Dynamic proofs of retrievability with low server storage

Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, and Clément Pernet, *Université Grenoble Alpes;* Michael Hanling and Daniel S. Roche, *United States Naval Academy*

https://www.usenix.org/conference/usenixsecurity21/presentation/anthoine

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

Dynamic proofs of retrievability with low server storage

Gaspard Anthoine, Jean-Guillaume Dumas, Mélanie de Jonghe, Aude Maignan, Clément Pernet
Université Grenoble Alpes, UMR CNRS 5224, LJK, 38000 Grenoble, France
{Firstname.Lastname}@univ-grenoble-alpes.fr, deJonghe.Melanie63@gmail.com

Michael Hanling, Daniel S. Roche
United States Naval Academy, Annapolis, Maryland, U.S.A.
MikeHanling@gmail.com, Roche@usna.edu

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 less than 5 minutes and costs less than $0.08 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

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 [9]. 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 [31].

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, 21], 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, Sia, SAFE Network, 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.

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.

Provable Data Possession (PDP). PDP audits [16, 23, 35] are 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

*https://filecoin.io, https://storj.io, https://sia.tech, https://safenetwork.tech, https://www.pp.io.

stored [13, 26]. Early PoR schemes did not provide an efficient update mechanism to alter individual data blocks, but more recent *dynamic* schemes have overcome this shortcoming [10, 34].

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 [32]	X	∞	X	$o(N)$
Ateniese et al. [5]	X	∞	X	$o(N)$
Ateniese et al. [6]	X	$O(1)$	$O(1)$	$o(N)$
Storj [36]	✓	$O(1)$	∞	$o(N)$
Juels et al. [23]	✓	$O(1)$	X	$O(N)$
Lavauzelle et al. [26]	✓	∞	X	$O(N)$
Stefanov et al. [35]	✓	∞	∞	$O(N)$
Cash et al. [10]	✓	∞	∞	$O(N)$
Shi et al. [34]	✓	∞	∞	$O(N)$
Here	✓	∞	∞	$o(N)$

Section 7 gives a detailed comparison 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, 14], and require verifiable storage of a large amount of essentially-random data. 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 recompute 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, 11, 19, 37]. The Filecoin network employs this type of verification. A different and promising approach, not based on timing assumptions, has recently been proposed by [12]. 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 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, [12] 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.

1.1 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 that 1TB of data can be audited in less than 5 minutes at a monetary cost of just 0.08 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 groups, 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.

1.2 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 5; the latter thus presents the formal setting, and also contains a publicly verifiable variant; Section 6 discusses the results of our open-source implementation and deployment on Google Compute Platform.

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 [34], except that we follow [23] 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 [17]. 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 server computation in all these algorithms is deterministic while 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.

- $(st_{\mathcal{C}}, st_{\mathcal{S}}) \leftarrow \mathbf{Init}(1^{\kappa}, 1^{\lambda}, b, M)$: 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 $st_{\mathcal{S}}$.
- $\{m_i, \mathbf{reject}\} \leftarrow \mathbf{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}})$: 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**.
- $\{(st'_{\mathcal{C}}, st'_{\mathcal{S}}), \mathbf{reject}\} \leftarrow \mathbf{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}})$: 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**.
- $\{\pi, \mathbf{reject}\} \leftarrow \mathbf{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}})$: On input of the client state $st_{\mathcal{C}}$ and the server state $st_{\mathcal{S}}$, outputs a successful transcript π or **reject**.
- $M \leftarrow \mathbf{Extract}(st_{\mathcal{C}}, \pi_1, \pi_2, \ldots, \pi_e)$: 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 an honest client and an honest 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,* $\mathbf{IsValid}(M, \mathbf{Init}(1^{\kappa}, 1^{\lambda}, b, M))$. *Furthermore, for any state such that* $\mathbf{IsValid}(M, (st_{\mathcal{C}}, st_{\mathcal{S}}))$ *and any index i with $0 \leq i < \lceil N/b \rceil$, we have*

- $\mathbf{Read}(i, st_{\mathcal{C}}, st_{\mathcal{S}}) = M[i]$;
- $\mathbf{IsValid}(M', \mathbf{Write}(i, a, st_{\mathcal{C}}, st_{\mathcal{S}}))$, *where $M'[i] = a$ and the remaining $M'[j] = M[j]$ for every $j \neq i$;*
- $\mathbf{Audit}(st_{\mathcal{C}}, st_{\mathcal{S}}) \neq \mathbf{reject}$;
- *For e audits* $\mathbf{Audit}_1, \ldots, \mathbf{Audit}_e$ *with independent randomness, with probability* $1 - \mathsf{negl}(\lambda)$*:*

$$\mathbf{Extract}(st_{\mathcal{C}}, \mathbf{Audit}_1(st_{\mathcal{C}}, st_{\mathcal{S}}), \ldots, \mathbf{Audit}_e(st_{\mathcal{C}}, st_{\mathcal{S}})) = M.$$

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. To distinguish between public and private verification, we consider now two types of client: a *Writer* who can run any of the **Init**, **Write**, **Read**, or **Audit** algorithms; and a *Verifier* that can only run the last two. Accordingly, we split the client state in two parts: the secret values, prv_C, and the published ones, pub_C, so that $st_C = pub_C \cup prv_C$. In a privately verifiable protocol, detecting such deviations requires both secret and public values. In a publicly verifiable protocol we distinguish the **Init** and **Write** algorithms (which use the full client state st_C) from the **Read** and **Audit** ones (which use only the public part pub_C). Detecting a deviation in **Init** and **Write** still require the full client state (both the secret and public parts), while detecting a deviation in **Read** and **Audit** must be possible using only the public parts of the client state. We use the following game between two observers O_1 and O_2, a potentially *malicious* server $\overline{\mathcal{S}}$ and an honest server $\mathcal{S}$ for the adaptive version of authenticity. This is the game of [34], generalized to exhibit a public/private distinction:

1. $\overline{\mathcal{S}}$ chooses an initial memory M. O_1 runs **Init**, sends pub_C to O_2 and sends the initial memory layout $st_{\mathcal{S}}$ to both $\overline{\mathcal{S}}$ and $\mathcal{S}$.
2. For a polynomial number of steps $t = 1, 2, ..., poly(\lambda)$, $\overline{\mathcal{S}}$ picks an operation op_t where operation op_t is either **Read**, **Write** or **Audit**. O_1 and O_2 execute their respective operations with both $\overline{\mathcal{S}}$ and $\mathcal{S}$.
3. $\overline{\mathcal{S}}$ is said to win the game, if any message sent by $\overline{\mathcal{S}}$ differs from that of $\mathcal{S}$ and neither O_1, nor O_2, did output **reject**.

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

Definition 3 (Authenticity)**.** *A PoR scheme satisfies* private adaptive authenticity *(or just* adaptive authenticity*), if no polynomial-time adversary $\overline{\mathcal{S}}$ has more than negligible probability in winning the above security game when O_1 also plays the role of O_2.*

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, [10] uses a *blackbox 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 [34], 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 O_1 running the honest client algorithms may only update its state st_C during **Write** operations, and that the **Audit** operations are independently randomized from the client side, but we make no assumptions about the state of the adversary $\overline{\mathcal{S}}$.

1. $\overline{\mathcal{S}}$ chooses an initial database M. The observer runs **Init** and sends the initial memory layout $st_{\mathcal{S}}$ to $\overline{\mathcal{S}}$;
2. For $t = 1, 2, ..., poly(\lambda)$, the adversary $\overline{\mathcal{S}}$ adaptively chooses an operation op_t where op_t is either **Read**, **Write** or **Audit**. The observer executes the respective algorithms with $\overline{\mathcal{S}}$, updating st_C and M according to the **Write** operations specified;
3. The observer runs e **Audit** algorithms with $\overline{\mathcal{S}}$ 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 $\overline{\mathcal{S}}$ is said to with the game if $e' \geq e/2$ and $\textbf{Extract}(st_C, \pi_1, \ldots, \pi_e) \neq M$.

Definition 4 (Retrievability)**.** *A PoR scheme satisfies retrievability if no polynomial-time adversary $\overline{\mathcal{S}}$ has more than negligible probability in winning the above security game.*

3 Time-space tradeoff lower bound

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., [10, 23, 34]) and some schemes with a low storage overhead but high computational cost for the server during audits (e.g., [5, 32, 33]).

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

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

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, 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 5. *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_S, the client stores a c-bit string st_C, and the *client virtual memory* in the language of [10] is the unique N-bit string M such that $\mathbf{IsValid}(st_C, st_S, M)$.

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

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_C, st_S) such that $\phi(st_C, st_S) = M$. 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 $\overline{\mathcal{S}}$ for the game of Definition 4 as follows: On the first step, $\overline{\mathcal{S}}$ chooses M_0 as the initial database, and uses this in the **Init** algorithm to receive server state st_S. Next, $\overline{\mathcal{S}}$ chooses k indices uniformly at random from the st_S of $(N+s)$ bits (where k is a parameter to be defined next), and flips those k bits in st_S to obtain a *corrupted* state st'_S. Finally, $\overline{\mathcal{S}}$ runs the honest **Audit** algorithm $2e$ times on step 3 of the security game, using this corrupted state st'_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

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

We assumed that $s \geq 0$ and $t \leq \frac{N}{4}$, thus we have that $k \geq 1$.

Let st_C be the initial client state (which is unknown to $\overline{\mathcal{S}}$) in the attack above with initial database M_0. From the correctness requirement (Definition 1) and the definition of t in our theorem, running $\mathbf{Audit}(st_C, st_S)$ must always succeed after examining at most t bits of st_S. Therefore, if the k flipped bits in the corrupted server storage st'_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 $\mathbf{Audit}(st_C, st'_S)$ 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 $\overline{\mathcal{S}}$ successfully passes at least e audits (as required) with probability at least $\frac{1}{2}$. We want to examine the probability that $\phi(st_C, st'_S) \neq M_0$, 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'_S, and from the upper bound of (4) above, the probability that $\phi(st_C, st'_S) \neq M_0$ 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 $\mathbf{Audit}(st_C, st'_S)$ 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 5, 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 arbitrarily decrease communication bandwidth during **`Audit`**s by increasing client persistent storage and computation time.

4.1 Overview

A summary of our four algorithms is shown in Figure 1, 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 [18]. 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 [20] 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 with a standard Merkle hash tree.

4.2 Matrix based approach for audits

The basic premise of our particular PoR is to treat the data, consisting of N bits , as a matrix $\mathbf{M} \in \mathbb{F}_q^{m \times n}$, where $\mathbb{F}_q$ is a suitable finite field of size q, and each chunk of $\lfloor \log_2 q \rfloor$ bits is considered as an element of $\mathbb{F}_q$. Crucially, the choice of field $\mathbb{F}_q$ 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 [20].

In the **`Init`** algorithm, the client chooses a secret random *control vector* $\mathbf{u} \in \mathbb{F}_q^m$ and computes a second secret control vector $\mathbf{v} \in \mathbb{F}_q^n$ 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* $\mathbf{x} \in \mathbb{F}_q^n$, and asks the server to compute a *response vector* $\mathbf{y} \in \mathbb{F}_q^m$ 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

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

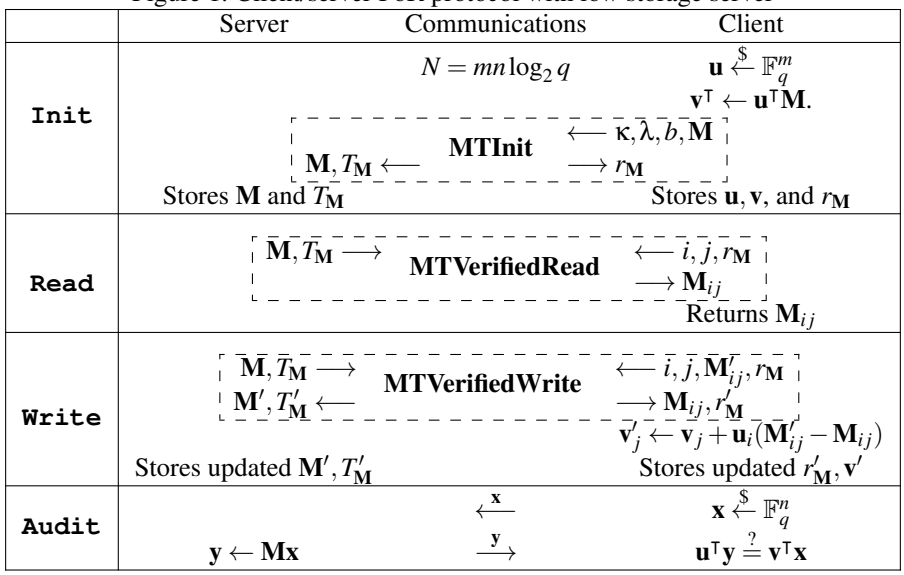
The proof of retrievability will rely on the fact that observing several successful audits allows, with high probability, recovery of the matrix $\mathbf{M}$, and therefore of the entire database.

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 almost no additional persistent storage on the server side, 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 5. 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 4) 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 $\mathbf{M}\mathbf{X} = \mathbf{Y}$, 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 [32] 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.

Figure 1: Client/server PoR protocol with low storage server



4.3 Merkle hash tree for updates

In our protocols, the raw database of size N bits is handled in two different ways. As seen in the previous section, the audits use chunks of $\lfloor \log_2 q \rfloor$ bits as elements of a finite field $\mathbb{F}_q$. Second, a Merkle hash tree with a different block size b is used here to ensure authenticity of individual **`Read`** operations. This tree 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 [25, 27]. To that end, we provide three algorithms to abstract the details of the Merkle hash tree: **MTInit**, **MTVerifiedRead** and **MTVerifiedWrite**.

These three algorithms are in fact 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 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.

5.1 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 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 and to support a potential evolution of the security requirements.

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, for $\mathbf{M} \in \mathbb{F}_q^{m \times n}$.

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 $\mathbf{U}$, the audit fails only if $(\mathbf{z} - \mathbf{y}) \neq \mathbf{0}$ but $\mathbf{U}$ is in its left nullspace. 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 [24]: 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 structure constraint on $\mathbf{U}$ 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 $E_K : \mathbb{F}_q \to \mathcal{C}$ (from the field to any ciphertext space $\mathcal{C}$) and a decryption function $D_K : \mathcal{C} \to \mathbb{F}_q$, 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 $\mathbf{V}$.

5.2 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 2.

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_S, st_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: $\mathbf{W} \leftarrow E_K(\mathbf{V}) \in \mathcal{C}^{t \times n}$; {elementwise}
  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: $st_S \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy, \mathbf{W}, T_\mathbf{W})$;
  Client: $st_C \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, K, r_\mathbf{W})$;
else
  Client: sends $m, n, q, \mathbf{M}$ to the Server;
  Server: $st_S \leftarrow (m, n, q, \mathbf{M}, T_\mathbf{M}, Strategy)$;
  Client: $st_C \leftarrow (m, n, q, t, \mathbf{s}, r_\mathbf{M}, Strategy, \mathbf{V})$;
end if

Algorithm 2 **`Read`**(st_S, st_C, i, j)

Input: $st_S, st_C, i \in [1..m], j \in [1..n]$
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** $\mathbf{M}_{ij}$

Algorithm 3 **`Write`**$(st_S, st_C, i, j, \mathbf{M}'_{ij}, Strategy)$

Input: $st_S, st_C, i \in [1..m], j \in [1..n], \mathbf{M}'_{ij} \in \mathbb{F}_q$
Output: st'_S, st'_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 $\mathbf{U}_{1..t,i} \leftarrow [\mathbf{s}_k^i]_{k=1\ldots t} \in \mathbb{F}_q^t$
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: $\mathbf{W}'_{1..t,j} \leftarrow E_K(\mathbf{V}'_{1..t,j}) \in \mathcal{C}^t$
  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'_S using $\mathbf{M}', T'_\mathbf{M}, \mathbf{W}'$, and $T'_\mathbf{W}$
  Client: Update st'_C using $r'_\mathbf{M}$ and $r'_\mathbf{W}$
else
  Server: Update st'_S using $\mathbf{M}'$ and $T'_\mathbf{M}$
  Client: Update st'_C using $r'_\mathbf{M}$ and $\mathbf{V}'$
end if

5.3 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 6. *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.

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

Table 2: 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	$O(b + \kappa \log N)$	$O(b + \kappa \log N)$		$O(b + \kappa \log N)$	

Algorithm 4 `Audit`$(st_S, st_C, Strategy)$

Input: st_S, st_C
Output: `accept` or `reject`

1: Client: $\rho \xleftarrow{\$} \mathbb{F}_q$ and sends it to the Server;
2: Let $\mathbf{x}^\intercal \leftarrow [\rho^1, \rho^2, \ldots, \rho^n]$
3: Server: $\mathbf{y} \leftarrow \mathbf{M}\mathbf{x} \in \mathbb{F}_q^m$; {$\mathbf{M}$ from st_S}
4: Server: sends $\mathbf{y}$ to Client;
5: **if** $(Strategy = externalization)$ **then**
6: Both: $\mathbf{W} \leftarrow$ **MTVerifiedRead**$(\mathbf{W}, T_{\mathbf{W}} \mid (1..t, 1..n), r_{\mathbf{W}})$;
7: Client: $\mathbf{V} \leftarrow D_K(\mathbf{W}) \in \mathbb{F}_q^{t \times n}$
8: **end if**
9: Client: Let $\mathbf{U} \leftarrow [\mathbf{s}_i^j]_{i=1\ldots t, j=1\ldots m} \in \mathbb{F}_q^{t \times m}$
10: **if** $(\mathbf{U}\mathbf{y} = \mathbf{V}\mathbf{x})$ **then**
11: Client: **return** `accept`
12: **else**
13: Client: **return** `reject`
14: **end if**

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 [30, Theorem 4.1], or [22, 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.

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

Input: st_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`

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

Theorem 7. *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+24λ*

satisfies correctness, adaptive authenticity and retrievability as defined in Definitions 1, 3 and 4.

Proof. Correctness comes from the correctness of the Merkle hash tree algorithms, and from the fact that, when all parties are honest, $\mathbf{U}\mathbf{y} = \mathbf{U}\mathbf{M}\mathbf{x} = \mathbf{V}\mathbf{x}$.

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 $\mathbf{z} \neq \mathbf{M}\mathbf{x}$ which the Client fails to reject, and 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 3).

For retrievability, we need to prove that Algorithm 5 succeeds with high probability on the last step of the security game from Definition 4. 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 6, 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. □

5.4 Publicly verifiable variant

Our scheme can also be adapted to meet the stricter requirement of *public verifiability* (see Section 2.2), wherein there are now two types of client: a *Writer* who can run any of the **`Init`**, **`Write`**, **`Read`**, or **`Audit`** algorithms; and a *Verifier* that can only run the last two.

The idea is that U and V will be published as g^U and g^V as to hide their values, while still enabling the dot product verification.

More precisely, we will employ the externalized storage strategy outlined in Section 5.1, so that the server holds all the information needed to perform audits. But this alone is not enough, as the public Verifier (and thus, the possibly-malicious server) must not learn the plaintext control vector values.

The challenge, then, is to support equality testing for dot products of encrypted values, without decrypting. Any linearly homomorphic encryption could be used, but this is actually more than what we need since decryption is not even necessary. Instead, we will simply employ a group where the discrete logarithm is hard, instead of the (relatively small) finite fields used before.

Further, we use a group of prime order, in order to be able to easily compute with exponents. In particular, thanks to the homomorphic property of exponentiation, we will perform some linear algebra over the group and need some notations for this. For a matrix $\mathbf{A}$, $g^{\mathbf{A}}$ denotes the coefficient-wise exponentiation of a generator g to each entry in $\mathbf{A}$. Similarly, for a matrix $\mathbf{W}$ of group elements and a matrix $\mathbf{B}$ of scalars, $\mathbf{W}^{\mathbf{B}}$ denotes the extension of matrix multiplication using the group action. If we have $\mathbf{W}=g^{\mathbf{A}}$, then $\mathbf{W}^{\mathbf{B}}=(g^{\mathbf{A}})^{\mathbf{B}}$. Futher, this quantity can actually be computed by working in the exponents first, i.e., it is equal to $g^{(\mathbf{AB})}$. For example:

$$\left(g^{\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]}\right)^{\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]}=\begin{bmatrix}g^a & g^b\\ g^c & g^d\end{bmatrix}^{\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]}=\begin{bmatrix}g^{ae+bf}\\ g^{ce+df}\end{bmatrix}=g^{\left(\left[\begin{smallmatrix}a & b\\ c & d\end{smallmatrix}\right]\left[\begin{smallmatrix}e\\ f\end{smallmatrix}\right]\right)}.$$

The resulting modified protocol is presented formally in Figure 2. In summary, the changes are as follows:

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 mapped to $\mathbb{G}$: $\mathbf{W}\leftarrow E(\mathbf{V})=g^{\mathbf{V}}$; second, the Writer also publishes an encryption of $\mathbf{U}$ as: $\mathbf{K}\leftarrow g^{\mathbf{U}}$ over an *authenticated* channel; $\mathbf{K}$ is called the *public key*.
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 8. *Note that the costly server-side computation during audits does not involve any group operations; only the clients must perform exponentiations. However, the field size p must be increased in order for the discrete logarithm to be hard. For a database of fixed bit-length, this increase in field size induces a cost overhead in the field arithmetic (up to quadratic in* $\log p$*) which is partly compensated by a corresponding decrease in the matrix dimension (linear in* $\log p$*), as* $N=mn\log p$.

Under Linearly Independent Polynomial (LIP) Security [1, Theorem 1][†], the Protocol of Figure 2 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

[†]LIP security reduces to the MDDH hypothesis, a generalization of the widely used decision linear assumption [1, 29]

Figure 2: Publicly verifiable Client/server PoR protocol with low storage server

	Server	Communications	Client
Init		$N = mn\log_2 q$ $\mathbb{G}$ of order p and gen. g	$s \xleftarrow{\$} S \subseteq \mathbb{Z}_p$ form $\mathbf{u} \leftarrow [\mathbf{s}^j]_{j=1..m} \in \mathbb{Z}_p^m$ $\mathbf{v}^\intercal \leftarrow \mathbf{u}^\intercal\mathbf{M}$, $\mathbf{w}^\intercal \leftarrow g^{\mathbf{v}} \in \mathbb{G}^n$.
		$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \leftarrow$ **MTInit** $\leftarrow \kappa, \lambda, b, \mathbf{M}, \mathbf{w}$ $\rightarrow r_\mathbf{M}, r_\mathbf{w}$	
	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	$\mathbf{M}, T_\mathbf{M} \rightarrow$ **MTVerifiedRead** $\leftarrow i, j, r_\mathbf{M}$ $\rightarrow \mathbf{M}_{ij}$		Return $\mathbf{M}_{ij}$
Write	$\mathbf{M}, T_\mathbf{M}, \mathbf{w}, T_\mathbf{w} \rightarrow$ **MTVerifiedRead** $\leftarrow i, j, r_\mathbf{M}, r_\mathbf{w}$ $\rightarrow \mathbf{M}_{ij}, \mathbf{w}_j$		$\delta \leftarrow \mathbf{u}_i(\mathbf{M}'_{ij} - \mathbf{M}_{ij})$
		$\xleftarrow{i, j, \mathbf{M}'_{ij}, \mathbf{w}'_j}$	$\mathbf{w}'_j \leftarrow \mathbf{w}_j \cdot g^\delta$
	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 $\mathbf{x} \leftarrow [r^i]_{i=1..n} \in \mathbb{Z}_p^n$	
	$\mathbf{w}, T_\mathbf{w} \rightarrow$ **MTVerifiedRead** $\leftarrow r_\mathbf{w}$ $\rightarrow \mathbf{w}$		$\mathbf{K}^y \stackrel{?}{=} \mathbf{w}^x$

$g^{\mathbf{U}_i} = \left(g^{s_i^j}\right)_{j=1..m}$ 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 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 9. Compared to Theorem 7, this requires the LIP security assumptions and a larger domain of the elements.

Theorem 9. *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 Figure 2 satisfies correctness, public authenticity and retrievability, as defined in Definitions 1, 2 and 4.*

Proof. In Figure 2, 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 [15, Lemma 1] and simulate **Init** via the following inputs to the attacker:

- $r \xleftarrow{\$} S \subseteq \mathbb{Z}_p^*$ and let $x = [r, r^2, \ldots, r^n]^\intercal$;
- 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 $\geq \varepsilon/m$.

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 $\geq t\varepsilon/m \geq \varepsilon/m$. 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 7. □

Remarks 10. *We mention a few small performance and implementation notes:*

- *If a Writer wants to perform an audit, she does not need to use the encrypted control vector* $\mathbf{K}$*, nor to store it. She just computes* $\mathbf{U}y$ *directly, then checks 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* $m = n = O(\sqrt{N/\log p})$.
- *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 most significant changes between the private and public modes are for the Verifier's and (to a much lesser extent) server's computation time during **`Audit`**s: we show in Section 6 that public verification is more expensive but this remains doable in a few seconds even on a constrained device.

6 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 tradeoff with increase costs for computation during audits?

Our experimental results are summarized in Tables 5 to 7. For a 1TB data file, the $O(\sqrt{N})$ communication cost of our audit entails less than 6MB of data transfer, and our implementation executes the $O(N)$ audit for this 1TB data file in less than 5 minutes for a monetary cost of about 8 cents 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. These results indicate that the 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 intact as a normal file on the Server's filesystem.

The remainder of this section gives the full details of our implementation and experimental setup.

6.1 Parameter selection

Our algorithm treats the database as if it is an $n \times m$ matrix with elements in a finite field. As seen in Section 5, we need the field size to be at least 40 bits or more in order to ensure authenticity.

We ran the experiments with two modes: a private one with a 57-bits prime and a public one with a 253-bits prime.

In order to maximize the block size while avoiding costly multiple-precision arithmetic, we used the largest 57-bit prime, $p = 144115188075855859$ for the private mode. This allows the input to be read in 7-byte (56-bit) chunks with no conversion necessary; each 56-bit chunk of raw data is treated as an element of $\mathbb{F}_p$. At the same time, because p is less than 64-bit, no computations require multiple-precision, and multiple multiplications can be accumulated in 128-bit registers before needing to reduce modulo p. Finally, choosing a prime close to (but less than) a power of 2 makes randomly sampling integers modulo p especially efficient (discarding sampled values larger than p will seldom happen).

To balance the bandwidth (protocol communications) and the client computation costs, we represent $\mathbf{M}$ as a square matrix with dimensions $m = n = \sqrt{N/56}$, where the 56 comes from our choice of $\mathbb{F}_p$. We also fixed the Merkle tree block size at 8KiB for all experiments and used SHA-512/224 for the Merkle tree hash algorithm.

For the public mode, we used the following libraries[‡]: `gmp-6.2.1` and `givaro-4.1.1` for arbitrary precision prime fields, `openblas-0.3.15` and `fflas-ffpack-2.4.3` for high-performance linear algebra, and `libsodium-1.0.18` for the elliptic curve. We ran the experiments with *ristretto255*, a 253-bits prime order subgroup of *Curve25519*. We still use a square matrix database, but now with $m = n = \sqrt{N/252}$. Depending on the database size, Theorem 9 shows that the computational security parameter of our next experiments is set to slightly less than 128 (namely between 117.78 and

[‡] https://gmplib.org, https://github.com/linbox-team/givaro, http://www.openblas.net, https://linbox-team.github.io/fflas-ffpack, https://download.libsodium.org.

120.2). The resulting asymptotic costs for these parameter choices, in both modes, 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	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

6.2 Experimental Design

Our implementation provides the **Init**, **Read**, **Write**, and **Audit** algorithms as described in the previous sections, 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[§].

Table 4: Google Cloud Server VMs
Costs as of May 2021. Each physical core is hyperthreaded as two vCPUs.

	Family	Physical Cores	Main Memory	Local SSDs	Cost/hour (USD)
C	f1-micro	0.1	0.6 GB	—	$0.01
S_1	n1-standard-2	1	7.5 GB	1.5TB	$0.26
S_4	n1-standard-8	4	30 GB	6TB	$1.04
S_{16}	n1-standard-32	16	120 GB	9TB	$2.51

The client machine C was a cheap f1-micro instance with a shared vCPU and low RAM, in the europe-west1 region. For the server, we used 3 different VMs S_1, S_4, S_{16} as listed in Table 4, all in the us-central1 region. The database file itself was stored on local SSD drives for maximal throughput in our audit experiments. Although our implementation does not use more disk space than the size of the database plus the size of the Merkle tree (never more than 1.007TB in our experiments), we over-provisioned the SSDs to achieve higher throughputs; the prices in Table 4 reflect the total VM instance and storage costs.

For testing, we generated files of size 1GB, 10GB, 100GB, and 1TB filled with random bytes. All times reported are total "wall-clock" time unless otherwise noted. Except where noted with an asterisk (*), where experiments were run only once, all values are the median over 11 runs, ignoring the first run in order to "warm up" caches etc. Note that this actually had a significant effect on the larger machine size which also has more RAM, as the 16-core machine can cache all sizes except the 1TB database in memory.

[§] https://cloud.google.com/compute/docs/machine-types.

6.3 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 the standard Linux checksum tools `md5sum` and `sha256sum`.

Table 5: Single-threaded experiments on Google Cloud
Values indicate the median number of seconds for a single run on the S_1 machine. Except where noted with (*), each experiment was performed 11 times. In all cases, after discarding at most one outlier value, the maximum relative difference between the runs was at most 20%.

Operation	1GB	10GB	100GB	1TB
MD5	1.87	20.58	202.51	2017.76*
SHA256	5.21	54.52	561.22	5413.35*
Init	2.46	29.42	284.75*	2772.14*
PRIVATE-VERIFIED AUDIT USING 57-BIT PRIME				
Client	0.00	0.00	0.00	0.01
Server	0.24	4.93	53.05	529.90
PUBLIC-VERIFIED AUDIT USING RISTRETTO255				
Client	0.53	1.67	5.37	16.81
Server	1.65	17.1	173.49	1725.75*

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.

Table 5 confirms that the cost of our **Audit** procedure scales linearly with the database size, as expected. Furthermore, we can see that audits are very efficient in practice, being even faster than the built-in checksum utilities in our tests. That is, the $O(N)$ running time of our **Audit** algorithm is actually feasible, for both the private and public mode, even at the terabyte scale. The public mode is slightly slower, as expected in Remark 8.

6.4 Parallel server speedup for audits

Our experimental results in Table 5 indicate good performance for our **Audit** algorithm, but at the larger end of database sizes such as 1TB, the $O(N)$ work performed by the server still incurs a significant delay of several minutes.

To demonstrate that a more powerful server can handle large-size **Audit**s even more efficiently, we used OpenMP to parallelize the main loop of our **Audit** algorithms. These routines are trivially parallelizable: each parallel core 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. The results are reported in Table 6. When the computation is CPU-bound, as is the case mostly with the public verified version that uses larger primes, CPU utilization is high and we achieve linear speedup compared to the single-core timings in Table 5. For the more efficient 57-bit private verification version, the speedup compared to Table 5 is sometimes more and sometimes less than linear, for two reasons that have to do with the I/O bottleneck between disk and CPU.

First, the larger machines S_4 and S_{16} that are used here do not just have more cores than S_1; they also have more RAM and more (over-provisioned) local SSD space. This allows S_4 to entirely cache the 10GB database and S_{16} to entirely cache the 10GB and 100GB databases, leading to sometimes super-linear speedup when the computation is I/O-bound.

The second observation is that, even using the fastest solution available (local SSDs) in Google Cloud, we could not achieve greater than roughly 4GB/sec throughput reading from disk. This effectively creates a "maximum speed" for any computation, which limits the benefit of additional cores especially for the 1TB audit with the small 57-bit prime. To a lesser extent these two phenomena also occur in the public mode. There, they are however partially compensated by a better parallelism pertaining an increase in the computations. However, we emphasize again that this is a *good thing* — our **Audit** algorithm is efficiently parallelizable, up to the inherent limiting speed of fetching data from the underlying storage.

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 6 as well. Interestingly, due to the disk throughput limitations discussed above, the 4-core VM is more cost-effective for private-verified audits.

Table 6: Multi-core server times for Audit
Values indicate the median number of seconds wall-time for a single run. Except where noted with (*), each experiment was performed 11 times. In all cases, after discarding at most one outlier, the maximum relative difference between the runs was at most 20%.

Server	Metric	1GB	10GB	100GB	1TB
PRIVATE-VERIFIED AUDIT USING 57-BIT PRIME					
S_4	Audit	0.06	0.62	29.08	278.37
	Cost	$0.00002	$0.0002	$0.008	$0.08
S_{16}	Audit	0.03	0.22	1.88	250.91
	Cost	$0.00002	$0.0002	$0.001	$0.175
PUBLIC-VERIFIED AUDIT USING RISTRETTO255					
S_4	Audit	0.45	4.37	51.45	536.09*
	Cost	$0.0001	$0.001	$0.015	$0.155
S_{16}	Audit	0.12	1.21	11.87	357.49*
	Cost	$0.0001	$0.001	$0.008	$0.249

6.5 Network communication costs

Having closely examined the server and client computation times, we finally turn to the $O(\sqrt{N})$ communication bandwidth between client and server during audits. Recall that our client C was located in western Europe and the servers S_1, S_4, S_{16} were located in central North America. 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.

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 7 summarizes that communication time of both $\mathbf{x}$ and $\mathbf{y}$ remains under two seconds. We also list the total amount of data communicated, which exhibits square root scaling as expected.

Table 7: Amount of Communication Per **Audit**
Values indicate the median number of seconds for a single run with the S_4 server. Each experiment was performed 11 times, with a maximum variance of 13% between runs.

Metric	1GB	10GB	100GB	1TB
Comm. (KB)	187	591	1868	5906
Time (s)	0.73	1.19	1.50	1.80

7 Detailed state of the art

PDP schemes, first introduced by Ateniese et al. [5], originally only considered static data storage. The original scheme was later adapted to allow dynamic updates by Erway et al. [16] 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, independently introduced by Juels et al. [23], 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.

7.1 Low storage overhead

The schemes of Ateniese et al. [5] or Sebé et al. [32] 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 8. 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 8: S-PDP on f blocks : The file M is composed of N/b blocks of bit-size b.
The computation is made mod Q, a 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 [8, §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 by Fiore and Gennaro in [18], but then it is not dynamic anymore.

Storj [36] (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 9. 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 $O(\log n)$.

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.

Table 9: 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		$N+O(\frac{N}{b}\sigma)$	$O(N\sigma)$
	Avg. Audit	$O(N+\frac{N}{b}\sigma)$	$O(\frac{N}{b}\log\sigma+\frac{N}{\sigma})$	$O(N)$
	Update		$b+O(\sigma)$	$O(b\sigma)$

Lavauzelle et al., [26] 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. [23] use block encryptions and sentinels in order to store static data with a cloud server. Shacham and Waters [33] 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. [35] were the first to consider a dynamic PoR scheme. Later improvements by Cash et al. or Shi et al. [10, 34] allow for dynamic updates and reduce the asymptotic complexity (see Table 10). 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.

Table 10: Shi et al. [34]: 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	$O(b\log N)$	$O(b+\log N)$	$O(b+\log N)$

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, 37]. Some use a weaker, *rational*, attacker model [11, 28], 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 11 compares the additional server storage and audit

Table 11: Comparison of our low server storage protocol with that of Shi et al. [34].

	Shi et al. [34]	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	$O(b)$	$O(1)$	$O(N^{1-\alpha})$

costs between [34] 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.

8 Conclusion

We presented new protocols for dynamic Proof of Retrievability, based on randomized linear algebra verification schemes over a finite field. Our protocols do not require any encoding of the database and are therefore near optimal in terms of persistent storage on the server side. They include also efficient unlimited partial retrievals and updates as well as provable retrievability 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, using for instance succinct non-interactive arguments of knowledge.

Acknowledgments

We thank the reviewers for their thoughtful comments and efforts towards improving our manuscript.

Availability

The source code and script to perform the experiments of Section 6 are available via the following GitHub repository: https://github.com/dsroche/la-por.

References

[1] Michel Abdalla, Fabrice Benhamouda, and Alain Passelègue. An algebraic framework for pseudorandom functions and applications to related-key security. In R. Gennaro and M. Robshaw, editors, *CRYPTO 2015*, pages 388–409, 2015.

[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, USA, August 2016.

[4] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of Space: When Space Is of the Essence. In *SCN*, pages 538–557, 2014.

[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, 2007.

[6] Giuseppe Ateniese, Roberto Di Pietro, Luigi V Mancini, and Gene Tsudik. Scalable and efficient provable data possession. In *SecureComm*, pages 1–10. ACM, 2008.

[7] Mihir Bellare and Chanathip Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *ASIACRYPT 2000*, pages 531–545, 2000.

[8] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. Verifiable delegation of computation over large datasets. In P. Rogaway, editor, *CRYPTO 2011*, pages 111–131, 2011.

[9] Erik Cambria, Anupam Chattopadhyay, Eike Linn, Bappaditya Mandal, and Bebo White. Storages are not forever. *Cognitive Computation*, 9:646–658, 2017.

[10] David Cash, Alptekin Küpçü, and Daniel Wichs. Dynamic proofs of retrievability via oblivious RAM. *J. Cryptol.*, 30(1):22–57, January 2017.

[11] Ethan Cecchetti, Ben Fisch, Ian Miers, and Ari Juels. Pies: Public incompressible encodings for decentralized storage. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *CCS 2019, London, UK, November 11-15, 2019*, pages 1351–1367, 2019.

[12] Ivan Damgård, Chaya Ganesh, and Claudio Orlandi. Proofs of replicated storage without timing assumptions. In *CRYPTO 2019*, pages 355–380, 2019.

[13] Yevgeniy Dodis, Salil Vadhan, and Daniel Wichs. Proofs of retrievability via hardness amplification. In *Theory of Cryptography*, pages 109–127, 2009.

[14] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *CRYPTO 2015*, pages 585–605, 2015.

[15] Kaoutar Elkhiyaoui, Melek Önen, Monir Azraoui, and Refik Molva. Efficient techniques for publicly verifiable delegation of computation. In *11th ACM AsiaCCS*, pages 119–128, 2016.

[16] 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.

[17] 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.

[18] Dario Fiore and Rosario Gennaro. Publicly verifiable delegation of large polynomials and matrix computations, with applications. In *CCS*, pages 501–512, 2012.

[19] Ben Fisch. PoReps: Proofs of Space on Useful Data. Technical Report 678, IACR ePrint, 2018.

[20] Rūsiņš Freivalds. Fast probabilistic algorithms. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979*, volume 74 of *LNCS*, pages 57–69, Olomouc, Czechoslovakia, September 1979.

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

[22] Nick Harvey. Chernoff bound, balls and bins, congestion minimization. Lecture 3 from CPSC 536N: Randomized Algorithms, 2015.

[23] Ari Juels and Burton S Kaliski Jr. PORs: Proofs of retrievability for large files. In *14th ACM CCS*, pages 584–597. ACM, 2007.

[24] 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.

[25] B. Laurie, A. Langley, E. Kasper, and Google. Certificate Transparency. RFC 6962, IETF, June 2013.

[26] Julien Lavauzelle and Françoise Levy dit Vehel. New proofs of retrievability using locally decodable codes. In *2016 IEEE ISIT*, pages 1809–1813, July 2016.

[27] Ralph C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *CRYPTO '87*, pages 369–378, 1988.

[28] Tal Moran and Ilan Orlov. Simple proofs of space-time and rational proofs of storage. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, August 18-22*, volume 11692 of *LNCS*, pages 381–409, 2019.

[29] Paz Morillo, Carla Ràfols, and Jorge L. Villar. The kernel matrix Diffie-Hellman assumption. In J. H. Cheon and T. Takagi, editors, *ASIACRYPT 2016*, pages 729–758, 2016.

[30] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[31] David Reinsel, John Gantz, and John Rydning. The Digitization of the World from Edge to Core. Technical Report US44413318, "IDC", 2018.

[32] 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.

[33] Hovav Shacham and Brent Waters. Compact proofs of retrievability. In *Theory and Application of Cryptology and Information Security*, pages 90–107, 2008.

[34] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *ACM CCS*, pages 325–336, 2013.

[35] 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, 2012.

[36] Storj labs Inc. Storj: A decentralized cloud storage network framework. Technical Report v2, 2016.

[37] Dimitrios Vasilopoulos, Melek Önen, and Refik Molva. PORTOS: Proof of data reliability for real-world distributed outsourced storage. In *SECRYPT*, pages 173–186, 2019.