Generic Efficient Dynamic Proofs of Retrievability

Mohammad Etemad Alptekin Küpçü
Crypto Group, Koç University, İstanbul, Turkey
{metemad, akupcu}@ku.edu.tr

Abstract

Together with its great advantages, cloud storage brought many interesting security issues to our attention. Since 2007, with the first efficient storage integrity protocols Proofs of Retrievability (PoR) of Juels and Kaliski, and Provable Data Possession (PDP) of Ateniese *et al.*, many researchers worked on such protocols. The first proposals worked for static or limited dynamic data, whereas later proposals enabled fully dynamic data integrity and retrievability.

Since the beginning, the difference between PDP and PoR models were greatly debated. Most notably, it was thought that dynamic PoR (DPoR) was harder than dynamic PDP (DPDP). Historically this was true: The first DPDP scheme was shown by Erway *et al.* in 2009, whereas the first DPoR scheme was created by Cash *et al.* in 2013. We show how to obtain DPoR from DPDP and PDP, together with erasure codes, making us realize that even though we did not know it, in 2009 we already could have had a DPoR solution.

We propose a general framework for constructing DPoR schemes. Our framework encapsulates all known DPoR schemes as its special cases. We further show practical and interesting optimizations that enable even better performance than Chandran *et al.* and Shi *et al.* constructions. For the first time, we show how to obtain audit bandwidth for DPoR that is independent of the data size, and how the client can greatly speed up updates with $O(\lambda\sqrt{n})$ local storage (where n is the number of blocks, and λ is the security parameter), which corresponds to less than 3 MB for 10 GB outsourced data, and can easily be obtained in today's smart phones, let alone computers.

Keywords: Dynamic proofs of retrievability, Dynamic provable data possession.

1 Introduction

There is a universal trend toward storage outsourcing through the cloud (e.g., Google Drive, Amazon S3, Microsoft OneDrive), bringing advantages such as cost saving, global access to data, and reduced management overhead. Yet, the most important disadvantage is that the data owner (*client*), by outsourcing her data to a cloud storage provider (*server*), loses the direct control over her data.

Therefore, the client expects having an *authenticated storage* and *guaranteed retrievability* [9, 27]. The former means that the client wants each data access to return the correct value; i.e., a value that is the most recent version of data that has been written by the client herself. The latter means that the client wants to make sure that her data is retrievable; i.e., she can retrieve *all* her data correctly. These authenticity and retrievability checks should be much more efficient than downloading the whole data.

A simple mechanism to provide an authenticated storage is to compute a digest (e.g., hash, MAC, or signature) of data and keep it locally after transferring the data to the server (or in case of a MAC or signature, the key is kept locally, while the tags can be stored at the server). But, the client needs to download the whole data and check it against the locally-stored digest to investigate the authenticity of her data, which is prohibitive given current trends of outsourcing tens of gigabytes of data.

Static techniques. Juels and Kaliski [19] proposed the first scheme to provide such a storage, which was called *proofs of retrievability (PoR)*. The client, before outsourcing her data, encodes it with an erasure-correcting code (ECC), which brings some redundancy while giving the guarantee that the server should manipulate a significant portion of the outsourced (encoded) data in order to impose a data loss or corruption. However, using PoR systems, such a misbehavior resulting in a large data loss or corruption will be caught with a very high probability.

At the same time, Ateniese *et al.* [1] proposed the *provable data possession* (PDP) framework. The main difference was that PDP was not using erasure-correcting codes, and hence was more efficient. Yet, PDP did not provide the same retrievability guarantee that PoR provided.

Shacham and Waters [24] improved the PoR construction and created *compact PoR*. They also later showed that, the compact PoR may be created from a PDP combined with erasure-correcting codes [25]. Yet, this only reflected the relationship between the *static* versions of PDP and PoR.

Dynamic techniques. Most of the existing PoR schemes support only *static* data [19, 25, 8, 13], i.e., they cannot update the outsourced data *efficiently*. In fact, secure and efficient update is the main problem with PoR schemes. Imagine the client wants to update a single block after outsourcing her data. To be efficient, it should affect as small part of the data as possible. But, the server can erase or modify all affected blocks, with a small probability of being caught. To prevent such a misbehavior, a small change should affect a significant fraction of the outsourced data that is not efficient. Efficiency and security are seemingly two conflicting properties related to update in PoR schemes.

Two *flawed* DPoR scheme proposals were given by Cash *et al.* [9], based upon observations by Küpçü [20]. The first proposal, which targets efficiency, divides the data into a number of equal-size blocks and applies the PoR pre-computations on each block separately (i.e., each block is small static PoR). To update a block, the client needs to access only that block. Again, the server, who observes the client update behavior, can remove the whole block while preserving a good chance of passing later audits. The second proposal shows that even permuting the blocks randomly will not solve the problem, as later updates will reveal the original locations of the updated parts to the server.

Cash *et al.* [9] provided the first efficient and secure dynamic PoR scheme using the Oblivious RAM (ORAM) [18]. Later improvements [27, 10], at a high level, separate the updated data from the original data, and store the update logs in a hierarchical data structure similar to ORAM.

On the other hand, the first dynamic PDP protocol was created by Erway *et al.* [15] in 2009; four years before the first dynamic PoR. The reason is that, since PDP-type schemes do not employ erasure-correcting codes, the above-mentioned problems did not exist. Interestingly enough, we show for the first time, how dynamic PDP and dynamic PoR schemes are related, using a general framework.

(D)PDP and (D)PoR differences. The security guarantee a PDP gives is weaker than a PoR in the sense that it guarantees that the client can retrieve *most* of the outsourced data, compared to the PoR that guarantees retrieving the *whole* data. Though erasure-correcting codes help providing full retrievability, they bring a problem as well: to perform a small update the whole outsourced data is required (discussed in more detail in Section 2.1). This is the main source of problems toward making PoR dynamic.

Our contributions. We analyze the existing work on PoR and DPoR in detail, and identify their weaknesses and strengths. Then, we propose a generic dynamic PoR scheme construction framework encompassing previous DPoR schemes as special cases, with various optimizations.

- Since the proposal of PDP [1] and PoR [19] in 2007, there was a trend to find the possible relations among these schemes. Based on our in-depth analysis of these schemes and deep understanding of their goals, we express that **a DPoR scheme can be built given black-box access to a DPDP** [15] **and a *static* PoR scheme** [25] (**which, in turn, can be obtained from a *static* PDP and an ECC scheme**). This is of utmost importance, as, in some sense, it means that in 2009 when the first DPDP scheme appeared, we indeed immediately could have had dynamic PoR schemes as well.
- We propose a generic efficient dynamic PoR scheme. **The existing dynamic PoR schemes [10, 27] are special cases of our general model**.
- One important difficulty in existing DPoR schemes is the excessive volume of communication. We show how to aggregate proofs of all blocks challenged in an audit into one, and reduce the audit bandwidth from $O(\lambda \log(n))$ [10, 27] to $O(\lambda)$, where n is the number of outsourced blocks and λ is the security parameter, obtaining an **optimally-efficient audit bandwidth**. Interestingly, one realizes that this is *below* the memory-checking bound of $O(\log(n)/\log\log(n))$ [14], since it is *communication* and *not computation*.
- Adding local storage to the client does not affect the asymptotic costs of the existing schemes [9, 27, 10] as pointed to by Williams and Sion [34]. In contrast, **we show how adding local storage to the client does improve the update cost in our scheme**.
- We propose the *erasure-coded and authenticated log* (*ECAL*) as an efficient data structure to store the update logs, in different configurations. While previous works' *reshuffling* operations require $O(n)$ temporary storage at the client for each update [9, 27, 10], our *equibuffers configuration* reduces it considerably into orders that are available in almost all existing hand-held electronic devices, letting us employ even mobile phones for updating data on the cloud storage. (A *rebuild* still requires large client memory, e.g., a computer, but it is done only once every $O(n)$ updates.) We consider a client storage of size $O(\lambda\sqrt{n})$ which is $\sim$3 MB for 10 GB outsourced

data. This is a reasonable assumption since most of hand-held electronic devices today have more than 3 MB local memory. Thus, without the need for large reshuffling memory, our protocol may even be used by smart phones for updating data (e.g., now you can update a text file on your secure Dropbox on the go).

- We observe that the client storage, S_{client}, and the update and audit computational costs on the server, C_{update} and C_{audit}, are related together via the relation $S_{client} * C_{update} * C_{audit} = O(n)$ (ignoring factors depending on the security parameter). The relation states that, for different environments with different client and server capabilities, one can choose these parameters in a way that the whole scheme exhibits the best performance.

1.1 Related Work

A simple solution for the problem of validating retrievability and integrity of the outsourced data is that the client computes a digest (e.g., Hash or MAC) of the whole data, keeps it locally, and transforms the data to the server. Later, on every audit, the client needs to download the whole data, compute its digest, and compare it with the locally-stored one. But, this is impractical since the communication complexity is linear in the data size [5].

PoR was first proposed by Juels and Kaliski [19] for *static* data. The data is erasure-coded and encrypted. Then, a set of *sentinel* blocks are appended, the result is permuted randomly, and outsourced. The sentinel blocks are used to check authenticity with high probability, and in case of any unauthorized manipulation, the erasure-correcting code will help recover the original data. Juels and Kaliski's PoR supports only a *limited* number of challenges.

Shacham and Waters [24] gave the first PoR schemes with full security proofs against arbitrary adversaries. They proposed two schemes, one using pseudorandom functions that is secure in the standard model, the other using BLS signatures [6] that is secure in the random oracle model. The former supports only private verifiability while the latter allows public verifiability. The main advantage over [19] is that it supports *unlimited* number of challenges.

Bowers *et al.* [8] proposed a theoretical framework for PoR design, enhancing the above constructions [19, 24]. It supports static data, with a limited number of challenges. Dodis *et al.* generalized the static PoR schemes [13].

PDP, first proposed by Ateniese *et al.* [1], is a very close line of work that provides probabilistic guarantees of data possession using a challenge-response mechanism. Similar schemes were later proposed targeting public verifiability [35, 32] and availability [7, 12, 5]. Curtmola *et al.* [11] integrated PDP with ECC to enhance the possession guarantee, and proposed the notion of *robust data possession*.

Dynamic PDP. The above schemes can only be used for archival purposes, i.e., they do not support dynamic data. Ateniese *et al.* [2] gave a dynamic PDP scheme where they pre-compute and store at the server a limited number of random challenges with the corresponding answers. Therefore, the number of challenges is limited and later updates affect all remaining answers. Erway *et al.* [15] proposed the first fully dynamic PDP scheme providing $O(\log(n))$ complexity for updates and audits, where n is the number of blocks. Later variants use other data structures [33, 31], supply additional properties [4], distribute and replicate [17], or enhance efficiency [16].

Dynamic PoR. Zhen *et al.* [23] and Zheng and Xu [36] claimed to give dynamic PoR schemes, but actually dynamic PDP schemes were given. They do not consider the erasure-correcting code upon performing updates. Similarly, Zheng and Xu [36] proposed another dynamic PoR scheme that is again a dynamic PDP in nature. The problem is that they apply the erasure-correcting code at the block level which means that retrievability is not guaranteed if a data loss is detected. Similarly, the emphasis of the scheme by Wang *et al.* [33] is on data integrity while they claim retrievability. The main reason is that, in our opinion, the distinction between PDP and PoR schemes was not well-understood at that point. We also contribute in this regard.

Stefanov *et al.* [28] proposed a dynamic PoR scheme inside a cloud file system called Iris, which is an authenticated file system providing data integrity and freshness as well as dynamic proofs of retrievability. However, Iris is not a fully-dynamic PoR scheme as it stores the erasure-coding data *locally* (on a trusted party called the *portal*). Moreover, by shifting the block verification to the file system, they reduced the burden on the PoR.

The first really dynamic PoR scheme with full security definition and proof was proposed by Cash *et al.* [9]. The scheme has constant client storage and polylogarithmic communication and computation. As a building block, they use an ORAM satisfying a special property called *next-read-pattern-hiding*. Although it achieves asymptotic efficiency, ORAM is a complicated and heavy cryptographic primitive that is (currently) not practically efficient.

Chandran *et al.* [10] proposed a locally updatable and locally decodable code, and used it to construct a dynamic PoR scheme. They erasure-code the data, and store it remotely inside a hierarchical authenticated data structure similar to ORAM in nature. Later updates are also erasure-coded and stored in the same data structure. Reading through that structure requires $O(n)$ cost, hence, they store the plain data and subsequent updates in another similar structure to support read operations efficiently.

Shi *et al.* [27] proposed a dynamic PoR scheme similar to [10], using the fast incrementally-constructable codes to achieve efficiency. Later, they improved their scheme by outsourcing some part of computation to the server, reducing the communication and client computation. Using the introduced homomorphic checksum, the client only performs the computation on these checksums, leaving computation on data itself to the server.

Relationship among these schemes. Shacham and Waters [25] showed that a PoR scheme can be obtained by employing a PDP system together with erasure-correcting codes. We will employ their result in our paper.

The first dynamic PDP schemes [15, 33], providing authenticity, appeared in 2009, while the first dynamic PoR schemes [9, 10, 27], with retrievability guarantee, were proposed in 2013. By analyzing the differences and similarities between DPDP and DPoR in-depth, for the first time, we show how to tie these schemes together. In fact, a DPoR scheme can be built using (black-box access to) DPDP and PoR (which, in turn, can be built using PDP and erasure-correcting codes [25]). This interestingly reveals that we could have had DPoR schemes in 2009 when the first DPDP schemes appeared. Moreover, existing DPoR schemes [10, 27] are special cases of our general model.

1.2 Preliminaries

Notation. We use $x \leftarrow X$ to denote x is sampled uniformly from the set X, $|X|$ to represent the number of elements in X, and $||$ to show concatenation. PPT denotes probabilistic polynomial time, and λ is the security parameter. By *log*, we mean the footprint an update operation leaves, while $\log(.)$ indicates logarithm calculation.

A function $\nu(\lambda) : Z^+ \rightarrow [0,1]$ is called *negligible* if $\forall$ *positive polynomials* p, $\exists$ *a constant* c such that $\forall\ \lambda > c$, $\nu(\lambda) < 1/p(\lambda)$. *Overwhelming* probability is greater than or equal to $1-\nu(\lambda)$ for some negligible function $\nu(\lambda)$. *Efficient algorithms* have expected running time polynomial in the security parameter. By *efficient algorithms*, we mean those with expected running time polynomial in the security parameter.

PDP. In a PDP scheme (e.g., [1]), the client divides a file F into n blocks, $F = (f_1, f_2, ..., f_n)$, computes a *tag* for each block, and transfers the file along with the tags to the server, deleting its local copy. Later, she sends a *challenge*, which is a subset of block indices selected randomly, to the server. Upon receipt, the server constructs a *proof* using the tags and blocks stored, and sends it back to the client for verification. The tags are *homomorphic* [3], meaning that it is possible to combine multiple of them into a single tag, reducing the proof size.

Dynamic PDP. The DPDP scheme of Erway *et al.* [15] stores the block tags in a rank-based authenticated skip list used to generate cryptographic proofs. The authenticated skip list supports update operations efficiently, with $O(\log(n))$ cost. The DPDP scheme consists of the following PPT algorithms [15]:

$\{sk, pk\} \leftarrow \texttt{KeyGen}(1^\lambda)$: run by the client to generate the secret and public key pair (sk, pk), which takes the security parameter as input. The client shares the public key with the server.

$\{e(F), e(info), e(M)\} \leftarrow \texttt{PrepareUpdate}(sk, pk, F, info, M_c)$: run by the client to prepare the file to be stored on the server. It takes as input the secret and public keys, the file F, the definition $info$ of the update, and the previous metadata M_c, and generates an encoded version of $e(F)$, the information $e(info)$ about the update, and the new metadata $e(M)$. The outputs are sent to the server.

$\{F_i, M_i, M_c, P_{M_c}\} \leftarrow \texttt{PerformUpdate}(pk, F_{i-1}, M_{i-1}, e(F), e(info), e(M))$: run by the server upon receipt of an update request. The public key pk, the previous version of the file F_{i-1}, the metadata M_{i-1}, and the outputs of `PrepareUpdate` are given as input. It generates the new version of the file F_i, the metadata M_i, together with the client metadata and its proof (M_c and P'_{M_c}).

$\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{VerifyUpdate}(sk, pk, F, info, M_c, M_c, P_{M_c})$ run by the client to verify the server's response with inputs of `PrepareUpdate` algorithm, M_c and P'_{M_c}. It outputs an acceptance or a rejection signal.

$\{c\} \leftarrow \texttt{Challenge}(sk, pk, M_c)$: given the secret and public keys (sk, pk), and the latest client metadata M_c as input, it generates a challenge c.

$\{P\} \leftarrow \texttt{Prove}(pk, F_i, M_i, c)$: given the public key, the latest version of the file and metadata, and the challenge, the server generates a proof P to be sent to the client.

$\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{Verify}(sk, pk, M_c, c, P)$: run by the client to verify the proof P, given the secret and public keys (sk, pk), the client metadata M_c, and the challenge c as input. It outputs an '`accept`' or a '`reject`' signal.

Erasure-correcting codes deal with errors occur during data transmission over a noisy channel, or data storage in a device. An $(n, k, d)_\Sigma$ erasure-correcting code over a finite alphabet Σ is a pair of efficient encoding and decoding algorithms $(\texttt{encode}, \texttt{decode})$ such that $\texttt{encode} : \Sigma^k \rightarrow \Sigma^n$ transforms a message $\mathbf{m} = (m_1, m_2, ..., m_k) \in \Sigma^k$ into a codeword $\mathbf{c} = (c_1, c_2, ..., c_n) \in \Sigma^n$, and `decode` : $\Sigma^{n-d+1} \rightarrow \Sigma^k$ recovers the original message from a codeword in the presence of at most $d-1$ erasures.

Compact PoR. Shacham and Waters [25] showed how to obtain (compact) PoR from an efficient PDP scheme together with an erasure-correcting code. At a high level, the data is erasure-coded, permuted (pseudo-)randomly, and the (PDP) tags are computed for the resulting blocks. **Throughout the paper, when we talk about PoR schemes, we mean this Shacham and Waters construction**. In general, we employ Ateniese *et al.* [1] PDP scheme as the building block, but at times we modify it to obtain an even more efficient PoR construction.

- $(pk, sk) \leftarrow \texttt{Kg}(1^\lambda)$: is a randomized algorithm that generates a public and private key pair (pk, sk) given the security parameter λ as input.
- $(M', \tau) \leftarrow \texttt{St}(sk, M)$: is a randomized algorithm that takes the secret key sk and a file M as input and produces a processed file M' and a tag τ as output.
- $\pi \leftarrow \texttt{P}(pk, M', \tau)$: is an algorithm run by the server that takes the public key pk, the outsourced file M', and the tag τ as input and generates a proof π.
- `accept/reject` $\leftarrow \texttt{V}(sk, pk, \pi)$: is an algorithm run by the client to verify the proof coming from the server. Given the secret and public keys and the proof π, it outputs an acceptance or a rejection signal.

2 Informal Technical Overview

2.1 Observations

By investigating the previous work, the problems they pointed to, and the given solutions, we made the following observations that show the conditions an efficient and secure dynamic PoR scheme should satisfy. We regard these as one of our main contributions:

- **Observation 1**. To ensure retrievability of the outsourced data, erasure-correcting codes can be used. Using an $(n, k, d)_\Sigma$ erasure-correcting code, if the adversary manages to manipulate a small part of the data (i.e., up to $d-1$ out of n blocks), the data retrievability is still guaranteed [19, 25, 11, 28, 10].
- **Observation 2**. If the adversary manipulates a significant part of the data (i.e., more than d-1 blocks), it cannot be recovered using the erasure-correcting code. An integrity checking mechanism is needed to catch such an adversary, with high probability [15, 11, 28]. Note that the integrity checking mechanism needs to detect only such significant modifications/deletions.
- **Observation 3**. Simple updates on erasure-coded data is not enough. If the data is erasure-coded, later updates should also affect the corresponding parts of erasure-correcting codes. Two possibilities are [20, 9]:
  - If a single update affects a small part of the encoded data (i.e., the code is locally updatable), then the server learns and later can erase the whole update without a high probability of getting caught. Thus, this option is *not secure*.
  - If a single update affects a huge part of the encoded data (e.g., the whole encoded data), requiring $\sim O(n)$ cost, then the server learns almost nothing about the update locations and cannot attack them. But such updates are very costly for the client, both communication-wise and computationally as she should download the whole encoded data, decode it, update it, encode it again (with different setting), and upload it to the server. Briefly, this option is not *efficient*.
- **Observation 4**. Therefore, it is better to store the update information separately, rather than applying the updates on the encoded data. There should be two parts of server storage: one part stores the encoded original data, and the other part stores the update information (referred to as the *log store*). The log store, which is empty at the outset, can grow to be as much as linear in the data size. When the log store becomes full, the updates in it will be applied on the original data. This generates the latest version of data and an empty log store [10, 27].
- **Observation 5**. Since the log store can be as large as (linear in the size of) the original data, the efficiency problem is again encountered. The remedy is to use a hierarchical data structure [9, 10, 27]. Each level is erasure-coded, updated, and audited independently, and possibly merged into the next level once filled up.
- **Observation 6**. Although the insecurity and inefficiency problem of the update logs can be solved using the observations above, we are now faced with a new problem: to read the latest version of some data, we need to decode the encoded data and apply all the logs, which requires $O(n)$ time. To solve, one should store an uncoded version of data, protected by a dynamic memory checking scheme, at the server [10, 27]. This frees the read operation from difficulties of struggling with the erasure-correcting codes. For read operations, the membership

proofs of the memory checking scheme serves as the authenticity proof (i.e., if the proof is accepted, we can be assured with high probability that the challenged data is kept intact on the server [15]).

- **Observation 7**. It is enough that the memory checking scheme is only responsible for *authenticity* of the data read, and hence the read operations need *not* be *oblivious*. This reveals the access pattern of the client, but access privacy is not a requirement of the dynamic PoR definition [27]. The log store, on the other hand, needs oblivious operations, due to the explained failed attempts regarding the update information enabling the server to create data loss or corruption. Shi *et al.* [27] and Chandran *et al.* [10] also followed a similar path and provided the privacy only for operations on the encoded log structure. Cash *et al.* [9], on the other hand, used the encoded data to read and update the outsourced data. Hence, both read and update operations are performed obliviously in their scheme. We also observe that *the log store can be append-only*.
- **Observation 8**. An ORAM structure performs both read and update operations in a similar manner that an adversary can not distinguish them. Therefore, a read operation cuts (or just copies) the requested data item and inserts it back in the ORAM from the top level [9], requiring reshuffling operation. Therefore, if the read operations will not be run through the ORAM, then there is *no need to perform the extra heavy reshufflings*. This is an important observation that we will utilize to construct an efficient structure for storing the logs.

2.2 Overview of Our Solution

Now that we have observed some important aspects of the design, we can present an overview of our general framework. In our framework, any update operation leaves a footprint, which is called a *log*. If these logs are kept securely and erasure-coded, even in case of any data loss, the data can be recovered using the logs. This is conceptually similar to what a database management system or a journal-based file system performs in the background. However, a main difference is that we do not trust the server to keep the logs correctly, so we should audit the server.

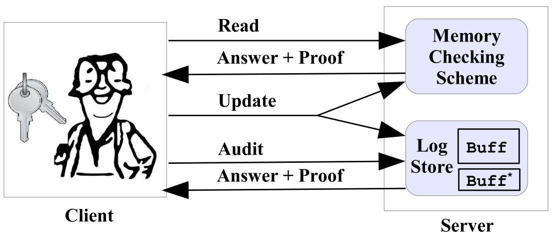

Figure 1: Our model.

Inspired by the existing work [9, 27, 10], our scheme has two parts: one is a data structure for keeping update logs securely and providing the retrievability guarantee, the other is a dynamic memory-checking scheme (e.g., DPDP [15]) that responds to read queries providing authenticity, as shown in Figure 1. The update operations affect both parts.

Initially, the client has some original data, which is stored twice at the server: once in the memory checking scheme (e.g., DPDP [15]), and once in the log store in an erasure-coded and garbled manner. Later, to update (insert, delete, or modify) a data block, the client prepares the corresponding command and directs it to the memory-checking part for execution. In parallel, she prepares the respective update log, to be appended to the existing logs in the log store.

During normal execution, read operations are responded by the memory-checking scheme equipped with authenticity proofs. However, in case of any data loss in the memory-checking part such that the read operation returns incorrect responses (or nothing), the log store is used to recover the requested data. If it cannot be recovered even using the logs, the server is misbehaving, and this will be caught with high probability. *Since the log store supports retrievability, it is enough that audits are performed over that part*. Answers to both the read and audit are accompanied with cryptographic proofs, so the client can verify them.

3 Erasure-Coded Authenticated Log

The log store plays an important role in our scheme. It is a special authenticated data structure (ADS) [30] that inspects integrity and guarantees retrievability of the logs, which in turn, guarantee retrievability of the outsourced data. We call it the *Erasure-Coded Authenticated Log* (*ECAL*). The ECAL first erasure-codes the logs (to guarantee retrievability), and garbles the result (e.g., by encrypting the blocks and permuting them randomly) to make locating any part of the original data difficult for the server. Finally, it provides authenticity using a homomorphic tag. Any scheme supplying *retrievability* and *authenticity* can be used to store the update logs. In fact, what we need is a PoR scheme with efficient update and audit, without caring about the read efficiency (since it is used rarely, only for lost data recovery and infrequent rebuilds).

We show that a static PoR scheme does the job, when used hierarchically. Hereafter, we use the compact PoR [25] as our static PoR building block. Any similar scheme can be used instead, but our optimizations are only applied over the PDP-based schemes such as compact PoR.

3.1 Basics and Definition of ECAL

The above observations demonstrate us how to build the ECAL. Briefly, we need an erasure-coded log (insuring retrievability) stored in a static memory checking scheme (providing integrity), satisfying efficiency requirements. Any scheme using ECC together with one of the integrity-preserving methods satisfies the requirements, and, if used for storing static data, is a PoR scheme [25]. To provide the efficiency, we use a hierarchical memory in which, each level employs a distinct instance of the compact PoR [25].

Using $(n,k,d)_\Sigma$ erasure-correcting codes to construct the ECAL guarantees the stored logs are still retrievable after up to $d-1$ erasures (Observation 1). The ECC, by itself, cannot guarantee the integrity, as more than d modifications leads the code to be decoded into a different value. To supply the integrity (Observation 2), we need to use either *blinded codes*, *MAC schemes*, or *Memory-checking schemes*. Any scheme using ECC together with one of these integrity-preserving methods satisfies the requirements, and, if used for storing static data, is a PoR scheme [25].

Each update the client performs on her data leaves a log. Regardless of the location of the updates on the plaintext data, the new logs will be appended to the end of the existing logs, i.e., the logs are append-only. Outsourcing all these logs at a rather safe place using ECAL guarantees that if the original data faced integrity problems, it can be recovered through the logs. Hence, the client needs to investigate integrity of her logs over time, helping her induce that either the data is fully retrievable, or the server is misbehaving. All definitions below follow closely those of Cash *et al.* [9].

However, due to the nature and application of the ECAL, read operation should not normally be fulfilled through the ECAL, since reading a data block requires reading, decoding, and reconstructing all logs, necessitating $O(n)$ cost. Therefore, read is not an efficient operation in ECAL, and hence, retrieving through the ECAL should be the last resort when other options fail. *This is the important reason why ECAL is not an efficient DPoR scheme is its own.* All the definitions below follow closely those of Cash *et al.* [9].

Definition 3.1 (ECAL) *An erasure-coded authenticated log scheme includes the following PPT interactive protocols between a stateful client and a stateful server:*

- $\texttt{LInit}(1^\lambda, 1^w, n, M)$: The client starts up this protocol to initialize an empty ECAL memory on the server, providing as input the security parameter λ, the word size w, and the memory size n. (The memory need not be bounded.) The initial data M is also outsourced into the initialized memory.
- $\texttt{LAppend}(l)$: The client uses this protocol to ask the server append the log l to the set of logs already stored.
- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{LAudit}()$: The client specifies a challenge vector and uses this protocol to check whether the server keeps storing the logs correctly (i.e., the logs are retrievable). She finally emits an acceptance or a rejection signal.

Both the client and the server create their own local states during execution of the `LInit` protocol. These local states will be used and updated during execution of the other protocols following `LInit`. We assume that `LInit` creates an empty memory at the server, but if the client has some initial data, she can send them all using `LInit` and ask the server to store them as the initial data, or she can append them one-by-one using `LAppend`. `LInit` and `LAppend` do not include verification. If the server misbehaves, it will be caught by the subsequent `LAudit` executions.

The definition of `LAppend` does not take into account the client's local memory. But note that, if the client has a local memory of size t, then she can keep the logs locally until her local memory gets filled, and then, she transfers all logs together to the server, reducing the total communication requirement (i.e., one may see l as a vector $\boldsymbol{l}$).

3.2 Security Definition

Correctness considers the honest behavior of the (client and the) server in a protocol execution. A scheme is *correct* if the following is satisfied with probability 1 over the randomness of the client: Reading the i^{th} data, results in a value v such that v is the i^{th} value that has already been written by an `LAppend` protocol. If less than i data exists, it returns $\perp$. Moreover, the `LAudit` protocol, once executed, results in an acceptance.

Authenticity. If the server deviates from honest behavior by providing proofs while he has manipulated the challenged part of data, the client should detect it with overwhelming probability. The authenticity game $\texttt{AuthGame}_{\widetilde{\mathcal{S}}}(\lambda)$ between a challenger and a malicious server $\widetilde{\mathcal{S}}$ is defined as:

- **Initialization**. The challenger starts the `LInit` protocol to initialize the environment. The challenger also starts a copy of the honest client $\mathcal{C}$ and the honest server $\mathcal{S}$, and runs `LInit` among them.

- **Setup**. The server $\widetilde{\mathcal{S}}$ asks the challenger to start a protocol execution (`LAppend` or `LAudit`) by providing the required information. The challenger starts two parallel executions of the same protocol between $\mathcal{C}$ and $\mathcal{S}$, and between itself (acting as the client) and $\widetilde{\mathcal{S}}$, using the information provided by the server. This is repeated polynomially-many times.
- **Challenge**. $\widetilde{\mathcal{S}}$ sends an audit request to the challenger, who initiates two parallel `LAudit` protocols, one between $\mathcal{C}$ and $\mathcal{S}$, and one between itself and $\widetilde{\mathcal{S}}$, using the same randomness.

$\widetilde{\mathcal{S}}$ wins the game if his answer is accepted while it differs from that of $\mathcal{S}$; in such a case the game returns 1. It is expected that $Pr[\texttt{AuthGame}_{\widetilde{\mathcal{S}}}(\lambda) = 1] \leq \nu(\lambda)$ for any efficient adversarial server $\widetilde{\mathcal{S}}$, and some negligible function $\nu(\lambda)$.

Definition 3.2 (Authenticity) *An ECAL scheme is authentic if no PPT adversary can win the above game with probability better than negligible in the security parameter.*

Retrievability. We want the ECAL to guarantee that if a malicious adversary performs more than $d-1$ erasures within some level[1], he should not pass the subsequent audit: i.e., if a malicious adversary passes the audit with a non-negligible probability, then he should have sufficient knowledge of *all* outsourced logs. The knowledge is formalized via existence of an efficient extractor that, given black-box access to the malicious adversary, can retrieve all logs L. The retrievability game among a challenger, an extractor, and a malicious server $\widetilde{\mathcal{S}}$ is as:

- **Initialization**. The challenger creates a copy of an honest client $\mathcal{C}$, and starts the `LInit` between $\mathcal{C}$ and $\widetilde{\mathcal{S}}$.
- **Setup**. $\widetilde{\mathcal{S}}$ adaptively asks the challenger to start a protocol `LAppend` or `LAudit`, giing the required information. The challenger forwards the request to the client who starts up the protocol. The adversary can repeat this process polynomially-many times. Call the final states of the client and malicious server, $st_{\mathcal{C}}$ and $st_{\widetilde{\mathcal{S}}}$, respectively.
- **Challenge**. The extractor repeats the `LAudit` protocol polynomially-many times with $\widetilde{\mathcal{S}}$ in the final state $st_{\widetilde{\mathcal{S}}}$ (via rewinding). Call the extracted data M$'$.

Definition 3.3 (Retrievability) *An ECAL scheme provides retrievability if there exists a PPT extractor such that for all PPT $\widetilde{\mathcal{S}}$, if $\widetilde{\mathcal{S}}$ passes the* `LAudit` *protocols with non-negligible probability, then at the end of the game we have M'= 'genuine data' with overwhelming probability.*

3.3 Generic ECAL Construction

Assume Σ_m=$\{0,1\}^{w'}$ and Σ_l=$\{0,1\}^{w}$ are two finite alphabets showing the message space and log space, respectively. The client initializes a compact PoR scheme [25] `CP=(Kg,St,P,V)`, puts the original data M=$(m_1,...,m_k) \in \Sigma_m^k$ inside it, and outsources the result to the server. The result will be stored at `Buff`* (part of the log store) and will not be changed until the next rebuild, which puts the last version of the client data inside a new compact PoR instantiation and outsources the result again into the `Buff`*. Later, she performs updates on the original data, and outsources the corresponding logs at the `Buff` in a way that the adversary cannot differentiate or locate the recently-added logs among the already-existing ones. The content of `Buff` changes as new logs are outsourced. The client wants to rest assured the logs are retrievable, which means that she can rebuild the final (or any intermediate) version of her data.

We present each update log as a single block in Σ_l, therefore, each data insertion, deletion, or modification appends a new block to the existing logs. Each update log contains the location on the plain data, the operation type, and the new value to be stored. Indeed, a log is of the form iOv, where i is the index on the plain data, $O \in \{I,D,M\}$ is an update operation, and v is the new value to be stored at the stated index (v is empty for deletion). As an example, let M ='$abcde$' be a message of length 5. The update log '$2Mf$' states that the value at the second location is modified to f. Applying the series of update logs $L = (2Mf, 5Mk, 3It, 5My, 4Ms)$ brings M to the final state M ='$aftsyk$'.

The *age* of an update log is the time elapsed since the time the log is arrived, i.e., the log that arrived first is the oldest one, and the log that arrived most recently is the youngest one. It is important to store the logs ordered according to their age, since applying the same logs in a different order may lead to a different data.

The logs $L = \{i_j O_j v_j\}_{j=1}^{k}$ are ordered according to their age and put in a compact PoR scheme [25], which generates an encoded version of the logs $C = (C_1, C_2, ..., C_n) \in \Sigma_l^n$. The encoded logs are then outsourced to the server. The authenticity is handled by the compact PoR, which ensures that the logs are retrievable (or, the server is caught misbehaving). This, in turn, ensures retrievability of the client data even if the outsourced uncoded data is corrupted.

[1]When levels are not of the same size in hierarchical configurations, then each level may have a different d parameter, depending on the erasure-correcting code used.

There are two types of memory on the server. Buff*, whose content is fixed between two rebuild operations and is the same among different configurations, stores the logs corresponding to the original data of the client. The other memory, Buff, stores the logs of the subsequent updates on the data, and its efficiency affects the whole scheme. We only consider and work on the later throughout this paper. By 'ECAL construction', we also mean the later. Below we present various ECAL constructions; all provide security, but with different efficiency.

An advantage of using PDP over MAC (e.g., schemes in [10, 27]) is that the PDP uses *homomorphic* tags, using which it can aggregate together a set of requested blocks, and send only one block to the client. This reduces the communication dramatically. Although Shi *et al.* [27] used a homomorphic checksum scheme, the outsourced checksums are encrypted, and hence are not ready to be aggregated. Therefore, they can only aggregate the blocks.

Once in every $O(n)$ updates, when the Buff becomes full, a rebuild operation applies all logs in the Buff on the client data stored at Buff*, empties the whole Buff, puts the latest state of the client's data in a new compact PoR, and stores the result at Buff* (whose size is increased if needed). Note, however, that our rebuild operation is very light compared to that of [9, 10, 27] due to the existence of the memory-checking scheme who can provide the client with the (authenticated) latest version of her data, i.e., she does not need to apply all logs on the original data one-by-one to compute the latest version. The client only verifies the received data and if accepted, puts it inside a new compact PoR instantiation and outsources the result again at Buff*.

3.4 Existing Configurations of the Buff

Linear configuration. Buff can be, in the simplest form, a one-dimensional buffer (of length n) storing the output of the compact PoR [25] constructed over the logs. The client stores a counter ctr to keep the size of the logs, initialized to zero and incremented every time a new log is created. To add a new log (or a set of new logs up to the size of the client local storage) to the Buff, the client should download all the existing outsourced logs, decode it to retrieve the plaintext logs, append the new log, initialize a new compact PoR instantiation to put the logs inside, and upload the result. Although the audit is an $O(\lambda)$ operation, this construction suffers from the same efficiency problem as the original static PoR: to prevent the adversary from tampering with the recently-added parts of the logs, a small update affects all the outsourced logs. Therefore, the amortized server computation and bandwidth (after n updates) is: $(1+2+...+n)/n = n/2 = O(n)$.

According to the Observation 7, we only care about obliviousness of the update (append) operations, which is provided by the compact PoR [25]. Combined with the Observation 8, it illustrates that the exact requirement is to only accumulate the logs given by the update operations, according to their age. This means that the intermediate reshufflings[2] (of smaller buffers into larger ones) are not required, which simplifies the log structure. This is an important difference between our scheme and those of [10, 27] that reshuffle the buffers.

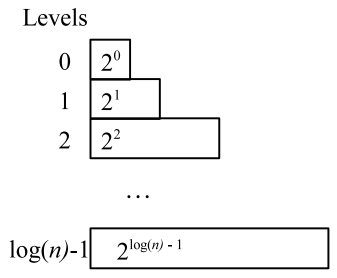

Figure 2: Incremental-buffers.

Incremental-buffers configuration. Similar to ORAM [18], there is a sequence of buffers whose sizes grow at a geometric rate [22] and are organized in a way that the buffer at the i^{th} level stores 2^i elements, as shown in Figure 2. In ORAM structures, once the smaller buffers are filled, they are *obliviously reshuffled* into the larger buffers [18, 22] to provide the *access privacy*, i.e., the client completely hides her data access pattern from the untrusted server [29]. Although an adversary can observe which physical storage locations are accessed, he has a negligible chance of learning the real access pattern of the client [26]. But, this is achieved at a very high cost, e.g., $O(\log^2(n)/\log\log(n))$ is the best known overhead [22, 29].

A dynamic PoR scheme using ORAM was proposed by Cash *et al.* [9]. Since access privacy is not specified by the PoR definition [25, 27], a complete ORAM scheme is not needed. Instead, only the hierarchical buffers idea from the ORAM is taken to preserve obliviousness of the update operations [10, 27]. This means that the adversary can observe the access pattern of the client. With each update, a buffer is re-encoded (in a new compact PoR), making it hard for the adversary to correlate the new content to the old one, or to locate the newly-added logs.

If the total size of the (encoded) log memory is n, there are $\log(n)$ levels, first level storing $2^0 = 1$ log and last level storing $2^{\log(n)-1} = n/2$ logs. In this structure, an update operation adds a new log to the first level, if it is empty. Otherwise, if the first empty level is j (the client can find it using her local state, as we will see later), then the logs

[2]There are two types of rebuild operations in most ORAM schemes [18, 26, 22] and ORAM-based dynamic PoR schemes [9, 27]: the *reshuffling* that is executed on each 2^j updates to reshuffle and transfer contents of all buffers $i < j$ into the buffer j, making empty all buffers $i < j$, and the *rebuild* that is executed when all buffers become full (i.e., once in every $O(n)$ updates) to read all buffers, apply them on the original data, and empty all buffers.

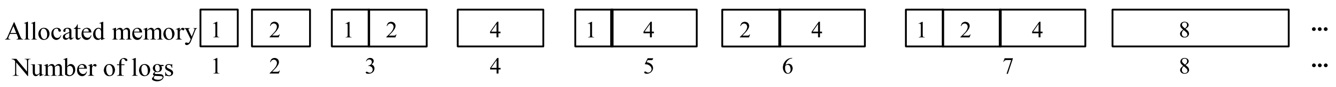

Figure 3: The server allocates more memory as new logs arrive.

stored at all levels $i < j$ are read by the client, the new log is appended, all are put in a new compact PoR scheme, the result is stored at level j, and all levels $i < j$ are emptied.

This is, however, a conceptual organization, and the server does not need to allocate such a memory at the outset. He allocates new memory as new logs arrive. For the sake of simple presentation, assume each log as a cell. The first log requires only one cell. When the second log comes in, it is merged with the already-existing one, and the result is stored at a memory piece with two cells. The occupied memory size increases as new logs arrive. This process is depicted in Figure 3.

3.5 Equibuffers Configuration

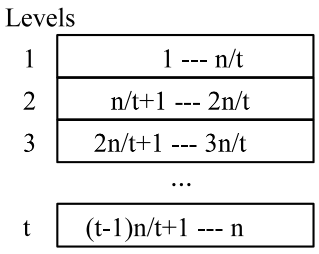

Figure 4: Equibuffers.

Direct application of the incremental-buffers configuration to the dynamic PoR is not suitable as it imposes unnecessary burden. First, although the upper-level buffers are small, the lower-level buffers are of size $O(n)$ that requires $O(n)$ temporary storage at the client to perform the reshuffling. This amount of memory is not available in most current hand-held devices, thus, they cannot update the outsourced data. Second, adding some (permanent) local storage to the client will not improve the asymptotic costs of this configuration as pointed to in [34] and proven in Appendix A. Third, managing a buffer divided into levels of different size is complicated for the client.

As an alternative, we propose the equibuffers configuration in which all levels are buffers of the same capacity. If the total size of the `Buff` is n and there are t levels, each level has size n/t. A simple representation is shown in Figure 4. An advantage of this configuration over the previous one is that all buffers are of the same size, hence, all operations become alike and simple. Since all levels are similar, we can fill them up from one end. Most importantly, this setting **does not require reshufflings** done in the incremental-buffers construction to combine the previous levels.

Another important advantage of this configuration is that **adding local storage to the client does improve the update complexity**. Assume there are t levels, each of size n/t, on the server, and a local storage of size n^{δ} with $1 < n^{\delta} \leq n/t$ on the client. Now, the client can accumulate n^{δ} update logs at her local storage and outsource them at once, reducing the amortized update complexity from $O(n/t)$ to $O(n^{1-\delta}/t)$. Setting $\delta = 1/2$, $\sqrt{n}$ local memory is available on almost all existing hand-held electronic devices (e.g., around 3 MB memory is required for 10 GB outsourced data).

Regarding the Observation 7 and 8, the update logs are accumulated into the buffers from one end, until all buffers become full. Then, the client can either ask for further storage, or perform a rebuild that stores the (encoded) latest state of her data at `Buff`* and resets `Buff`. This rebuild, however, can be performed using a computer.

3.6 ECAL Protocols

In both hierarchical configurations, the client's local state contains the keys of the compact PoR schemes in all levels and a counter `ctr`, which is incremented with each update. Using `ctr`, the client can find the buffer that the current update log should be directed to. One can think of these hierarchical buffers as a one-dimensional buffer storing the update logs sequentially, as shown in Figure 5. Each level is treated as a separate compact PoR scheme, with a different pair of keys stored at the client. For each buffer, a proof is computed containing an aggregation of the challenged tags, and a value based on the combination of the challenged logs. Since the compact PoR satisfies *blockless verification* [1], the challenged blocks themselves are not sent to the client. Totally, the servers answer contains $2(\log(n)+1)$ values, that we will reduce it to only two values.

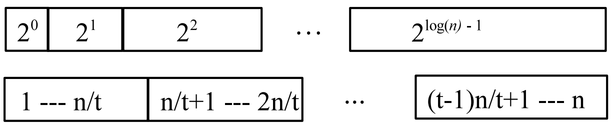

Figure 5: One-dimensional view of hierarchical configurations.

Note that each buffer uses a different PDP key. The reason is that the client needs to read the current content of the buffer to which the new logs will be written, add the new logs, and re-encode and re-compute the PDP tags. Using the old keys for generating the new tags makes *replay* attacks possible. Hence, a new pair of keys is computed and used. To alleviate this problem, we modify the PDP scheme [1] (later in this section) to use only one PDP instantiation, enabling us to homomorphically combine tags even from different levels into one, decreasing the proof size.

The general ECAL construction is as follows:

- $\texttt{LInit}(1^\lambda, 1^w, n, M)$: The client starts this protocol to initialize the empty buffers on the server, giving as input the security parameter λ, the block size w, the memory size n, and the initial data M. The client:
  - Sets $\texttt{ctr} = 0$.
  - Runs $(pk^*, sk^*) \leftarrow \texttt{CP.Kg}(1^\lambda)$ and shares pk^* with the server.
  - Puts the initial data M in a compact PoR: $C^* = \texttt{CP.St}(sk^*, M)$, and outsources C^* to the server.

  The server remembers the compact PoR's public key pk^*, and stores C^* in $\texttt{Buff}^*$.
- $\texttt{LAppend}(l)$: adds a new update log, l. The client:
  - Increases $\texttt{ctr}$ by one, i.e., $\texttt{ctr} = \texttt{ctr} + 1$.
  - Adds l to her local storage, `locbuff`. // `The client's local storage.`
  - If `locbuff` is full:
    * Computes the buffer number to put the new log into. This can be done for all constructions above. As an example, for equibuffers case with each level of size `buffsize`, she computes $j = \lceil \texttt{ctr}/(\texttt{buffsize} * \texttt{CodeRate}) \rceil$, where `CodeRate` is the code rate of `CP`. The current logs will be stored at $\texttt{buff}_j$.
    * Reads contents of $\texttt{buff}_j$ as C^j from the server, together with the associated authenticity information.
    * If the authenticity verification succeeds, decodes C^j to obtain L^j.
    * Adds the new logs (in `locbuff`) to the received logs according to their age: $L^j_{new} = L^j || \texttt{locbuff}$.
    * Runs $(pk_j, sk_j) \leftarrow \texttt{CP.Kg}(1^\lambda)$ to get generate the new keys for level j, and shares pk_j with server.
    * Performs the proper pre-computation again: $C^j_{new} = \texttt{CP.St}(sk_j, L^j_{new})$.
    * Outsources the encoded logs C^j_{new} to the server (to $\texttt{buff}_j$), and empties `locbuff`.

  The server is responsible for responding the clients read query for buffer j, and then replacing its contents with the new ones received from the client.
- $\texttt{LAudit}()$: The client specifies a challenge vector and sends it to the server to check whether the server keeps storing the logs correctly (it can also be used to read a log, e.g., the i^{th} log, by putting only i in the challenge vector). The important point is that the challenge should sample all logs to give the retrievability guarantee. The client, after verifying the server's answer, emits an acceptance or a rejection notification.
  - The client picks λ random locations from each buffer $\texttt{buff}_1, ..., \texttt{buff}_\texttt{t}$ and $\texttt{Buff}^*$, where t is the number of levels of `buff`, puts all in the challenge vector ch, and sends it to the server.
  - The server runs $(\sigma_i, \mu_i) \leftarrow \texttt{CP.P}(pk_i, \texttt{buff}_\texttt{i}, ch_i)$ for each buffer $\texttt{buff}_\texttt{i} \in \{\texttt{buff}_1, ..., \texttt{buff}_\texttt{t}, \texttt{Buff}^*\}$ to generate a proof of possession, where ch_i is a subset of ch containing indices of the blocks in $\texttt{buff}_\texttt{i}$. He sends all proofs $\{\sigma_i, \mu_i\}_{i=1}^t || \{\sigma^*, \mu^*\}$ to the client.
  - The client runs $\texttt{CP.V}(pk_i, sk_i, ch_i, \sigma_i, \mu_i)$ on all proofs received, and emits '`accept`' if all verified, or '`reject`' otherwise.

3.7 Optimized ECAL Construction

Communication-efficiency for audits. If all logs appear together, we can put them all inside a single compact PoR instantiation. But they grow gradually over the time, and we cannot simply add new logs to the existing ones inside a compact PoR instantiation. That is why we use the hierarchical buffer with each level as a separate compact PoR instantiation that generates separate (aggregated) proofs. These proofs cannot be aggregated further with those of the other levels, and this leads to increased communication.

To solve the problem, we change the tag generation algorithm of the PDP inside compact PoR in a way that we can handle adding later logs into the same PoR instantiation, without leading to replay attacks. Indeed, we can aggregate all challenged tags (of all buffers) into one, and reduce communication. We call the resulting scheme using this modified PDP the *modified compact PoR*. This immediately brings another advantage: the client needs to store only one pair of keys since there is only one PDP instantiation. This frees the client from key management difficulties.

In PDP [1], a random number v is generated and concatenated with the block number: $W_i = v||i$. W_i is used in tag generation to bind the generated tag to a specific PDP construction and the corresponding block. The tags are computed as $t_i = (h(W_i)g^{m_i})^d \ mod \ N$, for each block m_i. We use v differently, to bind each block to the last time the corresponding buffer was updated: $v_{\mathtt{buff_j}} = PRF_K(\mathtt{ctr})$. This way, on each update, a new (pseudo)random value is generated for the corresponding buffer, inside the same PDP construction. If a buffer is filled up, it becomes permanent and its tags will not be changed (until the next rebuild emptying all buffers).

The proof generation by the server remains the same. Our modification only affects the verification: the client should use the correct v values for each buffer. This, however, is not complicated since for each full buffer $\mathtt{buff}_j$, $\mathtt{CodeRate} * \Sigma_{i=0}^{j} |\mathtt{buff}_j|$ will be used as input to the PRF, where $\mathtt{CodeRate}$ is the code rate of the compact PoR in use. For the current working buffer, the largest value $\leq \mathtt{ctr}$ that is a multiple of the client local storage size is used. Thus, the client can easily compute the required v values for verification, and does not need to keep different keys locally.

The prime result of this modification is that the server can aggregate all challenged tags into one, and send it together with the corresponding hash value to the client. This reduces the proof size (and the client storage for keys) from $O(\lambda t)$ to $O(\lambda)$ in audits, where t is the number of levels of $\mathtt{buff}$.

Note that the modified compact PoR is not a fully dynamic scheme. It only supports append operation as the original PDP scheme [1]. Moreover, our modification does not affect its security. The same extractor works here, and the same security proof is applicable.

We compare the three configurations and select the optimized ones regarding the audit bandwidth. The server provides a proof for all audit commands coming from the client. Different configurations pose different proof sizes:

- **Linear**. There is only one modified compact PoR instantiation. $O(\lambda)$ blocks (and their tags) are accessed (and aggregated) to generate the proof. Thus, both the server computation and the communication are $O(\lambda)$.
- **Hierarchical**. There are t levels, and hence, t modified compact PoR instantiations. The challenge samples $O(\lambda)$ blocks at each level. The server accesses all challenged blocks (and their tags), leading to $O(\lambda t)$ computation. Using the stated optimization, the proofs of all levels can be aggregated together to reduce the proof size from $O(\lambda t)$ ([10, 27]) to $O(\lambda)$.

Client storage and preventing reshuffling. Reshuffling is used in ORAM [18] and ORAM-based dynamic PoR schemes [9, 27, 10] to obliviously transfer data (logs, in our case) from the smaller buffers into the larger ones. An immediate implication of reshuffling is that the client has to have a temporary memory to store and operate on the whole logs being transferred. However, the more logs are outsourced, the bigger temporary memory is required. This, on the other hand, states that these schemes does not suit the devices with limited amount of local memory.

- **Linear**: All the outsourced logs are put inside a single modified compact PoR scheme, forming a unit chunk of data. To append a new log(s), the whole logs should be downloaded, requiring a local memory that increases as more logs are outsourced (up to $O(n)$).
- **Incremental-buffers**: The outsourced logs are stored in a memory that is divided into t levels with increasing size, requiring reshuffling at some points. If the first empty level to store the new logs is j, then all levels $i<j$ should be transferred to the client and prepared to be stored on level j. This requires a local memory that is small at the outset, and increases as the new logs arrive, up to $O(n)$ when most of the small levels are full (when all full buffers are transferred into a buffer at some last level, the required memory becomes small, and increases again). Adding some permanent local storage will not alleviate the problem [34] as proven in Appendix A.
- **Equibuffers**: There are t levels, each of size n/t, that are all alike and used to store the logs, starting from one end, until all become full (in which case a rebuild is required). Once some buffers are filled up, their content will not be changed (until the next *rebuild* that occurs once in $O(n)$ updates). Therefore, at most one buffer of size $O(n/t)$ is required to be transferred back to the client, requiring $O(n/t)$ temporary local storage, and **no reshuffling is necessary during updates**. With $O(n/t)$ permanent local storage, the client accumulates $O(n/t)$ logs and outsource them altogether. This way, the redundant interactions to download and upload the half-empty buffers is eliminated, and the server will only receive and store a full buffer each time.

We observe that applying our modification on the equibuffers configuration, and using client local storage of size a level (at the server) achieves the best performance. The remaining thing is choosing t, as it directly affects the memory requirements and performance: the level capacity affects the update cost, and the number of levels affects the

Table 1: A comparison of different configurations of our scheme using the communication-efficient technique.

Configuration	Client storage	LAppend		LAudit	
		Server computation	Bandwidth	Server computation	Bandwidth
Linear	$O(1)$	$O(n)$	$O(\lambda n)$	$O(\lambda)$	$O(\lambda)$
	$O(n^{\delta})$	$O(n^{1-\delta})$	$O(\lambda n^{1-\delta})$	$O(\lambda)$	$O(\lambda)$
Incremental	$O(1)$	$O(\log(n))$	$O(\lambda \log(n))$	$O(\lambda \log(n))$	$O(\lambda)$
	$O(n^{\delta})$	$O(\log(n))$	$O(\lambda \log(n))$	$O(\lambda \log(n))$	$O(\lambda)$
Equibuffers	$O(1)$	$O(\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda)$
	$O(n^{\delta})$	$O(n^{1/2-\delta})$	$O(\lambda n^{1/2-\delta})$	$O(\lambda\sqrt{n})$	$O(\lambda)$
	$O(\sqrt{n})$	$O(1)$	$O(\lambda)$	$O(\lambda\sqrt{n})$	$O(\lambda)$

audit cost, directly. Having $t = \sqrt{n}$ levels, each of size $\sqrt{n}$, balances the update and audit costs. Therefore, the buff (of size n on the server) is divided into $\sqrt{n}$ buffers $\texttt{buff}_1, ..., \texttt{buff}_{\sqrt{n}}$, each of size $\sqrt{n}$. ($\texttt{Buff}^*$, of size n, stores the encoded version of the original data. This buffer is never appended, but is rebuilt once every $O(n)$ updates.)

Client local memory. With a local memory locbuff of size n^{δ} ($0 \leq \delta \leq 1/2$, up to the size of a level), the client accumulates the most recent logs and outsources them together, which decreases the (amortized) update complexity from $O(n^{1/2})$ to $O(n^{1/2-\delta})$. If $\delta = 1/2$, the whole client memory is sent to the server at once, who puts it on a level without further computation, and the (amortized) updates complexity will be $O(1)$. Moreover, there is no need to read some logs from the server and combine with those on the client side. For an outsourced file of size 10 GB, divided into 10×2^{20} blocks each of size 1 KB, the client stores at most $\sqrt{10} \times 2^{10}$ blocks locally, which corresponds to 3.16 MB. This amount of local memory is available in almost all today's smart mobile phones. Further, it is easy to transmit this amount of data using even GSM mobile networks.

Rebuild and reshuffling. Reshuffling is used in ORAM [18] and ORAM-based dynamic PoR schemes [9, 27, 10] to obliviously transfer logs from the smaller buffers into the larger ones. An immediate implication is that for handling an update, an $O(n)$ client memory is needed. Therefore, these schemes do not suit the devices with limited local memory. Our equibuffer configuration needs client memory in the size of a level only. Hence, given the local memory in the size of a level (e.g., $O(\sqrt{n})$), it does not require reshuffling anymore. However, all configurations require rebuild that needs $O(n)$ client memory. But it is performed only once in $O(n)$ updates, requiring a computer to be employed.

A comparison of the operation complexities for different ECAL configurations is given in Table 1. **The audit bandwidth is $O(\lambda)$ for all configurations**, using the modified compact PoR.

3.8 ECAL Security Proof

Theorem 3.1 *If* $\texttt{CP} = (\texttt{Kg}, \texttt{St}, \texttt{P}, \texttt{V})$ *is a secure compact PoR scheme [25], then* $\texttt{E} = (\texttt{LInit}, \texttt{LAppend}, \texttt{LAudit})$ *is a secure ECAL scheme according to definitions 3.2 and 3.3.*

Correctness immediately follows from the correctness of the underlying compact PoR scheme.

Authenticity is provided by the underlying compact PoR scheme. The linear buffer case is obvious. The incremental-buffers case was shown by Shi *et al.* [27]. We treat the equibuffers construction below, and then focus on our communication-efficient optimized version.

For the hierarchical equibuffers case, assume that there are t levels, and each level is a distinct compact PoR instantiation. If a PPT adversary $\mathcal{A}$ wins the ECAL authenticity game with non-negligible probability, we can use it to construct a PPT algorithm $\mathcal{B}$ who breaks the security of at least one of the compact PoR schemes used in one of the levels, with non-negligible probability. $\mathcal{B}$ acts as the challenger in the ECAL game with the adversary $\mathcal{A}$, and simultaneously, plays role of the server in compact PoR game played with the compact PoR challenger $\mathcal{C}$. He receives the public key of a compact PoR scheme from $\mathcal{C}$, and produces $t-1$ other pairs of compact PoR public and private keys himself. Then, he guesses some i and puts the received key in i^{th} position, and sends the t public keys to $\mathcal{A}$. From here on, $\mathcal{B}$ just keeps forwarding messages from $\mathcal{A}$ on the level i to $\mathcal{C}$, and vice versa. For other levels, he himself performs the operations. During the setup phase, $\mathcal{B}$ builds a local ECAL for herself that is invisible to the adversary $\mathcal{A}$, and thus will not affect his behavior. Finally, $\mathcal{A}$ selects a command, generates the answer and proof for the command, and sends them to $\mathcal{B}$.

When $\mathcal{A}$ wins the ECAL security game, if the guessed level i was the related compact PoR level, then $\mathcal{B}$ would also win the compact PoR security game. If the answer is different from the real answer in at least one level, while the proof is accepted, the adversary wins. $\mathcal{B}$ can realize it since she maintains a local copy. When $\mathcal{B}$ receives them, she selects the command, answer and proof parts for the i^{th} level, and forwards them to $\mathcal{C}$. If the guess of i was correct, then $\mathcal{B}$ would succeed. If $\mathcal{A}$ passes the ECAL verification with non-negligible probability p, then $\mathcal{B}$ passes the compact PoR verification with probability at least p/t.

Since we employ secure compact PoRs, p/t must be negligible, which implies that p is negligible, hence, $\mathcal{A}$ has negligible probability of winning the ECAL game. Our ECAL scheme is secure if the underlying compact PoRs are.

When the communication-efficient configuration is used, the reduction is even simpler (for both incremental and equibuffers configuration). All levels use the same key. The only difference is that, when $\mathcal{C}$ sends an append operation to $\mathcal{B}$, then $\mathcal{B}$ internally calculates the associated level i and sends the related append operation to $\mathcal{A}$. Further, observe that the only difference of our optimization from the original compact PoR is the tag calculation. Since we employ the same PRF idea, with just a slightly different input, this does not affect the security. Therefore, if $\mathcal{A}$ wins with probability p, $\mathcal{B}$ wins with the same probability.

Retrievability. We give a high level proof of retrievability here without going into details, since the proof is similar to the already-existing proofs [25, 19, 1, 9].

We reduce extractability of the incremental and equibuffers constructions to that of the compact PoR (the linear case is exactly a compact PoR). There are multiple compact PoR instances in both constructions. Due to the security of compact PoR, if the server manipulates more than $d-1$ blocks in some level, he will be caught with high probability (see [25]). Hence, if he can pass the verification, each level is extractable, which means that the portion of logs stored in that compact PoR instantiation is retrievable with overwhelming probability. Putting together all these PoR instantiations guarantees retrievability of the whole logs stored in these constructions with overwhelming probability.

For the communication-efficient configuration, the compact PoR extractability proof is again applicable [25], since changing the PRF input in the tag does not affect extractability. There is only one compact PoR in use, hence, its extractor works for this ECAL configuration as well.

4 Dynamic Proof of Retrievability

Since the ECAL stores the client data and guarantees its retrievability, it seems that the ECAL itself is a dynamic PoR scheme. However, due to the nature and application of the ECAL, reading a data block requires reading, decoding, and reconstructing all logs, necessitating $O(n)$ cost. Therefore, read is not an efficient operation in ECAL, and it should not normally be fulfilled through the ECAL. This means that retrieving through the ECAL should be the last resort when other options fail. *This is the important reason why ECAL is not an efficient DPoR scheme in its own.* If we do not care about read efficiency, we can use the ECAL as a DPoR scheme. Now, we are about to define our general efficient dynamic PoR framework.

Since access privacy is not a requirement in PoR definition [19], we can store the client data in a dynamic memory-checking scheme, e.g., DPDP [15], preserving its authenticity. Read operations will be handled through this memory-checking part. But, update operations will affect both the memory-checking part and the ECAL part. This will solve the read inefficiency problem of the ECAL.

This means that given a dynamic memory-checking scheme and a static PoR scheme, e.g., compact PoR [25], we can construct an efficient dynamic PoR scheme. Moreover, given an erasure-correcting code scheme and a static memory-checking scheme, e.g., PDP [1], (or any –homomorphic for efficiency– MAC or signature scheme), we can construct a static PoR [25]. Therefore, a dynamic PoR scheme can be constructed given black box access to a dynamic memory-checking scheme, an erasure-correcting code scheme, and a static memory-checking scheme.

4.1 Dynamic PoR using ECAL

According to our observations on making PoR dynamic, storing updates inside the data is neither efficient nor secure. Therefore, we store the updates inside an ECAL scheme (which aims to support retrievability), separately from the data. Moreover, we use the update logs differently from [10, 27], as they only support modification on the original data, but we support insertion and deletion, too. They only require the last version of data blocks to reconstruct the data (the number of blocks is fixed), but we need the whole logs.

The data itself is stored inside a dynamic memory-checking scheme (e.g., DPDP [15]) in plaintext form, since we apply later updates on and read through it. We refer to the DPDP part as '**D**' and the ECAL part as '**E**'. An informal description of dynamic PoR operations is given below.

- **Read** is used to retrieve the most up-to-date version of the data at a specific location. Since **D** maintains the last version of the data, read can be done through **D**. Moreover, **D** provides an authenticity proof (of size $O(\lambda \log n)$) for all data blocks read, that works as the proof of retrievability for those blocks.

- **Update** is performed on the outsourced data, and brings both **D** and **L** in an up-to-date state. Updating **D** requires $O(\lambda)$ communication and $O(\log n)$ server computation, and updating **L** depends on the underlying ECAL structure. The equibuffers structure with $O(\sqrt{n})$ buffers of size $O(\sqrt{n})$ each, and $O(\sqrt{n})$ client storage, performs updates with $O(\lambda)$ communication and $O(1)$ server computation (amortized).
- **Audit** challenges and checks authenticity of the outsourced logs to see if the server keeps storing them intact. It challenges λ random blocks from each non-empty buffer of **L** and verifies them. Since the challenge vector can bu generated by the server given the required keys, the client-to-server communication is $O(\lambda)$ [1]. Using the equibuffers setting, the server finds and aggregates the challenged blocks and their tags in $O(\lambda\sqrt{n})$ time. The proof includes two values, and is of size $O(\lambda)$. The client verification time is also $O(\lambda\sqrt{n})$.
- **Periodic rebuild**. Our scheme in the equibuffers setting **eliminates the reshuffling operation**, which is executed more frequently, and needs only the rare rebuilds. Moreover, we run the rebuild operation using the fresh data from **D** instead of combining and using the update logs, resulting in a much more efficient rebuild. The server sends the whole data from **D** to the client, which is an $O(n)$ operation. The client first verifies the whole data with DPDP. If accepted, this guarantees that it is the correct last version of the data, and the logs can be discarded completely. She then runs `LInit` and uploads the result. Hence, the communication and the client computation are also $O(n)$ at the worst case. Since this operation is executed once in every n updates, the amortized complexities will all be $O(1)$.

Definition 4.1 (Dynamic PoR) *A dynamic PoR scheme includes the following protocols (mostly from [9]) run between a stateful client and a stateful server. The client, using these interactive protocols, can outsource and later update her data at an untrusted server, while retrievability of her data is guaranteed (with overwhelming probability):*

- $\texttt{PInit}(1^\lambda, 1^w, n, M)$: given the alphabet $\Sigma = \{0,1\}^w$ and the security parameter λ, the client uses this protocol to initialize an empty memory of size n on the server, outsourcing there the initial data M.
- $\texttt{PUpdate}(i, OP, v)$: the client performs the operation $OP \in \{I, D, M\}$ on the i^{th} location of the memory (on the server) with input value v (if required).
- $(v, \pi) \leftarrow \texttt{PRead}(i)$: is used to read the value stored at the i^{th} location of the memory managed by the server. The client specifies the location i as input, and outputs some value v, and a proof π proving authenticity of v.
- $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{PAudit}()$: The client starts this protocol to check if the server keeps storing her data correctly. She emits an acceptance or a rejection signal.

4.2 Dynamic PoR Security Definitions

Since ECAL is a (inefficient) DPoR scheme, all its security definitions with proper protocol names are applicable here. In both games, the server $\widetilde{\mathcal{S}}$ asks the challenger to start a protocol execution (`PRead`, `PUpdate` or `PAudit`) by providing the required information.

4.3 DPoR Construction

Let $n, k \in \mathbb{Z}^+$ ($k<n$), and $\Sigma_l=\{0,1\}^w$ and $\Sigma_m=\{0,1\}^{w'}$ be two finite alphabets. The client is going to outsource a data $M=(m_1, ..., m_k) \in \Sigma_m^k$. She stores M inside a DPDP construction `D=(KeyGen,PrepareUpdate,Perform-Update,VerifyUpdate,Challenge,Prove,Verify)`. She also initializes an ECAL instantiation `E=(LInit, LAppend,LAudit)` to store the encoded logs. On each update, she updates both `D` and `E`, which support read and audit, respectively. Our dynamic PoR construction is as follows:

$\texttt{PInit}(1^\lambda, 1^w, n, M)$:

- The client runs $(pk, sk) \leftarrow \texttt{D.KeyGen}(1^\lambda)$ and shares pk with the server.
- The client runs $(e(M), e(\text{`}\texttt{full rewrite}\text{'}), e(st_c')) \leftarrow \texttt{D.PrepareUpdate}\ (sk, pk, M, \text{`}\texttt{full rewrite}\text{'}, st_c)$.
- The server runs $(M^1, st_s, st_c', P_{st_c'}) \leftarrow$`D.PerformUpdate`(pk, $e(M), e(\text{`}\texttt{full rewrite}\text{'}), e(st_c')$), where M^1 is the first version of the client data hosted by the server, and st_c' and $P_{st_c'}$ are the client's metadata and its proof, respectively, computed by the server, to be sent to the client.

- The client executes $\texttt{D.VerifyUpdate}(sk, pk, M, \text{‘}\texttt{full rewrite}\text{’}, st_c, st'_c, P_{st'_c})$, and outputs the corresponding acceptance or rejection notification.
- The client also stores the initial data using the ECAL: $\texttt{E.LInit}(1^\lambda, 1^w, n, M)$.

$\texttt{PUpdate}(i, OP, v)$:

- The client runs $(e(v), e(OP,i), e(st'_c) \leftarrow \texttt{D.PrepareUpdate}(sk, pk, v, (OP,i), st_c)$.
- The server runs $(m^j, st_s, st'_c, P_{st'_c}) \leftarrow \texttt{D.PerformUpdate}(pk, e(m^{j-1}), e(OP,i), e(st'_c))$, where m^{j-1} is the current version of the data on the server (to be updated into m^j). The server sends st'_c and $P_{st'_c}$ to the client.
- The client executes $\texttt{D.VerifyUpdate}(sk, pk, v, (OP,i), st_c, st'_c, P_{st'_c})$, and outputs the corresponding acceptance or rejection signal.
- The client, in parallel, prepares the corresponding log, $l =$‘$iOPv$’, and runs $\texttt{E.LAppend}(l)$.

$\texttt{PRead}(i)$:

- The client creates a DPDP challenge ch containing the block index i only, and sends it to the server. (Challenging only one block is a ‘read’ operation.)
- The server executes $P \leftarrow \texttt{D.Prove}(pk, m_j, st_s, ch)$ to generate and send the proof P (for the i^{th} block only).
- The client runs $\texttt{D.Verify}(sk, pk, st_c, ch, P)$ to verify the proof, and emits an acceptance or a rejection notification based on the result.
- If there was a problem reading from D, then she tries to read through the log structure E. In such a case, she needs to read the whole logs.
- If reading from E is not possible too, server misbehavior is detected. She goes to the arbitrator, e.g., [21].

$\texttt{PAudit}$:

- The client starts $\texttt{E.LAudit}()$. If it results in an acceptance, she outputs `accept`, otherwise, she outputs `reject` and contacts the arbitrator.

4.4 Dynamic PoR Security Proof

Theorem 4.1 *If* $\texttt{D} = (\texttt{KeyGen}, \texttt{PrepareUpdate}, \texttt{PerformUpdate}, \texttt{VerifyUpdate}, \texttt{Challenge}, \texttt{Prove}, \texttt{Verify})$ *is a secure DPDP scheme, and* $\texttt{E} = (\texttt{LInit}, \texttt{LAppend}, \texttt{LAudit})$ *is a secure ECAL scheme, then* $\texttt{DPoR} = (\texttt{PInit}, \texttt{PRead}, \texttt{PUpdate}, \texttt{PAudit})$ *is a secure DPoR scheme according to (the modified versions of) definitions 3.2 and 3.3.*

Proof 4.1 ***Correctness*** *of* DPoR *follows from the correctness of DPDP and ECAL. Since* DPoR *has nothing to do apart from DPDP and ECAL, if both of them operate correctly, any* $\texttt{PRead}(i)$ *will return the most recent version stored at the i^{th} location through DPDP, and all* PAudit *operations will lead to acceptance.*

Authenticity *of the plain data is provided by the underlying DPDP and ECAL schemes. Whenever a data is read, the underlying DPDP scheme sends a proof of integrity, assuring authenticity. When that fails, the logs will be read through ECAL, which also provides authenticity.*

In particular, if a PPT adversary $\mathcal{A}$ wins the DPoR authenticity game with non-negligible probability, we can use it in a straightforward reduction to construct a PPT algorithm $\mathcal{B}$ who breaks security of the underlying ECAL or DPDP schemes with non-negligible probability. Since both the ECAL and DPDP schemes are secure, the adversary has negligible probability of winning either of their respective games. Therefore, our DPOR is authentic supposed that the underlying ECAL and DPDP schemes are authentic.

Retrievability *immediately follows from retrievability of the ECAL. Since ECAL is secure, it guarantees retrievability of the logs that can be used to reconstruct and retrieve the plaintext data. We bypass the proof details as it is straightforward to reduce the retrievability of our dynamic PoR scheme to that of the underlying ECAL.*

4.5 Comparison to Previous Work

Investigating the *equibuffer* configuration of ECAL in detail and using the complexities in Table 1 reveals that the client storage (S_{client}), and the update and audit costs (C_{update} and C_{audit}) are related together via the following formula (ignoring factors depending on the security parameter): $S_{client} * C_{update} * C_{audit} = O(n)$.

This formula describes a nice trade-off between the client storage, and the update and audit costs, that can be used to design dynamic PoR schemes with different requirements. A similar statement is given by Cash *et al.* [9] for the

Table 2: A comparison of dynamic PoR schemes (comp. stands for computation, and the temporary memory is required at the client side to fulfill the updates. All schemes require a temporary memory of size $O(\lambda n)$ for rebuild.)

Scheme	**Client Storage**	**Read**		**Update**		**Audit**		**Temporary Memory**
		Server comp.	**Bandwidth**	**Server comp.**	**Bandwidth**	**Server comp.**	**Bandwidth**	
Cash *et al.* [9]	$O(\lambda)$	$O(\lambda \log^2 n)$	$O(\lambda \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda n)$
LULDC [10]	$O(\lambda)$	$O(\lambda \log^2 n)$	$O(\lambda \log^2 n)$	$O(\log n)$	$O(\log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda n)$
Shi *et al.* [27]	$O(\lambda)$	$O(\log n)$	$O(\lambda \log n)$	$O(\log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda^2 \log n)$	$O(\lambda n)$
Our scheme (Equibuffer)	$O(\lambda)$	$O(\log n)$	$O(\lambda \log n)$	$O(\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda)$	$O(\sqrt{n})$
	$O(\sqrt{n})$	$O(\log n)$	$O(\lambda \log n)$	$O(1)$	$O(\lambda)$	$O(\lambda\sqrt{n})$	$O(\lambda)$	$O(\lambda)$

linear configuration (in the Appendix A of their paper): for any $\delta > 0$, using blocks of size n^δ the complexity of read, update, and audit will be $O(1)$, $O(n^\delta)$, and $O(n^{1-\delta})$, respectively.

Our scheme covers the schemes given in [10, 27] as well. Using the incremental-buffers configuration together with MAC instead of PDP tags reduces our scheme to [27]. If, in addition, the incremental-buffers together with MAC is used to store the plain data as a memory-checking scheme instead of DPDP, the resulting scheme will be [10].

Our schemes pose important advantages over the previous work [9, 10, 27]. First, the bandwidth optimization makes the audit and (amortized) update bandwidth $O(\lambda)$, which are **optimal**. Second, our equibuffer configuration with reasonable amount of permanent client storage, i.e., $O(\sqrt{n})$ that is $\simeq$ 3 MB for an outsourced data of size 10 GB, makes possible using smart phones (and other hand-held electronic devices) for updating the outsourced data.

Table 2 represents a comparison among the dynamic PoR schemes. The server storage is $O(n)$ in all schemes. The operation complexities of our schemes are computed using the version with the equibuffer optimization and the audit bandwidth optimization applied on. These two optimizations can be applied independently. The bandwidth optimization, for instance, can be applied on top of previous work [27, 10] achieving optimal audit bandwidth in those configurations as well. The communication cost in our scheme, in different settings, is reduced to $O(\lambda)$. Therefore, we manage to obtain **the most general** and **efficient** DPoR construction known.

Acknowledgement

We would like to acknowledge the support of TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project numbers 112E115 and 114E487, as well as European Union COST Actions IC1206 and IC1306.

References

[1] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *CCS'07*. ACM, 2007.

[2] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *SecureComm*, page 9. ACM, 2008.

[3] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In *ASIACRYPT*, 2009.

[4] A. Barsoum and A. Hasan. Enabling dynamic data and indirect mutual trust for cloud computing storage systems. *Parallel and Distributed Systems, IEEE Transactions on*, 24(12):2375–2385, 2013.

[5] A. F. Barsoum and M. A. Hasan. Provable possession and replication of data over cloud servers. *Centre For Applied Cryptographic Research (CACR), University of Waterloo, Report*, 32:2010, 2010.

[6] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *ASIACRYPT*. Springer, 2001.

[7] K. Bowers, A. Juels, and A. Oprea. Hail: A high-availability and integrity layer for cloud storage. In *CCS'09*, pages 187–198. ACM, 2009.

[8] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: Theory and implementation. In *CCSW*, pages 43–54. ACM, 2009.

[9] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *EUROCRYPT'13*, pages 279–295. Springer, 2013.

[10] N. Chandran, B. Kanukurthi, and R. Ostrovsky. Locally updatable and locally decodable codes. In *TCC*, 2014.

[11] R. Curtmola, O. Khan, and R. Burns. Robust remote data checking. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 63–68. ACM, 2008.

[12] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS'08*, pages 411–420. IEEE, 2008.

[13] Y. Dodis, S. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*. Springer, 2009.

[14] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In *TCC*, pages 503–520. Springer, 2009.

[15] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS'09*, pages 213–222. ACM, 2009.

[16] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and O. Ozkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. Technical report, Cryptology ePrint Archive, Report 2013/645, 2013.

[17] M. Etemad and A. Küpçü. Transparent, distributed, and replicated dynamic provable data possession. In *ACNS*, 2013.

[18] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[19] A. Juels and B. S. Kaliski, Jr. Pors: proofs of retrievability for large files. In *CCS'07*, pages 584–597, New York, NY, USA, 2007. ACM.

[20] A. Küpçü. *Efficient cryptography for the next generation secure cloud: protocols, Proofs and Implementation*. Lambert Academic Publishing, 2010.

[21] A. Küpçü. Official arbitration with secure cloud storage application. *The Computer Journal*, 2013.

[22] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in) security of hash-based oblivious ram and a new balancing scheme. In *ACM SODA*. SIAM, 2012.

[23] Z. Mo, Y. Zhou, and S. Chen. A dynamic proof of retrievability (por) scheme with o(logn) complexity. In *IEEE ICC*, pages 912–916. IEEE, 2012.

[24] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology-ASIACRYPT 2008*, pages 90–107. Springer, 2008.

[25] H. Shacham and B. Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3), 2013.

[26] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious ram with o ((logn) 3) worst-case cost. In *ASIACRYPT*, pages 197–214. Springer, 2011.

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

[28] E. Stefanov, M. van Dijk, A. Juels, and A. Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238. ACM, 2012.

[29] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path oram: An extremely simple oblivious ram protocol. In *ACM CCS*, pages 299–310. ACM, 2013.

[30] R. Tamassia. Authenticated data structures. *Algorithms-ESA 2003*, pages 2–5, 2003.

[31] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou. Toward secure and dependable storage services in cloud computing. *IEEE Transactions on Services Computing*, 5(2):220–232, 2012.

[32] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[33] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In *ESORICS 2009*, pages 355–370. Springer, 2009.

[34] P. Williams and R. Sion. Usable pir. In *NDSS*, 2008.

[35] K. Zeng. Publicly verifiable remote data integrity. In *ICICS'08*, pages 419–434. Springer-Verlag, 2008.

[36] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy*, pages 237–248. ACM, 2011.

A The Impact of Client Storage

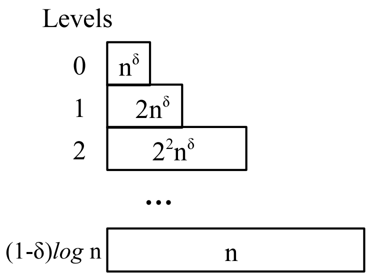

Figure 6: Server's memory layout when the client has n^δ local storage.

In the incremental-buffers configuration, adding local storage to the client will not change the update complexity asymptotically. Assume the client has a local storage of size n^δ used to keep updates locally, and send them all at once to the server (when becomes full). We can imagine the server's memory layout as in Figure 6. This represents a layout similar to Figure 2, but now each update operation carries a data of length n^δ that will be put in the first level buffer. The next update finds the first level full, merges its data with those in the first level and reshuffles them, and finally stores the result in the second level buffer (and empties the first level). This is repeated in a similar way as in the original configuration until the whole buffer becomes full. Stated differently, all operations are the same as the original incremental-buffers configuration, the only difference is the update data size.

There are $(1-\delta)\log(n)$ buffers, the first buffer of size n^δ and the last one of size $O(n)$. The first buffer will be written $n^{1-\delta}$ times, the second one $n^{1-\delta}/2$ times, and the last buffer one time. Now, we compute the number of update operation executions to update a total of n logs into the server:

$n^{1-\delta} * n^\delta + (n^{1-\delta}/2) * 2n^\delta + \ldots + 1 * n = n + n + \ldots + n = ((1-\delta)\log(n))n.$

Therefore, the amortized cost of a single update is $(1-\delta)\log(n) = O(\log(n))$, which is asymptotically same as the case where the client had constant local storage.