Dynamic proofs of retrievability with low server storage

Gaspard Anthoine, Jean-Guillaume Dumas, Michael Hanling, Mélanie de Jonghe, Aude Maignan, Clément Pernet, Daniel S. Roche

Dynamic proofs of retrievability with low server storage

Gaspard Anthoine* Jean-Guillaume Dumas* Michael Hanling†
Mélanie de Jonghe* Aude Maignan* Clément Pernet* Daniel S. Roche†

Abstract

Proofs of Retrievability (PoRs) are protocols which allow a client to store data remotely and to efficiently ensure, via audits, that the entirety of that data is still intact. A *dynamic* PoR system also supports efficient retrieval and update of any small portion of the data. We propose new, simple protocols for dynamic PoR that are designed for practical efficiency, trading decreased persistent storage for increased server computation, and show in fact that this tradeoff is inherent via a lower bound proof of time-space for any PoR scheme. Notably, ours is the first dynamic PoR which does not require any special encoding of the data stored on the server, meaning it can be trivially composed with any database service or with existing techniques for encryption or redundancy. Our implementation and deployment on Google Cloud Platform demonstrates our solution is scalable: for example, auditing a 1TB file takes just over 7 minutes and costs less than $0.12 USD. We also present several further enhancements, reducing the amount of client storage, or the communication bandwidth, or allowing *public verifiability*, wherein any untrusted third party may conduct an audit.

1 Introduction

1.1 The need for integrity checks

While various computing metrics have accelerated and slowed over the last half-century, one which undeniably continues to grow quickly is data storage. One recent study estimated the world's storage capacity at 4.4ZB ($4.4 \cdot 10^{21}$), and growing at a rate of 40% per year [11]. Another study group estimates that by 2025, half of the world's data will be stored remotely, and half of that will be in public cloud storage [35].

As storage becomes more vast and more outsourced, users and organizations need ways to ensure the *integrity* of their data – that the service provider continues to store it, in its entirety, unmodified. Customers may currently rely on the reputations of large cloud companies like IBM Cloud or Amazon AWS, but even those can suffer data loss events [2, 23], and as the market continues to grow, new storage providers without such long-standing reputations need cost-effective ways to convince customers their data is intact.

This need is especially acute for the growing set of *decentralized storage networks* (DSNs), such as Filecoin, Storj, SAFE Network, Sia, and PPIO, that act to connect users who need their data stored with providers ("miners") who will be paid to store users' data. In DSNs, integrity checks are useful at two levels: from the customer who may be wary of trusting blockchain-based networks, and within the network to ensure that storage nodes are actually providing their promised service. Furthermore, storage nodes whose sole aim is to earn cryptocurrency payment have a strong incentive to cheat, perhaps by deleting user data or thwarting audit mechanisms.

*Université Grenoble Alpes, Laboratoire Jean Kuntzmann, UMR CNRS 5224, Grenoble INP. 700 avenue centrale, IMAG-CS 40700, 38058 Grenoble, France. Gaspard.Anthoine@etu.univ-grenoble-alpes.fr, {Jean-Guillaume.Dumas,Aude.Maignan,Clement.Pernet}@univ-grenoble-alpes.fr, dejonghe.melanie63@gmail.com.

†United States Naval Academy, Annapolis, Maryland, United States. mikehanling@gmail.com, roche@usna.edu

1.2 Existing solutions

The research community has developed a wide array of solutions to the remote data integrity problem over the last 15 years. Here we merely summarize the main lines of work and highlight some shortcomings that this paper seeks to address; see Section 7 for a more complete discussion and comparison.

Provable Data Possession (PDP). PDP audits [27, 18, 39, 41] are practically efficient methods to ensure that a large fraction of data has not been modified. They generally work by computing a small *tag* for each block of stored data, then randomly sampling a subset of data blocks and corresponding tags, and computing a check over that subset.

Because a server that has lost or deleted a constant fraction of the file will likely be unable to pass an audit, PDPs are useful in detecting catastrophic or unintentional data loss. They are also quite efficient in practice. However, *a server who deletes only a few blocks is still likely to pass an audit*, so the security guarantees are not complete, and may be inadequate for critical data storage or possibly-malicious providers.

Proof of Retrievability (PoR). PoR audits, starting with [5], have typically used techniques such as error-correcting codes, and more recently Oblivious RAM (ORAM), in order to obscure from the server where pieces of the file are stored [30, 15]. Early PoR schemes did not provide an efficient update mechanism to alter individual data blocks, but more recent *dynamic* schemes have overcome this shortcoming [38, 12].

A successful PoR audit provides a strong guarantee of retrievability: if the server altered many blocks, this will be detected with high probability, whereas if only few blocks were altered or deleted, then the error correction means the file can still likely be recovered. Therefore, a single successful audit ensures with high probability that the *entire* file is still stored by the server.

The downside of this stronger guarantee is that PoRs have typically used more sophisticated cryptographic tools than PDPs, and in all cases we know of require *multiple times the original data size for persistent remote storage*. This is problematic from a cost standpoint: if a PoR based on ORAM requires perhaps 10x storage on the cloud, this cost may easily overwhelm the savings cloud storage promises to provide.

For our purpose, we have identified two main storage outsourcing type of approaches: those which minimizes the storage overhead and those which minimize the client and server computation. For each approach, we specify in Table 1 which one meets various requirements such as whether or not they are dynamic, if they can answer an unbounded number of queries and what is the extra storage they require.

Table 1: Attributes of some selected schemes

Protocol	PoR capable	Number of audits	Number of updates	Extra Storage
Sebé [36]	✗	∞	✗	$o(N)$
Ateniese et al. [5]	✗	∞	✗	$o(N)$
Ateniese et al. [6]	✗	$O(1)$	$O(1)$	$o(N)$
Storj [40]	✓	$O(1)$	∞	$o(N)$
Juels et al. [27]	✓	$O(1)$	✗	$O(N)$
Lavauzelle et al. [30]	✓	∞	✗	$O(N)$
Stefanov et al. [39]	✓	∞	∞	$O(N)$
Cash et al. [12]	✓	∞	∞	$O(N)$
Shi et al. [38]	✓	∞	∞	$O(N)$
Here	✓	∞	∞	$o(N)$

Section 7 gives a detailed comparaison with prior work.

Proof of Replication (PoRep) and others. While our work mainly falls into the PoR/PDP setting, it also has applications to more recent and related notions of remote storage proofs.

Proofs of space were originally proposed as an alternative to the computation-based puzzles in blockchains and anti-abuse mechanisms [4, 16], and require verifiable storage of a large amount of essentially-random data. These are not applicable to cloud storage, where the data must obviously not be random.

A PoRep scheme (sometimes called *Proof of Data Reliability*) aims to combine the ideas of proof of space and PoR/PDP in order to prove that *multiple copies* of a data file are stored remotely. This is important as, for example, a client may pay for 3x redundant storage to prevent data loss, and wants to make sure that three actual copies are stored in distinct locations. Some PoRep schemes employ slow encodings and time-based audit checks; the idea is that a server does not have enough time to re-compute the encoding on demand when an audit is requested, or even to retrieve it from another server, and so must actually store the (redundantly) encoded file [3, 21, 42, 13]. The Filecoin network employs this type of verification. A different and promising approach, not based on timing assumptions, has recently been proposed by [14]. An important property of many recent PoRep schemes is *public verifiability*, that is, the ability for a third party (without secrets) to conduct an audit. This is crucial especially for distributed storage networks (DSNs).

Most relevant for the current paper is that *most of these schemes directly rely on an underlying PDP or PoR* in order to verify encoded replica storage. For example, [14] states that their protocol directly inherits any security and efficiency properties of the underlying PDP or PoR.

We also point out that, in contrast to our security model, many of these works are based on a *rational actor model*, where it is not in a participant's financial interest to cheat, but a malicious user may break this guarantee, and furthermore that most existing PoRep schemes do not support *dynamic* updates to individual data blocks.

1.3 Our Contributions

We present a new proof of retrievability which has the following advantages compared to existing PDPs and PoRs:

Near-optimal persistent storage. The best existing PoR protocols that we could find require between $2N$ and $10N$ bytes of cloud storage to support audits of an N-byte data file, making these schemes impractical in many settings. Our new PoR requires only $N + O(N/\log N)$ persistent storage.

Simple cryptographic building blocks. Our basic protocol relies only on small-integer arithmetic and a collision-resistant hash function, making it very efficient in practice. Indeed, we demonstrate in practice that 1TB of data can be audited in 7 minutes at a monetary cost of just 0.12 USD.

Efficient partial retrievals and updates. That is, our scheme is a *dynamic* PoR, suitable to large applications where the user does not always wish to re-download the entire file.

Provable retrievability from malicious servers. Similar to the best PoR protocols, our scheme supports data recovery (*extraction*) via rewinding audits. This means, in particular, that there is only a negligible chance that a server can pass a *single* audit and yet not recover the entirety of stored data.

(Nearly) stateless clients. With the addition of a symmetric cipher, the client(s) in our protocol need only store a single decryption key and hash digest, which means multiple clients may easily share access (audit responsibility) on the same remote data store.

Public verifiability. We show a variant of our protocol, based on the difficulty of discrete logarithms in large group, that allows any third party to conduct audits with no shared secret.

Importantly, because our protocols store the data unencoded on the server, they can trivially be used within or around any existing encryption or duplication scheme, including most PoRep constructions. We can also efficiently support arbitrary server-side applications, such as databases or file systems with their own encoding needs.

The main drawback of our schemes is that, compared to existing PoRs, they have a higher asymptotic complexity for server-side computation during audits, and (in some cases) higher communication bandwidth during audits as well. However, we also provide a time-space lower bound that proves *any PoR scheme* must make a tradeoff between persistent space and audit computation time.

Furthermore, we demonstrate with a complete implementation and deployment on Google Compute Platform that *the tradeoff we make is highly beneficial in cloud settings*. Intuitively, a user must pay for the computational cost of audits only when they are actually happening, maybe a few times a day, whereas the extra cost of (say) 5x persistent storage *must be paid all the time*, whether the client is performing audits or not.

1.4 Organization

The rest of the paper is structured as follows:

- Section 2 defines our security model, along the lines of most recent PoR works;
- Section 3 contains our proof of an inherent time-space tradeoff in any PoR scheme;
- Section 4 gives an overview and description of our basic protocol, with detailed algorithms and security proofs delayed until Section 6;
- Section 5 discusses the results of our open-source implementation and deployment on Google Compute Platform;
- together with the formal setting, Section 6 also contains a publicly verifiable variant. Section 7 gives a detailed comparison with prior work.

2 Security model

We define a dynamic PoR scheme as consisting of the following five algorithms between a client $\mathcal{C}$ with state $st_{\mathcal{C}}$ and a server $\mathcal{S}$ with state $st_{\mathcal{S}}$. Our definition is the same as given by [38], except that we follow [27] and include the `Extract` algorithm in the protocol explicitly.

A subtle but important point to note is that, unlike the first four algorithms, `Extract` is not really intended to be used in practice. In typical usage, a cooperating and honest server will pass all audits, and the normal `Read` algorithm would be used to retrieve any or all of the data file reliably. The purpose of `Extract` is mostly to prove that the data is recoverable by a series of random, successful audits, and hence that the server which has deleted even one block of data has negligible chance to pass a single audit.

Our definitions rely on two distinct security parameters, κ for computational security and λ for statistical security. Typically values of $\kappa \geq 128$ and $\lambda \geq 40$ are considered secure [19]. One may think of κ having to do with offline attacks and λ corresponding only to online attacks which require interaction and where the adversary is more limited. Carefully tracking both security parameters in our analysis will allow us to more tightly tune performance without sacrificing security.

The client may use random coins for any algorithm; at a minimum, the `Audit` algorithm *must* be randomized in order to satisfy retrievability non-trivially.

- $(st_{\mathcal{C}}, st_{\mathcal{S}}) \leftarrow \texttt{Init}(1^{\kappa}, 1^{\lambda}, b, M)$: On input of the security parameters and the database M, consisting of N bits arranged in blocks of b bits, outputs the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$.
- $\{m_i, \texttt{reject}\} \leftarrow \texttt{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of an index $i \in 1..\lceil N/b \rceil$, the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs $m_i = M[i]$ or `reject`.
- $\{(st'_{\mathcal{C}}, st'_{\mathcal{S}}), \texttt{reject}\} \leftarrow \texttt{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of an index $i \in 1..\lceil N/b \rceil$, data a, the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs a new client state $st'_{\mathcal{C}}$ and a new server state $st'_{\mathcal{S}}$, such that now $M[i] = a$, or `reject`.
- $\{\pi, \texttt{reject}\} \leftarrow \texttt{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs a successful transcript π or `reject`.
- $M \leftarrow \texttt{Extract}(st_{\mathcal{C}}, \pi_1, \pi_2, \ldots, \pi_e)$: On input of independent `Audit` transcripts $\pi_1, \ldots, \pi_e$, outputs the database M. The number of required transcripts e must be a polynomially-bounded function of N, b, and κ.

2.1 Correctness

A correct execution of the algorithms by honest client and server results in audits being accepted and reads to recover the last updated value of the database. More formally, correctness is:

Definition 1 (Correctness)**.** *For any parameters κ, λ, N, b, there exists a predicate* `IsValid` *such that, for any database M of N bits,* $\texttt{IsValid}(M, \texttt{Init}(1^{\kappa}, 1^{\lambda}, b, M))$*. Furthermore, for any state such that* $\texttt{IsValid}(M, st_{\mathcal{C}}, st_{\mathcal{S}})$ *and any index i with $0 \leq i < \lceil N/b \rceil$, we have*

- $\texttt{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}}) = M[i]$*;*
- $\texttt{IsValid}(M', \texttt{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}}))$*, where $M'[i] = a$ and the remaining $M'[j] = M[j]$ for every $j \neq i$;*

- $\texttt{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}}) \neq \texttt{reject}$;
- *For e audits* $\texttt{Audit}_1, \ldots, \texttt{Audit}_e$ *with independent randomness, with probability* $1 - \mathsf{negl}(\lambda)$*:* $\texttt{Extract}(st_{\mathcal{C}}, \texttt{Audit}_1(st_{\mathcal{C}}, st_{\mathcal{S}}), \ldots, \texttt{Audit}_e(st_{\mathcal{C}}, st_{\mathcal{S}}))=M$.

Note that, even though $\mathcal{C}$ may use random coins in the algorithms, a correct PoR by this definition should have no chance of returning `reject` in any `Read`, `Write` or `Audit` with an honest client and server.

2.2 Authenticity and attacker model

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 an observer $\mathcal{O}$, a potentially *malicious* server $\bar{\mathcal{S}}$ and an honest server $\mathcal{S}$ for the adaptive version of authenticity, with the same game as [38]:

1. $\bar{\mathcal{S}}$ chooses an initial memory M. $\mathcal{O}$ runs Init and sends the initial memory layout $st_{\mathcal{S}}$ to both $\bar{\mathcal{S}}$ and $\mathcal{S}$.
2. For a polynomial number of steps $t = 1, 2, ..., poly(\lambda)$, $\bar{\mathcal{S}}$ picks an operation op_t where operation op_t is either `Read`, `Write` or `Audit`. $\mathcal{O}$ executes the operations with both $\bar{\mathcal{S}}$ and $\mathcal{S}$.
3. $\bar{\mathcal{S}}$ is said to win the game, if any message sent by $\bar{\mathcal{S}}$ differs from that of $\mathcal{S}$ and $\mathcal{O}$ did not output `reject`.

Definition 2 (Authenticity)**.** *A PoR scheme satisfies adaptive authenticity, if no polynomial-time adversary $\bar{\mathcal{S}}$ has more than negligible probability in winning the above security game.*

2.3 Retrievability

Intuitively, the retrievability requirement stipulates that whenever a malicious server can pass the audit test with high probability, the server must know the entire memory contents M. To model this, [12] use a *black-box rewinding access*: from the state of the server before any passed audit, there must exist an extractor algorithm that can reconstruct the complete correct database. As in [38], we insist furthermore that the extractor does not use the complete server state, but only the transcripts from successful audits. In the following game, note that the observer $\mathcal{O}$ running the honest client algorithms may only update its state $st_{\mathcal{C}}$ during `Write` algorithm, and hence the `Audit` algorithms are independently randomized from the client side, but we make no assumptions about the state of the adversary $\bar{\mathcal{S}}$.

1. $\bar{\mathcal{S}}$ chooses an initial database M. $\mathcal{O}$ runs `Init` and sends the initial memory layout $st_{\mathcal{S}}$ to $\bar{\mathcal{S}}$;
2. For $t = 1, 2, ..., poly(\lambda)$, the adversary $\bar{\mathcal{S}}$ adaptively chooses an operation op_t where op_t is either `Read`, `Write` or `Audit`. The observer executes the respective algorithms with $\bar{\mathcal{S}}$, updating $st_{\mathcal{C}}$ and M according to the `Write` operations specified;
3. The observer runs e `Audit` algorithms with $\bar{\mathcal{S}}$ and records the outputs $\pi_1, \ldots, \pi_{e'}$ of those which did not return `reject`, where $0 \leq e' \leq e$.
4. The adversary $\bar{\mathcal{S}}$ is said to with the game if $e' \geq e/2$ and $\texttt{Extract}(st_{\mathcal{C}}, \pi_1, \ldots, \pi_e) \neq M$.

Definition 3 (Retrievability)**.** *A PoR scheme satisfies retrievability if no polynomial-time adversary $\bar{\mathcal{S}}$ has more than negligible probability in winning the above security game.*

3 Time-space tradeoff lower bound

As we have seen, the state of the art in Proofs of Retrievability schemes consists of some approaches with a low audit cost but a high storage overhead (e.g., [27, 38, 12]) and some schemes with a low storage overhead but high computational cost for the server during audits (e.g., [5, 36, 37]).

Before presenting our own constructions (which fall into the latter category) we prove that there is indeed an inherent tradeoff in any PoR scheme between the amount of extra storage and the cost of performing audits. By *extra storage* here we mean exactly the number of extra bits of persistent memory, on the client or server, beyond the bit-length of the original database being represented.

Theorem 4 below shows that, for any PoR scheme with sub-linear audit cost, we have

$$(\text{extra storage size}) \cdot \frac{\text{audit cost}}{\log(\text{audit cost})} \in \Omega(\text{data size}). \tag{1}$$

None of the previous schemes, nor those which we present, make this lower bound tight. Nonetheless, it demonstrates that a "best of all possible worlds" scheme with, say, $O(\sqrt{N})$ extra storage and $O(\log N)$ audit cost to store an arbitrary N-bit database, is impossible.

The proof is by contradiction, presenting an attack on an arbitrary PoR scheme which does not satisfy the claimed time/space lower bound. Our attack consists of flipping k randomly-chosen bits of the storage. First we show that k is small enough so that the audit probably does not examine any of the flipped bits, and still passes. Next we see that k is large enough so that, for some choice of the N bits being represented, flipping k bits will, with high probability, make it impossible for any algorithm to correctly recover the original data. This is a contradiction, since the audit will pass even though the data is lost.

Readers familiar with coding theory will notice that the second part of the proof is similar to Hamming's bound for the minimal distance of a block code. Indeed, we can view the original N-bit data as a *message*, and the storage using $s + c$ extra bits of memory as an $(N + s + c)$-bit *codeword*. A valid PoR scheme must be able to extract (*decode*) the original message from an $(N + s + c)$-bit string, or else should fail any audit.

Theorem 4. *For any Proof of Retrievability scheme which stores an arbitrary database of N bits, uses at most $N + s$ bits of persistent memory on the server, c bits of persistent memory on the client, and requires at most t steps to perform an audit. Assuming $s \geq 0$, then either $t > \frac{N}{4}$, or*

$$(s + c)\frac{t}{\log_2 t} \geq \frac{N}{12}. \tag{2}$$

Proof. First observe that $N = 0$ and $t = 0$ are both trivial cases: either the theorem is always true, or the PoR scheme is not correct. So we assume always that $N \geq 1$ and $t \geq 1$.

By way of contradiction, suppose a valid PoR scheme exists with $s \geq 0$, $t \leq \frac{N}{4}$, and

$$(s + c)\frac{t}{\log_2 t} < \frac{N}{12}. \tag{3}$$

Following the definitions in Section 2, we consider only the `Audit` and `Extract` algorithms. The `Audit` algorithm may be randomized and, by our assumption, examines at most t bits of the underlying memory. At any point in an *honest* run of the algorithm, the server stores a $(N + s)$-bit string $st_{\mathcal{S}}$, the client stores a c-bit string $st_{\mathcal{C}}$, and the *client virtual memory* in the language of [12] is the unique N-bit string M such that $\texttt{IsValid}(st_{\mathcal{C}}, st_{\mathcal{S}}, M)$.

Define a map $\phi : \{0,1\}^{N+s+c} \to \{0,1\}^N$ as follows. Given any pair $(st_{\mathcal{C}}, st_{\mathcal{S}})$ of length-$N + s$ and length-c bit strings, run $\texttt{Extract}(st_{\mathcal{C}}, \texttt{Audit}_1(st_{\mathcal{C}}, st_{\mathcal{S}}), \ldots, \texttt{Audit}_e(st_{\mathcal{C}}, st_{\mathcal{S}}))$ repeatedly over all possible choices of randomness, and record the majority result. By Definition 1, we have that $\phi(st_{\mathcal{C}}, st_{\mathcal{S}}) = M$ whenever $\texttt{IsValid}(st_{\mathcal{C}}, st_{\mathcal{S}}, M)$.

Observe that this map ϕ must be onto, and consider, for any N-bit data string M, the preimage $\phi^{-1}(M)$, which is the set of client/server storage configurations $(st_{\mathcal{C}}, st_{\mathcal{S}})$ such that $\phi(st_{\mathcal{C}}, st_{\mathcal{S}}) = M$. By a pigeon-hole argument, there must exist some string M_0 such that

$$\#\phi^{-1}(M_0) \leq \frac{2^{N+s+c}}{2^N} = 2^{s+c}. \tag{4}$$

Informally, M_0 is the data which is most easily corrupted.

We now define an adversary $\bar{\mathcal{S}}$ for the game of Definition 3 as follows: On the first step, $\bar{\mathcal{S}}$ chooses M_0 as the initial database, and uses this in the `Init` algorithm to receive server state $st_{\mathcal{S}}$. Next, $\bar{\mathcal{S}}$ chooses k indices uniformly at random from the $st_{\mathcal{S}}$ of $(N + s)$ bits (where k is a parameter to be defined next), and flips those k bits in $st_{\mathcal{S}}$ to obtain a *corrupted* state $st'_{\mathcal{S}}$. Finally, $\bar{\mathcal{S}}$ runs the honest `Audit` algorithm $2e$ times on step 3 of the security game, using this corrupted state $st'_{\mathcal{S}}$.

What remains is to specify how many bits k the adversary should randomly flip, so that most of the $2e$ runs of the `Audit` algorithm succeed, but the following call to `Extract` does not produce the original database M_0.

Let

$$k = \left\lfloor \frac{N+s}{4t} \right\rfloor. \tag{5}$$

From the assumptions that $s \geq 0$ and $t \leq \frac{N}{4}$, we have that $k \geq 1$.

Let $st_{\mathcal{C}}$ be the initial client state (which is unknown to $\bar{\mathcal{S}}$) in the attack above with initial database M_0. From the correctness requirement (Definition 1) and the definition of t in our theorem, running $\texttt{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}})$ must always succeed after examining at most t bits of $st_{\mathcal{S}}$. Therefore, if the k flipped bits in the corrupted server storage $st'_{\mathcal{S}}$ are not among the (at most) t bits examined by the `Audit` algorithm, it will still pass. By the union bound, the probability that a single run of $\texttt{Audit}(st_{\mathcal{C}}, st'_{\mathcal{S}})$ passes is at least

$$1 - t\frac{k}{N+s} \geq \frac{3}{4}.$$

This means that the expected number of failures in running $2e$ audits is $\frac{e}{2}$, so the Markov inequality tells us that the adversary $\bar{\mathcal{S}}$ successfully passes at least e audits (as required) with probability at least $\frac{1}{2}$.

We want to examine the probability that $\phi(st_{\mathcal{C}}, st'_{\mathcal{S}}) \neq M_0$, and therefore that the final call to `Extract` in the security game does not produce M_0 and the adversary wins with high probability. Because there are $\binom{N+s}{k}$ distinct ways to choose the k bits to form corrupted storage $st'_{\mathcal{S}}$, and from the upper bound of (4) above, the probability that $\phi(st_{\mathcal{C}}, st'_{\mathcal{S}}) \neq M_0$ is at least

$$1 - \frac{2^{s+c} - 1}{\binom{N+s}{k}}. \tag{6}$$

Trivially, if $s + c = 0$, then this probability equals 1. Otherwise, from the original assumption (3), and because $\log_2(4t)/(2t) \leq 1$ for all positive integers t, we have

$$s + c + 2 \leq 3(s+c) < \frac{N \log_2 t}{4t} \leq \left(\frac{N}{4t} - 1\right) \log_2(4t).$$

Therefore

$$\binom{N+s}{k} \geq \left(\frac{N+s}{k}\right)^k > (4t)^{\frac{N+s}{4t} - 1} \geq 2^{s+c+2}.$$

Returning to the lower bound in (6), the probability that the final `Extract` does not return M_0 is at least $\frac{3}{4}$. Combining with the first part of the proof, we see that, with probability at least $\frac{3}{8}$, the attacker succeeds: at least e runs of $\texttt{Audit}(st_{\mathcal{C}}, st'_{\mathcal{S}})$ pass, but the final run of `Extract` fails to produce the correct database M_0. □

4 Retrievability via verifiable computing

We first present a simple version of our PoR protocol. This version contains the main ideas of our approach, namely, using matrix-vector products during audits to prove retrievability. It also makes use of Merkle hash trees during reads and updates to ensure authenticity.

This protocol uses only $N + o(N)$ persistent server storage, which is an improvement to the $O(N)$ persistent storage of existing PoR schemes, and is the main contribution of this work. The costs of our `Read` and `Write` algorithms are similar to existing work, but we incur an asymptotically higher cost for the `Audit` algorithm, namely $O(\sqrt{N})$ communication bandwidth and $O(N)$ server computation time. We demonstrate in the next section that this tradeoff between persistent storage and `Audit` cost is favorable in cloud computing settings for realistic-size databases.

Later, in Section 6, we give a more general protocol and prove it secure according to the PoR definition in Section 2. That generalized version shows how to achieve $O(1)$ persistent client storage with the same costs, or alternatively to trade arbitrarily small communication bandwidth during `Audit`s for increased client persistent storage and computation time.

4.1 Overview

A summary of our four algorithms is shown in Table 2, where dashed boxes are the classical, Merkle hash tree authenticated, remote read/write operations.

Our idea is to use verifiable computing schemes as, e.g., proposed in [20]. Our choice for this is to treat the data as a square matrix of dimension roughly $\sqrt{N} \times \sqrt{N}$. This allows for the matrix multiplication verification described in [22] to be used as a computational method for the audit algorithm.

Crucially, this does not require any additional metadata; the database M is stored as-is on disk, our algorithm merely treats the machine words of this unmodified data as a matrix stored in row-major order. Although the computational complexity for the `Audit` algorithm is asymptotically $O(N)$ for the server, this entails only a single matrix-vector multiplication, in contrast to some prior work which requires expensive RSA computations [5].

To ensure authenticity also during `Read` and `Write` operations, we combine this linear algebra idea above with a standard Merkle hash tree.

Table 2: Client/server PoR protocol with low storage server

	Server	Communications	Client
`Init`		$N = mn \log_2 q$	$\mathbf{u} \xleftarrow{\$} \mathcal{R}_q^m$ $\mathbf{v}^\intercal \leftarrow \mathbf{u}^\intercal \mathbf{M}$.
		MTInit $\longleftarrow \kappa, \lambda, b, \mathbf{M}$	
	$\mathbf{M}, T_{\mathbf{M}} \longleftarrow$	$\longrightarrow r_{\mathbf{M}}$	
	Stores $\mathbf{M}$ and $T_{\mathbf{M}}$		Stores $\mathbf{u}, \mathbf{v}$, and $r_{\mathbf{M}}$
`Read`	$\mathbf{M}, T_{\mathbf{M}} \longrightarrow$	**MTVerifiedRead** $\longleftarrow i, j, r_{\mathbf{M}}$	
		$\longrightarrow \mathbf{M}_{ij}$	
			Returns $\mathbf{M}_{ij}$
`Write`	$\mathbf{M}, T_{\mathbf{M}} \longrightarrow$	**MTVerifiedWrite** $\longleftarrow i, j, \mathbf{M}'_{ij}, r_{\mathbf{M}}$	
	$\mathbf{M}', T'_{\mathbf{M}} \longleftarrow$	$\longrightarrow \mathbf{M}_{ij}, r'_{\mathbf{M}}$	
			$\mathbf{v}'_j \leftarrow \mathbf{v}_j + \mathbf{u}_i(\mathbf{M}'_{ij} - \mathbf{M}_{ij})$
	Stores updated $\mathbf{M}', T'_{\mathbf{M}}$		Stores updated $r'_{\mathbf{M}}, \mathbf{v}'$
`Audit`		$\xleftarrow{\mathbf{x}}$	$\mathbf{x} \xleftarrow{\$} \mathcal{R}_q^n$
	$\mathbf{y} \leftarrow \mathbf{M}\mathbf{x}$	$\xrightarrow{\mathbf{y}}$	$\mathbf{u}^\intercal \mathbf{y} \stackrel{?}{=} \mathbf{v}^\intercal \mathbf{x}$

4.2 Matrix based approach for audits

The basic premise of our particular PoR is to treat the data, consisting of N bits organized in machine words, as a matrix $\mathbf{M} \in \mathcal{R}_q^{m \times n}$, where $\mathcal{R}_q$ is a suitable finite ring of size q. Crucially, the choice of ring $\mathcal{R}_q$ detailed below does not require any modification to the raw data itself; that is, any element of the matrix $\mathbf{M}$ can be retrieved in $O(1)$ time. At a high level, our audit algorithm follows the matrix multiplication verification technique of [22].

In the `Init` algorithm, the Client chooses a secret random control vector $\mathbf{u} \in \mathcal{R}_q^m$ and computes a second secret control vector $\mathbf{v} \in \mathcal{R}_q^n$ according to

$$\mathbf{v}^\intercal = \mathbf{u}^\intercal \mathbf{M}. \tag{7}$$

Note that $\mathbf{u}$ is held constant for the duration of the storage. This does not compromise security because no message which depends on $\mathbf{u}$ is ever sent to the Server. In particular, this means that multiple clients could use different, independent, control vectors $\mathbf{u}$ as long as they have a way to synchronize `Write` operations (modifications of their shared database) over a secure channel.

To perform an audit, the client chooses a random challenge vector $\mathbf{x} \in \mathcal{R}_q^n$, and asks the server to compute a response vector $\mathbf{y} \in \mathcal{R}_q^m$ according to

$$\mathbf{y} = \mathbf{M}\mathbf{x} \tag{8}$$

Upon receiving the response $\mathbf{y}$, the client checks two dot products for equality, namely

$$\mathbf{u}^\intercal \mathbf{y} \stackrel{?}{=} \mathbf{v}^\intercal \mathbf{x}. \tag{9}$$

The proof of retrievability will rely on the fact that observing several successful audits allows, with high probability, recovery of the matrix $\mathbf{M}$, and therefore of the entire database.

The audit algorithm's cost is mostly in the server's matrix-vector product. The client's dot products are much cheaper in comparison. For instance if $m = n$ are close to $\sqrt{N}$, the communication cost is bounded by $O(\sqrt{N})$ as each vector has about $\sqrt{N}$ values. We trade this infrequent heavy computation for no additional persistent storage, justified by the significantly cheaper cost of computation versus storage space.

A sketch of the security proofs is as follows; full proofs are provided along with our formal and general protocol in Section 6. The Client knows that the Server sent the correct value of $\mathbf{y}$ with high probability, because otherwise the Server must know something about the secret control vector $\mathbf{u}$ chosen randomly at initialization time. This is impossible since no data depending on $\mathbf{u}$ was ever sent to the Server. The retrievability property (Definition 3) is ensured from the fact that, after $\sqrt{N}$ random successful audits, with high probability, the original data $\mathbf{M}$ is the unique solution to the matrix equation $\mathbf{MX} = \mathbf{Y}$, where $\mathbf{X}$ is the matrix of random challenge vectors in the audits and $\mathbf{Y}$ is the matrix of corresponding response vectors from the Server.

Some similar ideas were used by [36] for checking integrity. However, their security relies on the difficulty of integer factorization. Implementation would therefore require many modular exponentiations at thousands of bits of precision. Our approach for audits is much simpler and independent of computational hardness assumptions.

4.3 Merkle hash tree for updates

While the audit operates on the data in word-size chunks as members of a finite ring $\mathcal{R}_q$, retrieving data is done at the byte level with support for retrieving any range of bytes (that is legal with the size of the data). A Merkle hash tree with block size b is used here to ensure authenticity of individual `Read` operations. This is a binary tree, stored on the server, consisting of $O(N/b)$ hashes, each of size 2κ for collision resistance.

The Client stores only the root hash, and can perform, with high integrity assurance, any read or write operation on a range of k bytes in $O(k + b + \log(N/b))$ communication and computation time. When the block size is large enough, the extra server storage is $o(N)$; for example, $b \geq \log N$ means the hash tree can be stored using $O(N\kappa/\log N)$ bits.

Merkle hash trees are a classical result, commonly used in practice, and we do not claim any novelty in our use here [31, 29]. To that end, we provide three algorithms to abstract the details of the Merkle hash tree.

These are all two-party protocols between a Server and a Client, but without any requirement for secrecy. A vertical bar | in the inputs and/or outputs of an algorithm indicates Server input/output on the left, and Client input/output on the right. When only the Client has input/output, the bar is omitted for brevity.

The **MTVerifiedRead** and **MTVerifiedWrite** algorithms may both fail to verify a hash, and if so, the Client outputs `reject` and aborts immediately. Our three Merkle tree algorithms are as follows.

MTInit$(1^\kappa, b, M) \mapsto (M, T_M \mid r_M)$. The Client initializes database M for storage in size-b blocks. The entire database M is sent to the Server, who computes hashes and stores the resulting Merkle hash tree T_M. The Client also computes this tree, but discards all hashes other than the root hash r_M. The cost in communication and computation for both parties is bounded by $O(|M|) = O(N)$.

MTVerifiedRead$(M, T_M \mid range, r_M) \mapsto M_{range}$. The Client sends a contiguous byte range to the server, i.e., a pair of indices within the size of M. This range determines which containing range of blocks are required, and sends back these block contents, along with left and right boundary paths in the hash tree T_M. Specifically, the boundary paths include all left sibling hashes along the path from the first block to the root node, and all right sibling hashes along the path from the last block to the root; these are called the "uncles" in the hash tree. Using the returned blocks and hash tree values, the Client reconstructs the Merkle tree root, and compares with r_M. If these do not match, the Client outputs `reject` and aborts. Otherwise, the requested range of bytes is extracted from the (now-verified) blocks and returned. The cost in communication and computation time for both parties is at most $O(|range| + b + \log(N/b))$.

MTVerifiedWrite$(M, T_M \mid range, M'_{range}, r_M)$

$$\mapsto (M', T'_M \mid M_{range}, r'_M).$$

The Client wishes to update the data M'_{range} in the specified range, and receive the *previous value* of that range, M_{range}, as well as an updated root hash r_M. The algorithm begins as **MTVerifiedRead** with the Server sending all blocks to cover the range and corresponding left and right boundary hashes from T_M. After the Client retrieves and verifies the old value M_{range} with the old root hash r_M, she updates the blocks with the new value M'_{range} and uses the same boundary hashes to compute the new root hash r'_M. Separately, the Server updates the underlying database M' in the specified range, then recomputes all affected hashes in T'_M. The asymptotic cost is identical to that for the **MTVerifiedRead** algorithm.

5 Experiments with Google cloud services

As we have seen, compared to other dynamic PoR schemes, our protocol aims at achieving the high security guarantees of PoR, while trading near-minimal persistent server storage for increased audit computation time.

In order to address the practicality of this tradeoff, we implemented and tested our PoR protocol using virtual machines and disks on the Google Cloud Platform service.

Specifically, we address two primary questions:

- What is the monetary cost and time required to perform our $O(N)$ time audit on a large database?
- How does the decreased cost of persistent storage trade-off with increase costs for computation during audits?

Our experimental results are summarized in Tables 4 to 6. For a 1TB data file, the $O(\sqrt{N})$ communication cost of our audit entails less than 12MB of data transfer, and our implementation executes the $O(N)$ audit for this 1TB data file in around 7 minutes for a monetary cost of less than $0.12 USD.

By contrast, just the extra persistent storage required by other existing PoR schemes would cost at least $40 USD or as much as $200 USD per month, not including any computation costs for audits. Thees results indicate that he communication and computation costs of our `Audit` algorithm are not prohibitive in practice despite their unfavorable asymptotics; and furthermore, our solution is the most cost-efficient PoR scheme available when few audits are performed per day.

We also emphasize again that a key benefit to our PoR scheme is its *composability* with existing software, as the data file is left in-tact as a normal file on the Server's filesystem.

The remainder of this section gives the full details of our implementation and experimental setup. The source code is available via the following anonymized github repository: https://anonymous.4open.science/r/c4295d1d-692a-4075-b233-1d7ab5468b43/

5.1 Parameter selection

To balance the bandwidth (protocol communications) and the client computation costs, we represent $\mathbf{M}$ as a square matrix with dimensions $m = n = \sqrt{N/64}$, where the 64 comes from our choice of $\mathcal{R}_q$ corresponding to 64-bit words (see Section 5.2). We also fixed the Merkle tree block size at 8KiB for all experiments and used SHA-512/224 for the Merkle tree hash algorithm. The resulting asymptotic costs for these parameter choices are summarized in Table 3.

Table 3: Proof of retrievability via square matrix verifiable computing

		Server	Comm.	Client
Storage		$N + o(N)$		$O(\sqrt{N})$
Comput.	`Init`	$O(N)$	N	$O(N)$
	`Audit`	$O(N)$	$O(\sqrt{N})$	$O(\sqrt{N})$
	`Read/Write`	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

5.2 Two Prime Calculations

In order to leave the data file unmodified in persistent storage, while allowing constant-time random access to individual matrix elements, we break the data into word-sized (8 byte) blocks, and choose a finite ring $\mathcal{R}_q$ with $q \geq 2^{64}$.

One possibility would be to set q as a prime larger than 2^{64}, but this would entail costly multiple-precision computations for the modular arithmetic. Instead, we chose the ring $\mathcal{R}_q = \mathbb{F}_{p_1} \times \mathbb{F}_{p_2}$ as the direct product of two finite fields, each of large prime order. When $q = p_1 p_2 \geq 2^{64}$, this ensures unique recovery of the database from images in $\mathcal{R}_q$ via Chinese remaindering, and also allows efficient computation without extended precision.

In our implementation, we chose $p_1 = 2^{31} - 1$ and $p_2 = 2^{36} - 5$. That p_1 is a Mersenne prime makes computations with it particularly efficient, but a second Mersenne prime of similar size does not exist. For the actual arithmetic we used the low-level routines provided by the open-source high performance number theory library Flint [24].

This two-prime setup is equivalent to storing two databases $\mathbf{M}_1$ and $\mathbf{M}_2$ in finite fields $\mathbb{F}_{p_1}$ and $\mathbb{F}_{p_2}$ respectively, and so the formal security proof of Theorem 6 applies as long as the smaller prime p_1 is larger than the column dimension n of the database matrix $\mathbf{M}$ (see Section 6 for more details). This means our implementation parameters satisfy the security proof requirements for sizes up to $N = 144$PB.

5.3 Experimental Design

Our implementation provides the `Init`, `Read`, `Write`, and `Audit` algorithms as described in the previous section, including the Merkle hash tree implementation for read/write integrity. As the cost of the first three of these are comparable to prior work, we focused our experiments on the `Audit` algorithm.

We ran two sets of experiments, using virtual machines and disks on Google Cloud's Compute Engine*.

The client machine was a basic f1-micro instance, 1 vCPU with 0.6GB memory, residing in Western Europe. All server virtual machines (VMs) were in the central U.S.: the main server was an n1-standard-2, 2 vCPU with 7.5GB memory, and the parallel VMs running MPI in the second set of experiments were all n1-standard-1 instances with 1 vCPU and 3.75GB memory. The data itself was stored on an attached 1.3TB SSD persistent disk. Test files of size 1GB, 10GB, 100GB, and 1TB were generated with random bytes. The server time in Table 4 measures CPU time only; all other times are "wall time" in actual seconds for operation completion.

*https://cloud.google.com/compute/docs/machine-types.

The client and server processes communicated over a trans-Atlantic TCP connection. As a baseline, we used ping and scp to determine the client-server network connection: it had an average round-trip latency of 101ms and achieved throughput as high as 19.1 MB/sec.

5.4 Audit compared to checksums

For the first set of experiments, we wanted to address the question of how "heavy" the hidden constant in the $O(N)$ is. For this, we compared the cost of performing a single audit, on databases of various sizes, to the cost of computing a cryptographic checksum of the entire database using MD5 or SHA256.

Table 4: Single-threaded experiments on Google Cloud
Values indicate the median number of seconds for a single run. For all except the 1TB column, each experiment was performed 5 times. In all cases, after discarding at most one outlier value, the maximum relative difference between the runs was less than 3%.

Operation		1GB	10GB	100GB	1TB
Init	Server	6.38	72.72	743.99	7728.03
	Wall	6.39	80.77	837.91	8586.08
Audit	Server	4.92	61.93	615.51	6372.50
	Wall	4.98	62.58	616.24	6373.48
MD5	Server	2.15	26.20	264.61	2635.60
	Wall	2.16	39.56	397.35	3976.68
SHA256	Server	6.02	62.77	633.02	6326.87
	Wall	6.03	62.85	633.24	6330.03

In a sense, a cryptographic checksum is another means of integrity check that requires no extra storage, albeit without the malicious server protection that our PoR protocol provides. Therefore, having an audit cost which is comparable to that of a cryptographic checksum indicates the $O(N)$ theoretical cost is not too heavy in practice.

The experiment took place in 4 stages. First, each file was run through the initialization algorithm, including creating the Merkle tree in a second pass. Then, each file was run through the `Audit` algorithm. Third, an MD5 digest was calculated for each file. Finally, a SHA256 digest was computed for each file. A Merkle tree was also created over each file. The results are organized into Table 4.

Per operation, the timings report the CPU time from the server side, and the total wall time from the client side. The difference is due mostly to I/O overhead; even for the audit, the client-side work to compute the two dot products is minimal.

There are two main conclusions to draw from the experiments. The first deals with our `Audit` algorithm following the theoretical bounds that were expected, and the second deals with how the run time compares to that of the hash functions.

Because the server computation time for an audit is $O(N)$, we expect the times to scale linearly, and our results support this. We also see that the running time is consistently between that of MD5 and SHA256 checksums, both in wall time and CPU time. This justifies the fact that the $O(N)$ time `Audit` algorithm, while more costly than other PoR and PDP schemes, is comparable to that of computing a cryptographic checksum.

We were surprised by the large disparity between the Server Time and the Wall Time in these experiments, both for our own `Audit` algorithm and for the checksum comparisons. We determined that this disparity is mostly due to I/O within the cloud datacenter, caused by the CPU waiting for the reads to the external drive.

5.5 Parallel audits using MPI

Our first round of experiments indicated that our `Audit` algorithm on the server was I/O bound, despite the favorable linear access pattern of the matrix-vector product computation. It seems that Google Cloud Platform throttles disk-to-VM I/O on a per-VM basis, so that even with many cores, the situation did not improve.

However, we were able to achieve good parallel speedup when running the `Audit` algorithm over multiple VMs in parallel using MPI. In this setup, the server VM waits for a connection from a client, who requests an audit, which is in turn performed by some number of VMs running in parallel, after which the results are collected and returned to the client. The simplicity of our `Audit` algorithm makes it trivially parallelizable, where each parallel VM performs the matrix-vector product on a contiguous subset of rows of $\mathbf{M}$, corresponding to a contiguous segment of the underlying file.

Because the built-in MD5 and SHA256 checksum programs do not achieve any parallel speedup, we focused only on our `Audit` algorithm for this set of experiments using MPI. The results are reported in Table 5. Our parallel speedup is not quite linear, but was sufficient to gain a significant improvement in the audit time, to just under 7 minutes in the case of a 1TB file using 16 VMs.

We also used these times to measure the total cost of running each audit in Google Cloud Platform, which features per-second billing of VMs and persistent disks, as reported in Table 5 as well. Note that the monetary cost for increasing parallel VMs is slightly decreasing for larger file sizes, indicating that even higher levels of parallelization may decrease the running time even further with no extra monetary cost.

Table 5: Audit with parallel VMs on Google Cloud
Values indicate the median number of seconds for a single run. For all except 4VM/1TB, each experiment was performed 5 times.

VMs	Metric	1GB	10GB	100GB	1TB
1	Audit	4.98	62.58	616.2	6373
	Speedup	1x	1x	1x	1x
	Cost	$0.00007	$0.0009	$0.012	$0.496
4	Audit	2.46	16.97	169.9	1651
	Speedup	2.02x	3.69x	3.63x	3.86x
	Cost	$0.00013	$0.0009	$0.010	$0.194
16	Audit	1.60	4.91	42.00	410.3
	Speedup	3.11x	12.75x	14.67x	15.53x
	Cost	$0.00034	$0.00104	$0.0091	$0.113

5.6 Communication and client computation

Besides the $O(N)$ complexity for server computation during an audit, the $O(\sqrt{N})$ cost of client computation and communication bandwidth in our scheme is also asymptotically worse than existing PoR schemes. However, our experiments suggest that in practice these are not significant factors.

The time it took the client to compute the two dot products to finish the audit never took more than 0.12 seconds in any case tested. This indicates that even low-powered client machines should be able to run this `Audit` algorithm without issue.

The time spent communicating the challenge and response vectors, $\mathbf{x}$ and $\mathbf{y}$, becomes insignificant in comparison to the server computation as the size of the database increases. In the case of our experiments, Table 6 summarizes that communication time of both $\mathbf{x}$ and $\mathbf{y}$ remains under three seconds. The amount of data communicated is also given to confirm the square root scaling.

In this experiment, as before, the client was located in a western European datacenter, while the server (and associated VMs) were co-located in the central United States. Comparing to the total audit times in Table 5, we see that the communication delays are insignificant compared to computation time.

Table 6: Amount of Communication Per Audit with 16 VMs
Values indicate the median number of seconds for a single run. Each experiment was performed 5 times, with a maximum variance of 19% between runs.

Metric	1GB	10GB	100GB	1TB
Comm. (kB)	358	1131	3578	11314
Time (s)	1.20	1.68	2.19	2.71

6 Formalization and Security analysis

In this section we present our PoR protocol in most general form, prove it satisfies the definitions of PoR correctness, authenticity, and retrievability, analyze its asymptotic performance and present a variant that also satisfies public verifiability.

Recall that our security definition and protocol rely on two security parameters: κ for computational security and λ for statistical security. In our main protocol, the only dependence on computational assumptions comes from the use of Merkle trees and the hardness of finding hash collisions. The κ parameter will also arise when we use encryption to extend the protocol for externalized storage and public verifiability.

Instead, the security of our main construction mostly depends on the statistical security parameter λ. Roughly speaking, this is because in order to produce an incorrect result that the client will accept for an audit, the adversary must provably *guess* a result and try it within the *online* audit protocol; even observing correct audits does not help the adversary gain an advantage. This intuition, rigorously analyzed below, allows us to instantiate our protocol more efficiently while providing strong security guarantees.

6.1 Improvements on the control vectors

The control vectors $\mathbf{u}$ and $\mathbf{v}$ stored by the Client in the simplified protocol from Section 4 can be modified to increase security and decrease persistent storage or communications.

Security assumptions via multiple checks. In order to reach a target bound $2^{-\lambda}$ on the probability of failure for the authenticity, it might *theoretically* be necessary to choose multiple independent $\mathbf{u}$ vectors during initialization and repeat the audit checks with each one. We will show that in fact only one vector is necessary for reasonable settings of λ, but perform the full analysis here for completeness.

First, to ease independence considerations we forget the two prime ring and consider instead that tests are performed in $\mathbb{F}_q$, a finite field of size q. Second, we model multiple vectors by inflating the vectors $\mathbf{u}$ and $\mathbf{v}$ to be blocks of t non-zero vectors instead; that is, matrices $\mathbf{U}$ and $\mathbf{V}$ with t rows each. To see how large t needs to be, consider the probability of the Client accepting an incorrect response during an audit. An incorrect answer $\mathbf{z}$ to the audit fails to be detected only if

$$\mathbf{U} \cdot (\mathbf{z} - \mathbf{y}) = \mathbf{0}, \tag{10}$$

where $\mathbf{y} = \mathbf{M}\mathbf{x}$ is the correct response which would be returned by an honest Server.

If $\mathbf{U}$ is sampled uniformly at random among matrices in $\mathbb{F}_q^{t \times m}$ with non-zero rows, then since the Server never learns any information about $\mathbf{U}$, audit fails only if the Server can guess a vector in the right nullspace of $\mathbf{U}$. This happens with probability at most $1/q^t$.

Achieving a probability bounded by $2^{-\lambda}$, requires to set $t = \left\lceil \frac{\lambda}{\log_2(q)} \right\rceil$. In practice, reasonable values of $\lambda = 40$ and $q > 2^{64}$ mean that $t = 1$ is large enough. If an even higher level of security such as $\lambda = 80$ is required, then still only 2 vectors are needed.

Random geometric progression. Instead of using uniformly random vectors $\mathbf{x}$ and matrices $\mathbf{U}$, one can impose a structure on them, in order to reduce the amount of randomness needed, and the cost of communicating or storing them. We propose to apply Kimbrel and Sinha's modification of Freivalds' check [28]:

select a single random field element ρ and form $\mathbf{x}^\intercal = [\rho, \ldots, \rho^n]$, thus reducing the communication volume for an audit from $m+n$ to $m+1$ field elements.

Similarly, we can reduce the storage of $\mathbf{U}$ by sampling uniformly at random t distinct non-zero elements $s_1, \ldots, s_t$ and forming

$$\mathbf{U} = \begin{bmatrix} s_1 & \cdots & s_1^m \\ \vdots & & \vdots \\ s_t & \cdots & s_t^m \end{bmatrix} \in \mathbb{F}_q^{t \times m}. \tag{11}$$

This reduces the storage on the client side from $mt + n$ to only $t + n$ field elements.

Then with a rectangular database and $n > m$, communications can be potentially lowered to any small target amount, at the cost of increased client storage and greater client computation during audits.

This impacts the probability of failure of the authenticity for the audits. Consider an incorrect answer $\mathbf{z}$ to an audit as in (10). Then each element $s_1, \ldots, s_t$ is a root of the degree-$(m-1)$ univariate polynomial whose coefficients are $\mathbf{z} - \mathbf{y}$. Because this polynomial has at most $m-1$ distinct roots, the probability of the Client accepting an incorrect answer is at most

$$\frac{\binom{m-1}{t}}{\binom{q}{t}} \leq \left(\frac{m}{q}\right)^t, \tag{12}$$

which leads to setting $t = \left\lceil \frac{\lambda}{\log_2(q) - \log_2(m)} \right\rceil$ in order to bound this probability by $2^{-\lambda}$. Even if $N = 2^{53}$ for 1PB of storage, assuming $m \leq n$, and again using $\lambda = 40$ and $q \geq 2^{64}$, still $t = 1$ suffices.

Externalized storage. Lastly, the client storage can be reduced to $O(\kappa)$ by externalizing the storage of the block-vector $\mathbf{V}$ at the expense of increasing the volume of communication. Clearly $\mathbf{V}$ must be stored encrypted, as otherwise the server could answer any challenge without having to store the database. Any IND-CPA symmetric cipher works here, with care taken so that a separate IV is used for each column; this allows updates to a column of $\mathbf{V}$ during a `Write` operation without revealing anything about the updated values.

In the following we will thus simply assume that the client has access to an encryption function E_K and a decryption function D_K, both parameterized with a secret key K. In order to assess the authenticity of each communication of the ciphered $\mathbf{V}$ from the Server to the client, we will use another Merkle-Hash tree certificate for it: the client will only need to keep the root of a Merkle-Tree built on the encryption of $\mathbf{V}$. With this, we next show how to efficiently and securely update both the database and this externalized ciphered control vector. Further, this ensures non-malleability outside of the encryption scheme: INT-CTXT (integrity of ciphertexts) together with IND-CPA implies IND-CCA2 [7, Theorem 2].

Since this modification reduces the client storage but increases the overall communication, we consider both options (with or without it; extern=`T` or extern=`F`), and we state the algorithms for our protocol with a *Strategy* parameter, deciding whether or not to externalize the storage of $\mathbf{V}$.

Table 7: Proof of retrievability via rectangular verifiable computing with structured vectors
($N = mn \log_2 q$ is the size of the database, $\kappa \geq \lambda$ are the computational and statistical security parameters, $b > \kappa \log N$ is the Merkle tree block size. Assume $\log_2 q$ is a constant.)

		Server	Communication		Client	
Strategy			extern=`T`	extern=`F`	extern=`T`	extern=`F`
Storage		$N + O(N\kappa/b)$			$O(\kappa)$	$O(n\kappa)$
Comput.	Setup	$O(N)$	$N + o(N)$	N	$O(N)$	
	Audit	N	$O(m + n\kappa)$	$O(m)$	$O(\kappa(m+n))$	
	Read/Write	$O(b + \kappa \log N)$	$O(b + \kappa \log N)$		$O(b + \kappa \log N)$	

6.2 Formal protocol descriptions

Full definitions of the five algorithms, `Init`, `Read`, `Write`, `Audit` and `Extract`, as Algorithms 1 to 5, are given below, incorporating the improvements on control vector storage from the previous subsection. They include subcalls to the classical Merkle hash tree operations defined in Section 4.3.

Then, a summary of the asymptotic costs can be found in Table 7.

Algorithm 1 `Init`$(1^\kappa, 1^\lambda, m, n, q, b, M, Strategy)$

Input: $1^\kappa, 1^\lambda; m, n, q, b \in \mathbb{N}; \mathbf{M} \in \mathbb{F}_q^{m \times n}$
Output: $st_\mathcal{S}$, $st_\mathcal{C}$

$t \leftarrow \lceil \lambda / (\log_2 q) \rceil \in \mathbb{N}$;
Client: $\mathbf{s} \xleftarrow{\$} \mathbb{F}_q^t$ with non-zero distinct elements {Secrets}
Client: Let $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t \times m}$
Client: $\mathbf{V} \leftarrow \mathbf{UM} \in \mathbb{F}_q^{t \times n}$ {Secretly stored or externalized}
Both: $(\mathbf{M}, T_\mathbf{M} \mid r_\mathbf{M}) \leftarrow \mathbf{MTInit}(1^\kappa, b, \mathbf{M})$
if $(Strategy = externalization)$ **then**
    Client: $K \xleftarrow{\$} \mathcal{K}$;
    Client: $\mathbf{W} \leftarrow E_K(\mathbf{V}) \in \mathcal{C}_q^{t \times n}$;
    Client: sends $m, n, q, \mathbf{M}, \mathbf{W}$ to the Server;
    Both: $(\mathbf{W}, T_\mathbf{W} \mid r_\mathbf{W}) \leftarrow \mathbf{MTInit}(1^\kappa, b, \mathbf{W})$
    Server: $st_\mathcal{S} \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy, \mathbf{W}, T_\mathbf{W})$;
    Client: $st_\mathcal{C} \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, K, r_\mathbf{W})$;
else
    Client: sends $m, n, q, \mathbf{M}$ to the Server;
    Server: $st_\mathcal{S} \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy)$;
    Client: $st_\mathcal{C} \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, \mathbf{V})$;
end if

Algorithm 2 `Read`$(st_\mathcal{S}, st_\mathcal{C}, i, j)$

Input: $st_\mathcal{S}$,$st_\mathcal{C}, i \in [1..m], j \in [1..n]$
Output: $\mathbf{M}_{ij}$ or `reject`

Both: $\mathbf{M}_{ij} \leftarrow \mathbf{MTVerifiedRead}(\mathbf{M}, T_\mathbf{M} \mid (i, j), r_\mathbf{M})$
Client: **return** $\mathbf{M}_{ij}$

6.3 Security

Before we begin the full security proof, we need the following technical lemma to prove that the `Extract` algorithm succeeds with high probability. The proof of this lemma is a straightforward application of Chernoff bounds.

Lemma 5. *Let $\lambda, n \geq 1$ and suppose e balls are thrown independently and uniformly into q bins at random. If $e = 4n + 24\lambda$ and $q \geq 4e$, then with probability at least $\exp(-\lambda)$, the number of non-empty bins is at least $e/2 + n$.*

Proof. Let $B_1, B_2, \ldots, B_e$ be random variables for the indices of bins that each ball goes into. Each is a uniform independent over the q bins. Let $X_{1,2}, X_{1,3}, \ldots, X_{e-1,e}$ be $\binom{e}{2}$ random variables for each pair of indices i, j with $i \neq j$, such that $X_{i,j}$ equals 1 iff $B_i = B_j$. Each $X_{i,j}$ is a therefore Bernoulli trial with $\mathbb{E}[X_{i,j}] = \frac{1}{q}$, and the sum $X = \sum_{i \neq j} X_{i,j}$ is the number of pairs of balls which go into the same bin.

Algorithm 3 `Write`$(st_{\mathcal{S}}, st_{\mathcal{C}}, i, j, \mathbf{M}'_{ij}, Strategy)$

Input: $st_{\mathcal{S}}, st_{\mathcal{C}}, i \in [1..m], j \in [1..n], \mathbf{M}'_{ij} \in \mathbb{F}_q$
Output: $st'_{\mathcal{S}}, st'_{\mathcal{C}}$ or `reject`

Both: $(\mathbf{M}', T'_{\mathbf{M}} \mid \mathbf{M}_{ij}, r'_{\mathbf{M}}) \leftarrow \mathbf{MTVerifiedWrite}(\mathbf{M}, T_{\mathbf{M}} \mid (i,j), \mathbf{M}'_{ij}, r_{\mathbf{M}})$
if $(Strategy = externalization)$ **then**
  Both: $\mathbf{W}_{1..t,j} \leftarrow \mathbf{MTVerifiedRead}(\mathbf{W}, T_{\mathbf{W}} \mid (1..t, j), r_{\mathbf{W}})$
  Client: $\mathbf{V}_{1..t,j} \leftarrow D_K(\mathbf{W}_{1..t,j}) \in \mathbb{F}_q^t$;
end if
Client: Let $\mathbf{U}_{1..t,i} \leftarrow [\mathbf{s}_k^i]_{k=1...t} \in \mathbb{F}_q^t$
Client: $\mathbf{V}'_{1..t,j} \leftarrow \mathbf{V}_{1..t,j} + \mathbf{U}_{1..t,i}(\mathbf{M}'_{ij} - \mathbf{M}_{ij}) \in \mathbb{F}_q^t$;
if $(Strategy = externalization)$ **then**
  Client: $\mathbf{W}'_{1..t,j} \leftarrow E_K(\mathbf{V}'_{1..t,j}) \in \mathcal{C}_q^t$
  Both: $(\mathbf{W}', T'_{\mathbf{W}} \mid \mathbf{W}_{1..t,j}, r'_{\mathbf{W}}) \leftarrow \mathbf{MTVerifiedWrite}(\mathbf{W}, T_{\mathbf{W}} \mid (1..t, j), \mathbf{W}'_{1..t,j}, r_{\mathbf{W}})$
  Server: Update $st'_{\mathcal{S}}$ using $\mathbf{M}', T'_{\mathbf{M}}, \mathbf{W}'$, and $T'_{\mathbf{W}}$
  Client: Update $st'_{\mathcal{C}}$ using $r'_{\mathbf{M}}$ and $r'_{\mathbf{W}}$
else
  Server: Update $st'_{\mathcal{S}}$ using $\mathbf{M}'$ and $T'_{\mathbf{M}}$
  Client: Update $st'_{\mathcal{C}}$ using $r'_{\mathbf{M}}$ and $\mathbf{V}'$
end if

We will use a Chernoff bound on the probability that X is large. Note that the random variables $X_{i,j}$ are *not* independent, but they are negatively correlated: when any $X_{i,j}$ equals 1, it only *decreases* the conditional expectation of any other $X_{i',j'}$. Therefore, by convexity, we can treat the $X_{i,j}$'s as independent in order to obtain an upper bound on the probability that X is large.

Observe that $\mathbb{E}[X] = \binom{e}{2}/q < e/8$. A standard consequence of the Chernoff bound on sums of independent indicator variables tells us that $\Pr[X \geq 2\mathbb{E}[X]] \leq \exp(-\mathbb{E}[X]/3)$; see for example [34, Theorem 4.1] or [25, Theorem 1].

Substituting the bound on $\mathbb{E}[x]$ then tells us that $\Pr[X \geq e/4] \leq \exp(-e/24) < \exp(-\lambda)$. That is, with high probability, fewer than $e/4$ pair of balls share the same bin. If n_k denotes the number of bins with k balls, the number of non-empty bins is:

$$\sum_{k=1}^{q} n_k = \left(e - \sum_{k=2}^{q} k n_k\right) + \sum_{k=2}^{q} n_k = e - \sum_{k=2}^{q} (k-1) n_k \geq e - \sum_{k=2}^{q} \binom{k}{2} n_k.$$

The latter is $> \frac{3}{4}e$ with high probability, which completes the proof, since $3e/4 = e/2 + e/4 = e/2 + n + 6\lambda$. □

We now proceed to the main result of the paper.

Theorem 6. *Let $\kappa, \lambda, m, n \in \mathbb{N}$, $\mathbb{F}_q$ a finite field satisfying $q \geq 16n + 96\lambda$ be parameters for our PoR scheme. Then the protocol composed of:*

- *the* `Init` *operations in Algorithm 1;*
- *the* `Read` *operations in Algorithm 2;*
- *the* `Write` *operations in Algorithm 3;*
- *the* `Audit` *operations in Algorithm 4; and*
- *the* `Extract` *operation in Algorithm 5 with $e = 4n + 24\lambda$*

satisfies correctness, adaptive authenticity and retrievability as defined in Definitions 1 to 3.

Algorithm 4 `Audit`$(st_{\mathcal{S}}, st_{\mathcal{C}}, Strategy)$

Input: $st_{\mathcal{S}}$, $st_{\mathcal{C}}$
Output: `accept` or `reject`

Client: $\rho \xleftarrow{\$} \mathbb{F}_q$ and sends it to the Server;
Let $\mathbf{x}^\intercal \leftarrow [\rho^1, \rho^2, \ldots, \rho^n]$
Server: $\mathbf{y} \leftarrow \mathbf{Mx} \in \mathbb{F}_q^m$; {$\mathbf{M}$ from $st_{\mathcal{S}}$}
Server: sends $\mathbf{y}$ to Client;
if $(Strategy = externalization)$ **then**
  Both: $\mathbf{W} \leftarrow \mathbf{MTVerifiedRead}(\mathbf{W}, T_{\mathbf{W}} \mid (1..t, 1..n), r_{\mathbf{W}})$;
  Client: $\mathbf{V} \leftarrow D_K(\mathbf{W}) \in \mathbb{F}_q^{t\times n}$
end if
Client: Let $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t\times m}$
if $(\mathbf{Uy} = \mathbf{Vx})$ **then**
  Client: **return** `accept`
else
  Client: **return** `reject`
end if

Algorithm 5 `Extract`$(st_{\mathcal{C}}, (\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_e, \mathbf{y}_e))$

Input: $st_{\mathcal{C}}$ and $e \geq 4n + 24\lambda$ audit transcripts $(\mathbf{x}_i, \mathbf{y}_i)$, of which more than $e/2$ are successful.
Output: $\mathbf{M}$ or `fail`

$\ell_1, \ldots, \ell_k \leftarrow$ indices of *distinct* successful challenge vectors $\mathbf{x}_{\ell_i}$
if $k < n$ **then**
  return `fail`
end if {Now $\mathbf{X}$ is Vandermonde with distinct points}
Form matrix $\mathbf{X} \leftarrow [\mathbf{x}_{\ell_1} | \cdots | \mathbf{x}_{\ell_n}] \in \mathbb{F}_q^{n\times n}$
Form matrix $\mathbf{Y} \leftarrow [\mathbf{y}\ell_1 | \cdots | \mathbf{y}\ell_n] \in \mathbb{F}_q^{m\times n}$
Compute $\mathbf{M} \leftarrow \mathbf{Y}\,\mathbf{X}^{-1}$
return $\mathbf{M}$

Proof. Correctness comes from the correctness of the Merkle hash tree algorithms, and from the fact that, when all parties are honest, $\mathbf{Uy} = \mathbf{UMx} = \mathbf{Vx}$.

For authenticity, first consider the secret control block vectors $\mathbf{U}$ and $\mathbf{V}$. On the one hand, in the local storage strategy, $\mathbf{U}$ and $\mathbf{V}$ never travel and all the communications by the Client in all the algorithms are independent of these secrets. On the other hand, in the externalization strategy, $\mathbf{U}$ never travels and $\mathbf{V}$ is kept confidential by the IND-CPA symmetric encryption scheme with key K known only by the client. Therefore, from the point of view of the server, it is equivalent, *in both strategies*, to consider either that these secrets are computed during initialization as stated, or that they are only determined *after* the completion of any of the operations.

Now suppose that the server sends an incorrect audit response $\mathbf{z} \neq \mathbf{Mx}$ which the Client fails to reject, and let Let $f \in \mathbb{F}_q[X]$ be the polynomial with degree at most $m-1$ whose coefficients are the entries of $(\mathbf{z} - \mathbf{Mx})$. Then from (10) and (11) in the prior discussion, each of the randomly-chosen values $s_1, \ldots, s_t$ is a root of this polynomial f. Because f has at most $m-1$ distinct roots, the chance that a single s_i is a root of f is at most $(m-1)/q$, and therefore the probability that all $f(s_1) = \cdots = f(s_t) = 0$, is at most $(m/q)^t$.

From the choice of $t = \lceil \lambda / \log_2(q/m) \rceil$, the chance that the Client fails to reject an incorrect audit response is at most $2^{-\lambda}$, which completes the proof of authenticity (Definition 2).

For retrievability, we need to prove that Algorithm 5 succeeds with high probability on the last step of the security game from Definition 3. Because of the authenticity argument above, all successful audit transcripts are valid with probability $1 - \mathsf{negl}(\lambda)$; that is, each $\mathbf{y} = \mathbf{Mx}$ in the input to Algorithm 5. This `Extract` algorithm can find an invertible Vandermonde matrix $\mathbf{X} \in \mathbb{F}_q^{n\times n}$, and thereby recover $\mathbf{M}$

successfully, whenever at least n of the values ρ from challenge vectors $\mathbf{x}$ are distinct.

Therefore the security game becomes essentially this: The experiment runs the honest `Audit` algorithm $e = 4n + 24\lambda$ times, each time choosing a value ρ for the challenge uniformly at random from $\mathbb{F}_q$. The adversary must then select $e/2$ of these audits to succeed, and the adversary wins the game by selecting $e/2$ of the e random audit challenges which contain fewer than n distinct ρ values.

This is equivalent to the balls-and-bins game of Lemma 5, which shows that the `Extract` algorithm succeeds with probability at least $1 - \exp(-\lambda) > 1 - 2^{-\lambda}$ for any selection of $e/2$ out of e random audits. □

6.4 Publicly verifiable variant

These algorithms can also be adapted to support *public verifiability*.

There a first type of client (now called a *Writer*) is authorized to run the `Init`, `Write`, `Read` and `Audit` algorithms, while a second type of client (now called a *Verifier*) can only run the last two. For our protocol, the idea is to provide equality testing on ciphered values, without deciphering. In a group where the discrete logarithm is hard this can be achieved, while preserving security, thanks to the additive homomorphic property of exponentiation. Any linearly homomorphic encryption could also be used, but as we do not need to be able to decipher, exponentiation suffices.

For this, there and in the following, $g^{\mathbf{A}}$, for a matrix $\mathbf{A}$, denotes the exponentiation coefficient by coefficient. Similarly, $\mathbf{W}^{\mathbf{B}}$, as in $(g^{\mathbf{A}})^{\mathbf{B}}$, is actually $\mathbf{W}^{\mathbf{B}} = g^{\mathbf{AB}}$, but this can be computed in the exponents if needed: $\left(g^{[a\ c]}\right)^{[b\ d]^\intercal} = (g^a)^b\,(g^c)^d = g^{ab+cd}$.

For instance on the Externalized strategy of Section 6.1, the informal modifications are:

1. Build a group $\mathbb{G}$ of large prime order p and generator g.
2. `Init`, in Algorithm 1, is run identically, except for two modifications: first, $\mathbf{W}$ is ciphered in $\mathbb{G}$: $\mathbf{W} \leftarrow E(\mathbf{V}) = g^{\mathbf{V}}$; second, the Writer also publishes an encryption of $\mathbf{U}$ as: $\mathbf{K} \leftarrow g^{\mathbf{U}}$ over an *authenticated* channel.
3. All the verifications of the Merkle tree root in Algorithms 2 to 4 remain unchanged, but the Writer must publish the new roots of the trees after each `Write` also over an authenticated and timestamped channel to the Verifiers.
4. Updates to the control vector, in Algorithm 3 are performed homomorphically, without deciphering $\mathbf{W}$: the Writer computes in clear, $\Delta \leftarrow (\mathbf{M}'_{ij} - \mathbf{M}_{ij})\mathbf{U}_{1..t,i}$, then updates $\mathbf{W}'_{1..t,j} \leftarrow \mathbf{W}_{1..t,j} \cdot g^{\Delta}$.
5. The dotproduct verification, in Algorithm 4 is performed also homomorphically: $\mathbf{K}^{\mathbf{y}} \stackrel{?}{=} \mathbf{W}^{\mathbf{x}}$.

Remark 7. *For the discrete logarithm to be hard, one has to increase the size of the field. Using a prime field implemented with a residue number system, the cost of performing the arithmetic grows linearly in the field size. This overhead is compensated by the equivalent decrease in the size of the matrix (recall that for a database of size N bits, we form a $m \times n$ matrix of bit-size $\log_2(q)$, so that $N = mn\log_2(q)$). Further, with an increased size of coefficient domain and classical security parameters, multiple checks are in fact usually not required anymore. So overall, only the setup and the dotproduct times are modified, see Table 9 for the latter with a 283 bits prime.*

The formalization of the obtained modified protocol, thus without multiple checks, is given as a Protocol in Table 8.

Under Linearly Independent Polynomial (LIP) Security [1, Theorem 1][†], the Protocol of Table 8 adds public verifiability to our dynamic proof of retrievability. Indeed, LIP security states that in a group $\mathbb{G}$ of prime order, the values $(g^{P_1(s)}, \ldots, g^{P_m(s)})$ are indistinguishable from a random tuple of the same size, when $P_1, \ldots, P_m$ are linearly independent multivariate polynomials of bounded degree and s is the secret. Therefore, in our modified protocol, each row $g^{\mathbf{U}_i} = \left(g^{\mathbf{s}_i^j}\right)_{j=1..m}$ is indistinguishable from a random tuple of size m since the polynomials X^j, $j = 1..m$ are independent distinct monomials. Then the idea is to reduce

[†]LIP security reduces to the MDDH hypothesis, a generalization of the widely used decision linear assumption [1, 33]

Table 8: Publicly verifiable Client/server PoR protocol with low storage server

	Server	Communications	Client
`Init`		$N = mn\log_2 q$	$s \xleftarrow{\$} S \subseteq \mathbb{Z}_p$
		$\mathbb{G}$ of order p and gen. g	form $\mathbf{u} \leftarrow [\mathbf{s}^j]_{j=1\dots m} \in \mathbb{Z}_p^m$
			$\mathbf{v}^\intercal \leftarrow \mathbf{u}^\intercal\mathbf{M}$, $\mathbf{w}^\intercal \leftarrow g^{\mathbf{v}} \in \mathbb{G}^n$.
		$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \longleftarrow$ **MTInit** $\longleftarrow \kappa, \lambda, b, \mathbf{M}, \mathbf{w}$; $\longrightarrow r_\mathbf{M}, r_\mathbf{w}$	
	Store $\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w}$		Publish $r_\mathbf{M}$, $r_\mathbf{w}$ and $\mathbf{K} = g^\mathbf{u}$
`Read`	$\mathbf{M}, T_\mathbf{M} \longrightarrow$ **MTVerifiedRead**	$\longleftarrow i, j, r_\mathbf{M}$; $\longrightarrow \mathbf{M}_{ij}$	Return $\mathbf{M}_{ij}$
`Write`	$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \longrightarrow$ **MTVerifiedRead**	$\longleftarrow i, j, r_\mathbf{M}, r_\mathbf{w}$; $\longrightarrow \mathbf{M}_{ij}, \mathbf{w}_j$	$\delta \leftarrow \mathbf{u}_i(\mathbf{M}'_{ij} - \mathbf{M}_{ij})$
		$\xleftarrow{i,j,\mathbf{M}'_{ij},\mathbf{w}'_j}$	$\mathbf{w}'_j \leftarrow \mathbf{w}_j \cdot g^\delta$
	Update $\mathbf{M}', T'_\mathbf{M}, \mathbf{w}', T'_\mathbf{w}$		Publish $r'_\mathbf{M}, r'_\mathbf{w}$
`Audit`		$\xleftarrow{r}$	$r \xleftarrow{\$} S \subseteq \mathbb{Z}_p^*$
	$\mathbf{y} \leftarrow \mathbf{M}\mathbf{x}$	form $\mathbf{x} \leftarrow [r^i]_{i=1\dots n} \in \mathbb{Z}_p^n$	
	$\mathbf{w}, T_\mathbf{w} \longrightarrow$ **MTVerifiedRead**	$\longleftarrow r_\mathbf{w}$; $\longrightarrow \mathbf{w}$	$\mathbf{K}^y \stackrel{?}{=} \mathbf{w}^x$

breaking the public verifiability to breaking a discrete logarithm. For this, the discrete logarithm to break will be put inside $\mathbf{U}$.

These modifications give rise to the following Theorem 8. Compared to Theorem 6, this requires the LIP security assumptions and a larger domain of the elements.

Theorem 8. *Under LIP security in a group $\mathbb{G}$ of prime order $p \geq \max\{16n + 96\lambda, m2^{2\kappa}\}$, where discrete logarithms are hard to compute, the Protocol of Table 8 satisfies correctness, adaptive authenticity and retrievability, and is publicly verifiable.*

Proof. In Table 8, Correctness is just to verify the dotproducts, but in the exponents; this is: $\mathbf{K}^y = g^{\mathbf{U}y} = g^{\mathbf{U}\mathbf{M}x} = \mathbf{W}^x$.

Public verifiability is guaranteed as $\mathbf{K}$ and $\mathbf{U}$, as well as the roots $r_\mathbf{M}$ and $r_\mathbf{w}$ of the Merkle trees for $\mathbf{M}$ and $\mathbf{W}$, are public.

Now for Authenticity: first, any incorrect $\mathbf{W}$ is detected by the Merkle hash tree verification. Second, with a correct $\mathbf{W}$, any incorrect y is also detected with high probability, as shown next.

Suppose that there exist an algorithm $\mathcal{A}(\mathbf{M}, \mathbf{K}, \mathbf{W}, r)$ that can defeat the verification with a fake y, with probability ϵ. That is the algorithm produces $\bar{y}$, with $\bar{y} \neq y = \mathbf{M}x$, such that we have the t equations:

$$\mathbf{K}^y = \mathbf{W}^x = \mathbf{K}^{\bar{y}}. \tag{13}$$

We start with the case $t = 1$. Let $A = g^a$ be a DLOG problem.

Then we follow the proof of [17, Lemma 1] and simulate `Init` via the following inputs to the attacker:

- $r \xleftarrow{\$} S \subseteq \mathbb{Z}_p^*$ and let $x = [r, r^2, \dots, r^n]^\intercal$;
- Sample $\mathbf{M} \xleftarrow{\$} S^{m\times n} \subseteq \mathbb{Z}_p^{m\times n}$ and $\mathbf{U} \xleftarrow{\$} S^m \subseteq \mathbb{Z}_p^m$.

- Randomly select also $k \in 1..m$ and, then, compute $\mathbf{K} = g^{\mathbf{U}} A^{\mathbf{e_k}}$, so that $\mathbf{K} = g^{\mathbf{U}+a\mathbf{e_k}}$, where $\mathbf{e_k}$ is the k-th canonical vector of $\mathbb{Z}_p^m$.
- Under LIP security [1, Theorem 3.1], $\mathbf{K}$ is indistinguishable from the distribution of the protocol $(g^{s_i^j})$.
- finally compute $\mathbf{W} = \mathbf{K}^{\mathbf{M}}$, thus also indistinguishable from the distribution of the protocol.

To simulate any number of occurences of `Write`, it is then sufficient to randomly select $\mathbf{M}'_{ij}$. Then compute and send to the attacker: $\mathbf{W}'_{1..t,j} \leftarrow \mathbf{W}_{1..t,j} \cdot K_{1..t,i}^{\mathbf{M}'_{ij}-\mathbf{M}_{ij}}$ (since $g^{\Delta} = g^{(\mathbf{M}'_{ij}-\mathbf{M}_{ij})\mathbf{U}_{1..t,i}} = K_{1..t,i}^{\mathbf{M}'_{ij}-\mathbf{M}_{ij}}$).

After that, the attacker answers an `Audit`, with $\bar{y} \neq y$ satisfying Equation (13). This is $g^{(\mathbf{U}+a\mathbf{e_k})\bar{y}} = g^{(\mathbf{U}+a\mathbf{e_k})\mathbf{M}x}$, equivalent to:

$$(\mathbf{U} + a\mathbf{e_k})(\bar{y} - \mathbf{M}x) \equiv 0 \mod p. \tag{14}$$

Since $\bar{y} \neq y \mod p$, then there is at least one index $1 \leq j \leq m$ such that $\bar{y}_j \neq y_j \mod p$. Since k is randomly chosen from $1..m$, the probability that $\bar{y}_k \neq y_k \mod p$ is at least $1/m$. If this is the case then with $z = \bar{y} - y$, we have $z_k \neq 0 \mod p$ and $\mathbf{U}z + az_k \equiv 0 \mod p$, so that $a \equiv -z_k^{-1}\mathbf{U}z \mod p$. This means that the discrete logarithm is broken with advantage $\geq \epsilon/m$.

Finally for any $t \geq 1$ the proof is similar except that A is put in different columns for each of the t rows of $\mathbf{U}$. Thus the probability to hit it becomes $\geq t/m$ and the advantage is $\geq t\epsilon/m \geq \epsilon/m$. This gives the requirement that $p \geq m2^{2\kappa}$ to sustain the best generic algorithms for DLOG.

Retreivability comes from the fact that y and x are public values. Therefore this part of the proof is identical to that of Theorem 6. □

Remarks 9.
- *If a writer wants to verify, she does not need to use the public key* $\mathbf{K}$*, nor to store it. Just compute* $\mathbf{U}y$ *directly, then check that* $g^{\mathbf{U}y} \stackrel{?}{=} \mathbf{W}^x$.
- *Even if* $\mathbf{U}$ *is structured,* $\mathbf{K}$ *hides this structure and therefore requires a larger storage. But any Verifier can just fetch it and* $r_{\mathbf{W}}$ *from the authenticated channel (for instance, electronically signed), as well as fetch* $\mathbf{W}$ *from the Server, and perform the verification on the fly. Optimal communications for the Verifier are then when* $m = O(\sqrt{N}) = n$.
- *To save some constant factors in communications, sending* $\mathbf{W}$ *or any of its updates* $\mathbf{W}'_{i,j}$ *is not mandatory anymore: the Server can now recompute them directly from* $\mathbf{M}$*,* $\mathbf{K}$ *and* $\mathbf{M}'$.

In terms of performance, the only modifications are for the client computation time, in the `Audit` part. Table 9 shows the difference between the private verification of Algorithm 4 and the public one (last line/column of Table 8). Public verification is more expensive but this remains doable in a few seconds or minutes even on a constrained device.

Table 9: Out-of-the-box Client average computation time for private and public audits using PBC-F type curves (Baretto-Naerig curves of embedding degree 12 from crypto.stanford.edu/pbc) on the client machine of Section 5. Times are median values of 11 runs, maximum relative difference with the median was 4.12% for the private audits and 0.87% for the public ones.

N	κ	$\log_2(q)$	$m = n$	private	public	
1GB	135.3	283	5510	0.004s	5.0s	PBC-F283
10GB	134.4	283	17422	0.014s	16.0s	
100GB	133.6	283	55094	0.044s	50.5s	
1TB	132.8	283	176300	0.141s	161.7s	

7 Detailed state of the art

PDP schemes, first introduced by [5] in 2007, originally only considered static data storage. The original scheme was later adapted to allow dynamic updates by [18] and has since seen numerous performance

improvements. However, PDPs only guarantee (probabilistically) that a *large fraction* of the data was not altered; a single block deletion or alteration is likely to go undetected in an audit.

PoR schemes, first introduced at the same CCS conference in 2007 by [27], provide a stronger guarantee of integrity: namely, that any small alteration to the data is likely to be detected. In this paper, we use the term PoR to refer to any scheme which provides this stronger level of recoverability guarantee.

PoR and PDP are usually constructed as a collection of phases in order to initialize the data storage, to access it afterwards and to audit the server's storage. Dynamic schemes also propose a modification of subsets of data, called write or update.

Since 2007, different schemes have been proposed to serve different purposes such as data confidentiality, data integrity, or data availability, but also freshness and fairness. Storage efficiency, communication efficiency and reduction of disk I/O have improved with time. Some schemes are developed for static data (no update algorithm) , others extend their audit algorithm for public verification, still others require a finite number of Audits and Updates. For a complete taxonomy on recent PoR schemes, see [41] and references therein.

7.1 Low storage overhead

The schemes of Ateniese et al. [5] or Sebé et al. [36] are in the PDP model. Both of them have a storage overhead in $o(N)$. They use the RSA protocol in order to construct homomorphic authenticators, so that a successful audit guaranties data possession on some selected blocks. When all the blocks are selected, the audit is deterministic but the computation cost is high. So in practice, [5] minimizes the file block accesses, the computation on the server, and the client-server communication. For one audit on at most f blocks,the S-PDP protocol of [5] gives the costs seen in Table 10. A robust auditing integrates S-PDP with a forward error-correcting codes to mitigate arbitrary small file corruption. Nevertheless, if the server passes one audit, it guarantees only that a portion of the data is correct.

Table 10: S-PDP on f blocks : The file M is composed of N/b blocks of bit-size b. The computation is made mod Q, where Q is the product of two large prime numbers.

		Server	Communication	Client
Storage		$N+m$		$O(1)$
Comput.	Setup		$N+f$	$O(bf)$
	Audit	$O(f)$	$O(1)$	$O(f)$

Later, Ateniese et al. [6] proposed a scheme secure under the random oracle model based on hash functions and symmetric keys. It has an efficient update algorithm but uses tokens which impose a limited number of audits or updates.

Alternatively, verifiable computing can be used to go through the whole database with Merkle hash trees, as in [9, §6]. The latter proposition however comes with a large overhead in homomorphic computations and does not provide an Audit mechanism. Verifiable computing can provide an audit mechanism, as sketched in the following paper [20], but then it is not dynamic anymore.

Storj [40] (version 2) is a very different approach also based on Merkle hash trees. It is a dynamic PoR protocol with bounded Audits. The storage is encrypted and cut into m blocks of size b. For each block and for a selection of σ salts, a Merkle Hash tree with σ leaves is constructed. The efficiency of Storj is presented Table 11.

Storj allows only a fixed number of audits (the number of seeds) before the entire data must be re-downloaded to restart the computation. This is a cost of $O(N\sigma)$ operations for the client every σ audits, and thus an average cost of $O(N)$. Our PoR supports unlimited and fast audits, of cost always $O(\log n)$.

Table 11: Storj-V2: The file M is composed of N/b blocks of bit-size b. σ is the number of salts.

		Server	Comm.	Client
Storage		$N+O(\frac{N}{b}\sigma)$		$O(\frac{N}{b}\sigma)$
Comput.	Setup	$O(N+\frac{N}{b}\sigma)$	$N+O(\frac{N}{b}\sigma)$	$O(N\sigma)$
	Avg. Audit		$O(\frac{N}{b}\log\sigma+\frac{N}{\sigma})$	$O(N)$
	Update		$b+O(\sigma)$	$O(b\sigma)$

7.2 Fast audits but large extra storage

PoR methods based on block erasure encoding are a class of methods which guarantee with a high probability that the client's entire data can be retrieved. The idea is to check the authenticity of a number of erasure encoding blocks during the data recovery step but also during the audit algorithm. Those approaches will not detect a small amount of corrupted data. But the idea is that if there are very few corrupted blocks, they could be easily recovered via the error correcting code.

Lavauzelle et al., [30] proposed a static PoR. The Init algorithm consists in encoding the file using a lifted q-ary Reed-Solomon code and encrypting it with a block-cipher. The Audit algorithm checks if one word of q blocks belongs to a set of Reed-Solomon code words. This algorithm has to succeed a sufficient number of times to ensure with a high probability that the file can be recovered. Its main drawback is that it requires an initialization quadratic in the database size. For a large data file of several terabytes this becomes intractable.

In addition to a block erasure code, PoRSYS of Juels et al. [27] use block encryptions and sentinels in order to store static data with a cloud server. Shacham and Waters [37] use authenticators to improve the audit algorithm. A publicly verifiable scheme based on the Diffie-Hellman problem in bilinear groups is also proposed.

Stefanov et al. [39] were the first to consider a dynamic PoR scheme. Later improvements by Cash et al. or Shi et al. [12, 38] allow for dynamic updates and reduce the asymptotic complexity (see Table 12). However, these techniques rely on computationally-intensive tools, such as locally decodable codes and Oblivious RAM (ORAM), and incur at least a 1.5x, or as much as 10x, overhead on the size of remote storage.

Recent variants include *Proof of Data Replication* or *Proof of Data Reliability*, where the error correction is performed by the server instead of the client [3, 42]. Some use a weaker, *rational*, attacker model [32, 13], and in all of them the client thus has to also be able to verify the redundancy; but we do not know of dynamic versions of these.

Table 12: Shi et al. [38]: The file M is composed of $\frac{N}{b}$ blocks of bit-size b.

		Server	Communication	Client
Storage		$O(N)$		$O(b)$
Comput.	Setup		$N+O(\frac{N}{b})$	$O(N\log N)$
	Audit	$O(b\log N)$	$O(b+\log N)$	$O(b+\log N)$
	Update	$O(b\log N)$	$O(b+\log N)$	$O(b+\log N)$

Table 13 compares the additional server storage and audit costs between [38] and the two variants of our protocol: the first one saving on communication, and the second one, externalizing the storage of the secret audit matrix V. In the former case, an arbitrary parameter α can be used in the choice of the dimensions: $m = N^{\alpha}$ and $n = N^{1-\alpha}/\log_2(q)$. This balances between the communication cost $O(N^{\alpha})$ and the Client computation and storage $O(N^{1-\alpha})$.

Note that efficient solutions to PoR for dynamic data do not consider the confidentiality of the file M, but assume that the user can encrypt its data in a prior step if needed.

Table 13: Comparison of our low server storage protocol with that of Shi et al. [38].

	Shi et al. [38]	Here extern=T	Here extern=F
Server extra-storage	$5N$	$o(N)$	$o(N)$
Server audit cost	$O(b \log N)$	$N+o(N)$	$N+o(N)$
Communication	$O(b+\log N)$	$O(\sqrt{N})$	$O(N^{\alpha})$
Client audit cost	$O(b+\log N)$	$O(\sqrt{N})$	$O(N^{1-\alpha})$
Client storage	$O(b)$	$O(1)$	$O(N^{1-\alpha})$

8 Conclusion

We presented new protocols for dynamic Proof of Retrievability, based on randomized linear algebra verification schemes over a finite ring. Our protocols does not require any encoding of the database and is therefore near optimal in terms of persistent storage on the server side. They include also efficient unlimited partial retreivals and updates as well as provable retreivability from malicious servers. They are implementable with simple cryptographic building blocks and are very efficient in practice as shown for instance on a Google Compute platform instance. With the addition of any IND-CPA symmetric cipher the clients become nearly stateless; adding a group where the discrete logarithm is hard also enables a public verification.

On the one hand, private proofs are very fast, less than a second on constrained devices. On the other hand, while still quite cheap, the public verification could nonetheless be improved. Precomputations of multiples of elements of $\mathbf{K}$ and $\mathbf{U}$, combined with dedicated methods for dotproduct in the exponents (generalizing of Shamir's trick for simultaneous exponentiations) might improve the running time. More generally, our verification is a dotproduct, or a polynomial evaluation when the control vectors are structured. This verification itself could be instead computed on the Server side and only verified by a client in a recursive manner, using for instance succinct non-interactive arguments of knowledge, like [8].

References

[1] Michel Abdalla, Fabrice Benhamouda, and Alain Passelègue. An algebraic framework for pseudorandom functions and applications to related-key security. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 388–409, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-47989-6_19.

[2] Lawrence Abrams. Amazon AWS Outage Shows Data in the Cloud is Not Always Safe. *Bleeping Computer*, September 2019.

[3] Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In *25th USENIX Security Symposium*, pages 1051–1068, Austin, TX, August 2016. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/armknecht.

[4] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of Space: When Space Is of the Essence. In *Security and Cryptography for Networks*, pages 538–557. Springer, 2014. doi:10.1007/978-3-319-10879-7_31.

[5] Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary Peterson, and Dawn Song. Provable data possession at untrusted stores. In *14th ACM CCS*, pages 598–609. ACM, 2007. doi:10.1145/1315245.1315318.

[6] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *4th international conference on Security and privacy in communication networks*, page 9. ACM, 2008. doi:10.1145/1460877.1460889.

[7] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, pages 531–545, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. doi:10.1007/s00145-008-9026-x.

[8] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. DEEP-FRI: Sampling Outside the Box Improves Soundness. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, volume 151 of *LIPIcs*, pages 5:1–5:32, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.ITCS.2020.5.

[9] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, pages 111–131, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22792-9_7.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sakura: A flexible coding for tree hashing. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, pages 217–234, Cham, 2014. Springer International Publishing.

[11] Erik Cambria, Anupam Chattopadhyay, Eike Linn, Bappaditya Mandal, and Bebo White. Storages are not forever. *Cognitive Computation*, 9:646–658, 2017. doi:10.1007/s12559-017-9482-4.

[12] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. *J. Cryptol.*, 30(1):22–57, January 2017. doi:10.1007/s00145-015-9216-2.

[13] Ethan Cecchetti, Ben Fisch, Ian Miers, and Ari Juels. Pies: Public incompressible encodings for decentralized storage. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019, London, UK, November 11-15, 2019*, pages 1351–1367. ACM, 2019. doi:10.1145/3319535.3354231.

[14] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. Proofs of replicated storage without timing assumptions. In *Advances in Cryptology – CRYPTO 2019*, pages 355–380. Springer, 2019. doi:10.1007/978-3-030-26948-7_13.

[15] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography*, pages 109–127. Springer, 2009. doi:10.1007/978-3-642-00457-5_8.

[16] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology – CRYPTO 2015*, pages 585–605. Springer, 2015. doi:10.1007/978-3-662-48000-7_29.

[17] Kaoutar Elkhiyaoui, Melek Önen, Monir Azraoui, and Refik Molva. Efficient techniques for publicly verifiable delegation of computation. In *11th ACM AsiaCCS*, pages 119–128, New York, NY, USA, 2016. ACM. doi:10.1145/2897845.2897910.

[18] C. Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Trans. Inf. Syst. Secur.*, 17(4):15:1–15:29, April 2015. doi:10.1145/2699909.

[19] David Evans, Vladimir Kolesnikov, and Mike Rosulek. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2(2-3):70–246, 2018. URL: http://dx.doi.org/10.1561/3300000019, doi:10.1561/3300000019.

[20] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *ACM CCS*, pages 501–512, New York, NY, USA, 2012. ACM. doi:10.1145/2382196.2382250.

[21] Ben Fisch. PoReps: Proofs of Space on Useful Data. Technical Report 678, IACR Cryptology ePrint Archive, 2018. URL: http://eprint.iacr.org/2018/678.

[22] Rūsiņš Freivalds. Fast probabilistic algorithms. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 57–69, Olomouc, Czechoslovakia, September 1979. Springer-Verlag. doi:10.1007/3-540-09526-8_5.

[23] Alissa Greenberg. Google Lost Data After Lightning Hit Its Data Center in Belgium. *Time*, August 2015.

[24] W. B. Hart. Fast Library for Number Theory: An Introduction. In *Third International Congress on Mathematical Software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. http://flintlib.org.

[25] Nick Harvey. Chernoff bound, balls and bins, congestion minimization. Lecture 3 from CPSC 536N: Randomized Algorithms, 2015. URL: https://www.cs.ubc.ca/~nickhar/W15/Lecture3Notes.pdf.

[26] Markus Jakobsson, Frank Thomson Leighton, Silvio Micali, and Michael Szydlo. Fractal merkle tree representation and traversal. In Marc Joye, editor, *Topics in Cryptology - CT-RSA 2003, The Cryptographers' Track at the RSA Conference 2003, San Francisco, CA, USA, April 13-17, 2003, Proceedings*, volume 2612 of *Lecture Notes in Computer Science*, pages 314–326. Springer, 2003. doi:10.1007/3-540-36563-X_21.

[27] Ari Juels and Burton S Kaliski Jr. Pors: Proofs of retrievability for large files. In *14th ACM CCS*, pages 584–597. ACM, 2007. doi:10.1145/1315245.1315317.

[28] Tracy Kimbrel and Rakesh Kumar Sinha. A probabilistic algorithm for verifying matrix products using $O(n^2)$ time and $\log_2 n + O(1)$ random bits. *Information Processing Letters*, 45(2):107–110, February 1993. doi:10.1016/0020-0190(93)90224-W.

[29] B. Laurie, A. Langley, E. Kasper, and Google. Certificate Transparency. RFC 6962, IETF, June 2013. URL: https://tools.ietf.org/html/rfc6962.

[30] Julien Lavauzelle and Françoise Levy dit Vehel. New proofs of retrievability using locally decodable codes. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 1809–1813, July 2016. doi:10.1109/ISIT.2016.7541611.

[31] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg. doi:10.1007/3-540-48184-2_32.

[32] Tal Moran and Ilan Orlov. Simple proofs of space-time and rational proofs of storage. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology - CRYPTO 2019, Santa Barbara, CA, USA, August 18-22*, volume 11692 of *Lecture Notes in Computer Science*, pages 381–409. Springer, 2019. doi:10.1007/978-3-030-26948-7_14.

[33] Paz Morillo, Carla Ràfols, and Jorge L. Villar. The kernel matrix Diffie-Hellman assumption. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 729–758, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. doi:10.1007/978-3-662-53887-6_27.

[34] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995. doi:10.1017/CBO9780511814075.

[35] David Reinsel, John Gantz, and John Rydning. The Digitization of the World from Edge to Core. Technical Report US44413318, "International Data Corporation (IDC)", 2018.

[36] Francesc Sebé, Josep Domingo-Ferrer, Antoni Martínez-Ballesté, Yves Deswarte, and Jean-Jacques Quisquater. Efficient remote data possession checking in critical information infrastructures. *IEEE Transactions on Knowledge and Data Engineering*, 20:1034–1038, 2008. doi:10.1109/TKDE.2007.190647.

[37] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 90–107. Springer, 2008. doi:10.1007/978-3-540-89255-7_7.

[38] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336, New York, NY, USA, 2013. ACM. URL: http://elaineshi.com/docs/por.pdf, doi:10.1145/2508859.2516669.

[39] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: A scalable cloud file system with efficient integrity checks. In *28th ACSAC*, pages 229–238. ACM, 2012. doi:10.1145/2420950.2420985.

[40] Storj labs Inc. Storj: A decentralized cloud storage network framework. Technical Report v2, 2016. URL: https://storj.io/storjv2.pdf.

[41] Choon Beng Tan, Mohd Hanafi Ahmad Hijazi, Yuto Lim, and Abdullah Gani. A survey on proof of retrievability for cloud data integrity and availability: Cloud storage state-of-the-art, issues, solutions and future trends. *J. Network and Comp. Applications*, 110:75–86, 2018. doi:10.1016/j.jnca.2018.03.017.

[42] Dimitrios Vasilopoulos, Melek Önen, and Refik Molva. PORTOS: Proof of data reliability for real-world distributed outsourced storage. In *16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT,*, pages 173–186. INSTICC, SciTePress, 2019. doi:10.5220/0007927301730186.

A Requirements for a Merkle hash tree implementation and overview of the formalized protocol with the externalization strategy

Table 14 presents an overview of the fully formalized protocol.

This table is a merge of the algorithms in Section 6, for the *Externalization* strategy. Its correctness, authenticity and retrievability are proven in Theorem 6. It uses two Merkle hash trees, one for the database M and one for the externalized control vectors $\mathbf{W}$.

We here give more details on the functions required in Section 4.3 for the handling of the Merkle hash trees. A Merkle tree [31] is a tree where the value associated with a node is a one-way function of the values of the node's children. We here consider only binary Merkle Hash trees.

For our purpose, an implementation of such trees must provide the following algorithms:

- $T \leftarrow$ **MTCreateTree**(X) creates a Merkle hash tree from a database X.
- $r \leftarrow$ **MTRootFromLeaves**(X) computes the root of the Merkle hash tree of the whole database X.
- $(L_1, L_2) \leftarrow$ **MTElementAndPath**$(index, range, X, T)$ is an algorithm providing the client with the requested list L_1 of contiguous *leaf elements* $X_{i=index, j \in range}$, together with
  the list L_2 constituted by the blocks containing $X_{i=index, j \in range}$ and by the corresponding lists of Merkle tree uncles.
- $r \leftarrow$ **MTRootFromPath**$(index, range, L_1, L_2)$ computes the root of the Merkle hash tree from a list L_1 of contiguous leaf elements and the associated blocks and path of uncles L_2.

Table 14: Externalized PoR

	Server	Communications	Client
`Init`		DB with N bits	$\mathbf{M} \in \mathbb{F}_q^{m \times n}$
		$\xleftarrow{1^\lambda, m, n, q, b}$	$r_{\mathbf{M}} \leftarrow \mathbf{MTRootFromLeaves}(\mathbf{M})$
	Stores $\mathbf{M}$	$\xleftarrow{\mathbf{M}}$	$t \leftarrow \lceil \lambda / \log_2(q) \rceil$
	$T_{\mathbf{M}} \leftarrow \mathbf{MTCreateTree}(\mathbf{M})$		$\mathbf{s} \xleftarrow{\$} S^t \subseteq \mathbb{F}_q^t$
			Form $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t \times m}$
			$K \xleftarrow{\$} \mathcal{K}$
	Stores $\mathbf{W}$	$\xleftarrow{\mathbf{W}}$	$\mathbf{V} \leftarrow \mathbf{UM}$, $\mathbf{W} \leftarrow E_K(\mathbf{V})$
	$T_{\mathbf{W}} \leftarrow \mathbf{MTCreateTree}(\mathbf{W})$		$r_{\mathbf{W}} \leftarrow \mathbf{MTRootFromLeaves}(\mathbf{W})$
			discard $\mathbf{M}, \mathbf{V}, \mathbf{W}$
`Read`		$\xleftarrow{i,j}$	
$\mathbf{M}_{ij}$	$(\mathbf{M}_{i,j}, L_{\mathbf{M}}) \leftarrow \mathbf{MTElementAndPath}(i, j, \mathbf{M}, T_{\mathbf{M}})$	$\xrightarrow{\mathbf{M}_{ij}, L_{\mathbf{M}}}$	$r_{\mathbf{M}} \stackrel{?}{=} \mathbf{MTRootFromPath}(i, j, \mathbf{M}_{ij}, L_{\mathbf{M}})$
`Write`			$r_{\mathbf{M}} \leftarrow \mathbf{MTRootFromPath}(i, j, \mathbf{M}'_{ij}, L_{\mathbf{M}})$
$\mathbf{M}'_{ij}$	$(\mathbf{W}_{1..t,j}, L_{\mathbf{W}}) \leftarrow \mathbf{MTElementAndPath}(1..t, j, \mathbf{W}, T_{\mathbf{W}})$	$\xrightarrow{\mathbf{W}_{1..t,j}, L_{\mathbf{W}}}$	$r_{\mathbf{W}} \stackrel{?}{=} \mathbf{MTRootFromPath}(1..t, j, \mathbf{W}_{1..t,j}, L_{\mathbf{W}})$
			$\mathbf{V}_{1..t,j} \leftarrow D_K(\mathbf{W}_{1..t,j})$
			$\mathbf{V}_{1..t,j} \leftarrow \mathbf{V}_{1..t,j} + (\mathbf{M}'_{ij} - \mathbf{M}_{ij})\mathbf{U}_{1..t,i}$
		$\xleftarrow{\mathbf{M}'_{ij}, \mathbf{W}'_{1..t,j}}$	
	$\mathbf{M}_{ij} \leftarrow \mathbf{M}'_{i,j}$		$\mathbf{W}'_{1..t,j} \leftarrow E_K(\mathbf{V}_{1..t,j})$
	$\mathbf{W}_{1..t,j} \leftarrow \mathbf{W}'_{1..t,j}$		$r_{\mathbf{W}} \leftarrow \mathbf{MTRootFromPath}(1..t, j, \mathbf{W}'_{1..t,j}, L_{\mathbf{W}})$
`Audit`	Form $x \leftarrow [r, r^2, \ldots, r^n]^\intercal$	$\xleftarrow{r}$	$r \xleftarrow{\$} S \subseteq \mathbb{F}_q$
	$y \leftarrow \mathbf{M}x$	$\xrightarrow{y, \mathbf{W}}$	$r_{\mathbf{W}} \stackrel{?}{=} \mathbf{MTRootFromLeaves}(\mathbf{W})$
			$\mathbf{V} \leftarrow D_K(\mathbf{W})$
			Form $x \leftarrow [r, r^2, \ldots, r^n]^\intercal$
			$\mathbf{U}y \stackrel{?}{=} \mathbf{V}x$

The requirements are thus that:

$$\forall i, r, X, \mathbf{MTRootFromLeaves}(X) = \mathbf{MTRootFromPath}\,(i, r, \mathbf{MTElementAndPath}(i, r, X, \mathbf{MTCreateTree}(X))\,) \quad (15)$$

As mentioned in Section 4.3, we need to consider two formats which contain the same N bits of data:

- A row-major matrix $M \in \mathbb{F}_q^{m \times n}$ where $N = m \times n \times \lfloor log_2 q \rfloor$. In this format, and for $1 \leq i \leq m$ and $1 \leq j \leq n$, $M_{ij} \in M$ is named a slot.
- The outsourced data can also be represented as a single continuous file F of $\lceil N/b \rceil$ equal sized blocks: $B_1, B_2, \ldots, B_{\lceil N/b \rceil}$, of size b. This blocking is independent of that used for M.

Then, let H be a hash function, $\{0,1\}^* \to \{0,1\}^{2\lambda}$, for a security parameter $\lambda \geq 128$, that is a hash function on more than 256 bits.

For instance, for **MTElementAndPath**, the client wants to read the slot M_{ij}. This corresponds to the block position $k = \left\lceil \frac{(i-1)n+j}{b} \right\rceil$. She more precisely receives from the server M_{ij}, the block B_k *containing* M_{ij} and the set of hash tree uncles L_k corresponding to B_k. She can then check the root of the hash tree with B_k and L_k.

Note that in practice, the algorithms for handling Merkle tree operations might need slightly more inputs (taken implicitly from the respective states of the Client and the Server) than those mentioned in Equation (15). In the following, whenever needed, these extra inputs will be added to the specification.

To be able to run this algorithm, the Server must therefore handle the Merkle hash tree associated to a database. This means having access to an algorithm creating the hash tree, and another algorithm to access the nodes. For these two tasks we use classical implementations:

- More precisely, $T \leftarrow \mathbf{MTCreateTree}(X)$ computes all the nodes of the Merkle hash tree of the whole database X viewed as an array of blocks of size b. A possibility is then to use [26, Algorithm 1].
- $h \leftarrow \mathbf{MTNode}(level, index, T)$ provides access to the node numbered *index* at the required level of the tree. If all the nodes are stored by the Server this is just a labelling of all these nodes. Another possibility for the server is to have a time/memory trade-off as in [26, Algorithm 3]. The idea is to store only the root of subtrees and to recompute hashes within subtrees.

For this, the server arranges hashes of the blocks as leafs in a binary hash tree of depth δ satisfying:

$$\delta = \left\lceil \log_2 \left(\frac{N}{b} \right) \right\rceil . \tag{16}$$

The nodes above the leafs are hashes of their two children.

The size of this tree T is $\sum_{i=1}^{d} 2\lambda 2^i = 2\lambda(2^{\delta+1}-1) < 8\lambda b = 8\lambda \frac{N}{b}$. This size is negligible if $1024 \leq 8\lambda \ll b$. For instance the following choice for b gives an always negligible size: $b = \mathcal{O}(\lambda \log(N))$.

Assuming the hash function is linear to compute, the cost of producing the tree is $O(N)$. This additional algorithm also immediatly gives a possible implementation for the computation of the root: build the tree and output its root as in Algorithm 6.

Algorithm 6 $r_X \leftarrow \mathbf{MTRootFromLeaves}(X)$

Input: a data base X - **System parameter:** b -
Output: the root of the Merkle Hash tree r_X

1: $T \leftarrow \mathbf{MTCreateTree}(X)$;
2: **return** root of T.

Now, to fetch a slot and compute the path of uncles, one just needs to have access to the tree nodes, as illustrated in Algorithm 7. The case of a range of contiguous slots is similar: consider just the uncles of the sub-tree linking all these contiguous blocks.

Algorithm 7 $(X_{ij}, L) \leftarrow \mathbf{MTElementAndPath}(i, j, X, T)$

Input: $i \in [1..m]$, $j \in [1..n]$, X the database, T the Merkle hash tree - **System parameter:** n, m, b -
Output: X_{ij} and L the list constituted by the block containing X_{ij} and by the corresponding list of Merkle tree hashes.

1: $k \leftarrow \left\lceil \frac{(i-1)n+j}{b} \right\rceil$; $B \leftarrow k$-th block of X; $L \leftarrow \{B\}$;
2: **for** $i = 1..depth(T)$ **do**
3: **if** k is odd **then**
4: $L \leftarrow L \cup \{\mathbf{MTNode}(i, k-1, T)\}$;
5: **else**
6: $L \leftarrow L \cup \{\mathbf{MTNode}(i, k+1, T)\}$;
7: **end if**
8: $k \leftarrow \lfloor k/2 \rfloor$;
9: **end for**
10: **return** $(X_{i,j}, L)$

Finally, to check the correctness of a slot X_{ij}, we need the block B_k and the list of δ uncles and to recompute the root, only from this list and the block. Therefore, a possible implementation of this recomputation is given in Algorithm 8, first with a single block. Here also the case of a range of contiguous slots is similar. The idea is to consider L as the union of the list of uncles and the block itself as its first element. Then the new slot X_{ij} to be considered replaces (for instance after a write operation) the old slot within the block B_k and the root is computed from this new block and the path of hashes.

Algorithm 8 $r \leftarrow$ **MTRootFromPath**$(i, j, X_{i,j}, L)$

Input: $i \in [1..m]$, $j \in [1..n]$, a slot X_{ij}, a list L constituted by the block B_k and the corresponding list of hashes - **System parameter:** n, m, b -

Output: r the root of the Merkle Hash tree.

$B \leftarrow L[1]$; $\ell \leftarrow (i-1)n + j \mod b$;
$B_\ell \leftarrow X_{i,j}$; {Update with the new value}
$r \leftarrow H(B)$; $k \leftarrow \left\lceil \frac{(i-1)n+j}{b} \right\rceil$.
for $i = 2..length(L)$ **do**
  if the $(i-1)$th bit of k is 1 **then**
    $r \leftarrow H(L[i], r)$;
  else
    $r \leftarrow H(r, L[i])$
  end if
end for
return r

The cost to recompute the root is that of hashing one block, and then of computing δ additional hashes of two hashes, that is $O(b + \delta\lambda)$. The difficulty for an attacker to pass this integrity test is that of the second preimage of the hash function, see [10], e.g., for more details.

From these, it is then easy to implement the API of Section 4.3: Table 15 propose an overview of the implementation of **MTInit**, **MTVerifiedRead** and **MTVerifiedWrite**.

Table 15: Implementation of **MTInit**, **MTVerifiedRead** and **MTVerifiedWrite**.

- **MTInit**$(1^\lambda, b, X) \mapsto (r_M \mid M, T_M)$

Verifier	$r_M \leftarrow$ **MTRootFromLeaves**(M)
Comm.	$(M)\downarrow$
Server	$T_M \leftarrow$ **MTCreateTree**(M)

- **MTVerifiedRead**$(i, j, r_M \mid M, T_M) \mapsto M_{i,j}$

V.	$r_M \stackrel{?}{=}$ **MTRootFromPath**$(M_{i,j}, i, j, L_M)$	
C.	$(i, j)\downarrow$	$(M_{i,j}, L_M)\uparrow$
S.	**MTElementAndPath**$(i, j, M, T_M) \to (M_{i,j}, L_M)$	

- **MTVerifiedWrite**$(i, j, M'_{i,j}, r_M \mid M, T_M)$

After **MTVerifiedRead**$(i, j, r_M \mid M, T_M)$:	
V.	$r_M \leftarrow$ **MTRootFromPath**$(M'_{i,j}, i, j, L_M)$
C.	$\left(M'_{i,j}\right)\downarrow$
S.	updates M, T_M