Outsourced Private Information Retrieval with Pricing and Access Control

by

Yizhou Huang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2013

Abstract

We propose a scheme for outsourcing Private Information Retrieval (PIR) to untrusted servers while protecting the privacy of the database owner as well as that of the database clients. We observe that by layering PIR on top of an Oblivious RAM (ORAM) data layout, we provide the ability for the database owner to perform private writes, while database clients can perform private reads from the database even while the owner is offline. We can also enforce pricing and access control on a per-record basis for these reads. This extends the usual ORAM model by allowing multiple database readers without requiring trusted hardware; indeed, almost all of the computation in our scheme during reads is performed by untrusted cloud servers. Built on top of a simple ORAM protocol, we implement a real system as a proof of concept. Our system privately updates a 1 MB record in a 16 GB database with an average end-to-end overhead of 1.22 seconds and answers a PIR query within 3.5 seconds over a 2 GB database.

We make an observation that the database owner can always conduct a private read as an ordinary database client, and the private write protocol does not have to provide a "read" functionality as a standard ORAM protocol does. Based on this observation, we propose a second construction with the same privacy guarantee, but much faster. We also implement a real system for this construction, which privately writes a 1 MB record in a 1 TB database with an amortized end-to-end response time of 313 ms.

Our first construction demonstrates the fact that a standard ORAM protocol can be used for outsourcing PIR computations in a privacy-friendly manner, while our second construction shows that an ad-hoc modification of the standard ORAM protocol is possible for our purpose and allows more efficient record updates.

Acknowledgements

I would like to thank my supervisor, Ian Goldberg, for his always constructive suggestions, feedback and support. I thank my thesis readers (Alfred Menezes and Douglas Stinson) for their time.

Dedication

This is dedicated to my parents, family and many friends.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Notation

Symbol	Explanation
T	length of access sequence
n	number of records
b	ORAM level base
L	number of ORAM levels
γ	constant in Goodrich et al.'s ORAM scheme [GMOT12]
τ	Collusion threshold for database sharing
M	matrix used to model database
r	number matrix rows
s	number of matrix columns
$\mathbb{F}$	some finite field
e_j	a standard basis vector
v	a share of vector
t	Collusion threshold for PIR queries
ℓ	number of servers in multi-server IT-PIR scheme
x	x coordinate of a point on a polynomial
M_{key}	matrix of keys
M_{ind}	matrix of encrypted indices
M_{rec}	matrix of encrypted records
E_i	message to be signed in a blind signature scheme
k	exponent used to blind a message
σ	a blinded signature
h	a secret key used to sign a blinded message
K	a symmetric encryption/decryption key
c_i	a Shamir's secret share of a constant c
O	a database owner
id	a unique id assigned to a record

Continued on next page

Continued from previous page

Symbol	**Explanation**
S	number of data items in each row of M_{rec}
N	number of data items in M_{rec}
R	number of rows in M_{rec}
K_i	the symmetric encryption/decryption key to record i
ER_i	encrypted content of record i under K_i
EK_i	encrypted K_i under the master secret key
IV	Initialization vector (IV) for block cipher
IV_r	IV used to generate ER
IV_k	IV used to generate EK
$ENC_{K,IV}$	symmetric encryption with key K and IV IV
MAC_i	Message Authentication Tag (MAC) tag for record i
MAC_{K_i}	A hash function keyed with K_i used to generate MAC
r_i	content of record i in plain text
$OFFSET_i$	offset of record i in M_{rec}
EI_i	encrypted $OFFSET_i$ under K_i for record i
p	number of partitions of M_{ind}
$part_i$	the i$^{\text{th}}$ partition of M_{ind}
q	size limit of each partition of M_{ind}
LOC_i	the i$^{\text{th}}$ location of a given partition of M_{ind}
Q	constant parameter for M_{ind}
P	number of partitions of M_{rec} in the alternative construction
B	number of blocks in each partitions of M_{rec}
d_i	list of unaccessed locations in partition i of M_{rec}
F	a constant determining the reshuffling frequency on M_{rec}
C	constant parameter for M_{rec}
$Trans$	a transcript generated by record update operations
X	an access sequence variable
β	a random bit in the security game
κ	security parameter
ϵ	some negligible function
DR	set of records that the server knows the decryption key of
bs	size of private buffer for oblivious shuffle

Chapter 1

Introduction

Private Information Retrieval, or *PIR*, is a privacy enhancing technology (PET) that allows clients to query a database in a privacy-preserving manner. The goal is that the database server should be able to respond to client requests without learning any nontrivial information about which record the client is seeking. A trivial solution is to download the entire database and issue queries locally. This solution is clearly information-theoretically secure: no matter how much computation the server employs, it cannot learn which record the client seeks; however, it is highly impractical to transmit large databases over the Internet. PIR protocols aim to provide the same level of privacy, while incurring a strictly sublinear communication cost.

PIR schemes can be *computational* or *information theoretic*. Computational PIR (CPIR) schemes use cryptographic techniques to encrypt the user's query in such a way that the server can combine the encrypted query with the plaintext database to yield the encrypted result. This encrypted result is then returned to the client, who can decrypt it. The security of these schemes rely on the security of the underlying encryption.

Information-theoretic PIR (IT-PIR) schemes, on the other hand, are "perfectly secure" in the same sense as above — even a server employing unlimited computation cannot determine what the client was after. However, in order to achieve sublinear communication and information theoretic security at the same time, one must employ multiple database servers [CKGS98], and rely on the assumption that some number of these servers are not colluding. This non-collusion assumption is not unusual with distributed PETs; other PETs such as Tor [DMS04] and electronic voting [CCC+09] make the same assumption.

In 2011, Olumofin and Goldberg [OG11] identified a CPIR scheme and a number of IT-PIR schemes that process PIR queries faster than trivially downloading the database.

Their experimental results show that the fastest scheme examined processes a PIR query on a 16 GB database in less than 10 seconds, over 3 orders of magnitude (1000 times) faster than downloading the database over a 10 Mb/s network.

1.1 Outsourcing PIR

Although the end-to-end PIR response time for databases of a few gigabytes is somewhat reasonable, doing PIR over a one-terabyte database using the same amount of computational power still requires over 10 minutes, which is beyond practicality. Even worse, as shown by experiments [OG11], when the size of the database exceeds the size of the RAM available on the local machine, the performance begins to deteriorate as disk access times dominate.

Luckily, the computation in most PIR schemes can be easily parallelized. A recent experimental study by Devet [Dev13] has shown that with the help of 64 cores, Goldberg's IT-PIR protocol [Gol07] is indeed about 64 times faster than in a single-core setting. This promising result, measured on databases of up to 256 GB, raises the possibility of reasonable private query times to databases of even larger sizes, if the required computational power is available.

Providing PIR services on large databases offers a strong motivation to outsource them to a cloud, where the computational power of hundreds of cores can be utilized. However, this outsourcing can come at a cost to privacy: although the PIR ensures the privacy of the database clients, and encryption can ensure the database contents are protected from the untrusted cloud, the database owner may also wish to protect his *updates* to the database from being observed by the cloud. Even the *update patterns* — which records get updated when, or how often — may be sensitive information. We will later formalize this notion as *outsourcing privacy*. While outsourcing privacy protects the database owner, the complementary notion of *information retrieval privacy* protects the database clients by hiding their access patterns. We aim to construct a system that provides both of these kinds of privacy. In our system, a database owner can privately update an outsourced database, and the database servers are able to serve PIR queries while the owner is completely offline.

1.2 Related Work

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [GO96], provides a solution for outsourcing storage to an untrusted server. With a reasonable amount of

	Multiple Readers	Multiple Writers	Avoids Trusted Hardware	Hides Access from DB Owner
ORAM [GO96]	×	×	✓	×
ORAM-aided PIR [WDDB06, WS08]	✓	✓	×	✓
Delegated ORAM [FCS+11]	✓	✓	✓	×
This work	✓	×	✓	✓

Table 1.1: This table shows how our protocol differs from related work. In all of the schemes, the access histories of clients are hidden from the untrusted server.

private storage on the client side, ORAM has been shown [WST12,SSS12,GMOT12] to be much more efficient than when it was first proposed [GO96]. ORAM allows a single user (who possesses a secret key) to read and write data to a database housed on an untrusted storage server. ORAM completely hides the access patterns of records from the server, in the sense that the server cannot even tell whether an access to the ORAM is a read or write operation, nor can it tell how the current access is related to previous ones. ORAM does not allow access from multiple users unless they share the same key: a user either has the key and is able to access the whole ORAM obliviously, or she does not have the key, and cannot access any record at all. It is not obvious how to enforce any access control or pricing which allows partial access to the database for entitled users. Also, users who share the secret key see the access histories of each other. In that sense, users who share the same key should really be conceptually treated as one single user, and what they are reading or writing is not oblivious to anyone holding the secret key, including the database owner.

Any ORAM scheme naturally yields a computational PIR scheme with trusted hardware [WDDB06, WS08]. The *private storage* required on the client side now sits on the trusted hardware, which keeps the required ORAM secret key within itself, and interacts with the untrusted server exactly the same way as an ORAM client would do. A database client simply tells the trusted hardware which record she wishes to retrieve and waits for the response through a secure channel, hoping that the trusted hardware does not leak her query to others, and does not fool her with a wrong answer.

Another piece of work of particular relevance to ours is Delegated Oblivious RAM proposed by Franz et al. [FCS+11]. Each record in the Oblivious RAM is encrypted and signed by a unique set of keys initially only known by the database owner. Giving out the decryption key to someone allows her to read that record "obliviously", and giving out both the decryption key and the signing key allows both read and write access to the record. However, the database owner is able to learn the access patterns for all the other users because she knows all the keys. Even worse, she is required to do so; the database owner has to come back periodically to look at the access history, reshuffling the ORAM according to that history to allow further unlinkable ORAM accesses.

It is not surprising that none of these schemes keeps the access histories of multiple clients private from the database owner, because a general ORAM models only a single client interacting with an untrusted storage. The notion of multiple clients was not introduced in ORAM's original design, which looks into hiding the access pattern of records from the untrusted storage, not hiding the access history of users from each other. Table 1.1 shows how our protocol is different from those above.

1.3 Our contribution

1. We propose a definition for *outsourcing privacy* that reflects the privacy interests of a database owner against both the untrusted servers housing the outsourced data, as well as database clients who access that data.

2. We make a key observation that an ORAM scheme and a PIR scheme can be fruitfully combined. We combine this observation with a novel server-side indexing structure to produce a system to allow a single database owner to privately and efficiently write data to, and multiple database clients to privately read data from, an outsourced database, meeting our above definition of outsourcing privacy. Moreover, our scheme does not rely on trusted hardware.

3. We implemented our system as a proof of concept. We experiment with databases up to 2 GB in size, with reasonable performance on a single commodity server. Based on the benchmarks, we predict the performance for our protocol running on databases up to the size of one terabyte, showing its feasibility when the database owner has a high-speed corporate Internet connection (at least 100 Mb/s), even if the database clients only have slow ADSL connections.

4. We propose an improved construction which runs much faster, by allowing the database owner to read records only through PIR queries (not through the standard ORAM protocol). We implement a system based on this construction on our local machine, and simulate its performance on a terabyte-sized database over ADSL connections, showing its feasibility even if the database owner only has a slow network such as ADSL.

Chapter 2

Background

2.1 Oblivious RAM

Oblivious RAM (ORAM) was first studied by Goldreich and Ostrovsky [GO96]. In their model, a CPU with some trusted storage of constant size wishes to conduct a computation in T virtual steps using n virtual items. Oblivious RAM simulates the computation in an untrusted storage such that for any two computations that require the same number of *virtual* steps, the two *actual* access sequences of *actual* items look indistinguishable to the untrusted storage. A trivial solution for ORAM is to scan all the actual items for each virtual step and rewrite every actual item with a semantically secure encryption scheme — decrypting and then re-encrypting the original value if the actual item is not to be updated, and decrypting and freshly encrypting the new value if it is. (Recall that *semantically secure* encryption satisfies the property that someone without the decryption key must be unable to distinguish two different encryptions of the same plaintext from encryptions of different plaintexts. We will assume in the rest of the thesis that any write operation to an ORAM will use a semantically secure encryption scheme unless otherwise specified.) This trivial solution requires $\Theta(T \cdot n)$ computation cost and communication cost. To bring down the asymptotic overhead, Goldreich and Ostrovsky gave two constructions for ORAM, a *Square Root Solution* and a *Hierarchy Solution*.

The *Square Root Solution* is composed of a *shelter* of size $\sqrt{n}$, as well as a *main part* that contains n real items and $\sqrt{n}$ dummy items. Both parts are encrypted. Items in the main part are randomly permuted using a secret nonce. Each access to the ORAM first iterates through the shelter. If the required virtual item is found in the shelter, then a dummy item from the main part is fetched. Otherwise, the required virtual item is fetched from the main part. In either case, the updated virtual item is appended to the end of the shelter. After every $\sqrt{n}$ accesses, the shelter becomes full, and all the items are obliviously reshuffled[1] into the main part using a new secret random permutation. Not a single actual item in the main cell is accessed twice between two consecutive shufflings, and previous accesses become unlinkable to the ones after a shuffling. Thus, no information about the access pattern is revealed to the adversary. In order to obliviously shuffle the ORAM, each item is given a tag produced by a hash function, and an oblivious sorting algorithm is executed with the tags treated as sorting keys. Using the $\Theta(n \cdot \log^2 n)$ sorting network by Batcher [Bat68], this Square Root Solution achieves an amortized overhead of $\Theta(\sqrt{n} \cdot \log^2 n)$ for each virtual access.

The *Hierarchy Solution* organizes the ORAM into L levels. Level i contains at most b^i real items for $i = 1, \cdots, L$, which are hashed to b^i buckets using a hash function unique to that level, where b is an integer greater than 1.[2] Each of the buckets is of size $s = \Theta(\log T)$ to reduce the probability of rehashing due to hash collisions filling up buckets. To access a virtual item, the CPU first scans all the buckets on the first level. Then, for each of following levels, the CPU scans the bucket that possibly stores the item required according to the hash function used to hash that level, or accesses a dummy item if the required item has already been found. Finally, the CPU writes the updated value to the top level (level 1). After b^{i-1} virtual accesses, level $(i-1)$ becomes potentially full,[3] and all its items are obliviously rehashed to level i along with the items already on level i, using a new hash function. The total number of actual items required is $\Theta(T \cdot \log^2 T)$, and the amortized cost for each virtual access is $\Theta(\log^3 T)$.

Reasonable asymptotic costs are achieved in both of their constructions [GO96]. However, an unrealistically large constant is hidden behind the big Θ notation because of the expensive oblivious sort required to reshuffle the ORAM periodically. For this reason, ORAM has long been considered an impractical protocol.

Recently, with the increasing popularity of cloud services, ORAM has been proposed as a way to outsource data storage to the cloud while hiding the access pattern of the underlying data. Encryption alone prevents the untrusted server from learning the contents of the outsourced data. However, the access pattern might be enough for the adversary to gain confidential information. For example, for a medical database, the access frequency

[1] Remove out-of-date or duplicated items; re-order and re-encrypt all the remaining items.

[2] b is often chosen to be 2 or 4. The best choice of b is not well studied in the literature, and it might depend on the particulars of the ORAM scheme. However, in all the hierarchical schemes we are aware of, a very large b is not desirable, as this defeats the purpose of the hierarchy structure.

[3] It is possible that level $i-1$ is not full at this point because of repeated accesses to the same virtual item. However, not rehashing would leak this access pattern.

of a record might help the adversary identify the disease the record is about, and reveals possible medical conditions of patients who access those identified records. ORAM makes accesses to the database indistinguishable; the server cannot tell which record is accessed, how the accesses are interrelated, nor whether a given access is a read or write operation.

In the data outsourcing model, the constraint of $O(1)$ client-side storage does not apply any more, and the practicality of ORAM has been revisited. We denote the number of records stored on the untrusted server by n. Built on top of the primitive of cuckoo hashing and an efficient randomized Shellsort, Goodrich et al. [GMOT11,GM11,GMOT12] propose several ORAM schemes with $\Theta(\log n)$ amortized access overhead and $\Theta(n^{1/\gamma})$ storage on the client side for some constant $\gamma > 1$. Stefanov et al. [SSS12] suggest keeping track of all the records on the client side, because the size of a data item is much larger than the size of its entry in an index. The ORAM is partitioned into smaller ORAMs, such that each small ORAM can fit in the client-side memory to allow very efficient oblivious reshuffling. The access overhead of their scheme is $\Theta(\log n)$. They claim that their construction is the most efficient scheme so far in practice, with private access times only 20–35 times slower than normal unprotected access times, under practical parameter choices. Recently, Williams et al. [WST12] implemented an oblivious file system called PrivateFS, which utilizes a set of optimizations to make ORAM practical. On a 1 TB database across 50 ms network links, they achieve multiple queries per second to the file system. The underlying ORAM uses a hierarchy structure similar to Goldreich et al.'s Hierarchy Solution. Instead of trying a bucket that possibly contains the required record, the client downloads an encrypted Bloom filter on each level that tells her whether that record is on that level; if not, a dummy item is fetched instead of a possibly real one. This allows the server to use a collision-free hash function for each level, which lowers the storage overhead on the server from $\Theta(n \log n)$ to $\Theta(n)$. An efficient oblivious merge sort with $\Theta(\sqrt{n})$ client storage scrambles a level as it becomes full, and succeeds with overwhelming probability.

We do not intend to make an exhaustive review of all the ORAM schemes in the literature, nor do we intend to cover full details of the schemes above. The important point is that the practicality of ORAM has been shown under the assumption that the client also has a moderate amount of private storage, which is entirely reasonable in the data-outsourcing setting. A typical client might work with a local private storage in the order of gigabytes, wishing to store a database in the order of terabytes to the cloud.

2.2 Goldberg's IT-PIR

Our construction builds on top of Goldberg's multi-server IT-PIR protocol [Gol07]. We choose Goldberg's IT-PIR for three reasons: 1) it supports the notion of τ-independence, which is important for protecting the privacy of the database owner, as will be discussed in more detail in Section 2.4; 2) it has an open-source implementation Percy++ [GDHH12]; and 3) it is experimentally quite efficient [OG11].

Goldberg's construction models the database as a r-by-s matrix M over some finite field $\mathbb{F}$. Let e_j be a standard basis vector in $\mathbb{F}^r$ with the j^{th} entry being 1, so that $e_j \cdot M$ yields exactly the j^{th} row of M. In the simplest version of the scheme, each of ℓ servers holds a copy of the matrix M. In order to retrieve the i^{th} record, the database client sends a share of e_i under Shamir secret sharing [Sha79] (with threshold t) to each of the servers, who sees a vector v that looks indistinguishable from one chosen uniformly at random, and computes $v \cdot M$. Because of the linearity of Shamir's secret sharing, by interpolating the resulting vectors using Lagrange interpolation, the i^{th} row can be reconstructed by the PIR client. Unless more than t servers collude to share the queries they received from the client, none of them learns anything whatsoever about which record the client is after. The communication cost is $\ell(r+s)$ field elements, which optimally equals $2\ell\sqrt{rs}$ when the matrix is square; i.e. $r = s$.

This PIR scheme also supports robustness and Byzantine robustness [DGH12]. For the above privacy parameter t, as long as at least $t+2$ servers respond to the query correctly, the other (misbehaving) servers will be identified, and the client will still be able to reconstruct the correct response. This allows us to withstand — and identify — servers that attempt to disrupt the protocol.

2.3 Symmetric PIR and Oblivious Transfer

Symmetric PIR (SPIR) protects the privacy of the database server by making sure that the database client learns only one record per access request. (This rules out the trivial download scheme, for example.) Oblivious Transfer (OT) provides the same privacy guarantee, but it does not have SPIR's constraint of sublinear communication cost, and so it is a strictly weaker notion. Coupled with anonymous credentials and zero-knowledge proofs, some of the SPIR and OT schemes in the literature can support pricing and access control over the records in the database, which is well-suited for e-commerce applications, such as selling e-books in a privacy-friendly way. We briefly introduce two flavors of such constructions below.

Henry et al.'s SPIR Built on top of Goldberg's IT-PIR protocol [Gol07] and Kate et al.'s polynomial commitments [KZG10], Henry et al.'s SPIR protocol [HOG11] supports tiered pricing, which naturally induces an SPIR scheme with access control. The database client proves to each database server that her query vector evaluates to a standard basis vector at $x = 0$ with an efficient batch zero-knowledge proof. If the proof is valid, the server receiving the proof is convinced that only one row is retrieved by multiplying the input vector with the database matrix. By utilizing the homomorphism of the polynomial commitments, the database client can also prove in zero-knowledge that her wallet, which is encoded as an anonymous credential, stores enough balance to purchase the record with a price corresponding to the tier encoded in the wallet. For full details, please refer to the paper [HOG11].

Camenisch et al.'s OT In Camenisch et al.'s OT construction [CDN09, CDN10], the entire encrypted database is published. The encryption key for the i^{th} record is a *unique signature* on the message "i". (A unique signature scheme is one in which there is exactly one valid signature for any given message and public key.) In order to decrypt a record, an OT client requests a blinded signature on the desired index. Since the signature is blinded, the signer does not learn the message to be signed, which is the index of the record. In order to enforce access control (AOT [CDN09]) or pricing (POT [CDN10]), access control or pricing information is encoded in the signature as well. The client proves that the blinded message is well-formed, and that her credential satisfies the access control policy specified in that blinded message. We refer readers to the papers [CDN09, CDN10] for further details.

2.4 SPIR and OT with Data Privacy

Our protocol will contain a component where the untrusted cloud servers need to provide access to records from a database M_{key} of symmetric keys, using pricing or access control to limit who gets to see which keys. This can be easily accomplished with the SPIR or OT protocols above. Importantly, however, *the cloud servers themselves* must not be allowed to see the keys.[4] We now provide two solutions to this problem.

[4]If a cloud server acts as a database client and purchases a key for itself, then it of course will learn that key. Note that this scenario does not violate our security notions, however.

τ-independence The idea of τ-independence was introduced by Gertner et al. [GGM98]. It ensures that a coalition of τ or fewer servers can deduce nothing nontrivial about *the contents of the database*. As observed in later work [HHG13], this feature is supported by Henry et al.'s SPIR protocol [HOG11]. With τ-independence enabled, rather than each server storing a copy of M_{key}, the servers instead each hold a Shamir secret share of M_{key}. Unless more than τ of them come together to combine their shares, none of them learns the contents of M_{key}.

Threshold signature In Camenisch et al.'s OT construction [CDN09, CDN10], the encryption key of a record is a unique signature on its index. In a nutshell, the client blinds a message E_i, which is a one-way function of the record index i (as well as of pricing or access control information, if needed), by raising it to a random power k. The server signs the blinded message ${E_i}^k$ using a secret key h by computing $\sigma = e({E_i}^k, h)$ where e is a bilinear pairing. The client then computes $K = \sigma^{1/k} = e(E_i, h)$ which is then the decryption key. We can prevent the servers from learning K by turning this into a *threshold signature scheme*. Now, the database owner generates ℓ secret shares $c_1, \cdots, c_\ell$ for the value $c = 1$ using Shamir secret sharing with threshold τ. Each server gets a share $h_j = h^{c_j}$ and uses it to compute $\sigma_j = e({E_i}^k, h_j)$. The client then performs Lagrange interpolation in the exponent to recover $\sigma = e({E_i}^k, h)$ with $\tau + 1$ valid responses. Each E_i $(i = 1, \cdots, n)$ is stored on all ℓ servers and is considered public information. Database clients can retrieve any portion of the E_i's using PIR queries and recover M_{key} entries for records they are entitled to access with the above threshold signature scheme.

In both solutions, if there is no coalition of servers exceeding some threshold τ, none of the servers learns any nontrivial information about M_{key} by providing the SPIR or OT service. In reality, cloud computing service providers care about their reputation. It is not an unrealistic assumption that they would honestly follow the protocol instead of actively breaching from it by talking to parties they are not supposed to talk to, although they might be curious to try to learn something from the transcripts they are allowed to see. It would interesting to examine the non-collusion assumption from the perspective of game theory, and provide more incentives for non-colluding behaviours. This is, however, out of the scope of the thesis.

Chapter 3

Constructions

We now describe the construction of our scheme. There are three parties involved: one *database owner*, denoted by O; ℓ servers each holding a copy of the outsourced database; and database clients who issue read queries for records stored in the database. In reality, each of the ℓ "servers" might be a cloud service itself, such as Windows Azure, Amazon AWS, etc. Note that in this case, ℓ servers do not refer to ℓ computation units within one cloud, but rather ℓ non-colluding clouds. We denote by *private storage* the storage local to O. The database owner stores *records* in the database; these correspond to the *virtual items* in Section 2.1 above. We denote the number of records by n, and each record is associated with a unique *id* ranging from 1 to n.

3.1 Privacy Constraints

We care about privacy both for the database clients and the database owner. We define *information retrieval privacy* and *outsourcing privacy* for them respectively below.

Information retrieval privacy The definition of *information retrieval privacy* starts with a database client retrieving a record with *id* i. Assuming that the number of colluding servers in transaction with the client does not exceed the privacy threshold t, none of the servers learns anything about i through the transaction. The database owner O also learns nothing about i.

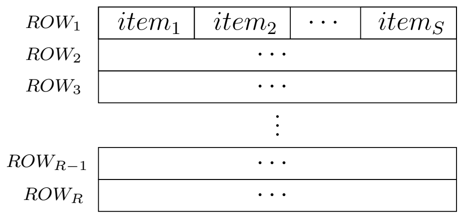

Figure 3.1: The layout of M_{rec}. Each row places S data items from the underlying ORAM in a level-by-level order, starting from the top level. $S = \lceil N/R \rceil$, where $N \approx 4n$ is the number of data items (n real plus about $3n$ dummy) in the ORAM and R is the number of rows in M_{rec}.

Outsourcing privacy The database owner O updates the database over time. Neither the database clients nor the untrusted servers learn anything about the update pattern for records they are not entitled to access.

All of the interactions between a database client and the servers are standard PIR or SPIR transactions, and the database owner can be completely offline during those transactions; thus it is easy to see that *information retrieval privacy* is guaranteed by the properties of PIR and SPIR.[1]

3.2 Overview

Our system stores three matrices on the cloud servers. First, M_{rec} stores the encrypted database records. These records are arranged logically into an ORAM and then laid out into a matrix by concatenating the elements of the ORAM in some deterministic order (say, level-by-level), and having each row of the matrix M_{rec} consist of some (integer) number of the ORAM elements so as to make the shape of M_{rec} as close to square as possible. Database clients will use PIR to retrieve rows of M_{rec}; this novel combination of ORAM and PIR will allow for multiple database clients to privately read records, while a single database owner can privately update the database. Figure 3.1 shows the layout of M_{rec}. The second matrix, M_{ind}, stores the encrypted indices that keep track of the location of each record within M_{rec}. Finally, M_{key} stores a list of uniformly random symmetric encryption keys $\{K_1, \ldots, K_n\}$, one for each record in the database.

[1]Note that although Goldberg's IT-PIR protocol is information-theoretically secure, the zero knowledge proofs required for SPIR makes information retrieval privacy protected only computationally when exactly t servers collude in Henry's SPIR scheme [HOG11].

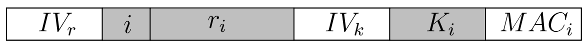

Figure 3.2: The layout of a data item. The light grey parts are encrypted.

M_{rec} and M_{ind} are replicated across each of the ℓ clouds, while M_{key} is distributed using one of the data privacy techniques from Section 2.4 so that no coalition of τ or fewer cloud providers can read the contents of M_{key}.

The data owner maintains a master secret key KEY, which is used to access M_{rec} and M_{ind} as described in detail below.

A data item in M_{rec} contains three parts (as shown in Figure 3.2): the encrypted content ER_i of the underlying database record, the encryption EK_i of key K_i (both under a semantically secure encryption scheme), and a MAC tag MAC_i. Here $ER_i = IV_r \| ENC_{K_i,IV_r}(i \| r_i)$, $EK_i = IV_k \| ENC_{KEY,IV_k}(K_i)$, $ENC_{K,IV}(\cdot)$ is symmetric encryption with key K and initialization vector IV, and $MAC_i = MAC_{K_i}(i \| EK_i \| ER_i)$,[2] where r_i is the content of the record with *id* i. We allow r_i to carry whatever necessary metadata is required by the particular ORAM scheme in use. EK_i helps the database owner recover K_i for reshuffling operations. For simplicity, we call the record with *id* i *the* i^{th} *record* or *record* i. A dummy data item can simply be a random string of the appropriate length.

The elements of M_{ind} can be thought of as a list of authenticated semantically secure encryptions, such as $(IV, EI_i, MAC_{K_i}(IV \| EI_i))$, where $EI_i = ENC_{K_i,IV}(i \| OFFSET_i)$ and $OFFSET_i$ indicates where record i resides within M_{rec}.

Every time a record is updated in M_{rec}, the ORAM will move records around, due to the rewrite to the top level or because of the reshuffling of some levels. Therefore, M_{ind} will also need to be updated. However, updating a subset of the entries in M_{ind} can leak information about the access pattern. For now, consider our scheme to update the *entire* M_{ind} for each update operation on M_{rec}. For records that do not change their offsets in M_{rec}, their entries in M_{ind} are simply re-encrypted using a new IV. We will provide a more efficient construction for M_{ind} in Section 3.4.

Now, a complete retrieval action for the i^{th} record in the database requires three PIR queries on the three matrices mentioned above.

1. A PIR query on M_{ind} for the offset of record i in M_{rec}. No access control is required for this PIR query, since the database client can decrypt the offset only if she has

[2]For good cryptographic hygiene, separate keys derived from K_i should be used for the encryption and the MAC. We elide this detail for ease of notation.

already retrieved K_i. There might be multiple data items corresponding to a record in M_{rec} (depending on the underlying ORAM scheme), and M_{ind} keeps track of the one that reflects the most recent update. With K_i, the database client is able to verify the MAC and decrypt the offset for record i.

2. An SPIR query over M_{key} to retrieve K_i. Pricing and access control can be enforced using existing schemes in the literature, such as Henry et al.'s PSPIR [HOG11] or Camenisch et al.'s ACOT or POT [CDN09, CDN10]. ACOT and POT are not SPIR schemes *per se*, because they all require downloading the whole encrypted database (albeit just the smaller database of keys M_{key} and not the entire database M_{rec}). However, as observed by Henry et al. [HHG13], clients can issue PIR queries to retrieve the part of the encrypted database they are interested in, and then conduct zero-knowledge proofs required in ACOT or POT with constant communication overhead, thus achieving overall the sublinear communication cost required by SPIR.

3. A PIR query on M_{rec} for the encrypted record. Note that the database client needs to learn $OFFSET_i$ before she knows which row (containing the desired record) to retrieve from M_{rec}.

Dummy entry in M_{key}. In a priced PIR scenario, a PIR user might not want to reveal the fact that she is retrieving the up-to-date version of a record she has already purchased by skipping the SPIR query. To circumvent this, we can add a dummy entry to M_{key} which allows database clients to purchase a dummy key with price 0. This of course requires that the price of an SPIR query be hidden from the SPIR servers, as in Henry et al.'s work [HOG11].

Sequence of PIR/SPIR queries. Each retrieval request should start with a PIR query on M_{ind}, followed by a SPIR query on M_{key}, and end with a PIR query on M_{rec}. Querying M_{key} before M_{ind} works as well for some applications, but not always, as we will discuss in Section 3.5.

3.3 Construction One

We must consider which ORAM scheme should be used for our construction. Some ORAM schemes cannot be employed directly for our purpose, such as that of Stefanov et al. [SSS12],

		Database size		
		64 GB	**256 GB**	**1 TB**
Record size	**4 KB**	240 MB	1 GB	4 GB
	64 KB	12 MB	48 MB	240 MB
	1 MB	704 KB	3 MB	12 MB
	8 MB	64 KB	288 KB	1.5 MB

Table 3.1: Size of local storage required by the database owner O for the index. The column headers show the number of bytes required if the database were to be stored on O without outsourcing; note that if the database is organized as an ORAM on an untrusted server, there is a storage overhead inherently required by ORAM, which is about 4 times in our ORAM scheme. The row headers indicate varying record sizes.

which stores some up-to-date data items in private storage; in our scheme, we require the database owner be able to be completely offline when clients read the database.

Another consideration is efficiency. There are various engineering factors in all dimensions that would affect the performance of a real-world system, especially in a cloud setting, such as data replication, load balancing, etc., the discussion of which is out of the scope of this thesis. We do not intend to find a particular ORAM scheme that would work best with those engineering factors, which are probably highly dependent on the specific underlying application as well. A concrete scheme that fulfills our privacy requirement is shown here for the sake of completeness. However, we do not argue that it is the most suitable ORAM scheme for all purposes.

We present a simplified version of William et al.'s ORAM scheme which replaces the Bloom filters with a full index that keeps track of where each individual record resides in the ORAM. This index is stored entirely on O, which consumes roughly $n \cdot \log_2(4n)$ bits of private storage, where $\log_2(4n)$ is the number of bits to encode a single index record.

The private storage also keeps track of a list of locations storing dummy items on each level that have not been visited; this consumes about $(2n) \cdot \log_2(2n)$ bits, where $2n$ is roughly the number of dummy items that the database owner needs to keep track of and $\log_2(2n)$ is the number of bits to encode each of them (since there are at most $2n$ locations on each level). Note here that the total number of blocks in the ORAM is about four times the number of records n.

Table 3.1 lists the estimated amount of private storage required given different record sizes measured in bytes. Each record is encrypted in a data item. A data item is slightly larger than the record because of the metadata encoded, such as EK_i, IV_k, etc.

	items	**dummy items**	**reshuffled**	**moved down**
level 1	2	0 to 2	every update	every two updates
level i ($i > 1$)	2^i	at least 2^{i-1}	every 2^{i-1} updates	every 2^i updates

Table 3.2: This table shows how each level in the ORAM is organized. Note the column "items" includes the number of dummy items. A reshuffle of level 1 means fetching the entire 2-item level and writing the entire updated level back.

Our ORAM is organized into L levels with 2^i items on the i^{th} level ($i = 1, \cdots, L$), including at least 2^{i-1} dummy items. An exception is the first level, where there are only two items, and no dummy items are required. Because the full index is in private storage, O knows in which level an up-to-date record resides and where exactly the corresponding item is on that level. An access to the ORAM starts with a single request, which fetches the target location on the target level, the entire first level, and also a unique dummy item from each of the other levels. The updated record is then written back to the top level. The outdated item should be erased by writing back a new dummy item. To avoid leaking which level the required record is at, every single dummy item accessed should be re-encrypted as well.

Initially, all the items are on the bottom level (so $n \leq 2^{L-1}$), and they are gradually moved to top levels with update accesses from O. After each 2^i accesses, the contents of levels 1 through i are moved down to level $i+1$, which is reshuffled using an oblivious sort algorithm, such as the $\Theta(n \cdot \log n)$ oblivious merge sort introduced by Williams and Sion [WS08]. Table 3.2 shows how each level is organized in our ORAM.

Our construction is similar to Williams et al.'s ORAM protocol, the key difference being that we do not need a Bloom filter on each level, because the owner-side index stored in O tells her directly where the target item is. The reason behind this simple construction is to make the guarantee for *outsourcing privacy* less obscure, and we think it is good enough for a proof-of-concept implementation, which is discussed in Section 4. Assuming an oblivious sorting scheme of complexity $\Theta(n \cdot \log n)$ is used to sort n items, after 2^{L-1} updates, the number of operations required for reshuffling is $\Theta(\sum_{i=1}^{L} 2^{i+1} \cdot (i+1) \cdot 2^{L-i}) = \Theta(2^L \cdot L^2)$, and all the items are moved back to the lowest level again. Therefore, the amortized cost for each update is $\Theta(\log^2 n)$.

How outsourcing privacy is protected. We call records that someone is entitled to access (due to access control or purchase) "disclosed records" to her. For update operations corresponding to the records disclosed to an untrusted cloud server, the server does learn the fact that those records are updated after these operations, but this is allowed by the definition of outsourcing privacy. For an update of a non-disclosed record to a server, by comparing M_{ind} with the older version, the server only learns that her disclosed records are not updated, and nothing more. In M_{rec}, she simply sees the entire first row is fetched with a new item written back that looks random to her, and for each of the following levels, a unique position is fetched since the last reshuffling, and that position is not any of the positions where her disclosed records are. Thus by looking at M_{rec}, the server cannot tell which record is being accessed, and how the current access could be linked to previous ones. The same argument holds for a database client, who has strictly less information than the cloud server.

3.4 Efficient server-side Index

From Table 3.1, we see that the size of a full indexing structure goes up to the order of gigabytes under some parameter choices. This is a manageable size for the client-side index in private storage, because accessing that structure from O is completely local and does not require any network transmissions. For M_{ind}, however, if after each database update, the entire structure needs to be re-encrypted and transmitted over the Internet, the overhead is rather high and seems unrealistic to deploy for databases with large numbers of records.

We propose an enhancement: partition the list of indices in M_{ind} into p parts denoted by $part_1, \cdots, part_p$. Each of these parts is organized as a queue of constant limited size, and $part_i$ contains the indices for records with id ranging from $(i-1) \cdot \lceil n/p \rceil + 1$ to $i \cdot \lceil n/p \rceil$ (though not in any particular order, and intermingled with dummy elements). When the index of a record needs to be updated, an index item should be appended to the end of the corresponding queue and when the size of a queue hits its limit, O needs to retransmit all the encrypted indices for that part. The choice of p will be discussed later in this section.

Each part is treated as a row in M_{ind} for the database clients to issue PIR queries; that is, when looking for the offset of record i in M_{rec}, the database client will perform a PIR query to retrieve row $\lceil {}^{i}/_{\lceil n/p \rceil} \rceil$ from M_{ind}. This will be a part containing the offset information for record i somewhere inside it. In order to find the right index record, the database client, once it learns K_i, simply tests *each MAC value* in the retrieved row to find the right one, which it then decrypts to yield $OFFSET_i$. The database client should test the MAC values starting from the end of the queue to get the most up-to-date $OFFSET_i$. Figure 3.3 shows the layout of M_{ind}.

To update an index in $part_j$, the database owner appends the updated index item to $part_j$ for the target record index, and appends a random string the same length as an index

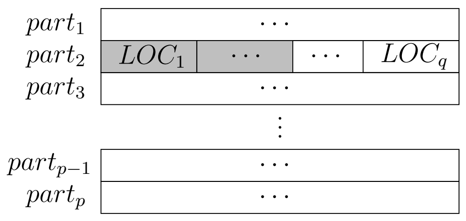

Figure 3.3: The layout of M_{ind}. Each row $part_i$ is organized as a limited-sized queue with a size limit of $q \approx Q \lceil n/p \rceil$, which stores the indices for records with id ranging from $(i-1) \cdot \lceil n/p \rceil + 1$ to $i \cdot \lceil n/p \rceil$ (as well as some dummy items). The light grey part in $part_2$ indicates the length of the current queue in $part_2$ (known only by O). When that queue grows to LOC_q, the entire $part_2$ needs to be rewritten by O. Q is a parameter that trades off write performance for read performance.

item as a dummy item for all the other parts. For this construction to work, we require O to store the entire index locally, which has been justified in Table 3.1.

If we limit each queue size to $Q \cdot \lceil n/p \rceil$, then for every $(Q-1) \lceil n/p \rceil$ index changes, O needs to replace each individual part only once. Thus the amortized end-to-end response time for updating one index is $p \cdot (1 + \frac{1}{Q-1}) \cdot U$, where U is the overhead to encrypt an index item and upload it to ℓ servers.

With p equal to 1, we achieve the maximum savings for update operations, but database clients will need to download the entire indexing structure for each query. A proper choice of p is required to strike a balance between the efficiency of update operations and PIR queries.

For example, for a 1 TB database with block size 1 MB, with 128-bit AES and 128-bit HMAC-MD5, the server-side index is roughly 96 MB, (there is about two times storage overhead if we set $Q = 2$), which can be partitioned into 64 parts, each with size 1536 KB, which could be transmitted over the Internet within a few seconds. The cost for updating one entry in the index here should be roughly equal to uploading $64 \cdot (1 + \frac{1}{2-1}) \cdot 48B = 6\,\text{KB}$ of data. Note that each index item is of size 48 bytes and that in an Internet setting, the cost for uploading 6 KB of data should dominate the cost to encrypt them. For a database client, M_{ind} is a matrix of dimension $64 \times 1536\,\text{KB}$, and thus the communication cost between the client and one server is then about 1.5 MB for each query. These parts should be initialized to different states to avoid the replacing of all parts simultaneously; this affords some measure of de-amortization. When a reshuffling of some level i in M_{rec} happens, we need to update the indices for more than one record. To avoid leaking access patterns, the database owner should pretend that 2^{i-1} records were updated (it might be true) and access each part 2^{i-1} times. When the cost of doing so becomes too high, it might just be more efficient to replace the part with an entirely new one using a single write. It is straightforward to verify that our efficient server-side indexing structure does not break *outsourcing privacy*.

3.5 Pricing and Access Control

In our construction, each record is associated with a unique key. We enforce pricing and access control when a database client retrieves this key obliviously, through Henry et al.'s PSPIR [HOG11], or Camenisch et al.'s ACOT [CDN09] and POT [CDN10].

Re-purchase on update. In some applications, it might be a desirable feature to force database clients to re-purchase a record after an important update. The database owner can change K_i to enforce a re-purchase for record i. To avoid leaking access patterns, the entire M_{key} needs to be re-shared if the underlying SPIR scheme is PSPIR with τ-independence. On the other hand, with an SPIR scheme based on POT modified with our threshold signature scheme, it is not obvious how to do this at all; that is, how to hide which K_i is updated while keeping all the other K_j ($j \neq i$) unchanged; we leave this as an open problem, and recommend sticking to PSPIR with τ-independence if support for forcing re-purchase of records is desired.

A client will realize she has to re-purchase a record when no entry in M_{ind} has a valid MAC tag. Then she can purchase either the updated key or the dummy entry from M_{key}, depending on whether she wishes to purchase the updated record or not. Note that this decision can be made only after learning whether the record has been updated since the last purchase; this justifies our choice of querying M_{ind} before M_{key} in Section 3.2.

3.6 Construction Two

The vector-matrix multiplication part of Goldberg's IT-PIR scheme [Gol07] is easily parallelizable. By utilizing highly parallelizable computation resources, Devet's experimental results [Dev13] have shown that computing a PIR query takes time inversely proportional to the number of cores in use. On the contrary, parallelizing ORAM accesses is not well studied. In fact, all of the ORAM schemes we are aware of in the literature involve expensive computation on the database-owner side when executing an oblivious reshuffle. Even if the underlying reshuffling algorithm were parallelizable, the *database owner* is unlikely to be able to provide enough computational resources to take significant advantage of it. Thus, in a cloud setting, PIR queries have the potential to be computed quickly for very large database given enough cores, while the speed of ORAM access is always somewhat limited by the available local computation resources.

Inspired by this observation, without violating our definition of *outsourcing privacy*, we propose an alternative scheme for updating database records. In a nutshell, in order to update a database record, the database owner first conducts the read protocol using one SPIR query and two PIR queries exactly the same way as she does in Construction One, and writes the updated record back to M_{rec} in an oblivious manner. Since we do not require M_{rec} to support oblivious *read* operations through the ORAM protocol (as the read operations are protected by the PIR protocol), M_{rec} can be organized to allow much more efficient oblivious writes.

If the database owner O simply wishes to read a record from the database without updating it, she still needs to write the same record back re-encrypted through an update operation. This makes read and write operations indistinguishable to the servers. We do not make a distinction between read and write operations from O in the remainder of this section.

Our construction is similar to Stefanov et al.'s ORAM construction [SSS12]. However, we do not require any up-to-date records to be stored in private storage, so that PIR reads from other clients can be executed while the database owner is completely offline.

We first describe how M_{rec} is organized, and what information is required to be kept in private storage.

M_{rec} is partitioned into P parts of equal length, each containing B blocks, with records encrypted and distributed uniformly at random among the parts and among locations within each part. Each part should be small enough to fit entirely in O's private storage, such that an oblivious reshuffle can be executed efficiently for individual parts. The owner keeps track of the part number as well as the offset within the part for each record. She also keeps track of, for each part i, a list d_i of locations that have not been accessed since the last reshuffle of this part.

In order to update a record, O chooses a random part j in M_{rec}, a random location i from the list d_j, writes the encrypted record to the i^{th} block of part j, and removes i from d_j. When a certain part has been accessed $F \cdot \frac{n}{P}$ times, that part is reshuffled. Here F is a constant that determines the frequency of reshuffle operations. After updating M_{rec},

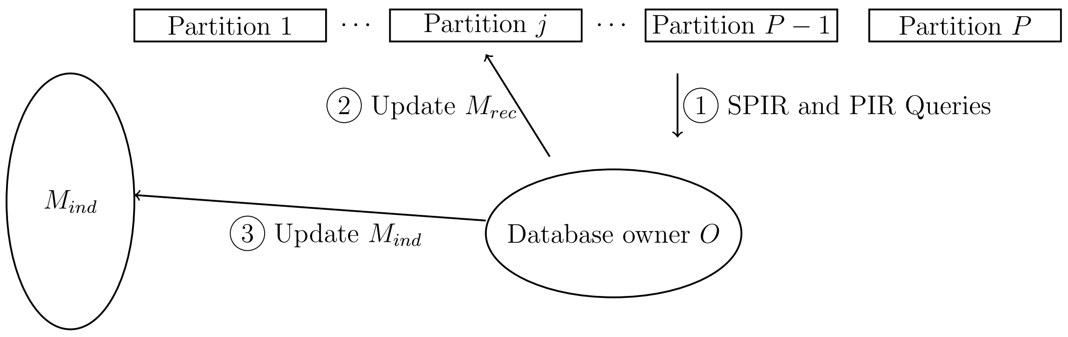

Figure 3.4: Our second construction. M_{rec} has P parts. To update a record, the database owner O first reads it through PIR queries, and then chooses a random part j and a random location i within part j that has not been touched since the last shuffle of part j. The same j's and the same i's are chosen for each server. Partition j is reshuffled, if it has been accessed $F \cdot \frac{n}{P}$ times since its last reshuffle, where F is a constant that determines reshuffling frequency. O then generates a list of index entries that need to be updated, and applies these updates to M_{ind} by either re-encrypting and replacing the whole M_{ind} or using the efficient index update scheme described in Section 3.4.

O should update M_{ind} entirely or append encrypted updated indices to each part of the efficient server index structure in Section 3.4. If a reshuffle operation occurs, the number of index items that need to be appended to each part of M_{ind} should be equal to B. Figure 3.4 shows an overview of our scheme.

The number of records stored in each part changes from time to time as the protocol runs. According to the standard bins and balls analysis [RS98], if $P = \sqrt{n}$, each part has at most $\sqrt{n} + o(\sqrt{n})$ records with a failure probability of $\frac{1}{poly(n)}$. In order to prevent any part from overflowing, $B = (1 + C + F)\frac{n}{P}$, where C is a constant that controls the failure probability. A larger C introduces a smaller probability of part overflows. A larger F improves the amortized performance, at the cost of slower private reads and reshuffling operations. One should choose a set of B, C, F, P for a given n that allows B to fit in private storage, while at the same time does not cause any part overflow after executing a reasonably large number of private writes. Given a parameter choice, we have a simulator that simulates millions of private writes in a few minutes, and reports an error should any part overflow occur. Since we did not find or derive any good indication of how the probability of failure relates to C numerically, we suggest that one should test her choice of B, C, F, P with the simulator extensively before deploying the system. In our experiment in Section 4.3, we set $P = \sqrt{n}$, $C = 0.75$ and $F = 0.5$. We observed no failures.

This protocol makes update operations more efficient for at least the two reasons below:

1. Each update not involving a reshuffle operation requires only encrypting and uploading a single block to M_{rec}.
2. Each part fits entirely in private storage, allowing oblivious reshuffle to be much more efficient than oblivious sort.

3.7 De-amortized ORAM

One drawback of ORAM is that an expensive periodic shuffling is required, which makes some accesses far more expensive than others, especially when a high-numbered level needs to be shuffled. The result is that some update operations have to queue up if the previous update happens to be an expensive one. For some applications, blocking an update operation for too long can be a serious problem. De-amortized ORAM, well studied in the literature [BMP11, GMOT11, KLO12, SCSL11, SSS12, WST12], makes each access to the ORAM bounded by a reasonable overhead.

For some of these schemes, de-amortization comes naturally in their construction, such as in the work of Shi et al. [SCSL11] and Stefanov et al. [SSS12]. We suspect that these two schemes can both be used in our construction with some slight modifications. For example, in Stefanov et al.'s scheme [SSS12] we can move those up-to-date records which are supposed to be stored in private storage to the untrusted servers by organizing them in a separate ORAM on each server. However, a careful security examination is required before building an outsourced PIR system on top of them.

The other works on de-amortized ORAM [BMP11, GMOT11, KLO12, WST12] follow a particular paradigm. The idea is to construct a new level preemptively in the background, which becomes ready right before the old level has to be discarded. Some extra space is required for the construction of the new level, because the old level should be kept in its entirety to serve continuing ORAM queries before the new level completes its shuffling. It is not hard to see that such a paradigm can be applied to our ORAM construction as well, such that outsourcing privacy is still guaranteed. After we apply this de-amortizing technique to M_{rec}, updates to M_{ind} are somewhat de-amortized naturally, because we can update M_{ind} along the way as a new level is being constructed gradually in M_{rec}.

Similar to Stefanov et al.'s ORAM scheme, our second construction allows update costs to be somewhat de-amortized because each reshuffle operates on only a small portion of the database. For further de-amortization, each of the reshuffle operations could be de-amortized using the standard de-amortizing technique, but to deploy the technique requires extra server storage overhead the same size as required in the original scheme.

3.8 Discussion

Compared to our first construction, our second construction avoids the expensive oblivious sort required for oblivious reshuffles. It requires on the servers storage overhead less than 1.5 times the size of the underlying database, as compared to three times for our first construction. This also makes PIR queries less expensive to compute. The most expensive operation of our second construction is to reshuffle a part, which is much cheaper than shuffling the entire database. Thus, in terms of worst-case performance, the second construction also outperforms the first one. A question arises naturally: why do we present the first construction at all?

We have emphasized in Section 3.3 that we do not intend to find the most suitable ORAM scheme to construct our protocol. We have chosen a simple ORAM scheme as a proof of concept, for the sake of an easy security analysis and an easy implementation. The point of presenting the first construction is to demonstrate the fact that a combination of an ORAM scheme and a PIR scheme can be fruitful. With further development in the ORAM literature, it is entirely possible that by plugging in some future ORAM scheme, our first construction may outperform the second construction presented in this thesis.

3.9 Security Analysis

We give a formal definition for *outsourcing privacy* in this section. Both PIR clients and database servers are our adversaries in our scheme, but PIR clients have strictly less information than the database servers, so we only consider the latter in our analysis. Also the transactions between the database owner and all servers are identical, so we only consider one of the servers.

Define an *access sequence* to be a sequence of record *id*'s the database owner updates. When the database owner updates the records in a particular access sequence, it induces a sequence of physical block accesses, which forms a *transcript*. The transcript $Trans$ contains a sequence of encrypted blocks and encrypted index items, a sequence of locations in M_{rec} and M_{ind} that are accessed, and the values written to a subset of these locations.

Recall that a record id is "disclosed" to the server if the server obtains its decryption key. This implies that the server is allowed to see the up-to-date decrypted content of this record, as well as its decrypted content at any point in the history. After all, there is not much we can do to prevent the server (or any other PIR users) from keeping copies of past versions of the encrypted database.

From the transcript, the server is able to learn some of the entries in the access sequence by trying to decrypt all the encrypted blocks and encrypted index items. She might try to deduce useful information from the sequence of access locations as well. Consider the game below:

Access Sequence Distinguishing Game. The server sets the length of the access sequence to be T, and picks a set of disclosed records DR. The challenger gives it the keys to, and locations of, those records. The server chooses two access sequences A_0 and A_1 each of length T, so long as they agree in all the entries with record id's from DR. The challenger chooses a uniformly random bit β, and sends back the transcript $Trans$ induced by A_β. The server then tries to determine the bit β given the transcript.

We define *outsourcing privacy* formally as the server wins the game only with probability $1/2 + \epsilon(\kappa)$, where ϵ is a negligible function in the security parameter κ from the underlying encryption scheme and pseudorandom permutation (used for reshuffling parts and determining the sequence of random locations to access in M_{rec}).

This definition captures the notion that the server's advantage in distinguishing two access sequences, that are both consistent with a seen transcript given the server's knowledge about disclosed records, is negligible.

Assume the server were able to gain a non-negligible advantage of winning the game. First, this advantage cannot be gained from the portion of $Trans$ corresponding to updating M_{ind}. In the case where M_{ind} needs to be re-encrypted for every record update, any gained advantage of distinguishing the two access sequences implies that non-trivial information is learned from encrypted index items to which the server does not know the decryption keys. This would imply that the underlying encryption scheme is not secure. Similarly, for the efficient construction of M_{ind} in Section 3.4, an advantage of distinguishing the two access sequences implies an advantage of distinguishing a valid ciphertext from a random string appended as a dummy item, which also implies that the underlying encryption scheme is broken.

Next, consider the possible advantage gained through the portion of $Trans$ corresponding to accessing M_{rec}.

In our first construction, for an entry in the access sequence corresponding to a non-disclosed record, the server sees, from the transcript, a pseudorandom location (selected using a pseudorandom permutation) not overlapping with any locations of disclosed records is accessed on each level. Nothing non-trivial about which record is being accessed can be learned from these locations. Any advantage possibly gained would imply either an advantage of distinguishing a valid ciphertext (without knowing the decryption key) from a random string, or deducing non-trivial information from the ciphertext, both implying an insecure encryption scheme.

In our second construction, a pseudorandom location (also selected using a pseudorandom permutation) is chosen for each write-back of an encrypted record, which is independent of the *id* of the record being accessed. Thus, no information about *id* is revealed from access locations. Any gained advantage of distinguishing the two access sequences implies that the server learns something non-trivial from the encrypted blocks she does not possess the decryption keys to. This also leads to the conclusion that the underlying encryption scheme is insecure.

To conclude, if we assume the underlying encryption scheme and pseudorandom permutation are secure, *outsourcing privacy* under our definition above is fulfilled in both of our constructions.

Chapter 4

Performance Evaluation

4.1 Experimental Setup

As a proof of concept, we implemented an end-to-end system that fulfills our privacy requirements. We require the binaries compiled from Percy++ [GDHH12], an open-source implementation of Goldberg's IT-PIR protocol, to make our system work. We used AES-128 for encryption and 128-bit HMAC-MD5 for message authentication codes. In all our experiments, one client, one database owner and three servers[1] ran on a machine with two quad-core 2.5 GHz Intel Xeon E5420 CPUs, 32 GB of 667 MHz DDR2 memory, and Ubuntu Linux 10.04.01.

4.2 Results for Construction One

Figure 4.1 and Figure 4.2 present the end-to-end response time for our system. All the databases in our experiments are stored entirely in RAM, and the figures show the computation time without the I/O time to read the database into memory. Unless otherwise specified, the size of each ORAM block in M_{rec} is set to be 1 MB, and when we mention the size of the database, it is the size of the original database before being organized into an Oblivious RAM; the ORAM containing the database is about four times as large as the underlying database. In order to update the server-side index M_{ind}, the database owner simply sends the entire encrypted M_{ind} over, which is small enough for our parameter

[1]There do not exist many cloud service providers, and we consider three to be a practical number.

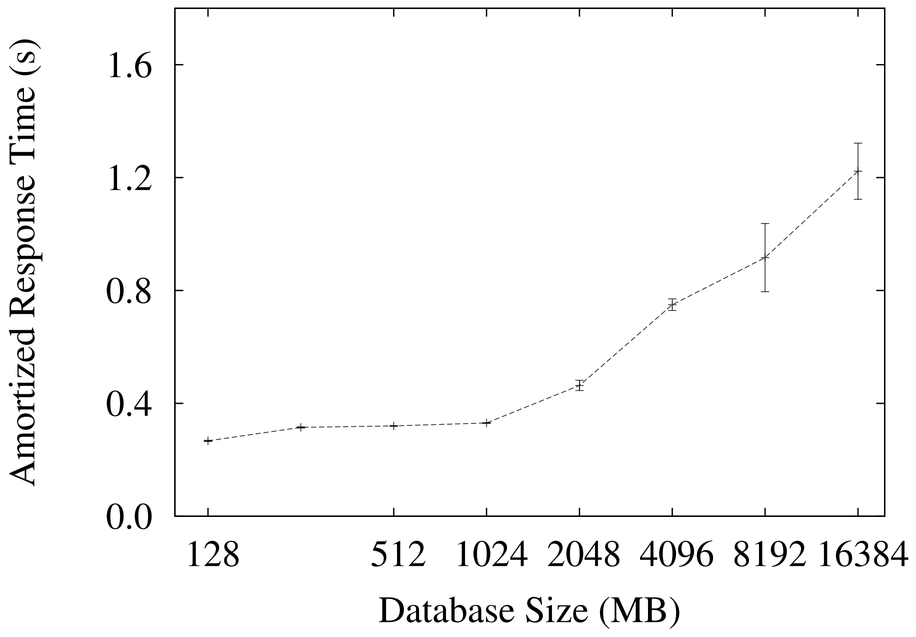

Figure 4.1: Construction One: Measured amortized end-to-end response time for private writes to ORAMs of size up to 64 GB (underlying databases being up to 16 GB). Each experimental trial consists of n updates for a database with n 1 MB records. We ran 100 trials of experiments for databases of less than 2 GB and 2 trials of experiments for larger databases. The average amortized response time and its standard deviation is shown in the figure. The small standard deviations for amortized response times on 2 GB and 4 GB databases resulted from a stable private write performance for small databases.

choice. (We did not use the enhancement of Section 3.4 in our implementation.) The end-to-end response time is measured entirely from the perspective of the client for PIR queries and of the database owner for update requests respectively. We benchmarked our system only for databases of size up to 16 GB, because the extraordinary amount of time required to gather enough samples of amortized cost for larger databases. We stress that in practice, one should use our second construction or an ORAM scheme with better worst-case performance (a de-amortized version of this construction for example) for such databases.

We predict the performance of our protocol for larger databases based on the number of block operations (e.g. uploading, downloading, encrypting, and decrypting blocks) and how fast each of them can be conducted according to our benchmarks of small databases. Considering an ORAM containing n records organized into L levels where $n = 2^{L-1}$, after

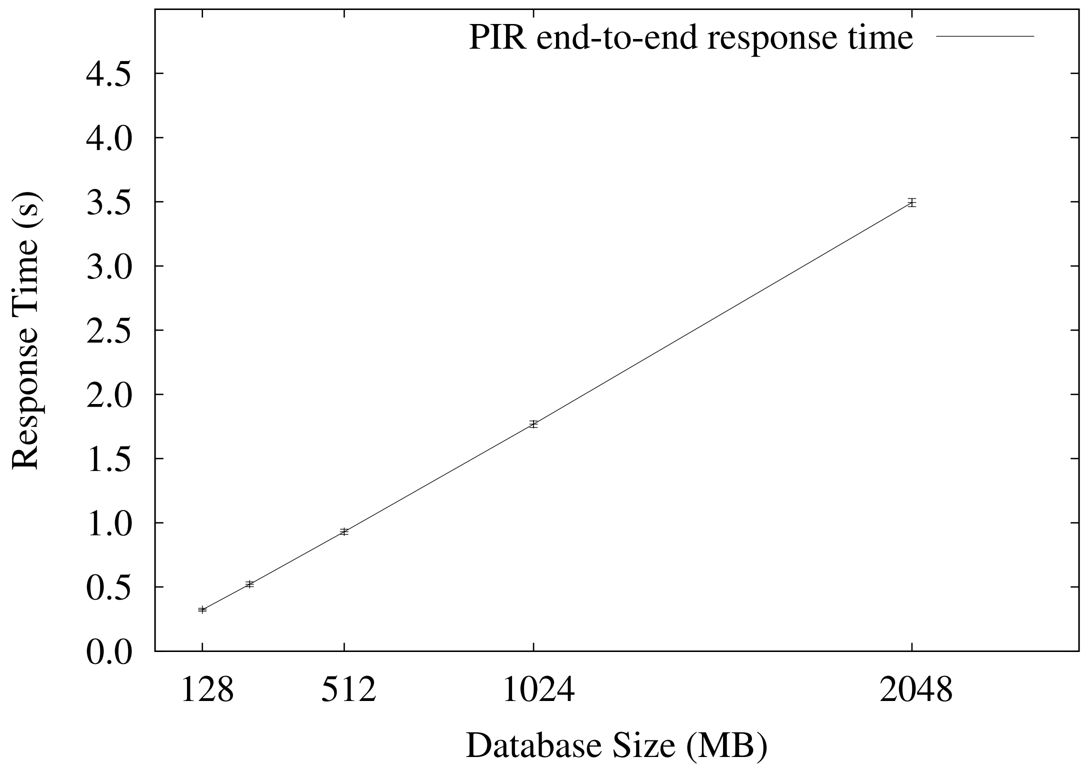

Figure 4.2: Construction One: Average end-to-end response time for private reads over ORAMs of different sizes up to 8 GB (underlying databases being up to 2 GB). The deviation is small, implying stable performance. For each parameter choice, we ran the experiment 100 times. The dominating overhead comes from retrieving the record from M_{rec}. For PIR queries over M_{rec} and M_{ind}, Chor's PIR scheme [CKGS98] is used for better performance. We do not test read performance for larger databases, because three 8 GB ORAMs already hit the limit of our RAM, and the performance will begin to suffer from hitting the disk. One of the points outsourcing PIR is to utilize the distributed storage of the cloud such that hitting the disk is not necessary and the linear performance can be maintained for very large databases.

n update requests for the purpose of reshuffling, the total number of blocks that need to be downloaded and decrypted is $\sum_{i=2}^{L-1}[2^i \cdot i + (2^{i+1}-2)] \cdot 2^{L-i-1} + 2^{L+1} - 2 + 2^L \cdot L = (\frac{1}{2}L^2 + \frac{7}{2}L - 2) \cdot 2^{L-1}$, and the total number blocks that needs to be encrypted and uploaded is $\sum_{i=2}^{L-1}(2^i \cdot i + 2^i) \cdot 2^{L-i-1} + 2^L + 2^L \cdot L = (\frac{1}{2}L^2 + \frac{5}{2}L - 1) \cdot 2^{L-1}$. According to this computation, to update an ORAM containing n records in our construction bears an amortized overhead of $\frac{1}{2}L^2 + \frac{9}{2}L - 1$ of download and decryption operations as well as $\frac{1}{2}L^2 + \frac{7}{2}L$ of upload and encryption operations on blocks. We also take into account the cost to update M_{ind} in our prediction without the efficient M_{ind} construction in Section 3.4. The result is plotted in Figure 4.3 for different network speeds. The take-away is that each update operation takes about one minute (amortized) for a database owner with corporate network connections, which is feasible, especially in applications where updates are infrequent.

Devet's experiment [Dev13] shows that the time for a cloud to compute a PIR query is inversely proportional to the number of cores it uses for the computation, which is not a surprising result at all.

To give an idea of how parallelization might push the boundary of PIR computation, for example, with the computation power of 256 cores in each untrusted cloud, we estimate that a PIR query over a 1 TB-sized database (organized into a 4 TB ORAM) takes about 7 seconds to compute using Chor's IT-PIR [CKGS98] option in Percy++, and less than 2 seconds to transmit between the servers and the PIR client (even if the client has slow ADSL speeds of 2 Mb/s upload and 10 Mb/s download). To justify our preference of Chor's IT-PIR over Goldberg's scheme [Gol07] for the M_{rec} and M_{ind} databases, we observe that τ-independence is not required for them (unlike for M_{key}). In addition, a realistic deployment may not use enough different cloud providers in parallel to effectively take advantage of the Byzantine robustness of Goldberg's scheme. The upside of making the choice to use Chor's scheme is that it is about 4 times faster than Goldberg's in the Percy++ implementation.

Figure 4.4 shows the end-to-end overhead to update one ORAM record when varying levels need to be shuffled after the update. (Recall that level i will be shuffled every 2^{i-1} updates.) De-amortizing techniques are not implemented, but for a system that is to be used in the real world, such techniques are recommended.

Our first construction here seems much slower than that of Williams et al.'s ORAM scheme [WST12]. However, note that in Williams et al.'s experiment, a block size of 4 KB is used in comparison to 1 MB in our measurement. Indeed, each of our update operations is updating 256 times as many bytes as in William et al.'s benchmarks. Figure 4.5 shows our prediction of update request performance for a block size of 256 KB. As the block size becomes even smaller, encrypting and transmitting index items is becoming a performance bottleneck. We suspect that employing our efficient M_{ind} construction would address this

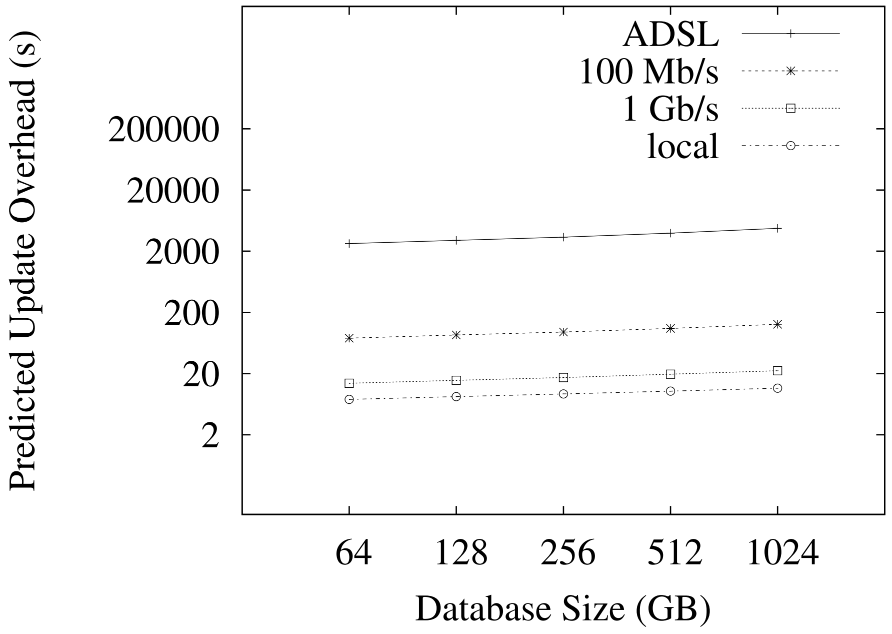

Figure 4.3: Construction One: Predicted amortized end-to-end response time for a private 1 MB record update by the database owner for larger databases. We assume that the ADSL connection bears upload and download throughputs of 2 Mb/s and 10 Mb/s respectively, and that the bidirectional throughput of a corporate network is 100 Mb/s. In a high-throughput network setting (100 Mb/s, Gigabit Ethernet, etc.), the prediction shows that our protocol is feasible over a terabyte-sized database. On the other hand, network transmission is a great bottleneck for ADSL users; this is an inherent limitation of the underlying ORAM scheme. If it is necessary for home users to be database owners, we propose using a less communication-intensive ORAM scheme (e.g. Stefanov et al.'s scheme [SSS12], with private cache organized into ORAM on servers) to address this issue, but a careful investigation of the privacy implications and the performance of the resulting construction is out of the scope of this thesis.

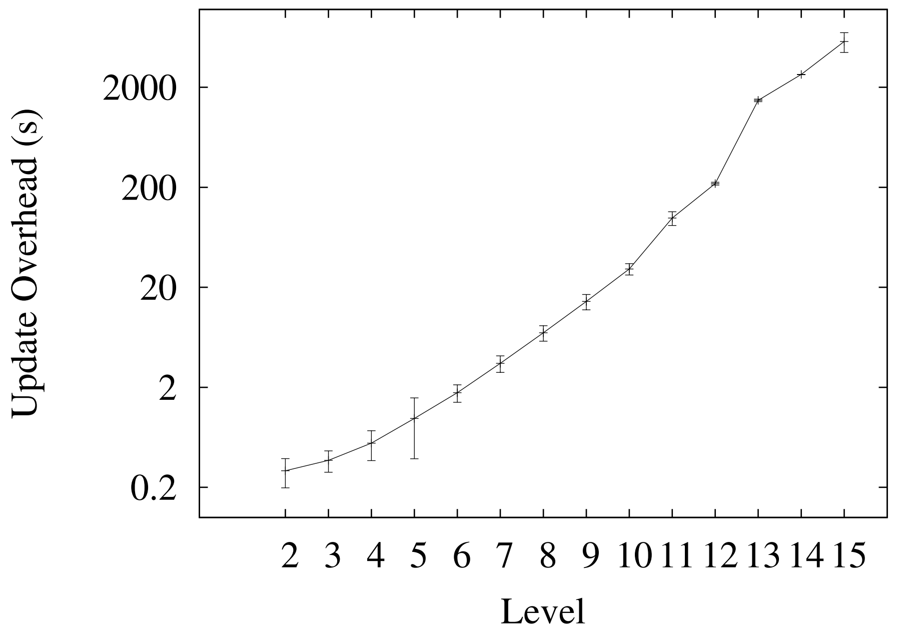

Figure 4.4: Construction One: End-to-end response time and its standard deviation for one ORAM update request when the given level needs to be shuffled after the request. Level i will be shuffled every 2^{i-1} updates. Note that a deeper slope appears between the update overhead of level 12 and level 13. This is because we allow up to 8 GB of private storage in our experiment. The reshuffles of level 2 to level 12 simply require downloading the whole reshuffling part and reshuffling it in the private storage, while reshuffling lower levels requires an oblivious sort that is more expensive. The numbers of samples for reshuffling level 14 and 15 are 2. Starting from level 13, the number of samples for reshuffling each level is twice as many as the number of samples for reshuffling the level with number one greater.

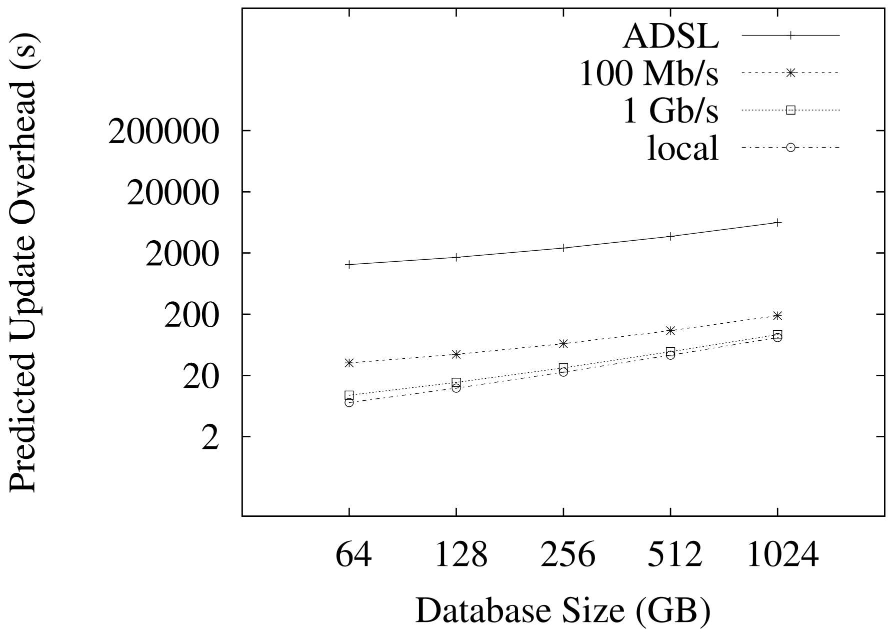

Figure 4.5: Construction One: Predicted amortized end-to-end response time for a private 256 KB record update for larger databases. We assume that the ADSL connection bears upload and download throughputs of 2 Mb/s and 10 Mb/s respectively, and that the bidirectional throughput of a corporate network is 100 Mb/s. Compared to Figure 4.3, the amortized response time for writes here is smaller. However, the difference becomes less obvious for larger databases, because of a larger M_{ind} that needs to be encrypted and transmitted per update for a smaller block size.

issue.

4.3 Results for Construction Two

We implemented a system based on our second construction. The experimental setup is the same as above. We also simulated the performance of our system running over ADSL connections. We assume that an ADSL connection bears upload throughputs of 2 Mb/s and download throughputs of 10 Mb/s respectively. In the simulation, we keep track of the numbers of bytes encrypted, decrypted and transmitted, and translate them into access overhead based on our benchmark. We set the encryption speed to be 121 KB/ms and decryption speed to be 114 KB/ ms in our simulation. Our estimation of encryption and decryption speeds is derived from 1000 trials of encryption and decryption operations on 1 MB blocks.

We do not test the performance of private reads for our second construction, since the private reads protocol executes in the same way as the first construction.[2] In the remaining part of this section, a "private write" does not include first reading the record with PIR queries.

Unless otherwise specified, we set P (the number of M_{rec} parts) to be $\sqrt{n}$, p (the number of M_{ind} parts) to be $\sqrt{n}$, and F (parameter for reshuffling frequency) to be 0.5. We will show and discuss later how the choice of P and p affects the performance of our system. Also, unless otherwise specified, the size of an M_{rec} block is 1 MB (and the record size is slightly smaller, due to encryption overhead), and we will show how our system performs with a smaller block size. Each experimental trial consists of private write operations between two consecutive reshuffles on M_{rec}. Unless specified, any experiment or simulation begins with a randomized state and is executed with 100 trials. A private write consists of privately updating both M_{rec} and M_{ind}.

Figure 4.6 shows the cost of each update access in 100 experimental trials (54654 private writes involved) over a 1 TB database (requiring 2.25 TB of actual server-side storage for M_{rec}). We can observe a sparse spectrum of dots around 65536 ms, which is corresponding to the cost of reshuffling a M_{rec} part. An more sparse spectrum appears between 1024 ms and 4096 ms, which is corresponding to the cost of replacing M_{ind} part(s). Note that we randomize the initial queue length of each M_{ind} part to reduce the number of M_{ind} parts that require replacement in the worst-case scenario. All the dots below 1024 ms can be

[2]Note that private reads are cheaper in this construction for the same underlying database because less server-side storage overhead is required here.

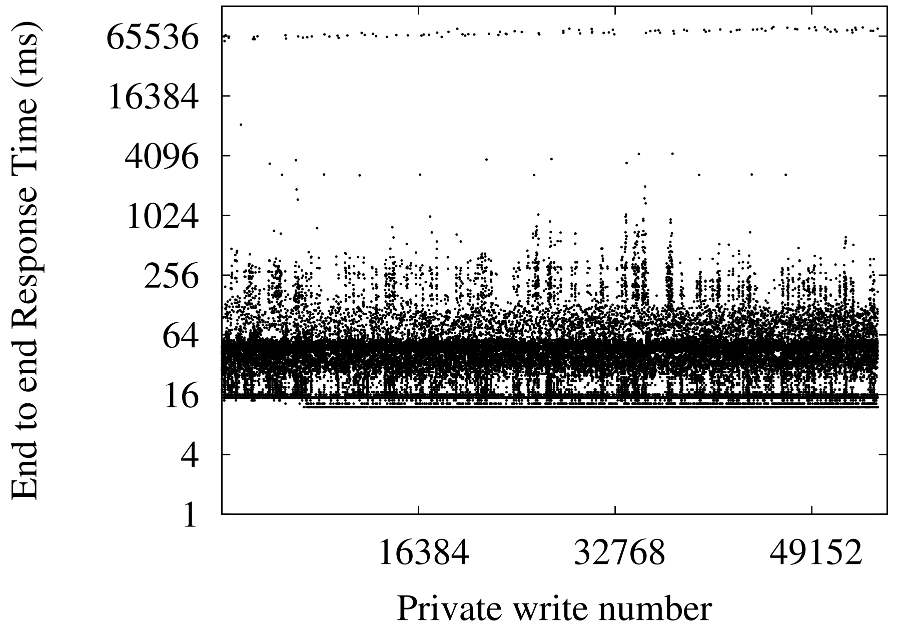

Figure 4.6: Construction Two: Measured end-to-end response time for each private write to database of size 1 TB (2.25 TB storage required for M_{rec}).

safely regarded as updates that do not involve any reshuffling or replacement operations; there are various update costs for these operations. We observe in our logs that they correspond to different disk access response times. The amortized access overhead is 182 ms.

Figure 4.7 and Figure 4.8 show the cumulative distribution (CDF) of write costs for databases of different sizes on a local machine and over ADSL connections respectively. As can be seen in Figure 4.7, most private writes take less than 100 ms to complete (note the logarithmic scale of the y-axis). The spikes nearing the right end of the x-axis correspond to reshuffling an M_{rec} part. Since we do not simulate the hard disk access cost, we observe sharper spikes in Figure 4.8. The little spike in the middle of each curve in Figure 4.8 corresponds to the replacement of an M_{ind} part. Over ADSL connections, the amortized response time for private writes on a 1 TB database is 63.5 s, but the response time in the worst case exceeds 8 hours. De-amortization techniques described in Section 3.7 can trade off worst-case performance for storage overhead; having logarithmic part size also provides a similar trade-off, which is discussed in further detail later in this section. Note that uploading 1 MB to three PIR servers over ADSL connections without any privacy protection for updates takes $\frac{3 \times 1 \times 8\, Mb}{2\, Mb/s} = 12s$, and Construction Two is only about five times slower amortized.

Figure 4.9 shows the distribution (CDF) of costs for updating M_{ind} over ADSL connections given two different choices of p (number of M_{ind} parts), with the underlying database being 1 TB. We choose p to be 1024 (in the order of $\sqrt{n}$) and 8 (in the order of $\log n$). The amortized cost for updating an M_{ind} entry is 2.8 s for $p = 1024$ and 80 ms for $p = 8$.[3] Consistent with the discussion in Section 3.4, a choice of smaller p introduces less overhead for updating M_{ind}. However, recall that this makes PIR queries slower, since M_{ind} has p rows and $\lceil n/p \rceil$ columns, which is far from square if p is small, introducing large communication overhead and more index items to decrypt on the client side.

Figure 4.10 shows the distribution (CDF) of private write costs with $p = 8$ and $p = 1024$ respectively on a 1 TB database over ADSL connections. We do not observe a great difference between the two curves. In fact, the amortized response time for private writes in our simulation is 70.0 s with $p = 8$ and 63.5 s with $p = 1024$. Without giving a thorough statistical analysis here, we suspect with our preliminary observation that the choice of p does not affect the overall private write performance too much. This is not too counter-intuitive, because updating M_{ind} is much cheaper than updating M_{rec} (amortized).

Figure 4.11 and Figure 4.12 show the distributions (CDF) of private write costs for two different choices of P (number of M_{rec} parts) over a 1 TB database on our local machine and over ADSL connections respectively. We choose P to be 1024 (in the order of $\sqrt{n}$) and 65536 (in the order of $n/\log n$). For both choices of P, we make extensive simulation in order to choose a proper C that is just large enough to avoid part overflows. We observe that for $P = 1024$, $C = 0.75$ never causes any overflow, and that for $P = 65536$, $C = 1.75$ never causes any overflow. The choice of a large P introduces a larger amortized response time but a better worst-case performance. Our simulation results show an amortized response time of 108 s, and a worst-case performance of 937 s for $P = 65536$ over ADSL connections.

Figure 4.13 shows the distributions (CDF) of private write costs on a 1 TB database for two different block sizes on our local machine and over ADSL connections respectively. On our local machine, even though the distributions are quite different, the amortized end-to-end performance, somewhat surprisingly, does not differ too much (176 ms for a block size of 64 KB and 181 ms for a block size of 1 MB). We suspect that the noise from disk access has dominated the advantage of processing smaller blocks in a local setting. In our simulation, the amortized response time of private writes with a block size of 64 KB is 18 s, and the worst-case performance is 22766 s, which outperforms the case of 1 MB block size.

[3]We run 6400 trials of simulation to make sure that they touch about 100 M_{ind} reshuffles for $p = 8$.

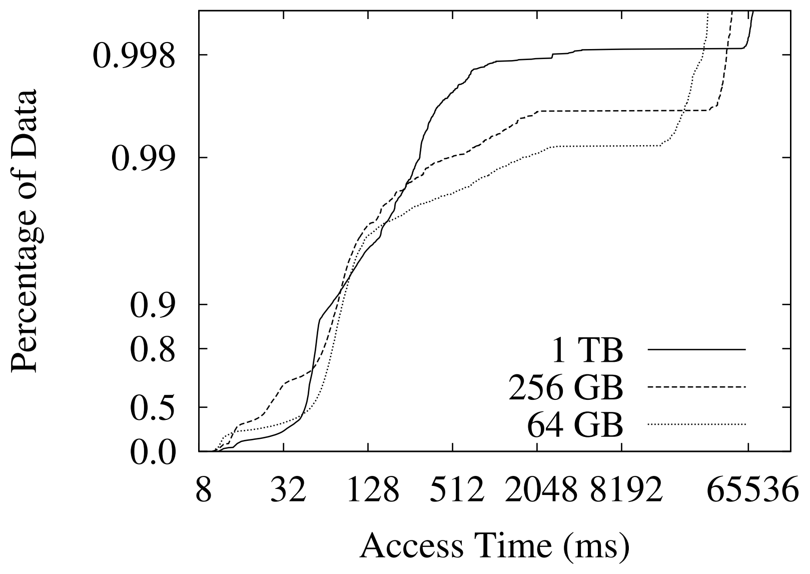

Figure 4.7: Construction Two: Distribution (CDF) of private write costs for databases of different sizes on our local machine.

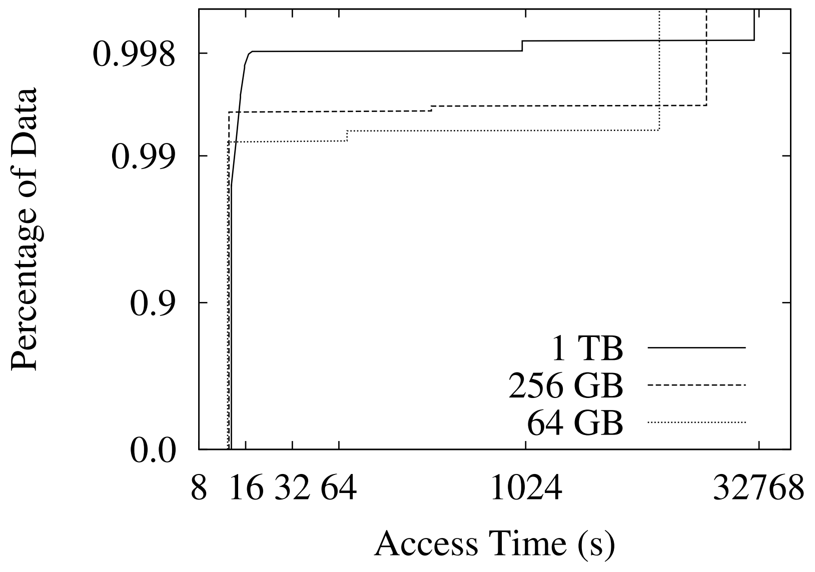

Figure 4.8: Construction Two: Simulated distribution (CDF) of private write costs for databases of different sizes over ADSL connections.

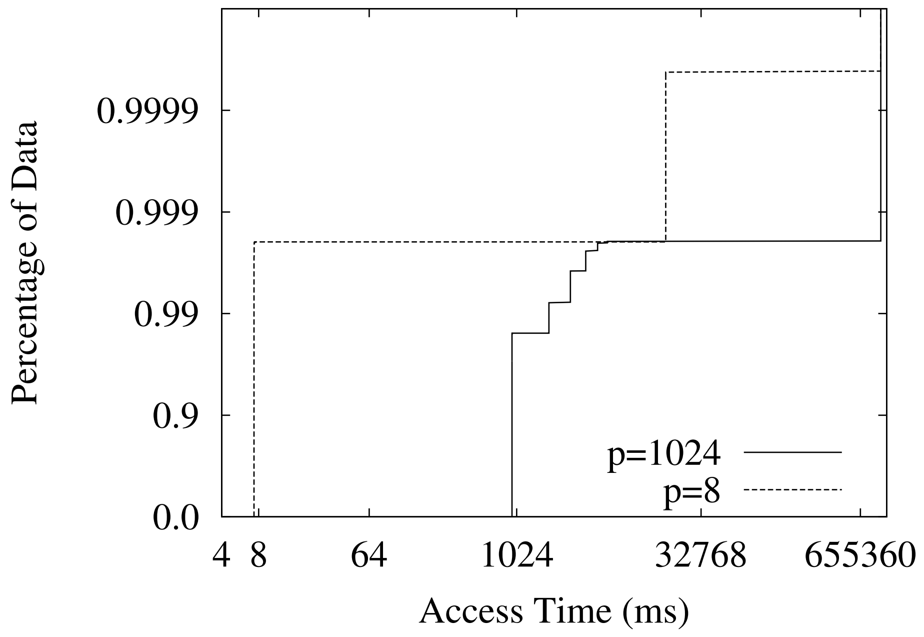

Figure 4.9: Construction Two: Simulated distribution (CDF) of index entry update costs with different p (number of M_{ind} parts) over ADSL connections. The underlying database size is 1 TB.

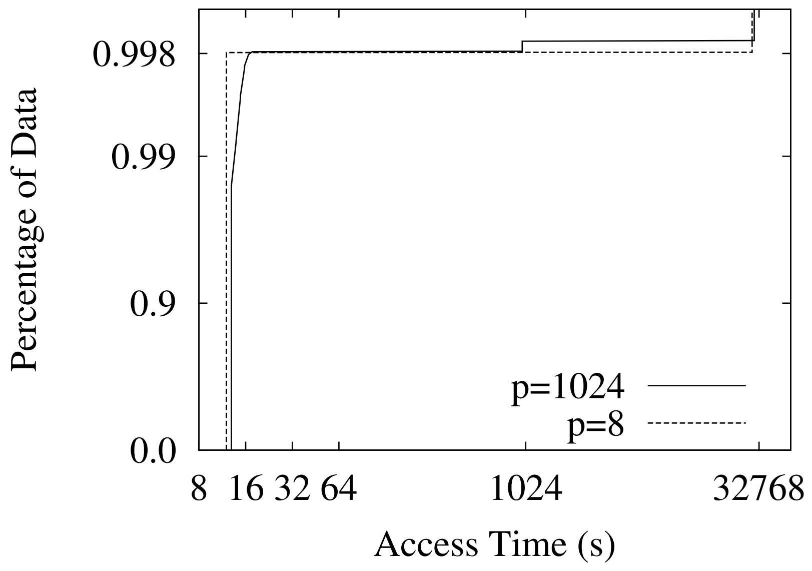

Figure 4.10: Construction Two: Simulated distribution (CDF) of private write costs with different p (number of M_{ind} parts) over ADSL connections. The underlying database size is 1 TB.

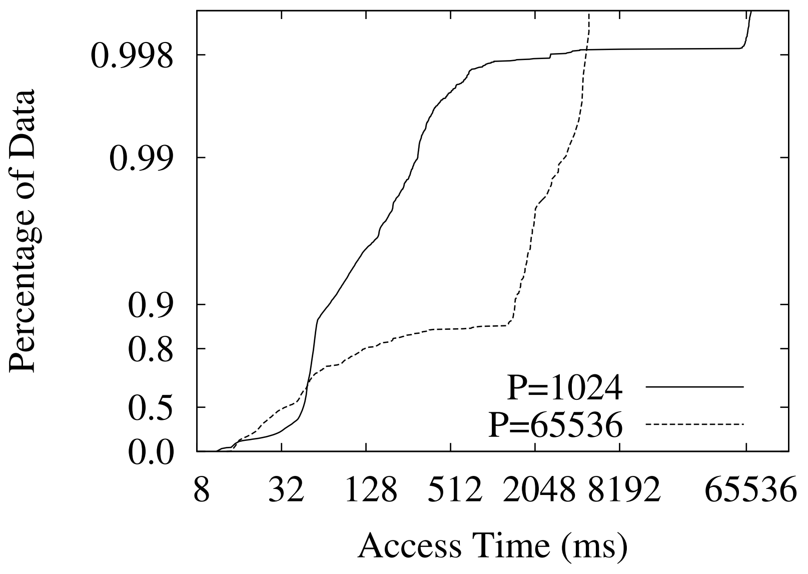

Figure 4.11: Distribution (CDF) of private write costs with different choices of P (number M_{rec} parts) on our local machine. The underlying database size is 1 TB.

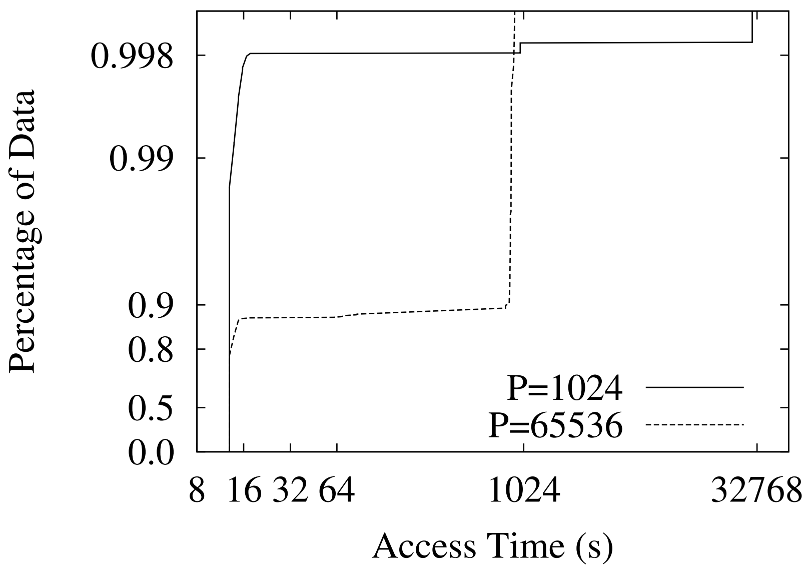

Figure 4.12: Simulated distribution (CDF) of private write costs with different choices of P (number of M_{rec} parts) over ADSL connections. The underlying database size is 1 TB.

Part size	Amortized cost	Worst-case cost
$\lceil\sqrt{n}\rceil$	About 1 minute	8 to 9 hours
$\lceil\log n\rceil$	About 2 minutes	About 15 minutes

Table 4.1: Construction Two: The amortized cost and worst-case cost of our system with different part sizes for M_{rec}.

This result indicates that our system delivers acceptable performance even over databases with small record sizes.

To conclude, Construction Two is indeed much faster than Construction One. Even the worst-case performance for private writes over a 1 TB database is controlled under 10 s with good parameter choices ($P = 65536$) on our local machine. Furthermore, our simulations have also shown a promising result that even over slow ADSL connections, with a logarithmic part size ($P = 65536$), the amortized response time for private writes on a 1 TB database is less than a couple of minutes, and the worst-case performance is about 15 minutes. With a square root part size ($P = 1024$), the response time of private writes on a 1 TB database is about one minute amortized, and between 8 to 9 hours in the worst case over ADSL connections. If updates to the database are infrequent, even this can be acceptable. Table 4.1 shows how different choices of part size affect the performance of our system.

4.4 Implementation Details

We implemented both of our constructions in C++. All of our implementations are available online at `https://crysp.uwaterloo.ca/software/`.

4.4.1 Oblivious Reshuffle

Recall that it is required that levels in our Oblivious RAM are reshuffled periodically. More specifically, after each 2^{i-1} accesses to the ORAM, all the items from level 1 to level $i-1$ are obliviously reshuffled into level i along with items already on level i. Such an oblivious reshuffle is composed of three steps: an oblivious labeling of phantom items, an oblivious removal of dummy items, and an oblivious merge sort on the real/phantom items, as described below in further detail.

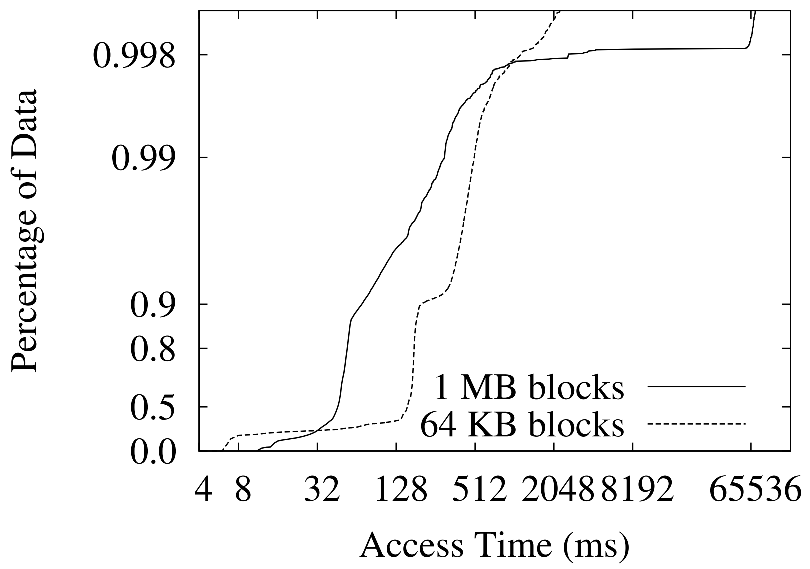

Figure 4.13: Distribution (CDF) of private write costs on our local machine with different choices of block size (of M_{rec}) on our local machine. The underlying database size is 1 TB.

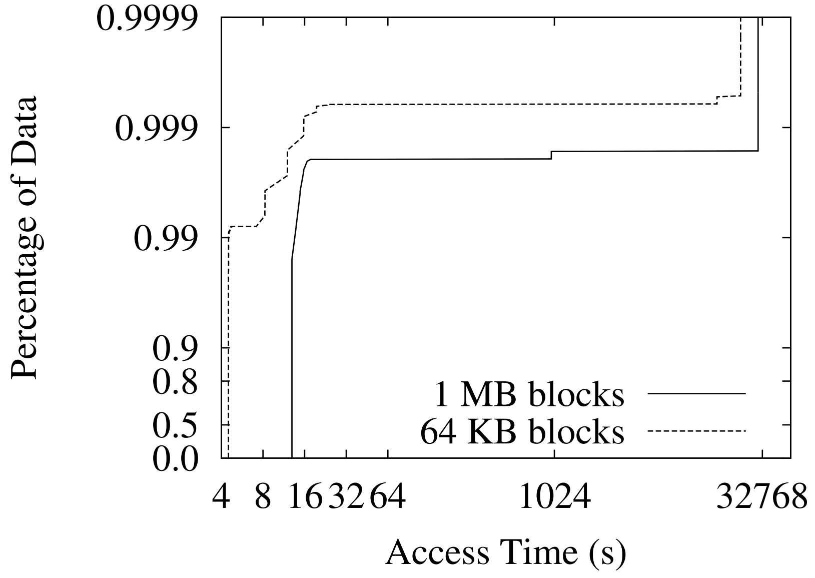

Figure 4.14: Distribution (CDF) of private write costs with different choices of block size (of M_{rec}) over ADSL connections. The underlying database size is 1 TB.

Step 1: Oblivious Labeling of Phantom Items. Our oblivious removal protocol is based on Williams et al.'s "Remove Fakes" protocol [WS08]. In order for this oblivious removal protocol to work (e.g. buffer on private storage does not overflow or underflow), the ratio of dummy items versus non-dummy items should be fixed to be roughly the same on each level that has a large number of blocks. In our ORAM scheme, it is possible that on some levels all the blocks contain dummy items, while on some other levels (e.g. the lowest level) only around half of the items are dummy. We label some dummy items to be phantom items, such that they will *not* be removed as dummy items during the oblivious removal protocol (although they are also random items as dummy items are). In this way, we are able to fix the dummy versus non-dummy items ratio to about 1:1 on each level. A position map is maintained in private storage, which keeps track of the *id* of the item stored in each block of the Oblivious RAM. A phantom item is labeled with a negative *id*, and thus could be easily identified and unlabeled after the oblivious reshuffle. Note that all the labeling and unlabeling operations do not need to touch the untrusted storage at all. See Algorithm 2 for the details of this step.[4]

Step 2: Oblivious Removal of Dummy Items. After Step 1, we employ Williams et al.'s "Remove Fakes" protocol to remove all the dummy items from level 1 to level i. A buffer in private storage is required to store non-dummy items read from the ORAM, with its size denoted by bs. We read the first bs consecutive items from ORAM, after which the local buffer is expected to be half full. We then read two blocks from ORAM and write one item back from the local buffer until we have touched the last block on level i.[5] With roughly equal probability, the number of items in the local buffer either increases or decreases. Such a random walk causes the buffer to overflow or empty with negligible probability, given a proper choice of buffer size on the order of $\sqrt{n}$ [WS08]. In the end, we write all the remaining items in the local buffer back to the ORAM.

Step 3: Oblivious Merge Sort. After Step 2, we employ Williams et al.'s "Oblivious Merge Sort" [WS08] to obliviously reorder the real items mixed with phantom items. The resulting sequence of real/phantom items are used to fill level i, after which the ORAM servers generate random blocks to fill all the levels above i. All the phantom items are then unlabeled in private storage.

[4]In the real implementation, real items' *id* ranges from 0 to $n-1$, and the level number ranges from 0 to $L-1$.

[5]We can scan the ORAM blocks starting from the end to achieve an in-place implementation that does not require extra storage.

```
Function    : LabelPhantomItemsOnLevel
Parameters: level, level_to_shuffle_into, target_num_non_dummy_items, current_label
begin
    level_offset ← NumBlocksAbove(level);
    num_non_dummy_items ← 0;
    for $i$ ← level_offset to level_offset + NumBlocksOn(level) do
        if PositionMap($i$) == DummyItemID then
            AddToList(list_of_dummy_item_indices, $i$);
        else
            num_non_dummy_items ← num_non_dummy_items + 1;
        endif
    endfor
    ShuffleList(list_of_dummy_item_indices);
    for $i$ ← 0 to target_num_non_dummy_items −num_non_dummy_items do
        index ← list_of_dummy_item_indices [$i$];
        while current_label < MaxRealItemID && LookupLevel(current_label) ≤
        level_to_shuffle_into do
            current_label ← current_label + 1;
        endw
        SetPositionMap(index, 0 − current_label);
        current_label ← current_label + 1;
    endfor
end
```

Algorithm 1: LabelPhantomItemsOnLevel

```
Function    : LabelPhantomItems
Parameters: level_to_shuffle_into

begin
    current_label ← 1;
    if first_level_is_full == true then
        LabelPhantomItemsOnLevel(1, level_to_shuffle_into, 2, current_label);
    else
        LabelPhantomItemsOnLevel(1, level_to_shuffle_into, 1, current_label);
    endif
    for $i \leftarrow 2$ to level_to_shuffle_into − 1 do
        LabelPhantomItemsOnLevel($i$, level_to_shuffle_into, NumBlocksOn($i$) /2,
        current_label);
    endfor
    if first_level_is_full == true then
        LabelPhantomItemsOnLevel(level_to_shuffle_into, level_to_shuffle_into,
        NumBlocksOn(level_to_shuffle_into) /2, current_label);
    else
        LabelPhantomItemsOnLevel(level_to_shuffle_into, level_to_shuffle_into,
        NumBlocksOn(level_to_shuffle_into) /2 + 1, current_label);
    endif
end
```
Algorithm 2: LabelPhantomItems

Chapter 5

Conclusion

We construct a protocol that allows one database owner to privately read from and write to a database, and multiple clients to privately read from the database. The access patterns of updates are completely hidden from parties who are not entitled to read those records, and the read histories of any user are completely hidden from all parties other than that user, under a standard non-collusion assumption and common cryptographic assumptions. The direct application of our protocol is in outsourcing Private Information Retrieval to untrusted cloud servers with access control and pricing. We implement and measure a real system that shows the practicality of our work for a 2 GB database. We estimate that for a terabyte-sized database with one-megabyte records, a private read can be served over the Internet in the order of seconds with moderate cloud computing power, and that a private write from the database owner over a high-speed network (e.g. 100 Mb/s) incurs an amortized response time of about one minute. We propose an improved construction based on the observation that the database owner can rely on PIR queries to retrieve records. We simulate the performance of our system based on this construction, which updates a one-megabyte record on a one-terabyte database in about one minute on average, even when the database owner only has slow ADSL connections. Future work includes de-amortizing private writes and examining the non-collusion assumption from the perspective of game theory.

References

[Bat68] Ken E. Batcher. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference*, 32:307–314, 1968.

[BMP11] Dan Boneh, David Mazieres, and Raluca Ada Popa. Remote oblivious storage: Making oblivious RAM practical. Technical report MIT-CSAIL-TR-2011-018, MIT, March 2011.

[CCC$^+$09] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily Shen, Alan T. Sherman, and Poorvi L. Vora. Scantegrity II: End-to-End Verifiability by Voters of Optical Scan Elections Through Confirmation Codes. *IEEE Transactions on Information Forensics and Security*, 4(4):611–627, Dec 2009.

[CDN09] Jan Camenisch, Maria Dubovitskaya, and Gregory Neven. Oblivious Transfer with Access Control. In *Proceedings of ACM CCS 2009*, pages 131–140, Chicago, Illinois, Nov 2009.

[CDN10] Jan Camenisch, Maria Dubovitskaya, and Gregory Neven. Unlinkable Priced Oblivious Transfer with Rechargeable Wallets. In *Proceedings of FC 2010*, pages 66–81, Jan 2010.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, Nov 1998.

[Dev13] Casey Devet. Evaluating Private Information Retrieval on the Cloud. Technical Report 2013-05, CACR, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-05.pdf.

[DGH12] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally Robust Private Information Retrieval. In *Proceedings of the 21st USENIX Security Symposium*, Bellvue, WA, August 2012.

[DMS04] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the 12th USENIX Security Symposium*, pages 303–320, San Diego, California, Aug 2004.

[FCS+11] Martin Franz, Bogdan Carbunar, Radu Sion, Stefan Katzenbeisser, Miroslava Sotakova, Peter Williams, and Andreas Peter. Oblivious outsourced storage with delegation. In *Proceedings of FC 2011*, pages 127–140, St. Lucia, Feb-Mar 2011.

[GDHH12] Ian Goldberg, Casey Devet, Paul Hendry, and Ryan Henry. Percy++. http://percy.sourceforget.net/, 2012. Accessed January 2013.

[GGM98] Yael Gertner, Shafi Goldwasser, and Tal Malkin. A Random Server Model for Private Information Retrieval or How to Achieve Information Theoretic PIR Avoiding Database Replication. In *Proceedings of RANDOM 1998*, pages 200–217, Barcelona, Spain, Oct 1998.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. *Automata, Languages and Programming*, pages 576–587, 2011.

[GMOT11] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop*, pages 95–100, Chicago, Illinois, Oct 2011.

[GMOT12] Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–167, Kyoto, Japan, Jan 2012.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol07] Ian Goldberg. Improving the Robustness of Private Information Retrieval. In *Proceedings of IEEE S&P 2007*, pages 131–148, Oakland, California, May 2007.

[HHG13] Ryan Henry, Yizhou Huang, and Ian Goldberg. One (Block) Size Fits All: PIR and SPIR with Variable-Length Records via Multi-Block Queries. In *Proceedings of NDSS 2013*, San Diego, Feb 2013.

[HOG11] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for Electronic Commerce. In *Proceedings of ACM CCS 2011*, pages 677–690, Chicago, Illinois, Oct 2011.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 143–156, Jan 2012.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-Size Commitments to Polynomials and Their Applications. In *Proceedings of ASIACRYPT 2010*, pages 177–194, Dec 2010.

[OG11] Femi Olumofin and Ian Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Proceedings of FC 2011*, pages 158–172, Feb 2011.

[RS98] Martin Raab and Angelika Steger. Balls into bins, a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, pages 159–170, 1998.

[SCSL11] Elaine Shi, Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proceedings of ASIACRYPT 2011*, pages 197–214, Seoul, South Korea, Dec 2011.

[Sha79] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, Nov 1979.

[SSS12] Emil Stefanov, Elaine Shi, and Dawn Song. Towards practical oblivious RAM. In *Proceedings of NDSS 2012*, San Diego, California, Feb 2012.

[WDDB06] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. Private information retrieval using trusted hardware. In *Proceedings of ESORICS 2006*, pages 49–64, 2006.

[WS08] Peter Williams and Radu Sion. Usable PIR. In *Proceedings of NDSS 2008*, San Diego, California, Feb 2008.

[WST12] Peter Williams, Radu Sion, and Alin Tomescu. PrivateFS: a parallel oblivious file system. In *Proceedings of ACM CCS 2012*, pages 977–988, Raleigh, North Carolina, Oct 2012.