Dynamic Proofs of Retrievability via Oblivious RAM

David Cash* Alptekin Küpçü† Daniel Wichs‡

September 21, 2012

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 to this problem crucially assume that the client data is *static* and do not allow it to be efficiently updated. Indeed, they all 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. Overcoming this limitation was left as the main open problem by all prior works.

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 execute an efficient audit protocol to ensure that the server maintains the *latest version* of the client data. The computation and communication complexity of the server and client in our protocols is only *polylogarithmic* in the size of the client's data. The starting point of our solution 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 for any individual data block are stored on the server and when they are being accessed by the client, using the algorithmic techniques of *oblivious RAM*.

*IBM Research, T.J. Watson. Hawthorne, NY, USA. cdc@gatech.edu

†Koç Univesity. Istanbul, Turkey. akupcu@ku.edu.tr

‡IBM Research, T.J. Watson. Hawthorne, NY, USA. wichs@cs.nyu.edu

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 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 *zero-knowledge proofs of knowledge* [14, 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 [19], and have since received much attention. We review the prior work and and closely related primitives (e.g., *sublinear auhtenticators* [23] 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 $\mathsf{Enc}(\mathbf{M})$

[1]Some of the more advanced PoR schemes (e.g., [27] [10]) 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.

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 [19]. 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 Appendix A for details). This way, the audit 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 attack is 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

[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).

[3] A variant of this idea was actually used by Juels and Kaliski [19] for extra efficiency in the static setting.

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.[4] 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.

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 7 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 [13], 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 sores 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.

We review the literature and efficiency of ORAM schemes in Section 6. 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

[4] The above only holds when the complexity of the updates and the audit are both $o(\sqrt{\ell})$, where ℓ is the size of the data. See Appendix A for a simple protocol of this form that archives square-root complexity.

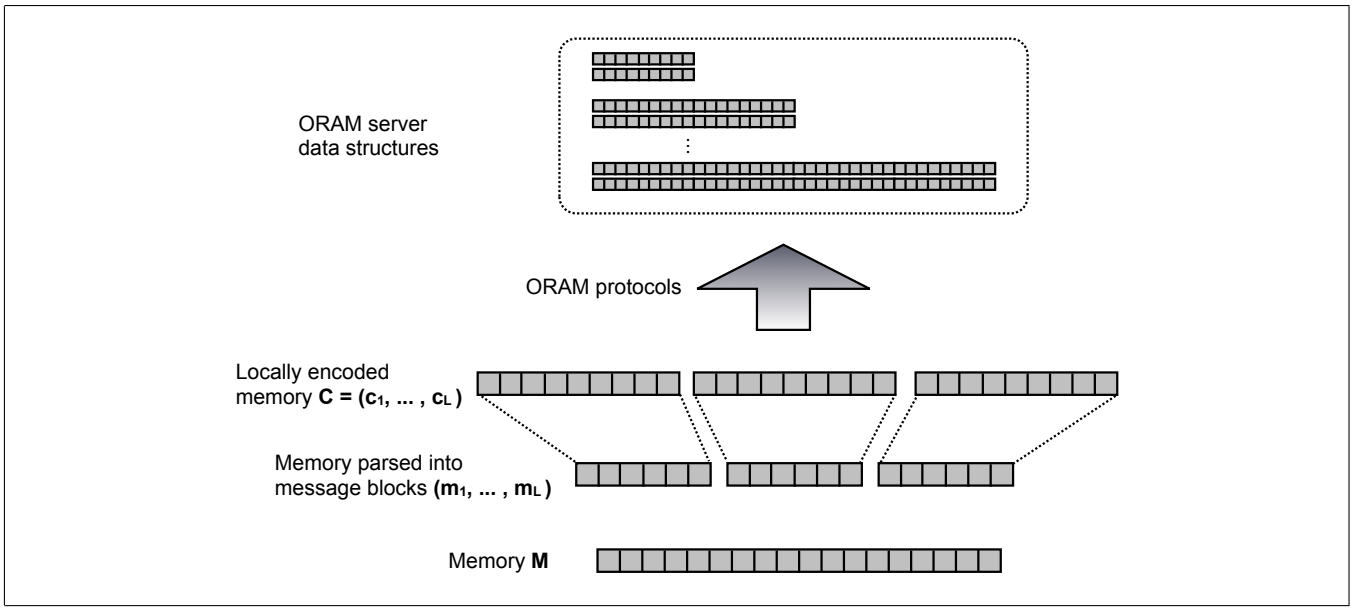

Figure 1: Our Construction

$\mathbf{M} \in \Sigma^\ell$ 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". In general, 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 $\mathcal{C}_{\mathsf{fin}}$.[5] 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. As mentioned earlier, the security of PoR is formalized via an extractor and we now give a high-level overview of how such an 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. 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.

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

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 Section 6, we look at one particularly efficient ORAM construction of Goodrich and Mitzenmacher [15] 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 present a new security notion called "next-read pattern hiding (NRPH)" for ORAM and a construction achieving this new notion, which may be of independent interest.

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 [19], building on a closely related notion called sublinear-authenticators of Naor and Rothblum [23]. 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.[6] 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 [21, 22].

[6] An alternative way to use PDPs can also achieve full security, at the cost of requiring that the server to read the entire client data during an audit, but still minimizing the communication complexity. If the data is large, say 10 GB, this is vastly impractical.

Static Data. PoR and PDP schemes for static data (without updates) have received much research attention [27, 10, 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, 8, 9] 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. [12] and Wang et al. [29] show how to achieve PDP security for *dynamic data*, supporting efficient updates. This is closely related to work on memory checking [5, 23, 11], 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. [28] 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*. See Appendix A for a very simple square-root solution in our setting.

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 andy *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). We find this abstract view of outsourcing

memory to be the simplest and most general to work with. Any higher-level data-structures and operations (e.g., allowing appends/inserts to data or implementing an entire file-system) can be easily done *on top of* this abstract notion of memory and therefore securely outsourced to the remote server.

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 $b = \texttt{accept}$.

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 $\texttt{AuthGame}_{\tilde{\mathcal{S}}}(\lambda)$ 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 $\mathtt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p)$ 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

$$\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) \stackrel{\text{def}}{=} \Pr\left[\tilde{\mathcal{S}}_{\mathsf{fin}} \stackrel{\mathsf{Audit}}{\longleftrightarrow} \mathcal{C}_{\mathsf{fin}} = \mathtt{accept}\right]$$

  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 `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 $\mathbf{M}'$.[7]
- 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 $\Pr[\mathtt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p) = 1] \leq \mathsf{negl}(\lambda)$.

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 analysis.

4 Oblivious RAM with Next-Read Pattern Hiding

An ORAM consists of protocols $(\mathbf{O}\mathsf{Init}, \mathbf{O}\mathsf{Read}, \mathbf{O}\mathsf{Write})$ 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 $\mathbf{O}\mathsf{Read}$ and $\mathbf{O}\mathsf{Write}$ to allow

[7] 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 this state at the end of the execution.

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.[8] For privacy, we define a new property called *next-read pattern hiding*. For completeness, we also define the standard notion of ORAM pattern hiding in Appendix B.

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.

[8] 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. See Section 6.4 for details.

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 $\mathbf{m} = \mathsf{Dec}(\mathbf{c})$.[9] 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.

It is easy to see that if the underlying ORAM scheme used in the above PORAM construction is secure in the standard sense of ORAM (see Appendix B) then the above construction preserves this ORAM security, hiding which locations are being accessed in each operation. 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.

Theorem 1. *Assume that* $\mathbf{O} = (\mathbf{OInit}, \mathbf{ORead}, \mathbf{OWrite})$ *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.*

[9] We can skip this step if the client already has the value $\mathbf{m}$ stored locally e.g. from prior read executions.

5.1 Proof of Theorem 1

The correctness and authenticity properties of PORAM follow immediately from those of the underlying ORAM scheme $\mathbf{O}$. The main challenge is to show that the *retrievability* property holds. As a first step, let us describe the extractor.

The Extractor. The extractor $\mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p)$ works as follows:

(1) Initialize $\mathbf{C} := (\perp)^{\ell_{\mathsf{code}}}$ where $\ell_{\mathsf{code}} = n(\ell/k)$ to be an empty vector.

(2) Keep rewinding and auditing the server by repeating the following step for $s = \max(2\ell_{\mathsf{code}}, \lambda) \cdot p$ times: Pick t distinct indices $j_1, \ldots, j_t \in [\ell_{\mathsf{code}}]$ at random and run the protocol $\mathbf{ORead}(j_1, \ldots, j_t)$ with $\tilde{\mathcal{S}}_{\mathsf{fin}}$, acting as $\mathcal{C}_{\mathsf{fin}}$ as in the audit protocol. If the protocol is accepting and $\mathcal{C}_{\mathsf{fin}}$ outputs $(v_1, \ldots, v_t)$, set $\mathbf{C}[j_1] := v_1, \ldots, \mathbf{C}[j_t] := v_t$. Rewind $\tilde{\mathcal{S}}_{\mathsf{fin}}, \mathcal{C}_{\mathsf{fin}}$ to their state prior to this execution for the next iteration.

(3) Let $\delta \stackrel{\text{def}}{=} (1 + \frac{k}{n})/2$. If the number of "filled in" values in $\mathbf{C}$ is $|\{j \in [\ell_{\mathsf{code}}] \ : \ \mathbf{C}[j] \neq \perp\}| < \delta \cdot \ell_{\mathsf{code}}$ then output fail_1. Else interpret $\mathbf{C}$ as consisting of $L = \ell/k$ consecutive codeword blocks $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L)$ with each block $\mathbf{c}_j \in \Sigma^n$. If there exists some index $j \in [L]$ such that the number of "filled" in values in codeword block $\mathbf{c}_j$ is $|\{i \in [n] \ : \ \mathbf{c}_j[i] \neq \perp\}| < k$ then output fail_2. Otherwise, apply erasure decoding to each codeword block $\mathbf{c}_j$, to recover $\mathbf{m}_j = \mathsf{Dec}(\mathbf{c}_j)$, and output $\mathbf{M} = (\mathbf{m}_1, \ldots, \mathbf{m}_L) \in \Sigma^\ell$.[10]

Proof by Contradiction. Assume that PORAM does *not* satisfy the retrievability property with the above extractor $\mathcal{E}$. Then there exists some efficient adversarial server $\tilde{\mathcal{S}}$ and some polynomials $p = p(\lambda), p' = p'(\lambda)$ such that, for infinitely many values $\lambda \in \mathbb{N}$, we have:

$$\Pr[\mathtt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p(\lambda)) = 1] > \frac{1}{p'(\lambda)} \tag{1}$$

Using the same notation as in the definition of $\mathtt{ExtGame}$, let $\tilde{\mathcal{S}}_{\mathsf{fin}}, \mathcal{C}_{\mathsf{fin}}$ be the final configurations of the malicious server $\tilde{\mathcal{S}}$ and client $\mathcal{C}$, respectively, after executing the protocol sequence P chosen by the server at the beginning of the game, and let $\mathbf{M}$ be the correct value of the memory contents resulting from P. Then (1) implies

$$\Pr\left[\begin{array}{c} \mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{p(\lambda)} \\ \wedge \mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p) \neq \mathbf{M} \end{array}\right] > \frac{1}{p'(\lambda)} \tag{2}$$

where the probability is over the coins of $\mathcal{C}, \tilde{\mathcal{S}}$ which determine the final configuration $\tilde{\mathcal{S}}_{\mathsf{fin}}, \mathcal{C}_{\mathsf{fin}}$ and the coins of the extractor $\mathcal{E}$. We now slowly refine the above inequality until we reach a contradiction, showing that the above cannot hold.

Extractor can only fail with $\{\mathsf{fail}_1, \mathsf{fail}_2\}$. Firstly, we argue that at the conclusion of $\mathtt{ExtGame}$, the extractor must either output the correct memory contents $\mathbf{M}$ or must fail with one of the error messages $\{\mathsf{fail}_1, \mathsf{fail}_2\}$. In other words, it can always detect *failure* and never outputs an incorrect value $\mathbf{M}' \neq \mathbf{M}$. This follows from the *authenticity* of the underlying ORAM scheme which guarantees that the extractor never puts any incorrect value into the array $\mathbf{C}$.

Lemma 1. *Within the execution of* $\mathtt{ExtGame}_{\tilde{\mathcal{S}},\mathcal{E}}(\lambda, p)$*, we have:*

$$\Pr[\mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p) \notin \{\mathbf{M}, \mathsf{fail}_1, \mathsf{fail}_2\}] \leq \mathsf{negl}(\lambda).$$

Proof of Lemma. The only way that the above bad event can occur is if the extractor puts an incorrect value into its array $\mathbf{C}$ which does not match encoded version of the correct memory contents $\mathbf{M}$. In particular, this

[10] The failure event fail_1 and the choice of δ is only intended to simplify the analysis of the extractor. The only real bad event from which the extractor cannot recover is fail_2.

means that one of the audit protocol executions (consisting of an $\mathbf{ORead}$ with t random locations) initiated by the extractor $\mathcal{E}$ between the malicious server $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and the client $\mathcal{C}_{\mathsf{fin}}$ causes the client to output some incorrect value which does not match correct memory contents $\mathbf{M}$, and *not* reject. By the *correctness* of the ORAM scheme, this means that the the malicious server must have deviated from honest behavior during that protocol execution, without the client rejecting. Assume the probability of this bad event happening is ρ. Since the extractor runs $s = \max(2\ell_{\mathsf{code}}, \lambda) \cdot p = \mathsf{poly}(\lambda)$ such protocol executions *with rewinding*, there is at least $\rho/s = \rho/\mathsf{poly}(\lambda)$ probability that the above bad event occurs on a single random execution of the audit with $\tilde{\mathcal{S}}_{\mathsf{fin}}$. But this means that $\tilde{\mathcal{S}}$ can be used to break the *authenticity of ORAM* with advantage $\rho/\mathsf{poly}(\lambda)$, by first running the requested protocol sequence P and then deviating from honest behavior during a subsequent $\mathbf{ORead}$ protocol without being detected. Therefore, by the authenticity of ORAM, we must have $\rho = \mathsf{negl}(\lambda)$. □

Combining the above with (2) we get:

$$\Pr\left[\begin{array}{c}\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{p(\lambda)} \\ \wedge \mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p) \in \{\mathsf{fail}_1, \mathsf{fail}_2\}\end{array}\right] > \frac{1}{p'(\lambda)} - \mathsf{negl}(\lambda) \tag{3}$$

Extractor can indeed only fail with fail_2. Next, we refine equation (3) and claim that the extractor is unlikely to reach the failure event fail_1 and therefore must fail with fail_2.

$$\Pr\left[\begin{array}{c}\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{p(\lambda)} \\ \wedge \mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p) = \mathsf{fail}_2\end{array}\right] > \frac{1}{p'(\lambda)} - \mathsf{negl}(\lambda) \tag{4}$$

To prove the above, it suffices to prove the following lemma, which intuitively says that if $\tilde{\mathcal{S}}_{\mathsf{fin}}$ has a good chance of passing an audit, then the extractor must be able to extract sufficiently many values inside $\mathbf{C}$ and hence cannot output fail_1. Remember that fail_1 occurs if the extractor does not have enough values to recover the whole memory, and fail_2 occurs if the extractor does not have enough values to recover some message block.

Lemma 2. *For any (even inefficient) machine $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and any polynomial $p = p(\lambda)$ we have:*

$$\Pr[\mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^\ell, 1^p) = \mathsf{fail}_1 \mid \mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) \geq 1/p] \leq \mathsf{negl}(\lambda).$$

Proof of Lemma. Let E be the bad event that fail_1 occurs. For each iteration $i \in [s]$ within step (2) of the execution of $\mathcal{E}$ let us define:

- X_i to be an indicator random variable that takes on the value $X_i = 1$ iff the $\mathbf{ORead}$ protocol execution in iteration i does not reject.
- G_i to be a random variable that denotes the subset $\{j \in [\ell_{\mathsf{code}}] \ : \ \mathbf{C}[j] \neq \perp\}$ of filled-in positions in the current version of $\mathbf{C}$ at the beginning of iteration i.
- Y_i to be an indicator random variable that takes on the value $Y_i = 1$ iff $|G_i| < \delta \cdot \ell_{\mathsf{code}}$ *and* all of the locations that $\mathcal{E}$ chooses to read in iteration i happen to satisfy $j_1, \ldots, j_t \in G_i$.

If $X_i = 1$ and $Y_i = 0$ in iteration i, then at least one position of $\mathbf{C}$ gets filled in so $|G_{i+1}| \geq |G_i| + 1$. Therefore the bad event E only occurs if fewer than $\delta\ell_{\mathsf{code}}$ of the X_i take on a 1 *or* at least one Y_i takes on a 1, giving us:

$$\Pr[E] \ \leq \ \Pr\left[\sum_{i=1}^{s} X_i < \delta\ell_{\mathsf{code}}\right] + \sum_{i=1}^{s} \Pr[Y_i = 1]$$

For each i, we can bound $\Pr[Y_i = 1] \leq \binom{\lfloor \delta \ell_{\mathsf{code}} \rfloor}{t} / \binom{\ell_{\mathsf{code}}}{t} \leq \delta^t$. If we define $\overline{X} = \frac{1}{s}\sum_{i=1}^{s} X_i$ we also get:

$$\begin{aligned}
\Pr\left[\sum_{i=1}^{s} X_i < \delta \ell_{\mathsf{code}}\right] &\leq \Pr\left[\overline{X} < 1/p - (1/p - \frac{\delta \ell_{\mathsf{code}}}{s})\right] \\
&\leq \exp(-2s(1/p - \delta\ell_{\mathsf{code}}/s)^2) \\
&\leq \exp(-s/p) \leq 2^{-\lambda}
\end{aligned}$$

where the second inequality follows by the Chernoff-Hoeffding bound. Therefore $\Pr[E] \leq 2^{-\lambda} + s\delta^t = \mathsf{negl}(\lambda)$ which proves the lemma. □

Use Estimated Success Probability. Instead of looking at the true success probability $\mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}})$, which we cannot efficiently compute, let us instead consider an estimated probability $\widetilde{\mathbf{Succ}}(\tilde{\mathcal{S}}_{\mathsf{fin}})$ which is computed in the context of $\texttt{ExtGame}$ by sampling $2\lambda(p(\lambda))^2$ different "audit protocol executions" between $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and $\mathcal{C}_{\mathsf{fin}}$ and seeing on which fraction of them does $\tilde{\mathcal{S}}$ succeed (while rewinding $\tilde{\mathcal{S}}_{\mathsf{fin}}$ and $\mathcal{C}_{\mathsf{fin}}$ after each one). Then, by the Chernoff-Hoeffding bound, we have:

$$\Pr\left[\widetilde{\mathbf{Succ}}(\tilde{\mathcal{S}}_{\mathsf{fin}}) \leq \frac{1}{2p(\lambda)} \;\middle|\; \mathbf{Succ}(\tilde{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{p(\lambda)}\right] \leq e^{-\lambda} = \mathsf{negl}(\lambda)$$

Combining the above with (4), we get:

$$\Pr\left[\begin{array}{c} \widetilde{\mathbf{Succ}}(\tilde{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{2p(\lambda)} \\ \wedge \mathcal{E}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) = \mathsf{fail}_2 \end{array}\right] > \frac{1}{p'(\lambda)} - \mathsf{negl}(\lambda) \tag{5}$$

Assume Passive Attacker. We now argue that we can replace the active attacker $\tilde{\mathcal{S}}$ with an efficient passive attacker $\hat{\mathcal{S}}$ who always acts as the honest server $\mathcal{S}$ in each protocol execution within the protocol sequence P and the subsequent audit, but can selectively fail by outputting $\perp$ at any point. In particular $\hat{\mathcal{S}}$ just runs a copy of $\tilde{\mathcal{S}}$ and the honest server $\mathcal{S}$ concurrently, and if $\tilde{\mathcal{S}}$ deviates from the execution of $\mathcal{S}$, it just outputs $\perp$. Then we claim that, within the context of $\texttt{ExtGame}_{\hat{\mathcal{S}},\mathcal{E}}$, we have:

$$\Pr\left[\begin{array}{c} \widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{2p(\lambda)} \\ \wedge \mathcal{E}^{\hat{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) = \mathsf{fail}_2 \end{array}\right] > \frac{1}{p'(\lambda)} - \mathsf{negl}(\lambda) \tag{6}$$

The above probability is equivalent for $\hat{\mathcal{S}}$ and $\tilde{\mathcal{S}}$, up to the latter deviating from the protocol execution without being detected by the client, either during the protocol execution of P or during one of the polynomially many executions of the next read used to compute $\widetilde{\mathbf{Succ}}(\tilde{\mathcal{S}})$ and $\mathcal{E}^{\tilde{\mathcal{S}}}$. The probability that this occurs is negligible, by *authenticity* of ORAM.

Permuted Extractor. We now aim to derive a contradiction from (6). Intuitively, if fail_2 occurs (but fail_1 does not), it means that there is some codeword block $\mathbf{c}_j$ such that $\hat{\mathcal{S}}_{\mathsf{fin}}$ is significantly likelier to fail on a next-read query for which at least one location falls inside $\mathbf{c}_j$, than it is for a "random" read query. This would imply an attack on next-read pattern hiding. We now make this intuition formal. Consider a modified "permuted extractor" $\mathcal{E}_{\mathsf{perm}}$ who works just like $\mathcal{E}$ with the exception that it permutes the locations used in the $\mathbf{ORead}$ executions during the extraction process. In particular $\mathcal{E}_{\mathsf{perm}}$ makes the following modifications to $\mathcal{E}$:

- At the beginning, $\mathcal{E}_{\mathsf{perm}}$ chooses a random permutation $\pi : [\ell_{\mathsf{code}}] \to [\ell_{\mathsf{code}}]$.
- During each of the s iterations of the audit protocol, $\mathcal{E}_{\mathsf{perm}}$ chooses t indices $j_1, \ldots, j_t \in [\ell_{\mathsf{code}}]$ at random as before, *but* it then runs $\mathbf{ORead}(\pi(j_1), \ldots, \pi(j_t))$ on the *permuted* values. If the protocol is accepting the extractor $\mathcal{E}_{\mathsf{perm}}$ still "fills-in" the *original* locations: $\mathbf{C}[j_1], \ldots, \mathbf{C}[j_t]$ (since we are only analyzing the event fail_2 we do not care about the values in these locations but only if they are filled in or not).

Now we claim that an execution of $\texttt{ExtGame}$ the permuted extractor $\mathcal{E}_{\mathsf{perm}}$ is still likely to result in the failure event fail_2. This follows from "next-read pattern hiding" which ensures that permuting the locations inside of the $\mathbf{ORead}$ executions (with rewinding) is indistinguishable.

Lemma 3. *The following holds within* $\texttt{ExtGame}_{\hat{\mathcal{S}},\mathcal{E}_{\mathsf{perm}}}$*:*

$$\Pr\left[\begin{array}{c}\widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}}) > \frac{1}{2p(\lambda)} \\ \wedge \mathcal{E}_{\mathsf{perm}}^{\hat{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) = \mathsf{fail}_2\end{array}\right] > \frac{1}{p'(\lambda)} - \mathsf{negl}(\lambda) \tag{7}$$

Proof of Lemma. Assume that (7) does not hold. Then we claim that there is an adversary $\mathcal{A}$ with non-negligible distinguishing advantage in $\texttt{NextReadGame}^{b}_{\mathcal{A}}(\lambda)$ against the ORAM.

The adversary $\mathcal{A}$ runs $\hat{\mathcal{S}}$ who chooses a PoR protocol sequence $P_1 = (op_0, \ldots, op_{q_2})$, and $\mathcal{A}$ translates this to the appropriate ORAM protocol sequence, as defined by the PORAM scheme. Then $\mathcal{A}$ chooses its own sequence $P_2 = (rop_1, \ldots, rop_{q_2})$ of sufficiently many read operations $\mathbf{ORead}(i_1, \ldots, i_t)$ where $i_1, \ldots, i_t \in [\ell_{\mathsf{code}}]$ are random distinct indices. It then passes P_1, P_2 to its challenger and gets back the transcripts of the protocol executions for stages (I) and (II) of the game.

The adversary $\mathcal{A}$ then uses the client communication from the stage (I) transcript to run $\hat{\mathcal{S}}$, getting it into some state $\hat{\mathcal{S}}_{\mathsf{fin}}$. It then uses the stage (II) transcripts, to compute $\mathcal{E}^{\hat{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) \stackrel{?}{=} \mathsf{fail}_2$ and to estimate $\widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}})$, without knowing the client state $\mathcal{C}_{\mathsf{fin}}$. It does so just by checking on which executions does $\hat{\mathcal{S}}_{\mathsf{fin}}$ abort with $\perp$ and which it runs to completion (here we use that $\hat{\mathcal{S}}$ is semi-honest and never deviates beyond outputting $\perp$). Lastly $\mathcal{A}$ outputs 1 iff the emulated extraction $\mathcal{E}^{\hat{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) = \mathsf{fail}_2$ *and* $\widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}}) \geq \frac{1}{2p(\lambda)}$.

Let b be the challenger's bit in the "next-read pattern hiding game". If $b = 0$ (not permuted) then $\mathcal{A}$ perfectly emulates the distribution of $\mathcal{E}^{\hat{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) \stackrel{?}{=} \mathsf{fail}_2$ and the estimation of $\widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}})$ so, by inequality (6):

$$\Pr[\texttt{NextReadGame}^{0}_{\mathcal{A}}(\lambda) = 1] \geq 1/p'(\lambda) - \mathsf{negl}(\lambda).$$

If $b = 1$ (permuted) then $\mathcal{A}$ perfectly emulates the distribution of the permuted extractor $\mathcal{E}^{\hat{\mathcal{S}}_{\mathsf{fin}}}_{\mathsf{perm}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) \stackrel{?}{=} \mathsf{fail}_2$ and the estimation of $\widetilde{\mathbf{Succ}}(\hat{\mathcal{S}}_{\mathsf{fin}})$ since, for the latter, it does not matter whether random reads are permuted or not. Therefore, since (7) is false by assumption, we have

$$\Pr[\texttt{NextReadGame}^{1}_{\mathcal{A}}(\lambda) = 1] \leq 1/p'(\lambda) - \mu(\lambda)$$

where $\mu(\lambda)$ is non-negligible. This means that the distinguishing advantage of the passive attacker $\mathcal{A}$ is non-negligible in the next-read pattern hiding game, which proves the lemma. □

Contradiction. Finally, we present an information-theoretic argument showing that, when using the permuted extractor $\mathcal{E}_{\mathsf{perm}}$, the probability of fail_2 is negligible over the choice of the permutation π. Together with inequality (7), this gives us a contradiction.

Lemma 4. *For any (possibly unbounded)* $\tilde{\mathcal{S}}$*, we have*

$$\Pr\left[\begin{array}{c}\mathbf{Succ}(\tilde{\hat{\mathcal{S}}}_{\mathsf{fin}}) > \frac{1}{2p(\lambda)} \\ \wedge \mathcal{E}_{\mathsf{perm}}^{\tilde{\mathcal{S}}_{\mathsf{fin}}}(\mathcal{C}_{\mathsf{fin}}, 1^{\ell}, 1^{p}) = \mathsf{fail}_2\end{array}\right] = \mathsf{negl}(\lambda).$$

Proof of Lemma. Firstly, note that an equivalent way of thinking about $\mathcal{E}_{\mathsf{perm}}$ is to have it issue random (un-permuted) read queries just like $\mathcal{E}$ to recover $\mathbf{C}$, but then permute the locations of $\mathbf{C}$ via some permutation $\pi : [\ell_{\mathsf{code}}] \to [\ell_{\mathsf{code}}]$ before testing for the event fail_2. This is simply because we have the distributional equivalence $(\pi(\texttt{random}), \texttt{random}) \equiv (\texttt{random}, \pi(\texttt{random}))$, where $\texttt{random}$ represents the randomly chosen locations for the audit and π is a random permutation. Now, with this interpretation of $\mathcal{E}_{\mathsf{perm}}$, the event fail_2 occurs only if (I) the un-permuted $\mathbf{C}$ contains more than δ fraction of locations with filled in (non $\perp$) values

so that fail_1 does not occur, and (II) the permuted version $(\mathbf{c}_1, \ldots, \mathbf{c}_L) = \mathbf{C}[\pi(1)], \ldots, \mathbf{C}[\pi(\ell_{\mathsf{code}})]$ contains some codeword block $\mathbf{c}_j$ with fewer than k/n fraction of filled in (non $\perp$) values.

We now show that, conditioned on (I) the probability of (II) is negligible over the random choice of π. Fix some index $j \in [L]$ and let us bound the probability that $\mathbf{c}_j$ is the *"bad"* codeword block with fewer than k filled in values. Let $X_1, X_2, \ldots, X_n$ be random variables where X_i is 1 if $\mathbf{c}_j[i] \neq \perp$ and 0 otherwise. Let $\overline{X} \stackrel{\text{def}}{=} \frac{1}{n}\sum_{i=1}^{n} X_i$. Then, over the randomness of π, the random variables $X_1, \ldots, X_n$ are sampled *without replacement* from a population of ℓ_{code} values (location in $\mathbf{C}$), at least $\delta\ell_{\mathsf{code}}$ of which are 1 ($\neq \perp$) and the rest are 0 ($= \perp$). Therefore, by Hoeffding's bound for sampling from finite populations without replacement (See section 6 of [18]), we have:

$$\begin{aligned} \Pr[\mathbf{c}_j \text{ is bad }] &= \Pr[\overline{X} < k/n] = \Pr[\overline{X} < \delta - (\delta - k/n)] \\ &\leq \exp(-2n(\delta - k/n)^2) = \mathsf{negl}(\lambda) \end{aligned}$$

By taking a union-bound over all codeword blocks $\mathbf{c}_j$, we can bound the probability in equation (7) by $\sum_{j=1}^{\ell/k} \Pr[\mathbf{c}_j \text{ is bad }] \leq \mathsf{negl}(\lambda)$.

We have already shown that fail_1 only occurs with negligible probability. We now showed that fail_2 for the permuted extractor also occurs with negligible probability, while the adversary succeeds with non-negligible probability. □

Combining the above lemma with equation (7), we get a contradiction, showing that the assumption in equation (1) cannot hold. Thus, as long as the adversary succeeds with non-negligible probability during audits, the extractor will also succeed with non-negligible probability in extracting the whole memory contents correctly.

6 ORAM Instantiation

The notion of ORAM was introduced by Goldreich and Ostrovsky [13], who also introduced the so-called *hierarchical scheme* having the structure seen in Figures 1 and 6.2. Since then several improvements to the hierarchical scheme have been given, including improved rebuild phases and the use of advanced hashing techniques [30, 26, 15].

We examine a particular ORAM scheme of Goodrich and Mitzenmacher [15] 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 other ORAM schemes from the literature also seemingly satisfy next-read pattern hiding, and we only focus on the above example for concreteness. However, in Appendix C 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. Therefore, next-read pattern hiding is a meaningful property beyond standard ORAM security and must be examined carefully.

Overview. We note that ORAM schemes are generally not described as protocols, but simply as a data structure in which the client's encrypted data is stored on the server. Each time that a client wants to perform a read or write to some address i of her memory, this operation is translated into a series of read/write operations on this data structure inside the server's storage. In other words, the (honest) server does not perform any computation at all during these 'protocols', but simply allows the client to access arbitrary locations inside this data structure.

Most ORAM schemes, including the one we will use below, follow a *hierarchical structure*. They maintain several *levels* of hash tables on the server, each holding encrypted address-value pairs, with lower tables having higher capacity. The tables are managed so that the most recently accessed data is kept in the top tables and the least recently used data is kept in the bottom tables. Over time, infrequently accessed data is moved into lower tables (obliviously).

To write a value to some address, just insert the encrypted address-value pair in the top table. To read the value at some address, one hashes the address and checks the appropriate position in the top table. If it *is* found in that table, then one hides this fact by sequentially checking random positions in the remaining tables. If it *is not* found in the top table, then one hashes the address again and checks the second level table, continuing down the list until it is found, and then accessing random positions in the remaining tables. Once all of the tables have been accessed, the found data is written into the top table. To prevent tables from overflowing (due to too many item insertions), there are additional periodic *rebuild phases* which obliviously moves data from the smaller tables to larger tables further down.

Security Intuition. The reason that we always write found data into the top table after any read, is to protect the privacy of repeatedly reading the same address, and ensuring that this looks the same as reading various different addresses. In particular, reading the same address twice will not need to access the same locations on the server, since after the first read, the data will already reside in the top table, and the random locations will be read at lower tables.

At any point in time, after the server observes many read/write executions, any subsequent read operation just accesses completely random locations in each table, from the point of view of the server. This is the main observation needed to argue standard pattern hiding. For next-read pattern hiding, we notice that we can extend the above to any set of q distinct executions of a subsequent read operation with distinct addresses (each execution starting in the same client/server state). In particular, each of the q operations just accesses completely random locations in each table, independently of the other operations, from the point of view of the server.

One subtlety comes up when the addresses are not completely *distinct* from each other, as is the case in our definition where each address can appear in multiple separate multi-read operations. The issue is that doing a read operation on the same address twice with rewinding will reveal the level at which the data for that address is stored, thus revealing some information about which address is being accessed. One can simply observe at which level do the accesses begin to differ in the two executions. We fix this issue by modifying a scheme so that, instead of accessing freshly chosen random positions in lower tables once the correct value is found, we instead access *pseudorandom positions that are determined by the address being read and the operation count.* That way, any two executions which read the same address *starting from the same client state* are *exactly* the same and do not reveal anything beyond this. Note that, without state rewinds, this still provides regular pattern hiding.

6.1 Technical Tools

Our construction uses the standard notion of a *pseudorandom-function* (PRF) where $F(K, x)$ denote the evaluation of the PRF F on input x with key K. We also rely on a *symmetric-key encryption* scheme secure against *chosen-plaintext attacks*, and let $\mathsf{Enc}(K, \cdot), \mathsf{Dec}(K, \cdot)$ denote the encryption/decryption algorithms with key K.

Encrypted cuckoo table. An encrypted cuckoo table [25, 20] consists of three arrays (T_1, T_2, S) that hold ciphertexts of some fixed length. The arrays T_1 and T_2 are both of size m and serve as *cuckoo-hash tables* while S is an array of size s and serves as an auxiliary *stash*. The data structure uses two hash functions $h_1, h_2 : [\ell] \to [m]$. Initially, all entries of the arrays are populated with independent encryptions of a special symbol $\perp$. To retrieve a ciphertext associated with an address i, one decrypts all of the ciphertexts in S, as well as the ciphertexts at $T_1[h_1[i]]$ and $T_2[h_2[i]]$ (thus at most $s + 2$ decryptions are performed). If any of these ciphertexts decrypts to a value of the form (i, v), then v is the returned output. To insert an address-value pair (i, v), encrypt it and write the ciphertext ct to position $T_1[h_1(i)]$, retrieving whatever ciphertext ct_1 was there before. If the original ciphertext ct_1 decrypts to $\perp$, then stop. Otherwise, if ct_1 decrypts to a pair (j, w), then re-encrypt the pair and write the resulting ciphertext to $T_2[h_2(j)]$, again retrieving whatever ciphertext ct_2 was there before. If ct_2 decrypts to $\perp$, then stop, and otherwise continue

this process iteratively with ciphertexts $\mathsf{ct}_3, \mathsf{ct}_4, \ldots$. If this process continues for $t = c \log n$ steps, then 'give up' and just put the last evicted ciphertext ct_t into the first available spot in the stash S. If S is full, then the data structure fails.

We will use the following result sketched in [15]: If $m = (1+\varepsilon)n$ for some constant $\varepsilon > 0$, and h_1, h_2 are random functions, then after n items are inserted, the probability that S has k or more items written into it is $O(1/n^{k+2})$. Thus, if S has at least λ slots, then the probability of a failure after n insertions is negligible in λ.

Oblivious table rebuilds. We will assume an oblivious protocol for the following task. At the start of the protocol, the server holds encrypted cuckoo hash tables $C_1, \ldots, C_r$. The client has two hash functions h_1, h_2. After the oblivious interaction, the server holds a new cuckoo hash table C'_r that results from decrypting the data in $C_1, \ldots, C_r$, deleting data for duplicated locations with preference given to the copy of the data in the lowest index table, encrypting each index-value pair again, and then inserting the ciphertexts into C'_r using h_1, h_2.

Implementing this task efficiently and obliviously is an intricate task. See [15] and [26] for different methods, which adapt the usage of oblivious sorting first introduced in [13].

6.2 ORAM Scheme

We can now describe the scheme of Goodrich et al, with our modifications for next-read pattern hiding. As ingredients, this scheme will use a PRF F and an encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$. A visualization of the server's data structures is given in Figure 6.2.

$\mathbf{O}\mathsf{Init}(1^\lambda, 1^w, \ell)$: Let L the smallest integer such that $2^L > \ell$. The client chooses $2L$ random keys $K_{1,1}$, $K_{1,2}, \ldots, K_{L,1}, K_{L,2}$ and $2L$ additional random keys $R_{1,1}, R_{1,2}, \ldots, R_{L,1}, R_{L,2}$ to be used for pseudo-random functions, and initializes a counter ctr to 0. It also selects an encryption key for the IND-CPA secure scheme. It instructs the server to allocate the following data structures:

- An empty array A_0 that will change size as it is used.
- L empty cuckoo hash tables $C_1, \ldots, C_L$ where the parameters in C_j are adjusted to hold 2^j data items with a negligible (in λ) probability of overflow when used with random hash functions.

The client state consists of all of the keys $(K_{j,0}, K_{j,1})_{j\in[L]}$, $(R_{j,0}, R_{j,1})_{j\in[L]}$, the encryption key, and ctr.

$\mathbf{O}\mathsf{Read}(i_1, \ldots, i_t)$: The client starts by initializing an array found of t flags to false. For each index i_j to be read, the client does the following. For each level $k = 1, \ldots, L$, the client executes

- Let $C_k = (T_1^{(k)}, T_2^{(k)}, S^{(k)})$
- If $\mathsf{found}[j] = \mathsf{false}$, read and decrypt all of $S^{(k)}$, $T_1^{(k)}[F(K_{k,1}, i_j)]$ and $T_2^{(k)}[F(K_{k,2}, i_j)]$. If the data is in any of these slots, set $\mathsf{found}[j]$ to true and remember the value as v_j.
- Else, if $\mathsf{found}[j] = \mathsf{true}$, then instead read all of $S^{(k)}$, $T_1^{(k)}[F(R_{k,1}, i_j \| \mathsf{ctr})]$ and $T_2^{(k)}[F(R_{k,2}, i_j \| \mathsf{ctr})]$ and ignore the results. Note that the counter value is used to create random reads when the state is not reset, while providing the same random values if the state is reset.

Finally, it encrypts and appends (i_j, v_j) to the end of A_0 and continues to the next index i_{j+1}. We note that above, when accessing a table using the output of F, we are interpreting the bitstring output by F as a random index from the appropriate range.

After all the indices have been read and written to A_0, the client initiates a *rebuild phase*, the description of which we defer for now.

$\mathbf{O}\mathsf{Write}(i_1, \ldots, i_t; v_1, \ldots, v_t)$: The client encrypts and writes (i_j, v_j) into A_0 for each j, then initiates a rebuild phase, described below.

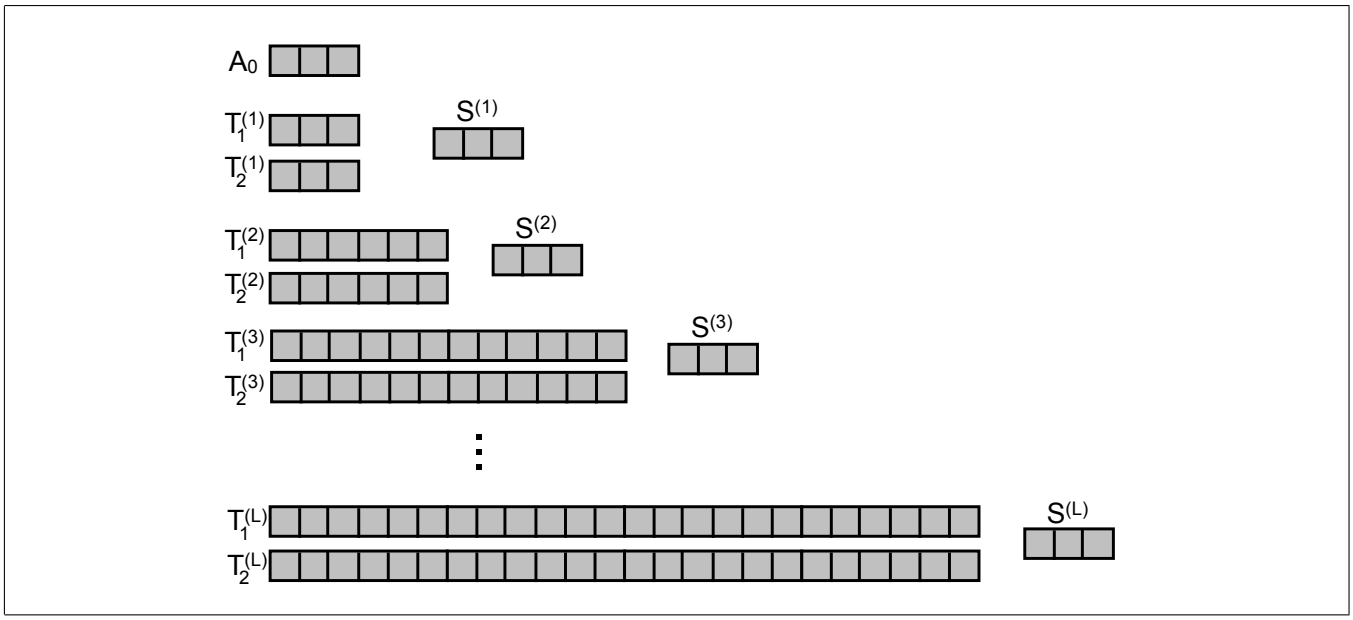

Figure 2: Server data structures in the ORAM instantiation.

Rebuild phase. We complete the scheme description by describing a rebuild phase, which works as follows. The client repeats the following process until A_0 is empty:

- Increment ctr.
- Remove and decrypt an item from A_0, calling the result (j, v).
- Let $r \geq 0$ be the largest integer such that 2^r divides (ctr mod 2^L).
- Select new keys $K_{r,1}, K_{r,2}$ and use the functions $F(K_{r,1}, \cdot)$ and $F(K_{r,2}, \cdot)$ as h_1 and h_2 to obliviously build a new cuckoo table C'_r holding the removed item (j, v) and all of the data items in $C_1, \ldots, C_{r-1}$, freshly recrypted and with duplicates removed.
- Then, for $j = 1$ to $r - 1$, set $K_{j,1}, K_{j,2}$ to fresh random keys and set the cuckoo tables $C_1, \ldots, C_r$ to be new, empty tables and C_r to be C'_r.

Note that the remaining tables $C_{r+1}, \ldots, C_L$ are not touched.

We can implement the rebuild phase using the any of the protocols (with small variations) from [15, 16]. The most efficient gives an amortized overhead of $\log \ell$ operations for all rebuilds, assuming that the client can *temporarily* locally store ℓ^δ memory slots during the protocol (but the client does need to store them between executions of the protocol). If we only allow the client to store a constant number of slots at any one time, then the we incur an overhead of $\log^2 \ell$. In either case the worst-case overhead is $O(\ell)$. Using the deamortization techniques from [16, 24], we can achieve worst-case complexity of $\log^2 \ell$, at the cost of doubling the server storage. This technique was analyzed in the original ORAM security setting, but it is not hard to extend our proof to show that it preserves next-read pattern hiding as well.

6.3 Next-Read Pattern Hiding

Theorem 2. *Assuming that F is a secure PRF, and the underlying encryption scheme is chosen-plaintext secure, then the scheme* **O** *described above is next-read pattern hiding.*

Proof. We show that for any efficient adversary $\mathcal{A}$, the probabilities that $\mathcal{A}$ outputs 1 when playing either $\mathtt{NextReadGame}^0_{\mathcal{A}}$ or $\mathtt{NextReadGame}^1_{\mathcal{A}}$ differs by only a negligible amount. In these games, the adversary $\mathcal{A}$ provides two tuples of operations $P_1 = (op_1, \ldots, op_{q_1})$ and $P_2 = (rop_1, \ldots, rop_{q_2})$, the latter being all multi-reads, and a permutation π on $[\ell]$. Then in $\mathtt{NextReadGame}^0_{\mathcal{A}}$, $\mathcal{A}$ is given the transcript of an honest client and server executing P_1, as well as the transcript of executing the multi-reads in P_2 with rewinds after each

operation, while in $\texttt{NextReadGame}^1_{\mathcal{A}}$ it is given the same transcript except that second part is generated by first permuting the addresses in P_2 according to π.

We need to argue that these inputs are computationally indistinguishable. For our analysis below, we assume that a rebuild phase never fails, as this event happens with negligible probability in λ, as discussed before. We start by modifying the execution of the games in two ways that are shown to be undetectable by $\mathcal{A}$. The first change will show that all of the accesses into tables appear to the adversary to be generated by random functions, and the second change will show that the ciphertexts do not reveal any usable information for the adversary.

First, whenever keys $K_{j,1}, K_{j,2}$ are chosen and used with the function F, we use random functions $g_{j,1}, g_{j,2}$ in place of $F(K_{j,1}, \cdot)$ and $F(K_{j,2}, \cdot)$.[11] We do the same for the $R_{j,1}, R_{j,2}$ keys, calling the random functions $r_{j,1}$ and $r_{j,2}$. This change only changes the behavior of $\mathcal{A}$ by a negligible amount, as otherwise we could build a distinguisher to contradict the PRF security of F via a standard hybrid argument over all of the keys chosen during the game.

The second change we make is that all of the ciphertexts in the transcript are replaced with independent encryptions of equal-length strings of zeros. We claim that this only affects the output distribution of $\mathcal{A}$ by a negligible amount, as otherwise we could build an adversary to contradict the IND-CPA security of the underlying encryption scheme via a standard reduction. Here it is crucial that, after each rewind, the client chooses new randomness for the encryption scheme.

We now complete the proof by showing that the distribution of the transcripts given to $\mathcal{A}$ is identical in the modified versions of $\texttt{NextReadGame}^0_{\mathcal{A}}$ and $\texttt{NextReadGame}^1_{\mathcal{A}}$. To see why this is true, let us examine what is in one of the game transcripts given to $\mathcal{A}$. The transcript for the execution of P_1 consists of ORead and OWrite transcripts, which are accesses to indices in the cuckoo hash tables, ciphertext writes into A_0, and rebuild phases. Finally the execution of P_2 (either permuted by π or not) with rewinds generates a transcript that consists of several accesses to the cuckoo hash tables, each followed by writes to A_0 and a rebuild phase.

By construction of the protocol, in the modified game the only part of the transcript that depends on the addresses in P_2 are the reads into $T_1^{(k)}$ and $T_2^{(k)}$ for each k. All other parts of the transcript are oblivious scans of the $S^{(k)}$ arrays and oblivious table rebuilds which do not depend on the addresses (recall the ciphertexts in these transcripts are encryptions of zeros). Thus we focus on the indices read in each $T_1^{(k)}$ and $T_2^{(k)}$, and need to show that, in the modified games, the distribution of these indices does not depend on the addresses in P_2.

The key observation is that, after the execution of P_1, the state of the client is such that each address i will induce a uniformly random sequence of indices in the tables that is independent of the indices read for any other address and independent of the transcript for P_1. If the data is in the cuckoo table at level k, then the indices will be

$$(g_{j,1}(i))_{j=1}^{k} \text{ and } (r_{j,1}(i \| \mathsf{ctr}))_{j=k+1}^{L}.$$

Thus each i induces a random sequence, and each address will generate an independent sequence. We claim moreover that the sequence for i is independent of the transcript for P_1. This follows from the construction: For the indices derived from $r_{j,1}$ and $r_{j,2}$, the transcript for P_1 would have always used a lower value for ctr. For the indices derived from $g_{j,1}$ and $g_{j,2}$, we have that the execution of P_1 would not have evaluated those functions on input i: If i was read during P_1, then i would have been written to A_0 and a rebuild phase would have chosen new random functions for $g_{j,1}$ and $g_{j,2}$ before the address/value pair i was placed in the j-th level table again.

With this observation we can complete the proof. When the modified games are generating the transcript for the multi-read operations in P_2, each individual read for an index i induces an random sequence of table

[11] As usual, instead of actually picking and using a random function, which is an exponential task, we create random numbers whenever necessary, and remember them. Since there will be only polynomially-many interactions, this only requires polynomial time and space.

reads amongst its other oblivious operations. But since each i induces a completely random sequence and permuting the addresses will only permute the random sequences associated with the addresses, the distribution of the transcript is unchanged. Thus no adversary can distinguish these games, which means that no adversary could distinguish $\texttt{NextReadGame}^0_{\mathcal{A}}$ and $\texttt{NextReadGame}^1_{\mathcal{A}}$, as required. □

6.4 Authenticity, Extensions & Optimizations

Authenticity. To achieve authenticity we sketch how to employ the technique introduced in [13]. A straightforward attempt is to tag every ciphertext stored on the server along with its location on the server using a message authentication code (MAC). But this fails because the sever can "roll back" changes to the data by replacing ciphertexts with previously stored ones at the same location. We can generically fix this by using the techniques of *memory checking* [5, 23, 11] at some additional logarithmic overhead. However, it also turns out that authenticity can also be added at almost no cost to several specific constructions, as we describe below.

Goldreich and Ostrovsky showed that any ORAM protocol supporting *time labeled simulation* (TLS) can be modified to achieve authenticity without much additional complexity. We say that an ORAM protocol *supports TLS* if there exists an efficient algorithm Q such that, after the j-th message is sent to the server, for each index x on the server memory, the number of times x has been written to is equal to $Q(j, x)$.[12] Overall, one implements the above tagging strategy, and also includes $Q(j, x)$ with the data being tagged, and when reading one recomputes $Q(j, x)$ to verify the tag.

Our scheme can be shown to support TLS in a manner very similar to the original hierarchical scheme [13]. The essential observation, also used there, is that the table indices are only written to during a rebuild phase, so by tracking the number of executed rebuild phases we can compute how many times each index of the table was written to.

Extensions and optimizations. The scheme above is presented in a simplified form that can be made more efficient in several ways while maintaining security.

- The keys in the client state can be derived from a single key by appropriately using the PRF. This shrinks the client state to a single key and counter.
- The initial table C_1 can be made larger to reduce the number of rebuild phases (although this does not affect the asymptotic complexity).
- We can collapse the individual oblivious table rebuilds into one larger rebuild.
- It was shown in [17] that all of the L cuckoo hash tables can share a single $O(\lambda)$-size stash S while still maintaining a negligible chance of table failure.
- Instead of doing table rebuilds all at once, we can employ a technique that allows for them to be done incrementally, allowing us to achieve worst-case rather than amortized complexity guarantees [16, 24]. These techniques come at the cost of doubling the server storage.
- The accesses to cuckoo tables on each level during a multi-read can be done in parallel, which reduces the round complexity of that part to be independent of t, the number of addresses being read.

We can also extend this scheme to support a dynamically changing memory size. This is done by simply allocating different sized tables during a rebuild that eliminate the lower larger tables or add new ones of the appropriate size. This modification will achieve next-read pattern hiding security, but it will not be standard pattern-hiding secure, as it leaks some information about the number of memory slots in use. One can formalize this, however, in a pattern-hiding model where any two sequences with equal memory usage are required to be indistinguishable.

[12] Here we mean actual writes on the server, and not OWrite executions.

Efficiency. In this scheme the client stores the counter and the keys, which can be derived from a single key using the PRF. The server stores $\log \ell$ tables, where the j-th table requires $2^j + \lambda$ memory slots, which sums to $O(\ell + \lambda \cdot \log \ell)$. Using the optimization above, we only need a single stash, reducing the sum to $O(\ell + \lambda)$. When executing **OR**ead, each index read requires accessing two slots plus the λ stash slots in each of the $\log \ell$ tables, followed by a rebuild. **OW**rite is simply one write followed by a rebuild phase. The table below summarizes the efficiency measures of the scheme.

Client Storage	$O(1)$
Server Storage	$O(\ell + \lambda)$
Read Complexity	$O(\lambda \cdot \log \ell) + \mathrm{RP}$
Write Complexity	$O(1) + \mathrm{RP}$

Table 1: Efficiency of ORAM scheme above. "RP" denotes the aggregate cost of the rebuild phases, which is $O(\log \ell)$, or $O(\log^2 \ell)$ in the worst-case, per our discussion above.

7 Efficiency

We now look at the efficiency of our PORAM construction, when instantiated with the ORAM scheme from section 6 (we assume the rebuild phases are implemented via the Goodrich-Mitzemacher algorithm [15] with the worst-case complexity optimization [16, 24].) Since our PORAM scheme preserves (standard) ORAM security, we analyze its efficiency in two ways. Firstly, we look at the overhead of PORAM scheme on top of just storing the data inside of the ORAM without attempting to achieve any PoR security (e.g., not using any error-correcting code etc.). Secondly, we look at the overall efficiency of PORAM. Third, we compare it with dynamic pdp [12, 29] 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** [12]
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 will achieve the same efficiency measures as above, but with ℓ taken to be amount of memory currently used by the memory access sequence. This is in contrast to the usual ORAM setting where ℓ is taken to be a (perhaps large) upper bound on the total amount of memory that will ever be used.

Acknowledgements

Alptekin Küpçü would like to acknowledge the support of TÜBİTAK, The Scientific and Technological Research Council of Turkey. David Cash and Daniel Wichs are sponsored by DARPA under agreement number FA8750-11-C-0096. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views expressed are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited).

References

[1] G. Ateniese, R. C. Burns, R. Curtmola, J. Herring, L. Kissner, Z. N. J. Peterson, and D. Song. Provable data possession at untrusted stores. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM CCS 07*, pages 598–609. ACM Press, Oct. 2007.

[2] G. Ateniese, S. Kamara, and J. Katz. Proofs of storage from homomorphic identification protocols. In M. Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 319–333. Springer, Dec. 2009.

[3] G. Ateniese, R. D. Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. Cryptology ePrint Archive, Report 2008/114, 2008. `http://eprint.iacr.org/`.

[4] M. Bellare and O. Goldreich. On defining proofs of knowledge. In E. F. Brickell, editor, *CRYPTO'92*, volume 740 of *LNCS*, pages 390–420. Springer, Aug. 1993.

[5] M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2/3):225–244, 1994.

[6] K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM CCS 09*, pages 187–198. ACM Press, Nov. 2009.

[7] K. D. Bowers, A. Juels, and A. Oprea. Proofs of retrievability: theory and implementation. In R. Sion and D. Song, editors, *CCSW*, pages 43–54. ACM, 2009.

[8] B. Chen, R. Curtmola, G. Ateniese, and R. C. Burns. Remote data checking for network coding-based distributed storage systems. In A. Perrig and R. Sion, editors, *CCSW*, pages 31–42. ACM, 2010.

[9] R. Curtmola, O. Khan, R. Burns, and G. Ateniese. Mr-pdp: Multiple-replica provable data possession. In *ICDCS*, 2008.

[10] Y. Dodis, S. P. Vadhan, and D. Wichs. Proofs of retrievability via hardness amplification. In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 109–127. Springer, Mar. 2009.

[11] C. Dwork, M. Naor, G. N. Rothblum, and V. Vaikuntanathan. How efficient can memory checking be? In O. Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Mar. 2009.

[12] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *ACM CCS 09*, pages 213–222. ACM Press, Nov. 2009.

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

[14] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[15] M. T. Goodrich and M. Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In L. Aceto, M. Henzinger, and J. Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, July 2011.

[16] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *CCSW*, pages 95–100, 2011.

[17] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *SODA*, pages 157–167, 2012.

[18] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):13–30, 1963.

[19] A. Juels and B. S. Kaliski Jr. Pors: proofs of retrievability for large files. In P. Ning, S. D. C. di Vimercati, and P. F. Syverson, editors, *ACM CCS 07*, pages 584–597. ACM Press, Oct. 2007.

[20] A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009.

[21] A. Küpçü. *Efficient Cryptography for the Next Generation Secure Cloud*. PhD thesis, Brown University, 2010.

[22] A. Küpçü. *Efficient Cryptography for the Next Generation Secure Cloud: Protocols, Proofs, and Implementation*. Lambert Academic Publishing, 2010.

[23] M. Naor and G. N. Rothblum. The complexity of online memory checking. In *46th FOCS*, pages 573–584. IEEE Computer Society Press, Oct. 2005.

[24] R. Ostrovsky and V. Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[25] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[26] B. Pinkas and T. Reinman. Oblivious RAM revisited. In T. Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 502–519. Springer, Aug. 2010.

[27] H. Shacham and B. Waters. Compact proofs of retrievability. In J. Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 90–107. Springer, Dec. 2008.

[28] E. Stefanov, M. van Dijk, A. Oprea, and A. Juels. Iris: A scalable cloud file system with efficient integrity checks. Cryptology ePrint Archive, Report 2011/585, 2011. `http://eprint.iacr.org/`.

[29] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou. Enabling public verifiability and data dynamics for storage security in cloud computing. In M. Backes and P. Ning, editors, *ESORICS 2009*, volume 5789 of *LNCS*, pages 355–370. Springer, Sept. 2009.

[30] P. Williams, R. Sion, and B. Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In P. Ning, P. F. Syverson, and S. Jha, editors, *ACM CCS 08*, pages 139–148. ACM Press, Oct. 2008.

A Simple Dynamic PoR with Square-Root Complexity

We sketch a very simple construction of dynamic PoR that achieves *sub-linear* complexity in its read, write and audit operations. Although the scheme is asymptotically significantly worst then our PORAM solution as described in the main body, it is significantly simpler and may be of interest for some practical parameter settings.

The construction starts with the first dynamic PoR proposal from the introduction. To store a memory $\mathbf{M} \in \Sigma^\ell$ on the server, the client divides it into $L = \sqrt{\ell}$ consecutive message blocks $(\mathbf{m}_1, \ldots, \mathbf{m}_L)$, each containing $L = \sqrt{\ell}$ symbols. The client then encodes each of the message blocks $\mathbf{m}_i$ using an $(n = 2L, k = L, d = L + 1)$-erasure code (e.g., Reed-Solomon tolerating L erasures), to form a codeword block $\mathbf{c}_i$, and concatenates the codeword blocks to form a string $\mathbf{C} = (\mathbf{c}_1, \ldots, \mathbf{c}_L) \in \Sigma^{2\ell}$ which it then stores on the server. We can assume the code is systematic so that the message block $\mathbf{m}_i$ resides in the first L symbols of the

corresponding codeword block $\mathbf{c}_i$. In addition, the client initializes a *memory checking scheme* [5, 23, 11], which it uses to authenticate each of the 2ℓ codeword symbols within $\mathbf{C}$.

To read a location $j \in [\ell]$ of memory, the client computes the index $i \in [L]$ of the message block $\mathbf{m}_i$ containing that location, and downloads the appropriate symbol of the codeword block $\mathbf{c}_i$ which contains the value $\mathbf{M}[j]$ (here we use that the code is systematic), which it checks for authenticity via the memory checking scheme. To write to a location $j \in [\ell]$ the client downloads the entire corresponding codeword block $\mathbf{c}_i$ (checking for authenticity) decodes $\mathbf{m}_i$, changes the appropriate location to get an updated block $\mathbf{m}'_i$ and finally re-encodes it to get $\mathbf{c}'_i$ which it then writes to the server, updating the appropriate authentication information within the memory checking scheme. The audit protocol selects $t = \lambda$ (security parameter) random positions within *every* codeword block $\mathbf{c}_i$ and checks them for authenticity via the memory checking scheme.

The read and write protocols of this scheme each execute the memory checking read protocol to read and write 1 and $\sqrt{\ell}$ symbols respectively. The audit protocol reads and checks $\lambda\sqrt{\ell}$ symbols. Assuming an efficient (poly-logarithmic) memory checking protocol, this means actual complexity of these protocols incurs another $O(\log \ell)$ factor and another constant factor increase in server storage. Therefore the complexity of the reads, writes, and audit is $O(1), O(\sqrt{\ell}), O(\sqrt{\ell})$ respectively, ignoring factors that depend on the security parameter or are polylogarithmic in ℓ.

Note that the above scheme actually gives us a natural trade-off between the complexity of the writes and the audit protocol. In particular, for any $\delta > 0$, we can set the message block size to $L_1 = \ell^\delta$ symbols, so that the client memory $\mathbf{M}$ now consists of $L_2 = \ell^{1-\delta}$ such blocks. In this case, the complexity of reads, writes, and audits becomes $O(1), O(\ell^\delta), O(\ell^{1-\delta})$ respectively.

B Standard Pattern Hiding for ORAM

We recall an equivalent definition to the one introduced by Goldreich and Ostrovsky [13]. Informally, standard pattern hiding says that an (arbitrarily malicious and efficient) adversary cannot detect which sequence of instructions a client is executing via the ORAM protocols.

Formally, for a bit b and an adversary $\mathcal{A}$, we define the game $\texttt{ORAMGame}^b_{\mathcal{A}}(\lambda)$ as follows:

- The attacker $\mathcal{A}(1^\lambda)$ outputs two equal-length ORAM protocol sequences $Q_0 = (op_0, \ldots, op_q), Q_1 = (op'_0, \ldots, op'_q)$. We require that for each index j, the operations op_j and op'_j only differ in the location they access and the values the are writing, but otherwise correspond to the same operation (read or write).
- The challenger initializes an honest client $\mathcal{C}$ and server $\mathcal{S}$, and sequentially executes the operations in Q_b, between $\mathcal{C}$ and $\mathcal{S}$.
- Finally, $\mathcal{A}$ is given the complete transcript of all the protocol executions, and he outputs a bit $\tilde{b}$, which is the output of the game.

We say that an ORAM protocol is pattern hiding if for all efficient adversaries $\mathcal{A}$ we have:

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

Sometimes we also want to achieve a stronger notion of security where we also wish to hide whether each operation is a read or a write. This can be done generically by always first executing a read for the desired location and then executing a write to either just write-back the read value (when we only wanted to do a read) or writing in a new value.

C Standard ORAM Security Does not Suffice for PORAM

In this section we construct an ORAM that *is* secure in the usual sense but *is not* next-read pattern hiding. In fact, we will show something stronger: If the ORAM below were used to instantiate our PORAM scheme then the resulting dynamic PoR scheme is not secure. This shows that some notion of security beyond regular ORAM is necessary for the security PORAM.

Counterexample construction. We can take any ORAM scheme (e.g., the one in Section 6 for concreteness) and modify it by "packing" multiple consecutive logical addresses into a single slot of the ORAM. In particular, if the client initializes the modified ORAM (called MORAM within this section) with alphabet $\Sigma = \{0,1\}^w$, it will translate this into initializing the original ORAM with the alphabet $\Sigma^n = \{0,1\}^{nw}$, where each symbol in the modified alphabet "packs" together n symbols of the original alphabet. Assume this is the same n as the codeword length in our PORAM protocol.

Whenever the client wants to read some address i using MORAM, the modified scheme looks up where it was packed by computing $j = \lfloor i/n \rfloor$, uses the original ORAM scheme to execute $\mathbf{O}\mathsf{Read}(j)$, and then parses the resulting output as $(v_0, \ldots, v_{n-1}) \in \Sigma^n$, and returns $v_{i \bmod n}$. To write v to address i, MORAM runs ORAM scheme's $\mathbf{O}\mathsf{Read}(\lfloor i/n \rfloor)$ to get $(v_0, \ldots, v_{n-1})$ as before, then sets $v_{i \bmod n} \leftarrow v$ and writes the data back via ORAM scheme's $\mathbf{O}\mathsf{Write}(\lfloor i/n \rfloor, (v_0, \ldots, v_{n-1}))$. It is not hard to show that this modified scheme retains standard ORAM security, since it hides which locations are being read/written.

We next discuss why this modification causes the MORAM to not be NRPH secure. Consider what happens if the client issues a read for an address, say $i = 0$, and then is rewound and reads another address that was packed into the same ORAM slot, say $i+1$. Both operations will cause the client to issue $\mathbf{O}\mathsf{Read}(0)$. And since our MORAM was deterministic, the client will access exactly same table indices at every level on the server on both runs. But, if these addresses were permuted to not be packed together (e.g., blocks were packed using equivalence classes of their indices ($\bmod \ell/n$)), then the client will issue $\mathbf{O}\mathsf{Read}$ commands on different addresses, reading different table positions (with high probability), thus allowing the server to distinguish which case it was in and break NRPH security.

This establishes that the modified scheme is not NRPH secure. To see why PORAM is not secure with MORAM, consider an adversary that, after a sequence of many read/write operations, randomly deletes one block of its storage (say, from the lowest level cuckoo table). If this block happens to contain a non-dummy ciphertext that contains actual data (which occurs with reasonable probability), then this attack corresponds to deleting some codeword block in full (because all codeword blocks corresponding to a message block was packed in the same ORAM storage location), even though the server does not necessarily know which one. Therefore, the underlying message block can never be recovered from the attacker. But this adversary can still pass an audit with good probability, because the audit would only catch the adversary if it happened to access the deleted block during its reads either by (1) selecting exactly this location to check during the audit, (2) reading this location in the cuckoo table slot as a dummy read. This happens with relatively low probability, around $1/\ell$, where ℓ is the number of addresses in the client memory.

To provide some more intuition, we can also examine why this same attack (deleting a random location in the lowest level cuckoo table) does not break PORAM when instantiated with the ORAM implementation from Section 6 that is NRPH secure. After this attack, the adversary still maintains a good probability of passing a subsequent audit. However, by deleting only a single ciphertext in one of the cuckoo tables, the attacker now deleted only a single codeword symbol, not a full block of n of them. And now we can show that our extractor can still recover enough of the other symbols of the codeword block so that the erasure code will enable recovery of the original data. Of course, the server could start deleting more of the locations in the lowest level cuckoo table, but he cannot selectively target codeword symbols belonging to a single codeword block, since it has no idea where those reside. If he starts to delete too many of them just to make sure a message block is not recoverable, then he will lose his ability to pass an audit.