Practical Dynamic Proofs of Retrievability

Elaine Shi
University of Maryland
elaine@cs.umd.edu

Emil Stefanov
UC Berkeley
emil@cs.berkeley.edu

Charalampos Papamanthou
University of Maryland
cpap@umd.edu

ABSTRACT

Proofs of Retrievability (PoR), proposed by Juels and Kaliski in 2007, enable a client to store n file blocks with a cloud server so that later the server can prove possession of all the data in a very efficient manner (i.e., with constant computation and bandwidth). Although many efficient PoR schemes for *static* data have been constructed, only two *dynamic* PoR schemes exist. The scheme by Stefanov *et al.* (ACSAC 2012) uses a large of amount of client storage and has a large audit cost. The scheme by Cash *et al.* (EUROCRYPT 2013) is mostly of theoretical interest, as it employs Oblivious RAM (ORAM) as a black box, leading to increased practical overhead (e.g., it requires about 300 times more bandwidth than our construction).

We propose a dynamic PoR scheme with constant client storage whose bandwidth cost is comparable to a Merkle hash tree, thus being very practical. Our construction outperforms the constructions of Stefanov *et al.* and Cash *et al.*, both in theory and in practice. Specifically, for n outsourced blocks of β bits each, writing a block requires $\beta + O(\lambda \log n)$ bandwidth and $O(\beta \log n)$ server computation (λ is the security parameter). Audits are also very efficient, requiring $\beta + O(\lambda^2 \log n)$ bandwidth. We also show how to make our scheme publicly verifiable, providing the first dynamic PoR scheme with such a property. We finally provide a very efficient implementation of our scheme.

Categories and Subject Descriptors

K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

Keywords

dynamic proofs of retrievability; PoR; erasure code

1. INTRODUCTION

Storage outsourcing (e.g., Amazon S3, Google Drive) has become one of the most popular applications of cloud computing, offering various benefits such as economies of scale and flexible accessibility. When the cloud *storage provider* (also referred to as *server*) is untrusted, an important challenge is how to offer provable outsourced storage guarantees. In particular, a *data owner* (also referred to as *client*) wishes to obtain the following guarantees:

- **Authenticated storage.** The client wishes to verify that data fetched from the server is correct, where correctness is equivalent to *authenticity* and *freshness*;
- **Retrievability.** The client needs assurance that the server is indeed storing all of the client's data, and that no data loss has occurred.

Proofs of Retrievability (PoR), initially defined and proposed by Juels and Kalisky [14], are designed to offer the above guarantees for storage outsourcing applications, while requiring small client-side state.

Most existing PoR schemes [7–9, 20, 26] however, are impractical due to the prohibitive costs associated with data updates or client storage. A recent PoR construction by Cash *et al.* [8] shows how to achieve asymptotic efficiency in the presence of updates with constant client storage—however, their scheme relies on Oblivious RAM (ORAM), a rather heavy-weight cryptographic primitive. Recent works on ORAM [21,24,25,30,31] have shown that even the fastest known constructions incur a large bandwidth overhead in practice. In particular, under typical parametrizations, and with $O(1)$ amount of client storage, known ORAM constructions require **400+** blocks be transferred between a client and a server for a single data access. In contrast, **our construction requires transferring only about 1.05 to 1.35 blocks per data access**, which makes it both practical and orders of magnitudes more efficient than the scheme of Cash *et al.* [8].

This paper proposes a light-weight dynamic PoR construction that achieves *comparable bandwidth overhead and client-side computation with a standard Merkle hash tree*, reducing the above cost dramatically. Specifically, for each read and write, our construction requires transferring $O(\lambda \log n)$ bits of cryptographic information (independent of the block size) in addition to transferring the block itself, where λ is the security parameter, and n is the total number of outsourced blocks. To understand the implications of this, note that a Merkle hash tree offers authenticated storage, but does not offer retrievability guarantees. This paper essentially shows that under settings where the client-server bandwidth is the bottleneck (as is often the case in practice—e.g., Bitcoin and various file sharing applications [1]), we can make PoR almost as efficient as a Merkle hash tree, i.e., ready to be deployed in practical applications today.

CCS'13, November 4–8, 2013, Berlin, Germany.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2477-9/13/11 ...$15.00.
http://dx.doi.org/10.1145/2508859.2516669.

Scheme	Client storage	Write cost		Audit cost		Verifiability
		Server cost	BW	Server cost	BW	
Iris [26]	$O(\beta\sqrt{n})$	$O(\beta)$	$O(\beta)$	$O(\beta\lambda\sqrt{n})$	$O(\beta\lambda\sqrt{n})$	secret
Cash *et al.* [8]	$O(\beta)$	$O(\beta\lambda(\log n)^2)$	$O(\beta\lambda(\log n)^2)$	$O(\beta\lambda(\log n)^2)$	$O(\beta\lambda(\log n)^2)$	secret
This paper (Section 5)	$O(\beta)$	$O(\beta \log n)$	$\beta + O(\lambda \log n)$	$O(\beta\lambda \log n)$	$\beta + O(\lambda^2 \log n)$	secret
This paper (Appendix)	$O(\beta\lambda)$	$O(\beta \log n)$	$\beta(1+\epsilon) + O(\lambda \log n)$ for any constant $\epsilon > 0$	$O(\beta\lambda \log n)$	$O(\beta\lambda \log n)$	public

β: block size (in terms of number of bits). λ: the security parameter. n: an upper bound on the number of blocks.
Server cost: includes the server-side disk I/O cost, and server computation.
BW: The client-server bandwidth cost.

Table 1: Comparison with existing dynamic PoR. All schemes have $O(\beta n)$ server side storage. Reads can be handled by keeping a separate, Merkle-hashed, up-to-date copy of the dataset; therefore each read costs $\beta + O(\lambda \log n)$. Numbers in this table are optimized for the typical case $\beta \geq \lambda$. We note that for the ORAM-based scheme of Cash *et al.*, we omit $\log \log n$ terms from the denominator.

In terms of disk I/O overhead on the server, our scheme also achieves asymptotic improvements for reads, writes, as well as audits, in comparison with the state-of-the-art scheme [8] (see Table 1). In our scheme, reads cost no more than a Merkle hash tree in terms of server disk I/O. Writes incur moderate server disk I/O: the server needs to read, write, and compute on $O(\log n)$ blocks for each write. However, our algorithms for writing access blocks *sequentially*, significantly reducing the disk seeks required for writes. We have a full-fledged, working implementation of our scheme and report detailed experimental results from a deployment. We also plan to open source our code in the near future.

We also point out that due to the blackbox application of ORAM, the scheme by Cash *et al.* [8] additionally offers access pattern privacy which we do not guarantee. In applications that demand access privacy, ORAM is necessary. We observe that the definition of PoR itself does not require access privacy, and when one requires only PoR guarantees (i.e., authenticated storage and retrievability) but not access privacy, a blackbox application of ORAM is impractical—in fact, as this paper shows, one can design truly practical PoR schemes when access privacy is not required. Table 1 summarizes the asymptotic performance of our scheme in comparison with related work.

1.1 Related Work

Comparison with proofs of data possession. A closely related line of research is called Proofs of Data Possession (PDP), initially proposed by Ateniese *et al.* [5]. We stress that PDP provides much weaker security guarantees than PoR. A successful PoR audit ensures that the server maintains knowledge of *all* outsourced data blocks; while a PDP audit only ensures that the server is storing *most* of the data. With PDP, a server that has lost a small number of data blocks can pass an audit with significant probability[1]. Erway *et al.* [10] recently demonstrated a dynamic PDP scheme with $\beta + O(\lambda \log n)$ cost for reads and writes, and $\beta + O(\lambda^2 \log n)$ cost for audits. *We stress that we provide the much stronger PoR guarantees,* roughly at the same practical and asymptotic overhead as dynamic PDP schemes.

[1]While some PDP schemes [6] achieve full security, *they require that the server read all of the client's data during an audit*, and thus is impractical.

Proofs of retrievability. Static proofs of retrievability were initially proposed by Juels and Kaliski [14], and later improved in a series of subsequent works [7–9, 17, 20, 26, 28, 32]. While some works [17, 28, 32] aim to achieve PoR, they essentially only achieve the weaker PDP guarantees when they wish to support dynamic updates efficiently.

The first dynamic proofs of retrievability construction was proposed by Stefanov, van Dijk, Juels, and Oprea as part of a cloud-based file system called Iris [26]. Although reads and writes in Iris are quite efficient, it requires $O(\lambda\beta\sqrt{n})$ bandwidth, server computation, and server I/O to perform an audit (it also requires $\beta\sqrt{n}$ local space). Cash *et al.* [8] proposed a dynamic POR scheme with constant client storage based on Oblivious RAM, but it requires $O(\beta\lambda(\log n)^2)$ bandwidth and server I/O to perform writes and audits. In contrast with these works, our scheme requires $\beta + O(\lambda \log n)$ write bandwidth, $\beta + O(\lambda^2 \log n)$ audit bandwidth, and constant storage.

2. INFORMAL TECHNICAL OVERVIEW

2.1 Previous Approaches

In this section we describe various approaches that could be used for the problem of dynamic proofs of retrievability and we highlight the problems of these approaches.

Strawman. We start from the most straightforward approach. Imagine that the client attaches a Message Authentication Code (MAC) to every block before uploading it to the server—to additionally ensure freshness under updates, one can use a Merkle hash tree instead of MACs. During the audit protocol, the client randomly samples a small number of blocks and ensures that the server possesses them by checking them against the MACs.

In fact, this approach illustrates the underlying idea of several Proof of Data Possession (PDP) schemes [10, 29]. The drawback of such an approach is that if the server has lost a small number of blocks (e.g., a $o(1)$ fraction), it can still pass the audit with significant probability.

Prior work: use of redundant encoding to boost detection probability. To address the aforementioned issue, prior PoR schemes [7, 9, 14, 20, 26] rely on erasure codes to boost the detection probability, and ensure that the server must possess all blocks to pass the audit test, which typically involves checking the authenticity of λ random code blocks, where λ is the security parameter. As a concrete example, suppose that the client outsources a total of n blocks, which are erasure coded into $m = (1 + c)n$ blocks for some constant $0 < c \leq 1$, such that knowledge of any n blocks suffices to recover the entire dataset. In this way, the server has to delete at least cn blocks to actually incur data loss—however, if the server deletes that many blocks, it will fail the above audit protocol with overwhelming probability (specifically with probability at least $1 - 1/(1 + c)^\lambda$).

Difficulty with updates. The issue with using erasure codes is that if the client wishes to update a single block, it has to additionally update cn parity blocks—this can be very expensive in practice.

One (failed) attempt to support updates (as described by Cash *et al.* [8]) is to use a *local* erasure coding scheme, i.e., each block only affects a small number of codeword blocks. For this approach to work, all the codeword blocks need to be randomly permuted, such that the server does not learn which codeword blocks correspond to one original data block—since otherwise, the server can selectively delete all codeword blocks corresponding to one block (say block i). Since the server has deleted only a small number of blocks, it can pass the audit with significant probability—however, in this case, block i would become irrecoverable.

The above approach fails to address the efficient update problem, since whenever the client updates a block, it reveals to the server which codeword blocks are related to this block—and this allows the server to launch the selective deletion attack as described above. To address this issue, Cash *et al.* [8] propose to employ ORAM to hide the access patterns from the server during updates. However, as mentioned in Section 1, known ORAM schemes incurs 400X or higher bandwidth overhead under typical parameterizations and constant client-side storage, and thus is impractical.

2.2 Our Idea

As before, suppose the client has n blocks, which are erasure-coded into $n + cn$ blocks for some small constant $c > 0$—we denote the erasure coded copy of data as $\mathbf{C}$. Now if the client needs to update a block, we encounter the issue that the client needs to update all of the cn parity blocks.

Our idea is to avoid the need to immediately update the cn parity blocks upon writes. Instead, the client will place the newly updated block into an erasure-coded log structure denoted $\mathbf{H}$, containing recently written blocks. During the audit, the client will sample blocks to check not only from the buffer $\mathbf{C}$, but also from the log structure $\mathbf{H}$. Note that since the buffer $\mathbf{C}$ does not get updated immediately upon writes, it may contain stale data—however, an up-to-date snapshot of all blocks is recoverable from the combination of $\mathbf{C}$ and $\mathbf{H}$, both of which are probabilistically checked during the audit. Two questions however remain:

1. How can reads be supported efficiently if the location of the up-to-date copy of a block is undetermined—since it can either exist in the buffer $\mathbf{C}$*, or in the log structure* $\mathbf{H}$*?*

The answer to this first question is relatively straightforward: one can always keep a separate, up-to-date, and memory-checked copy of all blocks just to support efficient reads. The client can verify the integrity (i.e., authenticity and fresheness) of the reads facilitated by the memory checking scheme (i.e., a Merkle hash tree). In our basic construction described in Section 4, this separate copy is denoted with $\mathbf{U}$. Section 5 describes further optimizations to reduce the server-side storage by a constant factor.

2. How can the log structure $\mathbf{H}$ *be efficiently updated upon writes?*

The answer to the latter question is much more involved, and turns out to be the main technical challenge we needed to overcome. Intuitively, when the log structure $\mathbf{H}$ is small, updating it upon writes should be relatively efficient. However, as $\mathbf{H}$ grows larger, updating it becomes slower. For example, in the extreme case, when $\mathbf{H}$ grows to as large as $\mathbf{C}$, updating $\mathbf{H}$ upon a write would cause an overhead roughly proportional to cn, as mentioned above.

Perhaps unsurprisingly, in order to achieve efficient amortized cost for updating $\mathbf{H}$ upon writes, we use a hierarchical log structure that is reminiscent of Oblivious RAM constructions [11]. In our construction, the log structure $\mathbf{H}$ consists of exactly $\lfloor \log n \rfloor + 1$ levels of exponentially growing capacity, where level i is an erasure coded copy of 2^i blocks. At a very high level, every 2^i write operations, level i will be rebuilt. Finally, the erasure-coded copy $\mathbf{C}$ can be informally (and a bit imprecisely) thought of as the top level of the hierarchical log, and is rebuilt every n write operations.

Despite the superficial resemblance to ORAM, our construction is *fundamentally different from using ORAM as a blackbox*, and thus orders of magnitude more efficient, since 1) we do not aim to achieve access privacy, or rely on access privacy to prove our PoR guarantees like in the scheme by Cash *et al.* [8]; and 2) each level of our hierarchical log structure $\mathbf{H}$ is erasure-coded. For this reason, we need a special erasure coding scheme that can be incrementally built over time (see Section 4 for details).

3. PRELIMINARIES

We begin with the definition of a dynamic PoR scheme, as given by Cash, Küpçü, and Wichs [8]. A dynamic POR scheme is a collection of the following four protocols between a stateful client $\mathcal{C}$ and a stateful server $\mathcal{S}$.

$(st, \bar{\mathcal{M}}) \leftarrow \mathsf{Init}(1^\lambda, n, \beta, \mathcal{M})$: On input the security parameter λ and the database $\mathcal{M}$ of n β-bit-size entries, it outputs the client state st and the server state $\bar{\mathcal{M}}$.

$\{B, \mathsf{reject}\} \leftarrow \mathsf{Read}(i, st, \bar{\mathcal{M}})$: On input an index $i \in [n]$, the client state st and the server state $\bar{\mathcal{M}}$, it outputs $B = \mathcal{M}[i]$ or reject.

$\{(st', \bar{\mathcal{M}}'), \mathsf{reject}\} \leftarrow \mathsf{Write}(i, B, st, \bar{\mathcal{M}})$: On input an index $i \in [n]$, data B, the client state st and the server state $\bar{\mathcal{M}}$, it sets $\mathcal{M}[i] = B$ and outputs a new client state st' and a new server state $\bar{\mathcal{M}}'$ or reject.

$\{\mathsf{accept}, \mathsf{reject}\} \leftarrow \mathsf{Audit}(st, \bar{\mathcal{M}})$: On input the client state st and the server state $\bar{\mathcal{M}}$, it outputs accept or reject.

Depending on whether the client state st in the above definition must be kept secret or not, we say that the dynamic PoR scheme is *secretly* or *publicly* verifiable.

3.1 Security Definitions

We define security, namely *authenticity* and *retrievability* in the same way as Cash *et al.* [8].

Authenticity. The authenticity requirement stipulates that the client can always detect (except with negligible probability) if any message sent by the server deviates from honest behavior. We use the following game between a challenger $\mathcal{C}$, a malicious server $\mathcal{S}^*$ and an honest server $\mathcal{S}$ for the adaptive version of authenticity, in the same way as Cash *et al.* [8].

- $\mathcal{S}^*$ chooses initial memory $\mathcal{M}$. $\mathcal{C}$ runs $\mathsf{Init}(1^\lambda, n, \beta, \mathcal{M})$ and sends the initial memory layout $\bar{\mathcal{M}}$ to $\mathcal{S}^*$. $\mathcal{C}$ also interacts with $\mathcal{S}$ and sends $\bar{\mathcal{M}}$ to $\mathcal{S}$.
- For polynomial number of steps $t = 1, 2, \ldots, \mathsf{poly}(\lambda)$, $\mathcal{S}^*$ picks an operation op_t where operation op_t is either $\mathsf{Read}(i, st, \bar{\mathcal{M}})$ or $\mathsf{Write}(i, B, st, \bar{\mathcal{M}})$ or $\mathsf{Audit}(st, \bar{\mathcal{M}})$. $\mathcal{C}$ executes the protocol with both $\mathcal{S}^*$ and $\mathcal{S}$.

$\mathcal{S}^*$ is said to win the game, if at any time, the message sent by $\mathcal{S}^*$ differs from that of $\mathcal{S}$, and $\mathcal{C}$ did not output reject.

DEFINITION 1 (AUTHENTICITY). *A PoR scheme satisfies adaptive authenticity, if no polynomial-time adversary $\mathcal{S}^*$ has more than negligible probability in winning the above security game.*

Retrievability. Intuitively, the retrievability requirement stipulates that whenever a malicious server can pass the audit test with non-negligible probability, the server must know the entire memory contents $\mathcal{M}$; and moreover, one is able to extract $\mathcal{M}$ by repeatedly running the Audit protocol with the server. Formally, retrievability is defined with a security game as below, in the same way as Cash *et al.* [8].

- **Initialization phase.** The adversary $\mathcal{S}^*$ chooses the initial memory $\mathcal{M}$. The challenger runs $\mathsf{Init}(1^\lambda, n, \beta, \mathcal{M})$, and uploads the initial memory layout $\bar{\mathcal{M}}$ to $\mathcal{S}^*$.
- **Query phase.** For $t = 1, 2, \ldots, \mathsf{poly}(\lambda)$, the adversary $\mathcal{S}^*$ adaptively chooses an operation op_t where op_t is of the form $\mathsf{Read}(i, st, \bar{\mathcal{M}})$, $\mathsf{Write}(i, B, st, \bar{\mathcal{M}})$, or $\mathsf{Audit}(st, \bar{\mathcal{M}})$. The challenger executes the respective protocols with $\mathcal{S}^*$. At the end of the query phase, suppose the state of the challenger and the adversary is $state_C$ and $state_S$ respectively and the final state of the memory contents is $\mathcal{M}$.
- **Challenge phase.** The challenger now gets blackbox rewinding access in the configuration $state_S$. The challenger runs the Audit protocol repeatedly for a polynomial number of times with the server $\mathcal{S}^*$, starting from the configuration $(state_C, state_S)$. Denote with $\pi_1, \pi_2, \ldots, \pi_{\mathsf{poly}(\lambda)}$ the transcripts of all the successful Audit executions.

DEFINITION 2 (RETRIEVABILITY). *A PoR scheme satisfies retrievability, if there exists a polynomial-time extractor algorithm denoted $\mathcal{M}' \leftarrow \mathsf{Extract}(state_C, \{\pi_i\})$, such that for any polynomial-time $\mathcal{S}^*$, if $\mathcal{S}^*$ passes the Audit protocol with non-negligible probability, then after executing the Audit protocol with $\mathcal{S}^*$ for a polynomial number of times, the $\mathsf{Extract}$ algorithm can output the correct memory contents $\mathcal{M}' = \mathcal{M}$ except with negligible probability.*

We note here that in the above definition, "correct memory contents" are the contents of the memory $\mathcal{M}$ after the end of the query phase in the security game.

3.2 Building Blocks

Erasure codes. Our construction makes use of erasure codes [27] as defined below.

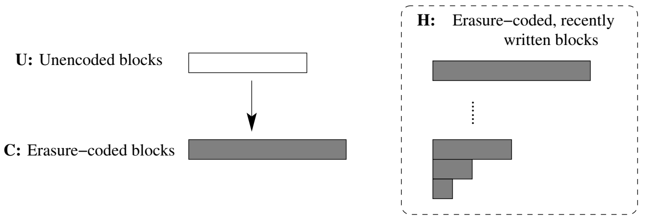

Figure 1: Server-side storage layout.

DEFINITION 3 (ERASURE CODES). *Let Σ denote a finite alphabet. An $(m, n, d)_\Sigma$ erasure code is defined to be a pair of algorithms $\mathsf{encode} : \Sigma^n \to \Sigma^m$, and $\mathsf{decode} : \Sigma^{m-d+1} \to \Sigma^n$, such that as long as the number of erasures is bounded by $d - 1$, decode can always recover the original data. A code is* maximum distance separable *(MDS), if $n + d = m + 1$.*

Authenticated structures. For describing our basic construction, we can assume that we use a standard Merkle hash tree [16] to ensure the authenticity and freshness of all blocks (encoded or unencoded, stored across buffers $\mathbf{U}$, $\mathbf{C}$ and $\mathbf{H}$). However, later, we will show that in fact, only the raw buffer $\mathbf{U}$ needs to be verified with a Merkle hash tree; whereas the erasure-coded copy $\mathbf{C}$ and the hierarchical log structure $\mathbf{H}$ only require time/location-dependent MACs to ensure authenticity and freshness. This will lead to significant savings (specifically, by a multiplicative $\log n$ factor).

4. BASIC CONSTRUCTION

To begin with, assume the data to be outsourced contains n blocks, and each block is from an alphabet Σ. For simplicity, we will first assume that n is determined in advance—we will later explain in Section 6 how to expand or shrink the storage.

We first describe a basic construction that requires the client to perform $O(\beta \log n)$ computation for each block written (recall β is the size of the block). Later in Section 5 we will describe how to rely on homomorphic checksums to reduce this cost to $\beta + O(\log n)$.

4.1 Server-Side Storage Layout

The server-side storage is organized in three different buffers denoted with $\mathbf{U}$ (stands for *unencoded*), $\mathbf{C}$ (stands for *coded*) and $\mathbf{H}$ (stands for *hierarchical*). We now explain in detail the function of these buffers (see also Figure 1).

Raw buffer. All up-to-date blocks are stored in original, unencoded format in a buffer called $\mathbf{U}$. Reads are performed by reading the corresponding location in $\mathbf{U}$. Writes update the corresponding location in the buffer $\mathbf{U}$ immediately with the newly written block. However, unlike reads, writes also cause updates to a hierarchical log as explained later.

Erasure-coded copy. In addition, we store an (m, n, d) erasure-coded copy of the data in a buffer $\mathbf{C}$, where $m = \Theta(n)$, and $d = m - n + 1 = \Theta(n)$, i.e., the code is maximum distance separable. The buffer $\mathbf{C}$ does not immediately get updated upon writes, and therefore may contain stale data.

Hierarchical log of recent writes. A hierarchical log structure denoted $\mathbf{H}$ stores recently overwritten blocks in erasure-coded format. $\mathbf{H}$ contains $k + 1$ levels, where $k = \lfloor \log n \rfloor$. We denote the levels of $\mathbf{H}$ as $(\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_k)$.

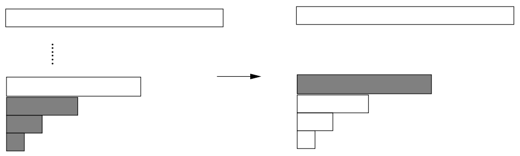

Figure 2: Rebuilding of level $\mathbf{H}_3$. When a newly written block is added to the hierarchical log structure $\mathbf{H}$, consecutive levels $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$ are filled. A rebuild operation for $\mathbf{H}_3$ occurs as a result, at the end of which $\mathbf{H}_3$ is filled, and $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$ are empty. Unlike ORAM schemes that employ oblivious sorting for the rebuilidng, our rebuilding algorithm involves computing linear combinations of blocks.

Each level $\ell \in \{0, 1, \ldots, k\}$ is an erasure code of some 2^ℓ blocks. For every block in $\mathbf{H}$, we define its age to be the time elapsed since when the block was written. Level $\mathbf{H}_0$ contains the most recently written block; and the age of the blocks contained in level $\mathbf{H}_\ell$ increases with ℓ. Particularly $\mathbf{H}_k$ (if filled) contains the oldest blocks.

Note that in practice, it is possible that the client keeps writing the same block, say B_i where $i \in [n]$. In this case, the hierarchical log structure $\mathbf{H}$ contains multiple copies of B_i. (as we will see later, duplicates are suppressed when the erasure-coded copy $\mathbf{C}$ is rebuilt.) Particularly, every time a block is written, the new value of the block along with its block identifier is added to the hierarchical log structure $\mathbf{H}$.

4.2 Operations

After describing the three types of buffers that we use to organize the original blocks, we are ready to give the high level intuition of the operations of our construction.

Reading blocks. In order to read a block, we read it directly from $\mathbf{U}$. Along with the block, the server returns the respective Merkle hash tree proof, allowing the client to verify the authenticity and the freshness of the block.

PoR audits. A PoR audit involves the following checks:

1. Checking the authenticity of $O(\lambda)$ random blocks from the erasure-coded copy $\mathbf{C}$;
2. Checking the authenticity of $O(\lambda)$ random blocks from each filled level $\mathbf{H}_\ell$, where $\ell \in [k]$.

Writing blocks and periodic rebuilding operations. Every write not only updates the raw buffer $\mathbf{U}$, but also causes the newly written block to be added to the hierarchical log structure $\mathbf{H}$ (specifically, the new block is always added into $\mathbf{H}_0$). Whenever a block is added, assume that levels $\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_\ell$ are consecutively filled levels where $\ell < k$, and level $\ell + 1$ is the first empty level. This leads to the rebuilding of level $\mathbf{H}_{\ell+1}$. At the end of this rebuilding: 1) level $\mathbf{H}_{\ell+1}$ contains an erasure code of all blocks currently residing in levels $\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_\ell$ and the newly added block; and 2) level $\mathbf{H}_0, \mathbf{H}_1, \ldots \mathbf{H}_\ell$ are emptied (see Figure 2).

This rebuilding technique has been previously used in various ORAM schemes (e.g., [25]). However, unlike ORAM schemes that require oblivious sorting for the rebuilding, our rebuilding process requires just computing an erasure code of the new level. This is a lot simpler since it just involves computing linear combinations of blocks (we employ a linear coding scheme). We will later show that it takes $O(\beta \cdot 2^\ell)$ time to rebuild level ℓ ($\ell = 0, 1 \ldots, \lfloor \log n \rfloor$), where β is the block size. Since each level ℓ is rebuilt every 2^ℓ write operations, the amortized rebuilding cost per write is $O(\beta \log n)$. In comparison, ORAM schemes [12, 13, 15] require roughly $O(\beta (\log n)^2 / \log \log n)$ or higher amortized cost for the rebuilding due to oblivious sorting.

For now, we assume that the client is performing the rebuilding for simplicity. Later in Section 5, we will show how to have the server perform the rebuilding computations, and the client instead simply verifies that the server adheres to the prescribed behavior. This will allow us to further reduce the bandwidth overhead from $O(\beta \log n)$ to $\beta + O(\lambda \log n)$.

Periodic rebuilding of the erasure coded copy. Finally, every n write operations, the erasure coded copy $\mathbf{C}$ is rebuilt, and all levels $\mathbf{H}_0, \mathbf{H}_1, \ldots, \mathbf{H}_k$ are emptied as a result. Recall that the log structure $\mathbf{H}$ may contain multiple copies of the same block, if that block is written multiple times. At the time $\mathbf{C}$ is rebuilt, all duplicates will be suppressed, and only the most recent copy of each block will be rebuilt into $\mathbf{C}$. As we later show, the cost of rebuilding $\mathbf{C}$ is $O(\beta n \log n)$ (see Figure 4). Since $\mathbf{C}$ is only rebuilt every n write operations, the amortized cost is $O(\beta \log n)$ per write.

4.3 Security

Before we introduce the rebuilding algorithm, which lies at the heart of our construction, we give an intuitive explanation of why the main security property of PoR, i.e., retrievability (see Definition 2), is satisfied by our construction. Our algorithm will maintain the following invariant:

INVARIANT 1. *Treat* $\mathbf{C}$ *as the* $(k+1)$*-th level* $\mathbf{H}_{k+1}$. *We will maintain the invariant that each level* $\mathbf{H}_\ell$ *where* $\ell \in \{0, 1, \ldots, k+1\}$ *is a* $(m_\ell, 2^\ell, d_\ell)$ *erasure coding of* 2^ℓ *blocks, where* $m_\ell = \Theta(2^\ell)$. *Furthermore, we will show that encoding is a maximum distance encoding scheme, such that* $d_\ell = m_\ell - 2^\ell + 1 = \Theta(2^\ell)$, *i.e., the encoding of level* $\mathbf{H}_\ell$ *can tolerate up to* $d = \Theta(2^\ell)$ *erasures.*

In our specific construction, each level is encoded into exactly twice as many blocks, i.e., $\mathbf{H}_\ell$ is a $(2^{\ell+1}, 2^\ell, 2^\ell)$ erasure coding of 2^ℓ recently written blocks. Similarly, $\mathbf{C}$ is also encoded into twice as many blocks.

Recall that our audit algorithm checks $O(\lambda)$ blocks for each level $\mathbf{H}_\ell$ and for $\mathbf{C}$. Intuitively, the server has to delete more than half of the blocks in any level $\mathbf{H}_\ell$ or $\mathbf{C}$ to incur any data loss. However, if the server deletes so many blocks, checking $O(\lambda)$ *random* blocks in level $\mathbf{H}_\ell$ or $\mathbf{C}$ will almost surely detect it (with probability at least $1 - 2^{-\lambda}$). Also, observe that the combination of the hierarchical structure $\mathbf{H}$ and the buffer $\mathbf{C}$ contains information about the up-to-date copy of all blocks. Therefore, we can intuitively conclude that as long as the above invariant is maintained, our audits can detect any data loss with overwhelming probability.

The formal proof requires showing that there exists an extractor which can extract the up-to-date copy of all blocks if the extractor can run the Audit algorithm a polynomial number of times, with blackbox rewinding access to the server. We defer the formal proof to the full online version [22].

4.4 Fast Incrementally Constructible Codes

To achieve the promised efficiency, our main technical challenge is to design a fast, incrementally constructible code. In our scheme, we employ a fast incrementally constructible code based on Fast Fourier Transform (FFT). At a very high level, each level $\mathbf{H}_\ell$ contains two size-2^ℓ FFT-based codes. As is well-known, FFT can be computed using a standard divide-and-conquer strategy [2,3], forming a butterfly network (see Figure 3). This means that $\mathbf{H}_\ell$ can be computed from two $\mathbf{H}_{\ell-1}$'s in time $O(\beta \cdot 2^\ell)$. Note that in comparison, the oblivious sorting in ORAM schemes take time $O(\beta \cdot \ell \cdot 2^\ell)$ time to rebuild level H_ℓ.

Other than our FFT-based encoding scheme, it is possible to employ known linear-time constructible codes [23], which take $O(c)$ time to encode c blocks. We choose to use our FFT-based codes because of the relative conceptual and implementation simplicity. Particularly, rebuilding a level $\mathbf{H}_\ell$ requires only taking $O(2^\ell)$ linear combinations of blocks—in fact, in Section 5, we will leverage this linearity property and a homomorphic checksum scheme to make the client-server the bandwidth cost per writes independent of the block size.

4.4.1 Detailed Code Construction

In our construction, level $\mathbf{H}_\ell$ contains 2^ℓ blocks, which are encoded into $2^{\ell+1}$ codeword blocks, such that knowledge of any 2^ℓ codeword blocks can recover the original 2^ℓ blocks.

Suppose the original blocks in level $\mathbf{H}_\ell$ are denoted as a vector $\mathbf{x}_\ell$, where each block in $\mathbf{x}_\ell$ arrived at time $t, t+1, t+2, \ldots, t+2^\ell-1 \pmod n$ respectively. For level $\mathbf{H}_\ell$, t is always a multiple of 2^ℓ. Note that the time t is only incremented for write requests since reads do not need to touch the hierarchical log.

Notation. For the description of our code, we define the partial bit-reversal function and permutation—this inherently results from the divide and conquer strategy of the FFT. Let $\psi_c(i)$ denote a partial *bit-reversal function* that reverses the least significant c bits of an integer i. For example, let $n = 8$, then $\psi_3(1) = 4$, $\psi_2(1) = 2$. Let $\pi_c(\mathbf{x})$ denote a partial *bit-reversal permutation*, where index i is permuted to index $\psi_c(i)$.

Let also w denote the $2n$-th primitive root of unity in an appropriate finite field defined explicitly in Figure 4.

Closed-form formula for each level. Level $\mathbf{H}_\ell$ contains code blocks output by the following linear encoding scheme:

$$\mathbf{H}_\ell := \pi_\ell(\mathbf{x}_\ell)\,[F_\ell,\ D_{\ell,t}F_\ell]\ , \qquad (1)$$

where F_ℓ is the 2^ℓ by 2^ℓ Fourier Transform matrix from the finite field defined in Figure 4 and $D_{\ell,t}$ is an appropriate diagonal matrix defined as below.

$$D_{\ell,t} := \mathsf{diag}(w^{\psi_{k-\ell}(t)}, w^{\psi_{k-\ell}(t+1)}, \ldots, w^{\psi_{k-\ell}(t+2^\ell-1)})\,.$$

We now have the following lemma (its proof can be found in the full online version [22]):

LEMMA 1. *Any $2^\ell \times 2^\ell$ submatrix of the generator matrix $G_\ell := [F_\ell,\ D_{\ell,t}F_\ell]$ is full rank.*

Note that the above lemma means that if we have any 2^ℓ (out of $2^{\ell+1}$) code blocks for a level H_ℓ, we can recover all original blocks in that level. As mentioned earlier, using the divide-and-conquer strategy for computing FFT transforms, we can efficiently build our codes over time. The detailed algorithm is presented in the next subsection.

For a concrete small example when $n = 8$, please refer to the full online version [22].

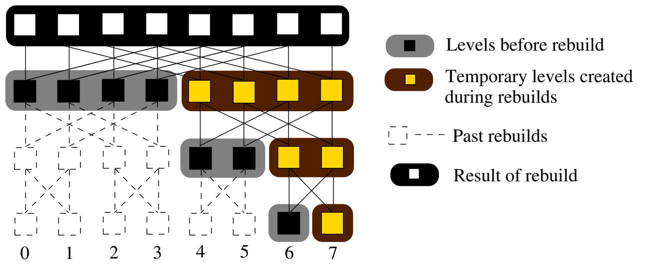

Figure 3: Butterfly network: the 8-th write operation encounters a full $\mathbf{H}_0$, $\mathbf{H}_1$, and $\mathbf{H}_2$. Blocks in $\mathbf{H}_0, \mathbf{H}_1, \mathbf{H}_2$ as well as the newly written block will be encoded and written to $\mathbf{H}_3$.

4.5 Detailed Protocol Description

The detailed protocol description is presented in Figure 4. The lemma below states that algorithm $\mathsf{mix}(\mathsf{A}_0, \mathsf{A}_1)$ specified in Figure 4 produces codes for the hierarchical log, satisfying Equation (1). Its proof can be found in the full online version [22].

LEMMA 2. *Algorithm $\mathsf{mix}(\mathsf{A}_0, \mathsf{A}_1)$ in Figure 4 ensures that each filled level $\mathbf{H}_\ell$ is a code of the form of Equation (1).*

4.6 Enhancements to Basic Construction

Segmenting blocks to avoid big integer operations. In the above description, each block is treated as a large integer during the encoding operations. We can avoid big integer arithmetic by dividing blocks into smaller segments. In our implementation, we choose a prime $p = \alpha n + 1$ for a small positive integer α, such that p can be represented with a basic integer type. We divide each block into $\beta_0 := \lceil \log p \rceil$ bits. and perform the mix algorithm described above on each smaller segment of the block. Note that using a smaller p does not affect the security of our scheme, since the parameter p is part of the erasure coding, and p is not related to the size of any cryptographic keys.

Ensuring authenticity and freshness. Since the last-write time of blocks in the buffers $\mathbf{H}$ and $\mathbf{C}$ are computable from the current time t, the client can simply use time and location encoded MACs to ensure authenticity and freshness of blocks in the buffers $\mathbf{H}$ and $\mathbf{C}$. Blocks in the buffer $\mathbf{U}$ need random access, therefore, we can use a standard Merkle hash tree (i.e., memory checking) to ensure freshness and authenticity of blocks in $\mathbf{U}$. We omit the details here since we will present an improved construction in Section 5 where we will revisit the authenticity and freshness issue.

THEOREM 1. *The basic dynamic PoR scheme of Figure 4 satisfies both authenticity (Definition 1) and retrievability (Definition 2).*

The authenticity and retrievability proofs of the above theorem can be found in the full online version [22].

5. IMPROVED CONSTRUCTION

In our basic construction (Section 4), every write incurs the reading and writing of $O(\log n)$ blocks, or $O(\beta \log n)$ bits on average, where β is the block size. In this section, we will describe an improved construction that achieves improved bandwidth and client computation overhead (however the server computation of the new construction remains

Main Dynamic PoR Construction

Let (encode, decode) *denote a maximum distance separable* (m, n, d) *erasure code, where* $m = O(n)$*.*

/ Authenticity: Standard Merkle-hash tree techniques can be used to verify the authenticity/freshness of blocks retrieved from the server. For clarity, we omit such details from the scheme description. Section 5 presents more optimized techniques for achieving authenticity/freshness. */*

```
Init($1^\lambda, n, \beta, \mathcal{M}$):
    $\mathbf{U} \leftarrow \mathcal{M}$; $\mathbf{C} \leftarrow \mathsf{encode}(\mathcal{M})$; $\mathbf{H} \leftarrow \emptyset$.
Read($i, st, \bar{\mathcal{M}}$): To read a block $i$, simply read $\mathbf{U}[i]$ and
    check its authenticity.
Write($i, B, st, \bar{\mathcal{M}}$): To overwrite block $i$ with $B$, let
    $\mathbf{U}[i] \leftarrow B$. Call HAdd($B$).
Audit($st, \bar{\mathcal{M}}$): Check the authenticity of $O(\lambda)$ number of
    blocks from the erasure coded copy $\mathbf{C}$ and $O(\lambda)$ number
    of blocks from each filled level $\mathbf{H}_\ell$.
```

Algorithm HAdd(B)

Suppose each $\mathbf{H}_\ell$ *is of the form* $\mathbf{H}_\ell := (X_\ell, Y_\ell)$*, where* X_ℓ *and* Y_ℓ *each stores* 2^ℓ *blocks.*

*Let the current write be the t-th write operation (*mod 2^k*). Let* $\psi(\cdot)$ *denote the bit reversal function, such that* $\psi(t)$ *outputs the value corresponding to reversing the bits of the binary representation of t.*

```
• If $\mathbf{H}_0$ is empty, let $X_0 = B$, $Y_0 = B \cdot w^{\psi(t)}$.
• Else, suppose levels $0, \ldots, \ell$ are consecutively full levels
  for some $\ell < k$, and level $\ell + 1$ is the first empty level.
    – Call HRebuildX($\ell + 1, B$).
    – Call HRebuildY($\ell + 1, B \cdot w^{\psi(t)}$).
• Every $2^k$ time steps, call CRebuild().
```

Parameters: Let $p = \alpha \cdot (2n) + 1$ denote a prime for some $\alpha \in \mathbb{N}$. Suppose p is chosen to be large enough to encode blocks of β bits.
Let g denote a generator of $\mathbb{Z}_p^*$. Let $w = g^\alpha \mod p$ be a $2n$-th primitive root of unity mod p.

Algorithms HRebuildX(ℓ, B), HRebuildY(ℓ, B)

*/** HRebuildY *is exactly the same as* HRebuildX*, except that all X's are replaced with Y's. Below we formally describe* HRebuildX *as an example. */*

Inputs: Consecutively filled levels $X_0 \ldots X_{\ell-1}$ (the X portion), and a new (possibly encoded) block B.
Output: A rebuilt X_ℓ. Levels $X_0, \ldots, X_{\ell-1}$ are emptied at the end of HRebuildX(ℓ, B).

```
• $\widetilde{X}_1 \leftarrow \mathsf{mix}(X_0, B)$
• For $i = 1$ to $\ell - 1$: $\widetilde{X}_{i+1} \leftarrow \mathsf{mix}(X_i, \widetilde{X}_i)$
    // $\widetilde{X}_i$'s are the red arrays in Figure 3.
• Output $X_\ell := \widetilde{X}_\ell$.
```

Algorithm mix($\mathsf{A}_0, \mathsf{A}_1$)

Inputs: Two arrays $\mathsf{A}_0, \mathsf{A}_1 \in \mathbb{Z}_p^{2^\ell}$ each of length 2^ℓ.

Outputs: A new array A of length $2 \cdot 2^\ell$.

```
• Let $\nu = w^{n/2^\ell}$ be a $2^{\ell+1}$-th primitive root of unity.
• For $i = 0$ to $2^\ell - 1$:
      $\mathsf{A}[i] := \mathsf{A}_0[i] + \nu^i \mathsf{A}_1[i] \pmod p$
      $\mathsf{A}[i + 2^\ell] := \mathsf{A}_0[i] - \nu^i \mathsf{A}_1[i] \pmod p$
• Output A.
```

Algorithm CRebuild()

```
• $\mathbf{C} \leftarrow \mathsf{encode}(\mathbf{U})$.
• Empty $\mathbf{H}$.
```

Figure 4: Basic protocol description. We assume that blocks are tagged with their block identifier.

the same), by removing the dependence on the block size. In our improved construction, writing a block incurs only $\beta + O(\lambda \log n)$ cost, where λ is the security parameter; and the typical value for λ is 128 or 256 bits. This means that *the bandwidth overhead and the client computation for write operations of our PoR construction is analogous to that of a Merkle hash tree.* Recall that by the definition of PoR, PoR is a strictly stronger primitive than a Merkle hash tree since PoR not only needs to guarantee authenticity and freshness—which a standard Merkle hash tree ensures—but it needs to additionally guarantee that the server is storing all of the client's data. Our construction basically shows that we can obtain the additional PoR guarantee (on top of a Merkle hash tree) almost for free.

5.1 Intuition

Recall that in the basic construction, the $O(\beta \log n)$ cost arises from the periodical rebuilding of the hierarchical log structure $\mathbf{H}$ and the erasure-coded copy $\mathbf{C}$. In the basic construction, the server is conceptually treated as a passive remote storage device, and the client performs all the computation. Therefore, the client needs to download the blocks from the server to perform computation over these blocks, when rebuilding any level $\mathbf{H}_\ell$ in the hierarchical log structure, or when rebuilding $\mathbf{C}$.

Our idea is to have the server perform the computation on behalf of the client, thus significantly reducing the client computation and bandwidth. Observe that in our basic construction, the algorithms for rebuilding each $\mathbf{H}_\ell$ or $\mathbf{C}$ are publicly known to the server, and the server can perform the rebuilding on its own. The client simply needs to check that the server performed the rebuilding correctly.

To achieve this performance improvement and avoid downloading the blocks during a rebuilding, we will attach a *homomorphic checksum* along with each (encoded or unencoded) block in the hierarchical log structure $\mathbf{H}$ or $\mathbf{C}$. Each homomorphic checksum is a collision-resistant summary of a block. Now, instead of performing the rebuilding over real data blocks, the client simply downloads checksums, checks their authenticity and performs the rebuilding over these homomorphic checksums.

The homomorphic checksum for each block is then tagged with its position and time written, and stored on the server in encrypted and authenticated format. This ensures that 1) the server does not know the values of the homomorphic checksum which is necessary for security as explained later; and 2) the client can always verify the correctness and freshness of the homomorphic checksum retrieved.

We note here that we choose to use a special type of homomorphic checksums that are not publicly verifiable (though we later show that this is not a limitation for making our scheme publicly verifiable). The reason for that is that our homomorphic checksum construction is designed for performance. In comparison, publicly verifiable homomorphic checksums (e.g., lattice-based ones [19] or RSA-based ones [5]) are not as efficient practice.

5.2 Homomorphic Checksums: Definitions

We now give the definition of a homomorphic checksum scheme, along with its definition of security (unforgeability).

As mentioned in Section 4.6, we segment a block into segments of bit-length $\beta_0 = \lceil \log p \rceil$. Therefore, we can write a block B in the form $B \in \mathbb{Z}_p^{\lceil \beta/\beta_0 \rceil}$.

Definition 4. *We define a homomorphic checksum scheme to consist of the following algorithms:*

$sk \leftarrow \mathbb{K}(1^\lambda)$. The key generation algorithm $\mathbb{K}$ takes in the security parameter 1^λ, the block size β, outputting a secret key sk.

$\sigma \leftarrow \mathbb{S}_{sk}(B)$. The authentication algorithm $\mathbb{S}$ takes in the secret key sk, a block $B \in \mathbb{Z}_p^{\lceil \beta/\beta_0 \rceil}$, and outputs a checksum σ. In our scheme σ has bit-length $O(\lambda)$.

Definition 5 (Unforgeability of checksum). *We say that a homomorphic checksum scheme is unforgeable, if for any (polynomial-time) adversary $\mathcal{A}$,*

$$\Pr\left[\begin{array}{lcl} sk \leftarrow \mathbb{K}(1^\lambda) & : & B_1 \neq B_2 \\ B_1, B_2 \leftarrow \mathcal{A}(1^\lambda) & & \mathbb{S}_{sk}(B_1) = \mathbb{S}_{sk}(B_2) \end{array}\right] \leq \mathsf{negl}(\lambda)\,.$$

Namely, an adversary who has not seen the secret key sk or any checksum cannot produce two blocks B_1 and B_2 that result in the same checksum. In our scheme, the clients encrypts all checksums before storing them at the server, thus the server does not see the secret key or any checksums.

Additive homomorphism. We require our homomorphic checksum scheme to achieve additive homomorphism, i.e., for any $sk \leftarrow \mathbb{K}(1^\lambda)$, for any blocks $B_1, B_2 \in \mathbb{Z}_p^{\lceil \beta/\beta_0 \rceil}$, for any $a_1, a_2 \in \mathbb{Z}_p$, it is $a_1\mathbb{S}_{sk}(B_1) + a_2\mathbb{S}_{sk}(B_2) = \mathbb{S}_{sk}(a_1B_1 + a_2B_2)$.

5.3 Homomorphic Checksum Construction

We now present a simple homomorphic checksum scheme.

$\mathbb{K}(1^\lambda)$: The client picks a random matrix $\mathbf{M} \overset{\$}{\leftarrow} \mathbb{Z}_p^{\rho \times \lceil \beta/\beta_0 \rceil}$ and lets $sk := \mathbf{M}$. The number of rows ρ is chosen such that $\rho\beta_0 = O(\lambda)$, i.e., we would like the resulting checksum to have have about $O(\lambda)$ number of bits.

$\mathbb{S}_{sk}(B)$: On input a block $B \in \mathbb{Z}_p^{\lceil \beta/\beta_0 \rceil}$, compute checksum $\sigma := \mathbf{M} \cdot B$. Note that the checksum compresses the block from β bits to $O(\lambda)$ bits.

Additive homomorphism. It is not hard to see that the above homomorphic checksum scheme satisfies additive homomorphism, since $a_1\mathbf{M}B_1 + a_2\mathbf{M}B_2 = \mathbf{M}(a_1B_1 + a_2B_2)$.

Unforgeability. Inthe full online version [22] we show that the above construction satisfies unforgeability in an information theoretic sense (i.e., even for computationally unbounded adversaries).

Theorem 2. *The above homomorphic checksum construction satisfies the unforgeability notion (Definition 5).*

Efficiency. As we saw in the definition of the homomorphic checksum, the blocks used are represented as vectors in $\mathbb{Z}_p^{\lceil \beta/\beta_0 \rceil}$. We use a very small p in our implementation—in fact we take $p = 3 \cdot 2^{30} + 1$ and therefore checksum operations do not need big integer operations and are highly efficient.

5.4 Using Homomorphic Checksums

Relying on homomorphic checksums, we can reduce the bandwidth cost and the client computation to $\beta + O(\lambda \log n)$ for write operations. To do this, we make the following modifications to the basic construction described in Section 4.

Store encrypted and authenticated checksums on server. First, for every (encoded or uncoded) block stored on the server, in the buffers $\mathbf{U}$, $\mathbf{C}$ and $\mathbf{H}$, the client attaches a homomorphic checksum, which is encrypted under an authenticated encryption scheme $\mathsf{AE} := (\mathbb{E}, \mathbb{D})$.

Let $sk := (\mathbf{M}, sk_0)$ denote the client's secret key (unknown to the server). Specifically $\mathbf{M} \in \mathbb{Z}_p^{\rho \times \lceil \beta/\beta_0 \rceil}$ is the random matrix used in the homomorphic checksum scheme, and sk_0 is a master secret key used to generate one-time keys for the authenticated encryption scheme AE.

For the buffers $\mathbf{C}$ and $\mathbf{H}$, suppose a block B is written to address addr on the server at time t. The client generates the following one-time key:

$$\kappa = \mathsf{PRF}_{sk_0}(0, \mathsf{addr}, t)$$

and attaches $\widetilde{\sigma}(B)$ with block B, where

$$\widetilde{\sigma}(B) := \mathsf{AE}.\mathbb{E}_\kappa(\mathbb{S}_M(B))\,.$$

For buffer $\mathbf{U}$, the client uses a fixed encryption key $\kappa = \mathsf{PRF}_{sk_0}(1)$ as blocks in $\mathbf{U}$ have unpredictable last-write times.

Ensuring authenticity and freshness. For each block in buffers $\mathbf{H}$ and $\mathbf{C}$, its time and location dependent $\widetilde{\sigma}(B)$ allows the client to verify the authenticity and freshness of the block. The client need not separately MAC the blocks in $\mathbf{H}$ or $\mathbf{C}$.

For blocks in $\mathbf{U}$, a Merkle tree is built over these $\widetilde{\sigma}(B)$'s for the $\mathbf{U}$ buffer, and the client keeps track of the root digest. After fetching a block B from buffer $\mathbf{U}$, the client fetches its corresponding $\widetilde{\sigma}(B)$, verifies $\widetilde{\sigma}(B)$ with the Merkle tree, and verifies that the block B fetched agrees with $\widetilde{\sigma}(B)$.

Rebuilding H based on homomorphic checksum. In our improved scheme, the $\mathsf{HRebuild}$ algorithm is executed by the server. In order to enforce honest server behavior, the client performs the $\mathsf{HRebuild}$ algorithm, not directly over the blocks, but over $\widetilde{\sigma}(B)$'s. In other words, imagine the client were to execute the $\mathsf{HRebuild}$ algorithm on its own:

- Whenever the client needed to read a block B from the server, it now reads $\widetilde{\sigma}(B)$ instead, and decrypts $\sigma(B) \leftarrow \mathsf{AE}.\mathbb{D}_\kappa(\widetilde{\sigma}(B))$. In the above, $\kappa := \mathsf{PRF}_{sk_0}(0, \mathsf{addr}, t)$, where addr is the physical address of block B, and t is the last time B is written. Note that for any block in the hierarchical log structure $\mathbf{H}$ and in the erasure-coded copy $\mathbf{C}$, the client can compute exactly when the block was last written from the current time alone. The client rejects if the decryption fails, which means that the server is misbehaving.
- Whenever the client needed to perform computation over two blocks B_1 and B_2, the client now performs the same operation over the homomorphic checksums $\sigma(B_1)$ and $\sigma(B_2)$. Recall that in $\mathsf{HRebuild}$, we only have addition and multiplications by known constants—therefore, our additively homomorphic checksum would work.
- Whenever the client needed to write a block to the server, it now writes the new $\widetilde{\sigma}(B)$ instead.

Rebuilding C based on homomorphic checksum. Similar to the above, the server rebuilds $\mathbf{C}$ on behalf of the client, and the client only simulates the rebuilding of $\mathbf{C}$ operating on the $\widetilde{\sigma}(B)$'s instead of the full blocks. However, slightly differently from the rebuilding of $\mathbf{H}$, the buffer $\mathbf{C}$ is rebuilt by computing an erasure code of the fresh data in $\mathbf{U}$.

One way to do this is to use the same FFT-based code as described in Section 4. The server can compute the code using the butterfly diagram (Figure 3) in $O(\beta n \log n)$ time. The client simply has to simulate this encoding process using the $\widetilde{\sigma}(B)$'s rather than the data blocks—therefore the client-server bandwidth is $O(\lambda n \log n)$ for each rebuild of $\mathbf{C}$.

5.5 Reducing Client-Server Audit Bandwidth

In our basic construction in Section 4, audits require transferring $O(\lambda)$ random from each level $\mathbf{H}_\ell$ and from $\mathbf{C}$—therefore the audit cost is $O(\lambda\beta \log n)$. However this can be reduced by observing that audited blocks can be aggregated using a linear aggregation scheme, and the client can check the "aggregated block" using the homomorphic "aggregate checksum". This is similar to the technique used by Shacham and Waters [20]. The new audit protocol is as below:

- *Challenge.* Client picks $O(\lambda)$ random indices for each level $\mathbf{H}_\ell$ and for $\mathbf{C}$. Client picks a random challenge $\rho_i \in \mathbb{Z}_p$ for each selected index. The random indices are denoted $I := \{\mathsf{addr}_1, \mathsf{addr}_2, \ldots, \mathsf{addr}_r\}$, where $r = O(\lambda \log n)$. The random challenges are denoted $R := \{\rho_1, \rho_2, \ldots, \rho_r\}$. Client sends $I := \{\mathsf{addr}_1, \mathsf{addr}_2, \ldots, \mathsf{addr}_r\}$ and $R := \{\rho_1, \rho_2, \ldots, \rho_r\}$ to the server.
- *Response.* Let B_{addr} denote the block at addr. Server sends client the corresponding $\widetilde{\sigma}(B_{\mathsf{addr}})$ for each requested index $\mathsf{addr} \in I$. Server also computes $B^* = \sum_{i=1}^r \rho_i B_{\mathsf{addr}_i}$, and sends B^* to the client.
- *Verification.* The client decrypts all $\widetilde{\sigma}(B)$'s received and obtains $\sigma(B)$'s. Client computes $v = \sum_{i=1}^r \rho_i \sigma(B_{\mathsf{addr}_i})$, and the checksum $\sigma(B^*)$. The client checks if $v \overset{?}{=} \sigma(B^*)$.

This modification reduces the bandwidth for audits to $\beta + O(\lambda^2 \log n)$ since only checksums and *one* aggregate block need to be communicated from the server to the client.

Theorem 3. *The improved dynamic PoR scheme that uses homomorphic checksums satisfies both authenticity (Definition 1) and retrievability (Definition 2).*

The proofs of the above theorem can be found in the full online version [22].

6. IMPLEMENTATION AND EVALUATION

We implemented a variant of our secretly-verifiable construction described in Section 5. The implementation is in C# and contains about 6,000 lines of code measured using SLOCCount [4].

6.1 Practical Considerations

Our implementation features several further optimizations in comparison with the theoretical construction in Section 5.

Reducing write cost. Our implementation is optimized for the common scenario where writes are frequent and audits are relatively infrequent. We apply a simple trick to reduce the client-server bandwidth for write operations; however, at slightly higher cost for audits.

To reduce the overhead of writing the log structure $\mathbf{H}$, the client can group about $O(\log n)$ blocks into a bigger block when writing to $\mathbf{H}$. The simplest instantiation is for the client to cache $O(\log n)$ blocks and write them in a batch as a single bigger block.

This optimization reduces the amortized cost of writing to $\mathbf{H}$ to $\beta + O(\lambda)$. Particularly, the client-server bandwidth overhead (in addition to transferring the block itself) can be broken down into two parts: 1) about $O(\lambda \log n)$ due to the Merkle hash tree for the buffer $\mathbf{U}$; and 2) $O(\lambda)$ for writing to the log structure $\mathbf{H}$—notice that this part is independent of n. This trick increases an audit's disk I/O cost to roughly $O(\lambda\beta \log^2 n)$ (but only $O(\lambda \log n)$ disk seeks), and increases an audit's client-server bandwidth to $\beta \log n + O(\lambda^2 \log n)$.

Deamortization. Our theoretic construction has low amortized cost per write, but high worst-case cost. We can perform a standard deamortization trick (introduced for similar multi-level hierarchical data structures used in several ORAM constructions [18, 31]) to spread the hierarchical encoding work over time, such that we achieve good worst-case performance. We implemented deamortization, therefore it is reflected in the experimental results.

Reducing disk seeks. Our hierarchical log features sequential data accesses when rebuilding level $\mathbf{H}_\ell$ from two arrays of size $2^{\ell-1}$. (see Figure 3 and Algorithm mix of Figure 4). Due to such sequential access patterns, we can use a bigger chunk size to reduce the number of seeks. Every time, we read a chunk and cache it in memory while performing the linear combinations in Algorithm mix.

6.2 Experimental Results

We ran experiments on a single server node with an i7-930 2.8 Ghz CPU and 7 rotational WD1001FALS 1TB 7200 RPM HDDs with 12 ms random I/O latency. Since reads are not much different from a standard Merkle-tree, we focus on evaluating the performance overhead of writes and audits.

6.2.1 Write Cost

Client-server bandwidth. Figures 5 and 6 depict the client-server bandwidth cost for our scheme. Figure 5 plots the total client-server bandwidth consumed for writing a 4KB block, for various storage sizes. We compare the cost to a standard Merkle hash tree, and show that our POR scheme achieves comparable client-server bandwidth.

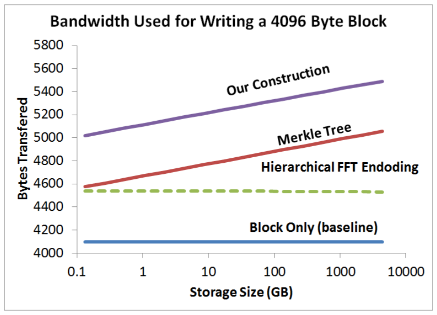

Figure 5: Client-server bandwidth for writing a 4KB block.

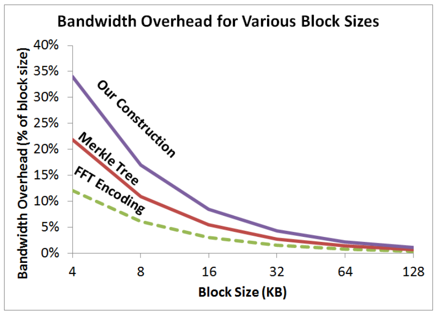

Figure 6: Percentage of bandwidth overhead for various block sizes. The total storage capacity is 1TB.

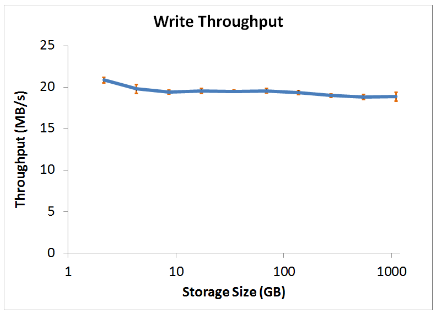

Figure 7: Throughput for write operations when client-server bandwidth is ample. The error bars indicate one standard deviation over 20 trials. A 4KB block size is used.

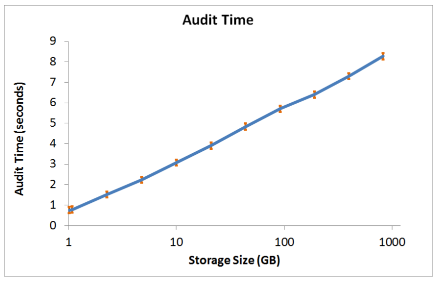

Figure 8: Time spent by server for performing an audit. Does not include network transfer time. Error bars denote 1 standard deviation from 20 trials. The majority of this time is disk I/O cost. A 4KB block size is chosen.

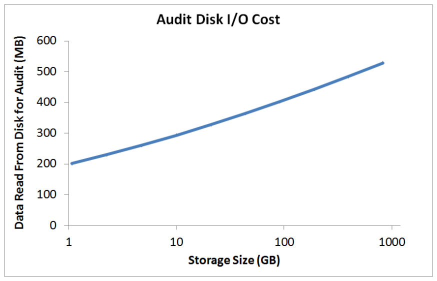

Figure 9: Disk I/O cost for each audit. Block size is chosen to be 4KB.

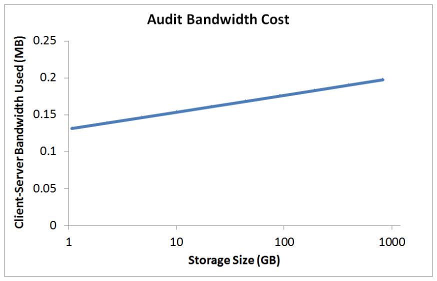

Figure 10: Client-server bandwidth for an audit. Block size is chosen to be 4KB.

We also plot the dotted "Hierarchical FFT encoding" curve, which represents the portion of our bandwidth cost dedicated to the rebuilding of the buffers **C** and **H**. As mentioned earlier in Section 6.1, our implementation features optimizations such that the client-server bandwidth overhead contains the sum of two parts 1) a $O(\lambda)$ part for rebuilding **H** and **C**. This is represented by the dotted line in Figure 5—note that it is independent of n, hence a straight-line; and 2) $O(\lambda \log n)$ bandwidth due to the Merkle tree for the up-to-date copy **U**.

Figure 6 shows the percentage of bandwidth overhead (not including transferring the block itself) for each write. In our scheme, the bandwidth overhead is indepedent of the block size. Therefore, the bigger the block size, the smaller the percentage of bandwidth overhead. We also compare against a standard Merkle-tree in the figure, and show the portion of the bandwidth overhead for the hierarchical encoding procedure.

Server disk I/O performance. Figure 7 shows the write throughput of our scheme. The server's disk I/O cost is about $O(\log n)$, i.e., it needs to read roughly $O(\log n)$ blocks for every write operation. Our experiments show that if server disk I/O were maxed out, the POR write throughput we can achieve would be roughly 20MB/s under our setup. We cache the smallest 10 levels of **H** in memory in this experiment. In practice, however, the client-server bandwidth is more likely to be the bottleneck, and the actual POR throughput achievable will be limited by the available client-server bandwidth.

While the server needs to read $O(\log n)$ blocks for each write, the number of seeks is very small. As mentioned earlier in Section 6.1, the hierarchical log structure **H** is mostly written in a sequential fashion, and since we choose a large chunk size (roughly 50MB), and cache chunks in memory, every write operation requires only 0.03 to 0.06 seeks on average. We cache the Merkle hash tree for **U** in memory in our experiments.

6.2.2 Audit Cost

We use a $\lambda = 128$ for these experiments, i.e., each audit samples 128 blocks from each level $\mathbf{H}_\ell$ and buffer **C**.

Figure 8 shows the time spent by the server for each audit operation—including time for reading disk and performing computation, but not including network transfer time between client and server (client-server network overhead is characterized separately in Figure 10). The majority of this time is spent on disk I/O, and is dominated by disk seeks. There are roughly $O(\lambda \log n)$ seeks per audit operation parallelized to 7 disks, and each seek takes roughly 12ms.

Figure 9 shows the disk I/O cost for an audit. As mentioned in Section 6.1, we optimize for writes at slightly higher audit cost, and the audit disk I/O cost is $O(\lambda\beta \log^2 n)$—this is why the line curves slightly, and is not linear.

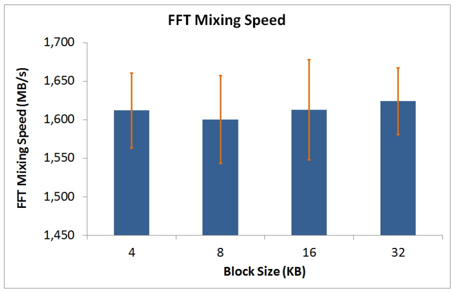

Figure 11: Computational throughput for hierarchical erasure coding. Error bars denote 1 standard deviation from 20 runs.

Figure 10 shows the client-server bandwidth consumed for an audit. As mentioned in Section 6.1, unlike our theoretic construction, our implementation chooses to speed up writes at slightly higher client-server bandwidth for audits, namely, $\beta \log n + O(\lambda \log n)$.

Computational overhead for hierarchical erasure coding. As shown in Figure 11, the hierarchical coding scheme can be computed at extremely fast speeds, i.e., >1600MB/s per level on a modern processor. To characterize the computational cost, we cached about 4GB of data in memory, and avoid performing disk fetches during this experiment.

7. REFERENCES

[1] `https://en.wikipedia.org/wiki/Hash_tree`.
[2] Fast fourier transform. `http://math.berkeley.edu/~berlek/classes/CLASS.110/LECTURES/FFT`.
[3] Fast fourier transform. `http://en.wikipedia.org/wiki/Fast_Fourier_transform`.
[4] Sloccount. `http://www.dwheeler.com/sloccount/`.
[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *CCS*, 2007.
[6] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, pages 111–131, 2011.
[7] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In *CCSW*, pages 43–54, 2009.
[8] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *Eurocrypt*, 2013.
[9] Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In *TCC*, pages 109–127, 2009.
[10] C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In *CCS*, 2009.
[11] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.
[12] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, 2011.
[13] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, 2012.
[14] A. Juels and B. S. K. Jr. Pors: proofs of retrievability for large files. In *ACM Conference on Computer and Communications Security*, pages 584–597, 2007.
[15] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, 2012.
[16] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, CRYPTO '89, 1989.
[17] Z. Mo, Y. Zhou, and S. Chen. A dynamic proof of retrievability (por) scheme with o(log n) complexity. In *ICC'12*, pages 912–916, 2012.
[18] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *STOC*, pages 294–303, 1997.
[19] C. Papamanthou, E. Shi, R. Tamassia, and K. Yi. Streaming authenticated data structures. In *EUROCRYPT*, 2013.
[20] H. Shacham and B. Waters. Compact proofs of retrievability. In *ASIACRYPT*, pages 90–107, 2008.
[21] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, pages 197–214, 2011.
[22] E. Shi, E. Stefanov, and C. Papamanthou. Practical dynamic proofs of retrievability. Technical report, 2013.
[23] D. A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42(6):1723–1731, 1996.
[24] E. Stefanov and E. Shi. Oblivistore: High performance oblivious cloud storage. In *IEEE Symposium on Security and Privacy*, 2013.
[25] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *NDSS*, 2012.
[26] 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, 2012.
[27] J. van Lint. *Introduction to Coding Theory*. Springer-Verlag, 1992.
[28] 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.
[29] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(5):847–859, 2011.
[30] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In *CCS*, 2008.
[31] P. Williams, R. Sion, and A. Tomescu. Privatefs: A parallel oblivious file system. In *CCS*, 2012.
[32] Q. Zheng and S. Xu. Fair and dynamic proofs of retrievability. In *CODASPY*, 2011.

APPENDIX

A. ACHIEVING PUBLIC VERIFIABILITY

Although our basic construction in Section 4 provides *public verifiability*, the more efficient scheme described in Section 5 does not, since the homomorphic checksum requires the client keep secret state. In this section we show how to turn the efficient scheme of Section 5 into publicly verifiable.

In a publicly verifiable setting, only a *trusted data source* can write data; however, anyone can perform *verifiable reads* and *PoR audits*.

Ensuring the public verifiability of reads comes for free, since a Merkle hash tree is maintained over the buffer $\mathbf{U}$. Therefore the trusted source can simply sign and publish the up-to-date root digest of this Merkle-hash tree and make it available to the public for verification of read operations.

We now focus on how to achieve public *auditability* so that to enable anyone challenge the server to prove that it possesses all data (owned by the source) and *still* maintain the bandwidth of the write operations to be low.

A.1 Public Auditability with Low Write Cost

To achieve public auditability, we build a separate Merkle hash tree over the blocks of each level $\mathbf{H}_\ell$, and one for $\mathbf{C}$.

The up-to-date root digests of all $O(\log n)$ hash trees will be publicized. During a public audit, a user with the root digests requests $O(\lambda)$ blocks at random for each level $\mathbf{H}_\ell$ and buffer $\mathbf{C}$, and checks them with the root digests.

One question remains: how does the trusted source keep track of these root digests without having to transfer original blocks during the rebuilding of the hierarchical levels? To address this question, we sketch our idea below, and leave the details and full proofs to the full online version [22].

Our idea is to have the server compute the new Merkle trees for a level (or the buffer $\mathbf{C}$) when it is being rebuilt. However, we need to protect against a malicious server that can potentially cheat and output the wrong digests. We apply a probabilistic checking idea again here.

When rebuilding a level $\mathbf{H}_\ell$ (or buffer $\mathbf{C}$), the following happens:

- As in the secretly-verifiable scheme (Section 5), the trusted source downloads the encrypted/authenticated homomorphic checksums and "simulates" the rebuilding over these $\widetilde{\sigma}(B)$'s.
- The server performs the rebuilding, computes the new digest h of $\mathbf{H}_\ell$, and commits h to the source.
- The source challenges $O(\lambda)$ random blocks in $\mathbf{H}_\ell$. For each challenged block B: the source downloads the block itself B, its $\widetilde{\sigma}(B)$, and the path in the Merkle tree necessary to verify block B. The client now checks that $\widetilde{\sigma}(B)$ verifies for B, and that the *already committed* root digest h verifies for B as well.

Proof intuition. At a high level, this idea works because if the server can pass the probabilistic check (of the committed root digest h), then at least a constant fraction of the rebuilt level $\mathbf{H}_\ell$ (or buffer $\mathbf{C}$) is correctly incorporated into the claimed digest h. Due to PoR's inherent erasure coding, it turns out that this suffices proving the retrievability of the publicly verifiable PoR scheme. The full proof is deferred to the full online version [22].

Write cost. Suppose the trusted source caches locally the smallest $\log\lambda + \log(2/\epsilon)$ levels consisting about $2\lambda/\epsilon$ number of blocks, for an arbitrarily small $0 < \epsilon < 1$. Note that these are the levels that are accessed more frequently during the write operations. We can then show that the source-server bandwidth for each write operation is $\beta(1+\epsilon) + O(\lambda \log n)$. The details of this analysis is elementary, and deferred to the full online version [22]. The above analysis assumes that exactly λ blocks are probabilistically checked for each Merkle tree hash tree of the remaining (uncached) levels.

A.2 Reducing Public Audit Cost

The publicly verifiable approach described above requires $O(\lambda \log n(\beta + \log n))$ overhead for public auditing. Particularly, for each of the $O(\log n)$ levels $\mathbf{H}_\ell$ as well as $\mathbf{C}$, $O(\lambda)$ blocks need to be checked; and to check a block involves downloading the block itself and $\log n$ hashes of the Merkle hash tree. With some additional tricks, in particular, by carefully aligning the Merkle tree structure with the hierarchical log structure $\mathbf{H}$, we can further improve the public audit overhead to $O(\lambda\beta \log n)$. We defer these details to the Appendix. The basic idea is as follows:

- Instead of building a separate Merkle tree per level $\mathbf{H}_\ell$, build a single Merkle tree over the entire hirarchical log structure $\mathbf{H}$, and another one for $\mathbf{C}$. Furthermore, the Merkle tree will be aligned on top of the hierarchical structure $\mathbf{H}$. Since $\mathbf{H}$ has exponentially growing levels, we can view $\mathbf{H}$ as a tree, where internal nodes are assigned values—in our case, the blocks are the values of internal nodes of the Merkle tree. The hash of each internal node in the Merkle tree is computed as $H(h_{\text{left}}, h_{\text{right}}, B)$, where h_{left} is the hash of the left child, h_{right} is the hash of the right child, and B is the block associated with that node. The client publishes the hash of the Merkle tree for $\mathbf{H}$ and the one for $\mathbf{C}$.
- During public audits, random checks for $\mathbf{C}$ are still done as before. To check $\mathbf{H}$, instead of randomly sample $O(\lambda)$ blocks from each level $\mathbf{H}_\ell$, the client randomly samples $O(\lambda)$ paths from the root to $O(\lambda)$ randomly selected leaf nodes, and sample the blocks on these paths. As a result $O(\lambda)$ blocks from each level gets sampled, and it is not hard to see that only $O(\lambda \log n)$ hashes needs to be transmitted to verify all the paths sampled – namely, hashes of all sibling nodes to the sampled paths need to be transmitted. This reduces the public audit overhead to $O(\lambda\beta \log n)$, assuming $\beta = O(\lambda)$.
- When a non-top level $\mathbf{H}_\ell$ gets rebuilt, the server rebuilds the Merkle hash tree, and the client performs the following probabilistic checking protocol.
  The client first retrieves all hashes at level $\mathbf{H}_{\ell+1}$, and makes sure that they are consistent with the old digest. The client then randomly samples $O(\lambda)$ blocks at level $\mathbf{H}_\ell$, to make sure that these blocks are correctly incorporated into the new root digest as claimed by the server.
  When the top level is rebuilt, the client simply checks $O(\lambda)$ random blocks in the top level, and ensures that they are correctly incorporated into the root digest.

THEOREM 4. *The dynamic PoR scheme with public verifiability satisfies both authenticity (Definition 1) and retrievability (Definition 2).* The proof is in the full online version [22].

A.3 Resizing the Storage

Our scheme can easily be modified to support insertions and deletions of blocks. Insertions can easily be supported by adding a level to the hierarchical log structure $\mathbf{H}$ whenever number of blocks doubles. Deletions can be supported by updating the deleted block with $\perp$. Further, whenever the number of deleted elements exceeds roughly half the size of the dataset, the client can rebuild and consilidate the hierarchical log by suppressing deleted items. We will provide more details in the online full version [22].

Acknowledgments

This work is partially supported by NSF grant CNS-1314857, a Google Research Award, the NSF Graduate Research Fellowship under Grant No. DGE-0946797, by a DoD National Defense Science and Engineering Graduate Fellowship, by Intel award through the ISTC for Secure Computing, and a grant from Amazon Web Services. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies. We would like to thank Hubert Chan, Jonathan Katz, and Hongsheng Zhou for helpful discussions, and the anonymous reviewers for their insightful feedback.