Outsourced Private Information Retrieval with Pricing and Access Control

Yizhou Huang University of Waterloo

Ian Goldberg University of Waterloo

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. We implement a real system as a proof of concept. Our system privately updates a 1 MB record in a 2 GB database with an average end-to-end overhead of 1.65 seconds and answers a PIR query within 3.5 seconds.

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 [6], 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 [9] and electronic voting [5] make the same assumption.

In work from 2011, Olumofin and Goldberg [22] 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.

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 [22], 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 [7] has shown that with the help of 64 cores, Goldberg's IT-PIR protocol [12] is indeed about 64 times faster than in a single-core setting. This promising re-

	Multiple	Multiple	Avoids	Hides Access History
	Readers	Writers	Trusted Hardware	from DB Owner
ORAM [14]	×	×	✓	×
ORAM-aided PIR [25, 26]	✓	✓	×	✓
Delegated ORAM [10]	✓	✓	✓	×
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.

sult, 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.

1.1 Related Work

Oblivious RAM (ORAM), first proposed by Goldreich and Ostrovsky [14], 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 [17, 24, 27] to be much more efficient than when it was first proposed [14]. 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 leads to a CPIR scheme with trusted hardware [25, 26]. 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. [10]. 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 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.

1.2 Our contributions

- 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 serverside 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.
- 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.

2 Background

2.1 Oblivious RAM

Oblivious RAM (ORAM) was first studied by Goldreich and Ostrovsky [14]. In their model, a CPU with some trusted storage of constant size wishes to conduct a computation in t virtual steps using m 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 is roughly 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 paper that any write operation to an ORAM will use a semantically secure encryption scheme unless otherwise specified.) This trivial solution requires $O(t \cdot m)$ computation 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{m} , as well as a main part that contains m real items and \sqrt{m} 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 cases, the updated virtual item is appended to the shelter in the end. After each \sqrt{m} accesses, the shelter becomes full, and all the items are obliviously reshuffled 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 $O(m \cdot \log^2 m)$ sorting network by Batcher [1], this Square Root Solution achieves an amortized overhead of $O(\sqrt{m} \cdot \log^2 m)$ for each virtual access.

The Hierarchy Solution organizes the ORAM into L levels. Level i contains at most b^i real items for i = $1, \dots, L$, which are hashed to b^i buckets using a hash function unique to that level. 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 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. After b^{i-1} virtual accesses, level (i-1) becomes potentially full, 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 $O(t \cdot \log^2 t)$, and the amortized cost for each virtual access is $O(\log^3 t)$.

Reasonable asymptotic costs are achieved in both of their constructions [14]. However, an unrealistically large constant is hidden behind the big *O* notation because of the expensive oblivious sort required to reshuffle the ORAM periodically. For this reason, ORAM has long been considered as 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

 $^{^{1}}$ 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.

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. 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. [15-17] propose several ORAM schemes with $O(\log n)$ amortized access overhead and $O(n^{1/r})$ storage on the client side for some constant r > 1. Stefanov et al. [24] suggest keeping track of all the records on the client side, because the size of a data item is much larger than its 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 $O(\log n)$. 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. [27] 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 $O(n \log n)$ to O(n). An efficient oblivious merge sort with $O(\sqrt{n})$ 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 key 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 dataoutsourcing 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 multiserver IT-PIR protocol [12]. 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++ [13]; and 3) it is experimentally quite efficient [22].

Goldberg's construction models the database as a rby-s matrix M over some finite field \mathbb{F} . Let e_i be a standard basis vector in \mathbb{F}^r with the j-th entry being 1, so that $e_i \cdot M$ yields exactly the i^{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 ith 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. [8] 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 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 two flavors of such constructions below.

Henry et al.'s SPIR Built on top of Goldberg's IT-PIR protocol [12] and Kate et al.'s polynomial commitments [20], Henry et al.'s SPIR protocol [19] 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 zeroknowledge 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 zeroknowledge 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 [19].

Camenisch et al.'s OT In Camenisch et al.'s OT construction [3, 4], 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 [3]) or pricing (POT [4]), access control or pricing information is encoded in the signature as well. The client proves that the blinded message is wellformed, and that her credential satisfies the access control policy specified in that blinded message. We refer readers to the papers [3,4] 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.² We now provide two solutions to this problem.

 τ -independence The notion of τ -independence was introduced by Gertner et al. [11]. It ensures that a coalition of τ or fewer servers can deduce nothing nontrivial about *the contents of the database*. This feature is supported by Henry et al.'s SPIR protocol [18]. 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, no one learns the contents of M_{key} .

Threshold signature As above, in Camenisch et al.'s OT construction [3,4], the encryption key of a record is an unique signature on its index. In a nutshell, the client blinds a message m by raising it to a random power k. The server signs the blinded message m^k using a secret key h by computing $L = e(m^k, h)$ where e is a bilinear pairing. The client then computes $K = L^{1/k} = e(m, h)$ 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 s_1, \dots, s_ℓ for the value 1 using Shamir secret sharing with threshold τ . Each server gets a share $h_j = h^{s_j}$ and uses it to compute $L_j = e(m^k, h_j)$. The client then performs Lagrange interpolation in the exponent to recover $L = e(m^k, h)$ with $\tau + 1$ valid responses.

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 our paper.

3 Construction

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.

²If 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.

ROW_1	it em ₁	item ₂		$item_p$		
ROW_2						
ROW_3						
:						
ROW_{R-1}						
ROW_R						

Figure 1: The layout of M_{rec} . Each row places p data items from the underlying ORAM in a level-by-level order, starting from the top level. $p = \lceil N/R \rceil$, 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} .

3.1 Privacy Constraints

We care about the 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.

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.³

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.



Figure 2: The layout of a data item. The light grey parts are encrypted.

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 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.

 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 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 IV IV, and $MAC_i = MAC_{K_i}(i||EK_i||ER_i)$, 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 i the ith 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.

³Note that although Goldberg's IT-PIR protocol is informationtheoretically 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 [19].

⁴For 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.

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 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 [19] or Camenisch et al.'s ACOT or POT [3,4]. 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. [18], 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 in overall achieving 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 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 [19].

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 Choice of ORAM scheme

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. [24], which stores some up-to-date data items in the 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 paper. 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 the completeness of our paper. However, we do not argue that it is the most suitable ORAM scheme for all purposes.

We present an 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 4 times the number of records n.

Table 3 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 slighly larger than the record because of the metadata encoded, such as EK_i , IV_k , etc.

Our ORAM is organized into L levels with 2^i items on the i-th level ($i = 1, \dots, L$), including at least 2^{i-1} dummy items. An exception is the first level, where there are only 2 items, and no dummy items are required. Because of the full index in the private storage, O knows which level the up-to-date record resides in 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 all 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

	items	dummy items	reshuffled	moved down
level 1	2	0	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 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.

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 $n \le 2^{L-1}$), 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 $\Theta(m \cdot \log m)$ oblivious merge sort introduced by Williams and Sion [26]. Table 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 5. Assuming an oblivious sorting scheme of complexity $\Theta(m \cdot \log m)$ is used to sort m 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.

		Database size		
		64 GB	256 GB	1 TB
ize	4 KB	240 MB	1 GB	4 GB
g p	64 KB	12 MB	48 MB	240 MB
Record size	1 MB	704 KB	3 MB	12 MB
Re	8 MB	64 KB	288 KB	1.5 MB

Table 3: Size of local storage required on the database owner \mathcal{O} for the index. The column headers show the number of bytes required if the database were to be stored on \mathcal{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.

3.4 Server-side Index

From Table 3, 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 the 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 reencrypted 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 m parts p_1, \dots, p_m . Each of these partitions is organized as a queue of constant limited size, and partition p_i contains the indices for records with id ranging from $(i-1) \cdot \lceil n/m \rceil + 1$ to $i \cdot \lceil n/m \rceil$ (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 partition.

Each partition 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 $\lceil i/\lceil n/m \rceil \rceil$ from M_{ind} . This will be a partition 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

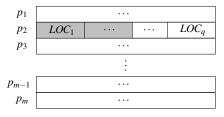


Figure 3: The layout of M_{ind} . Each row p_i is organized as a limited-sized queue with a size limit of $q \approx Q \lceil n/m \rceil$, which stores the indices for records with id ranging from $(i-1) \cdot \lceil n/m \rceil + 1$ to $i \cdot \lceil n/m \rceil$ (as well as some dummy items). The light grey part in p_2 indicates the length of the current queue in p_2 (known only by O). When that queue grows to LOC_q , the entire p_2 needs to be rewritten by O. Q is a parameter that trades off write performance for read performance.

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 shows the layout of M_{ind} .

To update an index in p_j , the database owner appends the updated index item to p_j for the target record index, and appends a random string the same length as an index item as a dummy item for all the other partitions. For this construction to work, we require O to store all the indexes locally, which has been justified in Table 3.

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

With m 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 m 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 2 times storage overhead if we set Q = 2), which can be partitioned into 64 partitions, each with size 1536 KB, which fits easily in any desktop. The cost for updating one index in our efficient M_{ind} construction should be roughly equal to uploading $64 \cdot (1 + \frac{1}{2-1}) \cdot 48 = 6$ 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 KB, and thus the communication cost between the client and one server is then about 1.5 MB for each query. These partitions should be initialized to different states to avoid the replacing of all partitions 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 partition 2^{i-1} times. When the cost of doing so becomes too high, it might just be more efficient to replace the partition 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 [19], or Camenisch et al.'s ACOT [3] and POT [4].

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.

4 Discussion

4.1 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 [2, 16, 21, 23, 24, 27], makes each access to the ORAM bounded by a reasonable overhead.

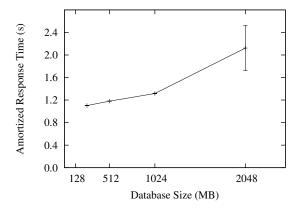


Figure 4: Measured amortized end-to-end response time for private writes to ORAMs of size up to 8 GB (underlying databases being up to 2 GB). Each experimental trial consists of n updates for a database with n records. The large standard deviation for the 8 GB ORAM (2 GB database) is an effect of our testing machine starting to run out of memory with three ORAM servers executing at the same time.

For some of these schemes, de-amortization comes naturally in their construction, such as in the work of Shi et al. [23] and Stefanov et al. [24]. 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 [24] we can move those up-to-date records which are supposed to be stored in the 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 follow a particular paradigm [2, 16, 21, 27]. 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} .

5 Performance Evaluation

As a proof of concept, we implemented an end-to-end system that fulfills our privacy requirements. We require the binaries compiled from Percy++ [13], an open-source implementation of Goldberg's IT-PIR protocol, to make our system work. We used AES-128 for encryption and

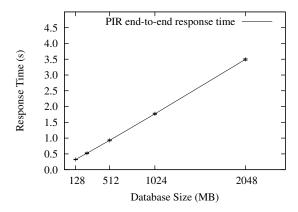


Figure 5: Average end-to-end response time for private reads over ORAMs of different sizes up to 8 GB (underlying databases being up to 2 GB). The deviation is small, implying stable performance. For each parameter choice, we ran the experiment 100 times. The dominating overhead comes from retrieving the record from M_{rec} . For PIR queries over M_{rec} and M_{ind} , Chor's PIR scheme [6] is used for better performance.

128-bit HMAC-MD5 for message authentication codes. In all our experiments, 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 9.10. Figure 4 and Figure 5 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 r_{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 organized into an Oblivious RAM. The ORAM containing the database is about four times as large as the 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 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. For larger databases that do not sit entirely within the memory of a single machine, it requires some engineering efforts to make our system work efficiently, especially in a cloud setting; we leave this for future work. Prior results [7, 27], however, have shown that both Oblivious RAM and Private Information Retrieval are feasible on databases of a terabyte scale.

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 data-

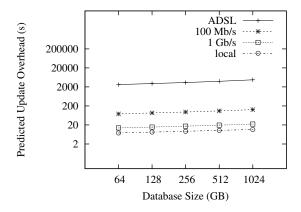


Figure 6: Predicted amortized end-to-end response time for a private 1 MB record update by the database owner for larger databases. We assume that the ADSL connection bears upload and download throughputs of 2 Mb/s and 10 Mb/s respectively, and that the bidirectional throughput of a corporate network 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 [24] with modification) to address this issue, but a careful investigation on the privacy implications and the performance of the new construction is out of the scope of this paper.

Considering an ORAM containing n records organized into L levels where $n = 2^{L-1}$, after n update requests for the purpose of reshuffling, the total number of blocks that need to be downloaded and decrypted is $\sum_{i=2}^{L-1} [2^i \cdot i + (2^{i+1} - 2)] \cdot 2^{L-i-1} + 2^{L+1} - 2 + 2^L \cdot L = (\frac{1}{2}L^2 + \frac{5}{2}L - 1) \cdot 2^{L-1}$, and the total number blocks that needs to be encrypted and uploaded is $\sum_{i=2}^{L-1} (2^i \cdot i + 2^i) \cdot 2^{L-i-1} + 2^L + 2^L \cdot L = (\frac{1}{2}L^2 + \frac{7}{2}L - 2) \cdot 2^{L-1}.$ 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 $\frac{1}{2}L^2 + \frac{7}{2}$ 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 6 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 [7] 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

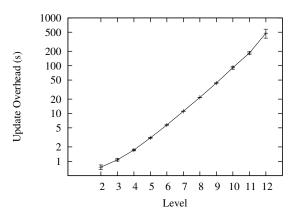


Figure 7: End-to-end response time for one ORAM update request when the given level needs to be shuffled after the request. Level i will be shuffled every 2^{i-1} updates.

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 [6] 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 [12] for the M_{rec} and M_{ind} databases, we observe that τ -independence is not required for them (unlike for M_{kev}). 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 7 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 construction seems much slower than that of Williams et al.'s ORAM scheme [27]. However, note the 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 8 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 issue.

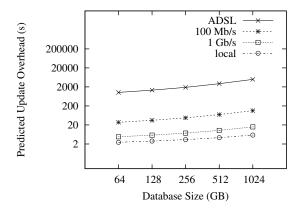


Figure 8: Predicted amortized end-to-end reponse time for a private 256 KB record update for larger databases. We assume that the ADSL connection bears upload and download throughputs of 2 Mb/s and 10 Mb/s respectively, and that the bidirectional throughput of a corporate network is 100 Mb/s.

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 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.

Acknowledgements. We thank NSERC and the Ontario Research Fund for making this work possible.

References

- [1] BATCHER, K. Sorting networks and their applications. *AFIPS Spring Joint Computer Conference 32* (1968), 307–314.
- [2] BONEH, D., MAZIERES, D., AND POPA, R. Remote oblivious storage: Making oblivious RAM practical. Technical report MIT-CSAIL-TR-2011-018, MIT, March 2011.
- [3] CAMENISCH, J., DUBOVITSKAYA, M., AND NEVEN, G. Oblivious Transfer with Access Control. In *Proceedings of ACM CCS* 2009 (Chicago, Illinois, Nov 2009), pp. 131–140.

- [4] CAMENISCH, J., DUBOVITSKAYA, M., AND NEVEN, G. Unlinkable Priced Oblivious Transfer with Rechargeable Wallets. In Proceedings of FC 2010 (Jan 2010), pp. 66–81.
- [5] CHAUM, D., CARBACK, R., CLARK, J., ESSEX, A., POPOVE-NIUC, S., RIVEST, R. L., RYAN, P. Y. A., SHEN, E., SHER-MAN, A. T., AND VORA, P. L. Scantegrity II: End-to-End Verifiability by Voters of Optical Scan Elections Through Confirmation Codes. *IEEE Transactions on Information Forensics and Security* 4, 4 (Dec 2009), 611–627.
- [6] CHOR, B., KUSHILEVITZ, E., GOLDREICH, O., AND SUDAN, M. Private Information Retrieval. *Journal of the ACM 45*, 6 (Nov 1998), 965–981.
- [7] DEVET, C. Evaluating Private Information Retrieval on the Cloud. Tech. Rep. 2013-05, CACR, 2013. http://cacr.uwaterloo.ca/techreports/2013/cacr2013-05.pdf.
- [8] DEVET, C., GOLDBERG, I., AND HENINGER, N. Optimally Robust Private Information Retrieval. In *Proceedings of the 21st* USENIX Security Symposium (Bellvue, WA, August 2012).
- [9] DINGLEDINE, R., MATHEWSON, N., AND SYVERSON, P. F. Tor: The Second-Generation Onion Router. In *Proceedings of the* 12th USENIX Security Symposium (San Diego, California, Aug 2004), pp. 303–320.
- [10] FRANZ, M., WILLIAMS, P., CARBUNAR, B., KATZEN-BEISSER, S., PETER, A., SION, R., AND SOTAKOVA, M. Oblivious outsourced storage with delegation. In *Proceedings of FC* 2011 (St. Lucia, Feb-Mar 2011), pp. 127–140.
- [11] GERTNER, Y., GOLDWASSER, S., AND MALKIN, T. A Random Server Model for Private Information Retrieval or How to Achieve Information Theoretic PIR Avoiding Database Replication. In *Proceedings of RANDOM 1998* (Barcelona, Spain, Oct 1998), pp. 200–217.
- [12] GOLDBERG, I. Improving the Robustness of Private Information Retrieval. In *Proceedings of IEEE S&P 2007* (Oakland, California, May 2007), pp. 131–148.
- [13] GOLDBERG, I., DEVET, C., HENDRY, P., AND HENRY, R. Percy++. http://percy.sourceforget.net/, 2012. Accessed January 2013.
- [14] GOLDREICH, O., AND OSTROVSKY, R. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM) 43*, 3 (1996), 431–473.
- [15] GOODRICH, M., AND MITZENMACHER, M. Privacy-preserving access of outsourced data via oblivious RAM simulation. Automata, Languages and Programming (2011), 576–587.
- [16] GOODRICH, M., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Oblivious RAM simulation with efficient worstcase access overhead. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop* (Chicago, Illinois, Oct 2011), pp. 95–100.
- [17] GOODRICH, M., MITZENMACHER, M., OHRIMENKO, O., AND TAMASSIA, R. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the 23rd Annual* ACM-SIAM Symposium on Discrete Algorithms (Kyoto, Japan, Jan 2012), pp. 157–167.
- [18] HENRY, R., HUANG, Y., AND GOLDBERG, I. 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).
- [19] HENRY, R., OLUMOFIN, F., AND GOLDBERG, I. Practical PIR for Electronic Commerce. In *Proceedings of ACM CCS 2011* (Chicago, Illinois, Oct 2011), pp. 677–690.
- [20] KATE, A., ZAVERUCHA, G. M., AND GOLDBERG, I. Constant-Size Commitments to Polynomials and Their Applications. In Proceedings of ASIACRYPT 2010 (Dec 2010), pp. 177–194.

- [21] KUSHILEVITZ, E., LU, S., AND OSTROVSKY, R. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of the 23rd Annual ACM-SIAM Sympo*sium on Discrete Algorithms (Jan 2012), pp. 143–156.
- [22] OLUMOFIN, F. G., AND GOLDBERG, I. Revisiting the Computational Practicality of Private Information Retrieval. In *Proceedings of FC 2011* (Feb 2011), pp. 158–172.
- [23] SHI, E., CHAN, T., STEFANOV, E., AND LI, M. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *Proceedings of ASIACRYPT 2011* (Seoul, South Korea, Dec 2011), pp. 197–214.
- [24] STEFANOV, E., SHI, E., AND SONG, D. Towards practical oblivious RAM. In *Proceedings of NDSS 2012* (San Diego, California, Feb 2012).
- [25] WANG, S., DING, X., DENG, R., AND BAO, F. Private information retrieval using trusted hardware. In *Proceedings of ESORICS* 2006 (2006), pp. 49–64.
- [26] WILLIAMS, P., AND SION, R. Usable PIR. In Proceedings of NDSS 2008 (San Diego, California, Feb 2008).
- [27] WILLIAMS, P., SION, R., AND TOMESCU, A. PrivateFS: a parallel oblivious file system. In *Proceedings of ACM CCS 2012* (Raleigh, North Carolina, Oct 2012), pp. 977–988.