# Locally Decodable Codes for Edit Distance

Rafail Ostrovsky[1] and Anat Paskin-Cherniavsky[2]

[1] UCLA, rafail@cs.ucla.edu
[2] UCLA, anpc@cs.ucla.edu

**Abstract.** Locally decodable codes (LDC) [1,5] are error correcting codes that allow decoding (any) individual symbol of the message, by reading only few symbols of the codeword. Consider an application such as storage solutions for large data, where errors may occur in the disks (or some disks may just crush). In such an application, it is often desirable to recover only small portions of the data (have random access). Thus, in such applications, using LDC provides enormous efficiency gains over standard error correcting codes (ECCs), that need to read the entire encoded message to learn even a single bit of information. Typically, LDC's, as well as standard ECC's decode the encoded messaged if upto some bounded fraction of the symbols had been modified. This corresponds to decoding strings of bounded Hamming distance from a valid codeword. An often more realistic metric is the edit distance, measuring the shortest sequence of insertions and deletions (indel.) of symbols leading from one word to another. For example, (few) indel. modifications is a more realistic model for mutations occurring in a genome. Even more commonly, communication over the web may sustain deletions (lost packets) and insertions (noise).[3] Standard ECC's for edit distance have been previously considered [7]. Furthermore, [7] devised codes with rate and distance (error tolerance) optimal upto constants. LDC's, originally considered in the setting of PCP's [1], have found many additional applications, and generated a lot of fascinating work (see [9] and references within). However, combining these two useful settings of LDC, and robustness against indel. errors has never been considered.
In this work, we study the question of constructing LDC's for edit distance. We demonstrate a strong positive result - LDC's for edit distance can be achieved, with similar parameters to LDC's for Hamming distance. More precisely, we devise a generic transformation from LDC for Hamming distance to LDC for edit distance with related parameters.

## 1 Introduction

In this work, we define and study the feasibility of locally decodable codes (LDC) for edit distance. Standard LDC codes are defined over the Hamming distance, allowing to decode individual symbols of the message by reading few symbols

[3] Edit distance is indeed "more expressive" then Hamming distance in the sense that $dist_E(x,y) \leq 2dist_H(x,y)$ always holds, while edit distance 2 may translate to Hamming distance $n$. For instance, consider $x = 1010\ldots10, y = 0101\ldots1$.

of the codeword. This provides enormous efficiency gains over standard error correcting codes (ECCs), that need to read the entire encoded message to learn even a single bit of information. Additionally, LDC's have a curious connection to cryptography, yielding protocols for PIR (private information retrieval) with related parameters, and vise versa [8].

We put forward a strong positive result, by demonstrating a compiler from standard LDC's into LDC's for edit distance, with only small losses to the parameters. In particular, the tolerated fraction of errors (typically a constant), and the code's rate are only degraded by a constant. The query complexity grows by polylogarithmic factors in the size of the codeword. The compiler is black box, in the sense that the LDC decoder for the resulting code uses the decoder of the original LDC code in a black box way (only reading and answering its queries to the purported codeword). Our main technique is reducing the task to the problem of searching an element in a large sorted list $L$ with a constant fraction $\delta$ of corrupted values. The search should succeed with overwhelming in $|L|$ probability for all but, say, $50\delta$ fraction of the queries into uncorrupted locations. The number of queries to the list should be polylogarithmic. We devise a comparison-based algorithm with $O(\log^{2+o(1)}|L|)$ queries for this task. This algorithm may be of independent interest, as for the more stringent setting where all uncorrupted entries should be recovered correctly, there exist polynomial lower bounds on the number of queries by comparison-based algorithms.

**Theorem 1.** *(Main thm., informal). Consider an $(\delta(n), q(n), \epsilon(n))$-LDC $L_H : \mathbb{F}_p^n \to \mathbb{F}_p^m$ for Hamming distance. Here $\delta(n)$ is some constant bound on the fraction of tolerated indel. errors, $q$ is the query complexity, and $\epsilon(n)$ is a bound on the worst case error in reading a message symbol if $\delta$ is respected. There exists a black box transformation from such $L_H$ into a $(c\delta, q \cdot polylog(m,p), \epsilon + neg(m))$-LDC $L_E : \mathbb{F}_p^n \to \mathbb{F}_p^m$ for edit distance, where $c$ is a (quite large) global constant. The code rate degrades by a constant. Encoding efficiency only degrades by a poly$(n)$ factor.*

The transformation from Theorem 1 is black box in the following sense. Let $D_H^{w_H}, D_E^{w_E}$ denote the decoders of the original LDC (for Hamming distance) and the decoder for the LDC for edit distance that we construct, respectively. $D_E^{w_E}$ receives an input $i$ and needs to decode $x_i$. Fur that purpose, it runs $D_H(i)$ as is, reading only the sequence of locations $D_H$ asks to query $w_H$ at, and answering them. To asnwer a query $j$ of $w_H$, it simulates the answer $w_H[j]$ using queries to its own oracle $w_E$, which "induces" a codeword $w_H$. Finally, the output of $D_H$ is returned.

On a high level, $L_E$ is a composition of $L_H$ with a standard code $I$ for edit distance. That is, to encode a message $m$, it computes $w_1 = L_H[m]$, divides $w_1$ into blocks $w_1, \ldots, w_T$, and outputs $I(w'_1) \circ \ldots \circ I(w'_T)$, where $w'_i$ is "almost" $w_i$. To answer a query on index $i$, the goal is to find the relevant block in $w_2$, and decode it to extract the relevant symbol. Even in standard codes, one central difficulty is in finding the block in $w_2$. For this purpose, the $w'_j$'s explicitly include their relative index: $w'_i = (i, w_i)$ . This transformation is in fact exactly the one used in SZ codes, where $L_H$ is replaced by a regular code $C$. The novelty of our construction, is in demonstrating that this transformation in fact preserves the parameters of LDC codes ( [7] show that for a careful choice of block length, it preserves distance and rate of standard ECC upto a constant), by devising a suitable decoder procedure. As we explain below, just adding the indices is not sufficient for LDC codes. If the entire codeword can be read, we can just "read off" the indices of all blocks, and use the (decoded) $(i, s)$ for $i$ in the right range as the values at location $i$. If there are relatively few errors, this will produce a $C$ codeword with few erasures (duplicate and missing entries) + changes (erroneous entries we do not know of), which can then be decoded. For LDC codes, the problem is in finding a block $w_i'$ by reading only $polylog(m)$ entries from the codeword (in particular, $|I(w_i')| = polylog(m)$. Although the location of the relevant block can move upto a $\delta\,|W_E|$ fraction of symbols, we should find it with high probability for "most" blocks.

Our main technical tool is a new algorithm for searching an element in a sorted list $L$ where up to a constant $\delta$ fraction of the original entries may be arbitrarily modified (and possibly out of order), looking at only $polylog|L|$ locations of the list. The algorithm performs a "clever" version of binary search, coping with errors to the degree we need. It guarantees locating at least a $1 - c\delta$ fraction of the lists' entries. This technique may be of independent interest. The problem of searching sorted lists with corruptions or errors in query's answers has been considered before ( [6,2] to mention a few). The main difference of our setting is that we get much lower query complexity at the cost of allowing incorrect answers for some $c\delta$ of the uncorrupted entries. Without this compromise, comparison-based algorithms (ours included) the query complexity is provably $\Omega(poly(n))$ for constant error fractions [2].

We also observe that our technique for transforming codes for Hamming distance into codes for edit distance applies to the setting of computational LDC's [4], and of Locally testable codes (see [3] and references within).

## 1.1 Our Technique in more detail.

Our starting point is the construction by Shulman et al [7], that converts (standard) error correcting codes for Hamming distance into ones for edit distance. Their construction is a composition of two codes as follows.

1. Start with a standard "outer" ECC $C_1 : \mathbb{F}_p^n \to \mathbb{F}_p^m$, and apply it to the plaintext message $x$, obtaining $y$.
2. Encode $y$ under a greedily constructed code $I$ for edit distance as follows - denote this new code by $C_2$). Divide $y$ into blocks of $\log m$ symbols each, resulting in $T = m/logm$ blocks. Encode each block $y_i$ at the $i$'th block using an "inner" (greedily constructed, exponential-time) code $I$, applied to $(i \circ y_i)$, obtaining $w_i = I(i \circ y_i)$. Output $w = w_1 \circ w_2 \ldots \circ w_T$ as the codeword. The code $I : \mathbb{F}_p^{\log m} \to \mathbb{F}_p^h$ has constant rate and tolerates a constant fraction of

errors. (the number of blocks is selected as such to ensure constant rate of $E_2$)[4]

The resulting code $C_2 \circ C_1$ is a code for edit distance. Their goal is to obtain codes with efficient (in $n$) encoding and decoding procedures, and constant distance and rate. Thus, they plug in $C_1$ with constant distance and rate parameters, and $C_2 \circ C_1$ inherits these properties (due to also constant rate and distance for $I$). The reason that SZ do not just use $I$ as the code for edit distance is its inefficiency of encoding and decoding, so it can only be practically applied to short blocks.

Besides constant (edit) distance, another property of the code $I$ that they need, is that for every pair of different codewords, the distance between a prefix $w_1$ of $w$, and a suffix $u_1$ of $u$ (or vise versa) of large enough (fractional) length, say 0.1, have large distance. This ensures that in a corrtuped codeword $w$, subsequences "close enough" to different codewords do not intersect "by much". Thus, a corrupted $w$ can be viewed as a sequence of codewords (of $I$) and possibly garbage between them, written one after the other (possibly upto small fractional overlaps). This way, every original $w_i$ that was corrupted by "not too much" will be recovered when scanning a small vicinity of $w_i$ (as is useful for LDC's), or when scanning the entire $w$ from left to right (as is useful for standard decoding, like in SZ). In fact, we will use a slightly stronger "no overlapping" property that is implicit in [7]'s construction.

This construction suggests the following simple transformation from standard LDC's into LDC's for edit distance. Plug the (standard) LDC $C_1$ as the outer code (instead of a standard ECC) into the construction. The code $L_E = C_2 \circ C_1$ is our LDC for edit distance! A decoder for $L_E$ acts as follows.

Simulate $C_1$'s decoder $D_1$. For any query $D_1$ makes, decode the corresponding block (by going to its vicinity), and retrieve the relevant query by decoding $I$. At the end, output whatever $D_1$ outputs. If $I$ has edit distance $\delta_I$, then a $\delta_E$ fraction of errors will corrupt (beyond repair) a $\leq \delta_E/\delta_I$ fraction of the blocks, but the rest will be correctly recovered (if found!). If the original LDC tolerates a $\delta_H$ fraction of errors, we may set $\delta_E = \delta_H \delta_I$. As $D_1$ sees at most a $\delta_E$ fraction of corrupted symbols, the new decoder's decoding probability is the same as $D_1$'s.

The main problem is that it is unclear how to find the required blocks. Due to deletion and insertion errors occurring before it, every block can be as far as $\delta m$ symbols from its expected index (in the original sequence $w_1, \ldots, w_T$). To cope with this, we develop a clever binary search technique allowing to find it, even in the presence of corruptions. More precisely, we reduce our problem to the following problem of searching a sorted list $L$ of length $T$ (only known to us upto a factor of 2) where upto some (constant) $\delta$ fraction of the entries may have been corrupted. Entries of the list are of the form $(i, s_i)$, where the sorting is by the unique keys $i$ (upto duplications introduced by corruptions). One wants

[4] In [7], the authors devise and use $I$ with binary input and output alphabet. SZ is easy to modify to work over larger alphabets, possibly allowing for better (constant) parameters.

to learn $s$ associated with $i$ in the list (or that $i$ does not appear in it). Design an algorithm $V(i)$ that makes $polylog(T)$ queries into the list that returns the correct value $c$ with probability $1 - neg(T)$ for at least a $1 - c\delta$ fraction of the original $(i, s)$ entries, where $c$ is a constant independent of $\delta$.

The constant "loss of correctness" factor $c$ above will just translate into a further decrease in tolerated $\delta_E$, namely $\delta_E \leq delta_H \delta_I / c$, so we can afford it. Roughly speaking, in uncorrupted entries in $L$ will correspond to the list of blocks $(1, w_1), \ldots, (T, w_T)$ that we not "deleted", but either modified or newly inserted. The former blocks will retain their original order, where a block's corresponding key is a blocks index $i$, and $s = w_i$.

In Section 2.1 we present our algorithm for searching sorted lists with corruptions. In Section 2.2 we discuss how to adapt this abstraction to searching a symbol $w_i$ in a codeword $w$ of $C_2 \circ C_1$ as above. There are several technical issues that need to be carefully treated here. In particular, the type of queries we chose for the the sorted list searching abstraction are easy to (approximately) implement given $w$.

Another point where our construction diverges from the original construction of SZ is in the efficiency of $I$. As mentioned above, the inner code $I$ is constructed in a greedy manner, with exponential complexity in the message space. In our case,this size is upto $\log m$ symbols, so encoding and decoding of that code may have complexity $m^{\log p}$ (bit operations), which is prohibitively high in the setting of LDC and ok for standard decoding of the entire message (although overall efficiency of decoding is a secondary goal in LDC, it is important in practice). Thus, we use a recursive version of the inner code (also mentioned in the paper), where every block $(i \circ y_i)$ is encoded by a SZ code (based on, say, Reed Solomon as the outer code), so the greedy part is now applied to messages of length $\leq \log(\log m \cdot \log p)$ bits. This comes only at a constant decrease in tolerated errors and rate.

*Remark 2.* For some settings of parameters, one would just rather fall back to standard SZ codes for edit distance, that read the entire codeword to decode. As explained above, some kind of "binary" search seems inevitable. Thus, we expect to lose a factor of $\log m$ in the query complexity (even if we were willing to give up such a factor on rate). For some codes, such as the Hadamard code, $\log m = n$, so there is no gain in query complexity. Nevertheless, the construction is non-trivial for most useful parameter settings of LDC.

## 1.2 Preliminaries.

In this paper $\mathbb{F}_p$ denotes a finite alphabet of size $p$ (typically, but not necessarily a finite field). We denote the Hamming distance of two strings $x, y \in \mathbb{F}_p^l$ by $dist_H(x, y)$. The edit distance between $x, y \in \mathbb{F}_p^*$, $dist_E(x, y)$ is the minimal number of insertion or deletion operations to be performed on $x$ to obtain $y$ (or visa versa, as $dist_E$ is a metric). We often just write "$dist(x, y)$" when the type of distance is clear from the context.

For a metric $dist \in \{dist_H, dist_E\}$, we say $C : \mathbb{F}_p^n \to \mathbb{F}_p^m$ is an error correcting code (ECC) with distance parameter $d$ if for any pair of codewords, $C(x) \leq C(y)$, we have $dist(C(x), C(y)) \geq d$. Alternatively, we will often measure the number of errors the code can tolerate (upto $(d-1)/2$). The codes' rate is the ratio $m/n$.

By default, we consider families of ECC's $C : \mathbb{F}_p^n \to \mathbb{F}_p^{m(n)}$ (sometimes $p$ depends on $n$ as well), and discuss their asymptotic parameters.

**Definition 3.** *An ECC, $L : \mathbb{F}_p^n \to \mathbb{F}_p^m$ is a $(\delta(n), q(n), \epsilon(n))$-LDC (locally decodable code) for Hamming distance if there exists a decoding algorithm $D^{w'}(i)$ such that for all $i \in [n], x \in \mathbb{F}_p^n$, and all $w' \in \mathbb{F}_p^m$ satisfying $dist_H(w', L(x)) \leq \delta m$, we have*

$$Pr[D^{w'}(i) = x_i] \geq 1 - \epsilon.$$

*Here $D$ reads at most $q(n)$ locations in $w'$.*

**Definition 4.** *LDC for edit distance is defined as LDC for Hamming distance (replacing $dist_H$ with $dist_E$ everywhere), with the minor difference that $D^{w'}$ is also given $|w'|$ as an additional input.*

We use the following family of codes for edit distance implicit in [7].

**Lemma 5.** *For every finite alphabet $\mathbb{F}_p$, there exists an integer $t_0$ and real $\delta > 0$, such that for all $t \leq t_0, \delta' \leq \delta$, there exists an ECC $I_{t,\delta'} : \mathbb{F}^t \to \mathbb{F}_p^m$ for edit distance tolerating upto $\delta' m$ insertions and deletions. $I_{t,\delta'}$ has constant (possibly depending on $\delta'$) rate $m/t$. Also, there exist a (global) constant $c$ such that for all $I_{t,\delta'}$ as above satisfy the following "no overlapping" property. For every pair of codewords $(ws_1, s_2v)$, if $|s_1| = |s_2| \geq 2\delta m$, then either $ws_1 = s_2v$, or $dist(s_1, s_2) \geq 1.5\delta m$. Furthermore, if $|s_1|, |s_2| \leq (1-\delta)m$, then it must be the case that $dist(s_1, s_2) \geq 1.5\delta m$. The codes' encoding and decoding complexity is $poly(m)$.*

*Proof sketch.* Roughly, we consider the buffered code variant, and view the $S_1$ codeword from SZ along with the 1/2-buffer before and after it as a single codeword of $I$. That is, $I$ encodes as in $S_1$, and appends 1/2-buffers of 0's as in SZ from both sides (there this is done only when composing $C_1$ with $I$). Let $\delta$ be the error fraction tolerated by $S_1$ (for sufficiently large $t$), and $m'$ be its output length. Instead of taking large buffers (say, some constant fraction of an $S_1$ codeword) as in SZ, we set it it to just $\delta m$. Consider first $S_1$ constructed greedily, with 1's interleaved into the $S_1$ codeword every other symbol (in the binary case), then the "no overlapping" property is satisfied. Note that $I$ has distance $(\delta_I)$ at least $\delta$. The (1/2)-buffers have a 0 fraction of 1's, while any non-buffer part of a codeword it overlaps with has a 1/2 fraction of 1's (both are of length $\delta m$). Thus, making $s_1, s_2$ of length at least $2\delta m$, and at most $(1-\delta)m$, would put $(s_1, s_2)$ at edit distance at least $2\delta m$. If $|s_1|, |s_2| \geq 1 - \delta m$ for small enough $\delta$, it must be the case that $ws_1 = s_2v$. There are two cases to consider. If $s_1, s_2$ are of length $\geq (1-\delta/3)m$, the codewords $ws_1, s_2v$ have distance at most $2/3\delta m$, and thus can not be a pair of distinct valid codewords (which have distance at least $2\delta(1-2\delta)m$). Otherwise, if $s_2v$ is a valid codeword, $ws_1$ has $4/3\delta$ consecutive 0's in its prefix and thus is invalid. If $ws_1$ is a valid codeword, then a 1/3 of $s_2v$'s $\delta m$ suffix has density $\geq 1/2$ or 1's, where it should have an all-0 buffer, and thus not a valid codeword.

Finally, we do not simply let $S_1$ be the greedily constructed code from SZ. The reason is that encoding and decoding of such a code has $exp(m)$, which is too inefficient for our purposes. Thus, we replace $S_1$ with a code for edit distance obtained ad in SZ for their buffered version. Namely, by composing some $C_1$ with constant rate and fractional distance and efficient encoding and decoding for Hamming distance (such as RS code with suitable parameters), with $I$ with a suitable $t$ (around $\log m$). Break $y = C_1(x)$ into $T = \Theta(|y|/\log|y|)$ blocks $w_1, \ldots, w_T$, of length $m_1 = m/T$ each. It outputs $S_1(1, w_1), \overline{0}, \ldots, \overline{0}, S_1(T, w_T)$, where the $\overline{0}$ are buffers of length, say $m_1/8$. It has some distance $\delta_1$ and some constant rate (somewhat worse then that of the greedy code). This variant of $S_1$ has $poly(m)$ encoding and decoding procedures. Finally, plugging it instead of $S_1$ in the above construction leads to a density of slightly below 1/2 in a $\delta_1 m$-long stretch in the $S_1$-part of the word, approaching 0.4 as $m_1$ grows, and still a 0 density for the buffer. Thus, we get $c = 1.5$ for large enough $m_1$ (and small enough $\delta_1$). □

We will need the following version of the Chernoff's bound.

**Lemma 6.** *Let $X_1, \ldots, X_1$ denote independent random variables, and let $X = \sum_i X_i$. Assume also that the support of each is $[0, B]$, for some $B > 0$. Then for $\epsilon > 0$, we have*

$$Pr[|X - E(X)| \geq \epsilon E(X)] \leq e^{-\Theta(\epsilon^2)E(X)}$$

## 2 Searching sorted lists with corruptions

As explained in the introduction, proving Theorem 1 boils down to the developing a search algorithm on a sorted list with (small) constant fraction $\leq \delta$ of corrupted entries, making a polylog number of queries to the list. The list is comprised from pairs of $(i, s_i)$ where $i$ is a unique key (before corruptions). The input to the algorithm is a key $i$, and it should return a corresponding $s_i$. We require that for all lists $L$ with a $\delta$ fraction of corrupted entries, the algorithm returns $s_i$ on query $i$ with probability $1-neg(|L|)$ on all but some $1-c\delta$ fraction of the (original) entries, for a constant $c$ independent of $\delta$. Clearly, $c$ can not be less then 1. In particular, if only the $s_i$'s are modified (keys are intact), there is no way to recover the original keys.

There exist algorithms in the literature in a similar setting with stronger guarantees and worse parameters. For instance, [2] consider algorithms that guarantee to recover $s_i$ for all values for which $(i, s_i)$ was not corrupted, and if there is no key $i$ in the sequence, corrupted or correct, the algorithm should output "not found". They prove that for such a stringent requirement, any comparison-based algorithm, that accesses some $\Omega(\log |L| + \delta \cdot |L|)$ locations errs on some input with probability at least $1/2$.[5]

### 2.1 Our approach

The natural approach to the question is to use binary search.

**A Warmup - random error locations.** To gain intuition, assume that the error locations were picked at random - each entry is corrupted with probability $\delta \leq \delta_0 = 0.2$. Jumping ahead, for our application to LDC for edit distance, this would happen if the insertions and deletions occurring are at random locations.

Assume that the success requirements of the algorithm need to only hold for "most" error patters (allowing high failure rates for all queries for a small fraction of error patterns). Then the following simple algorithm and analysis would work. Given a list of length $m = |L|$, proceed in levels, so that on every level we divide the interval at hand into three equal intervals (start with the entire list). For the middle interval, randomly and independently sample $\log^2 m$ entries, and record the fraction of keys smaller or larger then $i$, $(s, b)$ accordingly (for simplicity of analysis, sample with repitions). If some $(i, s)$ element is found, we stop and return $s$ corresponding to the first appearance of $i$ found immediately. Otherwise ($b = 1 - s$), return the corresponding $s$ for the first such appearance and terminate). If $s, b \geq 0.4$, proceed with the middle interval recursively. Otherwise, if $s > b$ proceed with intervals $(2, 3)$ as the new interval, if $b > s$ proceed with $(1, 2)$. We stop at intervals of size $\log^2 m$, and scan the entire interval; return $s$ corresponding to the first $(i, s)$ in the interval, or $\perp$ otherwise.

Quite straightforward analysis, implies that the above algorithm succeeds to achieve its goal for all but a small fraction of error patterns. One type of error is that of finding the wrong $(i, s)$ and terminating the search (even if the correct $(i, s)$ is located in a different interval). All these errors may only occur for at most $2\delta m$ of the keys $i$ originally present in $L$. Those which were "duplicated" elsewhere, and those that were modified into the new duplicates, possibly erasing their own information.

For all other entries (keys) on which the algorithm may err, entries of the form $(i, s)$ only appear in the interval that originally contains an entry $(i, s)$ for that key, and thus this type of error may not occur. In that case, only errors due to excluding an interval originally containing $i$ at some point along the recursion

[5] This is a certain restatement of their theorem 5, in terms of the number of elements involved in the comparison queries, rather then in terms of the number of comparisons made. The proof follows straightforwardly from their proof of that theorem. They demonstrate some matching upper bounds, leaving just a small gap. Still, even for an algorithm matching the lower bound perfectly, for the range of parameters where $\delta_0$ is a constant, the query complexity is $\Omega(|L|)$, which is unacceptable in our case. The key for obtaining a (comparison based) algorithm with query complexity $polylog(|L|)$, is the fact that we can rely on relaxed correctness guarantees as above.

may occur - we refer to these as type 2 errors. This type of errors is slightly trickier to bound - jumping ahead, it will occur with probability $> neg(m)$ for none of these other keys for this algorithm, and account for most of the errors of the algorithm for general errors.

A crucial point is that the probability (over picking error patters uniformly at random) that the search reaches an interval with density larger then $0.25 = 1.2\delta_0$ fraction of errors for *any* searched key $i$ is bounded by $poly(m) \cdot m^{-\Theta(\log m)}) = neg(m)$ (Chernoff + union bound). Here $poly(m)$ is a bound on the number of reachable nodes (for any key $i$) derived from $3^{\log_{1.5}(m)} = m^{\log_{1.5}(3)} \leq m^{2.71}$.

It is easy to see that if the latter happens, at every step of the recursion, the algorithm can make a type 2 error when moving to the next step of the recursion with probability at most $m^{\Theta(-\log m)}$.

This holds since if $i$ is in the first interval, (same holds for the 3rd interval), and assuming the error density in interval 2 is indeed $\leq 0.25$, then $\geq 0.75$ fraction of elements in 2 are bigger then $i$. Thus, having $s \geq 0.4$ (necessary for $b, s \geq 0.4$) is highly unlikely - recalling $(i, s)$ can not appear in interval 2 (we account for such keys $i$ in type 1 errors), $s$ has expectancy of at most most 1/3. By Chernoff bound, $s \geq 0.4$ thus has negligible in $m$ probability $exp(-\Theta(\log^2(m)))$. Otherwise, getting $s > b$ would require getting $s \geq 0.5$ for same expected value of 0.3 - again a $exp(-\Theta(\log^2(m)))$ probability.

Taking union bound over the path for any given key $i$, the overall error probability is $O(\log m \cdot m^{\Theta(-\log m)}) = neg(m)$.

The main difficulty is that for arbitrary error patterns, low density of errors in all intervals in not guaranteed. Thus, a more sophisticated analysis (for a somewhat more sophisticated algorithm, but quite along the lines of the one above) is required. In particular, the number of intervals we use will depend on $\delta_0$. The parameters tolerated by the above algorithm in this random errors setting are $c = 2$, and $\delta \leq 0.2$. The parameters we achieve for general errors will be worse $c = 50$, and $\delta$ bounded correspondingly ($\delta \leq 1/c$ to obtain any non-trivial correctness guarantee).

**The final protocol.** Let us first fully formalize our setting.

- The protocol is specified by an algorithm $S^L(i)$. It has oracle access to a sorted list $L$, where some constant $\delta$ fraction of elements have been corrupted (possibly not respecting the original order). Let $m'$ denote an approximation on $|L|$ upto a factor of 2 ($m'$ is available to $S$, while the exact $m = |S|$ is not). The input is a key $i$ to search.
- Oracle queries: There are two types of possible queries to the list. 1. $(v_0, v_1)$, where $(v_0, v_1)$ are fractional locations in the list. The entry $(i, s)$ at a randomly selected location inside the interval is returned. Query weight is 1. 2. Ask for the sequence of all points in an interval $(v_0, v_1, y)$. If the interval is of size $y$ or smaller, the sequence of all points in the interval will be returned. Otherwise, an error is returned. Query weight is $y$ (regardless of the query's outcome).

– Output: Given a key $i$, such that $(i, s_i)$ was present in the original list (before corruptions occurred), the correct output for it is $s_i$.
– Goal: Maximize the worst case fraction of keys $i$ originally present in the list (before corruptions) for which the reply is correct with probability $1 - neg(m)$. Total weight of queries should be $polylog(m)$ - we are not trying to optimize the concrete complexity.

**Construction 7** *Initialize the searched interval to $I = (0, 1)$, $\Delta = 3$ (or any, other constant $> 3$), $T = \Delta$, $r = \log^2 m'$. Repeat:*

1. *Make a type 2 query with $(I, r)$. If it returns a sequence of points, and one of them is of the form $(i, s)$, return $s$ corresponding to the first such $i$. Otherwise, return $\perp$. (we reached a short interval we can read completely)*
2. *Otherwise, divide $I$ into $T$ intervals $I_1, \ldots, I_T$ of equal size (up to $\pm 1$ due to rounding). Sample $r$ random locations in each of the intervals, resulting in $o_{i,1}, \ldots, o_{i,r}$ for the $i$'th interval.*
   (a) *If some sample is of the form $(i, s)$, return $s$ corresponding to the first such $i$.*
   (b) *Otherwise, for each interval, calculate the fractions $s, b = 1 - s$ of smaller and larger then $i$ sampled elements respectively. We say that interval $j$ votes against interval $k, k > j$ for $i$ , equivalently votes against $(I_k, i)$ if $b \geq 0.31$ (for $k < j$, if $s \geq 0.31$).Note that if $I_j$ votes against $(I_k, i)$, then it votes against all $(I_h, i)$ for $h$ on the same side of $I_j$ as $l_k$. We then say that $I_j$ votes against its left (right) side on $i$. For every interval $j$, we count the number of votes against $(I_j, i)$ over all other intervals.*
      i. *If there is exactly one interval with a minimum number of votes, fix $I$ to be that interval.*
      ii. *If there are two such adjacent intervals $I_j, I_{j+1}$ let $I$ be their union. Fix $T = 2\Delta$.*
      iii. *Otherwise, output $\perp$ and terminate.*

**Theorem 8.** *Construction 7 is an algorithm for searching on sorted lists (in a framework as defined above), tolerating a (small enough) constant[6] $\delta$ fraction of corruptions. For at least a $1 - 52\delta m$ fraction of the original lists' elements, it recovers them correctly with probability $\geq 1 - neg(m)$ ($\delta$ is the actual fraction of corruptions that occurred). It makes $O(\log^3 m)$ queries to the list.*

*Proof.* Throughout the analysis, we fix an arbitrary $L$ and set of corruptions $S$ of size $\delta |S| \leq \delta_0 |S|$ (but not the searched key $i$). We start level count at 1. On level $l$ of the algorithm, we refer to intervals $I$ that can be reached on that level as level-$l$ nodes (for instance, there is a single level-1 node). We refer to the $\Delta^i$ equal intervals partitioning the entire list as basic intervals (note that such intervals coincide with intervals considered in all level-$l$ nodes: such nodes contain either $\Delta$ or $2\Delta$ basic intervals). The size of an interval drops by a factor of at least $\Delta/2$

[6] Construction 7 can also handle subs-constant in $|L|$ $\delta$ with the same degradation factors, but constant $\delta$ is the most interesting parameter setting.

every time. Thus there are at most $\log_{\Delta/2}(m)$ iterations. In every iteration we make at most $2\Delta \log^2 m' = O(\log^2 m)$ queries of type 1, and queries weighting $O(\log^2 m)$ of type 2 at every step. Overall, the algorithm makes queries of weight $O(\log^3 m)$. We now turn to the correctness analysis. On a very high level, the intuition is that if there were no errors, the segment in which $i$ belongs (in the original list, before errors were introduced) is the one which will be chosen as the segment with which we proceed to the next recursion level (including one additional segment, unless $i$ is "close to the middle" , throwing out both adjacent segments). However, segments with errors, or incorrect estimations of $(s, b)$ may affect keys so that the "correct" segment is missed altogether, or too many segments are selected, and we decide to abort.

Several types of errors may occur.

1. In the worst case, all corrupted entries are not correctly recovered.
2. In the worst case, all corrupted entries duplicate some uncorrupted entries, and the incorrect payload is recovered.

As in the random case, errors that may occur as a result of terminating with a wrong $(i, s)$ may only occur for at most a $2\delta m$ fraction of the original entries (key) $i$ (see above). For all other values of the key $i$, this failure mode is impossible, and we have $s = 1 - b$ for all sampled elements in all intervals, but the one originally containing $i$. To obtain a bound on the fraction of other keys on which the algorithm may err with non-negligible probability, we may from now on assume that this is the case.

Let us refer to $I_j$'s with more then 0.3 fraction of corrupted entries as *bad*, and *good* otherwise. Fix some level $l$,and a key $i$ from basic interval $I_k$ in the original list. Consider basic intervals $I_j$ with $k \neq j$. There several possibilities as to votes of $I_j, I_k$ on input $i$:

**Observation 9** *1. Assume no entries of the form $(i, s)$ have been found in $I_j$. $I_j$ has at most (say) $0.3$ corrupted entries. In this case, with overwhelming probability, the interval votes against the "wrong" side for i, but not against the right side - denote $(w = 1, r = 0)$.*

*2. Assume the basic interval $I_k$ has a $\leq 0.3$ fraction of corrupted values. Then with overwhelming probability it votes against at least one of its sides.*

*Proof.* As in the warmup case, the proof is by calculating expected values of $(s, b)$ conditioned on $(i, s)$ keys not occurring in the interval in question. For item 1,2, assume wlog. that $j < k$. In item 1, at least a 0.7 fraction of elements in the interval $I_j$ are smaller then $i$ (for $j < k$), the expected fraction $b$ is $\leq 0.3$. Likewise, at least a 0.7 fraction of elements to be sampled (under the condition) are smaller then $i$. Thus, getting $r = 1$ or $w = 0$ would require $b \geq 0.31$ or $s \leq 0.3$ respectively. Applying Chernoff and union bound on the two events, $(r, w) \neq (0, 1)$ has probability $\leq exp(-\Theta(\log^2(m))$ of occuring. Item 2 is trivial (a taut, note that $(w = 0, r = 0)$ can occur with probability $> neg(l)$ only if $I_j$ has at least a (say) 0.3 fraction of its entries of the form $(i, s)$. For item 2, under our condition, trivially at least 0.5 of the sampled points vote against one of the sides (not all points are of the form $(i, s)$).

Let us assume wlog. that only the outcomes not explicitly listed in Observation 9 as (possibly) occurring with probability $neg(m)$ can in fact occur. In particular, if $I_j$ has a fraction $\geq 0.3$ of corrupted entries, we assume all outcomes are possible. We refer to such executions as likely executions. This is wlog., as taking union bound over the individual outcomes occurring with negligible probability ($poly(m)$ overall) results in an event with negligible probability. This is by taking union bound over the $\leq m^{2.71}$ nodes accessible on any input key $i$.

We are now ready to bound the set of keys that are incorrectly recovered in likely executions.

Items 1,2 together yield at most a $2\delta$ fraction of elements with wrong replies (with non-negligible probability). It remains to analyze how many uncorrupted entries $y$ with contents $(i, s)$ are not found because of aborting in step 2.1, or excluding the correct interval on a level $l$ of the recursion from the node $I$ selected for the next level, in some likely execution on input $i$. In this case, we say that uncorrupted entry $y$ is *injured* on level $l$ (note that the contents $(i, s)$ may not longer be unique in $I_k$ after corruptions, by injuring $(i, s)$ we mean its original index $y$ in the list is not properly detected). For brevity, in the sequel we refer to uncorrupted entries $y$ by their key $i$, in particular we refer to injuring $y$ with contents $(i, s)$ as injuring $i$ in $I_k$.

*A word of intuition.* Before delving into technical details, to bound the number of injured entries, let us consider a very high-level overview. Fix some list $L$ and set of corruptions. One key component of the analysis, is bounding the number of entries injured on a given level $l$ by some $c \cdot b_l$, where $c$ is some constant, and $b_l$ is the fraction of bad basic intervals on that level, which in turn is bounded by $\delta/0.3$ (of course, it makes sense to consider injured entries $i$ for inputs $i$). To make the bound well defined, we make sure it is independent of the algorithm's random choices. As the number of levels is $\Theta(\log |L|)$), naively summing over all levels results in $\Theta(\log n \cdot n) > n$, which is not a meaningful bound. The second key observation is that we should count the number of injured entries that are injured on each level for the *first time.* To bound these, we refine the first bound, and prove that newly injured entries on level $l$ are bounded by some $cb'_l$, where $b'_l$ is a subset of bad intervals on its level. Here we crucially use the fact that some intervals do not affect an interval $I_k$ because they are not in $I_k$'s node on that level, so $I_k$ does not "see" them.

**Observation 10** *(possbily) Overestimating the set of injured entries $i$ (over all likely executions with various input keys $i$) on every given level only increases the bound on the number of injured entries overall (over all likely executions).*

We often use this observation in subsequent analysis. For starters, we (pessimistically) assume that all entries that fall in bad basic intervals will not be recovered correctly. The following claim provides a simple necessary condition for $i$ in a good level-$l$ basic interval $I_k$ to be injured on level $l$.

*Claim.* Consider a good basic interval $I_k$ on some level $l$, and some likely execution on key $i$. An uncorrupted entry $i \in I_k$ in a basic level-$l$ interval $I_k$ is injured on level $l$ only if there exists an interval $I_j, k \neq j$ at distance at least 2 from $I_k$, such that the number of votes against $(I_k, i)$ is at least as large as the number of votes against $(I_j, i)$, or if $I_k$ is adjacent to a bad interval $I_j$.[7]

*Proof.* Assume the precondition (following "only if") of the claim does not hold. We show that $i$ is then not injured on level $l$. Clearly, $i$ can not be injured by $I_j$ at distance 2 or more from $I_k$ has less votes against $(I_j, i)$ then against $(I_k, i)$. It remains to check that both intervals $I_{k-1}, I_k$ do not receive the same number of votes (or less) for $i$ as $I_k$. As $I_{k-1}, I_{k+1}$ are both good, by Observation 9, none votes against $I_k$ with overwhelming probability. The difference in votes between $I_k$ and $I_{k-1}$ $(I_{k+1})$ (as all intervals besides $I_{k-1}, I_k$ vote in the same way for both) is $Vote(I_k, I_{k-1}, i) - Vote(I_{k-1}, I_k, i)$ (naturally, we let $Vote(I_k, I_j, i)$ be 1 if $I_k$ votes against $I_j$ on $i$, and 0 otherwise). The algorithm could still terminate in step if the difference above is 0 for both $I_{k-1}$ and $I_k$. However, by Observation 9 $I_k$ votes against at least one of its adjacent intervals in $i$ (in likely executions), so this also can not occur.

Let $i$ be an uncorrupted entry. For $I_k, I_j, i$ as in Claim 2.1 ($I_j$ is as in part 1 or 2 of the precondition), we say that $I_j$ potentially injures $(i, I_k)$ on level $l$ (and $i$ is potentially injured on level $l$). For all uncorrupted entries $i$ in bad $I_k$'s, we also define $(i, I_k)$ to be *potentially injured* (by "whom" is not specified in this case).

As $i$ being potentially injured on level $l$ is a necessary condition for being injured, by Observation 10, it is sufficient to bound the set of uncorrupted entries $i$ potentially injured on some level. The following "monotonicity" condition allows us to eliminate the dependence of our bound on the set of keys that can be injured on a given level on the random choices of the algorithm (in likely executions).

*Monotonicity.* Denote by $B_l$ the set of bad basic intervals, and by $G_l$ the set of good basic intervals $I_k$ on level $l$. Also, let $G'_l$ the set of uncorrupted entries in $G_l$.

*Claim.* Consider the set of likely executions on keys $i$ originally present in $L$. The set of keys $i \in G'_l$ that are injured on level $l$ (for input $i$) would be a superset if we modified the algorithm's decisions so that: 1. For all keys in $G'_l$, their interval $I_k$ votes against none of the directions. 2. All intervals $I_j$ in $B_l$ vote $(w = 0, r = 1)$ for all uncorrupted keys $i \in G_l$.Note that assumption 1 is and assumption 2 may be impossible in likely executions on $i$. However, we make these assumptions only for the sake of analyzing the set of keys injured on level $l$, and do not assume algorithm's execution actually changes. Under this assumption $(i, I_k)$, a good key $i$ can be injured on level $l$ if.

[7] Note that an error will not necessarily occur since $I_j$ and $I_k$ may not belong to the node that includes $i$ on that level (which includes $\Delta$ or $2\Delta$ of the basic intervals on that level), and thus may not be seen by the algorithm. However, this is consistent with our "overestimation principle".

– $I_k$ is a bad interval, or
– $(i, I_k)$ is potentially injured by some interval $I_j$. This happens iff. $[I_j, I_k)$ has at least as many bad intervals as good intervals.

*Proof.* Consider a likely execution $e_1$ on input $i$, and let $e_2$ denote an execution resulting from making modifications 1,2 as in the claim. Consider an uncorrupted entry $i$ in a good level $l$ interval $I_k$. As $I_k$ is good, by Claim 2.1 $(i, I_k)$ can be injured on level $l$, only if it is potentially injured by some $I_j$. Thus, assume there exists an interval $I_j$ (for notational simplicity, assume $j < k$, this is wlog.) that potentially injures $(I_k, i)$ in $e_1$. If the only such $I_j$ adjacent to $I_k$, let us pick this $I_j$ for further analysis. Such $I_j$ must be bad by definition of $I_j$ potentially injuring $(I_k, i)$. Otherwise, let us pick some bad $I_j$ with $k - j \geq 2$. To see there exists such a bad $I_j$, pick some $I_{j'}$ with $k - j \geq 2$ injuring $I_k$. If it's bad, set $I_j = I_{j'}$. If $I_{j'}$ is good, and all intervals in $(I_{j'}, I_k)$ are good, $I_k$ would get at least 2 votes less then $I_j$ - a contradiction. Thus, picking the closest to $I_{j'}$ bad interval in $(I_j, I_k)$, has the same number of votes as $I_j$, and thus also injures $(I_k, i)$.

Let $g, b$ denote the number of good and bad intervals in $(I_j, I_k)$.

For $I_j$ to potentially injure $(I_k, i)$, we must have $d = Votes(I_k, i) - Votes(I_j, i) \geq 0$. In $e_2$ $d = b + 1 - g - Vote(I_k, I_j, i) = b + 1 - g$, as bad intervals in $[I_j, I_k)$ contribute 1 to the difference, good intervals contribute $-1$ and $Vote(I_k, I_j, i) = 0$, intervals outside of $[I_j, I_k]$ always contribute 0.

Now, in $e_1$ not all bad intervals in $(I_j, I_k)$ may contribute either $1, 0, -1$, good intervals still contribute $-1$ and $Vote(I_k, I_j, i)$ contributes 0 or $-1$. Overall, the difference in $e_1$ could only decrease relatively to $d$, so $I_j$ must injure $(I_k, i)$ in $e_2$ as well.

Let $M_l$ denote the set of $I_k$'s containing uncorrupted entries $i$ potentially injured on level $l$. We bound the $G'_l$ by bounding the fraction of entries in $M_l$, and assuming all entries in $M_l$ are in $G'_l$, and have been injured. A set $B'_l \subseteq B_l$ potentially injures an interval $I_k$ in $M_l$, if some $I_j \in B'_l$ potentially injures $(I_k, i)$ (for an uncorrupted $i$), or if $I_k$ is itself bad. As a natural extension, we say a set $B'_l \subseteq B_l$ potentially injures a set $M'_l \subseteq M_l$ if it potentially injures all intervals in $M'_l$. Our main technical observation, bounding the set $M_l$, is as follows.

**Lemma 11.** *The set $B_l$ of bad intervals potentially injures a set $M_l$ of size at most $5|B_l|$ on that level. In particular, the set $N'_{l+1}$ of level-$l+1$ nodes intersecting $M_l$ contains at most $15|B_l|$ basic (level-$l$) intervals. We refer to these as "potentially injured" level-$l+1$* nodes *(note that the injury event occurs on level $l$).*

See Section B for a proof.

The definition of potential injury, does not take nodes into account (that is, it acts as if we allowed all basic intervals to "see" all other basic intervals on that level in step 2.2 of the algorithm). If we take nodes into account, the bound on $|M'_l|$ induced by the union of a set $N'_l$ of level-$l$ nodes depends only on bad basic intervals in $N'_l$. More precisely, we have the following extension of the above lemma.

**Observation 12** *Consider a set $N'_l$ of level-$l$ nodes. Let us line up the basic level-$l$ intervals contained in $N$ in their order (closing gaps when needed), obtaining a sequence $S$. Then the set $M_l \cap N$ (basic intervals potentially injured by $B_l$) is of size at most $5\,|B'_l|$, and at most $15\,|B'_l|$ basic intervals in injured level-$l+1$ nodes for $B'_l = B_l \cap N$.*

*Proof.* As in Lemma 11, we bound the number of potentially injured intervals in $N_l$ as induced by the sufficient condition in Claim 2.1. Thus, every bad interval in $N'_l$ remains bad in $S$, and every good interval is injured if it "sees" a bad $I_j$ such that $[I_j, I_k)$ has at least as many bad intervals as good ones. The bound now follows as in the proof of Lemma 11 (note that we still do not account for some of the boundaries induced by nodes, but rather view $S$ as a single node).

For every level $l$, let us bound the number of basic intervals in $M_l$ containing entries that are injured on level $l$ *for the first time.* Then, to obtain the overall bound on the number of injured keys, we just sum over all the bounds for first injured keys (counting “first injured” keys, prevents us from counting again keys that were already counted as injured at an earlier point along the path). Let $B'_l$ denote the set of basic intervals not contained (as sets of entries) in intervals in $B_j$ for $j < l$. Denote by $N'_l$ the set of level-$l$ nodes not contained in any node potentially injured on a previous level (recall nodes injured on level $l-1$ are level-$l$ nodes, and level $l$ is not injured by definition). The set $N'_l$ does not miss newly injured entries, as *all* entries in nodes outside of $N_l$ have been already counted as (potentially) injured on a previous level.

It follows from Observation 12, that the set of basic intervals in $N'_l$ injured on level $l$ is of size at most $5|B'_l|$. In more detail, in Observation 2 we have $B'_l \subseteq B_l \cap N'_l$. This is true since nodes in $N'_l$ simply do not intersect bad basic intervals from previous levels (recall all bad basic intervals on a level are defined as potentially injured). Now, applying Lemma 11, we obtain that $N'_l$ contains at most $15|B'_l|$ basic intervals in potentially injured level-$l+1$ nodes. Let us denote the fraction of $B'_l$ out of all level-$l$ basic intervals by $t_l$. Thus, by Lemma 11, at most $\sum_j 15t_j \cdot m$ uncorrupted entries were potentially injured on level $l$. On the other hand, as the $B'_l$'s are all disjoint sets of elements, and since every bad basic interval has at least a 0.3 fraction of incorrect elements, we have $\sum_j 0.3t_j m \leq \delta m$. Combining the two inequalities, we conclude that at most $50\delta l$ uncorrupted entries are potentially injured overall. Combining with possible losses from 1,2, we obtain that at most $52\delta \cdot l$ elements result in an incorrect output of our algorithm.

## 2.2 Transforming standard LDC into LDC for edit distance.

Let $C_1 : \mathbb{F}_p^n \to \mathbb{F}_p^m$ denote a $(\delta_1, n, \epsilon)$-LDC code for Hamming distance, and $C_2 : \mathbb{F}_p^m \to \mathbb{F}_p^h$ a corresponding SZ code. Let $Q_1$ denote a suitable decoding algorithm for $C_1$. To transform $C_1$ into an LDC for edit distance, we compose it with $C_2$ (as done in [7] for a standard ECC $C_1$, and $p = 2$)

Recall that $C_2 : \mathbb{F}_p^m \to \mathbb{F}_p^h$ on input $w_1$, divides it into $T = \lceil m/\log m \rceil$ blocks $w_{1,1}, \ldots, w_{1,T}$. Then there are $t = \lceil (m/T + \log m)/logp \rceil \leq \lceil 2 \log m \rceil$ symbols in a block. We let $I = I_{t,\delta}$, where $I_{t,\delta}$ is as guaranteed by Lemma 5 (we are not concerned that $t$ needs to be "large enough", wlog. we may consider only codes starting with large enough $n$ $(m)$). It outputs $w_2 = w_{2,1} \circ \ldots \circ w_{2,T}$, where $w_{2,i} = I(i \circ w_{1,i})$. Denote the output length of $I$ by $m_t (= O(\log m))$.

We claim that the code $C_2(C_1)$ is a $(\delta_1', q \cdot polylog(m), \epsilon + neg(m))$-LDC for edit distance, where $\delta_1' = c \cdot \delta_1$, for a some global constant $c$ that depends only on parameters of $I_t$. The decoder $Q_2$ for $C_2$ runs a straightforward simulation of $Q_1$, where the crux of technical difficulty is providing answers to $Q_1$'s oracle queries, using its own oracle. In slightly more detail.

**Construction 13** $Q_2^{w_2'}(i)$ :

1. *Run $Q_1^{\cdot}(i)$. When a query $k$ into $w_1'$ is made:*
   - *Calculate the index $i \in [T]$ of the block in which the $k$'th position in $w_2$ is located $(\lfloor k/T \rfloor + 1)$.*
   - *Execute $FindBlock^{w_2'}(i)$ using some $polylog(m)$ queries into $w_2'$. Let $v = (i' \circ w_{1,i}')$ denote its reply. Read $w_1[k]$ from $w_{1,i}'$ and forward it to $Q_1$ as the reply to its query.*
   - *Output the value that $Q_1$ outputs.*

That is, $FindBlock^{w_2'}(i)$ locates the $i$'th block of $w_2$, decodes it via $I$ as $some(i \circ s)$, where $s$ is (hopefully) the $i$'th block of $w_1$. As mentioned before, the main difficulty is in searching for the block in $w_2'$. We do not know where it is located in $w_2'$ (but only upto a distance of $\delta|w_2|$ symbols, or so).

The high level idea is to somehow interpret $w_2'$ as a list of sorted elements with corruptions (the sorting is by the index $i$ written in each block), and run the algorithm for sorted lists with corruptions on it. For this purpose, we should be able to make "backwards" translations of the list searching algorithm's queries into reading portions of $w_2'$. In particular, we show that for small enough $\delta_1$, blocks from $w_2$ $(= w_2'$ before modifications) form $1 - O(\delta_1)$ of the induced list's elements, and appear in their correct order. It turns out that a slightly generalized abstraction of a *weighted* sorted list with certain restrictions on list weights emerges. Fortunately we will be able to adapt our searching algorithm for unweighted lists to this more general setting. See Section A for precise details. We obtain our main theorem.

**Theorem 14.** *Consider an $(\delta(n), q(n), \epsilon(n))$-LDC $L_H : \mathbb{F}_p^n \to \mathbb{F}_p^m$ for Hamming distance, where $\delta(m)$ is some constant[8]. Consider a code $C_2(C_1)$ as defined at the beginning of this section (given $C_1$). This code is a $(c\delta, q \cdot polylog(m,p), \epsilon + neg(m))$-LDC $L_E : \mathbb{F}_p^n \to \mathbb{F}_p^m$ for edit distance, where $c$ is a (quite small) global constant. The code rate degrades by a constant. Encoding efficiency only degrades by a $poly(n)$ factor.*

[8] As opposed to our construction for searching on sorted lists with corruptions, here we require that $\delta$ is not subconstant. Otherwise, the degradation in $\delta$ could be superconstsant in the actual error fraction $\delta$.

The proof of this theorem follows by the construction outlined in this section. Namely, the decoder for $C_2(C_1)$ is outlined in Construction 13. $FindBlock^{w'_2}(i)$ runs $WS^{L'_2}(i,T)$, and implements its queries using oracle access to its own oracle $w'_2$, as described in Section A. The various efficiency properties of the resulting code follow from Lemma 5. The constant $c$ is a product of two constants resulting from the $FindBlock^{w'_2}$ algorithm. The core of this algorithm is a an algorithm for searching on weighted sorted lists with corruption. This constant is comparable to the (1/) 52 loss we get in the algorithm for searching unweighted lists 8. The other factor stems from the need to correct errors in $I$, and to make sure that the word $w'_2$ indeed induces a list with a small $(O(\delta))$ fraction of corruptions. This part is comparable to the fraction of errors tolerated by $I$.

## A Details for Section 2.2

**What list is induced by a codeword $w'_2$, and how do we query it?** First, let us better define what are the entries of the list $L'_2$ induced by $w'_2$. In $w_2$ the picture is clear, $w_{2,i}$ corresponds to the block $I(i \circ w_{1,i})$. After corruptions (at the level of individual symbols), some care should be taken when determining what the list entries are, and where each entry starts and ends. Recall $I$ has distance $\delta m_t$, and let $\delta_2 = \epsilon\delta$, for a (global) constant $\epsilon$ to be determined later.

1. We identify all subsequences of $w'_2$ that are $\delta$-close to a codeword of $I$ - denote as valid sequences. In particular, every $w_{2,i}$ that underwent $\delta_2$ or less insertion/deletion operations (when viewed separately from all other changes in the list) induces at least one such sequence. Fix a valid subsequence $v'_j = [l_j, r_j]$ of $w'_2$. Then by the no overlapping property of $I$ (Lemma 5), and the fact that $\epsilon \ll 1.5$ only the range $[l_j - (2+\epsilon)\delta m_t, l_j + (2+\epsilon)\delta m_t)]$ may contain other valid subsequences, and they must decode to the same codeword as $v'_j$. Otherwise, the closest valid subsequence starts at least $(1-(2-\epsilon)\delta)m_t$ symbols to the right or to the left of $l_j$.Also, by definition of edit distance, valid subsequences are of length $(1 \pm \delta_2)m_t$. We refer to such extended valid subsequences as "valid extensions". Thus, we can think of the list as a sequence of "valid extensions" of length $(1+O(\delta))m_t$ for a small hidden constant $(\leq 3)$. For now on, we do not explicitly specify this constant, or constants derived from it, but rather use the notation $O(\delta)$. It will the case that the derived constant terms $O(\delta)$ occurring throughout the following discussion will not be "too large". Valid extensions may overlap by some $O(\delta)m_t$ symbols that decode to the same $(i \circ s)$ entry each. Every such extension is defined by picking some valid subsequence $v'_j = [l_j, r_j]$, finding all valid subsequences in the range $[l_j - O(\delta)m_t, l_j + O(\delta)m_t)]$, and defining the bounds of the corresponding extension as the shortest interval that includes all these. Note that valid extensions are well defined (and independent of the choices of concrete $v'_j$).
2. The rest of the list consists of data stretches which are $\delta$-far from codewords (possibly long stretches, of length not necessarily close to multiples of $m_t$). Reading an entry in such an "empty stretch" will be interpreted as $(0, \overline{0})$.

Intuitively, the list induced by $w'_2$ is a *weighted* list, the length of which is the sum of weights of its individual elements. The weights are specified in multiples of $m_t$, and equal the length of an entry's "representation" in the list. The weights of entries corresponding to valid extensions are close to 1. The "empty stretches" correspond to sequences of entries of length 1 and one of some fractional weight at the end of a sequence. The valid extensions consist of "mildly corrupted" $w_{2,i}$'s, appearing in their original order (as in $w_2$), and new "fake" entries that are valid extensions created by indel operations corresponding to arbitrary values (possibly breaking the order), of weight close to 1 each.

To make the intuition of the induced list precise, we should specify how we implement reading a random entry from the induced list, which is the main type of query used by the sampling algorithm. We prove that indeed the output behaves (almost) exactly as if we read a random weighted entry from a list as described above.

The main type of query in the list searching algorithm is picking a list entry at random from a certain "long" interval - currently think of the interval as being the entire list).

*Sampling a random entry in the list induced by $w'_2$.* We are given an oracle to a (corrupted) codeword $w'_2$, of length $h$. To sample a random element of the induced weighted list $L'_2$, pick a random $l \in [h]$, and read an interval $D$ of $2m_t$ symbols starting at $l$. Search for a subsequence in $D$ that is $\delta_2$-close to a codeword of $I$. This can be accomplished by decoding and checking distance between the (re-encoded) result and the tested subseqeunce (note that decoding can be successful for much worse distances, and $I$ in fact tolerates edit distance of upto $\delta$).

If such subsequences exist, decode one, $v'_j$, with the lowest left end point and return the corresponding $(i \circ s)$. If none was found, output $(0, \overline{0})$. Calling this procedure costs reading $O(m_t)$ positions of $w'_2$.

Now we can precisely analyze what the induced list looks like, and what the distribution query's output is. For a valid extension $s_j$ "encoding" some $(i, s)$, there exists a (contiguous) interval $d_j$ of length $(1 \pm O(\delta))m_t$, roughly preceding $s_j$ ($d_j$ may intersect $s_j$ by some $O(\delta m_t)$), such that sampling $l$ in $d_j$ results in $v'_j$ in $s_j$, thus decoding to $(i \circ s)$ corresponding to $s_j$. Also, $d_j$'s corresponding to different valid extensions are disjoint (roughly, because the sampling procedure picks the first valid $v'_j$, and $O(\delta)$ above is much smaller then 1). A full proof of the above facts is easy, and is left to the full version. By definition of the sampling procedure, and the observation on the disjointness of the $d_j$'s, for $l$'s outside these $d_j$'s, the output is $(0, \overline{0})$. More precisely, elements in valid extensions and empty stretches are represented by their preimages in the sampling procedure (which are also contiguous stretches) as follows.

1. Entries sampled by the various $d_j$'s corresponding to valid extensions $s_j$ appear in the order of their appearance in $w'_2$.
2. Empty stretches in $w'_2$ are sampled iff. we hit a $j$ outside of any $d_j$. Empty stretches' weight thus changes by some $\pm O(\delta m_t)$, and they are represented

by stretches $e_j$ between the different $d_j$'s. Some sufficiently short stretches' weight may go down to 0, in which case we do not count them as elements of $L'_2$ at all. We interpret every such stretch as a sequence of weight-1 $(0, \overline{0})$ entries, where the last one has some fractional weight (completing the weight to $|e_j|$).

To summarize, the induced weighted sorted list $L'_2$ is as follows. We have oracle access to a sorted list with entries $(i, s)$ labeled $d_j$ or $e_{j,t}$ of various lengths, specified in multiples of $m_t$ (most are close to 1, and some can be smaller fractions), according to the length of the corresponding interval $d_j$ or $e_{j,t}$ in $w'_2$. Sampling an entry at random samples every entry with probability proportional to its weight.

We make the following simple observations about the list's size and weights.

– Simple calculations show that the number of list elements $d_j$ or $e_{j,t}$ is at most $(1 + O(\delta))T$. This roughly follows from the fact that $d_j$'s have length close to 1 (in $w'_2$), and that fractional $e_{i,j}$ elements appear at most once per empty stretch, the number of which is at most the number of $d_j$'s + 1, thus at most $(1 + O(\delta)T)$.
– For every interval or length $r \geq 2$ in $L'_2$, the total weight of elements in that interval is in the range $[r/2, 2r]$. More precisely, it is $(1 \pm O(\delta))r$. The worst weight density is obtained when short $e_{i,t}$'s of weight close to 0 are interleaved between other intervals ('regular' $e_{i,j}$'s are $d_j$'s, whose length is close to 1).

In all the above arguments, we relied on $\epsilon, \delta$ being small. To make everything go through, $\epsilon$ can be set to about 0.1.

Let us now connect the bound $\delta_1$ on $dist(w_2, w'_2)$ and properties of the resulting list. Assume $w_2$ sustained $\delta_1|w_2|$ indel operations. Then we have:

– At least $(1 - \delta_1/\delta_2)T$ of the blocks in $w_2$ sustained at most $\delta_2 m_t$ indel operations. These belong to valid extensions in $w'_2$ (inducing some $d_j$), enter the list $L'_2$ in their proper order, and return the correct $w_{2,i}$ when sampled. Their total weight is at least $(1 - O(\delta))(1 - \delta_1/\delta_2)T$.
– There are at most $\delta_1|w_2|$ symbols outside of valid extensions corresponding to $w'_{2,i}$'s as above. Thus, the total weight of corrupted entries (either in valid extensions or in empty stretches) in $L'_2$ is thus $\leq (1 + O(\delta))\delta_1 T \leq 2\delta_1 T$. Thus, for small enough $\delta_1$ the weight of original entries is at least $W_g = 1 - \delta_1/\delta_2 - O(\delta)$, and the total weight of $L'_2$ is at most $W_g + 2\delta_1$.

From the above calculations, we conclude that taking $\delta$ to be sufficiently small, and $\delta_1 \leq \epsilon\delta^2$, the relative weight of uncorrupted entries in $L'_2$ will be large enough so that the weight of uncorrupted entries is $1 - O(\delta_1/\epsilon\delta)$.

The list searching algorithm actually needs to make random samples in specified intervals. To specify a fractional interval $[\alpha, \beta]$ in the list, we will simply use the interval $[\alpha|w'_2|, \beta|w'_2|]$ in $w'_2$. The above procedure trivially modifies to randomly sampling an interval corresponding to some $w'_2[s, e]$ by just picking a random $l$ in that interval, and doing everything else as before. There are certain artifacts of possibly losing/reducing weight of some $O(1)$ entries at the beginning of an interval or the entire list. Jumping ahead, this can be neglected, since over all intervals reachable by the list searching algorithm, only $O(T/\log^2 T)$ elements fall into these areas, while we consider the range of constant error fractions.

A second type of query we need to handle is checking whether an interval $[s, e]$ in $w'_2$ has at most $y$ elements in it. Returning all elements if it does, To implement this query, check if the interval is of length $(2 + O(\delta))(y + 1)m_t$ or smaller (for a suitable constant). If so, recover the $d_j, e_{j,l}$'s in $D$, and check whether their number is $y + 1$ or higher. If so, return the recovered elements, otherwise return $\perp$. The query requires at most $3(y + 1)m_t$ queries to the LDC oracle $w'_2$.

**Adapting searching in corrupted sorted lists to the weighted setting.** We are ready to fully specify the abstraction of searching on weighted sorted lists with corruptions needed by the $FindBlock^{w'_2}$ procedure in our LDC query algorithm. We adapt the protocol for unweighted lists from Section 2.1 to the case at hand of weighted lists. The algorithm will put some preconditions on the weights in the list, that are satisfied by observations made about the induced list $L'_2$ in the previous section. Let us define the setting more precisely.

- The protocol is specified by an algorithm $WS^L_\delta(i)$. $L$ is a weighted sorted list (weights are strictly positive), where some of the elements have been corrupted (modified arbitrarily), where the total weight fraction of corrupted elements is some constant $\delta$ fraction of the total weight of elements in the list, $W = weight(L)$. The weights have the property that all sequences of $r \geq 3$ elements in the list have total weight in the range $[r/2, 2r]$ (in particular, $m \in [W/2, 2W]$).
- Parameters: Let $m'$ denote an approximation on the number of elements in $L$ (disregarding weights) upto a factor of 2. $m'$ is available to $WS$, while the exact $m = |L|$ is not). An estimation $\delta_0$ of $\delta$, accurate up to a factor of 2.
- Input: A key $i$ to search.
- Oracle queries: There are two types of possible queries to the list.
  1. $(v_0, v_1)$, where $(v_0, v_1)$ are fractional values. $(v_0, v_1)$ specify fractional weighted endpoints of the interval (may be closed or open on either side). That is, $v_0$ $(v_1)$ points to the $j$'th list entry where $j$ is the smallest index, such that $\sum_{k=1}^{j} weight(L_k)/weight(L) \leq v_0$. A random element is returned, by a probability distribution assigning each entry a probability measure proportional to its weight. Query weight is 1.
  2. $(v_0, v_1, t)$ : Ask for the sequence of all entries in an interval $(v_0, v_1)$. If the interval is of size $t$ or smaller, the sequence of all entries will be returned. Otherwise, an error is returned. Query weight is $m$ (regardless of the query's outcome).
- Output: Given a key $i$, such that $(i, s_i)$ was present in the original list (before corruptions occurred), the correct output for it is $s_i$.

– Goal: Maximize the worst case weighted fraction of keys $i$ originally present in the list for which the reply is correct with probability $1 - neg(m)$. In particular, this should hold for a $1 - O(\delta)$ fraction of uncorrupted elements.
– Query Complexity: Total weight of queries is $polylog(m)$ - we are not trying to optimize on the concrete complexity.

**Construction 15** *Initialize the searched interval to $I = (0, 1)$, $\Delta = 3$, $T = \Delta$, $r = \log^2 m'$.*

*Repeat:*

1. *Make a type 2 query with $(I, y = r)$. If it returns a sequence of points, and one of them is of the form $(i, s)$, return $s$ corresponding to the first such $i$. Otherwise, return $\perp$.*
2. *Otherwise, divide $I$ into $T$ intervals $I_1, \ldots, I_T$ of equal size. Sample $r$ random locations in each of the intervals, resulting in $o_{i,1}, \ldots, o_{i,r}$ for the i'th interval.*
   - *(a) If some sample is of the form $(i, s)$, return $s$ corresponding to the first such $i$.*
   - *(b) Otherwise, for each interval, count the fractional weights $s, b = 1 - s$ of smaller and larger then $i$ sampled elements respectively (relatively to the total weigh of elements read). We say that interval $j$ votes against interval $k, k > j$ for $i$, (or against $(I_k, i)$) if $b > 0.31$ (for $k < j$, if $s > 0.1$).Note that if $I_j$ votes against $(I_k, i)$, then it votes against all $(I_h, i)$ for $h$ on the same side of $I_j$ as $l_k$. We then say that $I_j$ votes against its left (right) side on $i$. For every interval $j$, we count the number of votes against $(I_j, i)$ over all other intervals.*
     - *i. If there is exactly one interval with a minimum number of votes, set $I$ to be that interval.*
     - *ii. If there are two such adjacent intervals $I_j, I_{j+1}$ set $I$ as their union. Fix $T = 2\Delta$.*
     - *iii. Otherwise, output $\perp$ and terminate.*

Analysis sketch. The analysis is a quite straightforward adaptation of the analysis of the algorithm for the unweighted case in Section 2.1, with the following minor changes.

– In the complexity analysis, we need to make sure that the number of elements in the list still drops by some constant factor per recursive step (and thus, there are $O(\log m)$ levels, where the leafs have intervals of length $\log^2 m' = O(\log^2 m)$), as before. This follows since the total weight of elements in an interval decreases by a factor of at least $2/\Delta$ in every recursive step, and the number of elements in an interval is at most 2 times its weight. As this holds for the entire list as well, we start with $W \leq 2m$, and the claim follows.
– Everywhere we consider fractions, we instead consider weighted fractions. Essentially, we adapt the definitions of $(b, s)$ to be weighted, and the definitions of bad and good intervals are the same relatively to the values of

$(b, s)$, and the analysis easily goes through. The analysis of votes etc. remain the same (just count votes). However, we are counting bad and "potentially injured" intervals, without accounting for weights. Thus, the final analysis, bounding "newly" injured entries, where basic intervals have equal weights also remains the same. One notable place where this requires some care is when applying Chernoff bounds to estimating $(b, s)$ in at an interval's $I_j$'s votes against its left or right side. For this purpose, we need to use the weighted variant of the Chernoff bound.

– Finally, as mentioned before, we are neglecting "edge effects" when reading entries of (long) intervals, where the first $O(1)$ elements in every interval are not accessible, or have reduced weights. These can be neglected as all of our intervals are of length at least $\log^2 m$, so these effects have overall sub-constant effect on the algorithm. The effects start being non-negligible for $\delta_1 = o(\log^2 m)$, but not for the range of constant $\delta$ that we are interested in. This is as opposed to the unweighted case, where the

**Theorem 16.** *Construction 7 is an algorithm for searching on weighted sorted lists (in a framework as defined above), tolerating a (small enough) constant $\delta_0$ fraction of corruptions. For at least a $(1 - O(\delta))m$ fraction of the original lists' elements, it recovers them correctly with probability $\geq 1 - neg(m)$. It makes $O(\log^3 m)$ queries to the list.*

## B Proof of Lemma 1, Section 2.1

Recall from Claim 2, that a sufficient condition on interval $I_k$ being in $M_l$ is as follows.

1. $I_k$ is a bad interval, or
2. There exists a bad interval $I_j$ (assume wlog. $j < k$) , so that $[I_j, I_k)$ has at least as many bad intervals $b$ as good intervals $g$.

Thus, it suffices bound the set of basic level-$l$ intervals satisfying the above condition. We prove the lemma by induction on the number $t$ of bad intervals on that level (for any number of basic intervals $n_l$ on that level). Base case, $t = 1$. Here only the bad interval itself and two adjacent intervals on both sides are potentially injured.

Step. Assume the lemma holds for all $n_l$ and $t' \leq t$. To prove for $t + 1$, consider the list $L'$ with $t$ bad intervals $I_1, \ldots, I_t$ (ordered from left to right) from which our list $L$ is obtained by transforming one interval, $I_0$ in $L'$ from good into bad. Assume for simplicity that it is located to the left of $I_1$ (this is wlog.). Our goal is to identify the set of intervals that were injured due to adding the new bad interval (and were not injured in $L'$). As making $I_0$ bad is the only change, the only ("meta"-)interval that can explain the injury of some good (basic, level-$l$) interval $I_k$ is $[I_0, I)$ (or $(I, I_0]$, if $I$ is located to the left of $I_0$). As to newly injured intervals to the right of $I_j$, there can be two options.

1. Assume $I_0$ is at distance at least 2 from $I_1$. Then it can not newly injure any intervals $I'$ located to the right of $I_1$ in $L'$. This is the case since by $I'$ not being injured in $L'$, it means that for all $I_j$'s located to the left of $I'$ $[I_j, I')$ satisfy $b < g$ (similarly all $(I', I_j)$ for $I_j$ to the right of $I'$, but these remained unchanged). The only new potential "injurer" for $I'$ is $[I_0, I')$. However, there the balance of $b, g$ is the same or worse as in $[I_1, I')$ ($b$ grows by 1, and $g$ grows by at most 1), so no new "injury" is inflicted on $I'$. As to intervals $I'$ in $(I_0, I_1)$, at most two intervals to the right of $I_0$ can become newly injured, by similar considerations of balance between $b, g$ in $(I_0, I')$.

2. Assume $I_0$ is right to the left of $I_1$. Then it improves over the balance of $b, g$ in $[I_1, I')$ for all $I'$ to the right of $I_1$ exactly by 1 (and it is the only change for all these $I'$'s). Let $I'$ be the leftmost newly injured such good interval (if any). Then it must be the case that $b - g$ in $[I_1, I')$ is $-1$ (and thus 0 in $[I_0, I')$). We show that no additional $I$'s to the right of $I'$ can be newly injured.

- In the interval $[I_h, I_{h+1}]$ that $I'$ is in, $b - g$ in $[I_0, I)$ for subsequent $I$'s decreases by 1 for every step to the right.
- For subsequent intervals, the situation is even wrose. Assume wlog. that $I$ is newly injured by $[I_0, I)$. This means that $[I_{h+1}, I)$ had a balance $b - g \leq -1$ (otherwise, it was already injured by $[I_{h+1}, I)$. Additionally, as $I$' was newly injured, $[I', I_{h+1})$ is of length at least 3 (good) intervals - otherwise $I$ was already injured by $(I', I_{h+1}]$ in $L'$. Overall, this implies that $b - g$ in $[I_0, I)$ is at most $-1 - 3 = -4$, which implies that $I$ is not injured by $[I_0, I)$ in $L$, and thus can not be newly injured as well - a contradiction.

Clearly, for $I$'s to the left of $I_0$, at most the two adjacent ones can be newly injured. Thus, turning $I_0$ into a bad interval, added at most 5 injured intervals (including $I_0$). Overall, we conclude that at most $5t$ intervals in $L$ are potentially injured if $t$ of the intervals are bad. $\square$

## References

1. Babai, L., Fortnow, L., Levin, L.A., Szegedy, M.: Checking computations in polylogarithmic time. In: Koutsougeras, C., Vitter, J.S. (eds.) STOC. pp. 21–31. ACM (1991)
2. Finocchi, I., Italiano, G.F.: Sorting and searching in the presence of memory faults (without redundancy). In: Babai, L. (ed.) STOC. pp. 101–110. ACM (2004)
3. Goldreich, O.: Short locally testable codes and proofs (survey). Electronic Colloquium on Computational Complexity (ECCC) (014) (2005), http://dblp.uni-trier.de/db/journals/eccc/eccc12.html#TR05-014
4. Hemenway, B., Ostrovsky, R., Strauss, M.J., Wootters, M.: Public key locally decodable codes with short keys. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) APPROX-RANDOM. Lecture Notes in Computer Science, vol. 6845, pp. 605–615. Springer (2011)
5. Katz, J., Trevisan, L.: On the efficiency of local decoding procedures for error-correcting codes. In: Yao, F.F., Luks, E.M. (eds.) STOC. pp. 80–86. ACM (2000)

6. Rivest, R.L., Meyer, A.R., Kleitman, D.J., Winklmann, K., Spencer, J.: Coping with errors in binary search procedures (preliminary report). pp. 227–232. ACM (1978), http://dblp.uni-trier.de/db/conf/stoc/stoc78.html#RivestMKWS78
7. Schulman, L.J., Zuckerman, D.: Asymptotically good codes correcting insertions, deletions, and transpositions. In: Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 669–674. SODA '97, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1997), http://dl.acm.org/citation.cfm?id=314161.314412
8. Yekhanin, S.: Locally Decodable Codes and Private Information Retrieval Schemes. Ph.D. thesis, Cambridge, MA, USA (2007), aAI0819886
9. Yekhanin, S.: Locally decodable codes. In: Kulikov, A.S., Vereshchagin, N.K. (eds.) CSR. Lecture Notes in Computer Science, vol. 6651, pp. 289–290. Springer (2011)