Theory*, vol. 65, no. 1, pp. 3–15, Jan. 2018.

[13] R. Gabrys and F. Sala, "Codes correcting two deletions," *IEEE Trans. Inf. Theory*, vol. 65, no. 2, pp. 965–974, Feb. 2019.

[14] J. Sima, N. Raviv, and J. Bruck, "Two deletion correcting codes from indicator vectors," *IEEE Trans. Inf. Theory*, vol. 66, no. 4, pp. 2375–2391, Apr. 2020.

[15] J. Sima and J. Bruck, "Optimal k-deletion correcting codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2019, pp. 847–851.

[16] V. Guruswami and J. Håstad, "Explicit two-deletion codes with redundancy matching the existential bound," 2020, *arXiv:2007.10592*. [Online]. Available: http://arxiv.org/abs/2007.10592

[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] D. Smith, T. G. Swart, K. A. S. Abdel-Ghaffar, H. C. Ferreira, and L. Cheng, "Interleaved constrained codes with markers correcting bursts of insertions or deletions," *IEEE Commun. Lett.*, vol. 21, no. 4, pp. 702–705, Apr. 2017.

[19] R. M. Roth, "Maximum-rank array codes and their application to crisscross error correction," *IEEE Trans. Inf. Theory*, vol. 37, no. 2, pp. 328–336, Mar. 1991.

[20] E. M. Gabidulin and N. I. Pilipchuk, "Error and erasure correcting algorithms for rank codes," *Des., Codes Cryptogr.*, vol. 49, no. 1, pp. 105–122, Dec. 2008.

[21] D. Lund, E. M. Gabidulin, and B. Honary, "A new family of optimal codes correcting term rank errors," in *Proc. IEEE Int. Symp. Inf. Theory*, Jun. 2000, p. 115.

[22] V. R. Sidorenko, "Class of correcting codes for errors with a lattice configuration," *Problemy Reredachi Informatsii*, vol. 12, no. 3, pp. 165–171, Mar. 1976.

[23] M. Blaum and J. Bruck, "MDS array codes for correcting a single criss-cross error," *IEEE Trans. Inf. Theory*, vol. 46, no. 3, pp. 1068–1077, May 2000.

[24] È. M. Gabidulin, "Optimum codes correcting lattice errors," *Problems Inf. Transmiss.*, vol. 21, no. 2, pp. 103–108, Apr. 1985.

[25] R. M. Roth, "Probabilistic crisscross error correction," *IEEE Trans. Inf. Theory*, vol. 43, no. 5, pp. 1425–1438, Sep. 1997.

[26] A. Wachter-Zeh, "List decoding of crisscross errors," *IEEE Trans. Inf. Theory*, vol. 63, no. 1, pp. 142–149, Jan. 2017.

[27] M. Hagiwara, "Conversion method from erasure codes to multi-deletion error-correcting codes for information in array design," in *Proc. Int. Symp. Inf. Theory Appl. (ISITA)*, 2020, pp. 274–278.

[28] A. Krishnamurthy, A. Mazumdar, A. McGregor, and S. Pal, "Trace reconstruction: Generalized and parameterized," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3233–3250, Jun. 2021.

[29] G. M. Tenengol'ts, "Nonbinary codes, correcting single deletion or insertion (Corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-30, no. 5, pp. 766–769, Sep. 1984.

[30] L. Welter, R. Bitar, A. Wachter-Zeh, and E. Yaakobi, "Multiple criss-cross insertion and deletion correcting codes," 2021, *arXiv:2102.02727*. [Online]. Available: http://arxiv.org/abs/2102.02727

Rawad Bitar (Member, IEEE) received the Diploma degree in computer and communication engineering from the Faculty of Engineering, Lebanese University, Roumieh, Lebanon, in 2013, the M.S. degree from the Doctoral School, Lebanese University, Tripoli, Lebanon, in 2014, and the Ph.D. degree in electrical engineering from Rutgers, The State University of New Jersey, New Brunswick, NJ, USA, in 2020. He is currently a Post-Doctoral Researcher with the Technical University of Munich. His research interests include information theory and coding theory with a focus on coding for insertions and deletions and coding for information theoretically secure distributed systems with application to machine learning.

Lorenz Welter (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical engineering and information technology from Karlsruhe Institute of Technology (KIT), Germany, in 2015 and 2018, respectively. He is currently pursuing the Ph.D. degree with the Coding and Cryptography Group, Institute of Communications Engineering, Technical University of Munich (TUM), under the supervision of Prof. Wachter-Zeh. His research interests include coding theory and its applications with focus on insertion and deletion errors.

Ilia Smagloy (Student Member, IEEE) received the B.Sc. degree in software engineering from the Technion—Israel Institute of Technology, Haifa, Israel, in 2020, where he is currently pursuing the M.Sc. degree in computer science, under the supervision of Prof. Yaakobi. His research interests include coding theory and its applications with focus on insertion and deletion errors.

Antonia Wachter-Zeh (Senior Member, IEEE) received the M.Sc. degree in communications technology from Ulm University, Germany, in 2009, and the Ph.D. degree from Ulm University and the Université de Rennes 1, Rennes, France, in 2013. From 2013 to 2016, she was a Post-Doctoral Researcher with the Technion—Israel Institute of Technology, Haifa, Israel, and from 2016 to 2020 a Tenure Track Assistant Professor at the Technical University of Munich (TUM), Munich, Germany. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, TUM. Her research interests include coding theory, cryptography and information theory and their application to storage, communications, privacy, and security. She was a recipient of the DFG Heinz Maier-Leibnitz-Preis and an ERC Starting Grant.

Eitan Yaakobi (Senior Member, IEEE) received the B.A. degree in computer science and mathematics and the M.Sc. degree in computer science from the Technion—Israel Institute of Technology, Haifa, Israel, in 2005 and 2007, respectively, and the Ph.D. degree in electrical engineering from the University of California San Diego, La Jolla, CA, USA, in 2011.

From 2011 to 2013, he was a Post-Doctoral Researcher with the Department of Electrical Engineering, California Institute of Technology, Pasadena, CA, USA, and the Center for Memory and Recording Research, University of California San Diego. He is currently an Associate Professor with the Computer Science Department, Technion—Israel Institute of Technology. His research interests include information and coding theory with applications to non-volatile memories, associative memories, DNA storage, data storage and retrieval, and private information retrieval. He received the Marconi Society Young Scholar in 2009 and the Intel Ph.D. Fellowship from 2010 to 2011.