Locally Updatable and Locally Decodable Codes

Nishanth Chandran[*] Bhavana Kanukurthi[†] Rafail Ostrovsky[‡]

Abstract

We introduce the notion of locally updatable and locally decodable codes (LULDCs). In addition to having low decode locality, such codes allow us to update a codeword (of a message) to a codeword of a different message, by rewriting just a few symbols. While, intuitively, updatability and error-correction seem to be contrasting goals, we show that for a suitable, yet meaningful, metric (which we call the Prefix Hamming metric), one can construct such codes. Informally, the Prefix Hamming metric allows the adversary to arbitrarily corrupt bits of the codeword subject to one constraint – he does not corrupt more than a δ fraction (for some constant δ) of the t "most-recently changed" bits of the codeword (for all $1 \leq t \leq n$, where n is the length of the codeword).

Our results are as follows. First, we construct binary LULDCs for messages in $\{0,1\}^k$ with constant rate, update locality of $\mathcal{O}(\log^2 k)$, and read locality of $\mathcal{O}(k^\epsilon)$ for any constant $\epsilon < 1$. Next, we consider the case where the encoder and decoder share a secret state and the adversary is computationally bounded. Here too, we obtain local updatability and decodability for the Prefix Hamming metric. Furthermore, we also ensure that the local decoding algorithm never outputs an incorrect message – even when the adversary can corrupt an arbitrary number of bits of the codeword. We call such codes locally updatable locally decodable-detectable codes (LULDDCs) and obtain dramatic improvements in the parameters (over the information-theoretic setting). Our codes have constant rate, an update locality of $\mathcal{O}(\log^2 k)$ and a read locality of $\mathcal{O}(\lambda \log^2 k)$, where λ is the security parameter.

Finally, we show how our techniques apply to the setting of dynamic proofs of retrievability (DPoR) and present a construction of this primitive with better parameters than existing constructions. In particular, we construct a DPoR scheme with linear storage, $\mathcal{O}(\log^2 k)$ write complexity, and $\mathcal{O}(\lambda \log k)$ read and audit complexity.

1 Introduction

Standard error correcting codes (ECC) enable the recovery of a message even when a large fraction of its codeword is corrupted. One disadvantage of ECCs is that, in order to read even a single bit of the data, the entire codeword needs to be decoded. This becomes very inefficient if a user frequently needs to access specific parts of the underlying data. (For example, think of the data as being a user's movie preferences. Then in order to learn his rating of a specific movie, we would need to decode the entire codeword.)Locally

[*]Email: Microsoft Research, India, Email: `nichandr@microsoft.com`. Part of this work was done while this author was at AT&T Labs - Security Research Center, NY.

[†]Department of Computer Science, UCLA, Email: `bhavanak@cs.bu.edu`. Research supported in part by NSF grants CNS-0830803; CCF-0916574; IIS-1065276; CCF-1016540; CNS-1118126; CNS-1136174; and in part by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

[‡]Department of Computer Science and Mathematics, UCLA, Email: `rafail@cs.ucla.edu`. Research supported in part by NSF grants CNS-0830803; CCF-0916574; IIS-1065276; CCF-1016540; CNS-1118126; CNS-1136174; US-Israel BSF grant 2008411, OKAWA Foundation Research Award, IBM Faculty Research Award, Xerox Faculty Research Award, B. John Garrick Foundation Award, Teradata Research Award, and Lockheed-Martin Corporation Research Award. This material is also based upon work supported by the Defense Advanced Research Projects Agency through the U.S. Office of Naval Research under Contract N00014-11-1-0392. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

decodable codes LDCs, introduced by Katz and Trevisan [15], overcome this problem and allow recovery of a single symbol of the message by reading only a few symbols of the potentially corrupted codeword. Another disadvantage of standard ECCs is that, in order to change even a single bit of the data, the entire codeword needs to be recomputed. A natural question to ask is: can we obtain codes which also allow us to change the underlying data by rewriting only a few symbols of the codeword? That is,

Can we build an ECC that allows you to decode and update the message by reading and/or modifying sub-linear number of symbols of the codeword?

In this work, we explore this question and its cryptographic connection.

1.1 Codes with locality

Locally Decodable Codes. As mentioned before, locally decodable codes (LDCs), introduced by Katz and Trevisan [15] are a class of error correcting codes, where every bit of the message can be probabilistically decoded by reading only a few bits of the (possibly corrupted) codeword. In more detail, a binary locally decodable code encodes messages in $\{0,1\}^k$ into codewords in $\{0,1\}^n$. The parameters of interest in such codes are: a) the rate of the code $\rho = \frac{k}{n}$; b) the distance δ, which signifies that the decoding algorithm succeeds even when δn of the bits of the codeword are corrupted; c) the locality r which denotes the number of bits of the codeword read by the decoding algorithm; and d) the error probability ϵ that denotes that for every bit of the message, the decoding algorithm successfully decodes it with probability $1-\epsilon$. Ideally, one would like to minimize both the length of the code as well as the locality; unfortunately, there is a trade-off between these parameters. On the one hand, we have the Hadamard code that has a locality of 2; however its length is exponential in k. (Indeed, the best code length for LDCs with constant locality are super-polynomial in k [29, 7, 5].). On the other hand, the best known codes with constant rate, [16, 10, 13], have a locality of $\mathcal{O}(n^{\epsilon})$ for any constant $0 < \epsilon < 1$. For a survey on locally decodable codes, see Yekhanin's survey [30].

Locally Updatable and Locally Decodable Codes. As we mentioned before, LDCs (and error correcting codes in general) are extremely useful as they provide reliability even when many bits of the codeword may be corrupted; unfortunately, the (unavoidable) price that we pay is that even small changes to the message result in a large change to the codeword. In this work, we ask *"can we have locally decodable codes that are locally updatable?"*. That is, can we have locally decodable codes such that in order to obtain a codeword of message m' from a codeword of message m (where m and m' differ only in one bit) one only needs to modify a few bits of the codeword? We call such codes locally updatable and locally decodable codes (LULDCs); the number of bits that are modified by the update algorithm is then referred to as the *update locality* and the number of bits read by the (local) decoding algorithm is referred to as the *read locality.*

The Prefix Hamming Metric. As in the case of LDCs, our goal is to tolerate a constant fraction of errors while achieving subliniear locality (for both read and update). However, a little thought reveals that updatability and error correction are conflicting goals – if a code tolerates a δ-fraction of errors then, to change even one bit of the data, at least 2δ-fraction of the codeword symbols do need to be re-written.

In light of this, we consider a weaker, yet meaningful, adversarial model of corruption. In this model, the adversary is still allowed to corrupt constant fraction of the bits of the codeword. However, the bits of the codeword have an "age" associated with them and the adversary is allowed to corrupt fewer of the younger/newer bits and is allowed to corrupt many of the older bits. Whenever we touch (i.e., write) a particular bit i of the codeword during an update procedure, this bit becomes a young bit with an age less than every other bit in the codeword. At this point of time, the i^{th} bit of the codeword is the youngest bit in the codeword. Now, suppose we touch the j^{th} bit of the codeword, then this bit becomes the youngest bit, with the i^{th} bit now becoming the second youngest bit of the codeword and so on. Note that if we were to now touch the i^{th} bit, it would once again become the youngest bit of the codeword.

We allow the adversary to corrupt a constant fraction of the bits of the codeword subject only to one constraint – he never corrupts more than a δ fraction of the t youngest bits (for all $1 \leq t \leq n$). We call this metric the *Prefix Hamming Metric.* This metric models a situation where the longer the time a bit of the codeword resides in the system, the easier it is for an adversary to corrupt it. That is, stored data (codeword bits) gets "stale" unless refreshed, and hence the more time the data is untouched, the more errors it will have.

Comparison with Tree Codes. Our error model is similar to the one considered by Schulman [24],[25] in his seminal work on Tree Codes. Tree codes were specifically designed for streaming messages and allow the encoding of messages one bit at a time; the corresponding codeword symbol for every bit of the message is obtained by traversing down a tree. The codeword of the message is obtained by simply concatenating all the individual codeword symbols. Schulman's code guarantees the following: consider any two (different) paths of length t beginning at a particular node in the tree (that denote two different messages); then, the codewords corresponding to these messages have Hamming distance at least αt (for some constant α). Alternately viewed, at any given instance, as long as the adversary does not corrupt more than a α fraction of the t most recently transmitted codeword symbols, the codeword will decode to the correct message. Tree codes were designed for arbitrary (polynomial length) messages; however, we do not know of explicit constructions of tree codes with constant rate.

In our work, the message and codeword lengths are fixed in advance. But the message bits can be *updated* in a streaming fashion by rewriting certain bits of the codeword. Our adversarial error model says the following: at any given instance, as long as the adversary does not corrupt more than a particular constant fraction of the t most recently rewritten bits of the codeword (for all t), the codeword will decode to the correct message.

1.2 Our Results

Information-theoretic Codes. We first construct an LULDC in the information-theoretic setting for the Prefix Hamming metric. We define this metric and such codes in detail in Section 2; for now, we give an overview of the result and the parameters that we achieve.

- *Result 1 (Informal):* We construct binary LULDCs for the Prefix Hamming metric for messages in $\{0,1\}^k$. Our codes have a rate of $\mathcal{O}(1)$, an amortized update locality of $\mathcal{O}(\log^2 k)$ and a worst case read locality of $\mathcal{O}(k^\epsilon)$ for any constant $\epsilon < 1$. For codes that operate on a larger alphabet Σ, with $|\Sigma| \geq \log k$, we can improve the update locality to $\mathcal{O}(\log k)$ (other parameters remaining the same).

Computational Codes. Next, we consider a scenario where the encoder and decoder share a secret state S and where the adversary is computationally bounded. In such a setting, we are able to provide the added guarantee that the (local) decoding algorithm never outputs an incorrect message, irrespective of the number of corrupted bits in the codeword. For the sake of clarity, we refer to such codes as locally updatable and locally decodable-detectable codes (LULDDCs). In addition to providing stronger guarantees, we also obtain dramatic improvements over the parameters achieved by our information-theoretic LULDC construction. In particular, we obtain the following parameters:

- *Result 2 (Informal):* We construct binary LULDDCs for messages in $\{0,1\}^k$. Our codes have constant rate, an amortized update locality of $\mathcal{O}(\log^2 k)$ and a worst case read locality of $\mathcal{O}(\lambda \log^2 k)$, where λ is the security parameter of the system.

Finally, we note that our techniques for building LULDDCs lend themselves to the construction of a Dynamic Proof of Retrievability (DPoR) scheme. Below we discuss our result on DPoR, which we believe, is of independent interest.

Dynamic Proofs of Retrievability. Informally, a proof of retrievability allows a client to store data on an untrusted server and later on, obtain a short proof from the server, that indeed all of the client's data is present on the server. In other words, the client can execute an audit protocol such that any malicious server that deletes or changes even a single bit of the client's data will fail to pass the audit protocol, except with negligible probability in the security parameter[1]. Proofs of retrievability, introduced by Juels and Kaliski [14], were initially defined on static data, building upon the closely related notion of sublinear authenticators defined by Naor and Rothblum [18]. Several works have studied the efficiency of such schemes [26, 6, 2, 1] with the work of Cash, Küpçü, and Wichs [3] considering the notion of proofs of retrievability on dynamically changing data; in other words, they constructed a proof of retrievability scheme that allowed for efficient updates to the data. Their DPoR scheme has $\mathcal{O}(k)$ server storage, $\mathcal{O}(\lambda)$ client storage, $\mathcal{O}(\lambda \log^2 k)$ read complexity, $\mathcal{O}(\lambda^2 \log^2 k)$ write and audit complexity[2]. We improve their parameters and obtain the following result:

- *Result 3 (Informal):* We obtain a construction of a dynamic proof of retrievability with $\mathcal{O}(k)$ server storage, $\mathcal{O}(\lambda)$ client storage, $\mathcal{O}(\lambda \log k)$ read complexity, $\mathcal{O}(\log^2 k)$ write complexity and $\mathcal{O}(\lambda \log k)$ audit complexity[3].

1.3 Our Techniques

We now give a high-level overview of the techniques used to obtain our results. We shall make use of the hierarchical data structure introduced by Ostrovsky [19],[20] in the context of oblivious RAMs. Oblivious RAMs [8, 19] allow efficient random access to memory without revealing the access pattern to an adversary that observes the reads and writes made to memory. ORAM protocols hide the access pattern by making use of several tools carefully put-together. Here we distill out exactly what we need for our construction. In particular, we will primarily make use of the hierarchical data structure, coupled with certain other techniques, to construct LULDCs.

Hierarchical Data Structure. At a high level, this data structure comprises of buffers $\mathsf{buff}_0, \cdots, \mathsf{buff}_\tau$ of increasing size. Buffer buff_i has 2^i elements and each element in the buffer is of the form $(\mathsf{index}, \mathsf{value})$. In addition, there is a special buffer, buff^* which has all bits of the message in order (and hence without an index). To read a value at a particular index i, we scan the buffers in top-down manner. To write (or re-write) a value v at index i, we write it to the top buffer. Writing to buffers evenutally fills them up. To handle this, buffers are periodically combined and moved to an empty buffer in some lower level in a careful manner.

LULDCs for the Prefix Hamming Metric. The first idea behind our construction in the information-theoretic setting is as follows. To achieve local decodability, we encode each buffer (including buff^*) with a locally decodable code (LDC). Whenever we wish to update a bit of the message, we will write it to the topmost buffer buff_0 and re-encode the top buffer using an LDC to encode this latest update. Naturally, the top buffer gets full after an update operation. Whenever we encounter a full buffer, we move its

[1]Formally, this guarantee is provided by requiring the existence of an extractor algorithm, that given black-box rewinding access to any malicious server that passes the audit with non-negligible probability, will extract all of the client's data, except with negligible probability.

[2]The work of Cash *et al.* [3] considered the complexity without explicitly including the (storage as well as verification) complexity of the MAC; if one did this, then the parameters obtained will all be larger by a factor of $\mathcal{O}(\lambda)$.

[3]These parameters include the cost for storage and verification of the MACs.

contents to the buffer below it (that is, we decode the entire buffer, combine top level buffers together and re-encode them at a level below, once again using an LDC for the encoding). When we wish to (locally) decode a particular index i of the message, we scan buffers one-by-one starting with topmost buffer. Now, note that we need to check if a particular index is found in a buffer or not. In order to do this, we always ensure that buffers store $(\mathsf{index}, \mathsf{value})$ pairs that are sorted according to the index value. This will enable us to perform a binary search (decoded via the underlying LDC) to check if a buffer contains a particular index i or not. Since we are performing the binary search via the decode algorithm of the underlying LDC, we must ensure that the decode does not fail with too high a probability; hence, we repeat the decode procedure at each level some fixed number of times to ensure this and make sure that our overall local decoding algorithm succeeds except with ϵ probability. When the index is found, we stop searching lower level buffers and output the value retrieved (our construction will always ensure that if an index value was updated, then the latest value of the index will be stored at a high level buffer). If the index is not found, then we read the corresponding element from the special buffer buff^*, once again using the underlying LDC.

Since we must store every updated element as a $(\mathsf{index}, \mathsf{value})$-pair, the above described technique will decrease the rate of the code by a factor of $\mathcal{O}(\log k)$. Hence, in order to ensure that our code has constant rate, we carefully choose the total number of buffers $\tau + 1$ in our construction to ensure that we obtain constant rate codes and yet achieve good update and read locality.

Now, in the above construction, we first show that the decode and update algorithms succeed (with small locality) as long as an adversary corrupts only a constant fraction of the bits of each buffer. We then proceed to show that if an adversary corrupts bits of the codeword according to the Prefix Hamming metric, then he can only corrupt a constant fraction of the bits of each buffer (within a factor of 2). This gives us our construction of LULDCs.

Computational LULDDCs. To obtain our construction in the computational setting, at a high level, we follow our information-theoretic construction. However, there are three main differences. First, when decoding the i^{th} bit of the codeword, we still scan each buffer to see if a "latest" copy of the i^{th} bit is present in that buffer. However, now, because we are in the computational setting, we no longer need to store the buffer in sorted order and perform a binary search. Instead, we simply use hash functions to check if a particular index is present in a buffer or not. Furthermore, we use cuckoo hash functions to minimize our read locality in this case. Second, we store each buffer using a computational LDC that has constant rate and $\mathcal{O}(\lambda)$ locality (such codes are obtained through the construction of Hemenway *et al.* [12]). Third, we authenticate each bit of the codeword using a message authentication code so that we never decode incorrectly (irrespecitve of the number of errors that the adversary introduces).

The above ideas do not suffice for our construction: in particular, if we applied these techniques, we do not obtain a constant rate code as MACing each bit of the codeword would result in a $\mathcal{O}(\lambda)$ blowup in the rate of the code. One could think of MACing $\mathcal{O}(\lambda)$ bits of the codeword, block by block, but then this would result in a $\mathcal{O}(\lambda^2)$ blowup in the read locality, as we must read λ bits now in each buffer through the underlying LDC. In order to obtain our result, we MAC each bit of the codeword using a constant size MAC; this technique is similar in spirit to the use of constant size MACs when authenticating codewords in the context of optimizing privacy amplification protocols [4]. To obtain our result, we make a careful use of these constant size MACs to verify the correctness of a codeword as well as to decode correctly (except with negligible probability).

Dynamic Proofs of Retrievability. Cash *et al.* [3] showed how to convert any oblivious RAM (ORAM) protocol that satisfied a special property (which they define to be next-read-pattern-hiding (NRPH)) into a dynamic proof of retrievability (DPoR) scheme. We show that we do not need an ORAM scheme with this property and the techniques used to construct LULDDCs can be used to directly build a DPoR scheme. Moreover, we do not need to hide the read and write access pattern, thereby leading to significant savings in the complexity. In particular, we show, that by encoding each buffer of the ORAM *structure* using a standard error correcting code (that is also appropriately authenticated with constant size MACs), and additionally storing authenticated elements of the raw data in the clear, we can use the techniques developed for LULDDCs to construct a DPoR scheme with $\mathcal{O}(k)$ server storage, $\mathcal{O}(\lambda)$ client storage, $\mathcal{O}(\lambda \log k)$ read complexity, $\mathcal{O}(\log^2 k)$ write complexity and $\mathcal{O}(\lambda \log k)$ audit complexity. Moreover, these parameters include the cost for storage and verification of the MACs.

We remark here, that in a concurrent and independent work, Shi *et al.* [27] show how to construct a dynamic proof of retrievability scheme using techniques similar in spirit to ours. On the one hand, our construction achieves slightly better parameters than theirs: namely, they obtain a write complexity of $\mathcal{O}(\lambda \log k)$ (as opposed to $\mathcal{O}(\log^2 k)$ in our work) and audit complexity of $\mathcal{O}(\lambda^2 \log k)$ (as opposed to $\mathcal{O}(\lambda \log k)$ in our work). On the other hand, their protocol is publicly verifiable, while our construction is not.

1.4 Organization of the paper

In Section 2, we introduce our notion of locally updatable and locally decodable codes as well as formally define the Prefix Hamming metric. We present our construction of locally updatable and locally decodable codes for the Prefix Hamming metric in Section 3. We consider the computational setting in Section 4 and construct locally updatable and locally decodable-detectable codes. Finally, we give our construction of a dynamic proof of retrievability scheme in Section 5. Due to the lack of space, we present further details of our schemes and proofs in the Appendix.

2 Definitions

Notation. Let k denote the length of the message. Let $\mathcal{M}$ denote a metric space with distance function $\mathsf{dis}(,)$. Let the set of all codewords corresponding to a message m be denoted by $\mathcal{C}_m$ – we will define this set shortly. Let n denote the length of all codewords. $m(i)$ denotes the i^{th} bit of message m for $i \in [k]$, where $[k]$ denotes the set of integers $\{1, 2, \cdots, k\}$.

2.1 Codes with Locality

Locally decodable codes. We first recall the notion of locally decodable codes. Informally, locally decodable codes allow the decoding of any bit of the message by only reading a few (random) bits of the codeword. Formally:

Definition 1 (Locally decodable codes)**.** *A binary code* $\mathcal{C} : \{0,1\}^k \to \{0,1\}^n$ *is* $(k, n, r_k, \delta, \epsilon)$-locally decodable *if there exists a randomized decoding algorithm* $\mathcal{D}$ *such that*

1. $\forall m \in \{0,1\}^k, \forall i \in [k], \forall c_m \in \mathcal{C}_m$*, and for all* $\hat{c}_m \in \{0,1\}^n$ *such that* $\mathsf{dis}(c_m, \hat{c}_m) \leq \delta n$*:*
$$\Pr[\mathcal{D}^{\hat{c}_m}(i) = m(i)] \geq 1 - \epsilon,$$
where the probability is taken over the random coins of the algorithm $\mathcal{D}$*.*

2. $\mathcal{D}$ *makes at most* r_k *queries to* $\hat{c}_m$*.*

Locally updatable codes. We now define the notion of locally updatable and locally decodable codes. A basic property that updatable codes must have is that one can convert a codeword of message m into a codeword of message m' (where m' and m differ possibly only at the i^{th} position), by changing only a few bits of the codeword of m. However, we will obtain codes that have a stronger property; namely, will ensure that we can convert any string that decodes to m into a string that decodes to m'. That is, let m and m' be two k-bit messages that (possibly) differ only in the i^{th} position, where $m'(i) = b_i$. For some appropriate metric space that defines a measure of closeness, given a string $\hat{c}_m$ that is "close" to a codeword for message m, our update algorithm (that writes bit b_i at position i) must convert $\hat{c}_m$ into a new string $\hat{c}_{m'}$ that is now "close" to a codeword for message m'. Furthermore, the update algorithm must query and change only a few bits of $\hat{c}_m$. Additionally, our code should also be locally decodable.

Before we present the formal definition of a locally updatable and locally decodable code, we first need to define the set of codewords $\mathcal{C}_m$ for a message m. Conceptually, with a locally updatable code, there are two kinds of codewords that correspond to a message m – ones obtained by computing $\mathcal{E}(m)$ and those obtained by computing updating the codeword of different message m'.

We let $m^{i^{b_i}}$ denote a message that is exactly the same as m except possibly at the i^{th} position (where it is b_i). Note that $m^{i^{b_i}}$ maybe equal to m itself.

Definition 2 (The set $\mathcal{C}_m$). *For a message m, if there exists a message $\bar{m}$, codeword $c_m = \mathcal{E}(\bar{m})$ (possibly $\bar{m} = m$ and $c_m = c_m$) and a (possibly empty) set of indices $\{i_1, \cdots, i_t\}$ such that $m = \bar{m}^{i_1^{b_1} \cdots i_t^{b_t}}$ and $c_m = u(....u(u(c_m, i_1, b_1), i_2, b_2), ..., i_t, b_t)$, then c_m is in the set $\mathcal{C}_m$.*

It is easy to see that $\mathcal{C}_m$ contains all the codewords that decode to m. We now present the formal definition of a LULDC.

Definition 3 (Locally updatable and locally decodable codes (LULDC)). *A binary code $\mathcal{C} : \{0,1\}^k \rightarrow \{0,1\}^n$ is $(k, n, w, r, \delta, \epsilon)$-*locally updatable and locally decodable *if there exist (possibly) randomized algorithms $\mathcal{E}_{\mathsf{LDC}}$, $\mathcal{U}$ and $\mathcal{D}$ such that the following conditions are satisfied:*

1. Local Updatability:
   - (a) *Let $m_0 \in \{0,1\}^k$ and let $c_{m_0} = \mathcal{E}_{\mathsf{LDC}}(m_0)$. Let m_t be a message obtained by any (potentially empty) sequence of updates. ($t = 0$ corresponds to the case where the codeword has not been updated so far.) Then $\forall m_0 \in \{0,1\}^k, \forall c_{m_0} \in \mathcal{C}_{m_0}, \forall t, \forall m_t$, $\forall i_{t+1} \in [k], \forall b_{t+1} \in \{0,1\}$, for all $\hat{c}_{m_t} \in \{0,1\}^n$ such that $\mathsf{dis}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$,*
     - *The actions of $\mathcal{U}^{\hat{c}_{m_t}}(i_{t+1}, b_{t+1})$, change $\hat{c}_{m_t}$ to $u(\hat{c}_{m_t}, i_{t+1}, b_{t+1}) \in \{0,1\}^n$, where $\mathsf{dis}(u(\hat{c}_{m_t}, i_{t+1}, b_{t+1}), c_{m_{t+1}}) \leq \delta n$ for some $c_{m_{t+1}} \in \mathcal{C}_{m_{t+1}}$, where m_{t+1} and m_t are identical except (possibly) at the i_{t+1}^{th} position, where $m_{t+1}(i_{t+1}) = b_{t+1}$.*
   - (b) *The total number of queries and changes that $\mathcal{U}$ makes to the bits of $\hat{c}_m$ is at most w.*
2. Local Decodabilty:
   - (a) *Let m_t denote the latest message. $\forall m_t \in \{0,1\}^k, \forall i \in [k], \forall c_{m_t} \in \mathcal{C}_{m_t}$, and for all $\hat{c}_{m_t} \in \{0,1\}^n$ such that $\mathsf{dis}(c_{m_t}, \hat{c}_{m_t}) \leq \delta n$:*
   $$\Pr[\mathcal{D}^{\hat{c}_{m_t}}(i) = m_t(i)] \geq 1 - \epsilon,$$
   where the probability is taken over the random coins of the algorithm $\mathcal{D}$.
   - (b) *$\mathcal{D}$ makes at most r queries to $\hat{c}_{m_t}$.*

2.2 The Prefix Hamming Metric

If we want codes that are truly updatable, the update locality w needs to be $<< \delta n$. However, as mentioned earlier, we cannot hope to achieve such locality for metrics where an adversary can *arbitrarily* corrupt a constant fraction of the bits of the codeword. (Indeed, if we updated a codeword from c_m to $c_{m'}$ with a locality of w, then by corrupting those w bits of $c_{m'}$, an adversary can ensure that the decoding algorithm does not output the correct message – in particular, the decode algorithm would output m instead of m'.)

In light of this, we turn to a new, yet meaningful metric, for which we can guarantee that even if an adversary corrupts a bounded number of bits of the codeword, though not in a completely arbitrary manner, our decode algorithm still functions correctly. At a high level, bits of the codeword "age" and the adversary can corrupt a fraction of the bits as a function of their age. Our metric relies crucially on the order in which bits were written or updated during the creation of a codeword – nonetheless, we abuse notation and refer to Prefix-Hamming as a metric. We first define the "age-ordering" of a codeword.

Definition 4 (Age-ordering of a codeword)**.** *Let $c \in \{0,1\}^n$. Let w_1 denote the index/position of the most recent bit of the codeword that was either written or updated. Let w_2 denote the unique index of the next most recent bit that was written/updated and so on, with w_n denoting the index of the earliest bit written (in comparison with the rest of the bits of the codeword). We call $\mathsf{w}_1, \cdots, \mathsf{w}_n$ the* age-ordering *of c. $c(\mathsf{w}_i)$ denotes the bit value of the codeword at index w_i. For all $1 \leq t \leq n$, let $c[1,t]$ denote the bits $c(\mathsf{w}_1), \cdots, c(\mathsf{w}_t)$.*

We are now ready to define how the adversary in our model can corrupt bits of the codeword. That is, we define our metric space and its distance function.

Definition 5 (The Prefix Hamming Metric)**.** *Let $c \in \{0,1\}^n$. Let $\mathsf{w}_1, \cdots, \mathsf{w}_n$ denote the age-ordering of c. Let $c' \in \{0,1\}^n$ and for $1 \leq t \leq n$, let $c'[1,t]$ denote the bits $c'(\mathsf{w}_1), \cdots, c'(\mathsf{w}_t)$. We say that the* Prefix Hamming *distance between c and c', denoted by $\mathsf{Prefix}(c,c')$ is $\leq \delta n$ if for all $1 \leq t \leq n$, $\mathsf{Hamm}(c[1,t], c'[1,t]) \leq \delta t$, where $\mathsf{Hamm}(x,y)$ denotes the Hamming Distance between any two strings x and y of equal length.*

3 LULDCs for the Prefix Hamming Metric

3.1 Our results

In this section, we show how to construct locally updatable locally decodable error correcting codes (LULDCs) that are resilient to a constant fraction of adversarial errors for the Prefix Hamming metric that we defined in Section 2.2. Formally, we show:

Theorem 1. *Let $\tau = \log k - \log(\log k + 1) - 1$. Let $\mathcal{C}_{\mathsf{LDC}}$ be a family of $(k_i, n_i, r_i, \epsilon, \delta)-$locally decodable code for Hamming distance with algorithms $(\mathcal{E}_{\mathsf{LDC}}, \mathcal{D}_{\mathsf{LDC}})$, where $k_i = 2^i(\log k + 1)$ for all $0 \leq i \leq \tau$. Additionaly, let $\mathcal{C}_{\mathsf{LDC}}$ contain a $(k^*, n^*, r^*, \epsilon, \delta)-$locally decodable code for Hamming distance, where $k^* = k$. Let $\rho_i = \frac{k_i}{n_i}$ for all i and let $\rho^* = \frac{k^*}{n^*}$. Then there exists a $(k, n, w, r, \epsilon, \frac{\delta}{2}) - \mathsf{LULDC}$ code $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$ for the Prefix Hamming metric achieving the following parameters:*

- ***Length of the code** (n)**:** $n = n^* + \sum_{i=0}^{\tau} n_i$.*
- ***Update locality** (w)**:** $w = (\log k + 1) \sum_{i=0}^{\tau} \frac{1}{\rho_i} + \frac{\log k + 1}{\rho^*}$, in the worst case.*
- ***Read locality** (r)**:** $r = \frac{8(1-\epsilon)}{(1-2\epsilon)^2} T \log \frac{T}{\epsilon} + r^*$, where $T = (\log k + 1)\left(r_0 + \sum_{1 \leq j \leq \tau} j r_j\right)$, in the worst case.*

As a corollary to Theorem 1, using the LDCs from [16, 10, 13] we obtain:

Corollary 1. *For every $\epsilon, \alpha > 0$, there exists a $(k, n, w, r, \epsilon, \delta) - \mathsf{LULDC}$ code $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$ for the Prefix Hamming metric achieving the following parameters, for some constant $0 < \delta < \frac{1}{4}$:*

- ***Length of the code** (n)**:** $n = \frac{2k}{1-\alpha}$.*

- ***Update locality*** (w)***:*** *$w = \mathcal{O}(\log^2 k)$, in the amortized sense.*
- ***Read locality*** (r)***:*** *$r = \mathcal{O}(k^{\epsilon'})$, for some constant ϵ', in the worst case.*

Large alphabet codes. We remark that for codes over larger alphabet Σ, with $|\Sigma| \geq \mathsf{c}\log k$ for some constant c, we can modify our code to obtain a better update locality of $\mathcal{O}(\log k)$ (other parameters remaining the same).

3.2 Code description

We will now construct the codes that will prove Theorem 1. Our codeword will have a structure similar to that of the hierarchical data-structure used by Ostrovsky [19, 20] in the construction of oblivious RAMs. Let $\tau = \log k - \log(\log k + 1) - 1$. Each codeword of $\mathcal{C}$ will consist of $\tau + 1$ buffers, $\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau$ and a special buffer buff^*. We will ensure that as updates take place, at any point of time, buff_i will be either empty or full (for all $i > 0$). A full buffer, buff_i, will contain an encoding of a set μ_i of 2^i elements. In particular, $\mu_i = [(a_i^1, v_i^1), \ldots, (a_i^{2^i}, v_i^{2^i})]$ where a_i^j is an address (between 0 and $k-1$) and v_i^j is the value corresponding to it. buff_i (when non-empty) will store $\psi_i = \mathcal{E}_{\mathsf{LDC}}(\mu_i)$. The special buffer buff^* will contain an encoding of the bits of the entire message in order, without address values; in particular, buff^* stores $\psi^* = \mathcal{E}_{\mathsf{LDC}}(m)$.

Encode algorithm. Our encoding algorithm works as follows:

Algorithm $\mathcal{E}(m)$:

1. Creates the $\tau + 1$ empty buffers $(\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau)$.
2. Let $\mu^* = \{m(1), \cdots, m(k)\}$, where $m(i)$ denotes the i^{th} bit of the message. It computes $\psi^* = \mathcal{E}_{\mathsf{LDC}}(\mu^*)$ and stores it in buff^*.

Local update algorithm. Our update algorithm updates a string $\hat{c}_m$ (such that $\mathsf{Prefix}(\hat{c}_m, c_m) \leq \delta n$, for some $c_m \in \mathcal{C}_m$) into a string $\hat{c}'_m$, setting $m(i)$ to b_i.

Algorithm $\mathcal{U}^{\hat{c}_m}(i, b_i)$:

1. If the first buffer is empty, computes $\mathcal{E}_{\mathsf{LDC}}(i, b_i)$ and stores it in buff_0.
2. If the first buffer is non-empty, it finds the first empty buffer. Let this be buff_j. It decodes all the buffers above it to get μ_0 to μ_{j-1} [4]. Recall that each μ_h is a set of (a, v) pairs where a denotes the address (of length $\log k$) and v denotes a value ($\in \{0, 1\}$). It merges all these pairs of values as well the pair (i, b_i) in a sorted manner (where the sorting is done on address) and stores it in μ_j. Note, there are 2^j elements and therefore μ_j is now a full buffer.

   Handling Repetitions: While merging elements from multiple buffers, we might encounter repetition of addresses. Instead of removing repetitions, we simply ensure that all values stored in the buffers until $j - 1$ store only the "latest value" corresponding to the repeated address. (The latest value is easy to determine – it is the first value corresponding to the buffer that you encounter when reading the buffers in a top-down manner. Of course, for the address being inserted, namely i, the latest value will be b_i.)

[4] Here, these buffers need not be decoded using the local decoding algorithm and one can obtain perfect correctness by simply running the standard decoding algorithm for the error correcting code.

3. The update algorithm computes $\psi_j = \mathcal{E}_{\mathsf{LDC}}(\mu_j)$ and stores it in buff_j.

4. The buffers from $\mu_{j-1} \ldots \mu_0$, in that order, are now set to empty by writing special symbols into it. Looking ahead, the order in which this done is important as this ensures that buff_h always has bits that are "younger" than the bits in buff_{h+1} for all h (when considering the age-ordering of the bits).

5. If none of the buffers are empty, namely, all buffers $\mathsf{buff}_0, \cdots, \mathsf{buff}_\tau$ are full, then the update algorithm simply re-computes a new encoding of the message using the LDC encode algorithm and stores it in buff^*. In other words, the algorithm decodes all the buffers to obtain the latest value of each bit, concatenates these bits together to form $\mu^* = \{m(1), \cdots, m(k)\}$ and encodes these bits to compute $\psi^* = \mathcal{E}_{\mathsf{LDC}}(\mu^*)$. Once again, the buffers from buff_τ to buff_0 are set to empty in that order by writing special symbols into it.

Local decode algorithm. Recall that our buffers satisfy the following conditions:

- The buffers are always sorted (based on the address a).
- If the address a "appears" in the same buffer multiple times, then all values corresponding to this address are the same. (This is guaranteed by the way we handle repetitions during our merging procedure.)
- Finally, across multiple buffers, the most recent value corresponding to an address appears in the higher buffer (i.e. a lower buffer value).

Algorithm $\mathcal{D}^{\tilde{c}_m}(i)$:

1. The decode algorithm starts with the top-most buffer (buff_0) and proceeds downwards until it finds the address i.

2. To search a buffer buff_j for the element i, it performs a binary search on elements stored in that buffer. Because buff_j contains an LDC encoding, we additionally need to use $\mathcal{D}_{\mathsf{LDC}}()$ algorithm to access these j elements. Since $\mathcal{D}_{\mathsf{LDC}}()$ might fail with ϵ probability to decode one coordinate of the underlying message, we need to repeat $\mathcal{D}_{\mathsf{LDC}}()$ multiple (i.e. λ) times to amplify the success probability (where λ is a carefully chosen parameter).

3. If element i is not found in any of the buffers buff_0 through buff_τ, then the algorithm simply (locally) decodes the i^{th} element from buff^* (which contains an LDC encoding of the message).

3.3 Proof of Theorem 1

We shall now prove Theorem 1; namely, we show that the construction described above in Section 3.2 is a locally updatable, locally encodable binary error correcting code (for the Prefix Hamming metric) with the parameters listed in Theorem 1. Instead of directly proving Theorem 1, we will instead show that the construction is a LULDC for a metric that we call the *Buffered-Hamming* metric. From this, the proof of Theorem 1 directly follows. We shall now define the Buffered-Hamming metric and its associated distance function.

Buffered-Hamming Distance. Let $c \in \{0,1\}^n$ comprise of buffers $\mathsf{buff} = \mathsf{buff}_0, \ldots, \mathsf{buff}_q$ of lengths $n_0, \ldots, n_q$ respectively. Let $c' \in \{0,1\}^n$ be another string with buffers $\mathsf{buff}' = \mathsf{buff}'_0, \ldots, \mathsf{buff}'_q$. Then we say that Buffered-Hamming Distance, $\mathsf{BHdis}(c_m, c') \leq \delta n$ if $\forall i$ $\mathsf{Hamm}(\mathsf{buff}_i, \mathsf{buff}'_i) \leq \delta n_i$.

Lemma 1. *Let $\tau = \log k - \log(\log k + 1) - 1$. Let $\mathcal{C}_{\mathsf{LDC}}$ be a family of $(k_i, n_i, r_i, \epsilon, \delta)-$locally decodable code for Hamming distance with algorithms $(\mathcal{E}_{\mathsf{LDC}}, \mathcal{D}_{\mathsf{LDC}})$, where $k_i = 2^i(\log k+1)$ for all $0 \le i \le \tau$. Additionaly, let $\mathcal{C}_{\mathsf{LDC}}$ contain a $(k^*, n^*, r^*, \epsilon, \delta)-$locally decodable code for Hamming distance, where $k^* = k$. Let $\rho_i = \frac{k_i}{n_i}$ for all i and let $\rho^* = \frac{k^*}{n^*}$. Then the construction described above in Section 3.2 is a $(k, n, w, r, \epsilon, \delta) - \mathsf{LULDC}$ code $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$ for the Buffered-Hamming metric achieving the following parameters:*

- ***Length of the code** (n): $n = n^* + \sum_{i=0}^{\tau} n_i$.*
- ***Update locality** (w): $w = (\log k + 1) \sum_{i=0}^{\tau} \frac{1}{\rho_i} + \frac{\log k + 1}{\rho^*}$, in the worst case.*
- ***Read locality** (r): $r = \frac{8(1-\epsilon)}{(1-2\epsilon)^2} T \log \frac{T}{\epsilon} + r^*$, where*

  $T = (\log k + 1) \left(r_0 + \sum_{1 \le j \le \tau} j r_j \right)$, in the worst case.

Proof. **Length of the code.** Recall that we have buffers in levels $0, 1, \ldots, \tau$. Each buffer encodes a message μ_j of length $k_j = 2^j(\log k + 1)$; the encoding is denoted ψ_j and is of length n_j. Buffer buff^* contains an LDC encoding of a message of length k. It is easy to see that the length of the code $n = n^* + \sum_{i=0}^{\tau} n_i$.

Read locality and Decode Correctness. We now analyze the read locality and the decodability of our code. Let $\hat{c}_m$ be the given (corrupted) codeword and let $\hat{c}_m$ be such that $\mathsf{BHdis}(\hat{c}_m, c_m) \le \delta n$, where $c_m \in \mathcal{C}_m$ for the most "recent" $m \in \{0,1\}^k$ (obtained after an encoding of a message and possible subsequent updates). We compute the read locality of our local decoding algorithm and also prove that for all $i \in [k]$, the decoding algorithm will output $m(i)$ with probability $\ge 1 - \epsilon$.

Let $\mu = \{\mu_0, \ldots, \mu_\tau\}$ and let $\psi = \{\psi_0, \ldots, \psi_\tau\}$, where $\psi_i = \mathcal{E}_{\mathsf{LDC}}(\mu_i)$. Let $\mathcal{C}^j_{\mathsf{LDC}}$ denote the locally decodable code used to encode μ_j. We use $\mu_x(y)$ to denote the y^{th} bit of μ_x. Recall that in order to read an index i of the message $m = m_0, \ldots, m_k$, the algorithm $\mathcal{D}^\psi(i)$ does a binary-search on the buffers in a top-down manner to see if there is a value corresponding to address i. The worst case locality occurs when m_i has never been updated. In this case, the binary search needs to be done on every buffer and will then conclude by performing a (local) deocoding for the i^{th} bit in buff^* which contains $\psi^* = \mathcal{E}_{\mathsf{LDC}}(m)$.

We first calculate the number of *bits* of μ_j (for $j \ge 1$), one would need to read, if we were doing the binary search directly over μ_j. There are 2^j elements i.e.,(a, v) pairs, in level j. So the binary search would need to look at j elements (in the worst case). Each element has length $\log k + 1$. The total number of bits of μ_j we access if we did a binary search over μ_j would be $j(\log k + 1)$ (for $j \ge 1$). $\mathcal{D}^\psi(i)$ learns these bits by making calls to $\mathcal{D}^{\psi_j}_{\mathsf{LDC}}$ which has locality r_j. Therefore the number of bits of ψ_j, read via calls to $\mathcal{D}^{\psi_j}_{\mathsf{LDC}}$, is at most $j(\log k + 1) r_j$ (for $1 \le j \le \tau$) and $(\log k + 1) r_j$ (for $j = 0$). (Recall, that in buff^*, a binary search is not performed and the decode algorithm simply decodes the (single) i^{th} bit of the message via LDC decode calls to ψ^*.)

Define a set Read and add (x, y) to it if $\mu_x(y)$ was accessed; let $T = |\mathsf{Read}|$. Then,

$$T = (\log k + 1) \left(r_0 + \sum_{1 \le j \le \tau} j r_j \right) \text{ and} \tag{1}$$

$$\text{the total decode locality } r = T\lambda + r^* \tag{2}$$

Equation 2 follows from that fact that in order to read a bit of μ_j correctly, we must amplify the success probability of $\mathcal{D}^{\psi_j}_{\mathsf{LDC}}$, by taking the majority of λ executions (Note, that just as in standard LDCs, even though our LULDC allows a decoding error of ϵ, we cannot afford to have an error of ϵ while reading every bit of our binary search in every buffer, as this would lead to an overall worse error probabaility). If the element is not found in the buffers buff_0 through buff_τ, then we only need to read 1 bit of the underlying message via a single LDC decoding call to ψ^* and hence we pay an additional r^* in our read locality.

In order to determine r, all that is left, is for us to determine λ. Let the variable $\#\mathsf{Succ}(x,y)$ denote the number of calls such that $\mathcal{D}_{\mathsf{LDC}}^{\psi'_x}(y) = \mu(x,y)$. Let $\mathsf{SuccRead}(x,y)$ denote that event that $\#\mathsf{Succ}(x,y) > \frac{\lambda}{2}$. First, since $\hat{c}_m$ is such that $\mathsf{BHdis}(\hat{c}_m, c_m) \leq \delta n$, it follows that, $\mathsf{Hamm}(\psi'_j, \psi_j) \leq \delta|\psi_j|$ for all $0 \leq j \leq \tau$ and $\mathsf{Hamm}(\psi^{*'}, \psi^*) \leq \delta|\psi^*|$. Now, since $\mathcal{C}_{\mathsf{LDC}}^{\psi'_j}$ has error-rate ϵ, $\mathbf{E}[\#\mathsf{Succ}(x,y)] = \lambda(1-\epsilon)$. By the Chernoff bound[5], $\Pr[\#\mathsf{Succ}(x,y) \leq \frac{\lambda}{2}] \leq p = e^{-\frac{\lambda(1-2\epsilon)^2}{8(1-\epsilon)}}$.

In other words,

$$\Pr[\mathsf{SuccRead}(x,y) = 0] \leq p = e^{-\frac{\lambda(1-2\epsilon)^2}{8(1-\epsilon)}} \tag{3}$$
$$\text{i.e.,} \sum_{(x,y)\in\mathsf{Read}} \Pr[\mathsf{SuccRead}(x,y) = 0] \leq Tp. \tag{4}$$

Our goal is to ensure that

$$\Pr\left[\bigwedge_{(\forall(x,y)\in\mathsf{Read})} \mathsf{SuccRead}(x,y) = 1\right] (\geq 1 - Tp) \geq 1 - \epsilon.$$

In other words, we need to set λ such that $Tp \leq \epsilon$. Substituting for $p = e^{-\frac{\lambda(1-2\epsilon)^2}{8(1-\epsilon)}}$, we get that

$$\lambda \geq \frac{8(1-\epsilon)}{(1-2\epsilon)^2} \log\left(\frac{T}{\epsilon}\right).$$

By setting $\lambda = \frac{8(1-\epsilon)}{(1-2\epsilon)^2} \log\left(\frac{T}{\epsilon}\right)$ and substituting in Equation 2, we get that the decode locality,

$$r = \frac{8(1-\epsilon)}{(1-2\epsilon)^2} T \log\frac{T}{\epsilon} + r^*.$$

This proves the correctness and the read locality of our decoding algorithm.

Update Locality and Correctness. First, we count the number of coordinates accessed in order to rewrite one bit of the message m_i. This includes the total number of coordinates read and written.

It is easy to see that in algorithm $\mathcal{U}^{\mathcal{C}_m}(x, b_x)$, buffer buff_j (for $0 \leq j \leq \tau$) is rewritten every 2^j steps. Buffer buff^* is re-written every $2^{\tau+1}$ steps. In 2^j updates (when $j < \tau + 1$), therefore, the total number of bits re-written is

$$\begin{aligned} &= 2^j \frac{|\mu_0|}{\rho_0} + 2^{j-1}\frac{|\mu_1|}{\rho_1} + \ldots + 2^0 \frac{|\mu_j|}{\rho_j} \\ &= 2^j |\mu_0| \sum_{0\leq i\leq j} \frac{1}{\rho_i} \text{ (since } \mu_i = 2\mu_{i-1}, \forall i) \end{aligned}$$

When $j \geq \tau + 1$, buff^* is re-written and hence, in this case, the total number of bits re-written is

[5]Recall that for a variable X with expectation $\mathbf{E}(X)$, the Chernoff bound states that for any $t > 0$, $\Pr[X \leq (1-t)\mathbf{E}(X)] \leq e^{-\frac{t^2\mathbf{E}(X)}{2}}$. In this case, $X = \#\mathsf{Succ}(x,y); \mathbf{E}(X) = \lambda(1-\epsilon); t = \frac{1-2\epsilon}{2-2\epsilon}$.

$$= 2^j \frac{|\mu_0|}{\rho_0} + 2^{j-1}\frac{|\mu_1|}{\rho_1} + \ldots + 2^{j-(\tau+1)}\frac{|\mu_\tau|}{\rho_\tau} + 2^{j-(\tau+1)}\frac{|k^*|}{\rho^*}$$
$$= 2^j|\mu_0| \sum_{0\le i\le\tau} \frac{1}{\rho_i} + 2^{j-(\tau+1)}\frac{|k^*|}{\rho^*}$$

The amortized update locality w per update is

$$|\mu_0| \sum_{0\le i\le\tau} \frac{1}{\rho_i} + \frac{|k^*|}{2^{\tau+1}\rho^*} = (\log k + 1) \sum_{0\le i\le\tau} \frac{1}{\rho_i} + \frac{\log k + 1}{\rho^*}.$$

Achieving a Worst-case Guarantee. Note that, similar to the constructions of oblivious RAMs, one can convert the amortized update locality into a worst-case guarantee on the write locality, by distributing the work over many write operations. At a high level, this works by maintaining an additional "working copy" of data structure. Once levels $1, \ldots, i-1$ of the first data structure are filled in, the contents of level i are computed. This process takes place even as levels $1, \ldots, i-1$ of the second data structure are being filled in. This gives us a worst case write locality of $w = (\log k + 1)\sum_{i=0}^{\tau} \frac{1}{\rho_i} + \frac{\log k+1}{\rho^*}$ for the Buffered Hamming metric. Note, however, that a similar argument does not translate to the setting of the Prefix Hamming metric (since one would need to re-write parts of buffers at various levels at various points of time) and hence we only get an amortized bound for this metric.

To show update correctness, we must now argue, that if we begin the update algorithm with a corrupted codeword $\hat{c}_{m_t}$, such that $\mathsf{BHdis}(\hat{c}_{m_t}, c_{m_t}) \le \delta n$ and update the message m_t to m_{t+1} (where m_t and m_{t+1} differ (possibly) only at the i_t^{th} position, where $m_{t+1}(i_t) = b_{t+1}$), then we modify $\hat{c}_{m_t}$ to $\hat{c}_{m_{t+1}}$ where $\mathsf{BHdis}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \le \delta n$ for some $c_{m_{t+1}}$ that is a codeword of m_{t+1}. To see this, observe that, the update algorithm decodes all buffers $\mathsf{buff}_0, \cdots, \mathsf{buff}_j$ for some $0 \le j \le \tau$ and possibly re-encodes these buffers into buff_{j+1}. Additionally, the update algorithm sets buffers $\mathsf{buff}_j, \cdots, \mathsf{buff}_0$ to empty. In certain cases, the update algorithm might re-write buffer buff^*. Note that if buff_{j+1} was written/re-encoded, then all buffers buff_j through buff_0 were also re-encoded. Similarly, if buff^* was re-encoded, then all buffers buff_τ through buff_0 were also re-encoded. Now, since $\mathsf{BHdis}(\hat{c}_{m_t}, c_{m_t}) \le \delta n$, it follows that all the buffers that were decoded by the update algorithm, decoded correctly and these buffers were then re-encoded without any errors. Hence, for all these buffers $0 \le h \le j+1$ in $\hat{c}_{m_{t+1}}$, $\mathsf{Hamm}(\hat{\psi_h}, \psi_h) \le \delta|\psi_h|$. For buffers that were not touched, since no change was made to these buffers, we still have that $\mathsf{Hamm}(\hat{\psi_h}, \psi_h) \le \delta|\psi_h|$ (for $h > j+1$ and for ψ^*). From these, it follows that $\mathsf{BHdis}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \le \delta n$.

This proves the update correctness as well as the update locality of our update algorithm. This completes the proof of Lemma 1. □

Lemma 2. *Let $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$ be the above described $(k, n, w, r, \epsilon, \delta)-\mathsf{LULDC}$ code for the Buffered-Hamming metric. Then $\mathcal{C}$ is a $(k, n, w, r, \epsilon, \frac{\delta}{2}) - \mathsf{LULDC}$ code for the Prefix Hamming metric.*

Proof. Note that in our code construction, during a write/update operation, we never change the bits of the codeword in a buffer buff_i without changing the bits of the codeword in a buffer buff_j for any $j < i$. Furthermore, even when we change the bits of the codeword in a buffer buff_i, we then change the bits of the codeword in buffers $\mathsf{buff}_{i-1}, \cdots, \mathsf{buff}_0$ in that order. This means that if we consider the age-ordering of c_m, denoted by $\mathsf{w}_1, \cdots, \mathsf{w}_n$, then the indices corresponding to a buffer buff_j will always precede indices corresponding to a buffer buff_i, for any $i > j$. Now, since every buffer buff_{i+1} is twice the size of buffer buff_i, it follows that if two codewords c_m and $\hat{c}_m$ are such that $\mathsf{Prefix}(c_m, \hat{c}_m) \le \frac{\delta n}{2}$, then $\mathsf{BHdis}(c_m, \hat{c}_m) \le \delta n$, which gives us our result. □

The proof of Theorem 1 now follows by simply combining Lemmas 1 and 2.

4 Computational setting

4.1 Codes for computationally bounded adversaries

In the previous section, we showed how to construct LULDC codes for the Prefix-Hamming metric. As noted before, we cannot construct LULDCs for metrics where the adverary can arbitrarily corrupt a constant fraction of the bits of the codeword. Since it is impossible to construct codes for the case of arbitrary adversarial errors, one could consider a setting where the decode algorithm will either decode to the correct message or *detect* if it is not able to do so; in other words, the decode algorithm will never output an incorrect message. Here too, it is easy to see that, unfortunately, one cannot have such information-theoretic error correcting codes. However, we show that by moving to the computationally-bounded adversarial setting, and by allowing the encoder/decoder to maintain a secret state S, one can construct error correcting codes with optimal rate that are locally updatable. Our code will provide the following guarantees:

- If the Prefix Hamming condition is satisfied, then every bit of the message will be locally decodable.
- Additionally, the (local) decoding algorithm will *never* output an incorrect bit of the message.

These guarantees allow us to achieve a tradeoff between *detecting* arbitrary adversarial errors and *decoding* a smaller class of errors. We will provide such a guarantee even when the adversary gets to observe the history of updates/writes made to the codeword; we denote the history of updates/writes made by hist[6].

We now define such locally updatable locally decodable-detectable error correcting codes (LULDDC). As before, we provide our definition for the binary case, but this can be generalized to codes for larger alphabet Σ. Let λ be the security parameter and $\mathsf{neg}(\lambda)$ denote a function that is negligible in λ. We begin with the definition of the Prefix Hamming metric for the computational setting.

Definition 6 (The Computational Prefix Hamming Metric). *Let $\mathsf{E} \in \{0,1\}^{r}$[7]. Let c be of the form $\mathsf{E}_1, \ldots, \mathsf{E}_n$. Let $\mathsf{w}_1, \cdots, \mathsf{w}_n$ denote the age-ordering of c. For some c' of the form $\mathsf{E}_1, \ldots, \mathsf{E}_n$ and for $1 \le t \le n$, let $c'[1,t]$ denote the elements $c'(\mathsf{w}_1), \cdots, c'(\mathsf{w}_t)$. We say that the* Computational Prefix Hamming[8] *distance between c and c', denoted by $\mathsf{Prefix}^{\mathsf{comp}}(c, c')$, is $\le \delta n$ if for all $1 \le t \le n$, $\mathsf{Hamm}(c[1,t], c'[1,t]) \le \delta t$, where $\mathsf{Hamm}(x, y)$ denotes the Hamming Distance between any elements x and y.*

Definition 7 (Locally updatable and locally decodable-detectable codes for adversarial errors (LULDDC)). *A binary code $\mathcal{C} : \{0,1\}^k \to \{0,1\}^n$ is $(k, n, w, r, \lambda, \mathsf{S})$*-locally updatable and locally decodable/detectable *if there exist randomized algorithms $\mathcal{U}$ and $\mathcal{D}$ such that the following conditions are satisfied:*

1. Local Updatability:
   (a) *Let the state be initialized to S_0. Let $m_0 \in \{0,1\}^k$ and let $c_{m_0} = \mathcal{E}(m_0, \mathsf{S}_0)$. Let m_t be a message obtained by any (potentially empty) sequence of updates. (Note that the state S is updated everytime an update is made.) Let hist contain the entire history of updates made on potentially corrupted codewords. Let $\hat{c}_{m_t}$ be the final codeword obtained.*
   Then $\forall m_0 \in \{0,1\}^k, \forall t, \forall m_t, \forall i \in [k], \forall b \in \{0,1\}$, for all probabilistic polynomial time (PPT) algorithms $\mathcal{A}$, for all hist and for all $\hat{c}_{m_t} \in \{0,1\}^n$ output by $\mathcal{A}(m_t, i, b, \mathsf{hist})$, the following condition holds with all but a negligible probability:

[6]While this is the same guarantee that we provide even in the information-theoretic setting, we make this explicit here as we wish to endow the computationally bounded adversary with as much power as possible.

[7]We will think of E as a bit b_i followed by its constant sized authentication tag $\sigma_i = \mathsf{MAC}(b_i)$.

[8]While the definition of the distance function is not computational, we call it the computational prefix hamming distance, as this distance function is used only for the computational LULDDC construction. In our LULDDC codes, security guarantees will hold for codeword corruptions made by computationally bounded adversaries.

- *If* $\exists c_{m_t} \in \mathcal{C}_{m_t}$ *such that* $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$*, then the actions of* $\mathcal{U}^{\hat{c}_{m_t}}(i, b, \mathsf{S}_t)$*, change* $\hat{c}_{m_t}$ *to* $u(\hat{c}_{m_t}, i, b, \mathsf{S}_t) \in \{0,1\}^n$*, where* $\mathsf{Prefix}^{\mathsf{comp}}(u(\hat{c}_{m_t}, i, b, \mathsf{S}_t), c_{m_{t+1}}) \leq \delta n$ *for some* $c_{m_{t+1}} \in \mathcal{C}_{m_{t+1}}$*, where* m_{t+1} *and* m_t *are identical except (possibly) at the* i^{th} *position, and* $m_{t+1}(i) = b$.

(b) *The total number of queries and changes that* $\mathcal{U}$ *makes to the bits of* $\hat{c}_{m_t}$ *is at most* w.

2. Local Decodabilty-Detectability:

(a) *Let* $m_t \in \{0,1\}^k$ *denote the latest message, as determined by* hist*. Then* $\forall \mathsf{hist}, \forall m_t \in \{0,1\}^k, \forall i \in [k]$*, for all probabilistic polynomial time (PPT) algorithms* $\mathcal{A}$ *and for all* $\hat{c}_{m_t} \in \{0,1\}^n$ *output by* $\mathcal{A}(m_t, i, \mathsf{hist})$*:*

- *If* $\exists c_{m_t} \in \mathcal{C}_{m_t}$ *such that* $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$*, then*

$$\Pr[\mathcal{D}^{\hat{c}_m}(i, \mathsf{S}) = m(i)] = 1 - \mathsf{neg}(\lambda),$$

where the probability is taken over the random coin tosses of the algorithm $\mathcal{D}$ *and randomness used to generate* S.

- *If* $\forall c_{m_t} \in \mathcal{C}_{m_t}, \mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_m, c_m) > \delta n$*, then*

$$\Pr[\mathcal{D}^{\hat{c}_m}(i, \mathsf{S}) = m(i) \text{ or } \perp] = 1 - \mathsf{neg}(\lambda),$$

where the probability is taken over the random coin tosses of the algorithm $\mathcal{D}$ *and randomness used to generate* S.

(b) $\mathcal{D}$ *makes at most* r *queries to* $\hat{c}_{m_t}$.

4.2 Our Results

In this section, we present a construction of a LULDDC in the computational setting. In particular, we show:

Theorem 2. *There exists a* $(k, n, w, r, \lambda, \mathsf{S})$ *locally updatable and locally decodable-detectable error correcting code* $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$*, for the Computational Prefix Hamming metric, achieving the following parameters, for some constant* $0 < \delta < \frac{1}{4}$*:*

- ***Length of the code*** (n)***:*** $n = \mathcal{O}(k)$.
- ***Update locality*** (w)***:*** $w = \mathcal{O}(\log^2 k)$*, in the amortized sense.*
- ***Read locality*** (r)***:*** $r = \mathcal{O}(\lambda \log^2 k)$*, in the worst case.*

Similar to the information-theoretic consturction, we use a *heirarchical data structure* to store our codewords. In addition, we use cuckoo hashing and private key locally decodable codes, both of which are reviewed in Appendix A.1.

LULDDC Overview. We start by recalling the construction of the information-theoretic LULDC code from Section 3.2. Recall that codewords had τ buffers. Each buff_j encoded 2^j ($\mathsf{address}$, value) pairs, stored in a sorted manner. We performed a binary search to search for a particular address, a within buff_j. The first difference is that we now use computational locally decodable codes to encode each buffer. (Such codes were introduced by [21]. In this work, we use the construction due to [12].) The next difference in the secret key setting is that we optimize the search performed on the buffers by using cuckoo hash functions[9].

[9]Cuckoo hash functions were first used in conjunction with the hierarchical data structure [19],[20] by Pinkas and Reinman [23] to obtain an ORAM construction. While it was shown that this construction does not hide the access pattern (i.e., which elements were read/written) [9],[17], as we will see, the underlying data structure coupled with cuckoo hashing can still be used securely to obtain a LULDDC code.

In particular, an element (a, v) is inserted at location $h_{\ell,1}(a)$ or $h_{\ell,2}(a)$. To search for an address a in a particular buffer buff_ℓ, our decode algorithm only needs to read locations $h_{\ell,1}(a)$ and $h_{\ell,2}(a)$. (Of course, as in the information-theoretic case, we don't store the buffers in the clear. Rather we store an encoding of the buffers, now computed using the codes of [12] and the locations, $h_{\ell,1}(a)$ and $h_{\ell,2}(a)$, are read via calls to the underlying decode algorithm.) The second difference from the information theoretic construction is that we now use message authentication codes to *detect* a scenario where the codeword has too many errors. (To ensure local decodability, we need to authenticate each bit of the codeword separately.) This guarantees that our computational LULDDC code never decodes to an incorrect message.

Optimizing Parameters. While the above approach does give us an LULDDC construction, it doesn't give us our desired parameters. In particular, message authentication tags need to be of length at least λ, causing a blow-up of at least λ in the parameters. To avoid this, we use constant-size MACs instead.

Constant-size Message Authentication Codes. Such message authentication codes (MAC) authenticate each bit of the message being authenticated (in this case, the codeword) with a tag of length $\mathcal{O}(1)$. While, individually, such MACs can be forged with constant probability, as we will see in our construction, they can be made secure when we are checking $\omega(\lambda)$ MAC values at a time.

At a high-level our decode algorithm will work as follows: we check the authenticity of λ randomly chosen bits of the codeword in each buffer. If most of the tags verify, we get a guarantee that less than a certain constant fraction of the bits of the codeword are corrupted. (Indeed, since each tag is computed with an independent MAC key, the odds that an adversary forges λ tags on his own, is negligible.) This, in turn, ensures that less than a constant fraction of bits of each codeword are corrupted, except with negligible probability[10], and therefore the codeword will decode correctly. (To the best of our knowledge, the idea of combining constant sized MACs with error correcting codes in such a way, was first used in the context of optimizing privacy amplification protocols in [4].) This combined with certain other ideas, give us the construction with parameters stated in Theorem 2. The final construction is described in Appendix A and the proof of Theorem 2 is presented in Appendix A.3.

5 Dynamic Proof of Retrievability

A proof of retrievability scheme enables a client, storing his data on an untrusted server, to execute an audit protocol such that a malicious server that deletes or changes even a single bit of the client's data will fail to pass the audit protocol, except with negligible probability in the security parameter. Proofs of retrievability, introduced by Juels and Kaliski [14], were initially defined on static data building upon the closely related notion of sublinear authenticators defined by Naor and Rothblum [18]. The work of Cash, Küpçü, and Wichs [3] considers this notion for dynamically changing data; in other words, they constructed a proof of retrievability scheme that allowed for efficient updates to the data. Cash *et al.* showed how to convert any oblivious RAM (ORAM) protocol that satisfied a special property (which they define to be next-read-pattern-hiding (NRPH)) to construct a dynamic proof of retrievability (DPoR) scheme. We show that the techniques used to construct LULDDCs can be used to build a DPoR scheme. In addition to being conceptually simple, our construction also significantly improves the parameters achieved by [3].

At a high-level, our construction follows the same approach as our LULDDC scheme. One main difference is that in addition to storing encoded messages in buff_0 to buff_τ and buff^*, we will store the decoded, authenticated, message of every buffer in another set of $\tau + 2$ buffers. The read algorithm works by reading these buffers (instead of the encoded buffers) and verifying their respective MACs. The write algorithm works the same as before – except that it writes to both encoded and unencoded buffers. The

[10]This condition remains true only if all the buffers contain codewords that are at least λ-bits long. We will ensure this by starting our buffers only at a particular level.

audit algorithm works by checking λ randomly chosen locations of each of the encoded buffers and verifying their MACs. Additionally, to obtain good write complexity, we use linear time encodable and decodable standard error correcting codes [28] to encode each buffer, as opposed to using locally decodable codes.

We use these and a few other ideas to ensure that the storage on the server's side is $\mathcal{O}(k)$. The complexity of the PWrite protocol is $\mathcal{O}(\log^2 k)$, similar to the complexity of the update algorithm of our LULDDC. The complexity of the PRead protocol is simply $\mathcal{O}(\lambda \log k)$ as we need to read a constant number of elements in each buffer (along with their MACs of length λ). Finally, the complexity of the Audit protocol is $\mathcal{O}(\lambda \log k)$ as we read λ elements of the codeword in each buffer, along with their constant-size MACs. The client storage is $\mathcal{O}(\lambda)$. We present the details of our DPoR scheme in Appendix B.

5.1 Dynamic PoR

A dynamic PoR scheme [3] comprises of four protocols $\mathsf{PInit}, \mathsf{PRead}, \mathsf{PWrite}$, and Audit between two stateful parties: the client $\mathcal{C}$ and a server $\mathcal{S}$ who is untrusted. The client stores some data m with the server and wishes to perform read, write, and audit operations on this data. In detail, the corresponding protocols are:

- $\mathsf{PInit}(1^\lambda, \Sigma, k)$: In this protocol, the client initializes an empty data storage on the server of length k, where each element in the data comes from an alphabet Σ. The security parameter is λ.
- $\mathsf{PRead}(i)$: In this protocol, the client reads the i^{th} location of the data and outputs some value v_i at the end of the protocol.
- $\mathsf{PWrite}(i, v_i)$: In this, the client sets the i^{th} location of the data to v_i.
- $\mathsf{Audit}()$: In this protocol, the client verifies that the server is maintaining the data correctly so that they remain retrievable. The client outputs either `accept` or `reject`.

The (private) state of the client is implicitly assumed in all the above protocols and the client may also output `reject` during any of the protocols if it detects any malicious behavior on the part of the server. A dynamic PoR scheme must satisfy three properties: *correctness, authenticity, and retrievability.* We refer the reader to Appendix B.1 for the formal definitions of these properties.

5.2 Construction

We now describe our construction of a dynamic PoR scheme. We first note that although an LULDDC is very similar to the notion of a dynamic PoR, we do not use the construction of our LULDDC directly to obtain a dynamic PoR. The reason is that, the LULDDC does not (by itself) support an efficient audit mechanism; on the other hand, an LULDDC satisfies an additional property that corrupted codewords decode as long as the Prefix Hamming condition is satisfied. This leads to a slightly less efficient construction for LULDDCs. Now, we show that we can use ideas developed in the construction of LULDDCs to obtain a dynamic PoR scheme. As is in the works of dynamic PoR [3], we work over an alphabet Σ and all elements that are stored on the server are elements of the alphabet. Our construction of dynamic PoR is very similar to our construction of LULDDCs. We present the construction here and defer the proof to Appendix B.

As before, the client will store τ buffers buff_0 to buff_τ along with a special buffer buff^*. A difference between LULDDCs and our dynamic PoR construction is that we will make use of a standard error correcting code (as opposed to a locally decodable error correcting code) to encode elements stored in each buffer; however, we will use codes that are linear time encodable and decodable (in order to minimize the computational complexity of our construction). Such codes were constructed in the work of Spielman [28]. We will denote such an error correcting code with the encoding algorithm $\mathcal{E}_{\mathsf{lin}}$ and $\mathcal{D}_{\mathsf{lin}}$. Another difference is that, in addition to storing encoded messages in buff_0 to buff_τ and buff^*, we will store the decoded, authenticated, message of every buffer in another set of $\tau+2$ buffers; call these buffers plain_0 to plain_τ and plain^*. Finally, we shall use two types of message authentication codes: to MAC the elements of buffers buff_0 to buff_τ and buff^* (that store codewords), we shall use constant size MACs; however, to MAC the elements of buffers plain_0 to plain_τ (that store elements of the message in the clear), we shall use MACs with MAC length λ. We shall abuse notation and denote both these MACs by MAC (it will be clear from context which type of MAC we use).

- $\mathsf{PInit}(1^\lambda, \Sigma, k)$: This protocol is very similar to the Encode algorithm of our LULDDC. Namely, when storing data $m = m(1), \cdots, m(k) = \mu^*$ on the server, with $m(i) \in \Sigma$, the client computes $\psi^* = \mathcal{E}_{\mathsf{lin}}(\mu^*)$ and $\eta^* = \{(\psi^*(j)||\sigma^*(j))\}$, where $\psi^*(j)$ is the j^{th} element of ψ^* and $\sigma^*(j) = \mathsf{MAC}(\psi^*(j))$. The client stores η^* in buff^*. Additionally the client will also store every element of m along with its MAC in plain^*[11].

- $\mathsf{PWrite}(i, v_i)$: To write element v_i into position i, $\mathcal{C}$ does as follows:

  - If the first buffer is non-empty, find the first empty buffer – this can be determined using ctr, but for now, we just assume that we learn this by decoding buffers in a top-down manner and scanning them to see if they contain any non-empty element. Let the first empty buffer be at level j.
  - Update S to S' so that it now contains an incremented counter.
  - We store (i, b_i) as well as all the non-empty elements from μ_0 to μ_{j-1} into μ_j. To do this, we decode $\psi_0 \cdots \psi_{j-1}$, insert the elements into μ_j and then compute $\mathcal{E}_{\mathsf{lin}}(\mu_j)$ to obtain ψ_j. We compute $\eta_j(\ell) = \{\psi_j(\ell), \sigma_j(\ell)\}$. (The authentication tags $\sigma_j(\ell)$ are recomputed with the *latest key* corresponding to level j, which in turn is computed from S').
  - Additionally, we store the plain message μ_j in plain_j. Note, that whenever reading an element, we read the element along with its MAC and reject if the MAC does not verify.
  - The buffers from $\mathsf{buff}_{j-1} \ldots \mathsf{buff}_0$, as well as $\mathsf{plain}_{j-1} \ldots \mathsf{plain}_0$, are now set to empty by writing special elements into it (along with appropriate MAC values).

- $\mathsf{PRead}(i)$: To read the i^{th} element of the most recent message stored on the server, the client does the following:

  - The algorithm starts with the top-most buffer (plain_0) and proceeds downwards until it finds the address i.
  - Note that plain_j contains μ_j in plaintext. To search a buffer buff_j for an index i, we read the locations $h_{j,1}(i)$ and $h_{j,2}(i)$. If either of these locations contains an entry (i, v) then v is the output of the algorithm.
  - If we reach the last buffer, plain^*, we read the element v stored at address i in plain^*. If the tag σ does not verify, for any element read (in any of the buffers), then the algorithm outputs `reject`, otherwise v is the output[12].

- $\mathsf{Audit}()$: The audit protocol works as follows:

[11]In order to reduce the storage complexity, every $\frac{\lambda}{|\Sigma|}$ elements are grouped together and MACed so that the storage complexity remains at $\mathcal{O}(k)$ and does not become $\mathcal{O}(k\lambda)$.

[12]Note, that because of the way we MAC the plaintext values in plain buffers, when we read a single element from plain, we may have to read an additional $\frac{\lambda}{|\Sigma|}$ elements in order to verify the MAC; we ignore this in the description for ease of exposition.

– For every buffer buff_0 to buff_τ as well as buff^*, pick λ locations of the codeword ψ_j (stored in buff_j) at random and read these λ elements along with their MAC values.
– If all the MACs verify, then output `accept`, otherwise output `reject`.

Acknowledgments. We thank the anonymous reviewers of TCC 2014 for their very valuable feedback.

References

[1] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference*, pages 319–333, 2009.

[2] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In *Proceedings of the first ACM Cloud Computing Security Workshop, CCSW 2009*, pages 43–54, 2009.

[3] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference*, pages 279–295, 2013.

[4] N. Chandran, B. Kanukurthi, R. Ostrovsky, and L. Reyzin. Privacy amplification with asymptotically optimal entropy loss. In *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 785–794, 2010.

[5] Y. M. Chee, T. Feng, S. Ling, H. Wang, and L. F. Zhang. Query-efficient locally decodable codes of subexponential length. *Computational Complexity*, 22(1):159–189, 2013.

[6] Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography, 6th Theory of Cryptography Conference, TCC 2009*, pages 109–127, 2009.

[7] K. Efremenko. 3-query locally decodable codes of subexponential length. In *STOC, Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, pages 39–44, 2009.

[8] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In *STOC*, pages 182–194, 1987.

[9] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In *ICALP (2)*, pages 576–587, 2011.

[10] A. Guo, S. Kopparty, and M. Sudan. New affine-invariant codes from lifting. In *ITCS, Innovations in Theoretical Computer Science*, pages 529–540, 2013.

[11] B. Hemenway and R. Ostrovsky. Public-key locally-decodable codes. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference*, pages 126–143, 2008.

[12] B. Hemenway, R. Ostrovsky, M. J. Strauss, and M. Wootters. Public key locally decodable codes with short keys. In *14th International Workshop, APPROX 2011*, pages 605–615, 2011.

[13] B. Hemenway, R. Ostrovsky, and M. Wootters. Local correctability of expander codes. In *ICALP (1)*, pages 540–551, 2013.

[14] A. Juels and B. Kaliski. Pors: proofs of retrievability for large files. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security*, pages 584–597, 2007.

[15] J. Katz and L. Trevisan. On the efficiency of local decoding procedures for error-correcting codes. In *STOC, Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 80–86, 2000.

[16] S. Kopparty, S. Saraf, and S. Yekhanin. High-rate codes with sublinear-time decoding. In *STOC*, pages 167–176, 2011.

[17] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, pages 143–156, 2012.

[18] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005)*, pages 573–584, 2005.

[19] R. Ostrovsky. An efficient software protection scheme. In *CRYPTO*, pages 610–611, 1989.

[20] R. Ostrovsky. Efficient computation on oblivious rams. In H. Ortiz, editor, *STOC*, pages 514–523. ACM, 1990.

[21] R. Ostrovsky, O. Pandey, and A. Sahai. Private locally decodable codes. In *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007*, pages 387–398, 2007.

[22] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[23] B. Pinkas and T. Reinman. Oblivious ram revisited. In T. Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2010.

[24] L. J. Schulman. Communication on noisy channels: A coding theorem for computation. In *33rd Annual Symposium on Foundations of Computer Science, FOCS*, pages 724–733, 1992.

[25] L. J. Schulman. Deterministic coding for interactive communication. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing, STOC*, pages 747–756, 1993.

[26] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology - ASIACRYPT 2008, 14th International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.

[27] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM Conference on Computer and Communications Security*, pages 325–336, 2013.

[28] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing, STOC*, pages 388–397, 1995.

[29] S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego*.

[30] S. Yekhanin. Locally decodable codes. *Foundations and Trends in Theoretical Computer Science*, 6(3):139–255, 2012.

A Computational LULDDC Construction

A.1 Building blocks

Cuckoo Hashing. In this technique, introduced by Pagh and Rodler [22], there are two hash functions (h_1, h_2) with associated hash tables (T_1, T_2). An element x is located at either $h_1(x)$ (in table T_1) or at $h_2(x)$ (in T_2). To insert an element x, it is inserted in its first location ($h_1(x)$), kicking out the element previously there. This displaced element is moved into its other location, possibly displacing another element. This process continues until no element is kicked out or it runs too long (i.e., more than $\mathsf{c} \log n$ steps for an appropriate constant c). In the latter case (which is referred to as a *failure*), new hash functions are chosen and the entire table is rehashed. Informally, the lemma from [22] states that if $m = (1+\varepsilon)n$ (where m is the size of the hash tables), the probability that the insertion of a new key causes failure (after n items have been inserted) is $\Theta(\frac{1}{n^2})$.

Private-key Locally Decodable Codes. Locally decodable codes in the computational setting were introduced in the work of Ostrovsky, Pandey, and Sahai [21] who constructed an encryption scheme where every bit of the message could be decrypted "locally" even when a fraction of the bits of the ciphertext were corrupted by a computationally bounded adversary. We refer the reader to [21, Definition 4] for the formal definition. In the public-key setting, Hemenway and Ostrovsky [11] constructed the first public key locally decodable codes and the best known public key locally decodable codes were given in the work of Hemenway *et al.* [12]. Of course, such codes are also private key locally decodable codes and further more these codes can be constructed from any semantically secure encryption scheme (namely from one-way functions in the private key setting). More formally, the theorem in [12] is:

Theorem 3 ([12] (restated))**.** *Assume the existence of a semantically secure encryption scheme. Then, there exists a private-key $(k, n, r, \zeta, \epsilon)$-locally decodable error correcting code with $n = \mathcal{O}(k)$, $r = \mathcal{O}(\lambda)$, $\zeta = \mathcal{O}(1)$, $\epsilon = \mathsf{neg}(\lambda)$, where neg is a function that is negligible in the security parameter λ. Furthermore, the size of the secret key of this code is $\mathcal{O}(\lambda)$.*

A.2 Our LULDDC Construction.

We now build our code (denoted $\mathcal{C}^{\mathsf{comp}}$) in the secret key setting. The secret state S consists of a counter ctr (that is incremented everytime an update takes place), and a key to a PRF. S is used to generate the various keys used by the code. Similar to the information-theoretic case, each codeword c of $\mathcal{C}^{\mathsf{comp}}$ consists of $\tau+1$ buffers, $\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau$, where $\tau = \log\left(\frac{k}{\log k}\right)$. In addition, there is a special buffer, buff^*, which has a structure different from the other buffers.

μ_i contains $(1+\gamma)2^i$ cells (for some $\gamma > 1$) – each being either a "non-empty" cell containing a $(\mathsf{address}, \mathsf{value})$-pair or an "empty" cell containing a special symbol π. There are at most 2^i *non-empty* elements in μ_i at any point of time, and these elements are stored using cuckoo hash functions $(h_{i,1}, h_{i,2})$. The remaining locations of μ_i are filled with empty elements. We let $\psi_i = \mathcal{E}_{\mathsf{LDC}}(\mu_i)$. For each bit j of ψ_i, let $\sigma_i(j) = \mathsf{MAC}(\psi_i(j))$. Set $\eta_i = \{(\psi_i(j)||\sigma_i(j))\}$. buff_i contains η_i. μ^* contains all the bits of m in order (without the address values). $\psi^* = \mathcal{E}_{\mathsf{LDC}}(\mu^*)$ and $\eta^* = \{(\psi^*(j)||\sigma^*(j))\}$. The codeword is $c_m = [\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau, \mathsf{buff}^*]$. Let α be a constant. We will pick α (as a function of δ and ζ) later on appropriately.

Encode algorithm. Our encoding algorithm works as follows:

Algorithm $\mathcal{E}(m, \mathsf{S})$:

1. Let $\mu^* = m(1), \cdots, m(k)$, where $m(i)$ denotes the i^{th} bit of the message. Let $\psi^* = \mathcal{E}_{\mathsf{LDC}}(\mu^*)$ and $\eta^* = \{(\psi^*(j)||\sigma^*(j))\}$, where $\psi^*(j)$ is the j^{th} bit of ψ^* and $\sigma^*(j) = \mathsf{MAC}(\psi^*(j))$.
2. Creates the $\tau+1$ *empty* buffers $(\mathsf{buff}_\tau, \ldots, \mathsf{buff}_0)$ in that order; i.e., the underlying μ_i contains only special symbols.

Local Update Algorithm. The update algorithm takes as input a (potentially corrupted) codeword $\hat{c}$, an index i, a bit b_i, and the latest state S. Let the latest value of the message, as determined by hist, be m. Then if there exists some codeword c_m such that $c \in \mathcal{C}_m$ and $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}, c) \leq \delta n$, then the update algorithm outputs $\hat{c}'$ where $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}', c') \leq \delta n$ such that $c' \in \mathcal{C}_{m'}$ and m' and m are identical except possibly at the i^{th} position, where $m'(i) = b_i$.

Recall that each codeword has multiple buffers of the form $\psi_i(j)||\sigma_i(j)$ where $\psi_i(j)$ is one bit of the codeword and $\sigma_i(j)$ is its constant sized message authentication tag. We refer to each of these $\psi_i(j)||\sigma_i(j)$ as an element of buff_i.

Algorithm $\mathcal{U}^{\mathfrak{C}_m}(i, b, \mathsf{S})$:

1. If the first buffer is empty, compute $\psi = \mathcal{E}_{\mathsf{LDC}}(i||b)$; $\sigma = \mathsf{MAC}(\psi)$ and insert $\eta = (\psi||\sigma)$ into the first buffer.
2. If the first buffer is non-empty, find the first empty buffer – note this can be determined easily from ctr. Let the first empty buffer be at level j.
3. Store (i, b_i) as well as all the non-empty elements from μ_0 to μ_{j-1} into μ_j. To do this, we decode $\psi_0, \cdots, \psi_{j-1}$, insert the elements into μ_j and then compute $\mathcal{E}_{\mathsf{LDC}}(\mu_j)$ to obtain ψ_j. We compute $\eta_j(\ell) = \{\psi_j(\ell), \sigma_j(\ell)\}$. (The authentication tags $\sigma_j(\ell)$ are recomputed with the *latest key* corresponding to level j.) When decoding $\psi_0, \cdots, \psi_{j-1}$, ensure that at least $(1-\delta)|\psi_j|$ MACs in every buffer verify; otherwise, output $\perp$.
4. Starting from buff_{j-1} up to buff_0, fill each of the buffers with empty elements in order. In other words, set the underlying μ_ℓs for each of the buffers to contain only special symbols.

   Handling Repetitions: Note that if some address a appears in multiple levels, the top-most of the levels has the most recent value corresponding to address a and therefore that is the value stored. We store only the latest value corresponding to each address a in the top-most level that it appears in. To ensure this, we insert elements into μ_j starting with the most recent (namely (i, b)) and proceeding in a top-down manner. We insert an element (a, v, σ) into location $h_{j,1}(a)$, only if neither $h_{j,1}(a)$ nor $h_{j,2}(a)$ already contain an element with address a. (Collisions due to cuckoo-hashing are resolved in the standard way as described above.)

   Optimization: Since the internal state S contains storage proportional to λ, the buffers $\mathsf{buff}_0, \ldots, \mathsf{buff}_{\log(\frac{2\lambda}{\log k})}$, may be stored in the secret state itself and therefore do not need to be authenticated. This will ensure that the length of all $\psi_j(\ell)$s (which are part of the codeword), is at least λ. It is because of the fact that we are always authenticating and verifying messages (here by messages, we mean the codeword ψ_j) of length at least λ, we are able to use constant sized tags in a secure manner.

Local Decode Algorithm. The algorithm for reading the i^{th} bit works as follows:

Algorithm $\mathcal{D}^{\mathfrak{C}_m}(i, \mathsf{S})$:

1. Randomly select λ elements from each of the buffers.
2. For each of the elements, verify that $\sigma(j) = \mathsf{MAC}(\psi(j))$. (Note that this verification is done with appropriate MAC keys generated from S.)
3. If, for even one level, less than $\alpha\lambda$ of the tags verify, then output $\perp$. Else go to the next step.
4. The decode algorithm starts with the top-most buffer (buff_0) and proceeds downwards until it finds the address i.
5. For now, assume that buff_j contains μ_j instead of its encoding. Then to search a buffer buff_j for an index i, we read the locations $h_{j,1}(i)$ and $h_{j,2}(i)$. If either of these locations contains an entry (i, v) then v is the output of the algorithm.

Since buff_j contains $\{\psi_j(\ell), \sigma_j(\ell)\}$, the steps we just described are implemented via calls to the underlying decoder $\mathcal{D}_{\mathsf{LDC}}$.

6. If we reach the last buffer, buff^*, we read the element v stored at address i in the buffer – once again, via calls to $\mathcal{D}_{\mathsf{LDC}}$. v is the output of the algorithm.

A.3 Proof of Theorem 2

We now proceed to show that code described above in Section A.2 is a LULDDC for the computational Prefix Hamming metric. As was the case in the information-theoretic setting, we shall instead show that the described code is a LULDDC for the computational Buffered-Hamming metric. From this, and Lemma 2, the proof of Theorem 2 will follow. The definition of the Computational Buffered-Hamming metric naturally follows from that of the Buffered-Hamming and the Computational Prefix Hamming metrics, but we present it below for the sake of completeness.

Definition 8 (The Computational Buffered-Hamming Metric)**.** *Let* $\mathsf{E} \in \{0,1\}^r$[13]*. Let* $c \in \mathsf{E}^n$ *comprise of buffers* $\mathsf{buff} = \mathsf{buff}_0, \ldots, \mathsf{buff}_q$ *of lengths* $n_0, \ldots, n_q$ *respectively. Let* $c' \in \mathsf{E}^n$ *be another string with buffers* $\mathsf{buff}' = \mathsf{buff}'_0, \ldots, \mathsf{buff}'_q$*. Then, we say that the* Computational Buffered-Hamming *distance between* c *and* c'*, denoted by* $\mathsf{BHdis}^{\mathsf{comp}}(c, c')$*, is* $\leq \delta n$ *if* $\forall i$ $\mathsf{Hamm}(\mathsf{buff}_i, \mathsf{buff}'_i) \leq \delta n_i$*.*

We will first show the following lemma. From this, and Lemma 2, the proof of Theorem 2 will follow.

Lemma 3. *The code described in Section A.2 is a* $(k, n, w, r, \lambda, \mathsf{S})$ *locally updatable and locally decodable-detectable error correcting code* $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$*, for the Computational Buffered-Hamming metric, achieving the following parameters, for some constant* $0 < \delta < \frac{1}{4}$*:*

- ***Length of the code*** (n)***:*** $n = \mathcal{O}(k)$*.*
- ***Update locality*** (w)***:*** $w = \mathcal{O}(\log k)$*, in the worst case.*
- ***Read locality*** (r)***:*** $r = \mathcal{O}(\lambda^2 \log k)$*, in the worst case.*

Proof. **Length of the code.** Recall that we have buffers in levels $0, 1, \ldots, \tau$ where $\tau = \log\left(\frac{k}{\log k}\right)$. Each buffer encodes a message μ_j of length $k_j = 2^j(\log k + 1)$. μ_j is then encoded into ψ_j using the constant rate LDC due to [12]. Each bit of ψ_j is then authenticated with a constant sized MAC. Therefore the length of each buffer buff_j is asymptotically bounded by the length of μ_j. In addition, the code has buffer buff^*. It is easy to see that buff^* has length $\mathcal{O}(k)$ – indeed, μ^* contains just the bits of m in order without the address values. Therefore the length of the code

$$n = \mathcal{O}(k) + \sum_{0 \leq j \leq \tau} \mathcal{O}(k_j) = \mathcal{O}(k).$$

Local Updatability. Our update algorithm takes as input a bit b, an index i and the state S. In addition it has oracle access to $\hat{c}_{m_t}$. If m_t is the latest value of the message, as determined by hist, and if there exists a codeword c_{m_t}, such that $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta k$ and $c_{m_t} \in \mathcal{C}_{m_t}$, then the algorithm outputs $\hat{c}_{m_{t+1}}$ with the following properties:

1. There exists a codeword, $c_{m_{t+1}}$ such that $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \leq \delta n$
2. m_{t+1} and m_t are identical except possibly at the ith position, where $m_{t+1}(i) = b$

[13] Again, think of E as a bit b_i followed by its constant sized authentication tag $\sigma_i = \mathsf{MAC}(b_i)$.

We first analyze the update locality, i.e., we determine the number of bits of $\hat{c}_{m_t}$ accessed in order to make an udpate. It is easy to see that in algorithm $\mathcal{U}^{\hat{c}_m}(i, b, \mathsf{S})$, buffer buff_j $(0 \leq j \leq \tau)$ is rewritten every 2^j steps. Buffer buff^* is re-written every $2^{\tau+1}$ steps. In 2^j updates (when $j < \tau + 1$), therefore, the total number of bits re-written is

$$\begin{aligned} &= 2^j|\mathsf{buff}_0| + 2^{j-1}|\mathsf{buff}_1| + \ldots + 2^0|\mathsf{buff}_j| \\ &= \mathsf{c}_0\left(2^j|\mu_0| + 2^{j-1}|\mu_1| + \ldots + 2^0|\mu_j|\right) \\ &= \mathsf{c}_0 j 2^j|\mu_0| \text{ (since } \mu_i = 2\mu_{i-1}, \forall i). \end{aligned}$$

The penultimate equation follows since we use a constant rate LDC code and a constant sized MAC tags, it follows that $|\mathsf{buff}_\ell| = \mathsf{c}_0(\mu_\ell)$ for some constant c_0.

When $j \geq \tau + 1$, buff^* is re-written and hence in this case, the total number of bits re-written is

$$= \mathsf{c}_0 j 2^j|\mu_0| + \mathsf{c}_1 2^{j-(\tau+1)}|\mu^*|$$

Substituting for $|\mu_0| = \log k$, $|\mu^*| = k$ and $\tau = \log(\frac{k}{\log k})$, we get that the amortized update locality w per update is $\mathcal{O}(\log^2 k)$. Note that, similar to the information-theoretic setting, one can convert the amortized update locality into a worst-case guarantee on the write locality, by distributing the work over the many write operations, giving us a worst case write locality of $\mathcal{O}(\log k)$.

To show update correctness, we must now argue, that if we begin the update algorithm with a corrupted codeword $\hat{c}_{m_t}$, such that $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$ and update the message m_t to m_{t+1} (where m_t and m_{t+1} differ (possibly) only at the i_t^{th} position, where $m_{t+1}(i_t) = b_{t+1}$), then we modify $\hat{c}_{m_t}$ to $\hat{c}_{m_{t+1}}$ where $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \leq \delta n$ for some $c_{m_{t+1}}$ that is a codeword of m_{t+1}. To see this, observe that, the update algorithm decodes all buffers $\mathsf{buff}_0, \cdots, \mathsf{buff}_j$ for some $0 \leq j \leq \tau$ and possibly re-encodes these buffers into buff_{j+1}. Additionally, the update algorithm sets buffers $\mathsf{buff}_j, \cdots, \mathsf{buff}_0$ to empty. Note, that when the update algorithm decodes these buffers, it checks that $(1-\delta)|\mathsf{buff}_h|$ MAC tags verify in each buffer. Note that if $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$, then $\mathsf{Hamm}(\hat{\psi}_h, \psi_h) \leq \delta|\psi_h|$, which in turn implies that at least $(1-\delta)|\mathsf{buff}_h|$ MAC tags verify in each buffer. Hence, the update algorithm will continue with the decoding and decode each buffer. Additionally in certain cases, the update algorithm might re-write buffer buff^*. Note that if buff_{j+1} was written/re-encoded, then all buffers buff_j through buff_0 were also re-encoded. Similarly, if buff^* was re-encoded, then all buffers buff_τ through buff_0 were also re-encoded. Now, since $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$, it follows that all the buffers that were decoded by the update algorithm, decoded correctly and these buffers were then re-encoded without any errors. Hence, for all these buffers $0 \leq h \leq j+1$ in $\hat{c}_{m_{t+1}}$, $\mathsf{Hamm}(\hat{\psi}_h, \psi_h) \leq \delta|\psi_h|$. For buffers that were not touched, since no change was made to these buffers, we still have that $\mathsf{Hamm}(\hat{\psi}_h, \psi_h) \leq \delta|\psi_h|$ (for $h > j+1$ and for ψ^*). From these, it follows that $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \leq \delta n$.

Decodability and Detectability. The decode algorithm has oracle access to a codeword $\hat{c}_m$ and receives as input S as well $i \in [k]$. If $\hat{c}_m$ is close to a codeword c_m where $c_m \in \mathcal{C}_m$ (and m is the latest codeword, as determined by hist), then the output of the decode algorithm should be $m(i)$. We must show two statements: first, we need to show that if $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_m, c_m) \leq \delta n$, then except with negligible probability (in the security parameter λ), our decode algorithm will output $m(i)$; second, we must show that even if $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_m, c_m) > \delta n$, the decode algorithm will either output $m(i)$ or $\perp$, except with negligible probability.

To show these two statements, let the underlying private-key locally decodable code (from [12]), that we use to encode every buffer, tolerate ζ fraction of errors; i.e., this code can locally decode every bit of the buffer so long as at most ζ fraction of the bits of the codeword are corrupted.

- Let us now show the first stament above. First observe that if $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_m, c_m) \leq \delta n$, then the strings $\hat{c}_m$ and c_m are such that for every j^{th} level buffer $\widehat{\mathsf{buff}_j} \in \hat{c}_m$ and $\mathsf{buff}_j \in c_m$ with values $\hat{\psi}_j$ and $\psi_j \in \mathsf{E}^{|\psi_j|}$ respectively, at least $(1-\delta)|\psi_j|$ of the elements of $\hat{\psi}_j$ and ψ_j have the same value. Now, our local decoding algorithm samples λ of these elements at random, and proceeds only if at least $\alpha\lambda$ of the MACs verify. Note that if at least $(1-\delta)|\psi_j|$ of the elements of $\hat{\psi}_j$ and ψ_j match, then except with probability $e^{-\frac{(1-(\alpha+\delta)^2)\lambda}{2(1-\delta)}}$ (which is negligible in λ when $\alpha < 1-\delta$), $\alpha\lambda$ of the elements will match with the correct codeword c_m and hence the MAC values of these elements will verify. Hence, except with negligible probability, $\alpha\lambda$ of the MAC values checked in each of the buffers will verify and therefore, except with negligible probability, the local decode algorithm will proceed to decode the buffers to get the required values (from which the message is obtained). Now, observe, that since since $\delta < \zeta$, and $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_m, c_m) \leq \delta n$, it follows that $\mathsf{Hamm}(\hat{\psi}_j, \psi_j) \leq \zeta|\psi_j|$, which implies that the local decoding algorithm for every buffer will function correctly. All that remains to be shown is that if all the buffers decode to the right value, then the local decode algorithm will indeed output $m(i)$; but, note that this follows from the correctness of our hierarchical data structure, in exactly the same way as the information-theoretic LULDC. This shows the first statement.

- Let us now show the second statement. To show this, observe that if less than $\alpha\lambda$ MACs verify in some buffer, then our local decode algorithm outputs $\perp$, satisfying the required condition for detectability. All that remains to show is that if $\alpha\lambda$ MACs verify in each buffer, then except with negligible probability, $\mathsf{Hamm}(\hat{\psi}_j, \psi_j) \leq \zeta|\psi_j|$, which implies that the local decoding algorithm for every buffer will decode to the right value and the output of our local decode algorithm will indeed be $m(i)$. That is, the local decode algorithm will never output an incorrect message. To see this, we shall prove the contra-positive. Namely, suppose $\mathsf{Hamm}(\hat{\psi}_j, \psi_j) > \zeta|\psi_j|$. We will then show, that except with negligible probability, less than $\alpha\lambda$ of the MACs will verify. First, recall that elements of $\hat{\psi}_j$ and ψ_j comprise of a bit along with their constant size MAC (under the appropriate key). Now, note that when an element of $\hat{\psi}_j$ and the corresponding element of ψ_j do not match, we only care about the case, when the bit value themselves dont match (as otherwise, decoding will still go through successfully). Now, observe, that when the bit value of an element of $\hat{\psi}_j$ and the bit value of the corresponding element of ψ_j are different, there can only be two cases: a) the bit of the element was changed and the corresponding (constant size) MAC of the bit was forged; or b) the bit of the element was changed and the corresponding MAC of the bit does not verify. Let forge denote the number of elements where the bit was changed, but the corresponding MAC of the bit still verifies (i.e., the MAC of the bit was forged). We shall consider 2 separate cases.

  - In case 1, consider the case when $\mathsf{forge} \geq \mathsf{c}_0\lambda$ for some constant c_0. Now, first observe that $\Pr[\mathsf{forge} \geq \mathsf{c}_0\lambda]$ is negligible in the security parameter. This follows from the fact that the security of the MAC implies that any particular MAC can only be forged with $\frac{1}{2^\ell}$ probability (+ negligible in λ), where $\ell \geq 1$ is a constant equal to the output length of the MAC; hence, except with negligible probability, $\mathsf{forge} < \mathsf{c}_0\lambda$.
  - In case 2, consider the opposite case; i.e., when $\mathsf{forge} < \mathsf{c}_0\lambda$. This means that, in a buffer with value, $\hat{\psi}_j$, the number of elements where the MAC does not verify is $\geq \zeta|\hat{\psi}_j| - \mathsf{c}_0\lambda \geq \zeta'|\hat{\psi}_j|$, for appropriately chosen constant $\zeta' \leq \zeta$ (this follows from the fact that $|\hat{\psi}_j| \geq \lambda$, for all j). Since the MAC does not verify in ζ' fraction of the elements, then except with probability $e^{-\frac{(\zeta'+\alpha-1)^2\lambda}{2\zeta'}}$ (which is negligible in λ for appropriately chosen α), we have that the number of MACs checked that do not verify will be $\geq (1-\alpha)\lambda$, which means the local decoding algorithm will output $\perp$, since less than $\alpha\lambda$ MAC values will verify.

This proves that, except with negligible probability, the decode algorithm will either output $m(i)$ or

$\perp$, even if $\mathsf{BHdis}^{\mathsf{comp}}(\hat{c}_m, c_m) > \delta n$.

A note on α: Note, that α and δ must satisfy two conditions: $\alpha + \delta < 1$ as well as $\alpha + \zeta' > 1$. In order to show the existence of such α and δ, first compute ζ'[14]. Now for any α such that $\alpha + \zeta' > 1$, we can compute appropriate δ such that $\alpha + \delta < 1$, and this gives us an appropriate LULDDC with error rate δ.

Let us finally analyze the decode locality. The locality due to the verification of λ MACs of each buffer is $\lambda \log k$. We now measure the locality due to the rest of the decode algorithm. To read an index i, we scan in a top-down manner and we need to read the elements of μ_ℓ stored at each level ℓ. If we reach the buff^*, we simply read the element corresponding to i^{th} location of μ^*. Since μ_ℓ and μ^* are stored as encodings, we need to read these locations via calls the $\mathcal{D}_{\mathsf{LDC}}$ algorithms. Recall that we need to read $2\log k + 1$ bits of μ in each level and the locality of $\mathcal{D}_{\mathsf{LDC}}$ algorithm is λ for reading one bit of the underlying message. Therefore the locality of $\mathcal{D}$ algorithm is $\mathcal{O}(\lambda \log k)$ per buffer. There are $\tau = \log k - \log\log k$ buffers. Therefore the total locality r is $\mathcal{O}(\lambda \log^2 k)$.

This completes the proof of Lemma 3. $\square$

The proof of Theorem 2 now follows from Lemma 3 and Lemma 2.

B Dynamic Proof of Retrievability

In this section, we show how to use our techniques to construct a dynamic proof of retrievability scheme. Informally, a proof of retrievability allows a client to store data on an untrusted server and later on obtain a short proof from the server, that indeed all of the clients data is present on the server. In other words, the client can execute an audit protocol such that any malicious server that deletes or changes even a single bit of the client's data will fail to pass the audit protocol, except with negligible probability in the security parameter. Proofs of retrievability, introduced by Juels and Kaliski [14], were initially defined on static data building upon the closely related notion of sublinear authenticators defined by Naor and Rothblum [18]. Several works have studied the efficiency of such scemes [26, 6, 2, 1] with the work Cash, Küpçü, and Wichs [3] considering the notion of proof of retrievability on dynamically changing data; in other words, they constructed a proof of retrievability scheme that allowed for efficient updates to the data. Cash *et al.* showed how to convert any oblivious RAM (ORAM) protocol that satisfied a special property (which they define to be next-read-pattern-hiding (NRPH)) to construct a dynamic proof of retrievability (DPOR) scheme. Here, we show that we do not need an ORAM scheme with this property and the techniques used to construct LULDDCs can be used to build a DPoR scheme.

B.1 Dynamic PoR

We now give the definition of Dynamic PoR from Cash *et al.* [3]. A dynamic PoR scheme comprises of four protocols $\mathsf{PInit}, \mathsf{PRead}, \mathsf{PWrite}$, and Audit between two stateful parties: the client $\mathcal{C}$ and a server $\mathcal{S}$ who is untrusted. The client stores some data m with the server and wishes to perform read, write, and audit operations on this data. More specifically, the corresponding interactive protocols are:

- $\mathsf{PInit}(1^\lambda, \Sigma, k)$: In this protocol, the client initializes an empty data storage on the server of length k, where each element in the data comes from an alphabet Σ. The security parameter is λ.
- $\mathsf{PRead}(i)$: In this protocol, the client reads the i^{th} location of the data and outputs some value v_i at the end of the protocol.
- $\mathsf{PWrite}(i, v_i)$: In this protocol, the client sets the i^{th} location of the data to be value v_i

[14]Note that ζ' can be obtained as function of ζ, the number of errors tolerated by the underlying LDC from [12] and c_0 above.

- $\mathsf{Audit}()$: In this protocol, the client verifies that the server is maintaining the data correctly so that they remain retrievable. The client outputs either `accept` or `reject`.

The (private) state of the client is implicitly assumed in all the above protocols and the client may also output `reject` during any of the protocols if it detects any malicious behavior on the part of the server. A dynamic PoR scheme must satisfy three properties: *correctness, authenticity, and retrievability.* For the definitions that follow, we say that $P = \{\mathsf{op}_0, \mathsf{op}_1, \cdots, \mathsf{op}_q\}$ is a dynamic PoR *protocol sequence* if $\mathsf{op}_0 = \mathsf{PInit}(1^\lambda, \Sigma, k)$ and, for $j > 0$, $\mathsf{op}_j \in \{\mathsf{PRead}(i), \mathsf{PWrite}(i, v_i), \mathsf{Audit}()\}$ for some index $i \in [k]$ and value $v_i \in \Sigma$.

Correctness. If $\mathcal{C}$ and $\mathcal{S}$ both follow the protocol honestly, then with probability 1 over the randomness of the client:

- For all $i \in [k]$, and any $\mathsf{op}_j = \mathsf{PRead}(i)$, the output of the client is indeed the correct value v_i; i.e., the client outputs whatever it would have, if it stored the data on its own memory and had read the i^{th} position of the data.
- Every execution of $\mathsf{Audit}()$ protocol results in $\mathcal{C}$ outputting `accept`.

Authenticity. Informally, this requires that the client always detects if any protocol message sent by the server deviates from the honest behavior. That is, consider the following game $\mathsf{AuthGame}_{\bar{\mathcal{S}}}(\lambda)$ between a malicious $\bar{\mathcal{S}}$ and a challenger:

- The malicious server specifies a valid protocol sequence $P = \{\mathsf{op}_0, \cdots, \mathsf{op}_q\}$.
- The challenger initializes a copy of the honest client $\mathcal{C}$ and an honest (deterministic) server $\mathcal{S}$. It executes P between $\mathcal{C}$ and $\bar{\mathcal{S}}$ while, in parallel, also passing a copy of every message from $\mathcal{C}$ to the honest server $\mathcal{S}$.
- During this protocol, if at any point of time, the message given by $\bar{\mathcal{S}}$ as a response to the client differs from the response given by $\mathcal{S}$ and $\mathcal{C}$ does not output `reject`, then the adversary wins the game and the game outputs 1. Otherwise, the game outputs 0.

The authenticity requirement states that for all PPT servers $\bar{\mathcal{S}}$, $\Pr[\mathsf{AuthGame}_{\bar{\mathcal{S}}}(\lambda) = 1]$ is negligible in the security parameter λ.

Retrievability. Informally, retrievability states that whenever the malicious server is in a state with a reasonable probability δ of successfully passing an audit, the server must *know* the entire content of the client's data. This is formalized via the existence of an efficient extractor $\mathcal{E}$ that can recover the data m given (black-box) access to the malicious server. Formally, define the game $\mathsf{ExtGame}_{\bar{\mathcal{S}},\mathcal{E}}(\lambda, \mathsf{p})$ between a malicious server $\bar{\mathcal{S}}$, extractor $\mathcal{E}$, and a challenger.

- $\bar{\mathcal{S}}$ specifies a protocol sequence $P = \{\mathsf{op}_0, \cdots, \mathsf{op}_q\}$. Let $m \in \Sigma^k$ be the correct value of data at the end of executing P.
- The challenger initializes a copy of honest client $\mathcal{C}$ and executes P between $\mathcal{C}$ and $\bar{\mathcal{S}}$. Let $\mathcal{C}_{\mathsf{fin}}$ and $\bar{\mathcal{S}}_{\mathsf{fin}}$ be the final states of the client and malicious server at the end of the interaction (this includes all random coins of the malicious server). Define $\mathsf{Succ}(\bar{\mathcal{S}}_{\mathsf{fin}})$ as the probability that an execution of a subsequent $\mathsf{Audit}()$ protocol between $\bar{\mathcal{S}}$ and $\mathcal{C}$ with states $\bar{\mathcal{S}}_{\mathsf{fin}}$ and $\mathcal{C}_{\mathsf{fin}}$ respectively results in the client outputting `accept` (this probability is only over the random coins of the client during this execution).

- Run $m' \leftarrow \mathcal{E}^{\bar{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, k, 1^{\mathsf{p}})$, where $\mathcal{E}$ gets black-box rewinding access to $\bar{\mathcal{S}}$ in its final configuration $\bar{\mathcal{S}}_{\mathsf{fin}}$.
- If $\mathsf{Succ}(\bar{\mathcal{S}}_{\mathsf{fin}}) \geq 1/\mathsf{p}$ and $m' \neq m$, then output 1, else output 0

Retrievability requires that there exists a PPT extractor $\mathcal{E}$ such that, for all PPT malicious servers $\bar{\mathcal{S}}$ and every $\mathsf{p} = \mathsf{p}(\lambda)$, we have $\Pr[\mathsf{ExtGame}_{\bar{\mathcal{S}},\mathcal{E}}(\lambda, \mathsf{p}) = 1] \leq \mathsf{neg}(\lambda)$.

B.2 Construction

We now describe our construction of a dynamic PoR scheme. We first note that although an LULDDC is very similar to the notion of a dynamic PoR, we do not use the construction of our LULDDC directly to obtain a dynamic PoR. The reason is that, the LULDDC does not (by itself) support an efficient audit mechanism; on the other hand, an LULDDC satisfies an additional property that corrupted codewords decode as long as the Prefix Hamming condition is satisfied. This leads to a slightly less efficient construction for LULDDCs. Now, we show that we can use ideas developed in the construction of LULDDCs to obtain a dynamic PoR scheme. As is in the works of dynamic PoR [3], we work over an alphabet Σ and all elements that are stored on the server are elements of the alphabet. Our construction of dynamic PoR is very similar to our construction of LULDDCs described in Section A.2.

As before, the client will store τ buffers buff_0 to buff_τ along with a special buffer buff^*. A difference between LULDDCs and our dynamic PoR construction is that we will make use of a standard error correcting code (as opposed to a locally decodable error correcting code) to encode elements stored in each buffer; however, we will use codes that are linear time encodable and decodable (in order to minimize the computational complexity of our construction). Such codes were constructed in the work of Spielman [28]. We will denote such an error correcting code with the encoding algorithm $\mathcal{E}_{\mathsf{lin}}$ and $\mathcal{D}_{\mathsf{lin}}$. Another difference is that, in addition to storing encoded messages in buff_0 to buff_τ and buff^*, we will store the decoded, authenticated, message of every buffer in another set of $\tau + 2$ buffers; call these buffers plain_0 to plain_τ and plain^*. Finally, we shall use two types of message authentication codes: to MAC the elements of buffers buff_0 to buff_τ and buff^* (that store codewords), we shall use constant size MACs; however, to MAC the elements of buffers plain_0 to plain_τ (that store elements of the message in the clear), we shall use MACs with MAC length λ. We shall abuse notation and denote both these MACs by MAC (it will be clear from context which type of MAC we use).

- $\mathsf{PInit}(1^\lambda, \Sigma, k)$: This protocol is very similar to the Encode algorithm of our LULDDC. Namely, when storing data $m = m(1), \cdots, m(k) = \mu^*$ on the server, with $m(i) \in \Sigma$, the client computes $\psi^* = \mathcal{E}_{\mathsf{lin}}(\mu^*)$ and $\eta^* = \{(\psi^*(j) || \sigma^*(j))\}$, where $\psi^*(j)$ is the j^{th} element of ψ^* and $\sigma^*(j) = \mathsf{MAC}(\psi^*(j))$. The client stores η^* in buff^*. Additionally the client will also store every element of m along with its MAC in plain^*[15].
- $\mathsf{PWrite}(i, v_i)$: To write element v_i into position i, $\mathcal{C}$ does as follows:
  - If the first buffer is non-empty, find the first empty buffer – this can be determined using ctr, but for now, we will just assume that we learn this by decoding buffers in a top-down manner and then scanning them to see if they contain any non-empty element. Let the first empty buffer be at level j.
  - Update S to S' so that it now contains an incremented counter.

[15]In order to reduce the storage complexity, every $\frac{\lambda}{|\Sigma|}$ elements are grouped together and MACed so that the storage complexity remains at $\mathcal{O}(k)$ and does not become $\mathcal{O}(k\lambda)$.

  - We store (i, b_i) as well as all the non-empty elements from μ_0 to μ_{j-1} into μ_j. To do this, we decode $\psi_0 \cdots \psi_{j-1}$, insert the elements into μ_j and then compute $\mathcal{E}_{\mathsf{lin}}(\mu_j)$ to obtain ψ_j. We compute $\eta_j(\ell) = \{\psi_j(\ell), \sigma_j(\ell)\}$. (The authentication tags $\sigma_j(\ell)$ are recomputed with the *latest key* corresponding to level j, which in turn is computed from S').
  - Additionally, we store the plain message μ_j in plain_j. Note, that whenever reading an element, we read the element along with its MAC and reject if the MAC does not verify.
  - The buffers from $\mathsf{buff}_{j-1} \ldots \mathsf{buff}_0$, as well as $\mathsf{plain}_{j-1} \ldots \mathsf{plain}_0$, are now set to empty by writing special elements into it (along with appropriate MAC values).

- $\mathsf{PRead}(i)$: To read the i^{th} element of the most recent message stored on the server, the client does the following:
  - The algorithm starts with the top-most buffer (plain_0) and proceeds downwards until it finds the address i.
  - Note that plain_j contains μ_j in plaintext. To search a buffer buff_j for an index i, we read the locations $h_{j,1}(i)$ and $h_{j,2}(i)$. If either of these locations contains an entry (i, v) then v is the output of the algorithm.
  - If we reach the last buffer, plain^*, we read the element v stored at address i in the buffer. If the tag σ does not verify, for any element read (in any of the buffers), then the algorithm outputs `reject`, otherwise v is the output[16].

- $\mathsf{Audit}()$: The audit protocol works as follows:
  - For every buffer buff_0 to buff_τ as well as buff^*, pick λ locations of the codeword ψ_j (stored in buff_j) at random and read these λ elements along with their MAC values.
  - If all the MAC checks verify, then output `accept`, otherwise output `reject`.

B.3 Correctness, Authenticity, Retrievability, and Complexity

First, observe that the correctness of the $\mathsf{PInit}(1^\lambda, \Sigma, k)$, $\mathsf{PWrite}(i, v_i)$, and $\mathsf{PRead}(i)$ algorithms follow easily from the correctness of the encode, update and decode algorithms of our LULDDC construction. To see the correctness of $\mathsf{Audit}()$, observe that if the codeword is honestly stored by the server, along with all the MAC values, then the client will output `accept` after any $\mathsf{Audit}()$ protocol. The authenticity of the protocol follows easily from the unforgeability of the MAC and correctness of update and decode algorithms of our LULDDC. To see that our protocol satisfies retrievability, observe that each of the buffers buff_0 to buff_τ as well as buff^* simply store encodings of messages stored in plain_0 to plain_τ and plain^*. Note, that the error correcting code (along with MACs) allow for a static proof of retrievability on each buffer. This is because, if we check the authenticity of λ random bits of the codeword and all MACs verify, then except with negligible probability, most of the bits of the codeword must be present on the server (and these must be correct bits of the codeword). This will allow an extractor algorithm to retrieve the contents of the buffer (for a formal proof of this, see [6]). Now, note that if an adversarial server were to pass the Audit protocol with some probability, then the server must pass the individual audit for each buffer with at least the same probability. But the audit protocol for each buffer is a static PoR and it has an extractor algorithm. Hence, the extractor for each of these buffers together gives us an extractor algorithm for all the buffers and hence the current message m.

[16]Note, that because of the way we MAC the plaintext values in plain buffers, when we read a single element from plain, we may have to read an additional $\frac{\lambda}{|\Sigma|}$ elements in order to verify the MAC; we ignore this in the description for ease of exposition.

First, observe that the storage on the server's side is $\mathcal{O}(k)$. Next, note that the complexity of the PWrite protocol is $\mathcal{O}(\log k)$, similar to the complexity of the update algorithm of our LULDDC. The complexity of the PRead protocol is simply $\mathcal{O}(\lambda \log k)$ as we need to read a constant number of elements in each buffer (along with its MAC of length λ)[17]. Finally, the complexity of the Audit protocol is $\mathcal{O}(\lambda \log k)$ as we read λ elements of the codeword in each buffer, along with their constant-szie MAC values. The client storage is $\mathcal{O}(\lambda)$.

[17]Again, because of the way we MAC the plaintext values in plain buffers, when we read a single element from plain, we will have to read an additional $\frac{\lambda}{|\Sigma|}$ elements in order to verify the MAC, but this has the same length as the MAC value (λ) and so our total complexity in each buffer is $\mathcal{O}(\lambda)$.