Beyond Single-Deletion Correcting Codes: Substitutions and Transpositions

Ryan Gabrys, *Member, IEEE*, Venkatesan Guruswami, *Fellow, IEEE*, João Ribeiro, and Ke Wu

***Abstract*— We consider the problem of designing low-redundancy codes in settings where one must correct deletions in conjunction with substitutions or adjacent transpositions; a combination of errors that is usually observed in DNA-based data storage. One of the most basic versions of this problem was settled more than 50 years ago by Levenshtein, who proved that binary Varshamov-Tenengolts codes correct one arbitrary edit error, i.e., one deletion *or* one substitution, with nearly optimal redundancy. However, this approach fails to extend to many simple and natural variations of the binary single-edit error setting. In this work, we make progress on the code design problem above in three such variations: 1) We construct linear-time encodable and decodable length-n non-binary codes correcting a single edit error with nearly optimal redundancy $\log n + O(\log\log n)$, providing an alternative simpler proof of a result by Cai *et al.* (IEEE Trans. Inf. Theory 2021). This is achieved by employing what we call *weighted VT sketches*, a new notion that may be of independent interest. 2) We show the existence of a binary code correcting one deletion *or* one adjacent transposition with nearly optimal redundancy $\log n + O(\log\log n)$. 3) We construct linear-time encodable and list-decodable binary codes with list-size 2 for one deletion *and* one substitution with redundancy $4\log n + O(\log\log n)$. This matches the Gilbert-Varshamov existential bound up to an $O(\log\log n)$ additive term.**

***Index Terms*— Synchronization errors, optimal codes, efficient encoding/decoding.**

Manuscript received 18 April 2022; accepted 25 August 2022. Date of publication 29 August 2022; date of current version 22 December 2022. The work of Venkatesan Guruswami was supported in part by the NSF under Grant CCF-1814603 and Grant CCF-2107347. The work of João Ribeiro was supported in part by the NSF under Grant CCF-1814603 and Grant CCF-2107347, in part by the NSF under Award 1916939, in part by the Defense Advanced Research Projects Agency (DARPA) Securing Information for Encrypted Verification and Evaluation (SIEVE) Program, in part by the Ripple, in part by the Department of Energy (DoE) National Energy Technology Laboratory (NETL) Award, in part by the JP Morgan Faculty Fellowship, in part by the PNC Center for Financial Services Innovation Award, and in part by the Cylab Seed Funding Award. The work of Ke Wu was supported in part by a DARPA SIEVE Award, SRI Subcontract under Grant 53978, and in part by the DARPA Prime under Contract HR00110C0086. An earlier version of this paper was presented at RANDOM 2022 [DOI: 10.4230/LIPIcs.APPROX/RANDOM.2022.8]. *(Corresponding author: João Ribeiro.)*

Ryan Gabrys is with the Department of Electrical and Computer Engineering, University of California, La Jolla, San Diego, CA 92093 USA (e-mail: ryan.gabrys@gmail.com).

Venkatesan Guruswami is with the Department of Electrical Engineering and Computer Science, University of California, Berkeley, Berkeley, CA 94720 USA, and also with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: venkatg@berkeley.edu).

João Ribeiro and Ke Wu are with the Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213 USA (e-mail: jlourenc@cs.cmu.edu; kew2@cs.cmu.edu).

Communicated by I. Tamo, Associate Editor At Large for Coding and Decoding.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TIT.2022.3202856.

Digital Object Identifier 10.1109/TIT.2022.3202856

I. Introduction

DELETIONS, substitutions, and transpositions are some of the most common types of errors jointly affecting information encoded in DNA-based data storage systems [1], [2]. Therefore, it is natural to consider models capturing the interplay between these types of errors, along with the best possible codes for these settings. More concretely, one usually seeks to pin down the optimal redundancy required to correct such errors, and also to design fast encoding and decoding procedures for low-redundancy codes. It is well-known that deletions are challenging to handle even in isolation, since they cause a loss of synchronization between sender and receiver. The situation where one aims to correct deletions in conjunction with other reasonable types of errors is even more difficult. Our understanding of this interplay remains scarce even in basic settings where only one or two such worst-case errors may occur.

One of the most fundamental settings where deletions interact with the other types of errors mentioned above is that of correcting a single *edit* error (i.e., a deletion, insertion, or substitution) over a *binary* alphabet. In this case, linear-time encodable and decodable binary codes correcting a single edit error with nearly optimal redundancy have been known for more than 50 years. Levenshtein [3] showed that the binary Varshamov-Tenengolts (VT) code [4] defined as

$$\mathcal{C} = \left\{ x \in \{0,1\}^n : \sum_{i=1}^{n} i \cdot x_i = a \mod (2n+1) \right\} \quad (1)$$

corrects one arbitrary edit error. For an appropriate choice of a, this code has redundancy at most $\log n+2$, and it is not hard to see that at least $\log n$ bits of redundancy are required to correct one edit error. Remarkably, a greedy Gilbert-Varshamov-type argument only guarantees the existence of single-edit correcting codes with redundancy $2\log n$ – much higher than what can be achieved with the VT code. We recommend Sloane's excellent survey [5] for a more in-depth overview of binary VT codes and their connections to combinatorics.

Although the questions of determining the optimal redundancy and giving nearly-optimal explicit constructions of codes in the binary single-edit setting have been settled long

ago, the underlying approach fails to extend to many simple, natural variations of this setting combining deletions with substitutions and transpositions. In this work, we make progress on these questions in three such fundamental variations, which we proceed to describe next. Although these variations are quite distinct from each other, we see the goal of this work as improving our understanding of how deletions and insertions interact with other fundamentally different types of errors.

A. Non-Binary *Single-Edit Correcting Codes*

We begin by considering the problem of correcting a single arbitrary edit error over a non-binary alphabet. This setting is especially relevant due to its connection to DNA-based data storage, which requires coding over a 4-ary alphabet. In this case, the standard *VT sketch*

$$f(x) = \sum_{i=1}^{n} i \cdot x_i \mod N, \tag{2}$$

which allows us to correct one binary edit error in (1) with an appropriate choice of N, is no longer enough. Instead, we present a natural extension of the binary VT code to a non-binary alphabet via a new notion of *weighted VT sketches*, which yields an order-optimal result.

Theorem 1: There exists a 4-ary[1] single-edit correcting code $\mathcal{C} \subseteq \{0,1,2,3\}^n$ with $\log n + \log\log n + 7 + o(1)$ bits of redundancy, where $o(1) \to 0$ when $n \to \infty$. Moreover, there exists a single edit-correcting code $\mathcal{C} \subseteq \{0,1,2,3\}^n$ with $\log n + O(\log\log n)$ redundant bits that supports linear-time encoding and decoding.

This problem was previously considered by Cai, Chee, Gabrys, Kiah, and Nguyen [6], who proved an analogous result. Our existential result requires 6 fewer bits of redundancy than the corresponding result from [6], and our explicit code supports linear time encoding and decoding procedures, while the explicit code from [6] requires $\Theta(n \log n)$ time encoding [7]. However, we believe that our more significant contribution in this setting is the simpler approach we employ to prove Theorem 1 via weighted VT sketches. The technique of weighted VT sketches seems quite natural and powerful and may be of independent interest.

We note that the existential result in Theorem 1 extends to arbitrary alphabet size q with $\log n + O_q(\log\log n)$ redundant bits, but we focus on $q = 4$ since it is the most interesting setting and provides the clearest exposition of our techniques. More details can be found in Section III, where we also present a more in-depth discussion on why the standard VT sketch (2) does not suffice in the non-binary case.

B. Binary Codes Correcting One Deletion or One Adjacent Transposition

As our second contribution, we consider the interplay between deletions and adjacent transpositions, which map 01 to 10 and vice-versa. An adjacent transposition may be seen as a special case of a burst of two substitutions. Besides its relevance to DNA-based storage, the interplay between deletions and transpositions is an interesting follow-up to the single-edit setting discussed above because the VT sketch is highly ineffective when dealing with transpositions, while it is the staple technique for correcting deletions and substitutions. The issue is that, if $y, y' \in \{0,1\}^n$ are obtained from $x \in \{0,1\}^n$ via any two adjacent transpositions of the form $01 \mapsto 10$, then $f(y) = f(y') = f(x) - 1$, where we recall that $f(z) = \sum_{i=1}^{n} i \cdot z_i \mod N$ is the VT sketch. This implies that knowing the VT sketch $f(x)$ reveals almost no information about the adjacent transposition, since correcting an adjacent transposition is equivalent to finding its location.

In this setting, the best known redundancy lower bound is $\log n$ (the same as for single-deletion correcting codes), while the the best known existential upper bound is $2\log n$, obtained by naively intersecting a single-deletion correcting code and a single-transposition correcting code. A code with redundancy $\log n + O(1)$ was claimed in [8, Sec. III], but the argument there is flawed. In this work, we determine the optimal redundancy of codes in this setting up to an $O(\log\log n)$ additive term via a novel marker-based approach. More precisely, we prove the following result, more details of which can be found in Section IV.

Theorem 2: There exists a binary code $\mathcal{C} \subseteq \{0,1\}^n$ correcting one deletion or one transposition with redundancy $\log n + O(\log\log n)$.

Since we know that every code that corrects one deletion also corrects one insertion [3], we also conclude from Theorem 2 that there exists a binary code correcting one deletion, one insertion, or one transposition with nearly optimal redundancy $\log n + O(\log\log n)$.

C. Binary Codes for *One Deletion* and *One Substitution*

To conclude, we make progress on the study of single-deletion single-substitution correcting codes. Recent work by Smagloy *et al.* [9] constructed efficiently encodable and decodable binary single-deletion single-substitution correcting codes with redundancy close to $6\log n$. On the other hand, it is known that $2\log n$ redundant bits are required, and a greedy approach shows the *existence* of a single-deletion single-substitution correcting code with redundancy $4\log n + O(1)$.

In this setting, we ask what improvements are possible if we relax the unique decoding requirement slightly and instead require that the code be *list-decodable with list-size* 2. There, our goal is to design a low-redundancy code $\mathcal{C} \subseteq \{0,1\}^n$ such that for any corrupted string $y \in \{0,1\}^{n-1} \cup \{0,1\}^n$ there are at most two codewords $x, x' \in \mathcal{C}$ that can be transformed into y via some combination of at most one deletion and one substitution. This is the strongest possible requirement after unique decoding, which corresponds to lists of size 1.

The best known *existential* upper bound on the optimal redundancy in the list-decoding setting is still $4\log n + O(1)$ via the Gilbert-Varshamov-type greedy algorithm. We give an explicit list-decodable code with list-size 2 correcting one deletion and one substitution with redundancy matching the existential bound up to an $O(\log\log n)$ additive term. At a high level, this code is obtained by combining the standard VT sketch (2) with *run-based sketches*, which have been recently used in the design of two-deletion correcting codes [10]. More

[1] A 4-ary alphabet is relevant for DNA-based data storage.

precisely, we have the following result, details of which can be found in Section V.

Theorem 3: There exists a linear-time encodable and decodable binary *list-size* 2 single-deletion single-substitution correcting code $\mathcal{C} \subseteq \{0,1\}^n$ with $4\log n + O(\log\log n)$ bits of redundancy.

Subsequently to the appearance of our work online [11], Song *et al.* [12] constructed a list-decodable code with list-size 2 for one deletion and one substitution with redundancy $3\log n + O(\log\log n)$.

D. Related Work

Recently, there has been a flurry of works making progress in coding-theoretic questions analogous to the ones we consider here in other extensions of the binary single-edit error setting.

A line of work culminating in [10], [13], and [14] has succeeded in constructing explicit low-redundancy codes correcting a constant number of worst-case deletions. Constructions focused on the two-deletion case have also been given, e.g., in [14], [15], and [10]. Explicit binary codes correcting a sublinear number of edit errors with redundancy optimal up to a constant factor have also been constructed recently [16], [17]. Other works have considered the related setting where one wishes to correct a burst of deletions or insertions [18], [19], [20], or a combination of duplications and edit errors [21]. Following up on [9], codes correcting a combination of more than one deletion and one substitution were given in [22] with sub-optimal redundancy. List-decodable codes in settings with indel errors have also been considered before. For example, Wachter-Zeh [23] and Guruswami *et al.* [24] study list-decodability from a linear fraction of deletions and insertions. Some works have also considered probabilistic models introducing deletions, insertions, and substitutions [25], [26].

Most relevant to our result in Section I-C, Guruswami and Håstad [10] constructed an explicit list-size two code correcting two deletions with redundancy $3\log n + O(\log\log n)$, thus beating the greedy existential bound in this setting.

With respect to the interplay between deletions and transpositions, Gabrys *et al.* [8] constructed codes correcting a single deletion *and* many adjacent transpositions. In an incomparable regime, Schulman and Zuckerman [27], Cheng *et al.* [28], and Haeupler and Shahrasbi [29] constructed explicit codes with good redundancy correcting a linear fraction of deletions and insertions and a nearly-linear fraction of transpositions. Adjacent transpositions are sometimes also called bit-shifts or peak-shifts. Klove [30] constructed perfect codes correcting deletions/insertions of 0's in conjunctions with adjacent transpositions.

II. Preliminaries

A. Notation and Conventions

We denote sets by uppercase letters such as S and T or uppercase calligraphic letters such as $\mathcal{C}$, and define $[n] = \{1,\ldots,n\}$, $[[n]] = \{0,1,\ldots,n-1\}$, and $S^{\leq k} = \bigcup_{i=0}^{k} S^i$ for any set S. The symmetric difference between two sets S and T is denoted by $S \triangle T$. We use the notation $\{\{a,a,b\}\}$ for multisets, which may contain several copies of each element. Given two strings x and y over a common alphabet Σ, we denote their concatenation by $x\|y$ and write $x[i:j] = (x_i, x_{i+1}, \ldots, x_j)$. We say $y \in \Sigma^k$ is a *k-subsequence* of $x \in \Sigma^n$ if there are k indices $1 \leq i_1 < i_2 < \cdots < i_k \leq n$ such that $x_{i_j} = y_j$ for $j = 1,\ldots,k$, in which case we also call x an *n-supersequence* of y. Moreover, we say $x[i:j]$ is an *a-run* of x if $x[i:j] = a^{j-i+1}$ for a symbol $a \in \Sigma$. We denote the base-2 logarithm by $\log$. A length-n *code* $\mathcal{C}$ is a subset of Σ^n for some alphabet Σ which will be clear from context. In this work, we are interested in the *redundancy* of certain codes (measured in bits), which we define as $n\log|\Sigma| - \log|\mathcal{C}|$.

B. Error Models and Codes

Since we will be dealing with three distinct but related models of worst-case errors, we begin by defining the relevant standard concepts in a more general way. We may define a worst-case error model over some alphabet Σ by specifying a family of *error balls* $\mathcal{B} = \{B(y) \subseteq \Sigma^* : y \in \Sigma^*\}$. Intuitively, $B(y)$ contains all strings that can be corrupted into y by applying an allowed error pattern. We proceed to define unique decodability of a code $\mathcal{C} \subseteq \Sigma^n$ with respect to an error model.

Definition 1 (Uniquely Decodable Code): We say a code $\mathcal{C} \subseteq \Sigma^n$ is *uniquely decodable (with respect to $\mathcal{B}$)* if $|B(y) \cap \mathcal{C}| \leq 1$ for all $y \in \Sigma^*$.

Throughout this work the underlying error model will always be clear from context, so we do not mention it explicitly. We will also consider *list-decodable* codes with small list size in Section V, and so we require the following more general definition.

Definition 2 (List-size t Decodable Code): We say a code $\mathcal{C} \subseteq \Sigma^n$ is *list-size t decodable (with respect to $\mathcal{B}$)* if $|B(y) \cap \mathcal{C}| \leq t$ for all $y \in \Sigma^*$.

Note that uniquely decodable codes correspond exactly to list-size 1 codes. Moreover, we remark that for the error models considered in this work and constant t, the best existential bound for list-size t codes coincides with the best existential bound for uniquely decodable codes up to a constant additive term.

We proceed to describe the type of errors we consider. A deletion transforms a string $x \in \Sigma^n$ into one of its $(n-1)$-subsequences. An insertion transforms a string $x \in \Sigma^n$ into one of its $(n+1)$-supersequences. A substitution transforms $x \in \Sigma^n$ into a string $x' \in \Sigma^n$ that differs from x in exactly one coordinate. An adjacent transposition transforms strings of the form ab into ba. More formally, a string $x \in \Sigma^n$ is tranformed into a string $x' \in \Sigma^n$ with the property that $x'_k = x_{k+1}$ and $x'_{k+1} = x_k$ for some k, and $x'_i = x_i$ for $i \neq k, k+1$.

We can now instantiate the above general definitions under the specific error models considered in this paper. In the case of a single edit, $B(y)$ contains all strings which can be transformed into y via at most one deletion, one insertion, or one substitution. In the case of one deletion *and* one substitution, $B(y)$ contains all strings that can be transformed into y by applying at most one deletion and at most one substitution. Finally, in the case of one deletion or one adjacent

transposition, $B(y)$ contains all strings that can be transformed into y by applying either at most one deletion or at most one transposition.

III. Non-Binary Single-Edit Correcting Codes

In this section, we describe and analyze the code construction used to prove Theorem 1. Before we do so, we provide some intuition behind our approach.

A. The Binary Alphabet Case as a Motivating Example

It is instructive to start off with the binary alphabet case and the VT code described in (1), which motivates our approach for non-binary alphabets. More concretely, we may wonder whether a direct generalization of $\mathcal{C}$ to larger alphabets also corrects a single edit error, such as

$$\mathcal{C}' = \left\{ x \in [[q]]^n \;\middle|\; \begin{array}{l} \sum_{i=1}^n i x_i = s \mod (1+2qn), \\ |\{i : x_i = c\}| = s_c \mod 2, c \in [[q]] \end{array} \right\},$$

where $[[q]] = \{0, 1, \ldots, q-1\}$.[2] However, this approach fails already over a ternary alphabet $\{0, 1, 2\}$. In fact, $\mathcal{C}'$ cannot correct worst-case deletions of 1's because it does not allow us to distinguish between

$$\ldots \underline{1}02 \ldots \qquad \text{and} \qquad \ldots 02\underline{1} \ldots,$$

which can be obtained one from the other by deleting and inserting a 1 in the underlined positions. More generally, there exist codewords $x \in \mathcal{C}'$ with substrings $(x_j = 1, x_{j+1}, \ldots, x_k)$ not consisting solely of 1's satisfying

$$\sum_{i=j+1}^{k} (x_i - 1) = 0. \tag{3}$$

This is problematic since the string x' obtained by deleting $x_j = 1$ from x and inserting a 1 between x_k and x_{k+1} is also in $\mathcal{C}'$.

In order to avoid the problem encountered by $\mathcal{C}'$, we instead consider a *weighted VT sketch* of the form

$$f_w(x) = \sum_{i=1}^{n} i \cdot w(x_i) \mod N \tag{4}$$

for some weight function $w : [[q]] \to \mathbb{Z}$ and an appropriate modulus N. Using f_w instead of the standard VT sketch $f(x) = \sum_{i=1}^n i x_i \mod N$ in the argument above causes the condition (3) for an uncorrectable 1-deletion to be replaced by

$$\sum_{i=j+1}^{k} (w(x_i) - w(1)) = 0.$$

Then, choosing $0 \le w(0) < w(1) \ll w(2) < \cdots < w(q-1)$ appropriately allows us to correct the deletion of a 1 in x given knowledge of $f_w(x)$ provided that x satisfies a simple runlength constraint. In turn, encoding an arbitrary message z into a string x satisfying this constraint can be done very efficiently via a direct application of the simple *runlength replacement* technique from [18] using few redundant bits. Theorem 1 is then obtained by instantiating the weighted VT sketch (4) with an appropriate weight function and modulus.

[2]We note that other generalizations of the VT code to non-binary alphabets (correcting only one insertion or one deletion) have been considered in the literature. E.g., see [31].

B. Code Construction

In this section, we present our construction of a 4-ary single-edit correcting code which leads to Theorem 1. As discussed in Section III-A, given an arbitrary string $x \in \{0, 1, 2, 3\}^n$ we consider a weighted VT sketch

$$f(x) = f_w(x) = \sum_{i=1}^{n} i \cdot w(x_i) \mod [1 + 2n \cdot (2 \log n + 12)],$$

where $w(0) = 0$, $w(1) = 1$, $w(2) = 2 \log n + 11$, and $w(3) = 2 \log n + 12$, along with the count sketches

$$h_c(x) = |\{i : x_i = c\}| \mod 2$$

for $c \in \{0, 1, 2\}$. Intuitively, the count sketches allow us to cheaply narrow down exactly what type of deletion or substitution occurred (but not its position). As we shall prove later on, successfully correcting the deletion of an a boils down to ensuring that

$$\sum_{i=j}^{k} (w(x_i) - w(a)) \neq 0 \tag{5}$$

for all $1 \le j \le k \le n$ such that there is $i \in [j, k]$ with $x_i \neq a$. We call strings x that satisfy this property for every a *regular*, and proceed to show that enforcing a simple runlength constraint on x is sufficient to guarantee that it is regular.

Lemma 1: Suppose $x \in \{0, 1, 2, 3\}^n$ satisfies the following property: If x' denotes the subsequence of x obtained by deleting all 1's and 3's and x'' denotes the subsequence obtained by deleting all 0's and 2's, it holds that all 0-runs of x' and all 3-runs of x'' have length at most $\log n + 3$. Then, x is regular.

Proof: First, note that when $a = 0, 3$ it follows that (5) holds trivially for all x. Thus, it suffices to consider $a = 1, 2$. Fix any x satisfying the property outlined in the lemma statement and $1 \le j \le k \le n$ such that there is $i \in [j, k]$ with $x_i \neq 1$. The runlength constraint on x' implies that there must be at least one 2 in $x[j, k]$ for every consecutive subsequence of $2\lceil \log n + 3 \rceil$ 0's that appears in $x[j, k]$. Since $w(0) - w(1) = -1$ and $w(2) - 1 = 2 \log n + 10 > 2\lceil \log n + 3 \rceil$, it follows that (5) holds. The argument for the case $a = 2$ is analogous using the fact that $w(3) - w(2) = 1$ and $w(1) - w(2) = -(2 \log n + 10) < -2\lceil \log n + 3 \rceil$. □

Let $\mathcal{G} \subseteq \{0, 1, 2, 3\}^n$ denote the set of regular strings. Given the above definitions, we set our code to be

$$\mathcal{C} = \mathcal{G} \cap \left\{ x \in \{0, 1, 2, 3\}^n \;\middle|\; \begin{array}{l} f(x) = s, \\ h_c(x) = s_c, c \in \{0, 1, 2\} \end{array} \right\} \tag{6}$$

for appropriate choices of $s \in \{0, \ldots, 1 + 2n \cdot (2 \log n + 12)\}$ and $s_c \in \{0, 1\}$ for $c = 0, 1, 2$. A straightforward application of the probabilistic method shows that most strings are regular.

Lemma 2: Let X be sampled uniformly at random from $\{0,1,2,3\}^n$. Then,

$$\Pr[X \text{ is regular}] \geq 7/8.$$

Proof: Let X' and X'' be the subsequences of X obtained by deleting all 1's and 3's or all 0's and 2's, respectively. Then, the probability that X' has a 0-run of length $\log n+4$ starting at i is $\frac{1}{16n}$. By a union bound over the fewer than n choices for i, it follows that X' has at least one such 0-run with probability at most $1/16$. Since the same argument applies to 3-runs in X'', a final union bound over the two events yields the desired result by Lemma 1. □

As a result, by the pigeonhole principle there exist choices of s, s_0, s_1, s_2 such that

$$|\mathcal{C}| \geq \frac{7 \cdot 4^n}{8 \cdot 2^3 \cdot (1 + 2n \cdot (2\log n + 12))}.$$

This implies that we can make it so that $\mathcal{C}$ has $\log n + \log\log n + 6 + o(1)$ bits of redundancy, where $o(1) \to 0$ when $n \to \infty$, as desired. If n is not a power of two, then taking ceilings yields at most one extra bit of redundancy for a total of $\log n + \log\log n + 7 + o(1)$ bits, as claimed.

It remains to show that $\mathcal{C}$ corrects a single edit in linear time and that a standard modification of $\mathcal{C}$ admits a linear time encoder. Observe that if a codeword $x \in \mathcal{C}$ is corrupted into a string y by a single edit error, we can tell whether it was a deletion, insertion, or substitution by computing $|y|$. Therefore, we treat each such case separately below.

C. Correcting One Substitution

Suppose that y is obtained from some $x \in \mathcal{C}$ by changing an a to a b at position i. Then, we can find $|w(a) - w(b)|$ by computing $h_a(y) - h_a(x)$ for $a = 0, 1, 2$. In particular, note that we can correctly detect whether no substitution was introduced, since this happens if and only if $h_a(y) = h_a(x)$ for $a = 0, 1, 2$. It also holds that

$$f(y) - f(x) = i \cdot (w(b) - w(a)).$$

Since

$$\begin{aligned} |i \cdot (w(b) - w(a))| &\leq n \cdot (2\log n + 12) \\ &< \frac{1 + 2n \cdot (2\log n + 12)}{2}, \end{aligned}$$

we can recover the position i by computing

$$i = \frac{|f(y) - f(x)|}{|w(b) - w(a)|}.$$

Note that these steps can be implemented in time $O(n)$.

D. Correcting One Deletion

Suppose that y is obtained from $x \in \mathcal{C}$ by deleting an a at position i. First, note that we can find a by computing $h_c(y) - h_c(x)$ for $c = 0, 1, 2$. Now, let $y^{(j)}$ denote the string obtained by inserting an a to the left of y_j (when $j = n$ this means we insert an a at the end of y). We have $x = y^{(i)}$ and our goal is to find i. Consider $n \geq j \geq i$ and observe that

$$\begin{aligned} f(x) - f(y^{(j)}) &= f(y^{(i)}) - f(y^{(j)}) \\ &= \sum_{\ell=i+1}^{j} (w(x_\ell) - w(a)), \end{aligned}$$

because $y_{\ell-1} = x_\ell$ for $\ell > i$. Since x is regular, it follows that $\sum_{\ell=i+1}^{j}(w(x_\ell) - w(a)) \neq 0$ unless $x_{i+1} = \cdots = x_j = a$. This suggests the following decoding algorithm: Successively compute $f(x) - f(y^{(j)})$ for $j = n, n-1, \ldots, 1$ until $f(x) - f(y^{(j)}) = 0$, in which case the above argument ensures that $y^{(j)} = x$ since we must be inserting a into the same a-run of x from which an a was deleted. This procedure runs in overall time $O(n)$, since we can compute $f(x) - f(y^{(j-1)})$ given $f(x) - f(y^{(j)})$ with $O(1)$ operations.

E. Correcting One Insertion

The procedure for correcting one insertion is very similar to that used to correct one deletion.[3] We present the argument for completeness. Suppose y is obtained from x by inserting an a between x_{i-1} and x_i (when $i = 1$ or $i = n+1$ this means we insert an a at the beginning or end of x, respectively). First, observe that we can find a by computing $h_c(x) - h_c(y)$ for $c = 0, 1, 2$. Let $y^{(j)}$ denote the string obtained from y by deleting $y_j = a$. Then, it holds that $y^{(i)} = x$ and for $j \geq i$ we have

$$\begin{aligned} f(x) - f(y^{(j)}) &= f(y^{(i)}) - f(y^{(j)}) \\ &= -\sum_{\ell=i}^{j-2} (w(x_\ell) - w(a)), \end{aligned}$$

because $y_\ell = x_{\ell-1}$ when $j > i$. As before, using the fact that x is regular allows us to conclude that $f(x) - f(y^{(j)}) = 0$ if and only if $x_i = \cdots = x_{j-2} = a$, in which case we are deleting an a from the correct a-run of x. Therefore, we can correct an insertion of an a in x by successively computing $f(x) - f(y^{(j)})$ for all j such that $y_j = a$ starting at $j = n+1$ and deleting y_j for the first j such that $f(x) - f(y^{(j)}) = 0$, in which case the argument above ensures that $y^{(j)} = x$. This procedure runs in time $O(n)$.

F. A Linear-Time Encoder

In the previous sections we described a linear-time decoder that corrects a single edit error in regular strings x assuming knowledge of the weighted VT sketch $f(x)$ and the count sketches $h_c(x)$ for $c = 0, 1, 2$. It remains to describe a low-redundancy linear-time encoding procedure for a slightly modified version of our code $\mathcal{C}$ defined in (6). Fix an arbitrary message $z \in \{0,1,2,3\}^m$. We proceed in two steps:

1) We describe a simple linear-time procedure based on runlength replacement that encodes z into a regular string $x \in \{0,1,2,3\}^{m+4}$;

[3]It is well known that every code that corrects one deletion also corrects one insertion [3]. However, this implication does not hold in general if we require efficient decoding too.

2) We append an appropriate encoding of the sketches $f(x)\|h_0(x)\|h_1(x)\|h_2(x)$ (which we now see as binary strings) to x that can be recovered even if the final string is corrupted by an edit error. This adds $O(\log\log n)$ bits of redundancy.

We begin by considering the first step. We can encode z into a regular string $x \in \{0,1,2,3\}^{m+4}$ by enforcing a runlength constraint using a simple runlength replacement technique [18, Appendix B].

Lemma 3: There is a linear-time procedure Enc that given $z \in \{0,1,2,3\}^m$ outputs $x = \mathsf{Enc}(z) \in \{0,1,2,3\}^{m+4}$ with the following property: If x' is obtained by deleting all 1's and 3's from x and x'' is obtained by deleting all 0's and 2's, it holds that all 0-runs of x' and all 3-runs of x'' have length at most $\lceil \log n + 2 \rceil$. Moreover, there is a linear-time procedure Dec such that $\mathsf{Dec}(x) = z$. In particular, x is regular by Lemma 1.

Proof: Let $z' \in \{0,2\}^{m'}$ with $m' \leq m$ denote the subsequence at positions $1 \leq i_1 < \cdots < i_{m'} \leq m$ of z obtained by deleting all 1's and 3's, and let $z'' \in \{0,1,2,3\}^{m-m'}$ denote the leftover subsequence. We may apply the runlength replacement technique from [18] to z' and z'' separately in order to obtain strings x' and x'' with the desired properties. For completeness, we describe it below. The final encoding x is obtained by inserting the symbols of x' into the positions $i_1, \ldots, i_{m'}, m+1, m+2$ of z and the symbols of x'' into the remaining positions.

The encoding of z' into x' proceeds as follows: First, append the string 20 to z'. Then, scan z' from left to right. If a 0-run of length $\lceil \log m' + 2 \rceil$ is found starting at i, then we remove it from z' and append the marker $\mathsf{bin}(i)\|22$ to z', where $\mathsf{bin}(i)$ denotes the binary expansion of i (over $\{0,2\}$ instead of $\{0,1\}$) to $\lceil \log m' \rceil$ bits. Note that the length of z' stays the same after each such operation, and the addition of a marker does not introduce new 0-runs of length $\lceil \log m' + 2 \rceil$. Repeating this procedure until no more 0-runs of length $\lceil \log m' + 2 \rceil$ are found yields a string $x' \in \{0,2\}^{m'+2}$ without 0-runs of length $\lceil \log m' + 2 \rceil \leq \lceil \log m + 2 \rceil$. This procedure, along with the transformation from x' to x, runs in time $O(n)$. The encoding of z'' into $x'' \in \{1,3\}^{m-m'+2}$ is analogous with 1 in place of 2 and 3 in place of 0.

It remains to describe how to recover z from x. It suffices to describe how to recover z' and z'' from x' and x'', respectively, in time $O(n)$. By the encoding procedure above, we know that if x' ends in a 0 then it follows that $z' = x'[1 : m' = |x'| - 2]$. If x' ends in a 2, it means that x' has suffix $\mathsf{bin}(i)\|22$ for some i. Then, we recover i from this suffix and insert a 0-run of length $\lceil \log m' + 2 \rceil$ in the appropriate position of x'. We repeat this until x' ends in a 0. This procedure also runs in time $O(n)$. The approach for x'' is analogous and yields z''. Finally, we can merge z' and z'' correctly to obtain z since we know that z' should be inserted into the positions occupied by x' in x (disregarding the last two symbols of x'). □

To finalize the description of the overall encoding procedure, let $x = \mathsf{Enc}(z)$ and define $(\overline{\mathsf{Enc}}, \overline{\mathsf{Dec}})$ to be an explicit coding scheme for strings of length $\ell = |f(x)\|h_0(x)\|h_1(x)\|h_2(x)|$ correcting a single edit error (a naive construction has redundancy $2\log\ell + O(1) = O(\log\log m)$). If

$$u = \overline{\mathsf{Enc}}(f(x)\|h_0(x)\|h_1(x)\|h_2(x)),$$

the final encoding procedure is

$$z \mapsto x\|u \in \{0,1,2,3\}^n,$$

which runs in time $O(m) = O(n)$ and has overall redundancy $\log m + O(\log\log m) = \log n + O(\log\log n)$.

Now, suppose y is obtained from $x\|u$ by introducing one edit error. We show how to recover z in time $O(n)$ from y. First, we can recover u by running $\overline{\mathsf{Dec}}$ on the last $|u|-1$, $|u|$, or $|u|+1$ symbols of y depending on whether a deletion, substitution, or insertion occurred, respectively. If the last $|u|$ symbols of y are not equal to u, we know that the edit error occurred in that part of y. Therefore, we have $x = y[1 : m+4]$ and can compute $z = \mathsf{Dec}(y[1 : m+4])$. Else, if the last $|u|$ symbols of y are equal to u, it follows that $y[1 : |y| - |u|]$ can be obtained from x via one edit error. This means we can recover x from $y[1 : |y| - |u|]$ and in turn compute $z = \mathsf{Dec}(x)$.

IV. Binary Codes Correcting One Deletion or One Transposition

In this section, we describe and analyze the code construction used to prove Theorem 2. As discussed in Section I-B, the adjacent transposition precludes the use of the standard VT sketch. Therefore, we undertake a radically different approach.

A. Code Construction and High-Level Overview of Our Approach

Our starting point is a marker-based segmentation approach considered by Lenz and Polyanskii [19] to correct bursts of deletions. We then introduce several new ideas. Roughly speaking, our idea is to partition a string $x \in \{0,1\}^n$ into consecutive short substrings $z_1^x, \ldots, z_\ell^x$ for some ℓ according to the occurrences of a special marker string in x. Then, by carefully embedding hashes of each segment z_i^x into a VT-type sketch, adding information about the *multiset* of hashes, and exploiting specific structural properties of deletions and adjacent transpositions, we are able to determine a short interval containing the position where the error occurred. Once this is done, a standard technique allows us to recover the true position of the error by slightly increasing the redundancy.

We now describe the code construction in detail. For a given integer $n > 0$, let $\Delta = 50 + 1000\log n$ and $\alpha = 1000\Delta^2 = O(\log^2 n)$. For the sake of readability, we have made no efforts to optimize constants, and assume n is a power of two to avoid using ceilings and floors. Given a string $x \in \{0,1\}^n$, we divide it into substrings split according to occurrences of the marker 0011. To avoid edge cases, assume that x ends in 0011 – this will only add 4 bits to the overall redundancy. Then, this marker-based segmentation induces a vector $z^x = (z_1^x, \ldots, z_{\ell_x}^x)$, where $1 \leq \ell_x \leq n$, and each string z_i^x has length at least 4, ends with 0011, and 0011 only occurs once in each such string. We may assume that $|z_i^x| \leq \Delta$ for all i. This will only add 1 bit to the overall redundancy, as captured in the following simple lemma.

Lemma 4: Suppose X is uniformly random over $\{0,1\}^n$. Then, we have

$$\Pr[|z_i^X| \leq \Delta, i = 1, \ldots, \ell_X] \geq \frac{1}{2}.$$

Proof: Since the probability that a fixed length-4 substring of X equals 0011 is $1/16$, it follows that the probability that $|z_i^X| > \Delta$ for any fixed i is at most

$$\left(\frac{15}{16}\right)^{\Delta/4-1} \leq \frac{1}{2n^3}.$$

A union bound over all $1 \leq i \leq n$ yields the desired statement. □

Our goal now will be to impose constraints on z^x so that (i) We only introduce $\log n + O(\log\log n)$ bits of redundancy, and (ii) If x is corrupted by a deletion or transposition in z_i^x, we can then locate a window $W \subseteq [n] = \{1, \ldots, n\}$ of size $|W| = O(\log^4 n)$ such that $z_i^x \subseteq W$. This will then allows us to correct the error later on by adding $O(\log\log n)$ bits of redundancy.

Since each z_i^x has length at most $\Delta = O(\log n)$, we will exploit the fact that there exists a hash function h with short output that allows us to correct a deletion, substitution, or transposition in all strings of length at most 3Δ. This is guaranteed by the following lemma.

Lemma 5: There exists a hash function $h : \{0,1\}^{\leq 3\Delta} \to [\alpha]$ with the following property: *If z' is obtained from z by at most two transpositions, two substitutions, or at most a deletion and an insertion, then $h(z) \neq h(z')$.*

Proof: We can construct such a hash function h greedily. Let $A(z)$ denote the set of such strings obtained from $z \in \{0,1\}^{\leq 3\Delta}$. Since $|A(z)| < \alpha$, we can set $h(z)$ so that $h(z) \neq h(z')$ for all $z' \in A(z) \setminus \{z\}$. □

With the intuition above and the hash function h guaranteed by Lemma 5 in mind, we consider the VT-type sketch

$$f(x) = \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x)) \mod (L = 10n \cdot \Delta \cdot \alpha + 1)$$

along with the count sketches

$$g_1(x) = \ell_x \mod 5,$$
$$g_2(x) = \sum_{i=1}^{n} \overline{x}_i \mod 3,$$

where $\overline{x}_i = \sum_{j=1}^{i} x_j \mod 2$. At a high level, the sketch $f(x)$ is the main tool we use to approximately locate the error in x. The count sketches $g_1(x)$ and $g_2(x)$ are added to allow us to detect how many markers are created or destroyed by the error, and to distinguish between the cases where there is no error or a transposition occurs.

With the above in mind, we define the preliminary code

$$\mathcal{C}' = \left\{ x \in \{0,1\}^n \;\middle|\; \begin{array}{l} (x_{n-3}, \ldots, x_n) = (0,0,1,1), \\ f(x) = s_0, \\ g_1(x) = s_1, g_2(x) = s_2, \\ \forall i \in [\ell_x] : |z_i^x| \leq \Delta \end{array} \right\}$$

for appropriate choices of s_0, s_1, s_2. Taking into account all constraints, the choice of Δ and α, and Lemma 4, the pigeonhole principle implies that we can choose s_0, s_1, s_2 so that this code has at most

$$4 + \log(10n \cdot \Delta \cdot \alpha + 1) + 1 + 2 + 2 + 1 = \log n + O(\log\log n) \quad (7)$$

bits of redundancy.

However, it turns out that the constraints imposed in $\mathcal{C}'$ are not enough to handle a deletion or a transposition. Intuitively, the reason for this is that, in order to make use of the sketch $f(x)$ when decoding, we will need additional information both about the hashes of the segments of x that were affected by the error and the hashes of the corresponding corrupted segments in the corrupted string y. Therefore, given a vector z^x and the hash function h guaranteed by Lemma 5, we will be interested in the associated *hash multiset*

$$H_x = \{\{h(z_1^x), \ldots, h(z_{\ell_x}^x)\}\}$$

over $[\alpha]$. As we shall see, a deletion or transposition will change this multiset by at most 4 elements. Therefore, we will expurgate $\mathcal{C}'$ so that any pair of remaining codewords x and x' satisfy either $H_x = H_{x'}$ or $|H_x \triangle H_{x'}| \geq 10$, where $\triangle$ denotes symmetric difference. This will allow us to recover the true hash multiset of x from the hash multiset of the corrupted string. The following lemma shows that this expurgation adds only an extra $O(\log m) = O(\log\log n)$ bits of redundancy.

Lemma 6: There exists a code $\mathcal{C} \subseteq \mathcal{C}'$ of size

$$|\mathcal{C}| \geq \frac{|\mathcal{C}'|}{\alpha^{10}}$$

such that for any $x, x' \in \mathcal{C}$ we either have $H_x = H_{x'}$ or $|H_x \triangle H_{x'}| \geq 10$.

Proof: Let $\mathcal{S}$ be the family of multisets over $[\alpha]$ with at most n elements. Order the multisets S in $\mathcal{S}$ in decreasing order according to the number $N(S)$ of codewords $x \in \mathcal{C}'$ such that $H_{x'} = S$. The expurgation procedure works iteratively by considering the surviving multiset S with the largest $N(S)$, removing all codewords $x \in \mathcal{C}'$ associated to $S' \in \mathcal{S}$ such that $S' \neq S$ and $|S \triangle S'| < 10$, and updating the values $N(S)$ for $S \in \mathcal{S}$. Since there are at most α^{10} multisets S' satisfying the conditions above and $N(S) \geq N(S')$ for all such S', we are guaranteed to keep at least a $\frac{1}{\alpha^{10}}$-fraction of every subset of codewords considered in each round of expurgation. This implies the desired result. □

We will take our *error-locating* code to be the expurgated code $\mathcal{C}$ guaranteed by Lemma 6. By (7) and the choice of α, it follows that there exists a choice of s_0, s_1, s_2 such that $\mathcal{C}$ has $\log n + O(\log\log n)$ bits of redundancy. We prove the following result in Section IV-C, which states that, given a corrupted version of $x \in \mathcal{C}$, we can identify a small interval containing the position where the error occurred.

Theorem 4: If $x \in \mathcal{C}$ is corrupted into y via one deletion or transposition, we can recover from y a window $W \subseteq [n]$ of size $|W| \leq 10^{10} \log^4 n$ that contains the position where the error occurred (in the case of a transposition, we take the error location to be the smallest of the two affected indices).

B. Error Correction From Approximate Error Location

In this section, we argue how we can leverage Theorem 4 to correct one deletion or one transposition by adding $O(\log\log n)$ bits of redundancy to $\mathcal{C}$, thus proving Theorem 2.

Let $L = 10^{10}\log^4 n$. We partition $[n]$ into consecutive disjoint intervals $B_1^{(1)}, B_2^{(1)}, \ldots, B_t^{(1)}$ of length $2L+1$. Moreover, we define a family of shifted intervals $B_1^{(2)}, \ldots, B_{t-1}^{(2)}$ where $B_i^{(2)} = [a+L, b+L]$ if $B_i^{(1)} = [a,b]$. For a given string $x \in \{0,1\}^n$, let $x^{(1,i)}$ denote its substring corresponding to $B_i^{(1)}$ and $x^{(2,i)}$ its substring corresponding to $B_i^{(2)}$.

The key property of these families of intervals we exploit is the fact that the window W of length at most L guaranteed by Theorem 4 satisfies either $W \subseteq B_i^{(1)}$ or $W \subseteq B_i^{(2)}$ for some i. As a result, we are able to recover $x^{(1,j)}$ (resp. $x^{(2,j)}$) for all $j \neq i$ from y if $W \subseteq B_i^{(1)}$ (resp. $W \subseteq B_i^{(2)}$). Moreover, we can also recover a string $y^{(i)}$ that is obtained from $x^{(1,i)}$ or $x^{(2,i)}$ via at most one deletion or one transposition. Therefore, it suffices to reveal an additional sketch which allows us to correct a deletion or a transposition in strings of length $2L+1 = O(\log^4 n)$ for each interval. Crucially, since we can already correctly recover all bits of x except for those in the corrupted interval, we may XOR all these sketches together and only pay the price of one such sketch. We proceed to discuss this more concretely.

Suppose $\widehat{f} : \{0,1\}^{2L+1} \to \{0,1\}^{\ell}$ is a sketch with the following property: *If $z \in \{0,1\}^{2L+1}$ is transformed into y via at most one deletion or one transposition, then knowledge of y and $\widehat{f}(z)$ is sufficient to recover z uniquely.* It is easy to construct such a sketch with $\ell = O(\log L) = O(\log\log n)$ [8]. For completeness, we provide an instantiation in Appendix . Armed with $\widehat{f}$, we define the full sketches

$$\widehat{g}_1(x) = \bigoplus_{i=1}^{t} \widehat{f}(x^{(1,i)})$$

and

$$\widehat{g}_2(x) = \bigoplus_{i=1}^{t-1} \widehat{f}(x^{(2,i)})$$

Note that $\widehat{g}_b(x)$ has length $\ell = O(\log\log n)$ for $b \in \{0,1\}$. Then, we take our final code to be

$$\widehat{\mathcal{C}} = \{x \in \mathcal{C} : \widehat{g}_1(x) = s_3, \widehat{g}_2(x) = s_4\}$$

which has redundancy $\log n + O(\log\log n)$ for some choice of s_3 and s_4. To see that $\widehat{\mathcal{C}}$ indeed corrects one deletion or one transposition, note that, by the discussion above, if the window W guaranteed by Theorem 4 satisfies $W \subseteq B_i^{(1)}$, then we can recover $\widehat{f}(x^{(1,i)})$ from $\widehat{g}_1(x)$ and y, along with a string $y^{(i)}$ obtained from $x^{(1,i)}$ by at most one deletion or one transposition. Then, the properties of $\widehat{f}$ ensure that we can uniquely recover $x^{(1,i)}$ from $y^{(i)}$ and the sketch $\widehat{f}(x^{(1,i)})$. The reasoning for when $W \subseteq B_i^{(2)}$ is analogous. This yields Theorem 2.

C. Proof of Theorem 4

We prove Theorem 4 in this section, which concludes our argument. Fix $x \in \mathcal{C}$ and suppose y is obtained from x via one deletion or one transposition. We consider several independent cases based on the fact that a marker cannot overlap with itself, that we can identify whether a deletion occurred by computing $|y|$, and that we can identify whether a transposition occurred by comparing $g_2(x)$ and $g_2(y)$.

1) Locating One Deletion: In this section, we show how we can localize one deletion appropriately. Fix $x \in \mathcal{C}$ and suppose that a deletion is applied to z_i^x. The following lemma holds due to the marker structure.

Lemma 7: A deletion either (i) Creates a new marker and does not delete any existing markers, in which case $\ell_y = \ell_x + 1$, (ii) Deletes an existing marker and does not create any new markers, in which case $\ell_y = \ell_x - 1$, or (iii) Neither deletes existing markers nor creates new markers, in which case $\ell_y = \ell_x$.

Proof: Without loss of generality, we may assume that the deletion is applied to the first bit of a 0-run or to the last bit of a 1-run in $x \in \mathcal{C}$. The desired result is implied by the following three observations: First, if the deletion is applied to a run of length at least 3, then no marker is created nor destroyed. Second, if the deletion is applied to a run of length 2, then a marker may be destroyed, but no marker is created. Finally, if the deletion is applied to a run of length 1, then a marker may be created, but no marker is destroyed. □

Note that we can distinguish between the cases detailed in Lemma 7 by comparing $g_1(x)$ and $g_1(y)$. Thus, we analyze each case separately:

1) $\ell_y = \ell_x$: In this case, we have

$$z^y = (z_1^x, \ldots, z_{i-1}^x, z_i', z_{i+1}^x, \ldots, z_{\ell_x}^x), \tag{8}$$

where z_i' is obtained from z_i^x by a deletion (in particular, $|z_i'| = |z_i^x| - 1$). Therefore, it holds that

$$\begin{aligned} f(x) - f(y) &= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad - \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot \alpha + h(z_j^y)) \mod L \\ &= i(|z_i^x| \cdot \alpha + h(z_i^x) - |z_i'| \cdot \alpha - h(z_i')) \\ &= i(\alpha + h(z_i^x) - h(z_i')), \end{aligned}$$

where the second equality uses (8) and $\ell_y = \ell_x$. Let H_y denote the hash multiset of y. Then, we know that $|H_x \triangle H_y| \leq 2$. Therefore, we can recover H_x from H_y, which means that we can recover $h(z_i^x) - h(z_i')$. Indeed, if $h(z_i^x) - h(z_i') = 0$ then $H_x = H_y$. On the other hand, if $h(z_i^x) - h(z_i') \neq 0$ then $|H_x \triangle H_y| = 2$ and we recover both $h(z_i^x)$ (the element in H_x but not in H_y) and $h(z_i')$ (the element in H_y but not in H_x). As a result, we know $m + h(z_i^x) - h(z_i')$. Since it also holds that $m + h(z_i^x) - h(z_i') \neq 0$ (because $|h(z_i^x) - h(z_i')| < m$), we can recover i from $f(x) - f(y)$. This gives a window W of length at most $\Delta = O(\log n)$.

2) $\ell_y = \ell_x - 1$: In this case, the marker at the end of z_i^x is destroyed, merging z_i^x and z_{i+1}^x. Observe that if $i = \ell_x$ then we can simply detect that the last marker in x was destroyed. Therefore, we assume that $i < \ell_x$, in which case we have

$$z^y = (z_1^x, \ldots, z_{i-1}^x, z_i', z_{i+2}^x, \ldots, z_{\ell_x}^x), \tag{9}$$

where $|z_i'| = |z_i^x| + |z_{i+1}^x| - 1$. Consequently, it holds that

$$\begin{aligned}
&f(x) - f(y)\\
&= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x))\\
&- \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot \alpha + h(z_j^y)) \mod L\\
&= i(|z_i^x| \cdot \alpha + h(z_i^x)) + (i+1)(|z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x))\\
&- i(|z_i'| \cdot \alpha + h(z_i')) + \sum_{j=i+2}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&= \sum_{j=i+2}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&+ i(\alpha + h(z_i^x) + h(z_{i+1}^x) - h(z_i'))\\
&+ (|z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x)).
\end{aligned}$$

Note that, since $|H_x \triangle H(y)| \leq 3$, we can recover H_x from H_y. In particular, this means that we know $h(z_i^x) + h(z_{i+1}^x) - h(z_i')$. Therefore, for $i' = \ell_y - 1, \ell_y - 2, \ldots, i$ we can compute the "potential function"

$$\begin{aligned}
\Phi(i') &= \sum_{j=i'+1}^{\ell_y} (|z_j^y| \cdot \alpha + h(z_j^y))\\
&\quad + i'(\alpha + h(z_i^x) + h(z_{i+1}^x) - h(z_i'))\\
&= \sum_{j=i'+2}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&\quad + i'(\alpha + h(z_i^x) + h(z_{i+1}^x) - h(z_i')).
\end{aligned}$$

Note that

$$\begin{aligned}
|\Phi(i) - (f(x) - f(y))| &= ||z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x)|\\
&\leq \Delta \cdot \alpha + \alpha\\
&\leq 10^7 \log^2 n. \qquad (10)
\end{aligned}$$

Moreover, we also have

$$\begin{aligned}
\Phi(i'-1) - \Phi(i') &= |z_{i'+1}^x| \cdot \alpha + h(z_{i'+1}^x)\\
&\quad - (\alpha + h(z_i^x) + h(z_{i+1}^x) - h(z_i'))\\
&\geq 4\alpha - 3\alpha = \alpha. \qquad (11)
\end{aligned}$$

This suggests the following procedure for recovering the window W. Sequentially compute $\Phi(i')$ for i' starting at $\ell_y - 1$ until we find $i^\star \geq i$ such that $|\Phi(i') - (f(x) - f(y))| \leq 10^6 \log^2 n$. This is guaranteed to exist since $i' = i$ satisfies this property. We claim that $i^\star - i \leq 10^7 \log n$. In fact, if this is not the case then the monotonicity property in (11) implies that

$$|\Phi(i) - (f(x) - f(y))| > \alpha \cdot 10^7 \log n > 10^7 \log^2 n,$$

contradicting (10). Since $|z_j^x| \leq \Delta$ for every j, recovering $i^\star$ also yields a window $W \subseteq [n]$ of size

$$|W| = 10^6 \log n \cdot \Delta = 10^9 \log^2 n$$

containing the error position, as desired.

3) $\ell_y = \ell_x + 1$: This case is similar to the previous one. We present it for completeness. The deletion causes the segment z_i^x to be split into two consecutive segments z_i' and z_i'' such that $|z_i'| + |z_i''| = |z_i^x| - 1$. Therefore, we have

$$z^y = (z_1^x, \ldots, z_{i-1}^x, z_i', z_i'', z_{i+1}^x, \ldots, z_{\ell_x}^x). \qquad (12)$$

We may compute

$$\begin{aligned}
&f(x) - f(y)\\
&= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x))\\
&- \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot \alpha + h(z_j^y)) \mod L\\
&= i(|z_i^x| \cdot \alpha + h(z_i^x)) - i(|z_i'| \cdot \alpha + h(z_i'))\\
&- (i+1)(|z_i''| \cdot \alpha + h(z_i'')) - \sum_{j=i+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&= - \sum_{j=i+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&+ i(\alpha + h(z_i^x) - h(z_i') - h(z_i''))\\
&- (|z_i''| \cdot \alpha + h(z_i'')).
\end{aligned}$$

As in the previous case, we can recover H_x from H_y, and this implies we can also recover $h(z_i^x) - h(z_i') - h(z_i'')$. Therefore, for $i' \geq i$ we can compute

$$\begin{aligned}
\Phi(i') &= - \sum_{j=i'+2}^{\ell_y} (|z_j^y| \cdot \alpha + h(z_j^y))\\
&\quad + i'(\alpha + h(z_i^x) - h(z_i') - h(z_i''))\\
&= - \sum_{j=i'+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x))\\
&\quad + i'(\alpha + h(z_i^x) - h(z_i') - h(z_i'')).
\end{aligned}$$

Then,

$$\begin{aligned}
|\Phi(i) - (f(x) - f(y))| &= ||z_i''| \cdot \alpha + h(z_i'')|\\
&\leq \Delta \cdot \alpha + \alpha\\
&\leq 10^7 \log^2 n, \qquad (13)
\end{aligned}$$

since $|z_i''| \leq |z_i^x| \leq \Delta$. Furthermore, for $i' > i$ we have

$$\begin{aligned}
\Phi(i') - \Phi(i'-1) &= |z_{i'}^x| \cdot \alpha + h(z_{i'}^x)\\
&\quad + (\alpha + h(z_i^x) - h(z_i') - h(z_i''))\\
&\geq 4\alpha - 2\alpha = 2\alpha. \qquad (14)
\end{aligned}$$

As in the previous case, we can exploit (13) and (14) to recover an appropriate window $W \subseteq [n]$ of size at most $10^9 \log^2 n$.

2) Locating One Transposition: In this section, we show how we can localize one transposition appropriately. Fix $x \in \mathcal{C}$ and suppose that a transposition is applied with the left bit in z_i^x (note that the right bit may be in z_{i+1}^x). Then, the following lemma holds.

Lemma 8: A transposition either (i) Creates a new marker and does not delete any existing markers, in which case $\ell_y = \ell_x + 1$, (ii) Deletes an existing marker and does not create any new markers, in which case $\ell_y = \ell_x - 1$, (iii) Neither deletes existing markers nor creates new markers, in which case $\ell_y = \ell_x$, (iv) Deletes two existing consecutive markers and does not create any new markers, in which case $\ell_y = \ell_x - 2$, or (v) Creates two consecutive new markers but does not delete any existing markers, in which case $\ell_y = \ell_x + 2$.

Proof: We obtain the desired statement via case analysis. If the leftmost bit of the adjacent transposition belongs to a run of length at least 3 in x, then no marker is created and at most one marker is destroyed, and likewise for the case where the leftmost bit belongs to some 0-run of length 2. On the other hand, the leftmost bit belongs to a 1-run of length 2, then no marker is created, but at most two markers may be destroyed (consider applying one transposition to the underlined bits in $001\underline{10}011$). If the leftmost bit belongs to a 0-run of length 1, then no marker is destroyed and at most two consecutive markers may be created (consider applying one transposition to the underlined bits in $001\underline{01}011$). Finally, if the leftmost bit belongs to a 1-run of length 1, then no marker is destroyed and at most one marker is created (consider applying one transposition to the underlined bits in $0\underline{10}1$). □

As before, we can distinguish between the cases detailed in Lemma 8 by comparing $g_1(x)$ and $g_1(y)$. Cases (i), (ii), and (iii) in Lemma 8 are analogous to the respective cases considered for a deletion in Section IV-C.1. Therefore, we focus on cases (iv) and (v).

1) $\ell_y = \ell_x$: In this case, we have
$$f(x) - f(y) = i(h(z_i^x) - h(z_i')),$$
and we can recover i by first recovering $h(z_i^x) - h(z_i') \neq 0$, which holds because z_i' is obtained from z_i^x via one transposition.
2) $\ell_y = \ell_x - 1$: In this case, we have
$$f(x) - f(y) = \sum_{j=i+2}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) + i(h(z_i^x) + h(z_{i+1}^x) - h(z_i')) + (|z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x)),$$
and we can then use the exact same approach as in Case (ii) from Section IV-C.1.
3) $\ell_y = \ell_x + 1$: In this case, we have
$$f(x) - f(y) = -\sum_{j=i+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) + i(h(z_i^x) - h(z_i') - h(z_i'')) - (|z_i''| \cdot \alpha + h(z_i'')),$$
and we can then use the exact same approach as in Case (iii) from Section IV-C.1.
4) $\ell_y = \ell_x - 2$: In this case, two consecutive markers are deleted and no new markers are created, so z_i^x, z_{i+1}^x, and z_{i+2}^x are merged into a corrupted segment z_i' satisfying $|z_i'| = |z_i^x| + |z_{i+1}^x| + |z_{i+2}^x|$. In general, we have
$$z^y = (z_1^x, \ldots, z_{i-1}^x, z_i', z_{i+3}^x, \ldots, z_{\ell_x}^x),$$
and so
$$\begin{aligned} f(x) - f(y) &= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad - \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot \alpha + h(z_j^y)) \mod L \\ &= 2 \sum_{j=i+3}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad + i(h(z_i^x) + h(z_{i+1}^x) + h(z_{i+2}^x) - h(z_i')) \\ &\quad + (|z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x)) \\ &\quad + 2(|z_{i+2}^x| \cdot \alpha + h(z_{i+2}^x)). \end{aligned}$$
Since $|H_x \triangle H_y| \leq 4$, we can recover H_x from H_y, which implies that we can recover $h(z_i^x) + h(z_{i+1}^x) + h(z_{i+2}^x) - h(z_i')$. As before, this means that for $i' \geq i$ we can compute the potential function
$$\begin{aligned} \Phi(i') &= 2 \sum_{j=i'+1}^{\ell_y} (|z_j^y| \cdot \alpha + h(z_j^y)) \\ &\quad + i'(h(z_i^x) + h(z_{i+1}^x) + h(z_{i+2}^x) - h(z_i')) \\ &= 2 \sum_{j=i'+3}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad + i'(h(z_i^x) + h(z_{i+1}^x) + h(z_{i+2}^x) - h(z_i')). \end{aligned}$$
Exploiting the fact that
$$\begin{aligned} |\Phi(i) - (f(x) - f(y))| &= (|z_{i+1}^x| \cdot \alpha + h(z_{i+1}^x)) \\ &\quad + 2(|z_{i+2}^x| \cdot \alpha + h(z_{i+2}^x)) \\ &\leq 3(\Delta \cdot \alpha + \alpha) \\ &\leq 10^{10} \log^2 n \end{aligned}$$
and
$$\Phi(i' - 1) - \Phi(i') \geq 8m - 3\alpha = 5\alpha,$$
we can use the approach from Section IV-C.1 to recover the relevant window $W \subseteq [n]$ of size at most $10^{10} \log^3 n$ containing the error position in y.
5) $\ell_y = \ell_x + 2$: This case is similar to the previous one. Two consecutive markers are created and none are deleted, meaning that z_i^x is transformed into two consecutive corrupted segments z_i', z_i'', and z_i'''. Therefore,
$$z^y = (z_1^x, \ldots, z_{i-1}^x, z_i', z_i'', z_i''', z_{i+1}^x, \ldots, z_{\ell_x}^x)$$
with $|z_i^x| = |z_i'| + |z_i''| + |z_i'''|$. We have
$$\begin{aligned} f(x) - f(y) &= \sum_{j=1}^{\ell_x} j(|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad - \sum_{j=1}^{\ell_y} j(|z_j^y| \cdot \alpha + h(z_j^y)) \mod L \\ &= -2 \sum_{j=i+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) + \\ &\quad i(h(z_i^x) - h(z_i') - h(z_i'') - h(z_i''')) \end{aligned}$$

$$- (|z_i''| \cdot \alpha + h(z_i'')) \\ - 2(|z_i'''| \cdot \alpha + h(z_i''')).$$

As above, we can recover H_x from H_y and thus also recover $h(z_i^x) - h(z_i') - h(z_i'') - h(z_i''')$. Consequently, for $i' \geq i$ we can compute the potential function

$$\begin{aligned}\Phi(i') &= -2 \sum_{j=i'+3}^{\ell_y} (|z_j^y| \cdot \alpha + h(z_j^y)) \\ &\quad + i'(h(z_i^x) - h(z_i') - h(z_i'') - h(z_i''')) \\ &= -2 \sum_{j=i'+1}^{\ell_x} (|z_j^x| \cdot \alpha + h(z_j^x)) \\ &\quad + i'(h(z_i^x) - h(z_i') - h(z_i'') - h(z_i''')).\end{aligned}$$

Since

$$\begin{aligned}|\Phi(i) - (f(x) - f(y))| &= (|z_i''| \cdot \alpha + h(z_i'')) \\ &\quad + 2(|z_i'''| \cdot \alpha + h(z_i''')) \\ &\leq 3(\Delta \cdot \alpha + \alpha) \\ &\leq 10^{10} \log^2 n\end{aligned}$$

and

$$\Phi(i') - \Phi(i' - 1) \geq 8m - 3\alpha = 5\alpha,$$

we can follow the previous approach to recover a window $W \subseteq [n]$ of size at most $10^{10} \log^3 n$ containing the error position.

V. Binary List-Size Two Code for One Deletion and One Substitution

In this section, we describe and analyze a binary list-size two decodable code for one deletion and one substitution, which yields Theorem 3. Departing from the approach of [9], our construction makes use of *run-based sketches* combined with the standard VT sketch. Run-based sketches have thus far been exploited in the construction of multiple-deletion correcting codes, including list-decodable codes with small list size [10].

A. Code Construction

We begin by describing some required concepts: Given a string $x = (x_1, \ldots, x_n) \in \{0,1\}^n$, we define its *run string* r^x by first setting $r_0^x = 0$ along with $x_0 = 0$ and $x_{n+1} = 1$, and then iteratively computing $r_i^x = r_{i-1}^x$ if $x_i = x_{i-1}$ and $r_i^x = r_{i-1}^x + 1$ otherwise for $i = 1, \ldots, n, n+1$. Note that every string x is uniquely determined by its run string r^x and vice-versa. Moreover, it holds that r^x defines a non-decreasing sequence and $0 \leq r_i^x \leq i$ for every $i = 1, \ldots, n, n+1$. As an example, the run string corresponding to $x = 011101000$ is $r^x = 0111234445$. We call r_i^x the *rank* of index i in x. We will denote the total number of runs in x by $r(x)$.

The main component of our code is a combination of the standard VT sketch

$$f(x) = \sum_{i=1}^{n} i x_i \mod (3n+1) \tag{15}$$

with the run-based sketches

$$f_1^r(x) = \sum_{i=1}^{n} r_i^x \mod (12n+1), \tag{16}$$

$$f_2^r(x) = \sum_{i=1}^{n} r_i^x (r_i^x - 1) \mod (16n^2+1) \tag{17}$$

originally considered in [10]. Additionally, we also consider the count sketches

$$h(x) = \sum_{i=1}^{n} x_i \mod 5 \quad \text{and} \quad h_r(x) = r(x) \mod 13. \tag{18}$$

Intuitively, the count sketches are used to distinguish different error patterns. The sketch $h(x)$ is used to determine the value of the bit deleted and the value of the bit flipped, while $h_r(x)$ is used to identify how the number of runs was affected by the errors. For each possible error pattern, we use the standard VT-sketch and the run-based sketches to decode. Given the above, our code is defined to be

$$\mathcal{C} = \left\{ x \in \{0,1\}^n \;\middle|\; \begin{array}{l} f(x) = s, f_1^r(x) = s_1^r, f_2^r(x) = s_2^r, \\ h(x) = u, h_r(x) = u_r \end{array} \right\}, \tag{19}$$

for an appropriate choice of $s \in [3n+1]$, $s_1^r \in [12n+1]$, $s_2^r \in [16n^2+1]$, $u \in [5]$, and $u_r \in [13]$. By the pigeonhole principle, there is such a choice which ensures $\mathcal{C}$ has redundancy $4 \log n + O(1)$.

In the remainder of this section, we first provide a high-level overview of our approach towards showing that $\mathcal{C}$ admits linear-time list-decoding from one deletion and one substitution with list-size 2. Then, we provide a formal case analysis. We remark that linear-time decoding and encoding of a slightly modified version of $\mathcal{C}$ (which has redundancy $4 \log n + O(\log \log n)$ instead) follow without difficulty from this analysis via standard methods. These algorithms are presented and analyzed in Sections V-D and V-E.

B. High-Level Overview of Our Approach

Fix $x \in \mathcal{C}$, and let y be the string obtained from x after one deletion at index d and one substitution at index e. We use x_e to denote the bit flipped and x_d to denote the bit deleted in x. When $d = e$, we have one deletion and no substitution. Our goal is to recover x from y.

We begin with some simple but useful remarks. First, we observe that one deletion and no substitution can be equivalently transformed to one deletion and one substitution. Thus, we will only consider the case in which we have one deletion and one substitution, i.e., $d \neq e$. We present a proof of this fact in Appendix . Second, the following structural lemma about the number of runs in a corrupted string will prove useful in our case analysis.

Lemma 9: If x' is obtained from x via one deletion, then either $r(x') = r(x)$ or $r(x') = r(x) - 2$. On the other hand, if x' is obtained from x via one substitution, then either $r(x') = r(x)$, $r(x') = r(x) - 2$, or $r(x') = r(x) + 2$.

Proof: The desired statement follows by case analysis. We have $r(x') = r(x) - 2$ when x' is obtained by deleting or

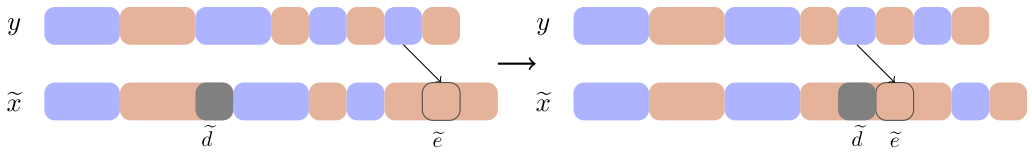

Fig. 1. Example of an elementary move. Suppose that the error pattern indicates that $x_d = 1$, $x_e = 1$, and the deletion does not reduce the number of runs while the substitution increases the number of runs by two. The process starts with the left figure in which a bit 1 is inserted at position $\widetilde{d}$, the end of a 1-run and the bit 1 at position $\widetilde{e} - 1$ is flipped. After an elementary move, $\widetilde{d}$ moves to the end of the next 1-run, and e moves to the next position that matches the error pattern $y_{\widetilde{e}-\widetilde{\delta}+1} = y_{\widetilde{e}-\widetilde{\delta}-1} = 0$.

flipping a bit in a run of length 1 in x. Otherwise, we have $r(x') = r(x)$ when x' is obtained by deleting a bit in a run of length at least 2 or by flipping the leftmost or rightmost bit in a run of length at least 2. In the remaining case where the flipped bit is in the middle of a run of length at least 3, we have $r(x') = r(x) + 2$. □

Combining Lemma 9 with the count sketches $h(x)$ and $h_r(x)$ and knowledge of y ensures that we can identify not only the values of x_d and x_e, but also $r(x)$. As a result, this allows us to split our analysis into several independent cases.

The process of decoding can be thought of as inserting a bit x_d before the d-th bit in y and flipping the $(e-\delta)$-th bit in y, where $\delta \in \{0,1\}$ is the indicator variable of whether $e > d$. Our goal is to find d and e. We will begin with a candidate position pair $(\widetilde{d}, \widetilde{e})$ with $\widetilde{d}$ is as small as possible with the property that, if $\widetilde{x}$ denotes the string obtained from y by inserting x_d before the d-th bit in y and flipping the bit at position $\widetilde{e} - \widetilde{\delta}$ in y, where $\widetilde{\delta}$ indicates whether $\widetilde{d} < \widetilde{e}$, then $f(\widetilde{x}) = f(x)$, $h_r(\widetilde{x}) = h_r(x)$, and $h_r(x') = h_r(\widetilde{x}')$, where x' (resp. $\widetilde{x}'$) denotes the string obtained from x (resp. $\widetilde{x}$) by deleting x_d (resp. $\widetilde{x}_{\widetilde{d}}$). We call such pairs *valid*. Intuitively, valid pairs are indistinguishable from the true error pattern (d, e) from the perspective of the VT sketch and the count sketches, and there may be several of them. However, crucially, many are ruled out via the run-based sketches. Note that the true error pattern (d, e) is a valid pair, so such pairs always exist.

Roughly speaking, our strategy is to start with some valid pair $(\widetilde{d}, \widetilde{e})$ and sequentially move to the *next* valid pair. This is done by moving $\widetilde{d}$ one index to the right and checking whether the *unique* index $\widetilde{e}$ that ensures $f(\widetilde{x}) = f(x)$ forms a valid pair $(\widetilde{d}, \widetilde{e})$. If this does not hold, then we move $\widetilde{d}$ one more index to the right, and repeat the process. We call this an *elementary move*. Note that since inserting a bit b into a b-run at any position gives the same output, we may always move $\widetilde{d}$ to the end of the next x_d-run in y (which may be empty). Figure 1 shows an example of an elementary move.

Considering this step-by-step process with elementary moves is useful because it turns out to be feasible to track how the different sketches change in each such move. In particular, the following equations will be useful to determine how $\widetilde{d}$ and $\widetilde{e}$ change in each elementary move. Recall that we regard y as a string obtained via one substitution at index $e - \delta$ from $x' \in \{0,1\}^{n-1}$, where x' is obtained via one deletion from x at index d. Note that

$$f(x) - f(x') = dx_d + \sum_{d}^{n-1} x'_i,$$

$$f(x') - f(y) = (e - \delta)[x_e - (1 - x_e)].$$

Moreover, we have $\sum_{d}^{n-1} x'_i = \sum_{d}^{n-1} y_i + \delta(2x_e - 1)$. Combining these three observations yields

$$f(x) - f(y) = dx_d + \sum_{i=d}^{n-1} y_i + e(2x_e - 1). \tag{20}$$

We prove that, during this sequential process, either f_1^r is monotonic and hence rules out all but one valid pair $(\widetilde{d}, \widetilde{e})$, or a convexity-type property of f_2^r, which implies that it takes on each value at most twice, rules out all but at most two valid pairs. The convexity of $f_2^r(x)$ is a consequence of the following lemma.

Lemma 10 ([10, Lemma 4.1]): Let a_i and a'_i be two sequences of non-negative integers such that $\sum_{i=1}^{n} a_i = \sum_{i=1}^{n} a'_i$ and there is a value t such that for all i satisfying $a_i < a'_i$ it holds that $a'_i \leq t$, and for all i satisfying $a_i > a'_i$ it holds that $a'_i \geq t$. Then, either $a_i = a'_i$ for all i, or $\sum_{i=1}^{n} a_i(a_i - 1) > \sum_{i=1}^{n} a'_i(a'_i - 1)$.

Finally, we note that, in the high level overview above, we ignored the fact that we do not have access to the intermediate string x', but we need to know $h_r(x')$. For example, if $r(y) = r(x)$, then there is uncertainty about $h_r(x')$. In fact, it could be that neither error changes the number of runs, or that both errors do change the number of runs but these cancel each other out. Since we are aiming for list-size 2 decoding, this is not problematic, and we handle it in the final decoding procedure.

C. Error Correction Properties

We now present a formal case analysis showing that our code is list-decodable from one deletion and one substitution with list-size 2. As discussed above, the fact that we can consider the cases below independently follows from our choice of count sketches $h(x)$ and $h_r(x)$ and Lemma 9.

1) If the Number of Runs Increases by Two: If $r(y) = r(x) + 2$, then it must be that $r(x) = r(x')$ and $r(y) = r(x') + 2$. This means that the deletion does not change the number of runs (and thus occurred in a run of length at least 2 in x), while the substitution affects a bit in the middle of a run of length at least 3. In particular, we have $y_{e-\delta-1} = y_{e-\delta+1} = 1 - y_{e-\delta}$. In this case, it follows that

$$f_1^r(x) - f_1^r(x') = r_d^x,$$
$$f_1^r(x') - f_1^r(y) = -(1 + 2(n - e + \delta)).$$

Therefore, for the run-based sketch $f_1^r(x)$ it holds that

$$f_1^r(x) - f_1^r(y) = r_d^x - (1 + 2(n - e + \delta)). \tag{21}$$

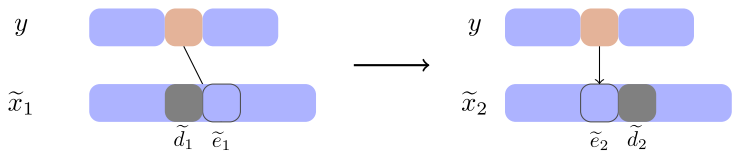

Fig. 2. An example of a take over step. If the take over happens, it must be that $\ell = 1$. The resulting $\widetilde{x}_1$ and $\widetilde{x}_2$ are the same.

We now proceed by case analysis on the value of x_d and x_e.

a) If $x_e = x_d = b$: In this case, when $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1-b)$-run of some length $\ell \geq 1$. According to (20), position $\widetilde{e}$ has to move to the left by ℓ so that $f(\widetilde{x}) = f(x)$. If we have $\widetilde{d} < \widetilde{e}$ before one elementary move but $\widetilde{d} > \widetilde{e}$ after that move, we call it a *take over step*. For each elementary move:

- If the move is not a take over step: Then, $r^{\widetilde{x}}_{\widetilde{d}}$ increases by 2 while $2(n-\widetilde{d}+\widetilde{\delta})+1$ increases by 2ℓ. Therefore, (21) implies that $f_1^r(\widetilde{x})$ strictly decreases after such a move whenever $\ell > 1$. If $\ell = 1$, then $\widetilde{e}$ moves by 1 to the left and $f_1^r(\widetilde{x})$ remains unchanged. However, since we need $1-b = y_{\widetilde{e}-\widetilde{\delta}} = 1 - y_{\widetilde{e}-\widetilde{\delta}}$ it follows that $\widetilde{e}$ cannot move only 1 position to the left, and so $\ell > 1$ necessarily.
- If the move is a take over step: Before the move, $\widetilde{d}$ is on the left of a $(1-b)$-run of length $\ell \geq 1$ while $\widetilde{e} > \widetilde{d}$ satisfies $y_{\widetilde{e}-1} = 1-b$ and $y_{\widetilde{e}-2} = y_{\widetilde{e}} = b$. After the move, $\widetilde{d}$ moves to the right of the $(1-b)$-run of length ℓ, while $\widetilde{e}$ is to the left of $\widetilde{d}$. Moreover, it must be that $y_{\widetilde{e}} = 1-b$ and $y_{\widetilde{e}-1} = y_{\widetilde{e}+1} = b$. To match the error pattern, the only possible case is that $\ell = 1$. To see why this is the case, note that when $\ell \geq 2$ the index $\widetilde{e}$ has to move to the left by at least $\ell + 2$ to match the error pattern $y_{\widetilde{e}-\widetilde{\delta}-1} = y_{\widetilde{e}-\widetilde{\delta}+1} = 1-y_{\widetilde{e}-\widetilde{\delta}}$. However, this move leads to $f(\widetilde{x}) \neq f(x)$, and thus does not yield a valid pair $(\widetilde{d}, \widetilde{e})$. When $\ell = 1$, let $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$ denote the position pair before and after the move, respectively. Then, these two pairs yield the same candidate solution $\widetilde{x}_1 = \widetilde{x}_2$. See Figure 2 for an example.

Taking into account both cases above, we see that $f_1^r(\widetilde{x})$ decreases during each elementary move, and decreases by at most $2n$ during the whole process. Since the value of $f_1^r(x)$ is taken modulo $12n+1$, there is only a unique pair $(\widetilde{d}, \widetilde{e})$ that yields a solution such that $f_1^r(\widetilde{x}) = f_1^r(x)$. Hence, $f(x)$ and $f_1^r(x)$ together with y uniquely determine one valid pair $(\widetilde{d}, \widetilde{e})$, which in turn yields a unique candidate solution $\widetilde{x} = x$.

b) If $x_d = 1 - x_e = b$: In this case, when $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1-b)$-run of some length $\ell \geq 1$. Then, $\widetilde{e}$ has to move to the right by ℓ so that $f(\widetilde{x}) = f(x)$. During each such move $f_1^r(\widetilde{x})$ strictly increases. For the whole process, $f_1^r(\widetilde{x})$ increases by at most $2n$. By a similar argument as above, we have that $f(x)$ and $f_1^r(x)$ together with y uniquely determine one valid pair $(\widetilde{d}, \widetilde{e})$ which yields the correct solution $\widetilde{x} = x$.

2) If the Number of Runs Decreases by Four: If $r(y) = r(x) - 4$, then it must be that $r(x') = r(x) - 2$ and $r(y) = r(x') - 2$. This means that the error pattern must satisfy $x_{d-1} = x_{d+1} = 1 - x_d$ and that $y_{e-\delta-1} = y_{e-\delta+1} = 1 - y_{e-\delta}$. In this case, since

$$f_1^r(x) - f_1^r(x') = r_d^x + 2(n+1-d),$$
$$f_1^r(x') - f_1^r(y) = 1 + 2(n-e+\delta),$$

we have that

$$f_1^r(x) - f_1^r(y) = [r_d^x + 2(n+1-d)] + [1 + 2(n-e+\delta)]. \quad (22)$$

We now proceed by case analysis on the value of x_d and x_e.

a) If $x_d = x_e = b$: We first find a solution of $\widetilde{d}$ and $\widetilde{e}$ such that $\widetilde{d}$ is as small as possible. During each elementary move, let $(\widetilde{d}_1, \widetilde{e}_1)$ denote the valid pair before the move and $(\widetilde{d}_2, \widetilde{e}_2)$ after the move. Let $\widetilde{x}_1$ and $\widetilde{x}_2$ be the resulting string, respectively. Recall that the rank of an index i in a string x is the i-th number in the run string r^x. For each elementary move, either $f_1^r(x)$ increases, or it remains unchanged. For those elementary moves in which $f_1^r(x)$ remains unchanged, we have the following cases:

- If $\widetilde{d}_1 < \widetilde{e}_1$ and $\widetilde{d}_2 < \widetilde{e}_2$: Since both the deletion and the substitution decreases the number of runs by two, for index i such that $\widetilde{d}_1 \leq i \leq \widetilde{d}_2$, the rank of index i in $\widetilde{x}_1$ is larger than the rank of index i in $\widetilde{x}_2$. Similarly, for index j such that $\widetilde{e}_2 \leq j \leq \widetilde{e}_1$, the rank of index j in $\widetilde{x}_1$ is smaller than the rank of index j in $\widetilde{x}_2$. By Lemma 10, we have that $f_2^r(\widetilde{x}_1) \leq f_2^r(\widetilde{x}_2)$. Therefore, if $\widetilde{d} < \widetilde{e}$ before and after an elementary move, then $f_2^r(\widetilde{x})$ is strictly increasing.
- If $\widetilde{d}_1 > \widetilde{e}_1$ and $\widetilde{d}_2 > \widetilde{e}_2$: For index i such that $\widetilde{e}_2 \leq i \leq \widetilde{e}_1$, the rank of index i in $\widetilde{x}_1$ is smaller than the rank of index i in $\widetilde{x}_2$. Similarly, for index j such that $\widetilde{d}_1 \leq j \leq \widetilde{d}_2$, the rank of index j in $\widetilde{x}_1$ is larger than the rank of index j in $\widetilde{x}_2$. By Lemma 10, we have that $f_2^r(\widetilde{x}_1) \geq f_2^r(\widetilde{x}_2)$. Therefore, if $\widetilde{d} > \widetilde{e}$ before and after an elementary move, then $f_2^r(\widetilde{x})$ is strictly decreasing.

Since when $\widetilde{d}$ moves to the right, $\widetilde{e}$ has to move to the left accordingly to be a valid pair, we can have at most one take over step. Moreover, $f_2^r(\widetilde{x})$ increases by at most $4n^2$ if $\widetilde{d} < \widetilde{e}$ before and after the elementary move, and $f_2^r(\widetilde{x})$ decreases by at most $4n^2$ if $\widetilde{d} > \widetilde{e}$ before and after the elementary move. Since the value of $f_2^r(x)$ is taken modulo $16n^2+1$, this implies that we have at most two candidate position solutions $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$, where $\widetilde{d}_1 < \widetilde{e}_1$ and $\widetilde{d}_2 > \widetilde{e}_2$ such that $f(\widetilde{x}_1) = f(\widetilde{x}_2) = f(x)$ and $f_2^r(\widetilde{x}_1) = f_2^r(\widetilde{x}_2) = f_2^r(x)$. Hence, $f(x)$, $f_1^r(x)$, and $f_2^r(x)$ together with y yield at most two candidate position solutions $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$, and thus at most two candidate solutions $\widetilde{x}_1$ and $\widetilde{x}_2$.

b) If $x_d = 1 - x_e = b$: We first find a valid pair $(\widetilde{d}, \widetilde{e})$ such that $\widetilde{d}$ is as small as possible. When $\widetilde{d}$ makes an elementary move to the right, $\widetilde{e}$ needs to move to the right such that $f(\widetilde{x}) = f(x)$. During such moves, if $\widetilde{d}$ moves

to the right by ℓ, then $r_{\widetilde{d}}^{\widetilde{x}}$ increases by at most ℓ, while $2(n+1-d)$ decreases by 2ℓ. Moreover, $2(n-e+\delta)$ is non-increasing. Therefore, by (22), $f_1^r(\widetilde{x})$ strictly decreases during the elementary moves. For the whole process, $f_1^r(\widetilde{x})$ decreases by at most $4n$. Hence, following a similar argument as above, $f(x)$ and $f_1^r(x)$ together with y uniquely determine a position pair $(\widetilde{d},\widetilde{e})$, and thus a unique candidate solution $\widetilde{x}$.

3) If the Number of Runs Decreases by Two: If $r(y) = r(x)-2$, it might be that the substitution decreases the number of runs by two: $r(x') = r(x)$ and $r(y) = r(x') - 2$; or that the deletion decreases the number of runs by two: $r(x') = r(x) - 2$ and $r(y) = r(x')$. We will treat these two sub-cases separately, and we will show that in each of these sub-cases we can uniquely recover x.

a) Substitution decreases the number of runs by two: We consider the case in which the substitution decreases the number of runs by two, i.e., $r(x') = r(x)$ and $r(y) = r(x')-2$. Since the substitution decreases the number of runs, it must flip a bit b in a b-run of length one, i.e., $y_{e-\delta-1} = y_{e-\delta+1} = y_{e-\delta}$, and we also have

$$f_1^r(x) - f_1^r(y) = r_d^x + (1 + 2(n - e + \delta)). \tag{23}$$

We will proceed by case analysis based on the value of x_d and x_e.

b) If $x_d = x_e = b$: We first find a valid pair $(\widetilde{d},\widetilde{e})$ such that $\widetilde{d}$ is as small as possible. When $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1-b)$-run of length $\ell \geq 1$. Henceforth, in this elementary move $\widetilde{e}$ must to move to the left by ℓ so that $f(\widetilde{x}) = f(x)$, according to (20). During such moves, $r_{\widetilde{d}}^{\widetilde{x}}$ strictly increases, while $2(n-e+\delta)$ is non-decreasing. As a result, by (23), $f_1^r(\widetilde{x})$ increases during each elementary move. For the whole process, $f_1^r(x)$ increases by at most $2n$. Therefore, analogously to previous cases, $f(x)$ and $f_1^r(x)$ together with y uniquely determine a valid position pair $(\widetilde{d},\widetilde{e})$, and thus a unique candidate solution $\widetilde{x} = x$.

c) If $x_d = 1-x_e = b$: We first find a valid pair $(\widetilde{d},\widetilde{e})$ such that $\widetilde{d}$ is as small as possible. When $\widetilde{d}$ makes an elementary move to the right, it must pass across a $(1-b)$-run of length $\ell \geq 1$. According to (20), $\widetilde{e}$ needs to move to the right by ℓ to ensure that $f(\widetilde{x}) = f(x)$.

- If this is not a take over step: Suppose that the valid pairs are $(\widetilde{d}_1,\widetilde{e}_1)$ and $(\widetilde{d}_2,\widetilde{e}_2)$ before and after the move, respectively. During each such move, the run number $r_{\widetilde{d}}^{\widetilde{x}}$ increases by 2 while $[1+2(n-\widetilde{e}+\widetilde{\delta})]$ decreases by 2ℓ. By (23), if $\ell > 1$, then $f_1^r(\widetilde{x})$ decreases; if $\ell = 1$, then $f_1^r(\widetilde{x})$ does not change.
  However, when $\ell = 1$, the rank of $\widetilde{d}_1$ and $\widetilde{d}_1 + 1$ in $\widetilde{x}_1$ is smaller than that in $\widetilde{x}_2$, while the rank of $\widetilde{e}_1$ and $\widetilde{e}_2$ in $\widetilde{x}_1$ is larger than that in $\widetilde{x}_2$. By Lemma 10, $f_2^r(\widetilde{x}_1) > f_2^r(\widetilde{x}_2)$. This implies that during such elementary moves, either $f_1^r(\widetilde{x})$ strictly decreases, or $f_2^r(\widetilde{x})$ strictly decreases.
- If this is a take over step: During each such move, the run number $r_{\widetilde{d}}^{\widetilde{x}}$ increases by 2 while $[1+2(n-\widetilde{e}+\widetilde{\delta})]$ decreases by $2\ell+2$. Therefore, $f_1^r(\widetilde{x})$ decreases.

During the whole process, $f_1^r(x)$ decreases by at most $2n$, while $f_2^r(x)$ decreases by at most $4n^2$. Consequently, $f(x)$ and $f_1^r(x)$ together with y uniquely determine a valid pair $(\widetilde{d},\widetilde{e})$, and thus a unique candidate solution $\widetilde{x} = x$.

d) Deletion decreases the number of runs by two: In this section we consider the case in which $r(x') = r(x) - 2$ and $r(y) = r(x')$. This means that the substitution happens at the beginning or end of a run. Moreover, the deletion pattern satisfies $x_{d-1} = x_{d+1} = 1-x_d$. Let γ be an indicator variable such that $\gamma = 1$ if the substitution is at the end of a run in y, and $\gamma = -1$ if the substitution is at the beginning of a run in y. Then we have that

$$f_1^r(x) - f_1^r(y) = r_d^x + 2(n+1-d) + \gamma. \tag{24}$$

We now proceed by case analysis cased on the value of x_d and x_e.

e) If $x_d = x_e = b$: We first find a valid pair $(\widetilde{d},\widetilde{e})$ such that $\widetilde{d}$ is as small as possible. There are two kinds of elementary moves of $\widetilde{d}$.

- The index $\widetilde{d}$ moves within a $(1-b)$-run of length at least two: In this case, when $\widetilde{d}$ moves to the right by ℓ, $\widetilde{e}$ has to move to the left by ℓ so that $f(\widetilde{x}) = f(x)$. In such a move, $r_{\widetilde{d}}^{\widetilde{d}}$ does not change, while $2(n+1-d)$ decreases by 2ℓ. According to (24), $f_1^r(\widetilde{x})$ decreases if $\ell > 1$. When $\ell = 1$, if γ changes from -1 to 1, the run-based sketch $f_1^r(\widetilde{x})$ remains unchanged by (24). This implies that during this move, $\widetilde{e}$ moves from the beginning of a b-run in y to the end of a b-run in y. Otherwise, the substitution will affect the number of runs. However, $\widetilde{e}$ has to move to the left by at least two to match the error pattern. Thus, it must be that $\ell > 1$.
- The index $\widetilde{d}$ moves across a b-run of length ℓ: In this case, $\widetilde{d}$ moves to the right by $\ell + \ell'$ for $\ell' \geq 2$, $\widetilde{e}$ has to move to the left by ℓ' such that $f(\widetilde{x}) = f(x)$. During such moves, $f_1^r(\widetilde{x})$ decreases.

For the whole process, $f_1^r(x)$ decreases by at most $2n$. Therefore, $f(x)$ and $f_1^r(x)$, together with y, uniquely determine one position pair $(\widetilde{d},\widetilde{e})$, and thus a unique candidate solution $\widetilde{x}$.

f) If $x_d = 1 - x_e = b$: We first find a valid pair $(\widetilde{d},\widetilde{e})$ such that $\widetilde{d}$ is as small as possible. There are two kinds of elementary moves of $\widetilde{d}$.

- The first one is that when $\widetilde{d}$ moves within a $(1-b)$-run of length at least two. In this case, when $\widetilde{d}$ moves to the right by ℓ, $\widetilde{e}$ has to move to the right by ℓ such that $f(\widetilde{x}) = f(x)$. During such moves, $f_1^r(\widetilde{x})$ is non-increasing. Moreover, $f_1^r(\widetilde{x})$ only remains unchanged during such moves if $\ell = 1$ and $\widetilde{e}$ moves from the beginning of a b-run of length two to the end of this b-run; otherwise the substitution will affect the number of runs. During such a move, $f_2^r(\widetilde{x})$ is increasing.
- The second one is that when $\widetilde{d}$ moves cross a b-run of length ℓ. In this case, $\widetilde{d}$ moves to the right by $\ell + \ell'$ for some $\ell' \geq 2$, $\widetilde{e}$ has to move to the right by ℓ' to match $f(x)$. During such moves, $f_1^r(\widetilde{x})$ decreases.

For the whole process, $f_1^r(x)$ decreases by at most $2n$, while $f_2^r(x)$ increases by at most $4n^2$. Therefore, $f(x)$, $f_1^r(x)$ and $f_2^r(x)$, together with y, uniquely determine a position pair $(\widetilde{d},\widetilde{e})$, and thus a unique candidate solution $\widetilde{x}$.

Fig. 3. An example of $\widetilde{x}_1 = \widetilde{x}_2$, where $\widetilde{x}_1$ is a candidate solution with $\gamma = 1$ while $\widetilde{x}_2$ is a candidate solution with $\gamma = -1$.

Fig. 4. An example of $\widetilde{x}_1 = \widetilde{x}_2$, where $\widetilde{x}_1$ is a candidate solution with $\gamma = 1$ while $\widetilde{x}_2$ is a candidate solution with $\gamma = -1$.

4) If the Number of Runs Does Not Change: If $r(y) = r(x)$, it might be that both errors do not change the number of runs: $r(x') = r(x)$ and $r(y) = r(x')$; or that deletion reduces two runs while substitution increases two runs: $r(x') = r(x) - 2$ and $r(y) = r(x') + 2$.

a) Both errors do not change the number of runs: In this case, let γ be an indicator variable satisfying $\gamma = 1$ if the substitution is at the end of a run in y, and $\gamma = -1$ if the substitution is at the beginning of a run in y. We have

$$f_1^r(x) - f_1^r(y) = r_d^x + \gamma. \tag{25}$$

Moreover,

$$f_2^r(x) - f_2^r(y) = \begin{cases} r_d^x(r_d^x - 1) + 2(r_e^x - 1), & \text{if } \gamma = 1, \\ r_d^x(r_d^x - 1) - 2\, r_e^x, & \text{if } \gamma = -1. \end{cases} \tag{26}$$

We now proceed by case analysis cased on the value of x_d and x_e.

b) If $x_d = x_e = b$: We have at most two solutions, one with $\gamma = 1$ and one with $\gamma = -1$. Assume that the pair $(\widetilde{d}_1, \widetilde{e}_1)$ corresponding to $\gamma = 1$ yields codeword $\widetilde{x}_1$, and that the pair $(\widetilde{d}_2, \widetilde{e}_2)$ corresponding to $\gamma = -1$ yields codeword $\widetilde{x}_2$. We now show that if $f(\widetilde{x}_1) = f(\widetilde{x}_2)$, $f_1^r(\widetilde{x}_1) = f_1^r(\widetilde{x}_2)$, and that $f_2^r(\widetilde{x}_1) = f_2^r(\widetilde{x}_2)$, then $\widetilde{x}_1$ must be same as $\widetilde{x}_2$.

Note that since $f_1^r(\widetilde{x}_1) = f_1^r(\widetilde{x}_2)$and $f_2^r(\widetilde{x}_1) = f_2^r(\widetilde{x}_2)$, by (25) and (26), we have

$$r_{\widetilde{d}_2}^{\widetilde{x}_2} = r_{\widetilde{d}_1}^{\widetilde{x}_1} + 2,$$
$$r_{\widetilde{d}_1}^{\widetilde{x}_1}(r_{\widetilde{d}_1}^{\widetilde{x}_1} - 1) + 2(r_{\widetilde{e}_1}^{\widetilde{x}_1} - 1) = r_{\widetilde{d}_2}^{\widetilde{x}_2}(r_{\widetilde{d}_2}^{\widetilde{x}_2} - 1) - 2\, r_{\widetilde{e}_2}^{\widetilde{x}_2}.$$

This means that $r_{\widetilde{e}_1}^{\widetilde{x}_1} + r_{\widetilde{e}_2}^{\widetilde{x}_2} = r_{\widetilde{d}_1}^{\widetilde{x}_1} + r_{\widetilde{d}_2}^{\widetilde{x}_2}$, or, equivalently,

$$r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} + r_{\widetilde{e}_2 - \widetilde{\delta}_2}^{y} = r_{\widetilde{d}_1 - 1}^{y} + r_{\widetilde{d}_2 - 1}^{y}, \tag{27}$$

where δ_1 (δ_2) is the indicator of whether $\widetilde{d}_1$ is smaller than $\widetilde{e}_1$ ($\widetilde{d}_2$ is smaller than $\widetilde{e}_2$, respectively). Since both $\widetilde{d}_1$ and $\widetilde{d}_2$ do not change the runs, there exists a single $(1-b)$-run of length ℓ between $\widetilde{d}_1$ and $\widetilde{d}_2$ in y, and that $\widetilde{e}_2 - \widetilde{e}_1 = \ell$.

- If $r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} > r_{\widetilde{d}_2 - 1}^{y}$: By (27), it must be that $r_{\widetilde{e}_2 - \widetilde{\delta}_2}^{y} < r_{\widetilde{d}_1 - 1}^{y}$. That is to say,

$$\widetilde{e}_2 - \widetilde{\delta}_2 < \widetilde{d}_1 - 1 < \widetilde{d}_2 - 1 < \widetilde{e}_1 - \widetilde{\delta}_1.$$

  However, this contradicts the fact that $\widetilde{e}_2 - \widetilde{e}_1 = \ell$.

- If $r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} = r_{\widetilde{d}_2 - 1}^{y}$: This is impossible since $y_{\widetilde{e}_1 - \widetilde{\delta}_1} = 1 - b$ while $y_{\widetilde{d}_2 - 1} = b$.
- If $r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} < r_{\widetilde{d}_2 - 1}^{y}$: By (27) and the fact that $f(\widetilde{x}_1) = f(\widetilde{x}_2)$, it must be that $r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} = r_{\widetilde{d}_1 - 1}^{y} + 1$, and $r_{\widetilde{e}_2 - \widetilde{\delta}_2}^{y} = r_{\widetilde{d}_2 - 1}^{y} - 1$, i.e., $\widetilde{e}_1 - \widetilde{\delta}_1$ is the end of the $(r_{\widetilde{d}_1 - 1}^{y} + 1)$-th run in y, and $\widetilde{e}_2 - \widetilde{\delta}_2$ is the beginning of this run. In this case, we must have $\widetilde{x}_1 = \widetilde{x}_2$. See Figure 3 for an example.

Therefore, in this case, $f(x)$, $f_1^r(x)$, and $f_2^r(x)$ together with y uniquely determine a valid pair $(\widetilde{d}, \widetilde{e})$ and thus a unique candidate solution $\widetilde{x} = x$.

c) If $x_d = 1 - x_e = b$: We have at most two solutions, one with $\gamma = 1$ and one with $\gamma = -1$. Assume that the pair $(\widetilde{d}_1, \widetilde{e}_1)$ corresponding to $\gamma = 1$ yields codeword $\widetilde{x}_1$, and that the pair $(\widetilde{d}_2, \widetilde{e}_2)$ corresponding to $\gamma = -1$ yields codeword $\widetilde{x}_2$. By a similar argument as above, it must be that $r_{\widetilde{e}_1 - \widetilde{\delta}_1}^{y} = r_{\widetilde{d}_1 - 1}^{y}$, $r_{\widetilde{e}_2 - \widetilde{\delta}_2}^{y} = r_{\widetilde{d}_2 - 1}^{y}$. These two position pairs actually yield the same result $\widetilde{x}_1 = \widetilde{x}_2$. See Figure 4 for an example.

Therefore, $f(x)$, $f_1^r(x)$, and $f_2^r(x)$ together with y uniquely determine a valid pair $(\widetilde{d}, \widetilde{e})$, and thus a unique candidate solution $\widetilde{x} = x$.

d) Deletion reduces the number of runs while the substitution increases the number of runs: In this case, we have

$$f_1^r(x) - f_1^r(x') = r_d^x + 2(n + 1 - d),$$
$$f_1^r(x') - f_1^r(y) = -1 - 2(n - e).$$

Therefore,

$$f_1^r(x) - f_1^r(y) = [r_d^x + 2(n+1-d)] - [1 + 2(n - e + \delta)]. \tag{28}$$

We now proceed by case analysis cased on the value of x_d and x_e.

e) If $x_d = x_e = b$: We first find a valid pair of $\widetilde{d}$ and $\widetilde{e}$ such that $\widetilde{d}$ is as small as possible. When $\widetilde{d}$ makes an elementary move to the right, $\widetilde{e}$ has to move to the left to match $f(x)$. During such moves, $r_{\widetilde{d}}^{\widetilde{x}} + 2(n + 1 - d)$ decreases, while $1 + 2(n - e + \delta)$ is non-decreasing. By (28), $f_1^r(\widetilde{x})$ decreases. During this process, $f_1^r(x)$ decreases by at most $4n$. By a similar argument as above, $f(x)$, $f_1^r(x)$, together with y uniquely determine a position pair $(\widetilde{d}, \widetilde{e})$, and thus a unique candidate solution $\widetilde{x}$.

f) If $x_d = 1 - x_e = d$: We first find a valid pair $(\widetilde{d}, \widetilde{e})$ such that $\widetilde{d}$ is as small as possible. When $\widetilde{d}$ moves to the right, $\widetilde{e}$ has to move to the right to make sure than $f(\widetilde{x}) = f(x)$. During such moves, $f_2^r(\widetilde{d})$ increases if $\widetilde{e} > \widetilde{d}$, and decreases if $\widetilde{e} < \widetilde{d}$ by a similar argument in Section V-C.2. Therefore, we have at most two solutions, $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$, where $\widetilde{d}_1 < \widetilde{e}_1$, $\widetilde{d}_2 > \widetilde{e}_2$, such that $f(\widetilde{x}_1) = f(\widetilde{x}_2) = f(x)$, and meanwhile, $f_2^r(\widetilde{x}_1) = f_2^r(\widetilde{x}_2) = f_2^r(x)$. Since $f_2^r(x)$ increases by at most $4n^2$ when $\widetilde{e} > \widetilde{d}$ and decreases by at most $4n^2$ when $\widetilde{e} < \widetilde{d}$, $f(x)$ and $f_2^r(x)$ together determine at most two possible position pairs $(\widetilde{d}_1, \widetilde{e}_1)$ and $(\widetilde{d}_2, \widetilde{e}_2)$, and thus at most two candidate solutions $\widetilde{x}_1$ and $\widetilde{x}_2$.

D. A Linear-Time Decoder

We now proceed to describe our decoding procedure for $\mathcal{C}$ based on the case analysis described in Sections V-B and V-C:

1) If $|y| = n$, i.e., there is no deletion, we use the VT sketch $f(x)$ to decode directly.
2) If $|y| = n - 1$, i.e., the error includes one deletion at some index x and one substitution at index e, we check $h_r(x) - h_r(y)$. By comparing $h(x)$ and $h(y)$ we can also recover the value of x_d and x_e.
   a) If the number of runs increases by two ($h_r(x) - h_r(y) = -2$), the analysis in Section V-C.1 shows that we can recover a unique candidate solution $\widetilde{x} = x$ that matches all the sketches simultaneously in linear time.
   b) If the number of runs decreases by four ($h_r(x) - h_r(y) = 4$), the analysis in Section V-C.2 implies we can recover at most two candidate solutions $\widetilde{x}_1$ and $\widetilde{x}_2$ that match all the sketches simultaneously in linear time, and we are guaranteed that $x \in \{\widetilde{x}_1, \widetilde{x}_2\}$.
   c) If the number of runs increases by two ($h_r(x) - h_r(y) = 2$), the analysis in Section V-C.3 implies that we can uniquely recover x in linear time *if* we know which of the errors (deletion or substitution) reduced the number of runs by 2. This property yields a linear time list-size 2 decoder as follows. We run the decoder for the two different cases above on y. We are guaranteed that one of the decoders will behave correctly and output x. The other decoder may behave arbitrarily, but we know that if it outputs more than one (possibly erroneous) candidate string then it is not the correct decoder, and we can then disregard its output. Therefore, in the worst case we obtain a list of size 2 containing x.
   d) If the number of runs does not change ($h_r(x) = h_r(y)$), the analysis in Section V-C.2 implies that we can uniquely recover x in linear time *if* we know whether both errors did not affect the number of runs or whether the deletion reduced the number of runs by 2 while the substitution increased it by 2. By an analogous argument to the previous item where we run both decoders, this property implies that we can recover a list of size 2 containing x in linear time.

Therefore, we can list decode from one deletion and one substitution with a list of size at most two in time $O(n)$.

E. A Linear-Time Encoder

In Sections V-B and V-C we gave a list decoding procedure that corrects one deletion and at most one substitution given knowledge of the VT sketch (15), the run-based sketches (16) and (17), and the count sketches in (18). In this section, we describe a linear-time encoding procedure for a slightly modified version of the code $\mathcal{C}$ defined in (19) with redundancy $4\log n + O(\log\log n)$ which inherits the same list decoding procedure and properties from Section V-D. This approach is standard and very similar to Section III-F.

Consider an arbitrary input string $x \in \{0,1\}^m$ for some fixed message length m. Let $(\overline{\mathsf{Enc}}, \overline{\mathsf{Dec}})$ denote the efficient encoding and decoding procedures of a code for messages of length

$$\ell = |f(x) \| f_1^r(x) \| f_2^r(x) \| h(x) \| h_r(x)|$$

correcting one deletion and one substitution (here, we represent the sketches via their binary representations). For example, we may take the code from [9], which has redundancy $6\log\ell + 8 = O(\log\log m)$. Let

$$u = \overline{\mathsf{Enc}}(f(x) \| f_1^r(x) \| f_2^r(x) \| h(x) \| h_r(x)).$$

Then, we take final encoding procedure Enc to be

$$\mathsf{Enc}(x) = x \| u \in \{0,1\}^n,$$

which runs in time $O(m) = O(n)$ with overall redundancy $|u| = 4\log n + O(\log\log n)$.

We now describe a linear-time decoding procedure. Suppose that $\mathsf{Enc}(x)$ is corrupted into a string y via at most one deletion and one substitution. First, note that we can recover u by running $\overline{\mathsf{Dec}}$ on the last $|u| - 1$ bits of y. Then, using the linear-time decoding procedure described in Section V-D, we can recover a list of size at most two containing x from u (which encodes the necessary sketches) and $y' = y[1 : m-1]$. This yields Theorem 3.

VI. Open Problems

Our work leaves open several natural avenues for future research. We highlight a few of them here:

- Given the effectiveness of weighted VT sketches in the construction of nearly optimal non-binary single-edit correcting codes in Section III with fast encoding and decoding, it would be interesting to find further applications of this notion.
- The code we designed in Section IV fails to correct an arbitrary substitution. Roughly speaking, the reason behind this is that one substitution may simultaneously destroy and create a marker with a different starting point. As the clear next step, it would be interesting to show the existence of a binary code correcting one *edit* error or one transposition with redundancy $\log n + O(\log\log n)$.

Fig. 5. Transforming a single deletion into one deletion and one substitution, when the single deletion does not change the number of runs.

Fig. 6. Transforming a single deletion into one deletion and one substitution, when the single deletion reduces two runs.

- We believe that the code we introduce and analyze in Section V is actually uniquely decodable under one deletion and one substitution. Proving this would be quite interesting, since then we would also have explicit uniquely decodable single-deletion single-substitution correcting codes with redundancy matching the existential bound, analogous to what is known for two-deletion correcting codes [10].

APPENDIX

In this section, we provide a concrete instantiation of the sketch $\widehat{f}$ used in Section IV-B which is implicit in [8]. Let $L' = 2L + 1$. We claim that we may take $\widehat{f} : \{0,1\}^{L'} \to \{0,1\}^{\ell}$ of the form

$$\widehat{f}(z) = \mathsf{bin}\left(\sum_{i=1}^{n} i z_i \mod (L'+1), \sum_{i=1}^{n} i\overline{z}_i \mod (2L'+1)\right),$$

where bin denotes binary expansion up to $\lceil \log(2L'+1) \rceil$ bits and $\overline{z}_i = \sum_{j=1}^{i} z_j \mod 2$. Note that in this case $\ell = O(\log L)$, as desired.

It remains to see that $\widehat{f}$ above satisfies the desired property. Suppose that y is obtained from $z \in \{0,1\}^{L'}$ via at most one deletion or one transposition. Our goal is to show that we can determine z uniquely from y and $\widehat{f}(z)$. First, note that we can detect if a deletion occurred by computing $|y|$. If $|y| = L' - 1$, then, as shown by Levenshtein [3], we can use y and the first part of $\widehat{f}(z)$ to recover z. Else, if $|y| = L'$, then we observe that an adjacent transposition in z is equivalent to a substitution in $\overline{z}$. Therefore, as shown as well by Levenshtein [3], we can use y and the second part of $\widehat{f}(z)$ to recover z since there is a unique correspondence between z and $\overline{z}$.

We proceed by case analysis:

- If the single deletion of b in the i-th run does not change the number of runs, then it is equivalent to one deletion of $1-b$ in the $(i-1)$-th run and one substitution at the beginning of the i-th run. See Figure 5 for an example.
- If the single deletion of b in the i-th run reduces the number of runs by two, then it is equivalent to one deletion in the $(i-1)$-th run and a substitution in the i-th run. See Figure 6 for an example.

REFERENCES

[1] S. M. H. T. Yazdi, R. Gabrys, and O. Milenkovic, "Portable and error-free DNA-based data storage," *Sci. Rep.*, vol. 7, p. 5011, Jul. 2017.
[2] L. Organick *et al.*, "Random access in large-scale DNA data storage," *Nature Biotechnol.*, vol. 36, no. 3, p. 242, 2018.
[3] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Doklady Akademii Nauk*, vol. 163, no. 4, pp. 845–848, 1965.
[4] R. R. Varshamov and G. M. Tenengolts, "Codes which correct single asymmetric errors," *Autom. Remote Control*, vol. 26, no. 2, pp. 286–290, 1965.
[5] N. J. A. Sloane, "On single-deletion-correcting codes," 2002, *arXiv math/0207197*.
[6] K. Cai, Y. M. Chee, R. Gabrys, H. M. Kiah, and T. T. Nguyen, "Correcting a single indel/edit for DNA-based data storage: Linear-time encoders and order-optimality," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3438–3451, Jun. 2021.
[7] D. Tan. (2020). *Implementation of Single-Edit Correcting Code*. [Online]. Available: https://github.com/dtch1997/single-edit-correcting-code
[8] R. Gabrys, E. Yaakobi, and O. Milenkovic, "Codes in the Damerau distance for deletion and adjacent transposition correction," *IEEE Trans. Inf. Theory*, vol. 64, no. 4, pp. 2550–2570, Apr. 2018.
[9] I. Smagloy, L. Welter, A. Wachter-Zeh, and E. Yaakobi, "Single-deletion single-substitution correcting codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2020, pp. 775–780.
[10] V. Guruswami and J. Hastad, "Explicit two-deletion codes with redundancy matching the existential bound," *IEEE Trans. Inf. Theory*, vol. 67, no. 10, pp. 6384–6394, Oct. 2021.
[11] R. Gabrys, V. Guruswami, J. Ribeiro, and K. Wu, "Beyond single-deletion correcting codes: Substitutions and transpositions," 2021, *arXiv:2112.09971*.
[12] W. Song, K. Cai, and T. T. Nguyen, "List-decodable codes for single-deletion single-substitution with list-size two," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2022, pp. 1004–1009.
[13] J. Brakensiek, V. Guruswami, and A. Zbarsky, "Efficient low-redundancy codes for correcting multiple deletions," *IEEE Trans. Inf. Theory*, vol. 64, no. 5, pp. 3403–3410, May 2018.
[14] J. Sima and J. Bruck, "On optimal k-deletion correcting codes," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3360–3375, Jun. 2021.
[15] R. Gabrys and F. Sala, "Codes correcting two deletions," *IEEE Trans. Inf. Theory*, vol. 65, no. 2, pp. 965–974, Feb. 2019.
[16] K. Cheng, Z. Jin, X. Li, and K. Wu, "Deterministic document exchange protocols, and almost optimal binary codes for edit errors," in *Proc. IEEE 59th Annu. Symp. Found. Comput. Sci. (FOCS)*, Oct. 2018, pp. 200–211.

[17] B. Haeupler, "Optimal document exchange and new codes for insertions and deletions," in *Proc. IEEE 60th Annu. Symp. Found. Comput. Sci. (FOCS)*, Nov. 2019, pp. 334–347.
[18] 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.
[19] A. Lenz and N. Polyanskii, "Optimal codes correcting a burst of deletions of variable length," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2020, pp. 757–762.
[20] S. Wang, J. Sima, and F. Farnoud, "Non-binary codes for correcting a burst of at most 2 deletions," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2021, pp. 2804–2809.
[21] Y. Tang and F. Farnoud, "Error-correcting codes for short tandem duplication and edit errors," *IEEE Trans. Inf. Theory*, vol. 68, no. 2, pp. 871–880, Feb. 2022.
[22] W. Song, N. Polyanskii, K. Cai, and X. He, "On multiple-deletion multiple-substitution correcting codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jul. 2021, pp. 2655–2660.
[23] A. Wachter-Zeh, "List decoding of insertions and deletions," *IEEE Trans. Inf. Theory*, vol. 64, no. 9, pp. 6297–6304, Sep. 2018.
[24] V. Guruswami, B. Haeupler, and A. Shahrasbi, "Optimally resilient codes for list-decoding from insertions and deletions," in *Proc. 52nd Annu. ACM SIGACT Symp. Theory Comput.*, Jun. 2020, pp. 524–537.
[25] M. C. Davey and D. J. C. MacKay, "Reliable communication over channels with insertions, deletions, and substitutions," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 687–698, Feb. 2001.
[26] M. Cheraghchi and J. Ribeiro, "An overview of capacity results for synchronization channels," *IEEE Trans. Inf. Theory*, vol. 67, no. 6, pp. 3207–3232, Jun. 2021.
[27] L. J. Schulman and D. Zuckerman, "Asymptotically good codes correcting insertions, deletions, and transpositions," *IEEE Trans. Inf. Theory*, vol. 45, no. 7, pp. 2552–2557, Nov. 1999.
[28] K. Cheng, Z. Jin, X. Li, and K. Wu, "Block edit errors with transpositions: Deterministic document exchange protocols and almost optimal binary codes," in *Proc. 46th Int. Colloq. Automata, Lang., Program. (ICALP)*, C. Baier, I. Chatzigiannakis, P. Flocchini, and S. Leonardi, Eds. Wadern, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, p. 37.
[29] B. Haeupler and A. Shahrasbi, "Synchronization strings: Explicit constructions, local decoding, and applications," in *Proc. 50th Annu. ACM SIGACT Symp. Theory Comput.*, Jun. 2018, pp. 841–854.
[30] T. Klove, "Codes correcting a single insertion/deletion of a zero or a single peak-shift," *IEEE Trans. Inf. Theory*, vol. 41, no. 1, pp. 279–283, Jan. 1995.
[31] G. M. Tenengolts, "Nonbinary codes, correcting single deletion or insertion (Corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-30, no. 5, pp. 766–769, Sep. 1984.

Ryan Gabrys (Member, IEEE) received the B.S. degree in mathematics and computer science from the University of Illinois at Urbana–Champaing in 2005, and the Ph.D. degree in electrical engineering from the University of California, Los Angeles in 2014. He is currently a Scientist jointly affiliated with the Naval Information Warfare Center and the California Institute for Telecommunications and Information Technology (Calit2) at the University of California, San Diego. His research interests include theoretical computer science and electrical engineering, coding theory, combinatorics, and communication theory.

Venkatesan Guruswami (Fellow, IEEE) received the bachelor's degree from the IIT Madras, Chennai, in 1997, and the Ph.D. degree from MIT in 2001. He was a Professor of computer science at Carnegie Mellon University. He is currently a Senior Scientist with the Simons Institute for the Theory of Computing and a Professor of EECS and mathematics with the University of California at Berkeley, Berkeley. His research interests include coding theory, pseudorandomness, approximate optimization, and computational complexity. He is a fellow of ACM and was a recipient of the Simons Investigator Award, the Presburger Award, the Packard and Sloan Fellowships, the ACM Doctoral Dissertation Award, and the IEEE Information Theory Society Paper Award. He is the President of the Computational Complexity Foundation. He currently serves as the Editor-in-Chief for the *Journal of the ACM*.

João Ribeiro received the B.Sc. degree in applied mathematics and computation from the Instituto Superior Técnico, Lisbon, Portugal, in 2015, the M.Sc. degree in computer science from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, in 2017, and the Ph.D. degree in computing from Imperial College London, U.K., in 2021. He is a Post-Doctoral Fellow with the Computer Science Department of Carnegie Mellon University. His research interests include coding theory and pseudorandomness along with their connections to cryptography and other areas of theoretical computer science.

Ke Wu received the bachelor's degree in information and computing science from the School of Mathematical Sciences Department, Fudan University, and the master's degree from Johns Hopkins University. She is currently pursuing the Ph.D. degree with the Computer Science Department of Carnegie Mellon University, under the supervision of Prof. Elaine Shi. She was a Research Assistant under the supervision of Prof. Xin Li at Johns Hopkins University. She works on the intersection of theoretical cryptography, game theory, and coding theory, as well as related areas in theoretical computer science.