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). 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 corrupt an arbitrary (constant fraction of) bits of the codeword subject to the constraint that he does not corrupt more than a δ fraction 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).

We first 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 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) by constructing binary LULDDCs for messages in $\{0,1\}^k$. Our codes have constant rate, an update locality of $\mathcal{O}(\lambda \log k)$ 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 show a construction of this primitive with better parameters than existing constructions. In particular, we construct a DPoR scheme with linear storage, $\mathcal{O}(\log k)$ write complexity, and $\mathcal{O}(\lambda \log k)$ read and audit complexity.

1 Introduction

1.1 Codes with locality

Locally Decodable Codes. Locally decodable codes (LDCs), introduced by Katz and Trevisan [14] are a class of error correcting codes, where every bit of the message can be probabilistically decoded by only reading 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 [6, 4].). On the other hand, the best known codes with constant rate, [15, 9, 12], 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 [27].

[*]Email: AT&T Labs – Security Research Center, New York, NY, Email: `nishanth@cs.ucla.edu`.

[†]Department of Computer Science, UCLA, Email: `bhavanak@cs.bu.edu`.

[‡]Department of Computer Science and Mathematics, UCLA, Email: `rafail@cs.ucla.edu`.

Locally Updatable and Locally Decodable Codes. 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. Updatability and error correction are orthogonal goals and indeed a little thought reveals that if an adversary can corrupt some fraction of arbitrary bits of the codeword, then one cannot obtain locally updatable codes with update locality less than the distance of the code. In light of this, we consider a weaker, yet meaningful, adversarial model of corruption. Informally, in our model, bits of the codeword have an "age". An adversary is allowed to corrupt a constant fraction of the bits of the codeword; however, he is allowed to corrupt fewer of the younger/newer bits and is allowed to corrupt many of the older bits. In a bit more detail, 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 again touch the i^{th} bit, this would once again become the youngest bit of the codeword.

We allow the adversary to corrupt arbitrary (constant fraction of) bits in the codeword subject only to the constraint that at no point of time does he corrupt more than a δ fraction of the t youngest bits (for all $1 \le t \le n$). 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; thus, the newer or younger bits are less corruptible than the older bits.

1.2 Our Results and Techniques

Our main results are as follows:

- *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 update locality of $\mathcal{O}(\log^2 k)$ and a read locality of $\mathcal{O}(k^{\epsilon})$ for any constant $\epsilon < 1$. For codes that operate on larger alphabet Σ, with $|\Sigma| \ge \log k$, we can improve the update locality to $\mathcal{O}(\log k)$ (other parameters remaining the same).

Next, we consider the case where the encoder and decoder share a secret state S and the adversary is computationally bounded. In this setting, we additionally 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. In addition to providing this guarantee, we obtain dramatic improvements over the codes constructed in the information-theoretic setting. In particular, we call such codes locally updatable and locally decodable-detectable codes (LULDDCs), and obtain the following result:

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

LULDCs for the Prefix Hamming Metric. The first idea behind our construction in the information-theoretic setting is as follows. We shall make use of the hierarchical data structure introduced by Ostrovsky [18, 19] in the context of oblivious RAMs. Oblivious RAMs [7, 18] allow efficient random access to memory without revealing the access pattern to an adversary that observes the reads and writes made to memory. However, we show that we can make use of the hierarchical data structure as a building block to construct LULDCs. At a high level, the data structure comprises of $\tau + 1$, buffers $\mathsf{buff}_0, \cdots, \mathsf{buff}_\tau$ of increasing size. Buffer buff_i is roughly of size 2^i. Additionally there is a special buffer (buff^*) will comprise of an encoding of the entire message (encoded using an appropriately chosen locally decodable code). 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 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 $(\text{index}, \text{value})$ pairs that are sorted according to the index value. This will enable us to perform a binary search (performing decodes through 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 $(\text{index}, \text{value})$, 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 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.* [11]). 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. To obtain our result, we make a careful use of constant size MACs to verify the correctness of a codeword as well as to decode correctly (except with negligible probability).

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 [13], were initially defined on static data, building upon the closely related notion of sublinear authenticators defined by Naor and Rothblum [17]. Several works have studied the efficiency of such scemes [25, 5, 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.* [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. 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 retrivability with $\mathcal{O}(k)$ server storage, $\mathcal{O}(\lambda)$ client storage, $\mathcal{O}(\lambda \log k)$ read complexity, $\mathcal{O}(\log k)$ write complexity and $\mathcal{O}(\lambda \log k)$ audit complexity.

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. 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 an 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, $\mathcal{O}(\log k)$ write complexity and $\mathcal{O}(\lambda \log k)$ audit complexity[3].

1.3 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 in the computational setting. Finally, we give our construction of a dynamic proof of retrievability scheme in Section 5.

2 Preliminaries

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

[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]Moreover, these parameters include the cost for storage and verification of the MACs.

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 coin tosses 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 only changing 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. We now present the formal definition.

Definition 2 (Locally updatable and locally decodable codes (LULDC))**.** *A binary code* $\mathcal{C} : \{0,1\}^k \to \{0,1\}^n$ *is* $(k, n, w, r, \delta, \epsilon)$-locally updatable and locally decodable *if there exist randomized algorithms* $\mathcal{U}$ *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. 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 coin tosses of the algorithm $\mathcal{D}$*.*

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

We denote $m^{i^{b_i}}$ to be 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. In order to define the correctness of the update algorithm, we shall now define the set of codewords $\mathcal{C}_m$ for every message m.

Definition 3 (The set $\mathcal{C}_m$)**.** *For a message m, if there exists a message $\bar{m}$, codeword $c_{\bar{m}} = \mathcal{E}(\bar{m})$ (possibly $\bar{m} = m$ and $c_{\bar{m}} = c_m$) 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 $c_m = u(....u(u(c_{\bar{m}}, i_1, b_1), i_2, b_2), ..., i_t, b_t)$, then c_m is in the set $\mathcal{C}_m$.*

2.2 The Prefix Hamming Metric

It is clear that one cannot hope to construct locally updatable locally decodable error correcting codes for metrics where an adversary can corrupt an arbitrary (bounded) number of bits of the codeword. For example, if we updated a codeword from c_m to c'_m with a locality of w, then simply by corrupting w of the 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'). If we truly want update locality, then indeed w must be $<< \delta n$ and hence the adversary could indeed corrupt the bits of c'_m in this manner. 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, our decode algorithm still functions correctly. Our metric is similar in spririt to the notion of tree codes introduced by Schulman [23, 24]. At a high level, bits of the codeoword "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. 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. In particular, we define our metric space and its distance function.

Definition 5 (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* $T = (\log k + 1) \left(r_0 + \sum_{1 \le j \le \tau} j r_j \right)$, *in the worst case.*

As a sample corollary to Theorem 1, using the LDCs from the works of [15, 9, 12] 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)***:*** $w = \mathcal{O}(\log^2 k)$.
- ***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 $|\Sigma| \ge \mathsf{c} \log k$ 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 [18, 19] 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 $\mathsf{Prefix}(\hat{c}_m, c_m) \le \delta n$, for some $c_m \in \mathcal{C}_m$) into a string $\hat{c'}_m$, when setting the i^{th} bit of m 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 that 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.)
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 $\mathcal{D}^{\hat{c}_m}(i)$:

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

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 a codeword $c_m \in \{0,1\}^n$ 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 code 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 $0 \leq i \leq \tau$. 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 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): $n = n^* + \sum\limits_{i=0}^{\tau} n_i$.*

- ***Update locality** (w): $w = (\log k + 1)\sum\limits_{i=0}^{\tau} \frac{1}{\rho_i} + \frac{\log k+1}{\rho^*}$, in the worst case.*

- ***Read locality** (r): $r = \frac{8(1-\epsilon)}{(1-2\epsilon)^2} T \log \frac{T}{\epsilon} + r^*$, where $T = (\log k + 1)\left(r_0 + \sum\limits_{1\leq j \leq \tau} j r_j\right)$, 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 then easy to see that the length of the code $n = n^* + \sum\limits_{i=0}^{\tau} n_i$.

Read locality and Decode Correctness. We shall 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 $\mathsf{BHdis}(\hat{c}_m, c_m) \leq \delta n$, 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 now 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 $\geq 1 - \epsilon$.

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 $j \geq 1$), 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 $j \geq 1$). $\mathcal{D}^{\psi}(i)$ 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 $1 \leq j \leq \tau$) 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 the set if $\mu_x(y)$ was accessed; also let $T = |\mathsf{Read}|$. Then

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

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

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 $\mathcal{D}^{\psi'_x}_{\mathsf{LDC}}(y) = \mu(x, y)$. Let $\mathsf{SuccRead}(x, y)$ denote that event that $\#\mathsf{Succ}(x, y) > \frac{\lambda}{2}$.

First, note that 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 $\mathsf{Hamm}(\psi^{*\prime}, \psi^*) \leq \delta|\psi^*|$. Now, since $\mathcal{C}^{\psi'_j}_{\mathsf{LDC}}$ 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

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

[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, $X = \#\mathsf{Succ}(x, y); \mathbf{E}(X) = \lambda(1 - \epsilon); t = \frac{1-2\epsilon}{2-2\epsilon}$

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 decode correctness as well as 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

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

The amortized update locality w per update is

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

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 the many write operations, giving us a worst case write locality of $w = (\log k + 1) \sum_{i=0}^{\tau} \frac{1}{\rho_i} + \frac{\log k+1}{\rho^*}$.

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 $\mathsf{BHdis}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$ 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 $\mathsf{BHdis}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \leq \delta n$ 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. 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 $\mathsf{BHdis}(\hat{c}_{m_t}, c_{m_t}) \leq \delta n$, 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 $\hat{c}_{m_{t+1}}$, $\mathsf{Hamm}(\psi'_h, \psi_h) \leq \delta|\psi_h|$. For buffers that were not touched, since no change was made to these buffers, we still have that $\mathsf{Hamm}(\psi'_h, \psi_h) \leq \delta|\psi_h|$ (for $h > j + 1$ and for ψ^*). From these, it follows that $\mathsf{BHdis}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \leq \delta n$.

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 a $(k, n, w, r, \epsilon, \delta) - \mathsf{LULDC}$ 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 codeword construction, during a write/update operation, we never change the bits of the codeword in a buffer buff_i but not change 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 $\mathsf{buff}_{i-1}, \cdots, \mathsf{buff}_0$. 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 $\mathsf{Prefix}(c_m, \hat{c}_m) \leq \frac{\delta n}{2}$, then $\mathsf{BHdis}(c_m, \hat{c}_m) \leq \delta n$, which gives us our result. $\square$

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 LDC codes for the prefix-condition metric. Ideally we would like to consider the case of adversarial errors, i.e. a setting where an adversary can corrupt some constant fraction of *arbitrary* bits of the entire codeword. It is easy to see that one cannot possibly construct locally updatable locally decodable codes. Indeed, if the adversary can corrupt a δ fraction of the bits of the codeword, and the code has update locality w ($\leq \delta n$), then an adversary could simply corrupt just the bits that were changed when updating a codeword and the codeword will decode to the incorrect message.

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 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:

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

These guarantees allow us to achieve a tradeoff between *detecting* arbitrary adversarial errors and *decoding* a restricted 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 will 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 case of binary codes, but this can be easily generalized to codes for larger alphabet Σ. We let λ be the security parameter. $\mathsf{neg}(\lambda)$ denotes a function that is negligible in the security parameter.

[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.

Definition 6 (Computational Prefix Hamming Metric)**.** *Let* $\mathsf{E} \in \{0,1\}^{r}$[7]*. Let* $c \in \mathsf{E}^n$*. Let* $\mathsf{w}_1, \cdots, \mathsf{w}_n$ *denote the age-ordering of* c*. Let* $c' \in \mathsf{E}^n$ *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 *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 conditions hold with all but a negligible probability:*
     - *If* $\forall c_{m_t} \in \mathcal{C}_{m_t}$*,* $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) > \delta n$*, then* $\mathcal{U}^{\hat{c}_{m_t}}(i,b,\mathsf{S}_t)$ *outputs* $\perp$*.*
     - *If* $\exists c_{m_t} \in \mathcal{C}_{m_t}$ *such that* $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \le \delta n$*, 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* $\mathsf{dis}(u(\hat{c}_{m_t}, i, b, \mathsf{S}_t), c_{m_{t+1}}) \le \delta n$ *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, where* $m_{t+1}(i) = b$*.*
   - *(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* $\forall m_t \in \{0,1\}^k, \forall \mathsf{hist}, \forall i \in [k]$*, 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* $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \le \delta n$*, 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 $\mathcal{D}$*.*
     - *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 $\mathcal{D}$*.*
   - *(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* $\mathcal{C} = (\mathcal{E}, \mathcal{D}, \mathcal{U})$ *achieving the following parameters, for some constant* $0 < \delta < \frac{1}{4}$*:*

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

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

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 we review below.

4.3 Building blocks

Cuckoo Hashing. In this technique, introduced by Pagh and Rodler [21], 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 [21] 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 [20] 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. In the public-key setting, Hemenway and Ostrovsky [10] 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.* [11]. 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 [11] is:

Theorem 3 ([11] (restated))**.** *Assume the existence of a semantically secure encryption scheme. Then, there exists a* $(k, n, r, \delta, \epsilon)$*-locally decodable error code with* $n = \mathcal{O}(k)$*,* $r = \mathcal{O}(\lambda)$*,* $\delta = \mathcal{O}(1)$*,* $\epsilon = \mathsf{neg}(\lambda)$*, where* neg *is a function that is negligible in the security parameter* λ*. Furthermore, the size of the secret key of this code is* $\mathcal{O}(\lambda)$*.*

4.4 Overview of the construction

We start by recalling the construction of an information-theoretic LULDC code from Section 3.2. We had τ buffers. Each buff_j encoded 2^j $(\mathsf{address}, \mathsf{value})$ pairs, stored in a sorted manner. We performed a binary search to search for a particular address, a within buff_j. In the secret key setting, we optimize this by using cuckoo hash functions. 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, only need to read locations $h_{\ell,1}(a)$ and $h_{\ell,2}(a)$. Another 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. This guarantees that our computational *LULDDC* code never decodes to an incorrect message.

Remark 1. *Cuckoo hash functions were first used in conjunction with the hierarchical data structure [18, 19] by Pinkas and Reinman [22] to obtain an ORAM construction. While this construction was shown to be an insecure ORAM [8, 16], the underlying data structure can still be used securely to obtain a LULDDC code.*

4.5 Code Description

We build our code (denoted $\mathcal{C}^{\text{comp}}$) in the secret key setting. The secret state S consists of a counter ctr, which is incremented everytime an update takes place, and a key to a PRF. S is used to generate keys the various keys used by the code. Similar to the information-theoretic case, each codeword c of $\mathcal{C}^{\text{comp}}$ 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 ($\gamma > 1$) – 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) and 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 $c_m = [\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau, \mathsf{buff}^*]$.

Encode algorithm. Our encoding algorithm works as follows:

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

1. Let $\mu^* = m(1), \cdots, m(k)$, where $m(i)$ denotes the ith 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 $(\mathsf{buff}_0, \ldots, \mathsf{buff}_\tau)$ 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 ith position, where $m'(i) = b_i$. If there doesn't exist a c such that $c \in \mathcal{C}_m$ and $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}, c) \leq \delta n$, then $\mathcal{U}(\hat{c}, i, b_i)$ outputs $\perp$.

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 $\mathcal{U}^{\hat{c}_m}(i, b, \mathsf{S})$:

1. Randomly select λ elements (each of the form $\psi(j), \sigma(j)$) 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 MAC keys generated appropriately from S.)
3. If, for even one level, less than $\alpha\lambda$ of the tags, then output 0. Else go to the next step.
4. Update S to S' so that it now contains an incremented counter.
5. 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.
6. 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.

7. 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 $\mathcal{E}_{\mathsf{LDC}}(\mu_j)$ 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, which in turn is computed from S').

8. 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 value corresponding to each address a. To ensure that we only store the latest values, 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 this 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 i^{th} bit works as follows:

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

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, then output 0. 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 $\mathcal{D}_{\mathsf{LDC}}$. For each bit of the codeword $\psi_j(\ell)$ read, we also authenticate its tag $\sigma_j(\ell)$.

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 $\mathcal{D}_{\mathsf{LDC}}$. If the tag σ verifies, for every bit of the codeword so read, then v is the output. Otherwise, the algorithm outputs $\perp$.

4.5.1 Proof of Theorem 2

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 [11]. 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

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

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},[8] such that $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t} \le \delta k)$ 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 $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_{t+1}}, c_{m_{t+1}}) \le \delta n$
2. m_{t+1} and m_t are identical except possibly at the i^{th} position, where $m_{t+1}(i) = b$

We can show using a Chernoff bound that there exists a suitable constant α satisfying the following properties:

1. When λ positions are randomly chosen in each buffer, each of the form $(\psi(j), \sigma(j))$, more than $\alpha\lambda$ of them will verify in each level, if $\exists c_{m_t} \in \mathcal{C}_{m_t}$ such that $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) \le \delta n$.
2. Along similar lines, less than $\alpha\lambda$ of them will verify in at least one level if $\forall c_{m_t} \in \mathcal{C}_{m_t}$, $\mathsf{Prefix}^{\mathsf{comp}}(\hat{c}_{m_t}, c_{m_t}) > \delta n$.

Therefore by randomly checking λ positions in each buffer, the update algorithm can determine whether or not to do the update.

When an update is done, we determine the number of bits of $\hat{c}_{m_t}$ accessed in order to make the udpate. It is easy to see that in algorithm $\mathcal{U}^{\hat{c}_m}(i, b, \mathsf{S})$, buffer buff_j $(0 \le j \le \tau)$ 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 $|\mathsf{buff}_\ell| = \mathsf{c}(\mu_\ell)$ for some constant c_0.

When $j \ge \tau + 1$, 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 $\mathcal{O}(\log k)$.

In addition, at the start of the update algorith, we also verify λ elements (of the form $\psi(j), \sigma(j)$) in each buffer. This verification accesses $\lambda \log k$ bits of c_{m_t}. Therefore the total locality, w, is bounded by $\lambda \log k$.

[8]Note that the update algorithm needs to determine if such a c_{m_t} exists, using only the state S and $\hat{c}_{m_t}$. In particular, it does without access to either hist or knowledge of m_t.

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 $m(i)$. Similar to the case of local updatability above, we can show that by randomly verifiying λ MAC tags in each level, the decode algorithm can check if the codeword $\hat{c}_m$ is close to some $c_m \in \mathcal{C}_m$. If it is close, then it means that in each level, the codeword stored in $\hat{c}_m$ is close to the one stored in c_m in that level. Therefore, the correctness of the decode algorithm follows from the decodabality of the LDC code from [11].

The locality due to the verification 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 $\tau < \log k$ buffers. Therefore the total locality r is $\mathcal{O}(\lambda \log^2 k)$. □

5 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 [13], were initially defined on static data building upon the closely related notion of sublinear authenticators defined by Naor and Rothblum [17]. Several works have studied the efficiency of such scemes [25, 5, 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.

5.1 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 client 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
- $\mathsf{Audit}()$: 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 $\Pr[\mathsf{ExtGame}_{\bar{\mathcal{S}},\mathcal{E}}(\lambda, \mathsf{p}) = 1] \leq \mathsf{neg}(\lambda)$.

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 described in Section 4.5.

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 ue 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 [26]. 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^*[9].

- $\mathsf{PWrite}(i, v_i)$: 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.
  - 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').

[9] 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 $\mathcal{O}(k\lambda)$.

  - 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 $\mathsf{buff}_{j-1} \dots \mathsf{buff}_0$, as well as $\mathsf{plain}_{j-1} \dots \mathsf{plain}_0$, 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[10].

- $\mathsf{Audit}()$: 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`.

5.3 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 [5]). 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.

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

[10]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.

(along with its MAC of length λ)[11]. 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 $\mathcal{O}(\lambda)$.

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 on the Theory and Application of Cryptology and Information Security*, 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 on the Theory and Applications of Cryptographic Techniques*, pages 279–295, 2013.

[4] 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.

[5] 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.

[6] 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.

[7] O. Goldreich. Towards a theory of software protection and simulation by oblivious rams. In A. V. Aho, editor, *STOC*, pages 182–194. ACM, 1987.

[8] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious ram simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP (2)*, volume 6756 of *Lecture Notes in Computer Science*, pages 576–587. Springer, 2011.

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

[10] 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.

[11] B. Hemenway, R. Ostrovsky, M. J. Strauss, and M. Wootters. Public key locally decodable codes with short keys. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 14th International Workshop, APPROX 2011*, pages 605–615, 2011.

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

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

[11] 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 $\mathcal{O}(\lambda)$.

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

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

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

[17] 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.

[18] R. Ostrovsky. An efficient software protection scheme. In G. Brassard, editor, *CRYPTO*, volume 435 of *Lecture Notes in Computer Science*, pages 610–611. Springer, 1989.

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

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