Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2011-018 March 30, 2011

Remote Oblivious Storage: Making Oblivious RAM Practical

Dan Boneh, David Mazieres, and Raluca Ada Popa

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

CSAIL

Remote Oblivious Storage: Making Oblivious RAM Practical

Dan Boneh, David Mazières, Raluca Ada Popa
Stanford University and Massachusetts Institute of Technology

Abstract

Remote storage of data has become an increasingly attractive and advantageous option, especially due to cloud systems. While encryption protects the data, it does not hide the access pattern to the data. A natural solution is to access remote storage using an Oblivious RAM (ORAM) which provably hides all access patterns. While ORAM is asymptotically efficient, the best existing scheme (Pinkas and Reinman, Crypto'10) still has considerable overhead for a practical implementation: for M stored items, it stores 4 times and sometimes 6 times more items remotely, requires $O(\log^2 M)$ round trips to storage server per request, and periodically blocks all data requests to shuffle all storage (which is a lengthy process).

In this paper, we first define a related notion to ORAM, *oblivious storage (OS)*, which captures more accurately and naturally the security setting of remote storage. Then, we propose a new ORAM/OS construction that solves the practicality issues just outlined: it has a storage constant of ≈ 1, achieves $O(1)$ round trips to the storage server per request, and allows requests to happen concurrently with shuffle without jeopardizing security. Our construction consists of a new organization of server memory into a flat main part and a hierarchical shelter, a client-side index for rapidly locating identifiers at the server, a new shelter serving requests concurrent with the shuffle, and a data structure for locating items efficiently in a partially shuffled storage.

Keywords: oblivious RAM, oblivious storage, traffic analysis, cloud storage.

1 Introduction

Today, outsourcing data to a cloud system or to other remote entity/server is becoming increasingly popular and beneficial for individual users and companies. Most companies protect confidentiality of their outsourced data using encryption. While encryption protects that data, *it does not protect access patterns to the data.* Often access patterns reveal too much. For example, consider a medical database (used by a clinic) encrypted and stored on a remote server. Suppose the database contains a file with information for each disease. During H1N1 peak time, likely, the doctors/clinic retrieve the H1N1 record frequently. Using this information, the server can identify the H1N1 record. This is problematic because, if the server knows when a certain patient goes to the doctor, it can see whether the supposed H1N1 record is retrieved, in which case the server can deduce important private information about the medical condition of the patient: "he is suspected of having H1N1". Moreover, by doing frequency analysis of access patterns to files and using some public studies of frequency of diseases, a server can identify some diseases.

Goldreich and Ostrovsky [8] investigated the problem of hiding access patterns in the context of RAM machines. Their motivation was to hide the program executed by a processor from an attacker snooping on the traffic to main memory. Their model consists of a physically shielded CPU that contains a key secret to the outside world; the key is used to store the program encrypted in memory. The CPU progressively fetches instructions from the program and decrypts them using its internal (and also shielded) registers. The execution is said to be *oblivious* (and the RAM machined called an *oblivious RAM* – ORAM) if the access patterns for any two inputs causing the same number of accesses to RAM are indistinguishable. Goldreich and Ostrovsky propose two solutions. The first solution, called Square-Root, has low constants, but much higher complexity so is only presented as a first step towards the second solution, the Hierarchical solution. The Hierarchical solutions makes any program oblivious by replacing M stored items with $O(M \log M)$ items and replacing a direct access to an item in RAM by $O(\log^3 M)$ accesses to RAM.

An elegant recent work of Pinkas and Reinman [13] improves both the complexity and constants of the Goldreich-Ostrovsky approach by leveraging recent techniques such as Cuckoo Hashing and Randomized Shell Sort. In their construction, storing M items is replaced by storing $\approx 8M$ (and occasionally $\approx 12M$). Each data access is replaced by $O(\log^2 M)$ requests to memory. Both solutions [8] and [13] are efficient asymptotically, adding only polylogarithmic overhead.

Limitations of ORAM in practice. For concrete parameters, the state-of-the-art ORAM has the following limitations:

1. *Storage cost.* Considering the large quantities of data involved there is a strong desire not to expand storage needs in the cloud, not even by a factor of 2. The best approach [13] requires 8 times storage overhead and occasionally 12 times (and an optimized version can achieve 4 times overhead and occasionally 8). In fact, most cloud providers charge the customer per megabyte stored. In practice, even a factor of two, most often discards the profit made by a company by outsourcing the data to a cloud, or the price a user is willing to pay for their stored data.
2. *Blocking requests while shuffling.* The ORAM solutions require periodic shuffling of all or most of the data in memory. In previous work, during a shuffle, requests to the data are blocked to preserve security. In a cloud storage system with lots of data, shuffling can take many hours and the clients cannot afford to lose access to the data for that long.
3. *Client-server interactions.* A client–server interaction requires one network round trip time between the client and the server. Such latency seems to be one of the most important challenges with outsourcing data to the cloud or other remote server [10, 3]. The best approach, [13], requires about $\log M$ client-server sequential interactions for one request and the next request cannot start before the previous one finished. Decreasing the number of client–server interactions is essential for a practical protocol.

Our Contributions. We begin by introducing a closely related concept called *oblivious storage* (OS) that captures more accurately the privacy requirements of cloud storage. Since the original ORAM motivation is a shielded CPU accessing an encrypted program it is not surprising that variations in the model are needed when applying the techniques to privacy in the cloud. While the existing ORAM solutions [8, 13], with some relatively straightforward extensions, also satisfy the OS requirements, we show in Section 2 that there are constructions that satisfy the ORAM definitions, but fail to satisfy the OS requirements.

Second, we provide new constructions that reduce the overhead of ORAM to more practical values:

- We present a new shuffling algorithm that allows requests to proceed *concurrently* with data shuffling so there is no need to block access to data while shuffling. Moreover, shuffling happens "in-place," meaning that the total storage at the server at any point in time is only $\approx M$. While our algorithms copy small parts of data, they never perform a copy of a linear fraction of memory.

  To do so we introduce two new techniques. First, we provide a protocol to locate some data from a partially-shuffled memory efficiently; we use a data structure called shuffleMap that summarizes where each identifier can be found at some middle point during the shuffle. Second we add a second "shelter" (called the *concurrent shelter*) to prevent repetitions in the data accessed before, during, and after the shuffle. Considering that requests access partially-shuffled data and all accesses to shuffleMap are not oblivious, understanding and proving why security is preserved was quite delicate.

- We present a hybrid algorithm between the Square-Root algorithm of [8] and the optimized Hierarchical algorithm from [13]. The algorithm, coupled with a client-stored index, is attempting to achieve the small constants and the small number of interactions between client and server of the Square-Root solution of [8], while mostly maintaining the better asymptotics of the Hierarchical solution of [13].

With these contributions, we eliminate all the limitations discussed above: the storage constant is about 1, accesses can proceed in parallel with shuffling, and a data request requires a constant number (≈ 5) of client–server interactions (round trips).

Our algorithms make use of limited storage capacity at the client, which is available in all real world cloud settings. The storage capacity at the client is much smaller than the one in the cloud (on the order of square root), but is nevertheless larger than $O(1)$. In contrast, the original ORAM model assumed $O(1)$ storage at the client (indeed, CPUs only have a small number of registers available).

2 Security Definitions

There are two main parties: a server and a client. The storage server has a large amount of storage, called the *server storage*, where it stores the data from the client. The client also has some storage capacity, though considerably less than the server. The more storage available at the client, the faster will our algorithms be.

We chose to represent the storage at the server as a random-access memory with a "key-value" interface similar to that provided by cloud storage services (e.g., Amazon S3 [2] and Windows Azure [4]). The customer is charged based on the amount of storage used and the number of requests performed. Most cloud services use a hashing algorithm allowing $O(1)$ data retrieval time. To avoid ambiguity of terms, we use the term "identifier-block" pair (or simply "id-block" pair) instead of key-value pair. We use the notation v to denote an identifier, to be consistent with the virtual addresses/identifiers from ORAM. We use the notation b to denote a certain block.

Let c be the size of the identifiers (in bits) and B be the sum of the size of a block and the size of an identifier. Note that, in practice, B is much larger than c. For example, a common value for B is 4 KB and c is 32 bits, resulting in $B = 1024c$. We decided to introduce notation for B and c, despite its absence in previous work, because of this large difference in size; this will allow us to avoid large constants when designing algorithms. There are M id–block pairs in the server storage, resulting in a total of MB storage.

At a high-level, we try to prevent any information about the data from being leaked to the server or to an adversary who has access to both the data stored at the server and the messages exchanged between the client and server. We consider a *curious, but honest adversary.* We use a definition similar to semantic security where the adversary cannot distinguish between the access patterns of any two programs. The adversary cannot tamper with the data requested to and returned by the server, but would like to learn information about the content of the data. Allowing the adversary to tamper with the data is a straightforward extension to our protocol similar to the one from [8] and we will thus not focus on this case. (If identifiers are selected from a large field to prevent repeats, adding a simple MAC to the block corresponding to each identifier will allow detection of tampering.)

We are now ready to define formally the notion of *oblivious storage* (OS). OS models more naturally and accurately both the setting and security requirements of remote storage on an untrusted server. Figure 1 provides a high-level overview of the following security definitions.

Definition 1 (**Server**). *Given positive integers M, B, c, an (M, B, c)-*server*, S, is an interactive algorithm (formally modeled as an interactive Turing machine). The server S has access to $O(MB)$ storage modeled as random access memory consisting of entries of the form (v, b), where $|v| = c$ and $|b| = B - c$. (When S starts, its storage contains pairs $(i, 0)$, for $i = 1 \ldots M$). The server S responds to the following messages:*

- get(v)*: S returns v for which there exists entry (v, b) in its storage or $\perp$ if there is no such entry.*

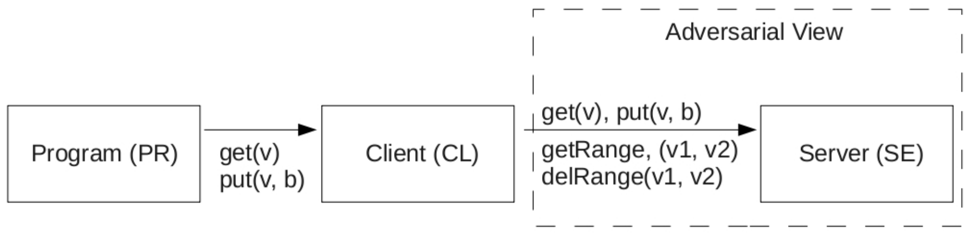

Figure 1: Overview of the parties involved in OS.

- $\mathsf{put}(v, b)$*: S places entry (v, b) in its memory; if an entry for v existed already, S overwrites it.*
- $\mathsf{getRange}(v_1, v_2)$*: S retrieves from memory all entries (v, b) for some v, $v_1 \leq v \leq v_2$, and b.*
- $\mathsf{delRange}(v_1, v_2)$*: S deletes all entries (v, b) for some v, $v_1 \leq v \leq v_2$, and b.*
- halt*: S halts.*

One might wonder why we define getRange when it can be implemented with get. Indeed, getRange is entirely optional for security. However, we chose to introduce it because, in real storage, performing such queries rather than their get equivalents results in much smaller constants, and it is useful to design and evaluate our algorithms in terms of such interfaces.

Definition 2 (Client). *Given positive integers k, m, B, c, a (k, m, M, B, c)-client, C, is an interactive algorithm (formally modeled as an interactive Turing machine) with $m + O(B)$ memory. Upon starting, the client C receives as input a secret key* SK*, with $|\mathsf{SK}| = k$. C responds to messages of the form* $\mathsf{get}(v)$ *and* $\mathsf{put}(v, b)$*. For each* $\mathsf{get}(v)$*, the client must return the value b that corresponds to the most recent* put *to* v.

Definition 3 (Program). *Given positive integers k, M, B, c, a (k, M, B, c)-prog, P, is an interactive algorithm that can send to a client $(k, *, M, B, c)$-client messages of the form* $\mathsf{get}(v)$ *and* $\mathsf{put}(v, b)$ *for some v, b s.t. $|v| = c$ and $|b| = B - c$. The program P only sends a new message to the client after it received an answer to the previous message.*

Definition 4 (Storage). *Given positive integers k, M, B, c, a (k, M, B, c)-storage is a pair of an (k, m, M, B, c)-client and an (M, B, c)-server, where the client sends messages to the server in order to satisfy requests received from some (k, M, B, c)-prog. The input to storage is* SK*, with $|\mathsf{SK}| = k$, and it becomes the input to the client.*

Definition 5 (Adversarial View). *For any (k, M, B, c)-storage, (C, S), the adversarial view, $\mathsf{VIEW}^{(C,S)}$, is a string-valued function defined on all programs P that are (m, M, c, B)-prog. $\mathsf{VIEW}^{(C,S)}(P)$ contains the list of all requests sent by C to S in chronological order and all the data stored at S after each request from C, for the duration of C's interaction with P.*

Security game. We define what it means for a (k, m, M, c, B)-storage, ST, to be oblivious using a security game between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$.

1. **Setup.** $\mathcal{C}$ picks at random SK with $|\mathsf{SK}| = k$ and begins simulating ST on input SK.
2. **Preparation.** $\mathcal{A}$ picks adaptively n programs $Q_1, \ldots, Q_n$ and sends them to C. After simulating the storage on each Q_i, C sends to $\mathcal{A}$: $\mathsf{VIEW}^{\mathsf{ST}}(Q_i)$ based on which $\mathcal{A}$ adaptively selects Q_{i+1}.
3. **Query.** $\mathcal{A}$ picks two programs P_1 and P_2 that are (k, M, B, c)-prog.
4. **Response.** $\mathcal{C}$ picks a bit b at random. Then, it simulates further ST on an interaction with P_b and sends $\mathsf{VIEW}^{S}(P_b)$ to $\mathcal{A}$.

5. **Challenge.** $\mathcal{A}$ returns a bit b', which is a guess at the value of b.

We define the advantage of the adversary $\mathcal{A}$ as the quantity

$$\mathrm{Adv}_{\mathcal{A}} := |\Pr[b' = 1|b = 0] - |\Pr[b' = 1|b = 1]|. \tag{1}$$

Definition 6 (Oblivious Storage)**.** *A (k, m, M, B, c)-*storage *is oblivious if, for all polynomial time adversaries $\mathcal{A}$, $Adv_{\mathcal{A}}$ is a negligible function of k.*

Differences between ORAM and OS. OS captures more naturally and comprehensively the outsourced storage setting and its appropriate security properties.

Let us first establish the correspondence between components in ORAM and OS. Recall that, in the ORAM setting, an encrypted program lies in memory and a CPU executes one instruction from the program at a time by fetching it from memory. ORAM aims to hide accesses to the program in memory. The client from OS loosely corresponds to the CPU in ORAM and the server in OS to the memory in ORAM. Therefore, the data at the server in OS corresponds to the ORAM program and any other in-memory data. The programs interacting with the client in OS correspond to one fixed, well-known program in ORAM: the interpreting job of the CPU which fetches and executes instructions sequentially. Let's denote this program *command executer*.

The main differences from ORAM are:

1. *Existence of Programs.* We allow a variety of programs to interact with the client; in the real-world scenario, users of some remote cloud storage can run any kind of programs that generate accesses to the server.
2. *Stronger security definition.* The adversary in OS is stronger and its ability to get certain information is motivated by practice.
   (a) OS provides *stronger access pattern indistinguishability.* ORAM requires that access patterns to the encrypted ORAM program be indistinguishable for any two encrypted programs *that produce the same number of accesses*. OS requires that access patterns be indistinguishable for any inputs of the same program (even if they produce a different number of accesses) and for any programs. The reason is that the access pattern length can leak important information about the data stored at the server. For example, if a program reads block 1 and if the block's content is 0, it makes two more accesses to some other blocks, else it halts. By observing the number of accesses, an adversary can tell whether the value of block 0 is 1. Moreover, in real use cases, users can run any kind of programs requesting data from the storage and mere knowledge of these programs can leak information about the data at the server (e.g., if a financial program is running, the data at the server is financial).
   (b) The adversary should be able to see the state of the unencrypted data at the server before the final challenge (i.e. the encrypted program in ORAM). After it sees the data, it should not be able to distinguish between the access patterns of two challenge programs of its choice. Such protection is needed; for example, consider an employee of a company that formally had access to the data of the company, but was subsequently fired; he must not be able to see or deduce any future changes to the data from the access patterns to it.
   (c) The adversary should be adaptive. It can see VIEW and the output of the client to the program and select new programs for the challenger. Such adversarial behavior is encountered in practice because the adversary itself may sometimes be a user of a storage system.
3. *Presence of non-constant client storage.* Most client machines using a remote storage system posses a significant amount of memory (RAM and disk). Moreover, in most cloud storage systems, the client

is formed of a cluster of local machines that together contain a significant amount of storage, though still less than the server-side storage capacity.

Thus, our OS model is different and our oblivious OS definition stronger than ORAM's. However, we point out that the current ORAM solutions (e.g., [8], [14], [15], [13]) can still satisfy our definition with a few straightforward extensions (e.g., the client running the ORAM protocol should perform an initial shuffle before running any program, execute requests at a constant rate, and keep the total number of requests for each program equal).

Nevertheless, one can exhibit a pedagogical example protocol that satisfies the ORAM definition, but does not satisfy our OS definition and is in fact not a secure in practice. Given a program, the client begins by following the ORAM protocol as long as the requests of the program are the same with the requests of a program executer defined above (that is, they correspond to fetching identifiers sequentially or fetching an identifier according to a branch jump). At the first request that deviates from the next request of a program executor, the client executes in plain all future requests, making the access patterns visible. This protocol follows the ORAM definition because, as long as the program is a command executor, it executes the ORAM protocol; for other programs, it leaks access patterns.

3 Related Work

There has been significant related work on Oblivious RAM [8, 6, 11, 13, 14, 15, 1, 5]. Some of this work focuses on finding algorithms that do not rely on any cryptographic assumptions such as the work by Ajtai [1] and Damgård et al. [5]. While the lack of cryptographic assumptions is very useful, the complexity costs of these schemes are much higher: the storage overhead is polylogarithmic in the number of data pairs, as opposed to being O(1), and the number of requests made for one program request is polylogarithmic. As such, we will focus on cryptographic approaches to Oblivious RAM.

Goldreich and Ostrovsky [8] first introduced the notion of oblivious RAM and provided two solutions: the Square-Root and the Hierarchical solutions.

3.1 The "Square-Root" Solution

We describe the "Square-Root" solution [8] in terms of the terminology introduced in Section 2. Consider the server memory divided in two parts: the main part and a "shelter." The main part contains M pairs of client data and $\sqrt{M}$ dummy pairs. The shelter is an append-only log of pairs that can grow to size $\sqrt{M}$. For the Square-Root solution only, consider that any data placed in the shelter has the high-order bits of the identifier v set to a distinguished value to allow clients to retrieve the entire shelter with a single $\mathsf{getRange}$.

When the client receives a $\mathsf{get}(v)$ or $\mathsf{put}(v, b)$ request, it begins by issuing a $\mathsf{getRange}$ command for the shelter and scanning the shelter's contents. If the client finds v in the shelter, it sends a get request to the server for a dummy identifier-block pair from the main part; otherwise, it makes a request for v from the main part. The client now appends v and the block data (b for a put or the block value returned by get for a get) to the shelter. When the shelter grows to $\sqrt{M}$ entries, the client merges its contents back into the main part and shuffles the main part, both operations being done obliviously.

The Square-Root solution has remarkably small constants. Each get involves three requests. Even though each operation requires a sequential scan of the shelter, there are only three client-server interactions/ round trips: a retrieval of the shelter and two one-pair accesses. However, the complexity of the Square-Root solution is clearly not satisfactory because $B\sqrt{M}$ can grow large even for reasonable workloads.

3.2 Hierarchical Solution

The hierarchical solution [8] reduces the number of server requests per program request to $O(\log^4 M)$ or $O(\log^3 M)$ with a higher constant. The storage cost, though, becomes $O(MB \log M)$. The idea is to organize the server's memory into a hierarchy with $O(\log M)$ levels, the size of each level growing exponentially.

To perform a request (get/put) for v, the client performs a request to each level of the hierarchy sequentially; for each level, the client checks if v exists on that level and if so, it makes dummy requests on subsequent levels; if v is not found on that level, the client continues with requests looking for v at the next level. After the client traverses the hierarchy in this way, the client adds an entry for v and its associated value to the first level of the hierarchy.

If a level, i, in the hierarchy gets full, the client must "spill" its contents onto level $i+1$ and shuffle the latter level, both of which should be done obliviously.

William and Sion [14] provide an improvement to the hierarchical solution: using $O(B\sqrt{M})$ memory at the client, they reduce the number of requests at the server generated for one program request to $O(\log^2 M)$, keeping the server storage cost at $O(MB \log M)$. The idea is to use an oblivious merge sort algorithm which merges two sorted lists of items by doing a linear pass of the two lists, copying data items in client memory, merging them, and putting them back at the server sorted. The reason the linear scan is possible is that among the first n items of the merged list, with overwhelming probability, there are at most $n/2 + O(\sqrt{n})$ items from each list. William et al. [15] improve the hierarchical solution by storing an encrypted Bloom filter for each level of the hierarchy and using it to learn if an identifier is stored in the level.

Recent work by Pinkas and Reinman [13], improves both complexity and constants over all these hierarchical approaches. They use recent algorithmic tools such as Cuckoo hashing [12] and Randomized Shell Sort [9]. Each request is replaced by $O(\log^2 M)$ requests, with at least $O(\log M)$ client-server interactions, and storage used is $\approx 4MB$, and $\approx 6MB$ during shuffling, with their proposed optimizations applied.

The Hierarchical solution is more complex and has higher constants than the Square-Root solution. One particular worrisome constant is the cost of client-server interaction; as discussed, such interactions are expensive because they require a network round trip between client and server. In the Hierarchical solution, this constant is multiplied by $O(\log M)$: each request gets transformed in $O(\log M)$ *sequential* server requests. One such server request can only be processed after the previous one completed: the client cannot just perform all requests at once because it would violate security or correctness.

In practice, both constants and complexity matter. Of course, it would be desirable to have both the low constants of the Square-Root solution and the complexity of Hierarchical solution, which is what our algorithm provides.

3.3 Performance Comparison

Let's compare the complexity and relevant constants of our OS scheme with the performance of relevant related work. Besides the amortized number of requests, we also compute the *online* number of requests. This is the number of requests that are performed for one program request and does not contain requests for shuffling. We include this metric as well because, in most cases in practice, this measures the delay of a request; requests for shuffling can happen after the actual request when the system is less busy with requests.

We also compare these protocols based on the number of individual get or put requests the client makes at the server (each getRange and delRange are counted as many times as there are identifiers in their range). We include this metric for fairness, because this was the metric used for analysis in previous work. We adjust all previous work values to our notation using B and c.

Table 1 compares our proposal, described in Section 4, to existing ORAM solutions. We illustrate OS for two values of m: $B\sqrt{M}$ and $\sqrt{MB \log MB}$. As we discuss in Section 5, the second value is of particular interest because its gives good asymptotics and its value is actually small for real-life parameters: 2 MB of client memory for 1 TB of server storage.

We can make the following observations:

- The number of online interactions in OS are constant (and the amortized number of round trips is either

Table 1: Comparison of our OS scheme to related work. If "online" is not specified, the metric is amortized. The storage cost for the Pinkas-Reinman approach is computed after applying the proposed optimizations [13].

Protocol	No. roundtrips		No. of get/put		Concurrent	Client memory	Server storage
	Online	Amortized	Online	Amortized			
Goldreich Ostrovsky	$O(\log M)$	$O(\log^3 M)$	$O(\log M)$	$O(\log^3 M)$	No	$O(1)$	$O(MB \log MB)$
MergeSort	$O(\log M)$	$O(\log^2 M)$	$O(\log M)$	$O(\log^2 M)$	No	$O(B\sqrt{M})$	$O(MB \log MB)$
Pinkas Reinman	$O(\log M)$	$O(\log^2 M)$	$O(\log M)$	$O(\log^2 M)$	No	$O(1)$	$\approx 4MB$ $(+2MB$ on shuffle)
OS	$O(1)$	$O(\frac{\log M}{B})$	$O(\log M)$	$O(\frac{\sqrt{M}}{B} \log MB)$	Yes	$B\sqrt{M}$	$\approx MB$
OS	$O(1)$	$O(1)$	$O(\log M)$	$O(\sqrt{\frac{M \log MB}{B}})$	Yes	$\sqrt{MB \log MB}$	$\approx MB$

also constant or approximately constant) whereas it is logarithmic (and polylogarithmic) in the other approaches. Even if the Pinkas-Reinman solution would use client memory to reduce interactions during shuffle, they still require the logarithmic term for security. We removed this term using the hybrid construction and the client-stored index, as discussed in Section 4.

- We increase the number of amortized operations, but we decrease the number of amortized interactions. We believe that, in a real system, much of shuffling is done at times with low traffic so the actual requests are not affected by the delay of shuffling.
- Our storage constant is one, unlike previous approaches.
- We allow concurrent requests (up to m/c such requests) during shuffle. In previous work, if a request happens when one of the larger levels are being shuffled, it has to wait for $O(M \log M)$ requests to finish before it can proceed.

4 Our OS Protocol

4.1 Overview

How can we maintain the asymptotic properties of the Hierarchical solution and have the low constants of the Square-Root solution?

Our protocol is a hybrid of the two solutions. First, we keep the shelter from the Square-Root solution, but we make it hierarchical. Second, we use client memory to record a map between levels in the hierarchical shelter and identifiers that are cached in them. Note that this map only contains identifiers and level numbers, and does not store any data blocks, which are considerably larger in size than identifiers. Using this client map, the client does not have to scan the whole shelter to find an identifier and can avoid the bad complexity of the Square-Root solution. Moreover, all the requests to the levels of the hierarchy can be performed in one client–server interaction because the client now knows which level contains the desired identifier and for which level he needs to request dummy values; in this way, we avoid the $O(\log M)$ client-server interactions from previous work.

Third, we support data requests concurrently with shuffling. There are two main ideas here. First is to find a data structure, called shuffleMap, that summarizes concisely where each identifier can be found at some middle point in the shuffle stage; this data structure should be smaller than $O(M)$ because otherwise the storage cost constant will not be one any more. The second idea is to add a new shelter, denoted *the concurrent shelter* to maintain security. The concurrent shelter is used only during shuffling to store the identifier–block pairs that have been accessed during shuffling. The original and concurrent shelters together

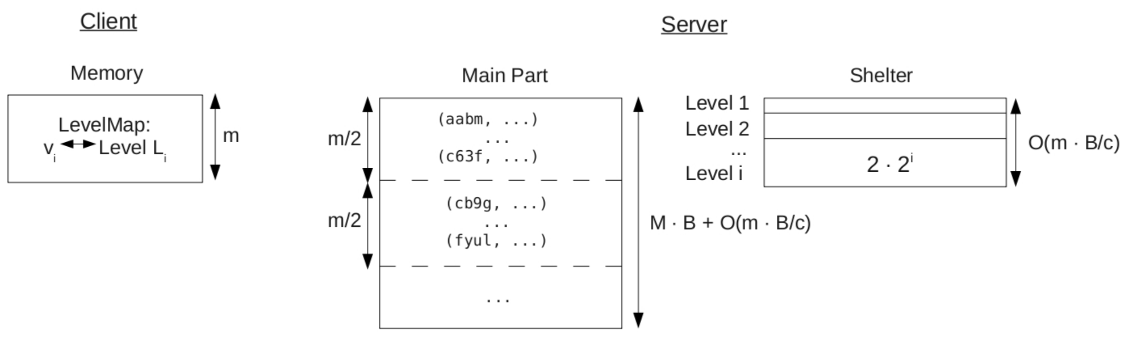

Figure 2: Memory layout.

ensure that over the entire period covering before, during, and after shuffling no single identifier–block is requested from the server (get) more than once (which would violate security). The concurrent shelter also enables shuffling in place. Accesses to shuffleMap are not oblivious and our proof shows that this does not affect security.

4.2 Protocol

Figure 2 shows the layout of memory for our protocol, and we now elaborate on each part of it.

The client stores in memory a map, called the LevelMap, mapping each identifier for which there is an entry in the shelter to the level numbers in which the entry exists. The reverse is also stored in the map: a map from each level number to all identifiers present in that level in the shelter. LevelMap consists of any efficient search data structure, (for example, a B+ tree), where the search field is the identifier and the node in the B+ tree corresponding to the identifier contains the corresponding levels in the shelter. Therefore, the client can store $O(m/c)$ such identifier–level pairs. Then, the shelter will store at most $O(m/c)$ identifier–block pairs and there will be NoLevels $:= O(\log(m/c))$ levels in the shelter. The shelter size will be $O(mB/c)$. Note that, in practice, $B \gg c$ so the maximum shelter size will often be much larger than client memory.

The main part is similar to the one from the Square-Root solution. Besides M identifier–block pairs, the main part also contains $D = O(mB/c)$ pairs of a dummy identifier and a random block value. The dummy identifiers are selected to be larger than any actual identifier; let $\{d_0, \ldots, d_0 + D - 1\}$ be the set of identifiers of all dummy variables. Like in the Square-Root solution, the client makes a request to a dummy identifier to mask the fact that it found the identifier of interest in the shelter.

The shelter is a simplified version of the hierarchy from [8, 13]. Level i of the hierarchical shelter contains $2 \cdot 2^i$ identifier–block pairs: 2^i actual pairs and the rest are dummy pairs. Because of the client-stored map, there is no need for hashing and bucketing as in previous approaches, which needed these techniques to hide when a request for a certain identifier does not find that identifier; with the level–identifier map, the client knows exactly what identifiers to request in each level so its requests will not fail. This makes the size of the shelter smaller by a factor of $\log M$ compared to [8] and a factor of two compared to [13] and avoids the constants and complexity of (Cuckoo) hashing. The dummy identifiers in each shelter also have identifiers starting from d_0 incrementally onwards. Consider that within each level i, all identifiers begin with the prefix "i :" to allow getRange to request a whole level and to make sure insertions in a certain level stay logically together. For simplicity, we will not include this detail in our algorithms below.

The period between the end of two overall shuffles is denoted an epoch. The period between two level shuffles is denoted a sub-epoch. During each epoch a different random secret key s is used to permute identifiers stored in the main part. For each level i, during each sub-epoch, the client uses a different random secret key s_i to permute identifiers in level i.

In addition to identifier–block pairs, the client maintains in memory the following information: a secret key SK with $|\mathsf{SK}| = k$ for encrypting blocks, the value s and all s_i mentioned above, as well as a dummy counter d for the main part and dummy counters d_i for each level that allow the client to fetch the next dummy identifier by incrementing these values (and thus preventing repeated accesses to the same dummy identifier).

We use PRP_s to denote a length-preserving pseudorandom permutation [7] with domain and range being the identifier space. The main part consists of all the identifiers encrypted with PRP_s and each pair in the level i of the shelter has the identifier permuted with PRP_{s_i}. For the rest of the paper, for simplicity, we say that the client searches, requests, or puts an identifier v from/to the server to mean that the client searches, requests, or puts the permuted identifier v, permuted using a PRP seeded with a client secret. Let $\mathsf{encrypt}_{\mathsf{SK}}, \mathsf{decrypt}_{\mathsf{SK}} : \{0,1\}^B \to \{0,1\}^B$ be an encryption and its corresponding decryption algorithm given key SK. Also, the client re-encrypts each data block before it puts it at the server. To prevent having to store many PRP seeds, the client can deduce the seeds from SK using a PRP.

In OS, get and put consist of three steps: request values from the shelter, request a value from the main part, and insert a value in the shelter. These three steps are similar to the ones from the Square-Root solution and any access to the shelter is similar to the one in the Hierarchical solution. In the following procedure, if a program request comes at the client when the client is performing a shuffle, the client should execute conc_get and conc_put with (v, b) as arguments instead of for any get and put to be sent at the server, respectively; we will present conc_get and conc_put in the next subsection, but for simplicity, they should be thought of as a typical get or put for now.

Algorithm 1 ($\mathsf{get}(v)/\mathsf{put}(v, b)$ of program to the client)**.**

```
1. If $v \notin \mathsf{LevelMap}$, the client does the following.
       // $v$ is not in the shelter, so request dummies from shelter and $v$ from the main part
   1) In one client–server interaction, send the server a list of requests to an unused dummy identifier
      for each level ($\mathsf{get}(\mathsf{PRP}_{s_i}(d_i))$), receive the corresponding dummy blocks, and discard their
      values.
   2) Send $\mathsf{get}(\mathsf{PRP}_s(v))$ at the server and let $b$ be the result.
   3) Execute $\mathsf{shelterInsert}(v, b)$,
   4) Send $\mathsf{decrypt}_{\mathsf{SK}}(b)$ to the requesting program.
2. Else
   // $v$ is in the shelter, so request a dummy id from the main part
   1) Let $l$ be the highest level of $v$, given by $\mathsf{LevelMap}(v)$.
   2) In one client-server interaction, send the server a list of requests: dummy requests for levels
      $i \neq l$, $\mathsf{get}(\mathsf{PRP}_{s_i}(d_i))$, and a request to $v$, $\mathsf{get}(\mathsf{PRP}_{s_l}(v))$. Let $b$ be the answer from the server
      for $v$.
   3) Send $\mathsf{get}(\mathsf{PRP}_s(d))$ to the server and ignore the result.
   4) Execute $\mathsf{shelterInsert}(v, b)$.
   5) Send $\mathsf{decrypt}_{\mathsf{SK}}(b)$ to the requesting program.
```

Algorithm 2 ($\mathsf{shelterInsert}(v, b)$)**.** The client executes the following.

```
1. $b := \mathsf{encrypt}_{\mathsf{SK}}(\mathsf{decrypt}_{\mathsf{SK}}(b))$ // refresh the encryption
2. Send $\mathsf{put}(\mathsf{PRP}_{s_1}(v), b)$ to the server // put in the first level of the hierarchical shelter
```

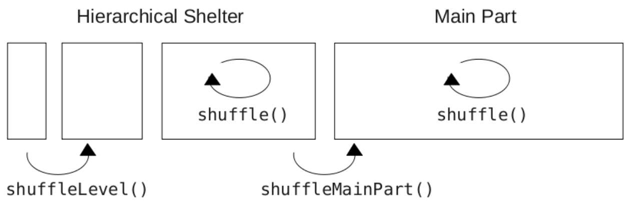

Figure 3: Shuffle algorithms overview.

3. Execute shuffleLevel() (to be presented in the next subsection) to "spill" the data in a full level onto the next level, repeating the procedure as long as a level is full. If the last level of the hierarchy gets full, call shuffleMainPart().

4.3 Shuffling protocols

Figure 3 shows an overview of our shuffling algorithms:

- shuffle(): Obliviously shuffles a part of the server's memory in which the identifiers have been permuted with the same PRP. This algorithm is in-place and allows concurrent accesses.
- shuffleMainPart() : Obliviously incorporates a flat (non-hierarchical) shelter into the main part of storage and obliviously shuffles the latter. This algorithm is in-place and allows concurrent accesses.
- shuffleLevel() : Obliviously "spills" a level of the hierarchy i into level $i+1$ of the hierarchy. This algorithm does not allow requests to proceed concurrently and is not in-place. One could follow the example in shuffleMainPart() and similarly allow concurrent accesses and design an in-place scheme, but this more complicated algorithm does not bring significant gains because shuffling levels is not as expensive as shuffling the whole main part, and does not provide significant space savings.

Allowing concurrent accesses and shuffling in-place are two goals in tension because it seems natural to make a copy of the data on which to allow accesses to happen while shuffling the original data in parallel. Our solution has three main ideas:

- During shuffling, the client builds a data structure shuffleMap that summarizes to what identifier each identifier was mapped after a step of the shuffle and stores it at the server. This allows the client to find a requested identifier in a partially-shuffled memory efficiently.
- The client creates a new shelter, *the concurrent shelter*, where any concurrent requests must be stored to prevent repetitions with the data accessed in the preceding epoch, during the shuffle, and in the following epoch.
- The client either only allows a maximum number of concurrent requests during a shuffle or delays such requests, both in order to ensure that shuffling ends before a new shuffle must begin.

shuffle() splits the memory in portions that are $O(m)$ in size and thus fit in the client's memory, and shuffle pairs of these portions at a time.

Algorithm 3 (shuffle())**.**

1. Logically split the memory in ranges of pairs such that each contains a chunk of data of size $O(m)$, numbered by $i = 1 \ldots \#$Chunks. #Chunks = $O(MB/m)$.
2. Using getRange, get each chunk, one at a time, delete the chunk from the server using delRange, add to each identifier the prefix "i :" of the chunk in which it is located, and put back the range at the server using put (all requests sent in a message).
3. For $i := 0$ to $\log \#\text{Chunks} - 1$ do:

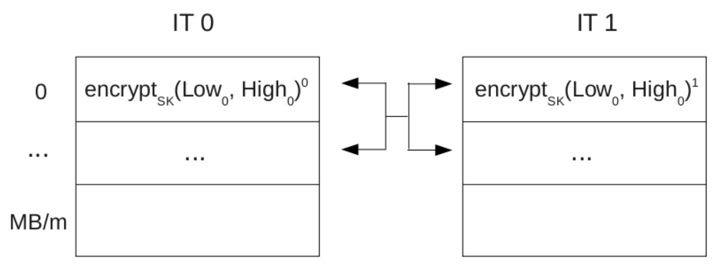
IT 0
IT 1
0
...
MB/m
encrypt$_{SK}$(Low$_0$, High$_0$)^0
encrypt$_{SK}$(Low$_0$, High$_0$)^1
...
...

Figure 4: The shuffleMap. Each box corresponds to an iteration of the for loop in Algorithm 3 for the main part. Each entry j of box i represents an encryption of the lowest and highest permuted id in the j-th chunk, where the ids are permuted with the permutation used at iteration i of the shuffle. The arrows at the right of the "IT i" box indicate which chunks are combined in iteration i of the shuffle to become what chunks for the next iteration. The shuffleMap is also stored at the server as id-block pairs: the identifier of iteration i and chunk j is "IT(i, j)".

1) Form logical pairs of chunks: chunk j is paired with chunk $j^* = j + 2^i$, if chunk j was not already paired with an earlier chunk
2) Shuffle each chunk pair at a time using PRP$_i$ (seeded with a random number to be used only for this step and this epoch): copy both chunks locally using getRange$(j : 0 \ldots 0, j : 9 \ldots 9)$ and getRange$(j^* : 0 \ldots 0, j^* : 9 \ldots 9)$, delete them from the server using delRange, apply PRP$_i$(PRP$_{i-1}^{-1}$()) to each identifier (here, let PRP$_0$ be the permutation with which the identifiers were permuted at the server), re-encrypt the blocks, sort the pairs by the new identifier, add the prefix "j :" to all the identifiers in the first half of the pairs and the prefix "j^* :" to all identifiers in the second half of the pairs, put each range at the server using put (all sent in the same message).
3) Construct "IT i" in the shuffleMap.

4. Make a pass through entire memory removing the prefixes from all identifiers. Recall the last permutation used for processing requests.

After shuffle(), the client deletes the shuffleMap(), except for the data corresponding to the last iteration. In Step 2, since the ids are permuted randomly, there can be a different number of ids in each range of the id space of equal size so such data can aid with the delimitation of data chunks in the next shuffle.

The client will perform $O(MB/m \log(MB/m))$ requests (round trips) and it will get/put $O(M \log(MB/m))$ pairs.

In the following algorithm, we incorporate obliviously a flat level of the hierarchical shelter into the main part and shuffle the main part obliviously.

Algorithm 4 (shuffleMainPart())**.**

1. Create a new hierarchical shelter, the *concurrent shelter*, containing no pairs.
2. Make a copy of the current shelter at the server; the copy is denoted the *copy shelter*.
3. Shuffle the main part using shuffle().
4. Shuffle the original shelter using shuffle().
5. For each item in the shelter, update directly the item in the main part. If the item in the shelter is a dummy, update a dummy from the main part.
6. Shuffle the main part using shuffle().
7. Delete the copy shelter and the original shelter. The concurrent shelter becomes the new shelter.

The complexity of shuffling is given by the complexity of shuffling the main part.

Assume we want to shuffle level i into level $i+1$. The algorithm functions mostly like $\mathsf{shuffleMainPart}()$, except for some details.

Algorithm 5 ($\mathsf{shuffleLevel}(i)$: Shuffles level i into level $i+1$)**.**

1. Shuffle level i using $\mathsf{shuffle}()$.
2. Shuffle level $i+1$ using $\mathsf{shuffle}()$.
3. For each pair in i, update a pair in $i+1$. For each identifier from level i for which there is the same (permuted) identifier in level $i+1$, update the value in level $i+1$ with the value from level i; for all other identifiers in level i that are not dummies, set the value of a dummy variable to contain their value.
4. Delete all contents from level i.
5. Shuffle level $i+1$ using $\mathsf{shuffle}()$.

Once the last level of the hierarchy is filled, the client uses $\mathsf{shuffleMainPart}()$ to shuffle the last level into the main part. The shelter in $\mathsf{shuffleMainPart}()$ is the last level in the hierarchy.

We can see that $\mathsf{shuffle}()$ is "in-place" because it always deletes a chunk of data before adding a new chunk of equal length. $\mathsf{shuffleMainPart}()$ does not make any copy of the main part, just of the shelter which has an asymptotically smaller size than the main part. Therefore, the overall storage at the server is $MB + O(mB/c)$, where the constant in $O(mB/c)$ is about 5 (4 from shelter and 1 from dummies stored in the main part).

While shuffling, any request to an identifier in the shelter will be satisfied from the copy shelter. We need to explain how requests to the main part are satisfied when the main part is being shuffled considering that we cannot make a copy of the main part because it would introduce too much storage overhead. Given an identifier v, applying a permutation can no longer provide us with the identifier at the server because the items are partially shuffled at the server (and in particular, we do not know the prefix of the permuted v in the current shuffle iteration). To find the identifier at the server corresponding to v, when the client performs $\mathsf{shuffle}()$ on the main part, it builds $\mathsf{shuffleMap}$ described in Figure 4. $\mathsf{shuffleMap}$ has a size of $MB/m \log(MB/m)$ and is deleted at the end of shuffle. Accesses to $\mathsf{shuffleMap}$ are *not oblivious*, but as we will see in Section 5, our proof of security still holds. Given an identifier v, using a binary search and knowledge of PRP_0 (the PRP used at the beginning of the shuffle for the ids in the main part), the client can find the chunk in which v is located. The client then only needs to check the two chunks in which v could have been mapped to in the next iteration of $\mathsf{shuffle}()$ and so forth until the current iteration. This requires $O(\log(MB/m))$ requests and interactions. If $m > MB/m \log(MB/m)$, the client can store the entire $\mathsf{shuffleMap}$ and avoid the extra requests and interactions.

Algorithm 6 ($\mathsf{conc\_get}(v)/\mathsf{conc\_put}(v,b)$ by the client to the server.)**.** Consider the appropriate case:

// **Case 1: This request is not concurrent with** $\mathsf{shuffleMainPart}()$ **and accesses main part.**

1. Simply perform $\mathsf{get}(v)/\mathsf{put}(v,b)$ at the server and return any results.

//**Case 2: This request is concurrent with** $\mathsf{shuffleMainPart}()$ **and accesses the main part.**

2. The client first needs to perform requests to the *concurrent shelter* to make sure that the identifier of interest is not there. Requests to this shelter also follow the hierarchical shelter algorithm. If the desired id is found in the concurrent shelter, the client needs to make a dummy request to the the main part, else it needs to request v from the main part.
3. To request an id from the main part, if this request comes in Algorithm 4
   - before Step 3 starts: perform the dummy or permuted-v request directly to the main part as in the unchanged algorithm;

- Step 3 starts – Step 4 starts: use shuffleMap to find the permuted identifier at the server and request it;
- Step 4 starts – Step 6 starts: make the request to the main part using the last permutation used by the client in the shuffle at Step 3;
- Step 6 starts – Step 7 starts: use shuffleMap to find the permuted identifier at the server and request it;
- after Step 7 starts: make the request to the main part using the last permutation used by the client in the shuffle at Step 6.

4. Add an entry to the concurrent shelter as in the original algorithm.

   // Case 3: This request accesses the original shelter.
5. If shuffleMainPart() is happening, use the copy shelter. If shuffleLevel() is happening, the client first finishes this shuffle and then proceeds with the request.

Note that one shuffle must finish before the next one needs to start. Since a shelter can hold as many as $O(m/c)$ pairs before it needs to be shuffled into the main part, at most $O(m/c)$ requests can happen concurrently for the duration of shuffleMainPart().

One can choose to have shuffleLevel also be concurrent in a very similar way to shuffleMainPart (in fact, even leveraging the fact that the shuffle does not need to be in place and can make copies of the levels currently shuffled). In that case, we need to prevent level $i-1$ from getting filled and spilling onto level i before we finished shuffling of level i into $i+1$. In order to fill level $i-1$, one needs 2^{i-1} accesses. Shuffling will take $O(2^{i+1}B/m \log(2^{i+1}B/m)$ requests for every 2^{i-1} requests, and summing over all levels, some requests will have to be delayed is $O(B/m \log^2(m/c))$ shuffling requests.

From all the protocols above, we can see that the number of online client-server interactions is $5+ O(B/m \log^2 m/c) = O(1)$ and of amortized interactions is $O(B/m \log^2(m/c))+ O(\frac{MBc}{m^2} \log(MB/m))$, where the last term comes from the amortized cost of shuffling the main part.

We discussed how the client executes requests, and now we need to discuss of how the client starts and ends. The client will perform a shuffle() of the main part (and include all shelter data into the main part) before interacting with any program to prevent known initial configurations to leak information about the requests made. To prevent the adversary from getting information about data content from access pattern length, each client needs to send an equal number (standard number) of requests to the server for any program and initiate the get/put protocol for each request at a constant rate. If the program finishes, the client sends "cover" requests to arbitrary identifiers to maintain the standard number of requests per program.

Theorem 1. *The storage protocol defined in this section is an oblivious storage according to Definition 6.*

The proof is in the appendix.

5 Discussion

What is the smallest m that makes the amortized number of interactions constant? The amortized number of interactions is $O(B/m \log^2 m/c + (MBc/m^2) \log MB/m) = O((MBc/m^2) \log MB/m)$, when $m \gg B$. We can see that $m = \sqrt{MB \log MB}$ bits makes the number of interactions constant. Moreover, such value for m is very convenient for practice: to store 1TB of data we need $m = 2$ MB and to store 1 PB of data we need ≈ 86MB of data. 1 PB is a large amount of storage for an application, yet most commodity computers have a main memory of at least 1 GB.

We now give some suggestions for building a real system in practice.

Caching can benefit OS because it avoids running the algorithm when there are repeated requests. In fact, the overhead of the any oblivious algorithm comes from trying to mask repeated accesses, which is exactly what caching mitigates.

As discussed, one should use as much client memory as there is available because this will increase performance. In most practical cases, in fact, the client can afford to use memory size of $O(M)$ provided that the client does not store the content of the blocks. For example, an implementation of OS would require $\approx$ 2 GB of memory at the client to store our metadata for 1 TB of data at the server, each data item being 4 KB blocks; similarly, for a server storage of 100 GB, our algorithm requires 200 MB of client storage. In this case, our algorithm has $O(\log M)$ amortized number of get/put requests per one client request and the number of round trips will also be a small constant (≈ 3).

References

[1] M. Ajtai. Oblivious RAMs without cryptographic assumptions. *ACM Symposium on Theory of Computing (STOC)*, 2010.

[2] Amazon. Amazon s3 service level agreement, 2009. URL `http://aws.amazon.com/s3-sla/`. `http://aws.amazon.com/s3-sla/`.

[3] Paula Bernier. Latency matters: cloud computing, Rich Media, Financial Industry Drive Changes in Data Center, 2010. URL `http://www.tmcnet.com/voip/features/articles/95550-latency-matters-cloud-computing-rich-media-financial-industry.htm`.

[4] Microsoft Corp. Windows Azure. URL `http://www.microsoft.com/windowsazure`. `http://www.microsoft.com/windowsazure`.

[5] I Damgård, S. Meldgaard, and J.B. Nielsen. Perfectly secure oblivious RAM without random oracles. *Cryptology ePrint Archive, Report 2010/108*, 2010. URL `http://eprint.iacr.org/2010/108`.

[6] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. *ACM Symposium on Theory of Computing (STOC)*, 1987.

[7] O. Goldreich. *Foundations of Cryptography*, volume Volume I, Basic Tools. Cambridge University Press, 2003.

[8] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/233551.233553.

[9] M. T. Goodrich. Randomized Shellsort: A simple oblivious sorting algorithm. *Proceedings 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[10] Peter Green. Cloud computing and latency, 2010. URL `http://www.agathongroup.com/blog/2010/04/cloud-computing-and-latency/`.

[11] R. Ostrovsky. Efficient computation on oblivious RAMs. *ACM Symposium on Theory of Computing (STOC)*, 1990.

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

[13] Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. *CRYPTO*, 2010.

[14] P. Williams and R. Sion. Usable PIR. *Network and Distributed Systemm Security Symposium (NDSS)*, 2008.

[15] P. Williams, R. Sion, and Carbunar B. Building castles our of mud: Practical access pattern privacy and correctness on untrusted storage. *ACM Conference on Computer and Communications Security (CCS)*, 2008.

A Proof of Theorem 1

Preliminaries. Id-s are selected from $\mathsf{ID} = \Sigma^k \times \Sigma^k$ and they are represented as two strings from Σ^k separated by colon (the first being the prefix).

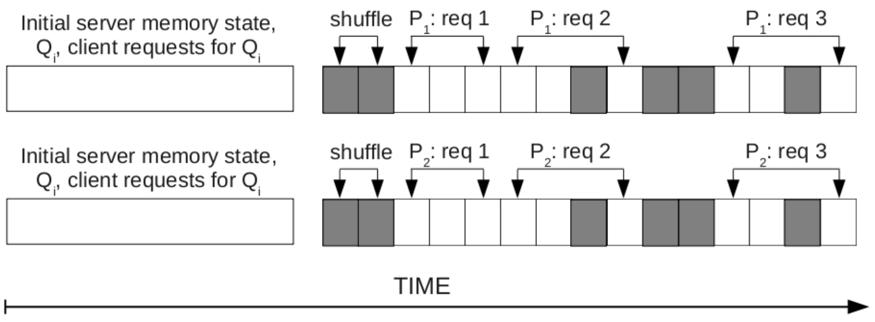
Initial server memory state,
Q_i, client requests for Q_i
shuffle
P_1: req 1
P_1: req 2
P_1: req 3
Initial server memory state,
Q_i, client requests for Q_i
shuffle
P_2: req 1
P_2: req 2
P_2: req 3
TIME

Figure 5: INFO_1 and INFO_2: Every small square indicates a request to the server and every filled in square indicates a request at the server that is part of a shuffle procedure. As such, requests 2 and 3 are concurrent.

We use the notation $x \overset{R}{\leftarrow} X$ to mean that x is drawn at random from the distribution given by X. Let $(d_1 \overset{R}{\leftarrow} D_1) \sim (d_2 \overset{R}{\leftarrow} D_2)$ denote computational indistinguishability between two distributions D_1 and D_2.

Let INFO_b be a string representing the amount of information available at the adversary when the client is interacting with program P_b.

Proof. **Strategy.** The proof strategy is to characterize the amount of information available at the adversary in the case when $P_b = P_1$ and $P_b = P_2$ and prove that the distribution of INFO_1 and INFO_2 are computationally indistinguishable. Then, the adversary cannot distinguish between P_1 and P_2 with chance significantly better than half. Once we characterize these two strings of information, we proceed to eliminate parts of these strings until they become very simple and indistinguishability is easy to prove. In the process of elimination, we maintain the guarantee that if the distributions of the newly formed strings are indistinguishable, then the distributions of the original strings were indistinguishable as well.

The information at the adversary. INFO_b consists of: the initial configuration of the unencrypted data at the server (pairs $(i, 0)$ for $i \in \{1, \ldots, M\}$), programs Q_i, $\forall i$, P_b, $\mathsf{VIEW}(Q_i)$, and $\mathsf{VIEW}(P_b)$.

Recall from Definition 5 that the state of server memory is included in the view after each request. Note that this state can be reconstructed from the initial data configuration and the requests the client makes to the server; it is thus superfluous information and we can discard it.

Note that each data block is (re)encrypted with fresh coins every time it is put at the server. Therefore, we can ignore block contents in $\mathsf{VIEW}(*)$.

Therefore, INFO_b becomes: the initial configuration of unencrypted data at the server, programs Q_i and P_i, and all requests with their id arguments from $\mathsf{VIEW}(Q_i)$ and $\mathsf{VIEW}(P_b)$.

By analyzing our protocols, we can see an important observation: *the structure of the information strings is the same for any program.* This is because the client makes the same type and number of requests to the server for any program request, the shuffle procedures start deterministically after the same number of requests (independent of what the requests are and the contents of the data), all shuffle procedures make the same number of requests and combine the same chunks of memory independent of the data content, and the client ensures that the same number of requests (and at the same rate) are sent to the server for every program. Figure 5 shows schematically the information available at the adversary.

Concurrent requests. During a concurrent request to v, the client makes requests to $\mathsf{shuffleMap}$. The access patterns of the client's requests are entirely determined by $\mathsf{PRP}_i(v)$, for each PRP_i used during shuffling. To make the indistinguishability analysis simple, we replace all get accesses to $\mathsf{shuffleMap}$ for v in an epoch with the sequence $\mathsf{get}(\mathsf{PRP}_0(v)), \ldots, \mathsf{get}(\mathsf{PRP}_{\#\mathsf{Chunks}-1}(v))$. If the indistinguishability of the new information strings holds, so does the indistinguishability of the original ones holds (because the original information strings were a polynomial function of the new strings).

Introducing $\mathsf{PRP}_0(v), \ldots, \mathsf{PRP}_{\#\mathsf{Chunks}-1}(v)$ in the information strings and proving indistinguishability may sound alarming: we are allowing the adversary to track v during shuffling (in particular, initial and final permuted identifiers). This point is rather subtle and this proof will show formally why it holds. Meanwhile, here is some intuition. Repeated accessed to the same identifier using the same PRP leak access patterns. Even though both $\mathsf{PRP}_0(v)$ and $\mathsf{PRP}_{\#\mathsf{Chunks}-1}$ are revealed, v has never been accessed with $\mathsf{PRP}_0(v)$ (because it would have been in the shelter and not fetched from the main part) and will never be accessed again with $\mathsf{PRP}_{\#\mathsf{Chunks}-1}$ (because v is placed in the concurrent shelter which becomes the new shelter for the new epoch and thus will be retrieved from the shelter and not from the main part).

Discarding the information from the adaptive programs. Recall that the client executes each program by starting with a complete shuffle of the data. Let INFO_{P_b} be the part of $VIEW(P_b)$ after the first shuffle corresponding to P_b and INFO_{Q_i} be the rest of INFO_b. Note that INFO_{P_b} only depends on the data put in the last step of the for loop of shuffle() because all future requests will be based on this new server data. Note that INFO_{Q_i} is the same (chosen from the same distribution) for INFO_1 and INFO_2 so it is indistinguishable. We can just focus on proving indistinguishability of INFO_{P_1} and INFO_{P_2} if these are independent of INFO_{Q_1}. This is true because at the last step of the for loop of shuffle(), we apply a PRP with a fresh random seed to all identifiers (which are distinct) from the previous step of the four loop – this is indistinguishable from placing random identifiers at the server.

Ignoring getRange **and** delRange**.** Due to common structure of INFO_{P_1} and INFO_{P_2} as argued above, all getRange and delRange will be at the same positions in INFO_{P_1} and INFO_{P_2} and moreover their contents will be the same (this is because in all our algorithms they have fixed arguments of the form $(i : 0 \ldots 0, i : 9 \ldots 9)$).

No repeats property. Therefore, all that is left to prove is that the distribution of the sequence of identifiers in get and put requests at the server for programs P_1 and P_2 are indistinguishable. Due to common structure of these sequences, the positions of get and put in the two strings are the same.

Note that each identifier in a get or put has been permuted with a PRP of random seed. For the adversary who does not know the random seed, these look like random numbers. However, some of these numbers may repeat. For each configuration of repeats, we need to prove that the probability of appearing in any of the two distributions is the same.

There are two cases when an id v together with a PRP are repeated:

Case 1. Exactly two put have the same permuted identifier. This can happen in Algorithm 4– Step 5 and in Algorithm 5 – Step 3. Basically, it happens when a memory chunk A of n elements needs to be used to update a memory chunk B of m elements and the later needs to be shuffled. The algorithms shuffle A, then B, then make each update from A tom some pair in B and then shuffle B again. Since B has been shuffled with a PRP with a random seed, the chance that i updates j for any program is the same.

Case 2. Exactly one get uses the same permuted identifier as exactly one put. This happens with every get because the data had to be put at the server.

First, let's prove why only one get can have the same permuted identifier as a put. get is used to fetch elements from the main part, the shelter, or shuffleMap.

- Once v is fetched from the main part, it will never be fetched from the main part until the next epoch when v will be in the main part permuted with a different PRP. When v is placed in the shelter it is permuted with a different PRP.
- When v is fetched from a level l of the shelter, it is placed on the top level of the shelter and permuted with a different PRP. The identifier v from level l will never be accessed again (until the next shuffle

of level l) because v will be found in the higher levels and dummies will be used from l.

- Recall that we replaced the get accesses to shuffleMap with $get(\mathsf{PRP}_i)$ for every PRP_i used in the shuffle. The sequence of $\mathsf{put}(\mathsf{PRP}_0(v)), \mathsf{get}(\mathsf{PRP}_0(v)), \ldots, \mathsf{put}(\mathsf{PRP}_{\#\mathsf{Chunks}-1}(v)), \mathsf{get}(\mathsf{PRP}_{\#\mathsf{Chunks}-1}(v))$ has the same probability of appearing for any program because we are using freshly seeded PRPs.

The put requests for which there is a get with the same permuted id are only the put in the last step of shuffle() (Algorithm 3 - Step 4) and the ones to shuffleMap. The reason is that all other puts are followed by getRange.

Since shuffle() permutes pairs obliviously according to a randomly seeded PRP, the chance of any configuration of repeats is the same for any program.

Having treated all cases, we showed the the distributions of INFO_1 and INFO_2 are indistinguishable, thus completing our proof.

□