Generic Efficient Dynamic Proofs of Retrievability

Mohammad Etemad
metemad@ku.edu.tr
Crypto Group, Koç University
İstanbul, Turkey

Alptekin Küpçü
akupcu@ku.edu.tr
Crypto Group, Koç University
İstanbul, Turkey

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 difference among PDP and PoR models were greatly debated. The first DPDP scheme was shown by Erway *et al.* in 2009, while the first DPoR scheme was created by Cash *et al.* in 2013. We show how to obtain DPoR from DPDP, PDP, and erasure codes, making us realize that even though we did not know it, we could have had a DPoR solution in 2009.

We propose a general framework for constructing DPoR schemes that encapsulates known DPoR schemes as its special cases. We show practical and interesting optimizations enabling better performance than Chandran *et al.* and Shi *et al.* constructions. For the first time, we show how to obtain constant audit bandwidth for DPoR, 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 $\sim$ 3MB for 10GB outsourced data, and can easily be obtained in today's smart phones, let alone computers.

1. INTRODUCTION

In a data outsourcing scheme, the client expects an *authenticated storage* and *guaranteed retrievability* [6, 22]. The former means she wants each data access to return the correct value; i.e., the most recent version of data written by the client herself. The latter means the client wants to make sure 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.

The authenticated storage can be provided easily by computing a digest (e.g., hash) of data and keeping it locally after transferring data to the server. But, the client needs to download the whole data and check it against the digest to investigate the authenticity of her data, which is prohibitive given current trends of outsourcing tens of gigabytes of data, even by home users.

Static techniques. Juels and Kaliski [16] proposed the first scheme, called *proofs of retrievability* (PoR), to provide such a storage. Before outsourcing her data, the client encodes it with an erasure-correcting code (ECC), which brings some redundancy while ensuring that the server should manipulate a significant part of the outsourced (encoded) data to impose a data loss or corruption. However, such a misbehavior will be caught with a very high probability.

Dynamic techniques. Early PoR schemes support only *static* data [16, 21, 11] and do not support *efficient* updates on the outsourced data. In fact, secure and efficient update is the main problem with PoR schemes.

Cash *et al.* [6] provided the first efficient and secure dynamic PoR scheme using the Oblivious RAM (ORAM) [15]. Later improvements [22, 8], 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.* [12] in 2009; four years before the first dynamic PoR. The reason is that, since PDP-type schemes do not employ erasure-correcting codes, the abovementioned 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. While the PoR guarantees retrieving the *whole* data, PDP only guarantees the client can retrieve *most* of the outsourced data. Though erasure-correcting codes help providing full retrievability, they bring the above problems (discussed in more detail in Section 2).

CCSW'16, October 28 2016, Vienna, Austria
 ISBN 978-1-4503-4572-9/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2996429.2996439

Our contributions. We analyze security requirements of a dynamic PoR scheme, propose a generic dynamic PoR scheme construction framework encompassing existing schemes as special cases, and propose various optimizations.

1.1 Related Work

PoR was first proposed by Juels and Kaliski [16] for *static* data, supporting only a *limited* number of challenges. Shacham and Waters [20] gave the first PoR schemes fully secure against arbitrary adversaries, supporting *unlimited* number of challenges. Bowers *et al.* [5] proposed a theoretical framework for PoR design. Dodis *et al.* generalized the static PoR schemes [11].

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 [29, 26] and availability [4, 10]. Curtmola *et al.* [9] integrated PDP with ECC to enhance the possession guarantee in *robust data possession* scheme.

Dynamic PDP. 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. Erway *et al.* [12] proposed the first fully dynamic PDP scheme with $O(\log(n))$ update and audit complexity, where n is the number of blocks. Later variants supply additional properties [3], use other data structures [27, 25], distribute and replicate [14], or enhance efficiency [13]. The DPDP scheme definition is given in Appendix A.

Dynamic PoR. Zhen *et al.* [19] and Zheng and Xu [30] claimed to give dynamic PoR schemes, but actually dynamic PDP schemes were given. Similarly, the emphasis of [27] is on data integrity while retrievability is claimed. The main reason is that, in our opinion, the difference between PDP and PoR was not understood well at that point. We also contribute in this regard.

Stefanov *et al.* [23] proposed Iris as a dynamic PoR scheme. However, Iris is not a fully-dynamic PoR scheme as it stores the erasure-coding data *locally* (on a trusted party called the *portal*).

The first really dynamic PoR scheme with full security definition and proof was proposed by Cash *et al.* [6, 7]. 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*.

In their dynamic PoR scheme, Chandran *et al.* [8] erasure-coded the data, and outsourced it inside a hierarchical data structure similar to ORAM. Later updates are also erasure-coded and stored in the same data structure. Reading through that structure requires $O(n)$ cost, hence, they stored the plain data and subsequent updates in another similar structure.

Shi *et al.* [22] proposed an efficient dynamic PoR scheme similar to [8]. The scheme is later improved by outsourcing some parts of computation to the server, reducing the communication and client computation.

Relationship among these schemes. Shacham and Waters [21] showed *a PoR scheme can be obtained by employing a PDP scheme with erasure-correcting codes*. We will employ their result in our paper.

The first dynamic PDP schemes [12, 27], providing authenticity, appeared in 2009, while the first dynamic PoR schemes [6, 8, 22], with retrievability guarantee, were proposed in 2013. By analyzing their differences and similarities, 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 can be built using PDP and erasure-correcting codes [21]). This interestingly reveals that we could have had DPoR schemes when the first DPDP schemes appeared.

1.2 Preliminaries

We use $x \leftarrow \mathrm{X}$ to denote x is sampled uniformly from the set X, $|\mathrm{X}|$ 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.

Erasure-correcting codes deal with correcting errors that 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 (`encode`, `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 $\texttt{decode} : \Sigma^{n-d+1} \rightarrow \Sigma^k$ recovers the original message from a codeword in the presence of at most $d-1$ erasures.

A **PoR** scheme is defined as follows [16, 21]:

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

2. OVERVIEW

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.

- **Observation 1**. To ensure retrievability of the outsourced data, erasure-correcting codes are 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 [16, 21, 9, 23, 8].
- **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 [12, 9, 23]. Note that the integrity checking mechanism needs to detect only such *significant* manipulations.
- **Observation 3**. A simple update on erasure-coded data is not enough [6]:
  - If a single update affects a small part of the encoded data (i.e., a locally updatable code), the server learns and later can erase the whole update without a high probability of getting caught. This option is *not secure*.
  - If a single update affects a huge part (the whole) of the encoded data, requiring $\sim O(n)$ cost, the server learns almost nothing about the update locations and cannot attack them. But such updates are *not efficient*.
- **Observation 4**. Therefore, it is better to store the update information separately, rather than applying the updates on the encoded data. Thus, there should be two parts of server storage: one part stores the encoded original data, and the other (the *log store*) stores the update information. 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 it holds will be applied on the original data. This *rebuild* operation generates the latest version of data and empties the log store [8, 22].
- **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 [6, 8, 22]. Each level is erasure-coded, updated, audited, and possibly merged into the next level via *reshuffling*.
- **Observation 6**. To read the latest version of some data, we need to decode the encoded data and apply all the logs, which is an $O(n)$ operation. Therefore, an uncoded version of data, protected by a dynamic memory checking scheme [8, 22], is stored at the server. This frees the read operation from 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 read data is kept intact on the server [12]). The memory checking scheme must also be *dynamic* to support updates.
- **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 [22]. The log store, on the other hand, needs oblivious operations, due to the observations above. We also observe that *the log store can be append-only*, since update logs are only appended with each update, unlike the data protected by the memory checking scheme, since the uncoded data is actually updated.
- **Observation 8**. ORAM performs both update and read operations in a similar way that an adversary cannot distinguish them. However, as our read operations will not be run through the ORAM, there is *no need to perform the extra heavy reshufflings*. This is an important observation that we utilize to construct an efficient structure for storing the logs.

Overview of Our Solution. 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.

Our scheme has two parts: a data structure (an authenticated log scheme given in Section 3) for keeping update logs securely and providing the retrievability guarantee, and a dynamic memory-checking scheme (e.g., DPDP [12]) 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), 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 pre-

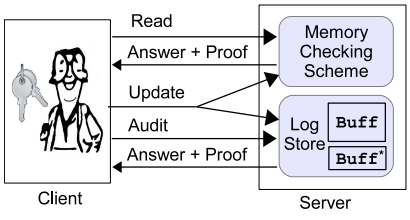

Figure 1: Our model.

pares the corresponding command and directs it to the dynamic 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), which is detected via the proof, 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 to be verifiable.

3. ERASURE-CODED AUTHENTICATED LOG

The log store plays an important role in our scheme. It is an authenticated data structure that inspects integrity and guarantees retrievability of the logs, which in turn, guarantee retrievability of the outsourced data. We propose an *Erasure-Coded Authenticated Log* (*ECAL*) scheme to serve these goals. It erasure-codes the logs (to guarantee retrievability) and garbles the result (e.g., by encrypting and permuting the data randomly) to make locating any part of original data difficult for the server.

Any scheme supplying *retrievability* and *authenticity* can be used to store the update logs. In fact, what we need is a *(static)* PoR scheme with efficient append and audit, without caring about read efficiency (it is used rarely, only for lost data recovery). A hierarchical memory where each level employs a (possibly distinct) instance of PoR is preferred for efficiency.

Each update on the outsourced data leaves a log. Regardless of the update locations on the 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 check integrity of logs over time, helping her induce that either the data is fully retrievable, or the server is misbehaving. The definitions below follow closely those of [6].

DEFINITION 3.1 (**ECAL**). *An erasure-coded authenticated log scheme includes the following PPT 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 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 already-stored logs.
- $\texttt{accept/reject} \leftarrow \texttt{LAudit}()$: The client specifies a challenge vector and uses this protocol to check if 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`. If the client has some initial data, she can send them using `LInit` as 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.

3.1 ECAL Security Definitions

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} log 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 logs exists, it returns $\perp$. Moreover, the `LAudit` protocol 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 $\in \{\texttt{LAppend}, \texttt{LAudit}\}$ 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.

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

Retrievability. We want the ECAL to guarantee that if a malicious adversary performs more than $d-1$

erasures within some level (Each level may have a different d parameter if they have different sizes.), he should not pass the subsequent audit: If a malicious adversary passes the audit with a non-negligible probability, 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 retrieva- bility game among a challenger $\mathcal{C}$, an extractor, and a malicious server $\widetilde{\mathcal{S}}$ is:

- **Initialization**. $\mathcal{C}$ acts as an honest client and starts the `LInit` with $\widetilde{\mathcal{S}}$.
- **Setup**. $\widetilde{\mathcal{S}}$ adaptively asks $\mathcal{C}$ to start a protocol $\in \{\texttt{LAppend}, \texttt{LAudit}\}$ by providing the required information. $\mathcal{C}$ acts as the client in those protocols. $\widetilde{\mathcal{S}}$ 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. Call the final version of the *uncoded* data, considering all `LAppend` operations performed, $\mathcal{D}$.
- **Challenge**. The extractor repeats the `LAudit` protocol polynomially-many times with $\widetilde{\mathcal{S}}$ in final state $st_{\widetilde{\mathcal{S}}}$ (via rewinding). Call the extracted data $\mathcal{D}'$.

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 we have $\mathcal{D}' = \mathcal{D}$ with overwhelming probability.*

The ECAL security proofs are given in Appendix C.

3.2 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. 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 client's initial data. The other memory, `Buff`, stores the logs of the subsequent updates on the data, and its efficiency affects the whole scheme.

The client initializes a PoR instance `P=(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`* and will not be changed until the next rebuild. Later, she performs updates on the original data, and outsources their logs at the `Buff` in a way that the adversary cannot locate or differentiate them form the already-existing logs. The content of `Buff` changes as new logs are outsourced. The client wants to rest assured that 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, $O \in \{I, D, M\}$ denotes insert/delete/modify, 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,1D)* brings M to the final state M='*ftsyk*'.

The *age* of an update log is the time elapsed since the log is arrived, i.e., the log arrived first is the oldest one, and the log arrived most recently is the youn- gest 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 PoR instance, which generates an encoded version of the logs $C = (C_1, C_2, ..., C_n) \in \Sigma_l^n$ to be outsourced. The authenticity and retrievability of the logs are handled by the 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.

Once in every $O(n)$ updates, when the `Buff` becomes full, a rebuild operation applies all logs in the `Buff` on the data stored at `Buff`*. It empties the whole `Buff`, puts the latest version of the data in a new PoR instance, 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 [6, 8, 22] due to the existence of memory-checking scheme, which 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 data received from the memory checking part, and if accepted, puts it inside a new PoR instance and outsources the result again at `Buff`*. Lastly, a rebuild requires $O(n)$ temporary client storage, but since it is very rare, it can be done on a computer, rather than a mobile device.

3.3 The `Buff` Configurations

Linear configuration. `Buff` can be, in the simplest form, a one-dimensional buffer (of length n) storing the output of the PoR instance constructed over the logs. The client stores a counter `ctr` to keep the size of 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), the client should download all the existing outsourced logs, decode it to retrieve the plaintext logs, append the new log, initialize a new PoR instance to put the logs inside, and upload the result. This requires a local memory that increases with the log size, up to $O(n)$. Although the audit is an $O(\lambda)$ operation, this construction suffers from the same efficiency problem as the original static PoR: to prevent

the server from tampering with the recently-added parts of the logs, a small update affects all the outsourced logs. The amortized server computation and bandwidth (after n updates) is: $(1 + 2 + ... + n)/n = O(n)$.

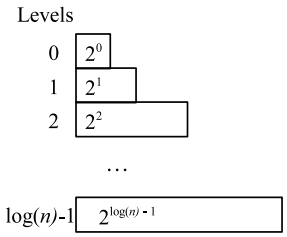

Figure 2: Inc.-buffers.

Incremental-buffers configuration. There is a sequence of buffers whose sizes grow at a geometric rate [18] and are organized in a way that the buffer at the i^{th} level stores 2^i elements, as shown in Figure 2. If the total size of the (encoded) log memory is n, then 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 discuss later), then the logs stored at all levels $i < j$ are read by the client, the new log is appended, all are put in a new PoR instance, the result is stored at level j, and all levels $i < j$ are emptied. This requires a local memory equal to the sum of all levels $i < j$ that increase up to $O(n)$ when the small levels are full. With each update, a buffer is re-encoded (in a new PoR instance), making it hard for the adversary to correlate the new content to the old one, or to locate the newly-added logs. Such an operation is called *reshuffling*, and the best known overhead for an oblivious reshuffling operation is $O(\log^2(n)/\log\log(n))$ [18, 24].

Equibuffers configuration. 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 a basic 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 [28] and shown in Appendix B. Third, managing a buffer divided into levels of different size is hard for the client.

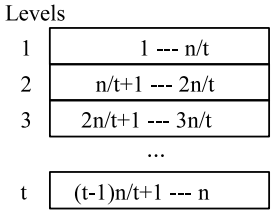

Figure 3: Equibuffers.

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, as represented in Figure 3. 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, and we can fill them up from one end. In essence, we have t linear PoR instances, each holding up to n/t encoded log blocks.

Once some buffers are filled up, their content will not be changed (until the next *rebuild* that occurs once in $O(n)$ updates). Hence, at most one buffer of size $O(n/t)$ is required to be transferred 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 can keep and outsource $O(n/t)$ logs together. The redundant interactions to download and upload half-empty buffers are eliminated, and the server only receives and stores a full buffer each time.

In both these hierarchical configurations with t levels, the challenge samples $O(\lambda)$ blocks at each level. The server accesses all challenged blocks (and their tags), leading to $O(\lambda t)$ computation and communication.

Another important advantage of the equibuffers 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 cost from $O(n/t)$ to $O(n^{1-\delta}/t)$. Setting $\delta = 1/2$ and $t = \sqrt{n}$ achieves a very good balance: $\sqrt{n}$ levels each of size $\sqrt{n}$. Each time, the whole client memory is sent to the server, 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 the local logs. 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 mobile phones. It is easy to transmit this amount of data using even mobile networks.

Shi *et al.* [22] also suggested to keep a local storage of size $O(\log(n))$ to accumulate update logs and outsource them all together to achieve $O(1)$ amortized update cost. However, the worst case computation, communication, and client temporary storage all are $O(n)$. While, with $O(\sqrt{n})$ local storage, the worst case computation and communication costs of our equibuffers configuration are $O(\sqrt{n})$, and we do not need temporary storage at the client side. (They do not have an audit optimization as what we propose in the next section.)

Rebuild vs. reshuffle. Reshuffling is used to obliviously transfer logs from smaller buffers into the larger ones in the incremental-buffers scheme, whereas it is never employed in the equibuffers scheme. An immediate implication of this reshuffling is that the client needs a temporary memory to store and operate on the whole logs being transferred. The more logs are outsourced, the bigger temporary memory is required. This states that those schemes do not suit devices with limited amount of local memory. A rebuild, on the other hand, applies to all ECAL configurations, but it is necessary only once every $O(n)$ updates.

A comparison of the ECAL configurations is given in Table 1, where $t = \sqrt{n}$ for equibuffer configuration.

Table 1: A comparison of different ECAL schemes regarding LAppend and LAudit. ('S. Comp.' stands for server computation, and $O(\lambda)$ multipliers omitted for simplicity.)

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

3.4 ECAL Protocols

In both incremental- and equibuffers configurations, the client's local state contains the keys of the PoR instances 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. Each level is treated as a separate PoR instance, with a different pair of keys stored at the client, requiring a separate proof from the server. The general ECAL construction is as follows:

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

  The server keeps the PoR's public key pk^*, and stores C^* in $\texttt{Buff}^*$.
- $\texttt{LAppend}(l)$: appends a new update log l. The client:
  - Increases ctr by one, i.e., ctr++.
  - Adds l to her local storage, locbuff.
  - 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 containing buffsize many logs, she computes $j = \lceil \texttt{ctr}/\texttt{buffsize} \rceil$. The current logs will be stored at $\texttt{buff}_j$.
    - Reads contents of $\texttt{buff}_j$ as C^j from the server, with its authenticity proof. (For incremental-buffers, reads all buffers up to j.)
    - If the authenticity verification succeeds, decodes C^j to obtain L^j.
    - Adds the new logs according to their age: $L^j_{new} = L^j || \texttt{locbuff}$.
    - Runs $(pk_j, sk_j) \leftarrow \texttt{P.Kg}(1^{\lambda})$, and shares pk_j with the server.
    - Re-encodes the logs via PoR: $C^j_{new} = \texttt{P.St}(sk_j, L^j_{new})$.
    - Outsources the encoded logs C^j_{new} (to $\texttt{buff}_j$), and empties locbuff.
- LAudit(): The client specifies a challenge vector and sends it to the server to check whether he 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 $\texttt{buff}_1, ..., \texttt{buff}_t$ and $\texttt{Buff}^*$, puts all in the challenge vector ch, and sends it to the server.[1]
  - The server runs $(\sigma_i, \mu_i) \leftarrow \texttt{P.P}(pk_i, \texttt{buff}_i, ch_i)$ for each buffer $\texttt{buff}_i \in \{\texttt{buff}_1, ..., \texttt{buff}_t, \texttt{Buff}^*\}$ to generate a proof of possession, where t is the number of levels of Buff and ch_i is a subset of ch containing indices in $\texttt{buff}_i$. He sends all proofs $\{\sigma_i, \mu_i\}_{i=1}^{t} || \{\sigma^*, \mu^*\}$ to the client.
  - The client runs $\texttt{P.V}(pk_i, sk_i, ch_i, \sigma_i, \mu_i)$ on all proofs received, and emits 'accept' if all verified, and 'reject' otherwise.

3.5 Achieving Constant Audit Bandwidth

In both the incremental- and equibuffers configurations, each level uses a different PoR key. Therefore, the proofs of different levels cannot be aggregated further, leading to increased communication. We give an optimization to achieve constant audit bandwidth that is applicable only on PDP-based schemes, e.g., compact PoR [21]. The idea is to change the PDP tag generation algorithm in a way that different levels can use the same PoR instance, without leading to replay attacks. Indeed, we can aggregate all challenged tags (of all buffers) into one, and reduce communication to $O(\lambda)$. This immediately brings another advantage: the client needs to store only one pair of keys since there is only one PoR instance.

In PDP [1], a random number v is generated and concatenated with the block number i to obtain $W_i = v||i$,

[1]The client may send a PRF key instead, to simplify communication (see e.g., [1]).

which is used in tag generation to bind the generated tags to a specific PDP instance (identified by v) and the corresponding blocks (identified by i). The tags are computed as $t_i = (h(W_i)g^{m_i})^d \ mod \ N$, for each block m_i, where $N = pq$ is an RSA modulus with p and q as two safe primes, g is a generator of quadratic residues QR_N, and d is the secret key.

We use a different v for each buffer that is the output of a pseudo-random function F, given the last `ctr` value as input: $v_{\texttt{buff}_j} = F_K(\texttt{ctr})$. This binds each block to the last time the respective buffer was updated. When some new logs of size s are being appended to a half-empty buffer $\texttt{buff}_j$, the client increases `ctr` to $\texttt{ctr}+s$, reads and decodes contents of $\texttt{buff}_j$, appends new logs, computes the new v value as $v_{\texttt{buff}_j} = F_K(\texttt{ctr}+s)$, generates the respective tags (for the whole $\texttt{buff}_j$), and outsources the result. This process goes on until $\texttt{buff}_j$ is filled up, after which, its content becomes permanent and its tags will not be changed in the equibuffers case (until the next rebuild emptying all buffers). The process will be repeated with the next empty buffer $\texttt{buff}_{j+1}$, afterwards.

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 $\texttt{buff}_j$, $\Sigma_{i=0}^{j}|\texttt{buff}_j|$ will be used as input to the PRF. For the current working buffer, the largest value $\leq$ `ctr` that is a multiple of the client local storage size is used. For incremental-buffers configuration, v can be calculated based on the counter.

The prime result of this modification is that, since all levels use the same PoR key, the server can aggregate all challenged tags into one and send it 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 `Buff`. Note that the modified 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 and extractability. We use pseudorandom v values instead of random values. The same extractor works here, and the same security proof is applicable.

4. DYNAMIC PROOF OF RETRIEVABILITY

Since ECAL stores the client data and guarantees its retrievability, it seems 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. Retrieving through the ECAL should be the last resort when other options fail. *This is the reason why ECAL is not an efficient DPoR scheme in its own.*

Since access privacy is not a requirement in PoR definition [16], we can store the client data in a dynamic memory-checking scheme, e.g., DPDP [12], preserving its authenticity. Read operations will be handled through this memory-checking part. But, update operations will affect both the memory-checking and ECAL parts. This solves the read efficiency problem.

Given a static PoR and a dynamic memory-checking scheme, we can construct an efficient dynamic PoR scheme. Moreover, given an erasure-correcting code and a static memory-checking scheme, we can construct a static PoR [21]. Hence, 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. Note that except our audit bandwidth optimization that is not black-box, all other solutions presented work in a black-box manner.

DEFINITION 4.1 (**Dynamic PoR**). *A dynamic PoR scheme includes the following protocols (mostly from [6]) 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:*

- $\texttt{PInit}(1^\lambda, 1^w, n, M)$: given the alphabet $\Sigma = \{0,1\}^w$ and the security parameter λ, the client uses this protocol to initialize a 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).
- $v \leftarrow \texttt{PRead}(i)$: 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 obtains some value v and a proof π proving authenticity of v from the server. If the proof is accepted, it outputs v, and $\perp$ otherwise.
- $\texttt{accept/reject} \leftarrow \texttt{PAudit}()$: The client starts this protocol to check if the server keeps her data correctly. She emits an acceptance or a rejection signal.

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 $\in \{\texttt{PRead}, \texttt{PUpdate}, \texttt{PAudit}\}$ by providing the required information.

4.1 DPoR Construction

Let $n, k \in \mathbb{Z}^+$ (k<n), and Σ_l=$\{0,1\}^w$ and Σ_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,PerformUpdate,VerifyUpdate,Challenge,Prove,Verify)`, and initializes an ECAL instance `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:
PInit$(1^\lambda, 1^w, n, M)$:

- The client runs $(pk, sk) \leftarrow$ D.KeyGen(1^λ) and shares pk with the server.
- The client runs $(e(M), e(\text{`full rewrite'}), e(st'_c)) \leftarrow$ D.PrepareUpdate$(sk, pk, M, \text{`full rewrite'}, st_c)$.
- The server runs $(M^1, st_s, st'_c, P_{st'_c}) \leftarrow$D.PerformUpdate (pk,$e(M), e(\text{`full rewrite'}), e(st'_c))$, where M^1 is the first version of the client data, 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 D.VerifyUpdate$(sk, pk, M, \text{`full rewrite'}, st_c, st'_c, P_{st'_c})$, and outputs the corresponding acceptance or rejection notification.
- The client also stores the initial data using the ECAL: E.LInit$(1^\lambda, 1^w, n, M)$.

PUpdate(i, OP, v):

- Client runs $(e(v), e(OP, i), e(st'_c)) \leftarrow$ D.PrepareUpdate$(sk, pk, v, (OP, i), st_c)$.
- The server runs $(M^j, st_s, st'_c, P_{st'_c}) \leftarrow$ 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 returns st'_c and $P_{st'_c}$.
- The client executes D.VerifyUpdate$(sk, pk, v, (OP, i), st_c, st'_c, P_{st'_c})$, and outputs the corresponding acceptance or rejection signal.
- The client prepares the corresponding log, $l =$'$iOPv$', and runs E.LAppend(l).

PRead(i):

- The client creates a DPDP challenge containing the block index i only, and sends it to the server. (Challenging only one block is a 'read'.)
- The server executes $P \leftarrow$ D.Prove(pk, M^j, st_s, i) to generate and return P.
- The client runs D.Verify(sk, pk, st_c, i, P) to verify the proof.
- If it is accepted, outputs the value read.
- Otherwise, she tries to read through E. She needs to read the whole logs.
- If reading from E was successful, she outputs the value read.
- Otherwise, misbehavior of the server is detected. She output $\perp$, and goes to the arbitrator, e.g., [17].

PAudit:

- The client starts E.LAudit() and outputs the result.

4.2 Dynamic PoR Security Proof

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

PROOF. **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, then any 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 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 either the ECAL scheme or the DPDP scheme (or both of them) with non-negligible probability. Since both the ECAL and DPDP schemes are secure, the adversary has negligible probability of winning either of them. 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 uncoded 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. □

4.3 Comparison to Previous Work

Investigating the *equibuffer* configuration of ECAL in detail and the complexities in Table 1 reveals that the client storage (S_C), and the update and audit costs of the server (C_U and C_A) are related together via the following formula (ignoring factors depending on the security parameter): $S_C \times C_U \times C_A = O(n)$.

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.* [6] for the linear configuration (in the Appendix A of their paper): for any $\delta > 0$, using client storage 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 in [8, 22]. Using the incremental buffers together with MAC instead of PDP tags reduces our scheme to [22]. If, in addition, the incremental-buffers together with MAC is used to store

Table 2: A comparison of dynamic PoR schemes ('S. comp.' stands for 'Server computation', and 'BW' is used for 'bandwidth'. All schemes require a temporary memory of size $O(\lambda n)$ for rebuild.)

Scheme	Client Storage	Read		Update		Audit		Update temp. memory
		S. comp.	BW	S. comp.	BW	S. comp.	BW	
Cash *et al.* [6]	$O(\lambda)$	$O(\lambda \log^2 n)$	$O(\lambda \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda^2 \log^2 n)$	$O(\lambda n)$
LULDC [8]	$O(\lambda)$	$O(\lambda \log^2 n)$	$O(\lambda \log^2 n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda n)$
Shi *et al.* [22]	$O(\lambda)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda^2 \log n)$	$O(\lambda n)$
Our scheme	$O(\lambda)$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda\sqrt{n})$	$O(\lambda)$	$O(\lambda\sqrt{n})$
(Equibuffer)	$O(\lambda\sqrt{n})$	$O(\lambda \log n)$	$O(\lambda \log n)$	$O(\lambda)$	$O(\lambda)$	$O(\lambda\sqrt{n})$	$O(\lambda)$	$O(\lambda)$

the uncoded data as a memory-checking scheme instead of DPDP, the resulting scheme will be [8].

Our schemes pose important advantages compared to the previous work [6, 8, 22]. 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 ~ 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 securely.

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 audit bandwidth optimization, for instance, can be applied on top of previous work [22, 8] to achieve optimal audit bandwidth. 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 114E487 and 112E115, and European Union COST Actions IC1206 and IC1306.

5. 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] A. Barsoum and A. Hasan. Enabling dynamic data and indirect mutual trust for cloud computing storage systems. *IEEE TPDS'13*, 24(12):2375–2385, 2013.

[4] 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.

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

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

[7] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. *Journal of Cryptology*, 2015.

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

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

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

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

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

[13] E. Esiner, A. Küpçü, and Ö. Özkasap. Analysis and optimization on flexdpdp: A practical solution for dynamic provable data possession. In *ICC*, 2014.

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

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

[16] 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.

[17] A. Küpçü. Official arbitration with secure cloud storage application. *The Computer Journal*, 2015.

[18] 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.
[19] 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.
[20] H. Shacham and B. Waters. Compact proofs of retrievability. In *Advances in Cryptology-ASIACRYPT 2008*, pages 90–107. Springer, 2008.
[21] H. Shacham and B. Waters. Compact proofs of retrievability. *Journal of cryptology*, 26(3), 2013.
[22] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336. ACM, 2013.
[23] 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.
[24] 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.
[25] 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.
[26] 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.
[27] 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.
[28] P. Williams and R. Sion. Usable pir. In *NDSS*, 2008.
[29] K. Zeng. Publicly verifiable remote data integrity. In *ICICS'08*, pages 419–434. Springer-Verlag, 2008.
[30] 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.

APPENDIX

A. DPDP SCHEME DEFINITION

The **DPDP** scheme is defined as follows [12]:

$(sk, pk) \leftarrow \texttt{KeyGen}(1^\lambda)$: run by the client to generate the secret and public key pair (sk, pk), given the security parameter λ 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 $e(M)$.

$(F_i, M_i, M'_c, P_{M'_c}) \leftarrow \texttt{PerformUpdate}(pk, F_{i-1}, M_{i-1}, e(F), e(info), e(M))$: 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, and the client metadata and its proof (M'_c and P'_{M_c}).

$\texttt{accept/reject} \leftarrow \texttt{VerifyUpdate}(sk, pk, F, info, M_c, M'_c, P_{M'_c})$ 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, 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 F_i and metadata M_i, and the challenge c, the server generates a proof P to be sent to the client.

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

B. THE IMPACT OF CLIENT STORAGE

We show that 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 she uses to keep update logs locally, and send them all at once to the server (when it becomes full). We can imagine the server's memory layout as in Figure 4.

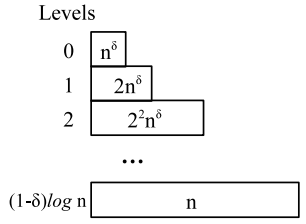

Figure 4: Server's memory layout when the client has n^δ local storage.

This represents a similar layout to the 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, 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:

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

Hence, 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 storage.

C. ECAL SECURITY PROOF

THEOREM C.1. *If* $\mathtt{CP} = (\mathtt{Kg}, \mathtt{St}, \mathtt{P}, \mathtt{V})$ *is a secure PoR scheme,* $\mathtt{E} = (\mathtt{LInit}, \mathtt{LAppend}, \mathtt{LAudit})$ *is a secure ECAL scheme according to definitions 3.2 and 3.3.*

Correctness immediately follows from that of the underlying PoR scheme.

Authenticity is provided by the underlying PoR scheme. The linear buffer case is obvious. The incremental-buffers case was shown by Shi *et al.* [22]. We treat the equibuffers construction below, and then focus on our communication-efficient optimized version.

For the equibuffers case, assume that there are t levels, and each level is a distinct PoR instance. 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 PoR instances 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 PoR game played with the PoR challenger $\mathcal{C}$. He receives the public key of a PoR scheme from $\mathcal{C}$, and produces $t-1$ other pairs of 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 the other levels, he himself performs the operations.

When $\mathcal{A}$ wins the ECAL security game, if the guess of i was correct, $\mathcal{B}$ would also win the PoR security game. Thus, if $\mathcal{A}$ passes the ECAL verification with non-negligible probability p, then $\mathcal{B}$ passes the PoR verification with probability at least p/t, where t is the number of levels, which is polynomial in the security parameter. Since we employ secure PoRs, p/t must be negligible, which implies that p is negligible, hence, $\mathcal{A}$ has negligible probability of winning the ECAL game. Therefore, our ECAL scheme is secure if the underlying PoRs are.

When the communication-efficient configuration is used, the reduction is even simpler (for both incremental- and equi-buffers). 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 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 proof sketch of retrievability without going into details, since the full proof is similar to the already-existing proofs [21, 16, 1, 6].

We reduce extractability of the incremental- and equi-buffers constructions to that of the PoR (the linear case is exactly a PoR). There are multiple PoR instances in incremental- and equi-buffers constructions. Due to the security of PoR, if the server manipulates more than $d-1$ blocks in some level, he will be caught with high probability (see [21]). Hence, if he can pass the verification, each level is extractable, which means that the portion of logs stored in that PoR instance is retrievable with overwhelming probability. Putting together all these PoR instances guarantees retrievability of the whole logs stored in these constructions with overwhelming probability.

For the communication-efficient configuration, the PoR extractability proof is again applicable [21], since changing the PRF input in the tag does not affect extractability (which only depends on obtaining linearly-independent combinations of data items). The original reduction to the RSA assumption and PRF indistinguishability also remains unaffected [1]. There is only one PoR in use whose extractor works for this ECAL configuration as well.