Outsourced Private Information Retrieval*

Yizhou Huang
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
y226huang@cs.uwaterloo.ca

Ian Goldberg
Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
iang@cs.uwaterloo.ca

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. Our system is compatible with existing PIR access control and pricing schemes 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. We make a second 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 the two observations, we construct an end-to-end system that privately updates a 1 MB record in a 1 TB database with an amortized end-to-end response time as low as 300 ms when the database owner has a fast network connection to the database servers, and about 1 minute over a slow ADSL connection. Private read times by the database readers are on the order of seconds in either case.

Categories and Subject Descriptors

H.2.0 [**General**]: Security, integrity, and protection; H.2.4 [**Database Management**]: Systems—*Distributed databases, Query processing*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*; K.4.4 [**Computers and Society**]: Electronic Commerce—*Security, Payment schemes*

Keywords

Privacy Enhancing Technologies; Private Information Retrieval; Outsourcing Computation; Oblivious RAM

*An extended version of this paper is available. [17]

WPES'13, November 4, 2013, Berlin, Germany.

ACM 978-1-4503-2485-4/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517840.2517854.

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.

Information-theoretic PIR (IT-PIR) schemes are "perfectly secure" in the same sense as above—even a server employing unlimited computation cannot determine what the client was after. On the contrary is computational PIR (CPIR), whose security relies on certain cryptographic assumptions. However, in order to achieve sublinear communication and information-theoretic security at the same time, one must employ multiple database servers [5], 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 [8] and electronic voting [4] make the same assumption.

In work from 2011, Olumofin and Goldberg [18] 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 [18], when the size of the database exceeds the size of the RAM available, 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 [6] has shown that with the help of 64 cores, Goldberg's IT-PIR protocol [10] 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.

The obvious way to provide *outsourcing privacy* is to let the database owner re-encrypt the entire database and upload it for every single update; this is of course quite impractical, however, for reasonably large databases.

1.2 An overview 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 require ℓ servers because our protocol uses multi-server IT-PIR schemes.

In a nutshell, O wishes to provide PIR services to database clients, but she does not have enough computational resources available to compute PIR queries quickly over large databases (e.g. a 1 TB database). Thus, she decides to store her database, which is an array of n records with equal lengths,[1] on ℓ servers (in some encrypted form); these servers can perform the PIR computations while O is completely offline. She can also update her records without revealing the access pattern; thus, O is considered to be a "writer".

Database clients wish to read records from the database. PIR allows them to do so without revealing the read access patterns to either O or the ℓ servers. These clients are considered to be "readers".

If anyone is allowed to read any record, then it is not very interesting to construct a protocol that hides the write access patterns, because anyone can easily find out which records are updated and when by issuing read queries. However, realistically, O might want clients to *pay* for records before they are allowed to access them, or she might want to enforce some *access control* policies over the records. In such cases, our protocol guarantees neither the readers nor the database servers learn the write access patterns of records they are not entitled to access.

[1]More complex database layouts can be easily layered on top of this simple primitive.

Our contributions

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. The system meets our definition of outsourcing privacy, and pricing and access control can be enforced on a per-record basis using existing schemes.

3. We propose a concrete efficient construction, 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, showing the feasibility of outsourcing PIR computations over a 1 TB database, even if the database owner only has a slow network such as ADSL.

2. RELATED WORK

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [12], provides a solution for outsourcing storage to an untrusted server. With a reasonable amount of *private storage* on the client side, ORAM has been shown [15, 20, 23] to be much more efficient than when it was first proposed [12]. ORAM allows a single user (who possesses a secret key) to read and write data to a database housed on an untrusted storage server, and completely hides the access patterns of records from the server. 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.

However, 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 to allow 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 leads to a computational PIR scheme with trusted hardware [21,22]. 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. [9]. Each record in the Oblivious RAM is encrypted and signed by a unique set of keys initially only known by the database

	Multiple Readers	Multiple Writers	Avoids Trusted Hardware	Hides Access from DB Owner
ORAM [12]	×	×	✓	×
ORAM-aided PIR [21, 22]	✓	✓	×	✓
Delegated ORAM [9]	✓	✓	✓	×
This work	✓	×	✓	✓

Table 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.

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 from 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 shows how our protocol is different from those above.

3. BACKGROUND

3.1 Oblivious RAM

Oblivious RAM (ORAM) was first studied by Goldreich and Ostrovsky [12]. 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. They gave two constructions for ORAM, a *Square Root Solution* and a *Hierarchy Solution*, achieving for each ORAM access amortized costs of $\Theta(\sqrt{n} \cdot \log^2 n)$ and $\Theta(\log^3 T)$ respectively [12].

Although the asymptotic costs look reasonable [12], an unrealistically large constant is hidden behind the big Θ notation because of the expensive oblivious sort required to reshuffle the ORAM periodically, with the limited $O(1)$ trusted storage on the CPU. 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 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.

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. Larger client-side storage allows more efficient oblivious sorts [13–15, 23] and makes it possible to keep some index structure on the client side [20]. In reality, 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. It has been shown by previous works [15, 20, 23] that ORAM can be practical in this data-outsourcing scenario.

3.2 Goldberg's IT-PIR

Our construction builds on top of Goldberg's multi-server IT-PIR protocol [10]. We choose Goldberg's IT-PIR for the following two reasons: 1) it has an open-source implementation Percy++ [11]; 2) it is experimentally quite efficient [18].

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 (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. [7] 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.

3.3 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 does not have SPIR's constraint of sublinear communication cost, and so 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 one flavor of such constructions below.

In Camenisch et al.'s OT construction [2, 3], 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 blind signature on the desired index. Since the signature is blind, the signer does not learn the message to be signed, which is the index of the record. In order to enforce access control (AOT [2]) or pricing (POT [3]), 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. As observed by Aiello et al. [1], Camenisch et al.'s OT construction can be easily turned into SPIR.

3.4 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.[2] We now provide one solution to this problem.

Threshold signature.

In Camenisch et al.'s OT construction [2, 3], 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 $K_i = \sigma^{1/k} = e(E_i, h)$ which is then the decryption key.

We can prevent the servers from learning K_i 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 identically 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 K_i for records they are entitled to access with the above threshold signature scheme. If there is no coalition of servers exceeding some threshold τ, none of the servers learns any nontrivial information about M_{key} by hosting E_i's. 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 be 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 this paper.

[2]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.

Note that if more than t servers collude, they can learn database clients' queries, while if more than τ servers collude, they can discover the database owner's updates and the contents of the database. Database clients must contact at least $\max(\tau, t) + 1$ servers to perform a private read (at least $\tau + 1$ to learn K_i through the threshold signature scheme, and at least $t + 1$ to fetch the encrypted record through the PIR scheme). It is not required, but not unreasonable, to set $\tau = t$.

4. CONSTRUCTION

We now describe the construction of our scheme. We denote by *private storage* the storage local to the database owner O, and the number of records by n. Each of these records is associated with a unique *id* ranging from 1 to n.

4.1 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.

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.

There is no communication between different clients in the protocol, so we do not model a privacy notion among different clients. Our protocol does not afford a path for a client to learn other clients' secrets.

It is easy to see that *information retrieval privacy* is guaranteed by the properties of PIR and SPIR; our contribution is in additionally achieving outsourcing privacy in an efficient construction that supports multiple clients. We present a formal definition for *outsourcing privacy* and a more thorough security analysis for our construction in Appendices A and B. Note that our protocol does reveal the overall frequency of updates to the database, but not to individual records.

4.2 Overview

Our system stores three matrices (databases) 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}. The second matrix, M_{ind}, stores the encrypted index that keeps 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.

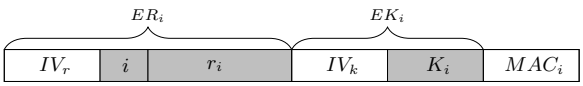


Figure 1: The layout of a data item. The light grey parts are encrypted.

M_{rec} and M_{ind} are replicated across each of the ℓ clouds, while M_{key} is distributed using the data privacy technique from Section 3.4 so that no coalition of τ or fewer cloud providers can read the contents of M_{key}.

The database 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 1): the encrypted content ER_i of the underlying 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 IV IV, and $MAC_i = MAC_{K_i}(i \| EK_i \| ER_i)$,[3] where r_i is the content of the record with *id* i. 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 of the updated record 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 4.3.

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

1. A PIR query on M_{ind} for the offset $OFFSET_i$ 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 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 Camenisch et al.'s ACOT or POT [2, 3]. 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. [16], clients can issue PIR queries to retrieve the part of the encrypted database they are interested in, and then conduct the 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. Given $OFFSET_i$ retrieved from M_{ind} above, the client can determine which row to retrieve from M_{rec}.

[3]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.

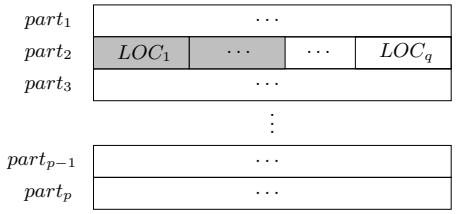


Figure 2: The layout of M_{ind}. Each row $part_i$ is organized as a limited-sized queue with a size limit of $q \approx Q\lceil n/p \rceil$, which stores the indices for records with *id* ranging from $(i-1) \cdot \lceil n/p \rceil + 1$ to $i \cdot \lceil n/p \rceil$ (as well as some dummy items). The 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.

4.3 Efficient server-side Index

The size of a full index goes up to the order of gigabytes in a terabyte-sized database. This is a manageable size for private storage, because accessing that index 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 $part_1, \cdots, part_p$. Each of these parts is organized as a queue of constant limited size, and part $part_i$ 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 the 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 2 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

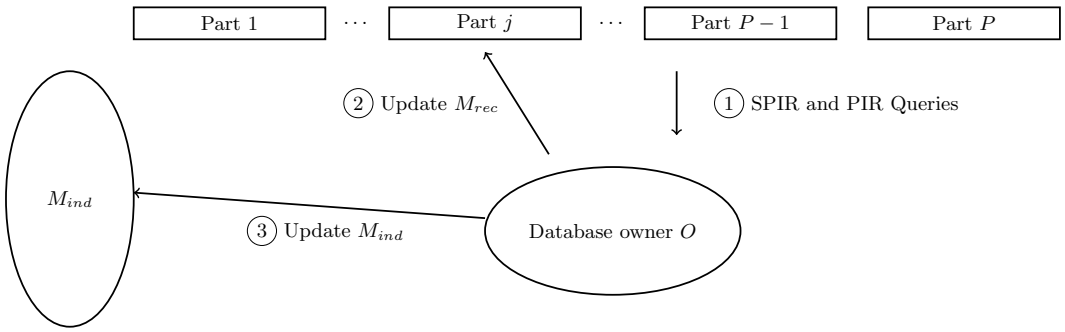


Figure 3: M_{rec} is partitioned into P parts. To update a record, the database owner O first reads it through PIR queries, and then chooses a random part j and a random location i within part j that has not been touched since the last shuffle of part j. The same j's and the same i's are chosen for each server. Part j is reshuffled if it has been accessed $F \cdot \lceil \frac{n}{P} \rceil$ 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 4.3.

index item as a dummy item for all the other part. For this construction to work, we require O to store the entire index locally.

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 entry 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; compare this to the $n \cdot U$ approximate cost for the naive solution from Section 4.2.

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.

The parts should be initialized to different states to avoid the replacing of all parts simultaneously; this affords some measure of de-amortization. When the underlying ORAM storing M_{rec} shuffles the locations of multiple records, we need to update multiple indices in M_{ind}. To avoid leaking access patterns, the database owner should pad the number of updates of M_{ind} to the maximum number that might be produced by the ORAM scheme at that time. 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*.

4.4 Strawman: Standard ORAM + PIR

We have not touched which specific ORAM scheme should be used for a concrete construction. It is not immediately obvious that, by plugging in some arbitrary ORAM scheme, the resulting system would be secure; a case-by-case security analysis is required. In the extended version of this paper [17], we provide a concrete construction using a standard ORAM, and argue that the system preserves both of our privacy constraints using that specific ORAM scheme.

4.5 A more efficient construction: Write-only ORAM + PIR

We present a concrete construction in this section. In our construction, we modify the standard ORAM notion, which allows both private reads and private writes, into a write-only ORAM model. Before getting to the details, we consider the observation below.

The vector-matrix multiplication part of Goldberg's IT-PIR scheme [10] is easily parallelizable. By utilizing highly parallelized computation resources, Devet's experimental results [6] 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 databases given enough cores, while the speed of ORAM access is always somewhat limited by the available local computation resources.

Inspired by this observation, we propose a write-only Oblivious RAM model 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 (if she does not hold copies of the keys K_i locally) and two PIR queries as described in Section 4.2, and writes the updated record back to M_{rec} in an oblivious manner (described later in this section). Since we do not require M_{rec} to support oblivious *read* operations through the standard ORAM protocol (as the read operations are protected by the PIR protocol), M_{rec} can be organized into a "write-only" ORAM 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 that of Stefanov et al.'s ORAM [20]. However, their ORAM caches up-to-date records in O's private storage. We must avoid this in our setting, 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

Record size	Database size: 64 GB	256 GB	1 TB
4 KB	192 MB	768 MB	3 GB
64 KB	12 MB	48 MB	192 MB
1 MB	768 KB	3 MB	12 MB
8 MB	96 KB	384 KB	1.5 MB

Table 2: Size of private storage required for private index and unvisited locations, with $\sqrt{n}$ parts and a reshuffling frequency $F = 0.5$. 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 our write-only ORAM on an untrusted server, there is a storage overhead inherently required by the ORAM, which is about $(C+F)$ times in our ORAM scheme. The row headers indicate varying record sizes.

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 single part. The client maintains a *private index*, which 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 $F \cdot \lceil \frac{n}{P} \rceil$ times, that part is reshuffled. (F is a constant that determines the frequency of reshuffle operations.) After updating M_{rec}, O should update M_{ind} entirely or append encrypted updated indices to each part of the efficient server index structure in Section 4.3. 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. Adding a new record to the database can be done by replacing a dummy record with a real one, and deleting a record can be done the other way around. Of course, there will be a maximum number of records limited by the initial parameter choice. Figure 3 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 [19], 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, $B = (1 + C + F)\lceil \frac{n}{P} \rceil$, where C is a constant that controls the failure probability. A larger C introduces a smaller probability of part overflow. 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 to 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 5, we set $P = \sqrt{n}$, $C = 0.75$ and $F = 0.5$. We observed not a single failure.

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}, with only a one-way trip to each server.

2. Each part fits entirely in private storage, allowing for very efficient oblivious shuffling without having to resort to oblivious sorting networks.

Table 2 lists the amount of private storage required to store the private index and unvisited locations given different record sizes measured in bytes in our experiments when $P = \sqrt{n}$ and $F = 0.5$. We use 64 bits to encode record *id*'s and locations, which should be enough for most applications. Note that each record is encrypted in a block that is slightly larger than the record itself because of the metadata encoded, such as EK_i, IV_k, etc. Table 3 compares some other ORAM schemes with our protocol. The pseudocode for the algorithms appears in Appendix D.

To make sure the database clients have a consistent view of the database, each of the servers should keep an integer version number, initialized to zero and incremented per update. A database client can then check and/or specify the version number of the database to use. We also note that in any event, the Byzantine robustness of the underlying PIR will allow for the recovery of the correct ciphertext when only a small number of servers disagree on the database version.

4.6 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 Camenisch et al.'s POT [3] or ACOT [2].

We are not innovating in the aspect of pricing and access control. These techniques are standard in the literature, and they are treated as a black box in this paper. As discussed in Section 1.2, however, protecting outsourcing privacy would not be of interest unless the database owner wishes to enforce some restrictions on accessing the database.

The cost for zero-knowledge proofs is $O(1)$ in Camenisch et al.'s OT protocol [2, 3]. Shown by Henry et al.'s experimental result [16], over a 16-gigabyte subset of the *Librivox Free Audio Book Collection* (`https://librivox.org/`), the overheads for zero-knowledge proofs are less than 100 ms. This is a promising result, since the zero-knowledge proof overhead is a constant irrelevant to the size of the database.

5. PERFORMANCE EVALUATION

5.1 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++ [11], 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. One client, one database owner and three servers 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. 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.

Scheme	Amortized Computation (Writes)	Amortized Computation (Reads)	Worst-case Computation	Amortized Bandwidth Multiplication	Number of Round Trips	Multiple Readers
Goldreich et al. [12]	$\Theta((\log n)^3)$	$\Theta((\log n)^3)$	$\Theta(n)$	$\Theta((\log n)^3)$	$\Theta(\log n)$	×
Goodrich et al. [15]	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n \log n)$	$\Theta(\log n)$	$\Theta(\log n)$	×
Williams et al. [23]	$\Theta(\log n \log\log n)$	$\Theta(\log n \log\log n)$	$\Theta(\log n \log\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	×
Stefanov et al. [20]	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\sqrt{n})$	$\Theta(\log n)$	$\Theta(1)$	×
This work, on each server for M_{rec}	$\Theta(1)$	$\Theta(n)$	$\Theta(\sqrt{n})$	$\Theta(1)$	$\Theta(1)$	
This work, on O for M_{rec}	$\Theta(\ell)$	$\Theta(\ell\sqrt{n})$	$\Theta(\ell\sqrt{n})$	$\Theta(\ell)$	$\Theta(1)$	
This work, on each server for M_{ind}	$\Theta(\sqrt{n})$	$\Theta(n)$	$\Theta(n)$	$\Theta(\sqrt{n})$	$\Theta(1)$	✓
This work, on O for M_{ind}	$\Theta(\ell\sqrt{n})$	$\Theta(\ell\sqrt{n})$	$\Theta(\ell \cdot n)$	$\Theta(\ell\sqrt{n})$	$\Theta(1)$	

Table 3: A comparison of different ORAM schemes with our work. ℓ is the number of database servers in our work, and n is the number of database items. We assume $P = p = \lceil\sqrt{n}\rceil$. Note that the constants behind the Θs for M_{ind} are much smaller than those for M_{rec}, as index items (tens of bytes) are significantly smaller than data items (which may be 1 MB, for example).

5.2 Read Performance

We are not benchmarking the read operations over large databases, mainly for two reasons. First, the performance of multi-server IT-PIR is well studied in the literature [6, 18], and the dominating overhead of reads comes from retrieving the record from M_{rec}. Second, we do not have enough computational resources available to efficiently handle databases exceeding the size of our RAM (32 GB). One of the points of 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. Thus, we analyze our read performance for large databases below based on prior related works, assuming reasonably large computational resources available in the clouds.

Devet's experiment [6] 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 2.25 TB write-only ORAM) takes about 4 seconds to compute using Chor's IT-PIR [5] option in Percy++, and less than 2 seconds to transmit between a server 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 [10] for a large database (e.g. 1 TB), we argue that 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.

5.3 Write Performance

In addition to benchmarking the system on our local machine, 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. Since our protocol requires, for each update, only a one-way trip to each server, we do not simulate the network latency. In this section, a "private write" does not include first reading the record with PIR queries.

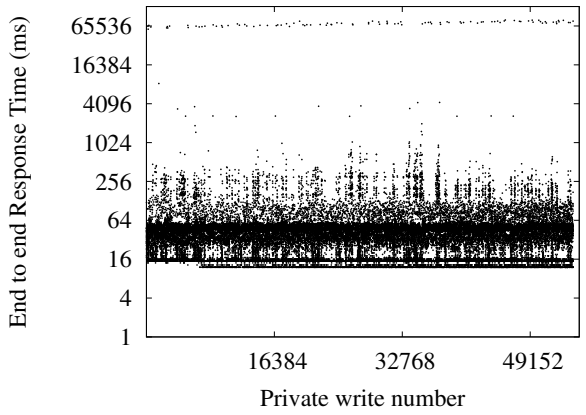


Figure 4: Measured end-to-end response time for each private write to database of size 1 TB (2.25 TB storage required for M_{rec}).

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 $\sqrt{n}$, Q to be 2, and F (parameter for reshuffling frequency) to be 0.5. We will show and discuss later how the choice of P affects the performance of our system. 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 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 corresponds to the cost of reshuffling an M_{rec} part. Another sparse spectrum appears between 64 ms and 4096 ms, corresponding to the cost of replacing one or more M_{ind} parts. All the dots below 64 ms can be safely regarded as updates that do not involve any reshuffling op-

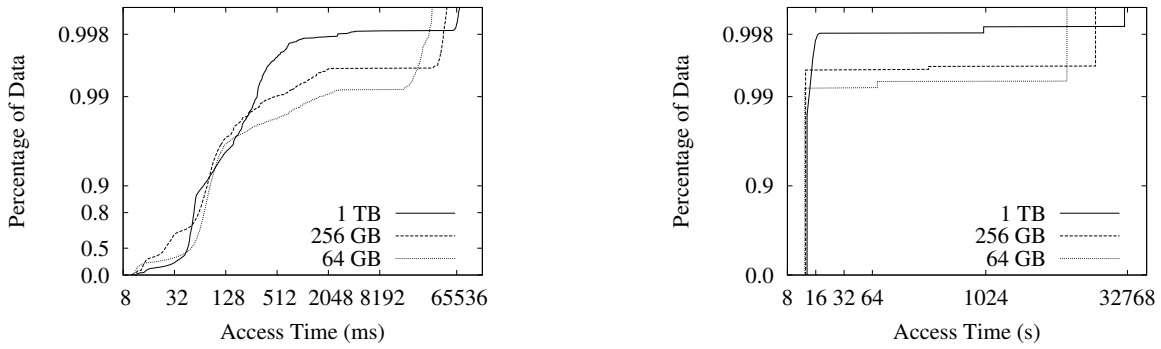


Figure 5: Distribution (CDF) of private write costs for databases of different sizes on our local machine (left) and over ADSL connections (right).

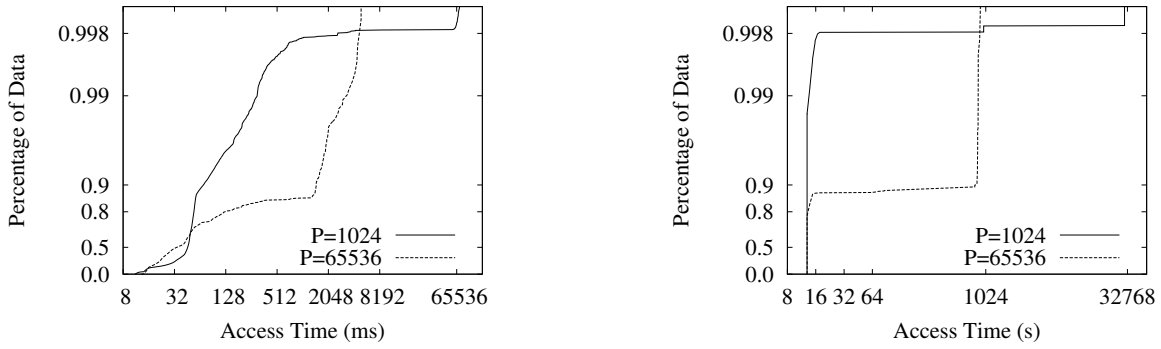


Figure 6: Distribution (CDF) of private write costs with different choices of P (number of M_{rec} parts) on our local machine (left) and over ADSL connections (right). The underlying database size is 1 TB.

erations on M_{rec}; there are various update costs for these operations. We observe in our logs that some of them correspond to different numbers of M_{ind} parts being replaced, and some correspond to different disk access response times. The amortized access overhead is 182 ms.

Figure 5 shows the distribution (CDF) of write costs for databases of different sizes on a local machine and over ADSL connections. On our local machine, 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 on the right figure. The little spike in the middle of each curve in Figure 5 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. Having logarithmic part size can trade off worst-case performance for storage overhead and amortized performance, 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 $\frac{3\times1\times8\,Mb}{2\,Mb/s} = 12\,s$, and our protocol is only about 5 times slower amortized.

Figure 6 shows the distributions (CDF) of private write costs for two different choices of P (the number of M_{rec} parts) over a 1 TB database on our local machine and over ADSL connections. 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.

Since we are organizing a database instead of a filesystem, 1 MB should be a more representative block size than 4 KB, which is a widely chosen parameter in the ORAM literature. We also tested the performance of our system with smaller block sizes; see Appendix C for details. We do not measure the throughput of our system, which we believe is not as important a metric as it would be in a filesystem.

Here are some observations from our experiments. 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, however, even this can be acceptable. Table 4 shows how the choice of P affects the end-to-end performance over ADSL connections. The database owner can of course opt for a high-speed corporate Internet connection (at least 50 times faster in uploading and 10 times faster in downloading than ADSL) to dramatically

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

Table 4: End-to-end performance under different choices of P over ADSL connections.

reduce the end-to-end overhead, since the network transmission is the bottleneck of the private write protocol. As a reference of how high-speed Internet can affect the performance of our system, if all the transmissions take place on a single machine, we achieve an amortized end-to-end response time of 300 ms (see Figures 4 and 5).

6. 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 any 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 1 TB database. For a terabyte-sized database with one-megabyte records, a private read can be served in the order of seconds with moderate cloud computing power, and that a private write from the database owner incurs an amortized response time of about one minute over a slow ADSL connection, or about 300 ms over a fast network.

7. REFERENCES

[1] B. Aiello, Y. Ishai, and O. Reingold. Priced Oblivious Transfer: How to Sell Digital Goods. In *Proceedings of EUROCRYPT 2001*, pages 119–135, May 2001.

[2] J. Camenisch, M. Dubovitskaya, and G. Neven. Oblivious Transfer with Access Control. In *Proceedings of ACM CCS 2009*, pages 131–140, Chicago, Illinois, Nov 2009.

[3] J. Camenisch, M. Dubovitskaya, and G. Neven. Unlinkable Priced Oblivious Transfer with Rechargeable Wallets. In *Proceedings of FC 2010*, pages 66–81, Jan 2010.

[4] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen, A. T. Sherman, and P. 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.

[5] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private Information Retrieval. *Journal of the ACM*, 45(6):965–981, Nov 1998.

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

[7] C. Devet, I. Goldberg, and N. Heninger. Optimally Robust Private Information Retrieval. In *Proceedings of the 21st USENIX Security Symposium*, Bellvue, WA, August 2012.

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

[9] M. Franz, B. Carbunar, R. Sion, S. Katzenbeisser, M. Sotakova, P. Williams, and A. Peter. Oblivious outsourced storage with delegation. In *Proceedings of FC 2011*, pages 127–140, St. Lucia, Feb-Mar 2011.

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

[11] I. Goldberg, C. Devet, P. Hendry, and R. Henry. Percy++. http://percy.sourceforget.net/, 2012. Accessed January 2013.

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

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

[14] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. 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.

[15] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. 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.

[16] R. Henry, Y. Huang, and I. 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.

[17] Y. Huang and I. Goldberg. Outsourced Private Information Retrieval with Pricing and Access Control. Technical Report 2013-11, CACR, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-11.pdf.

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

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

[20] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proceedings of NDSS 2012*, San Diego, California, Feb 2012.

[21] S. Wang, X. Ding, R. H. Deng, and F. Bao. Private information retrieval using trusted hardware. In *Proceedings of ESORICS 2006*, pages 49–64, 2006.

[22] P. Williams and R. Sion. Usable PIR. In *Proceedings of NDSS 2008*, San Diego, California, Feb 2008.

[23] P. Williams, R. Sion, and A. Tomescu. PrivateFS: a parallel oblivious file system. In *Proceedings of ACM CCS 2012*, pages 977–988, Raleigh, North Carolina, Oct 2012.

APPENDIX

A. FORMAL SECURITY DEFINITION

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. The transactions between the database owner and all servers are identical, except when a part is downloaded from only one of the servers for reshuffling, which does not give that server any extra information; therefore, we need 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 index items, and a sequence of locations in M_{rec} and M_{ind} to which these ciphertexts are written.

Recall that a record *id* is "disclosed" to the server if the server knows 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 different 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.

B. SECURITY ANALYSIS

In this section, we provide a reduction from the security of private writes to the security of the underlying encryption scheme as well as the underlying pseudorandom permutation. We assume that all the database servers and the database owner are honest-but-curious. This is a reasonable assumption, since both cloud service providers and the owner of a large database who can afford such outsourcing computations are likely to care about their reputations.

We argue that in such an honest-but-curious setting, the server has only a negligible advantage in winning the *access sequence distinguishing game*.

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 the entire 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 4.3, 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}.

Note that a pseudorandom location (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* is fulfilled in our construction.

One might wonder, however, if the execution of private writes and private reads can be securely composed. We argue below that this is a needless worry in our case.

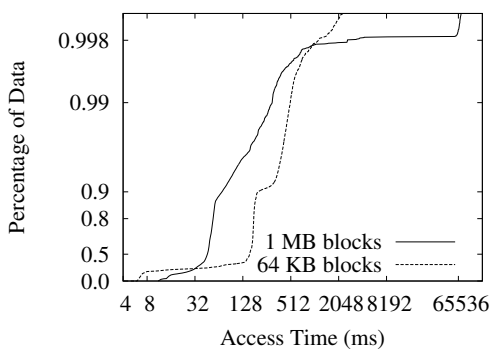
First, private reads should not affect the security of private writes, because the private reads are "read only" operations, which modify neither M_{rec}, M_{ind} nor M_{key}.

On the other hand, the private write protocol should not affect the security of the private reads. Assuming the underlying PIR queries are secure, no matter which records have been updated and when, as long as the chosen servers are in a synchronized state to serve PIR queries, each database sees three random vectors as input and produces three random vectors as output. Also, the write protocol only touches M_{rec} and M_{ind}, which is independent of the generation and verification of the required zero-knowledge proofs.

C. DIFFERENT BLOCK SIZES

We examine how our system performs with a smaller block size. The result indicates that our system delivers acceptable performance even over databases with small record sizes.

Figure 7 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 of the system running over ADSL connections, 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, both outperforming the case of a 1 MB block size.



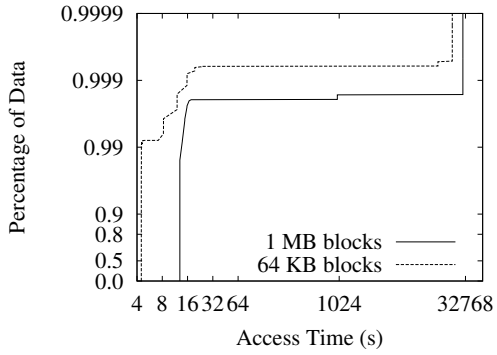


Figure 7: Distribution (CDF) of private write costs with different choices of block size (of M_{rec}) on our local machine (left) and over ADSL connections (right). The underlying database size is 1 TB.

D. PSEUDOCODE

Function : `UpdateMRec`
Parameters: rid,encryption_of_record_rid
begin
  $i \leftarrow$ a random part numbered from 1 to P;
  $j \leftarrow$ first element from d_i;
  // We shuffle elements in d_i according to a
  // pseudorandom permutation every time d_i is
  // re-generated
  remove j from d_i;
  reshuffle_threshold $\leftarrow \lceil \frac{n}{P} \rceil \cdot F$;
  num_access_to_part_i $\leftarrow$ num_access_to_part_i +1;
  write encryption_of_record_rid to the j^{th} location of part i in M_{rec} on all ℓ servers;
  if num_access_to_part_i == reshuffle_threshold **then**
    Download the entire part i from server 1;
    Upload re-encrypted and reshuffled part i to all ℓ servers, generating list_of_updated_records;
    $d_i \leftarrow$ list of locations with dummy items in the new part i;
    Shuffle d_i according to a pseudorandom permutation;
    `UpdateMInd`(list_of_updated_records, B);
    // This is described in Algorithm 2
    num_access_to_part_i $\leftarrow 0$;
  else
    `UpdateMInd`({rid },1);
  endif
end

Algorithm 1: Pseudocode on O for updating record rid in M_{rec}

Function : `UpdateMInd`
Parameters: list_of_updated_records,
num_updated_entries_with_padding
begin
  Update list_of_updated_records to local index;
  reshuffle_threshold $\leftarrow (Q-1) \cdot \lceil n/p \rceil$;
  for $pp \leftarrow 1$ **to** p **do**
    if num_access_to_part_pp + num_updated_entries_with_padding > reshuffle_threshold **then**
      // Replace this part
      Encrypt index items for records $(pp-1) \cdot \lceil n/p \rceil + 1$ to $pp \cdot \lceil n/p \rceil$;
      Replace part pp with the above ciphertexts on all ℓ servers;
      num_access_to_part_pp $\leftarrow 0$;
    else
      num_access_to_part_pp $\leftarrow$ num_access_to_part_pp + num_updated_entries_with_padding;
      for each rid in list_of_updated_records **do**
        if $rid \in [(pp-1) \cdot \lceil n/p \rceil + 1, pp \cdot \lceil n/p \rceil]$ **then**
          Append the encrypted index item for rid to part pp on all ℓ servers;
        else
          Append a dummy index item (random string) to part pp on all ℓ servers;
        endif
      endfor
      for count $\leftarrow 1$ **to** num_updated_entries_with_padding-`Length`(list_of_updated_records) **do**
        Append a dummy index item (random string) to part pp on all ℓ servers;
      endfor
    endif
  endfor
end

Algorithm 2: Pseudocode on O for updating M_{ind}