Dynamic Proofs of Retrievability via Oblivious RAM*

David Cash[1,**], Alptekin Küpçü[2,***], and Daniel Wichs[3,†]

[1] Rutgers University
[2] Koç University
[3] Northeastern University

Abstract. Proofs of retrievability allow a client to store her data on a remote server (e.g., "in the cloud") and periodically execute an efficient *audit* protocol to check that all of the data is being maintained correctly and can be recovered from the server. For efficiency, the *computation* and *communication* of the server and client during an audit protocol should be significantly smaller than reading/transmitting the data in its entirety. Although the server is only asked to access a few locations of its storage during an audit, it must *maintain full knowledge of all client data* to be able to pass.

Starting with the work of Juels and Kaliski (CCS '07), all prior solutions require that the client data is *static* and do not allow it to be efficiently updated. Indeed, they store a redundant encoding of the data on the server, so that the server must delete a large fraction of its storage to 'lose' any actual content. Unfortunately, this means that even a single bit modification to the original data will need to modify a large fraction of the server storage, which makes updates highly inefficient.

In this work, we give the first solution providing proofs of retrievability for *dynamic* storage, where the client can perform arbitrary reads/writes on any location within her data by running an efficient protocol with the server. At any point in time, the client can also execute an audit protocol to ensure that the server maintains the *latest version* of its data. The computation and communication complexity of the server and client in our protocols is only *polylogarithmic* in the size of the data. Our main idea is to split up the data into small blocks and redundantly encode each block of data individually, so that an update inside any data block only affects a few codeword symbols. The main difficulty is to prevent the server from identifying and deleting too many codeword symbols belonging to any single data block. We do so by hiding where the various codeword symbols are stored on the server and when they are being accessed by the client, using the techniques of *oblivious RAM*.

* A full version of this work is available as ePrint report 2012/550.

** Research conducted while at IBM Research, T.J. Watson.

*** Supported by TÜBİTAK, the Scientific and Technological Research Council of Turkey, under project number 112E115.

† Research conducted while at IBM Research, T.J. Watson and supported by DARPA under agreement number FA8750-11-C-0096.

T. Johansson and P. Nguyen (Eds.): EUROCRYPT 2013, LNCS 7881, pp. 279–295, 2013.

1 Introduction

Cloud storage systems (Amazon S3, Dropbox, Google Drive etc.) are becoming increasingly popular as a means of storing data reliably and making it easily accessible from any location. Unfortunately, even though the remote storage provider may not be trusted, current systems provide few security or integrity guarantees. Guaranteeing the *privacy* and *authenticity* of remotely stored data while allowing efficient access and updates is non-trivial, and relates to the study of *oblivious RAMs* and *memory checking*, which we will return to later. The main focus of this work, however, is an orthogonal question: How can we efficiently verify that the entire client data is being stored on the remote server in the first place? In other words, what prevents the server from deleting some portion of the data (say, an infrequently accessed sector) to save on storage?

Provable Storage. Motivated by the questions above, there has been much cryptography and security research in creating a provable storage mechanism, where an untrusted server can *prove* to a client that her data is kept intact. More precisely, the client can run an efficient *audit* protocol with the untrusted server, guaranteeing that the server can only pass the audit if it maintains full *knowledge* of the entire client data. This is formalized by requiring that the data can be efficiently *extracted* from the server given its state at the beginning of any successful audit. One may think of this as analogous to the notion of extractors in the definition of *zero-knowledge proofs of knowledge* [15,4].

One trivial audit mechanism, which accomplishes the above, is for the client to simply download all of her data from the server and check its authenticity (e.g., using a MAC). However, for the sake of efficiency, we insist that the *computation* and *communication* of the server and client during an audit protocol is much smaller than the potentially huge size of the client's data. In particular, the server shouldn't even have to *read* all of the client's data to run the audit protocol, let alone *transmit* it. A scheme that accomplishes the above is called a *Proof of Retrievability* (PoR).

Prior Techniques. The first PoR schemes were defined and constructed by Juels and Kaliski [18], and have since received much attention. We review the prior work and and closely related primitives (e.g., *sublinear authenticators* [21] and *provable data possession* [1]) in Section 1.2.

On a very high level, all PoR constructions share essentially the same common structure. The client stores some *redundant encoding* of her data under an erasure code on the server, ensuring that the server must delete a significant fraction of the encoding before losing any actual data. During an audit, the client then checks a few random locations of the encoding, so that a server who deleted a significant fraction will get caught with overwhelming probability.

More precisely, let us model the client's input data as a string $\mathbf{M} \in \Sigma^\ell$ consisting of ℓ symbols from some small alphabet Σ, and let $\mathsf{Enc} : \Sigma^\ell \to \Sigma^{\ell'}$ denote an erasure code that can correct the erasure of up to $\frac{1}{2}$ of its output symbols. The client stores $\mathsf{Enc}(\mathbf{M})$ on the server. During an audit, the client selects a small random subset of t out of the ℓ' locations in the encoding, and

challenges the server to respond with the corresponding values, which it then checks for authenticity (e.g., using MAC tags). Intuitively, if the server deletes more than half of the values in the encoding, it will get caught with overwhelming probability $> 1 - 2^{-t}$ during the audit, and otherwise it retains knowledge of the original data because of the redundancy of the encoding. The complexity of the audit protocol is only proportional to t which can be set to the *security parameter* and is independent of the size of the client data.[1]

Difficulty of Updates. One of the main limitations of all prior PoR schemes is that they do not support efficient updates to the client data. Under the above template for PoR, if the client wants to modify even a single location of $\mathbf{M}$, it will end up needing to change the values of at least half of the locations in $\mathsf{Enc}(\mathbf{M})$ on the server, requiring a large amount of work (linear in the size of the client data). Constructing a PoR scheme that allows for efficient updates was stated as the main open problem by Juels and Kaliski [18]. We emphasize that, in the setting of updates, the audit protocol must ensure that the server correctly maintains knowledge of the *latest version* of the client data, which includes all of the changes incurred over time. Before we describe our solution to this problem, let us build some intuition about the challenges involved by examining two natural but *flawed* proposals.

First Proposal. A natural attempt to overcome the inefficiency of updating a huge redundant encoding is to encode the data "locally" so that a change to one position of the data only affects a small number of codeword symbols. More precisely, instead of using an erasure code that takes all ℓ data symbols as input, we can use a code $\mathsf{Enc} : \Sigma^k \to \Sigma^n$ that works on small blocks of only $k \ll \ell$ symbols encoded into n symbols. The client divides the data $\mathbf{M}$ into $L = \ell/k$ *message blocks* $(\mathbf{m}_1, \ldots, \mathbf{m}_L)$, where each block $\mathbf{m}_i \in \Sigma^k$ consists of k symbols. The client redundantly encodes each message block $\mathbf{m}_i$ individually into a corresponding *codeword block* $\mathbf{c}_i = \mathsf{Enc}(\mathbf{m}_i) \in \Sigma^n$ using the above code with small inputs. Finally the client concatenates these codeword blocks to form the value $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \Sigma^{Ln}$, which it stores on the server. Auditing works as before: The client randomly chooses t of the $L \cdot n$ locations in $\mathbf{C}$ and challenges the server to respond with the corresponding codeword symbols in these locations, which it then tests for authenticity.[2] The client can now read/write to any location within her data by simply reading/writing to the n relevant codeword symbols on the server.

The above proposal can be made secure when the block-size k (which determines the complexity of reads/updates) and the number of challenged locations t (which determines the complexity of the audit) are both set to $\Omega(\sqrt{\ell})$ where ℓ is the size of the data (see the full version [8] for details). This way, the audit

[1] Some of the more advanced PoR schemes (e.g., [24,11]) optimize the communication complexity of the audit even further by cleverly compressing the t codeword symbols and their authentication tags in the server's response.

[2] This requires that we can efficiently check the *authenticity* of the remotely stored data $\mathbf{C}$, while supporting efficient updates on it. This problem is solved by *memory checking* (see our survey of related work in Section 1.2).

is likely to check sufficiently many values in *each* codeword block $\mathbf{c}_i$. Unfortunately, if we want a truly efficient scheme and set $n, t = o(\sqrt{\ell})$ to be small, then this solution becomes completely insecure. The server can delete a single codeword block $\mathbf{c}_i$ from $\mathbf{C}$ entirely, losing the corresponding message block $\mathbf{m}_i$, but still maintain a good chance of passing the above audit as long as none of the t random challenge locations coincides with the n deleted symbols, which happens with good probability.

Second Proposal. The first proposal (with small n, t) was insecure because a cheating server could easily identify the locations within $\mathbf{C}$ that correspond to a single message block and delete exactly the codeword symbols in these locations. We can prevent such attacks by pseudo-randomly permuting the locations of all of the different codeword-symbols of different codeword blocks together. That is, the client starts with the value $\mathbf{C} = (\mathbf{C}[1], \ldots, \mathbf{C}[Ln]) = (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \Sigma^{Ln}$ computed as in the first proposal. It chooses a pseudo-random permutation $\pi : [Ln] \to [Ln]$ and computes the permuted value $\mathbf{C}' := (\mathbf{C}[\pi(1)], \ldots, \mathbf{C}[\pi(Ln)])$ which it then stores on the server in an encrypted form (each codeword symbol is encrypted separately). The audit still checks t out of Ln random locations of the server storage and verifies authenticity.

It may seem that the server now cannot immediately identify and *selectively* delete codeword-symbols belonging to a single codeword block, thwarting the attack on the first proposal. Unfortunately, this modification only re-gains security in the static setting, when the client never performs any operations on the data.[3] Once the client wants to update some location of $\mathbf{M}$ that falls inside some message block $\mathbf{m}_i$, she has to reveal to the server where all of the n codeword symbols corresponding to $\mathbf{c}_i = \mathsf{Enc}(\mathbf{m}_i)$ reside in its storage since she needs to update exactly these values. Therefore, the server can later selectively delete exactly these n codeword symbols, leading to the same attack as in the first proposal.

Impossibility? Given the above failed attempts, it may even seem that truly efficient updates could be inherently incompatible with efficient audits in PoR. If an update is efficient and only changes a small subset of the server's storage, then the server can always just *ignore* the update, thereby failing to maintain knowledge of the latest version of the client data. All of the prior techniques appear ineffective against such attack. More generally, any audit protocol which just checks *a small subset of random* locations of the server's storage is unlikely to hit any of the locations involved in the update, and hence will not detect such cheating, meaning that it cannot be secure. However, this does not rule out the possibility of a very efficient solution that relies on a more clever audit protocol, which is likelier to check recently updated areas of the server's storage and therefore detect such an attack. Indeed, this property will be an important component in our actual solution.

[3] A variant of this idea was actually used by Juels and Kaliski [18] for extra efficiency in the static setting.

1.1 Our Results and Techniques

Overview of Result. In this work, we give the first solution to *dynamic PoR* that allows for efficient updates to client data. The client only keeps some short local state, and can execute arbitrary read/write operations on any location within the data by running a corresponding protocol with the server. At any point in time, the client can also initiate an audit protocol, which ensures that a passing server must have complete knowledge of the *latest version* of the client data. The cost of any read/write/audit execution in terms of server/client work and communication is only *polylogarithmic* in the size of the client data. The server's storage remains linear in the size of the client data. Therefore, our scheme is optimal in an asymptotic sense, up to polylogarithmic factors. See Section 6 for a detailed efficiency analysis.

PoR via Oblivious RAM. Our dynamic PoR solution starts with the same idea as the first proposal above, where the client redundantly encodes small blocks of her data individually to form the value $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \Sigma^{Ln}$, consisting of L codeword blocks and $\ell' = Ln$ codeword symbols, as defined previously. The goal is to then store $\mathbf{C}$ on the server in some "clever way" so that that the server cannot selectively delete too many symbols within any single codeword block $\mathbf{c}_i$, even after observing the client's read and write executions (which access exactly these symbols). As highlighted by the second proposal, simply permuting the locations of the codeword symbols of $\mathbf{C}$ is insufficient. Instead, our main idea it to store all of the individual codeword symbols of $\mathbf{C}$ on the server using an *oblivious RAM* scheme.

Overview of ORAM. Oblivious RAM (ORAM), initially defined by Goldreich and Ostrovsky [14], allows a client to outsource her *memory* to a remote server while allowing the client to perform random-access reads and writes in a *private* way. More precisely, the client has some data $\mathbf{D} \in \Sigma^d$, which she stores on the server in some carefully designed privacy-preserving form, while only keeping a short local state. She can later run efficient protocols with the server to read or write to the individual entries of $\mathbf{D}$. The read/write protocols of the ORAM scheme should be efficient, and the client/server work and communication during each such protocol should be small compared to the size of $\mathbf{D}$ (e.g., *polylogarithmic*). A secure ORAM scheme not only hides the *content* of $\mathbf{D}$ from the server, but also the *access pattern* of which *locations* in $\mathbf{D}$ the client is reading or writing in each protocol execution. Thus, the server cannot discern any correlation between the physical locations of its storage that it is asked to access during each read/write protocol execution and the logical location inside $\mathbf{D}$ that the client wants to access via this protocol.

In our work, we will also always use ORAM schemes that are *authenticated*, which means that the client can detect if the server ever sends an incorrect value. In particular, authenticated ORAM schemes ensure that the most recent version of the data is being retrieved in any accepting read execution, preventing the server from "rolling back" updates.

Construction of Dynamic PoR. A detailed technical description of our construction appears in Section 5, and below we give a simplified overview. In our PoR construction, the client starts with data $\mathbf{M} \in \Sigma^\ell$ which she splits into small message blocks $\mathbf{M} = (\mathbf{m}_1, \ldots, \mathbf{m}_L)$ with $\mathbf{m}_i \in \Sigma^k$ where the block size $k \ll \ell = Lk$ is only dependant on the security parameter. She then applies an error correcting code $\mathsf{Enc} : \Sigma^k \to \Sigma^n$ that can efficiently recover $\frac{n}{2}$ erasures to each message block individually, resulting in the value $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \Sigma^{Ln}$ where $\mathbf{c}_i = \mathsf{Enc}(\mathbf{m}_i)$. Finally, she initializes an ORAM scheme with the initial data $\mathbf{D} = \mathbf{C}$, which the ORAM stores on the server in some clever privacy-preserving form, while keeping only a short local state at the client.

Whenever the client wants to read or write to some location within her data, she uses the ORAM scheme to perform the necessary reads/writes on each of the n relevant codeword symbols of $\mathbf{C}$ (see details in Section 5). To run an audit, the client chooses t ($\approx$ security parameter) random locations in $\{1, \ldots, Ln\}$ and runs the ORAM read protocol t times to read the corresponding symbols of $\mathbf{C}$ that reside in these locations, checking them for authenticity.

Catching Disregarded Updates. First, let us start with a sanity check, to explain how the above construction can thwart a specific attack in which the server simply disregards the latest update. In particular, such attack should be caught by a subsequent audit. During the audit, the client runs the ORAM protocol to read t *random* codeword symbols and these are *unlikely* to coincide with any of the n codeword symbols modified by the latest update (recall that t and n are both small and independent of the data size ℓ). However, the ORAM scheme stores data on the server in a highly organized data-structure, and ensures that the most recently updated data is accessed during *any* subsequent "read" execution, even for an unrelated logical location. This is implied by ORAM security since we need to hide whether or not the location of a read was recently updated or not. Therefore, although the audit executes the "ORAM read" protocols on random logical locations inside $\mathbf{C}$, the ORAM scheme will end up scanning recently updated ares of the server's actual storage and check them for authenticity, ensuring that recent updates have not been disregarded.

Security and "Next-Read Pattern Hiding". The high-level security intuition for our PoR scheme is quite simple. The ORAM hides from the server where the various locations of $\mathbf{C}$ reside in its storage, even after observing the access pattern of read/write executions. Therefore it is difficult for the server to reach a state where it will fail on read executions for most locations within some single codeword block (lose data) without also failing on too many read executions altogether (lose the ability to pass an audit).

Making the above intuition formal is quite subtle, and it turns out that standard notion of ORAM security does *not* suffice. The main issue is that that the server may be able to somehow delete *all* (or most) of the n codeword symbols that fall within *some* codeword block $\mathbf{c}_i = (\mathbf{C}[j+1], \ldots, \mathbf{C}[j+n])$ without knowing *which* block it deleted. Therefore, although the server will fail on any subsequent read if and only if its location falls within the range $\{j+1, \ldots, j+n\}$, it will not learn anything about the location of the read itself since it does not

know the index j. Indeed, we will give an example of a contrived ORAM scheme where such an attack is possible and our resulting construction of PoR using this ORAM is *insecure*.

We show, however, that the intuitive reasoning above can be salvaged if the ORAM scheme achieves a new notion of security that we call *next-read pattern hiding (NRPH)*, which may be of independent interest. NRPH security considers an adversarial server that first gets to observe many read/write protocol executions performed sequentially with the client, resulting in some final client configuration $\mathcal{C}_{\mathsf{fin}}$. The adversarial server then gets to see various possibilities for how the "*next read*" operation would be executed by the client for various distinct locations, where each such execution starts from the same *fixed* client configuration $\mathcal{C}_{\mathsf{fin}}$.[4] The server should not be able to discern any *relationship* between these executions and the locations they are reading. For example, two such "next-read" executions where the client reads two consecutive locations should be indistinguishable from two executions that read two random and unrelated locations. This notion of NRPH security will be used to show that server cannot reach a state where it can *selectively* fail to respond on read queries whose location falls within some small range of a single codeword block (lose data), but still respond correctly to most completely random reads (pass an audit).

Proving Security via an Extractor. We now give a high-level overview of how our PoR extractor works. In particular, we claim that we can take any adversarial server that has a "good" chance of passing an audit and use the extractor to efficiently recover the latest version of the client data from it. The extractor initializes an "empty array" $\mathbf{C}$. It then executes random audit protocols with the server, by acting as the honest client. In particular, it chooses t random locations within the array and runs the corresponding ORAM read protocols. If the execution of the audit is successful, the extractor fills in the corresponding values of $\mathbf{C}$ that it learned during the audit execution. In either case, it then rewinds the server and runs a fresh execution of the audit, repeating this step for several iterations.

Since the server has a good chance of passing a random audit, it is easy to show that the extractor can eventually recover a large fraction, say $> \frac{3}{4}$, of the entries inside $\mathbf{C}$ by repeating this process sufficiently many times. Because of the *authenticity* of the ORAM, the recovered values are the correct ones, corresponding to the latest version of the client data. Now we need to argue that there is no codeword block $\mathbf{c}_i$ within $\mathbf{C}$ for which the extractor recovered fewer than $\frac{1}{2}$ of its codeword symbols, as this would prevent us from applying erasure decoding and recovering the underlying message block. Let FAILURE denote the above bad event. If all the recovered locations (comprising $> \frac{3}{4}$ fraction of the total) were distributed uniformly within $\mathbf{C}$ then FAILURE would occur with negligible probability, as long as the codeword size n is sufficiently large in the security parameter. We can now rely on the NRPH security of the ORAM to ensure that FAILURE also happens with negligible probability in our case.

[4] This is in contrast to the standard sequential operations where the client state is updated after each execution.

We can think of the FAILURE event as a function of the locations queried by the extractor in each audit execution, and the set of executions on which the server fails. If the malicious server can cause FAILURE to occur, it means that it can distinguish the pattern of locations actually queried by the extractor during the audit executions (for which the FAILURE event occurs) from a randomly permuted pattern of locations (for which the FAILURE event does not occur with overwhelming probability). Note that the use of rewinding between the audit executions of the extractor forces us to rely on NRPH security rather than just standard ORAM security.

The above presents the high-level intuition and is somewhat oversimplified. See Section 4 for the formal definition of NRPH security and Section 5 for the formal description of our dynamic PoR scheme and a rigorous proof of security.

Achieving Next-Read Pattern Hiding. We show that standard ORAM security does *not* generically imply NRPH security, by giving a contrived scheme that satisfies the former but not the latter. Nevertheless, all natural ORAM constructions in the literature *do* essentially satisfy NRPH security. In the full version [8], we look at one particularly efficient ORAM construction of Goodrich and Mitzenmacher [16] in depth, and prove that (with minor modifications) it is NRPH secure.

Contributions. We call our final scheme PORAM since it combines the techniques and security of PoR and ORAM. In particular, other than providing provable dynamic cloud storage as was our main goal, our scheme also satisfies the strong *privacy* guarantees of ORAM, meaning that it hides all contents of the remotely stored data as well as the access pattern of which locations are accessed when. It also provides strong *authenticity* guarantees (same as *memory checking*; see Section 1.2), ensuring that any "read" execution with a malicious remote server is guaranteed to return the latest version of the data (or detect cheating). In brief, our contributions can be summarized as follows:

- We give the first asymptotically efficient solution to PoR for outsourced dynamic data, where a successful audit ensures that the server knows the latest version of the client data. In particular:
  - Client storage is small and independent of the data size.
  - Server storage is linear in the data size, expanding it by only a small constant factor.
  - Communication and computation of client and server during *read*, *write*, and *audit* executions are polylogarithmic in the size of the client data.
- Our scheme also achieves strong *privacy* and *authenticity* guarantees, matching those of *oblivious RAM* and *memory checking*.

We mention that the PORAM scheme is simple to implement and has low concrete efficiency overhead *on top of* an underlying ORAM scheme with NRPH security. There is much recent and ongoing research activity in instantiating/implementing truly practical ORAM schemes, which are likely to yield correspondingly practical instantiations of our PORAM protocol.

1.2 Related Work

Proofs of retrievability for *static* data were initially defined and constructed by Juels and Kaliski [18], building on a closely related notion called sublinear-authenticators of Naor and Rothblum [21]. Concurrently, Ateniese et al. [1] defined another related primitive called *provable data possession* (PDP). Since then, there has been much ongoing research activity on PoR and PDP schemes.

PoR vs. PDP. The main difference between PoR and PDP is the notion of security that they achieve. A PoR audit guarantees that the server maintains knowledge of *all* of the client data, while a PDP audit only ensures that the server is storing *most* of the client data. For example, in a PDP scheme, the server may lose a small portion of client data (say 1 MB out of a 10 GB file) and may maintain an high chance of passing a future audit. On a technical level, the main difference in most prior PDP/PoR constructions is that PoR schemes store a *redundant encoding* of the client data on the server. For a detailed comparison, see Küpçü [19,20].

Static Data. PoR and PDP schemes for static data (without updates) have received much research attention [24,11,7,2], with works improving on communication efficiency and exact security, yielding essentially optimal solutions. Another interesting direction has been to extend these works to the multi-server setting [6,9,10] where the client can use the audit mechanism to identify faulty machines and recover the data from the others.

Dynamic Data. The works of Ateniese et al. [3], Erway et al. [13] and Wang et al. [27] show how to achieve PDP security for *dynamic data*, supporting efficient updates. This is closely related to work on memory checking [5,21,12], which studies how to authenticate remotely stored dynamic data so as to allow efficient reads/writes, while being able to verify the authenticity of the latest version of the data (preventing the server from "rolling back" updates and using an old version). Unfortunately, these techniques alone cannot be used to achieve the stronger notion of PoR security. Indeed, the main difficulty that we resolve in this work, how to efficiently update *redundantly encoded data*, does not come up in the context of PDP.

A recent work of Stefanov et al. [26] considers PoR for dynamic data, but in a more complex setting where an additional trusted "portal" performs some operations on behalf of the client, and can cache updates for an extended period of time. It is not clear if these techniques can be translated to the basic client/server setting, which we consider here. However, even in this modified setting, the complexity of the updates and the audit in that work is proportional to *square-root* of the data size, whereas ours is *polylogarithmic*.

2 Preliminaries

Notation. Throughout, we use λ to denote the *security parameter*. We identify *efficient* algorithms as those running in (probabilistic) polynomial time in λ and their input lengths, and identify *negligible* quantities (e.g., acceptable error

probabilities) as $\mathsf{negl}(\lambda) = 1/\lambda^{\omega(1)}$, meaning that they are asymptotically smaller than $1/\lambda^c$ for every constant $c > 0$. For $n \in \mathbb{N}$, we define the set $[n] \stackrel{\text{def}}{=} \{1, \ldots, n\}$. We use the notation $(k \bmod n)$ to denote the unique integer $i \in \{0, \ldots, n-1\}$ such that $i = k \pmod n$.

Erasure Codes. We say that $(\mathsf{Enc}, \mathsf{Dec})$ is an $(n, k, d)_\Sigma$-*code with efficient erasure decoding* over an alphabet Σ if the original message can always be recovered from a corrupted codeword with at most $d-1$ erasures. That is, for every *message* $\mathbf{m} = (m_1, \ldots, m_k) \in \Sigma^k$ giving a *codeword* $\mathbf{c} = (c_1, \ldots, c_n) = \mathsf{Enc}(\mathbf{m})$, and every corrupted codeword $\tilde{\mathbf{c}} = (\tilde{c}_1, \ldots, \tilde{c}_n)$ such that $\tilde{c}_i \in \{c_i, \perp\}$ and the number of erasures is $|\{i \in [n] \ : \ \tilde{c}_i = \perp\}| \leq d-1$, we have $\mathsf{Dec}(\tilde{\mathbf{c}}) = \mathbf{m}$. We say that a code is *systematic* if, for every message $\mathbf{m}$, the codeword $\mathbf{c} = \mathsf{Enc}(\mathbf{m})$ contains $\mathbf{m}$ in the first k positions $c_1 = m_1, \ldots, c_k = m_k$. A systematic variant of the Reed-Solomon code achieves the above for any integers $n > k$ and any *field* Σ of size $|\Sigma| \geq n$ with $d = n - k + 1$.

Virtual Memory. We think of *virtual memory* $\mathbf{M}$, with *word-size* w and *length* ℓ, as an array $\mathbf{M} \in \Sigma^\ell$ where $\Sigma \stackrel{\text{def}}{=} \{0,1\}^w$. We assume that, initially, each location $\mathbf{M}[i]$ contains the special *uninitialized symbol* $\mathbf{0} = 0^w$. Throughout, we will think of ℓ as some large polynomial in the security parameter, which upper bounds the amount of memory that can be used.

Outsourcing Virtual Memory. In the next two sections, we look at two primitives: *dynamic PoR* and *ORAM*. These primitives allow a client to *outsource* some virtual memory $\mathbf{M}$ to a remote server, while providing useful security guarantees. Reading and writing to some location of $\mathbf{M}$ now takes on the form of a protocol execution with the server. The goal is to provide security while preserving efficiency in terms of client/server computation, communication, and the number of server-memory accesses per operation, which should all be *poly-logarithmic* in the length ℓ. We also want to optimize the size of the client storage (independent of ℓ) and server storage (not much larger than ℓ).

3 Dynamic PoR

A *Dynamic PoR* scheme consists of protocols $\mathbf{P}\mathsf{Init}, \mathbf{P}\mathsf{Read}, \mathbf{P}\mathsf{Write}, \mathsf{Audit}$ between two *stateful* parties: a client $\mathcal{C}$ and a server $\mathcal{S}$. The server acts as the curator for some virtual memory $\mathbf{M}$, which the client can *read*, *write* and *audit* by initiating the corresponding interactive protocols:

- $\mathbf{P}\mathsf{Init}(1^\lambda, 1^w, \ell)$: This protocol corresponds to the client initializing an (empty) virtual memory $\mathbf{M}$ with word-size w and length ℓ, which it supplies as inputs.
- $\mathbf{P}\mathsf{Read}(i)$: This protocol corresponds to the client reading $v = \mathbf{M}[i]$, where it supplies the input i and outputs some value v at the end.
- $\mathbf{P}\mathsf{Write}(i, v)$: This protocol corresponds to setting $\mathbf{M}[i] := v$, where the client supplies the inputs i, v.
- Audit: This protocol is used by the client to verify that the server is maintaining the memory contents correctly so that they remain retrievable. The client outputs a decision $b \in \{\texttt{accept}, \texttt{reject}\}$.

The client $\mathcal{C}$ in the protocols may be *randomized*, but we assume (w.l.o.g.) that the honest server $\mathcal{S}$ is deterministic. At the conclusion of the $\mathbf{P}\mathsf{Init}$ protocol, both the client and the server create some long-term local state, which each party will update during the execution of each of the subsequent protocols. The client may also output `reject` during the execution of the $\mathbf{P}\mathsf{Init}$, $\mathbf{P}\mathsf{Read}$, $\mathbf{P}\mathsf{Write}$ protocols, to denote that it detected some misbehavior of the server. Note that we assume that the virtual memory is initially *empty*, but if the client has some initial data, she can write it onto the server block-by-block immediately after initialization. For ease of presentation, we may assume that the state of the client and the server always contains the security parameter, and the memory parameters $(1^\lambda, 1^w, \ell)$. We now define the three properties of a dynamic PoR scheme: *correctness*, *authenticity* and *retrievability*. For these definitions, we say that $P = (op_0, op_1, \ldots, op_q)$ is a dynamic PoR *protocol sequence* if $op_0 = \mathbf{P}\mathsf{Init}(1^\lambda, 1^w, \ell)$ and, for $j > 0$, $op_j \in \{\mathbf{P}\mathsf{Read}(i), \mathbf{P}\mathsf{Write}(i, v), \mathsf{Audit}\}$ for some index $i \in [\ell]$ and value $v \in \{0,1\}^w$.

Correctness. If the client and the server are both *honest* and $P = (op_0, \ldots, op_q)$ is some protocol sequence, then we require the following to occur with probability 1 over the randomness of the client:

- Each execution of a protocol $op_j = \mathbf{P}\mathsf{Read}(i)$ results in the client outputting the correct value $v = \mathbf{M}[i]$, matching what would happen if the corresponding operations were performed directly on a memory $\mathbf{M}$. In particular, v is the value contained in the most recent prior write operation with location i, or, if no such prior operation exists, $v = \mathbf{0}$.
- Each execution of the Audit protocol results in the decision $b = \mathtt{accept}$.

Authenticity. We require that the client can always *detect* if any protocol message sent by the server deviates from honest behavior. More precisely, consider the following game $\mathtt{AuthGame}_{\tilde{\mathcal{S}}}(\lambda)$ between a malicious server $\tilde{\mathcal{S}}$ and a challenger:

- The malicious server $\tilde{\mathcal{S}}(1^\lambda)$ specifies a valid protocol sequence $P = (op_0, \ldots, op_q)$.
- The challenger initializes a copy of the honest client $\mathcal{C}$ and the (deterministic) honest server $\mathcal{S}$. It sequentially executes $op_0, \ldots, op_q$ between $\mathcal{C}$ and the malicious server $\tilde{\mathcal{S}}$ while, in parallel, also passing a copy of every message from $\mathcal{C}$ to the honest server $\mathcal{S}$.
- If, at any point during the execution of some op_j, any protocol message given by $\tilde{\mathcal{S}}$ differs from that of $\mathcal{S}$, and the client $\mathcal{C}$ does not output `reject`, the adversary wins and the game outputs 1. Else 0.

For any efficient adversarial server $\tilde{\mathcal{S}}$, we require $\Pr[\mathtt{AuthGame}_{\tilde{\mathcal{S}}}(\lambda) = 1] \leq \mathsf{negl}(\lambda)$. Note that authenticity and correctness together imply that the client will always either read the correct value corresponding to the latest contents of the virtual memory or reject whenever interacting with a malicious server.

Retrievability. Finally we define the main purpose of a dynamic PoR scheme, which is to ensure that the client data remains retrievable. We wish to guarantee that, whenever the malicious server is in a state with a reasonable probability δ

of successfully passing an audit, he must *know* the entire content of the client's virtual memory $\mathbf{M}$. As in "proofs of knowledge", we formalize *knowledge* via the existence of an efficient *extractor* $\mathcal{E}$ which can recover the value $\mathbf{M}$ given (black-box) access to the malicious server.

More precisely, we define the game $\texttt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p)$ between a malicious server $\tilde{\mathcal{S}}$, extractor $\mathcal{E}$, and challenger:

- The malicious server $\tilde{\mathcal{S}}(1^\lambda)$ specifies a protocol sequence $P = (op_0, \ldots, op_q)$. Let $\mathbf{M} \in \Sigma^\ell$ be the correct value of the memory contents at the end of executing P.
- The challenger initializes a copy of the honest client $\mathcal{C}$ and sequentially executes $op_0, \ldots, op_q$ between $\mathcal{C}$ and $\tilde{\mathcal{S}}$. Let $\mathcal{C}_{\mathsf{fin}}$ and $\tilde{\mathcal{S}}_{\mathsf{fin}}$ be the final configurations (states) of the client and malicious server at the end of this interaction, including all of the random coins of the malicious server. Define the success-probability $\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) \stackrel{\text{def}}{=} \Pr\left[\tilde{\mathcal{S}}_{\mathsf{fin}} \stackrel{\mathsf{Audit}}{\longleftrightarrow} \mathcal{C}_{\mathsf{fin}} = \texttt{accept}\right]$ as the probability that an execution of a subsequent Audit protocol between $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and $\mathcal{C}_{\mathsf{fin}}$ results in the latter outputting $\texttt{accept}$. The probability is only over the random coins of $\mathcal{C}_{\mathsf{fin}}$ during this execution.
- Run $\mathbf{M}' \leftarrow \mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p)$, where the extractor $\mathcal{E}$ gets *black-box rewinding access* to the malicious server in its final configuration $\tilde{\mathcal{S}}_{\mathsf{fin}}$, and attempts to extract out the memory contents as $\mathbf{M}'$.[5]
- If $\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) \geq 1/p$ and $\mathbf{M}' \neq \mathbf{M}$ then output 1, else 0.

We require that there exists a probabilistic-poly-time extractor $\mathcal{E}$ such that, for every efficient malicious server $\tilde{\mathcal{S}}$ and every polynomial $p = p(\lambda)$ we have $\Pr[\texttt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p) = 1] \leq \mathsf{negl}(\lambda)$.

The above says that whenever the malicious server reaches some state $\tilde{\mathcal{S}}_{\mathsf{fin}}$ in which it maintains a $\delta \geq 1/p$ probability of passing the *next audit*, the extractor $\mathcal{E}$ will be able to extract out the correct memory contents $\mathbf{M}$ from $\tilde{\mathcal{S}}_{\mathsf{fin}}$, meaning that the server must retain full *knowledge* of $\mathbf{M}$ in this state. The extractor is efficient, but can run in time polynomial in p and the size of the memory ℓ.

A Note on Adaptivity. We defined the above *authenticity* and *retrievability* properties assuming that the sequence of read/write operations is adversarial, but is chosen *non-adaptively*, before the adversarial server sees any protocol executions. This seems to be sufficient in most realistic scenarios, where the server is unlikely to have any influence on which operations the client wants to perform. It also matches the security notions in prior works on ORAM. Nevertheless, we note that our final results also achieve adaptive security, where the attacker can choose the sequence of operations op_i adaptively after seeing the execution of previous operations, if the underlying ORAM satisfies this notion. Indeed, most prior ORAM solutions seem to do so, but it was never included in their analyses.

[5] This is similar to the extractor in zero-knowledge proofs of knowledge. In particular $\mathcal{E}$ can execute protocols with the malicious server in its state $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and rewind it back to this state at the end of the execution.

4 Oblivious RAM with Next-Read Pattern Hiding

An ORAM consists of protocols $(\mathbf{OInit}, \mathbf{ORead}, \mathbf{OWrite})$ between a client $\mathcal{C}$ and a server $\mathcal{S}$, with the same syntax as the corresponding protocols in PoR. We will also extend the syntax of $\mathbf{ORead}$ and $\mathbf{OWrite}$ to allow for reading/writing from/to multiple distinct locations simultaneously. That is, for arbitrary $t \in \mathbb{N}$, we define the protocol $\mathbf{ORead}(i_1, \ldots, i_t)$ for *distinct* indices $i_1, \ldots, i_t \in [\ell]$, in which the client outputs $(v_1, \ldots, v_t)$ corresponding to reading $v_1 = \mathbf{M}[i_1], \ldots, v_t = \mathbf{M}[i_t]$. Similarly, we define the protocol $\mathbf{OWrite}(i_t, \ldots, i_t; v_1, \ldots, v_t)$ for *distinct* indices $i_1, \ldots, i_t \in [\ell]$, which corresponds to setting $\mathbf{M}[i_1] := v_1, \ldots, \mathbf{M}[i_t] := v_t$.

We say that $P = (op_0, \ldots, op_q)$ is an *ORAM protocol sequence* if $op_0 = \mathbf{OInit}(1^\lambda, 1^w, \ell)$ and, for $j > 0$, op_j is a valid (multi-location) read/write operation. We require that an ORAM construction needs to satisfy *correctness* and *authenticity*, which are defined the same way as in PoR. (Traditionally, authenticity is not always defined/required for ORAM. However, it is crucial for our use. As noted in several prior works, it can often be added at almost no cost to efficiency. It can also be added generically by running a *memory checking* scheme on top of ORAM.) We now define a new property called *next-read pattern hiding*.

Next-Read Pattern Hiding. Consider an *honest-but-curious* server $\mathcal{A}$ who observes the execution of some protocol sequence P with a client $\mathcal{C}$ resulting in the final client configuration $\mathcal{C}_{\mathsf{fin}}$. At the end of this execution, $\mathcal{A}$ gets to observe how $\mathcal{C}_{\mathsf{fin}}$ would execute the *next* read operation $\mathbf{ORead}(i_1, \ldots, i_t)$ for various different t-tuples $(i_1, \ldots, i_t)$ of locations, but always starting in the same client state $\mathcal{C}_{\mathsf{fin}}$. We require that $\mathcal{A}$ cannot observe any correlation between these next-read executions and their locations, up to *equality*. That is, $\mathcal{A}$ should not be able to distinguish if $\mathcal{C}_{\mathsf{fin}}$ instead executes the next-read operations on *permuted locations* $\mathbf{ORead}(\pi(i_1), \ldots, \pi(i_t))$ for a permutation $\pi : [\ell] \to [\ell]$.

More formally, we define $\mathtt{NextReadGame}^b_{\mathcal{A}}(\lambda)$, for $b \in \{0, 1\}$, between an adversary $\mathcal{A}$ and a challenger:

- The attacker $\mathcal{A}(1^\lambda)$ chooses an ORAM protocol sequence $P_1 = (op_0, \ldots, op_{q_1})$. It also chooses a sequence $P_2 = (rop_1, \ldots, rop_{q_2})$ of valid multi-location read operations, where each operation is of the form $rop_j = \mathbf{ORead}(i_{j,1}, \ldots, i_{j,t_j})$ with t_j distinct locations. Lastly, it chooses a permutation $\pi : [\ell] \to [\ell]$. For each rop_j in P_2, define a permuted version $rop'_j := \mathbf{ORead}(\pi(i_{j,1}), \ldots, \pi(i_{j,t_j}))$. The game now proceeds in two stages.
- *Stage I.* The challenger initializes the honest client $\mathcal{C}$ and the (deterministic) honest server $\mathcal{S}$. It sequentially executes the protocols $P = (op_0, \ldots, op_{q_1})$ between $\mathcal{C}$ and $\mathcal{S}$. Let $\mathcal{C}_{\mathsf{fin}}, \mathcal{S}_{\mathsf{fin}}$ be the final configuration of the client and server at the end.
- *Stage II.* For each $j \in [q_2]$: challenger either executes the original operation rop_j if $b = 0$, or the permuted operation rop'_j if $b = 1$, between $\mathcal{C}$ and $\mathcal{S}$. At the end of each operation execution it resets the configuration of the client and server back to $\mathcal{C}_{\mathsf{fin}}, \mathcal{S}_{\mathsf{fin}}$ respectively, before the next execution.
- The adversary $\mathcal{A}$ is given the transcript of all the protocol executions in stages I and II, and outputs a bit $\tilde{b}$ which we define as the output of the game.

Note that, since the honest server $\mathcal{S}$ is deterministic, seeing the protocol transcripts between $\mathcal{S}$ and $\mathcal{C}$ is the same as seeing the entire internal state of $\mathcal{S}$ at any point time.

We require that, for every efficient $\mathcal{A}$, we have

$$\left|\Pr[\mathtt{NextReadGame}^0_{\mathcal{A}}(\lambda)=1]-\Pr[\mathtt{NextReadGame}^1_{\mathcal{A}}(\lambda)=1]\right|\leq \mathsf{negl}(\lambda).$$

5 PORAM: Dynamic PoR via ORAM

We now give our construction of dynamic PoR, using ORAM. Since the ORAM security properties are preserved by the construction as well, we happen to achieve ORAM and dynamic PoR simultaneously. Therefore, we call our construction PORAM.

Overview of Construction. Let $(\mathsf{Enc}, \mathsf{Dec})$ be an $(n, k, d = n-k+1)_{\Sigma}$ systematic code with efficient erasure decoding over the alphabet $\Sigma = \{0,1\}^w$ (e.g., the systematic Reed-Solomon code over $\mathbb{F}_{2^w}$). Our construction of dynamic PoR will interpret the memory $\mathbf{M} \in \Sigma^{\ell}$ as consisting of $L = \ell/k$ consecutive *message blocks*, each having k alphabet symbols (assume k is small and divides ℓ). The construction implicitly maps operation on $\mathbf{M}$ to operations on *encoded memory* $\mathbf{C} \in (\Sigma)^{\ell_{\mathsf{code}}=Ln}$, which consists of L *codeword blocks* with n alphabet symbols each. The L codeword blocks $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L)$ are simply the encoded versions of the corresponding message blocks in $\mathbf{M} = (\mathbf{m}_1, \ldots, \mathbf{m}_L)$ with $\mathbf{c}_q = \mathsf{Enc}(\mathbf{m}_q)$ for $q \in [L]$. This means that, for each $i \in [\ell]$, the value of the memory location $\mathbf{M}[i]$ can only affect the values of the encoded-memory locations $\mathbf{C}[j+1], \ldots, \mathbf{C}[j+n]$ where $j = n \cdot \lfloor i/k \rfloor$. Furthermore, since the encoding is *systematic*, we have $\mathbf{M}[i] = \mathbf{C}[j+u]$ where $u = (i \mod k) + 1$. To read the memory location $\mathbf{M}[i]$, the client will use ORAM to read the codeword location $\mathbf{C}[j+u]$. To write to the memory location $\mathbf{M}[i] := v$, the client needs to update the entire corresponding codeword block. She does so by first using ORAM to read the corresponding codeword block $\mathbf{c} = (\mathbf{C}[j+1], \ldots, \mathbf{C}[j+n])$, and decodes to obtain the original memory block $\mathbf{m} = \mathsf{Dec}(\mathbf{c})$. She then locally updates the memory block by setting $\mathbf{m}[u] := v$, re-encodes the updated memory block to get $\mathbf{c}' = (c_1, \ldots, c_n) := \mathsf{Enc}(\mathbf{m})$ and uses the ORAM to write $\mathbf{c}'$ back into the encoded memory, setting $\mathbf{C}[j+1] := c'_1, \ldots, \mathbf{C}[j+n] := c'_n$.

The Construction. Our PORAM construction is defined for some parameters $n > k, t \in \mathbb{N}$. Let $\mathbf{O} = (\mathbf{OInit}, \mathbf{ORead}, \mathbf{OWrite})$ be an ORAM. Let $(\mathsf{Enc}, \mathsf{Dec})$ be an $(n, k, d = n-k+1)_{\Sigma}$ systematic code with efficient erasure decoding over the alphabet $\Sigma = \{0,1\}^w$ (e.g., the systematic Reed-Solomon code over $\mathbb{F}_{2^w}$).

- $\mathbf{P}\mathsf{Init}(1^\lambda, 1^w, \ell)$: Assume k divides ℓ and let $\ell_{\mathsf{code}} := n \cdot (\ell/k)$. Run the $\mathbf{O}\mathsf{Init}(1^\lambda, 1^w, \ell_{\mathsf{code}})$ protocol.
- $\mathbf{P}\mathsf{Read}(i)$: Let $i' := n \cdot \lfloor i/k \rfloor + (i \mod k) + 1$ and run the $\mathbf{O}\mathsf{Read}(i')$ protocol.
- $\mathbf{P}\mathsf{Write}(i, v)$: Set $j := n \cdot \lfloor i/k \rfloor$ and $u := (i \mod k) + 1$.
  - Run $\mathbf{O}\mathsf{Read}(j+1, \ldots, j+n)$ and get output $\mathbf{c} = (c_1, \ldots, c_n)$.
  - Decode $\mathbf{m} = (m_1, \ldots, m_k) = \mathsf{Dec}(\mathbf{c})$.

- Modify position u of $\mathbf{m}$ by locally setting $m_u := v$. Re-encode the modified message-block $\mathbf{m}$ by setting $\mathbf{c}' = (c'_1, \ldots, c'_n) := \mathsf{Enc}(\mathbf{m})$.
- Run $\mathbf{O}\mathsf{Write}(j+1, \ldots, j+n; c'_1, \ldots, c'_n)$.

– Audit: Pick t distinct indices $j_1, \ldots, j_t \in [\ell_{\mathsf{code}}]$ at random. Run $\mathbf{O}\mathsf{Read}$ $(j_1, \ldots, j_t)$ and return `accept` iff the protocol finished without outputting `reject`.

If, any ORAM protocol execution in the above scheme outputs `reject`, the client enters a special rejection state in which it stops responding and automatically outputs `reject` for any subsequent protocol execution.

As our main result, we now prove that if the ORAM scheme satisfies next-read pattern hiding (NRPH) security then the PORAM construction above is also a secure dynamic PoR scheme. See the full version [8] for a proof of the following theorem.

Theorem 1. *Assume that* $\mathbf{O} = (\mathbf{O}\mathsf{Init}, \mathbf{O}\mathsf{Read}, \mathbf{O}\mathsf{Write})$ *is an ORAM with next-read pattern hiding (NRPH) security, and we choose parameters* $k = \Omega(\lambda)$, $k/n = (1 - \Omega(1))$, $t = \Omega(\lambda)$. *Then the above scheme* $\mathsf{PORAM} = (\mathbf{P}\mathsf{Init}, \mathbf{P}\mathsf{Read}, \mathbf{P}\mathsf{Write}, \mathsf{Audit})$ *is a dynamic PoR scheme.*

ORAM with NPRH Security. The notion of ORAM was introduced by Goldreich and Ostrovsky [14], who also introduced the so-called *hierarchical scheme*. Since then several improvements to the hierarchical scheme have been given, including improved rebuild phases and the use of advanced hashing techniques (e.g., [23,16] etc.).

In the full version of our work [8], we examine a particular ORAM scheme of Goodrich and Mitzenmacher [16] and show that (with minor modifications) it satisfies *next-read pattern hiding* security. Therefore, this scheme can be used to instantiate our PORAM construction. We note that most other ORAM schemes from the literature that follow the hierarchical structure also seemingly satisfy next-read pattern hiding, and we only focus on the above example for concreteness. However, in the full version of our work, we show that it is *not* the case that *every* ORAM scheme satisfies next-read pattern hiding, and in fact give an example of a contrived scheme which does not satisfy this notion and makes our construction of PORAM completely insecure. We also believe that there are natural schemes, such as the ORAM of Shi et al. [25], which do not satisfy this notion. Therefore, next-read pattern hiding is a meaningful property beyond standard ORAM security and must be examined carefully.

6 Efficiency

We now look at the efficiency of our PORAM construction (when instantiated with the ORAM scheme of Goodrich-Mitzemacher [16] with the worst-case complexity optimization [17,22]). We analyze efficiency in three ways: firstly, we look at the overhead of PORAM scheme on top of just storing the data inside of the ORAM , secondly, we look at the overall efficiency of PORAM, and thirdly, we compare it with dynamic PDP [13,27] which does not employ erasure codes

and does not provide full retrievability guarantee. In the table below, ℓ denotes the size of the client data and λ is the security parameter. We assume that the ORAM scheme uses a PRF whose computation takes $O(\lambda)$ work.

PORAM Efficiency	**vs. ORAM**	**Overall**	**vs. Dynamic PDP** [13]
Client Storage	Same	$O(\lambda)$	Same
Server Storage	$\times\ O(1)$	$O(\ell)$	$\times\ O(1)$
Read Complexity	$\times\ O(1)$	$O(\lambda \log^2 \ell)$	$\times\ O(\log \ell)$
Write Complexity	$\times\ O(\lambda)$	$O(\lambda^2 \times \log^2 \ell)$	$\times\ O(\lambda \times \log \ell)$
Audit Complexity	Read $\times\ O(\lambda)$	$O(\lambda^2 \times \log^2 \ell)$	$\times\ O(\log \ell)$

By modifying the underlying ORAM to dynamically resize tables during rebuilds, the resulting PORAM instantiation can achieve the same efficiency measures as above with ℓ taken to be amount of memory currently used, rather than the maximum memory use.

Disclaimer

References

1. Ateniese, G., Burns, R.C., Curtmola, R., Herring, J., Kissner, L., Peterson, Z.N.J., Song, D.: Provable data possession at untrusted stores. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM CCS 2007, pp. 598–609. ACM Press (October 2007)
2. Ateniese, G., Kamara, S., Katz, J.: Proofs of storage from homomorphic identification protocols. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 319–333. Springer, Heidelberg (2009)
3. Ateniese, G., Pietro, R.D., Mancini, L.V., Tsudik, G.: Scalable and efficient provable data possession. Cryptology ePrint Archive, Report 2008/114 (2008)
4. Bellare, M., Goldreich, O.: On defining proofs of knowledge. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 390–420. Springer, Heidelberg (1993)
5. Blum, M., Evans, W.S., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. Algorithmica 12(2/3), 225–244 (1994)
6. Bowers, K.D., Juels, A., Oprea, A.: HAIL: a high-availability and integrity layer for cloud storage. In: Al-Shaer, E., Jha, S., Keromytis, A.D. (eds.) ACM CCS 2009, pp. 187–198. ACM Press (November 2009)
7. Bowers, K.D., Juels, A., Oprea, A.: Proofs of retrievability: theory and implementation. In: Sion, R., Song, D. (eds.) CCSW, pp. 43–54. ACM (2009)
8. Cash, D., Kupcu, A., Wichs, D.: Dynamic proofs of retrievability via oblivious ram. Cryptology ePrint Archive, Report 2012/550 (2012)

9. Chen, B., Curtmola, R., Ateniese, G., Burns, R.C.: Remote data checking for network coding-based distributed storage systems. In: Perrig, A., Sion, R. (eds.) CCSW, pp. 31–42. ACM (2010)
10. Curtmola, R., Khan, O., Burns, R., Ateniese, G.: Mr-pdp: Multiple-replica provable data possession. In: ICDCS (2008)
11. Dodis, Y., Vadhan, S., Wichs, D.: Proofs of retrievability via hardness amplification. In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 109–127. Springer, Heidelberg (2009)
12. Dwork, C., Naor, M., Rothblum, G.N., Vaikuntanathan, V.: How efficient can memory checking be? In: Reingold, O. (ed.) TCC 2009. LNCS, vol. 5444, pp. 503–520. Springer, Heidelberg (2009)
13. Erway, C.C., Küpçü, A., Papamanthou, C., Tamassia, R.: Dynamic provable data possession. In: Al-Shaer, E., Jha, S., Keromytis, A.D. (eds.) ACM CCS 2009, pp. 213–222. ACM Press (November 2009)
14. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM 43(3), 431–473 (1996)
15. Goldwasser, S., Micali, S., Rackoff, C.: The knowledge complexity of interactive proof systems. SIAM Journal on Computing 18(1), 186–208 (1989)
16. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) ICALP 2011, Part II. LNCS, vol. 6756, pp. 576–587. Springer, Heidelberg (2011)
17. Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Oblivious RAM simulation with efficient worst-case access overhead. In: CCSW, pp. 95–100 (2011)
18. Juels, A., Kaliski Jr., B.S.: Pors: proofs of retrievability for large files. In: Ning, P., di Vimercati, S.D.C., Syverson, P.F. (eds.) ACM CCS 2007, pp. 584–597. ACM Press (October 2007)
19. Küpçü, A.: Efficient Cryptography for the Next Generation Secure Cloud. PhD thesis, Brown University (2010)
20. Küpçü, A.: Efficient Cryptography for the Next Generation Secure Cloud: Protocols, Proofs, and Implementation. Lambert Academic Publishing (2010)
21. Naor, M., Rothblum, G.N.: The complexity of online memory checking. In: 46th FOCS, pp. 573–584. IEEE Computer Society Press (October 2005)
22. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: 29th ACM STOC, pp. 294–303. ACM Press (May 1997)
23. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 502–519. Springer, Heidelberg (2010)
24. Shacham, H., Waters, B.: Compact proofs of retrievability. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 90–107. Springer, Heidelberg (2008)
25. Shi, E., Chan, T.-H.H., Stefanov, E., Li, M.: Oblivious RAM with $o((\log n)^3)$ worst-case cost. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 197–214. Springer, Heidelberg (2011)
26. Stefanov, E., van Dijk, M., Oprea, A., Juels, A.: Iris: A scalable cloud file system with efficient integrity checks. Cryptology ePrint Archive, Report 2011/585 (2011)
27. Wang, Q., Wang, C., Li, J., Ren, K., Lou, W.: Enabling public verifiability and data dynamics for storage security in cloud computing. In: Backes, M., Ning, P. (eds.) ESORICS 2009. LNCS, vol. 5789, pp. 355–370. Springer, Heidelberg (2009)