Path-PIR: Lower Worst-Case Bounds by Combining ORAM and PIR

Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan

College of Computer and Information Science
Northeastern University, Boston MA-02115, USA
{travism,blass,ahchan}@ccs.neu.edu

Abstract. Recent research results on "bucketed" ORAM reduce communication of N-capacity storage with blocks of length l bits down to poly-logarithmic complexity $O(l \cdot \log^3 N)$. The individual buckets, however, are constructed using traditional ORAMs which have worst-case communication complexity being linear in their size. PIR protocols are able to provide better worst-case bounds, but have traditionally been less practical than ORAM due to the fact that they require $O(N)$ computation complexity on the server. This paper presents Path-PIR, a hybrid construction between PIR and ORAM that overcomes the individual weaknesses of each. Path-PIR's main idea is to replace the individual buckets in the ORAM construction by Shi et al. [15] with buckets backed by PIR. We show that this leads to substantially smaller data transfer costs for many databases of practical size and lower worst-case costs, $O(l \cdot \log^2(N) + \log^3(N))$, than the existing construction. Additionally, the typically high computational cost of PIR is offset by the small size of the individual buckets. We also show that Path-PIR has very low latency, i.e., a low amount of data is required before a user receives the result of his data request (less than 10 times the block size). Using Amazon EC2, we demonstrate that monetary cost induced by the server's PIR computation are far outweighed by the savings in data transfer.

1 Introduction

Cloud computing and cloud storage are becoming an attractive option for businesses and governmental organizations in need of scalable and reliable infrastructures. Cloud providers, e.g., Amazon or Google, have substantial expertise and resources, allowing them to rent their services at very competitive prices. Cloud users are drawn by the ability to pay for only what they need, but maintain the ability to scale up if requirements change. Users can now take advantage of highly reliable storage solutions without investing large amounts of money for data centers upfront.

Unfortunately, there is a significant downside to storing data in the cloud. For various reasons, cloud providers cannot always be fully trusted and may not treat sensitive user data very carefully. For example, it has been reported in the real-world that rogue employees or hackers have stolen important data [5, 17].

Encryption of data at rest provides a partial solution to this problem, but it is not sufficient. Even if the cloud (now the "adversary") cannot read the encrypted data, it may be able to learn valuable information based on when and how often a user accesses data. As a motivating example, consider a hospital that outsources their patient records to the cloud in order to save on replication and IT costs. If the adversary sees that, e.g., an oncologist accesses a patient's data, he can learn with some degree of certainty that this person has cancer. An adversary could slowly aggregate information on data accesses to learn potentially important secrets. As it is generally difficult to quantify what external knowledge adversaries may have and what inferences they could make, it is important to hide a user's access pattern as well as the data being accessed.

There are traditionally two ways to hide a user's access pattern, given only a single server: Oblivious RAM (ORAM) [4] and Private Information Retrieval (PIR) [7]. The approach taken by ORAM is to arrange the data in such a way that the user never touches the same piece twice. Through replication and careful structuring, a user can do a number of operations before he is forced to re-"shuffle" the database into a fresh state that allows him to continue accessing data without leaking information. ORAMs feature low amortized communication complexity and do not require any computation on the server, but occasionally the user must download and reshuffle the entire database. This can become impractical in cloud scenarios, especially if the user is a low-powered or communication-constricted device.

Private Information Retrieval, in contrast with ORAM, hides the target of each individual query, independent of all previous queries. This can be accomplished by using a homomorphic encryption which the server uses to operate over the entire database, selecting out the block of data that the user has requested. The user generates requests as ciphertexts to which the server does not know the key, so the server does not learn which block was chosen by the user. Since PIR does not try to hide a sequence of accesses, but each access individually, the amortized cost is equal to the worst-case cost. Unfortunately, the requirement that the server computes over the entire database for each query is often impractical, especially for large databases.

This paper presents Path-PIR, a new hybrid construction combining ORAM and PIR, thereby overcoming the individual drawbacks of each. Path-PIR's rationale is to augment the recently proposed bucketed ORAM ("Path-ORAM") by Shi et al. [15] using PIR. As detailed later, we replace access to individual buckets of the ORAM by PIR queries. As a result, we achieve PIR's better communication worst-case guarantees, while at the same time the general ORAM setup reduces the portion of the database that PIR must compute over. We stress that this paper focuses only on ORAM techniques requiring constant client memory complexity, e.g., contrary to Stefanov et al. [16] which requires at least square-root memory complexity. As we will discuss below, high client memory can be impractical in many real-world situations.

Our **contributions** in this paper can be summarized as follows:

1. Path-PIR, a framework for replacing "bucket" ORAMs in the Shi et al. [15] construction with a PIR-backed bucket.
2. an instantiation of our framework with several different PIR protocols that improve communication costs from $O(l \cdot \log^3(N)$ to $O(\log^3(N) + l \cdot \log^2(N))$ or $O(l \cdot \log^{2.5}(N))$ which is significantly better than related work in many real-world cloud scenarios. Path-PIR requires only constant memory complexity. In addition, using somewhat homomorphic encryption [3, 11] as discussed later, Path-PIR's communication complexity can be reduced further to $O(l \cdot \log N + \log^2(N) \cdot \log\log(N))$.
3. an improvement to Path-PIR which allows for optimal latency (the amount of communication spent before the user has access to the requested data) in retrieving blocks of size $l > \log N$.
4. a real-world implementation of Path-PIR, along with an evaluation performed on Amazon's public EC2 cloud. Our evaluation shows that the additional computation imposed by PIR is outweighed by the significant data transfer savings. We show that Path-PIR allows for *faster* and *cheaper* (under current Amazon AWS pricing [1]) operations than previous constructions.

2 Related Work

There exists a large body of work on improving Oblivious RAM since the original concept by Goldreich and Ostrovsky [4]. For example, Pinkas and Reinman [14] and Boneh et al. [2] have reduced amortized communication to poly-logarithmic complexity. However, even if these constructions feature low amortized cost, worst-case complexity is still $O(N \cdot l)$, which is prohibitive in many scenarios. This is due to the fact that, after a certain number of operations, the entire database needs to be downloaded and reshuffled by the user.

Recently, there have been several approaches that provide better-than-linear worst-case bounds. Kushilevitz et al. [8] achieve this by deamortizing an existing ORAM constructions and obtain $O(l \cdot \log^2(N)/\log\log(N))$ worst-case complexity. Unfortunately, this scheme suffers from very high constants in the order of several thousands and is not efficient for many useful database sizes and parameters in practice.

Shi et al. [15] have proposed another ORAM that has worst case bounds $O(l \cdot \log^3(N))$ using an entirely new construction ("Path-ORAM"). Instead of deamortizing previous schemes, Shi et al. [15] show that a large ORAM can be composed of many smaller "bucket" ORAMs. For each operation, a small fraction of the buckets are shuffled, so there is no need for one large, expensive shuffle of the entire database. Path-ORAM requires only constant user memory by using a recursive memory access technique. Path-PIR augments Path-ORAM, and we will present details later in Section 3.

Stefanov et al. [16] have shown that a bucket-based construction can actually achieve $O(l \cdot \log(N))$ amortized *and* worst-case complexity. However, this is achieved only with either linear user memory complexity or with square-root user memory complexity at the cost of additional communication complexity. Achieving constant user memory is an important requirement, because it allows applications with constrained devices like smart phones and embedded systems. Additionally, the constants that govern user memory are significantly higher with Stefanov et al. [16] than Path-ORAM: for example, a 1 TB database of 1 MB files consumes approximately 800 MB of user memory in the square-root construction of Stefanov et al. [16] which is not available in many situations. On the other hand, Path-ORAM requires only 4 MB (even under linear user memory).

3 Oblivious RAM

Let N denote the capacity, i.e., maximum number of blocks that can be stored in a database $D = \{d_0, \dots, d_{N-1}\}$ at one time. We assume that all blocks are of equal size, and let l denote the size of each block in bits. We assume that $l > c \cdot \log(N)$ for some $c > 1$.

Definition 1. *An Oblivious RAM protocol is a set of interactions between a user and a server comprised of the following user functions:*

Read(x) : The user retrieves the value of the block with identifier x from the server.
Write(x, y) : The user changes the value of the block with identifier x to y. If block x is not present in the database, that block is added.

We also give a brief definition of ORAM security (obliviousness) and refer readers to related work [2, 4, 8, 14–16] for details.

Definition 2 (Obliviousness). *An Oblivious RAM construction is secure,* iff *for any PPT adversary any two series of data accesses χ and γ, where $|\chi| = |\gamma|$, the corresponding access patterns $AP(\chi)$ and $AP(\gamma)$ induced on the server are computationally indistinguishable with probability $1 - \epsilon(k)$, where ϵ is a negligible function, and k a sufficiently large security parameter.*

3.1 Path-ORAM

In Path-ORAM, Read(x) and Write(x, y) are emulated by the following set of operations:

1. ReadAndRemove(x) – Returns the value of the block with identifier x, or $\perp$ if x identifies a dummy or if x does not exist in the ORAM. Additionally, this operation removes block x from the ORAM.
2. Add(x,y) – Adds a block with identifier x and value y to the ORAM.
3. Pop() – Returns a real data block if the ORAM contains such a block and a dummy otherwise.

A traditional Read can emulated by ReadAndRemove followed by Add to put the block back in the ORAM. Similarly, Write can be emulated with a ReadAndRemove on a dummy value if the block does not exist in the ORAM yet, and an Add with the new value of the block. Conceptually, this set of operations is more conducive to an ORAM construction, because it hints at the idea that when reading a block, there must be an active relocation of the block in order to hide future accesses to the block.

3.2 Tree Construction

Assume for simplicity that N is a power of two. In order to amortize the cost of shuffling, Path-ORAM uses a tree of $2N-1$ "bucket" ORAMs arranged in a tree of depth $\log(N)$. These internal ORAMs are each fully-functioning ORAMs with a capacity of $n := \log(N)$ "slots". The buckets must have three properties: (1) support a non-contiguous identifier space (2) support ReadAndRemove and Add (3) have zero probability of failure. In Path-PIR, we will replace the bucket ORAMs with PIR operations, so these are the three properties our construction must have in order to be sound.

When blocks are added to the ORAM, they are inserted in the root bucket. Each block is tagged with a random number $t \in \{0, \ldots, N-1\}$, which corresponds to a leaf node towards which that block will be moving. The user stores a map M which, for each block in the ORAM, contains the value t for that block. $M(x)$ denotes the value t for x, stored in the user memory. As this would imply $O(M)$ user memory, Shi et al. [15] show how a recursive mechanism to reduce user memory complexity to $O(1)$. However, for the sake of clarity, we will assume $O(N)$ user memory when discussing Path-PIR. The recursive technique can be applied equally to our construction since it does not depend on the makeup of the individual buckets.

ReadAndRemove – Assuming that a block x starts at the root bucket and moves down the tree towards its respective leaf node, block x will always be found somewhere along the path from the root to $M(x)$. Therefore, a ReadAndRemove can be performed by executing ReadAndRemove(x) on every bucket along the path from the root to $M(x)$. One bucket will store block x. Block x will be removed from this bucket, and all other buckets along the path will return $\perp$.

Add – A new leaf node $t \leftarrow \{0, \ldots, N-1\}$ is randomly chosen, and the user inserts block x with value y into the root bucket, tagged with leaf node t.

Every Read and Write operation consists of one ReadAndRemove and one Add. Two Read or Write operations to the same block will be completely independent, because a new random t is chosen for each *Add*. Therefore, this construction achieves obliviousness.

Tree balancing. To facilitate the movement of blocks towards leaf nodes, and to prevent internal buckets from overflowing, the user must Evict blocks from internal buckets to their children. At each level of the tree, the user randomly picks $\xi \in \mathbb{N}$ buckets and executes Pop to read and remove one data block from them. The user then writes to each of the child buckets, moving data blocks toward the correct leaf nodes and performing dummy operations on those children

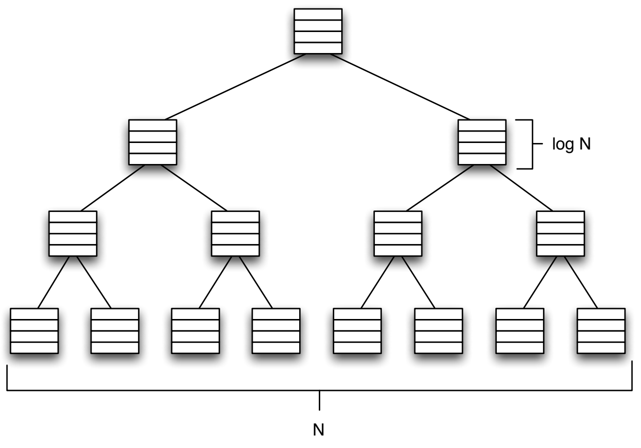

Fig. 1. Layout of server-side storage. $2 \cdot N$ buckets, each of size $n = log(N)$ "slots", are arranged in a binary tree. Blocks enter at the root node and are propagated toward the leaves by Evict.

which are not on the correct path maintaining obliviousness. One can show that $\xi = 2$ is sufficient to keep any buckets from overflowing with high probability, if Evict is performed after every Read or Write operation [15].

Complexity. Assuming each bucket ORAM with individual capacity of $n = \log(N)$ has communication complexity $R(n)$ for its operations, we can calculate the overall cost for this tree construction. ReadAndRemove performs one operation on each of the $\log(N)$ buckets, so its cost is $\log(N) \cdot R(n)$. Add operates only on the root bucket, and so has complexity simply $R(n)$. Evict operates on $3 \cdot \xi \cdot n$ buckets (one parent and $\xi = 2$ children for each bucket evicted) and so has cost $3 \cdot \xi \cdot \log(N) \cdot R(n)$. For all bucket ORAMs, the worst-case cost is $O(n)$. For the individual buckets, $n = \log(N)$, so the worst-case cost for eviction (the most expensive operation) is $3 \cdot \xi \cdot \log(N) \cdot log(N)$. Therefore, regardless of which bucket construction is used the overall worst-case complexity for Path-ORAM is $O(l \cdot \log^2(N))$. Recursively storing the user memory requires at most $\log(N)$ additional ORAMs, adding another $\log(N)$ factor to the overall cost and resulting in $O(l \cdot \log^3(N))$.

4 Path-PIR's Hybrid Construction

Since PIR has sub-linear worst-case communication complexity, we replace the bucket ORAMs with PIR queries to obtain better overall worst-case performance. Our goal in Path-PIR is to create a "PIR-bucket" replacing the bucket ORAMs at each node in the tree. However, it is not sufficient to simply replace the ORAM buckets with PIR, because buckets must have the ability to add and change blocks in order to support all the necessary ORAM operations. Therefore, in addition to standard PIR reading, we also need an equivalently secure writing protocol which we called "PIR-writing". To begin, we will briefly define PIR and discuss relevant details of the PIR protocols we will be using.

Definition 3. *A Private Information Retrieval protocol is a set of interactions between a user and a server comprised of the following functions:*

$\mathsf{PrepareQuery}(x)$: Given a private input $x \in \{1, 2, \ldots, N\}$, the user generates a query which is designed to retrieve the block at index x from the server.
$\mathsf{ExecuteQuery}(q)$: The server receives query q prepared by the user and executes it over the database. The response, which encodes the requested block, is sent back to the user.
$\mathsf{DecodeResponse}(\mathsf{r})$: The user receives the server's response to its query and decodes it to retrieve the requested block.

Along the same lines of "obliviousness" in ORAM, we briefly define security ("Privacy") for PIR. Details can be found in Ostrovsky and Skeith [13].

Definition 4 (Privacy). *A Private Information Retrieval protocol is secure,* iff *for any PPT adversary any two indices χ and γ, the corresponding queries $Q = \mathsf{PrepareQuery}(\chi)$ and $Q' = \mathsf{PrepareQuery}(\gamma)$ are computationally indistinguishable with probability $1 - \epsilon(k)$, where ϵ is a negligible function, and k a sufficiently large security parameter.*

We consider only single-server, computationally-secure PIR protocols.

Different from ORAM, PIR does not require keeping a state in between queries. Consequently, it can also be used to retrieve data from a public, unencrypted database. Since PIR protocols are stateless, each invocation of the protocol must cause the server to perform $O(l \cdot N)$ computation. At a minimum, the server must "touch" each of the blocks in the N-capacity database or it could learn which blocks were not chosen by the user.

We now describe the different techniques to perform PIR in Path-PIR (as listed in the "internal bucket organization" column in Table 1).

Linear Kushilevitz and Ostrovsky [7] have shown that a more efficient protocol can be constructed using an IND-CPA *additively* homomorphic encryption scheme $(\mathcal{K}, \mathcal{E}, \mathcal{D})$. For a scheme to be additively homomorphic, it must satisfy the following condition: $(\exists \oplus)(\forall x, y) : \mathcal{E}(x) \oplus \mathcal{E}(y) = \mathcal{E}(x + y)$, where $\oplus$ is an efficiently computable function. Note that, given this property, it is also true that $\forall x, y : \mathcal{E}(x) \cdot y = \mathcal{E}(x \cdot y)$, where "$\cdot$" denotes *scalar* multiplication, which can be viewed as repeated application of $\oplus$. The above functions can be implemented using an additively homomorphic cipher as follows:

1. $\mathsf{PrepareRead}(\mathsf{x})$ – The user generates a vector $Q = \{q_0, ..., q_n\}$ where $\forall i \neq \mathsf{x} : q_i = \mathcal{E}(0)$ and $q_{\mathsf{x}} = \mathcal{E}(1)$.
2. $\mathsf{ExecuteRead}(\mathsf{q})$ – The server computes a dot product of Q with the vector D (using the scalar multiply operator of the homomorphic cipher) and returns the result, $\mathcal{E}(d_{\mathsf{x}})$.
3. $\mathsf{DecodeResponse}(\mathsf{r})$ – The user computes $m = \mathcal{D}(r)$.

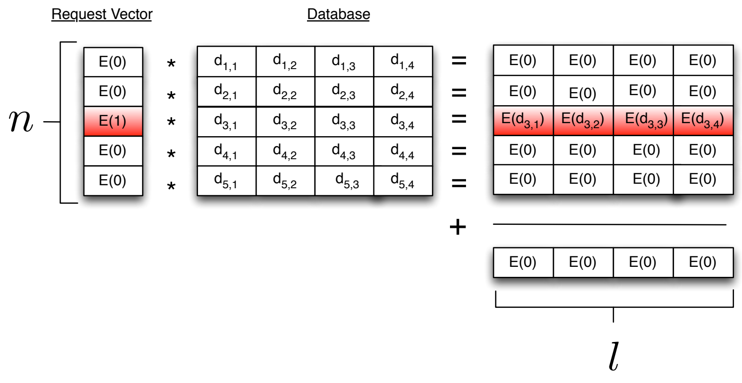
Request Vector
Database
n
E(0)
E(0)
E(1)
E(0)
E(0)
$d_{1,1}$ $d_{1,2}$ $d_{1,3}$ $d_{1,4}$
$d_{2,1}$ $d_{2,2}$ $d_{2,3}$ $d_{2,4}$
$d_{3,1}$ $d_{3,2}$ $d_{3,3}$ $d_{3,4}$
$d_{4,1}$ $d_{4,2}$ $d_{4,3}$ $d_{4,4}$
$d_{5,1}$ $d_{5,2}$ $d_{5,3}$ $d_{5,4}$
E($d_{3,1}$) E($d_{3,2}$) E($d_{3,3}$) E($d_{3,4}$)
l

Fig. 2. PIR using the linear scheme. The dot product of the request vector (size n) and the database is computed. The result has size l.

The above computations are sound, because: $\forall x : \mathcal{E}(0) \cdot x = \mathcal{E}(0)$ and $\forall x : \mathcal{E}(1) \cdot x = \mathcal{E}(x)$. All blocks that the user is not interested in are "zeroed out", and the sum of the products will be equal to an encryption of the single block requested (see Figure 2). The communication cost for this protocol is $O(l+k \cdot N)$, where k is the block size of the cipher. If l is large in relation to N, this protocol can actually be very efficient. However, it is still linear in N and so is not very desirable in a general setting.

$\boldsymbol{\sqrt{N}}$ Kushilevitz and Ostrovsky [7] also show that PIR is possible with improved communication complexity using a twist on the above protocol: instead of sending N ciphertexts to the server, one can arrange the database blocks in a $\sqrt{N} \times \sqrt{N}$ matrix and achieve PIR by sending only $\sqrt{N}$ ciphertexts, requesting one entire row of the matrix. The downside of this approach is that the user receives $\sqrt{N} - 1$ blocks that he is not interested in. The communication complexity here is $l \cdot \sqrt{N}$. This protocols excels precisely when the linear protocols complexity is high, that is when $N > l$.

log $(\boldsymbol{N})$ There exist PIR protocols [9] that achieve logarithmic communication cost in N, but they do not easily adapt to PIR-writing which is required for Path-PIR. However, if we allow "somewhat" homomorphic encryption [3, 11], which supports many additions and a small number of multiplications, we can achieve poly-logarithmic PIR and PIR-writing. We are not aware of an existing PIR with poly-logarithmic PIR *and* PIR-writing. We propose a simple technique to accomplish this as follows. With $\log(N)$ multiplications, we can transform a vector of $2 \cdot \log(N)$ ciphertexts into a vector of N ciphertexts equivalent to the one used in the linear PIR construction. If the user wishes to retrieve the block with index i, having binary representation $i = b_1 b_2 \cdots b_{\log(N)}$, it prepares a set of $\log(N)$ two-element vectors $\{\{v_{1,0}, v_{1,1}\}, \{v_{2,0}, v_{2,1}\}, \cdots, \{v_{\log(N),0}, v_{\log(N),1}\}\}$. Here, $v_{i,b_i} = \mathcal{E}(1)$ and $v_{i,1-b_i} = \mathcal{E}(0)$. For every record in the database d_s, the server multiplies together one element from each vector, chosen based on the binary representation of s. If s is expressed in binary as $s = b'_1 b'_2 \cdots b'_{\log(N)}$, the value will be $v_{1,b'_1} \cdot v_{2,b'_2} \cdot \cdots \cdot v_{\log(N),b'_{\log(N)}}$. Notice that if $s \neq j$, then at least one encryption of zero will be in that product, and the result will be an encryption of zero $\mathcal{E}(0)$. Therefore, only for the single block with index i requested by the user the product will be $\mathcal{E}(1)$, and the protocol continues as above in the linear construction.

This technique requires a somewhat homomorphic encryption scheme which supports at least $\log(N)$ multiplications. However, it may still be practical. We will show later that, in our hybrid Path-PIR construction, we require only $\log\log(N)$ multiplications which results in acceptable performance using existing somewhat homomorphic encryption schemes.

4.1 PIR-Writing

We define PIR-writing [10] as follows:

Definition 5. *A PIR-writing protocol is a set of interactions between a user and a server comprised of the following functions:*

$\mathsf{PrepareWrite}(\mathsf{x},\mathsf{y})$: Given a private input $\mathsf{x} \in \{0, 1, ..., N\}$, the user generates a query which is designed to update block at index x on the server with the new value y.

$\mathsf{ExecuteWrite}(\mathsf{q})$: The server receives query q prepared by the user and executes it over the database, updating the corresponding block to its new value.

We stress that, in contrast to PIR, PIR-writing *cannot* be performed on unencrypted databases. As with ORAM, if the database was unencrypted, the server would learn immediately which record was changed. Still, PIR-writing has one interesting feature which is not subsumed by ORAM: it is also stateless. PIR-writing only requires a long-term key. In contrast, ORAM, even under constant user memory, requires state to be updated with each operation.

Linear Path-PIR's linear PIR protocol above can be adapted to a PIR-writing protocol in a straightforward manner. If, instead of D, the server holds $C = \{\mathcal{E}(d_1), \ldots, \mathcal{E}(d_N)\}$, the protocol runs as follows:

1. $\mathsf{PrepareQuery}(x,y)$ – The user generates a vector $Q = \{q_1, \ldots, q_n\}$ where $\forall i \neq x : q_i = \mathcal{E}(0)$ and $q_x = \mathcal{E}(1)$. Additionally, the user calculates $y' = y - d_x$ and returns the query (Q, y').
2. $\mathsf{ExecuteQuery}(q)$ – The server computes $\Delta C = y' \cdot Q$ and adds it to C componentwise.

As before, multiplying by encryptions of zero will result in encryptions of zero, meaning that every block not being updated has an encryption of zero added to it which corresponds to a re-encryption. The single block being updated has an encryption of y' added to it, resulting in a new value of y. This protocol requires that the user knows the current value of d_{x}, but this can be accomplished with a prior execution of PIR.

An additional problem with this protocol is that the server learns y', the difference between the old value of d_{x} and the new value. If, however, the user first encrypts the blocks with an IND-CPA encryption before applying the homomorphic encryption, the server sees only a difference between two ciphertexts. Since the encryption is IND-CPA, this does not give any information to the adversary.

$\sqrt{N}$ Path-PIR's square-root protocol above can be adapted in a similar way, with y being a $\sqrt{N}$ sized vector, where all the elements but one are zero. If the database is encrypted with another layer of IND-CPA encryption, then instead of zero, these elements will be the difference between the current enryptions of the blocks and a re-encryption of the underlying data.

$\log(N)$ Lipmaa and Zhang [10] have proposed a PIR-writing protocol with poly-logarithmic communication, but it requires that the database grows linearly with the number of write operations. However, since we do not bound the number of write operations, this approach cannot be used. Still, we can adapt our PIR construction above to PIR-writing trivially, since the output of the vector expansion on the server is equivalent to the vector that is sent in the linear construction, in which the user sends the $\log(N)$ sized compressed vector instead of the full linear one.

4.2 Replacing internal ORAM buckets with PIR

For the internal ORAM buckets, as stated above, we only need to provide a PIR capable of performing ReadAndRemove and Add, and that allows for a non-contiguous identifier space. In order to support the Add operation and the "remove" part of ReadAndRemove, any PIR construction requires also PIR-writing. From a high level perspective, our idea in Path-PIR is to implement ReadAndRemove and Add with *one* invocation of PIR and PIR-writing, respectively. PIR does not naturally support a non-contiguous identifier space, because it only retrieves a specific "row" from the server. In Path-PIR, we overcome this by storing an encrypted *map* on the server which identifies the block in each slot of a bucket. Again, let n designate the capacity of a bucket and to N for the capacity of the entire ORAM. Let us assume the user has an IND-CPA additively homomorphic encryption scheme $(\mathcal{E}, \mathcal{D}, \mathcal{K})$ and a simple IND-CPA symmetric encryption scheme $(\mathcal{E}', \mathcal{D}', \mathcal{K}')$ such as AES-CBC with random IVs. We construct an n slot bucket in Path-PIR, meeting the above conditions, as follows:

It is sufficient to show that we can implement an oblivious bucket that supports ReadAndRemove and Add, and that allows for a non-contiguous identifier space. By non-contiguous identifier space we mean that a bucket may hold n items, but the identifiers for those items may be from the set $\{0, \ldots, 2^m\}$ with $m > n$. This is required for the tree construction, because there are, overall, N

Algorithm 1: ReadAndRemove

```
Input: Identifier $x$ of block to retrieve
Output: Value of block $x$ or $\perp$ if block does not exist
begin
    Read and decrypt $U = \{u_1, .., u_n\}$ from the server
    $i \leftarrow 0$
    $exists \leftarrow false$
    for $j \in \{1, ..., n\}$ do
        if $u_j = x$ then
            $i \leftarrow j$
            $u_j \leftarrow \perp$
            $exists \leftarrow true$
        end
    end
    Reencrypt $U$ and send back to the server
    $Q \leftarrow \mathsf{PrepareRead(i)}$
    $R \leftarrow \mathsf{ExecuteRead(Q)}$
    if $exists$ then
        return $\mathcal{D}'(\mathcal{D}(\mathsf{DecodeResponse(R)}))$
    else
        return $\perp$
    end
end
```

elements in the ORAM, with N unique identifiers, and each bucket has capacity only $n = \log(N)$. Therefore, there will be more possible identifiers than slots in the bucket. Standard PIR does not support a non-contiguous identifier space, as the "identifiers" are the row indices of each block in the database. We will overcome this in Path-PIR by using a *map*, stored on the server, which relates block identifiers to rows and allows us to use PIR with arbitrary identifiers.

The first thing to note is that, in order to support the Add operation and the "remove" part of ReadAndRemove, any construction attempting this will also have to use a PIR-writing protocol to mask these operations. At a high level, the idea will be to implement ReadAndRemove and Add with one invocation of PIR and PIR-writing respectively. Let n designate the capacity of a bucket (as opposed to N for the capacity of the entire ORAM). In Path-PIR, we construct a store for the internal ORAM buckets meeting the above conditions for n blocks as follows:

Data storage The server will store n tuples $(\mathcal{E}'(t), \mathcal{E}'(u), \mathcal{E}(\mathcal{E}'(v)))$. t is the leaf node that block is moving toward, u is the block identifier and v is the actual data ("value") of the block. If the slot is empty (i.e., no block is currently stored there) then u is set to some canonical dummy value $\perp$. The value for each block is stored doubly-encrypted so that we can use the PIR-writing protocols outlined above.

Algorithm 2: Add

```
Input: Identifier $x$ of block to add and value $y$ of said block
Output:
begin
    Read and decrypt $U = \{u_1, .., u_n\}$ from the server
    $i \leftarrow 0$
    /* First, find an empty block in the bucket */
    for $j \in \{1, ..., n\}$ do
        if $u_j = \perp$ then
            $i \leftarrow j$
        end
    end
    /* Mark that block with its new identifier */
    $u_i \leftarrow x$
    Reencrypt $U$ and send back to the server
    /* Read the existing block value */
    $Q \leftarrow \mathsf{PrepareRead(i)}$
    $R \leftarrow \mathsf{ExecuteRead(Q)}$
    /* Calculate the difference between the old and new values */
    $oldValue \leftarrow \mathcal{D}(\mathsf{DecodeResponse(R)})$
    $changeValue \leftarrow \mathcal{E}'(y) - oldValue$
    /* Write the change back to the bucket */
    $Q \leftarrow \mathsf{PrepareWrite(i, changeValue)}$
    ExecuteQuery(Q)
end
```

ReadAndRemove(x) The user reads all the encrypted u values from the server (we will call these values the map) and learns in which slot block x resides in. If the requested block is present in this store at slot i, the user changes its u_i value to $\perp$, reencrypts all u values with fresh randomness and sends them back to the server. This marks the row as a ”dummy” and effectively performs the “remove” part of ReadAndRemove. All rows in the map are reencrypted so the server does not learn which block the user was actually interested in. The user then executes PrepareRead(i) and sends it to the server. The server executes the query over $V = v_1, ..., v_n$, returns the response and the user decrypts it with $\mathcal{D}$ and $\mathcal{D}'$ to obtain the value for block x. We do not have to remove or change the value v corresponding to the block that we are reading, but only change its identifier to $\perp$. Future Add operations will simply overwrite the existing value.

Add(x,y) The user reads all encrypted u values from the server and selects an empty block i where $u_i = \perp$. The user sets $u_i = x$, reencrypts all u values and sends them back to the server. The user then runs PrepareWrite(i,y) and the server executes the PIR-writing query over V, changing the value in the i^{th} slot to y. Note that in order to calculate the query for PIR-writing, the user must already know the old value of the block. Therefore, there is an implicit PIR query that occurs as part of PrepareWrite, but it has the same communication complexity as the PIR-writing query.

Complexity Analysis The communication complexity for Path-PIR's "PIR-bucket" is $O(n{\cdot}k{+}P(n))$, where k is the block size of the additively homomorphic encryption, and $P(n)$ is the complexity of the underlying PIR protocol. For our linear scheme above, the communication complexity is $O(n \cdot k + l)$ so the overall communication complexity is just $O(n \cdot k + l)$. For the square-root ($\sqrt{N}$) scheme however, it is $O(l \cdot \sqrt{n})$, but we still require the $O(n \cdot k)$ to download the bucket map, negating any benefit from the square-root technique itself. Unlike ORAM, however, our PIR-bucket requires $O(n \cdot l)$ computation. When used in the larger ORAM construction, $n = \log(N)$, so this computation is quite reasonable as we will demonstrate in Section 5.

Security The overall ORAM structure maintains security as long as the buckets in turn meet the security requirements of an ORAM. It is easy to see that PIR, in combination with a compatible PIR-writing protocol, form a bucket which makes any access patterns indistinguishable. This follows directly from the security definition of PIR, which guarantees any queries are indistinguishable.

4.3 Variations and Discussion

Lower communication for ReadAndRemove At the tree level, the cost for a ReadAndRemove operation is $\log(N) \cdot P(\log(N))$. With $P(n) = O(n \cdot k + l)$, this results in $k \cdot \log^2(N) + l \cdot \log(N)$. Note, however, that if we use the above technique, but store the bucket map at the user, then we achieve $O(\log(N))$ memory complexity, and we already know exactly in which slot the block we want resides in. We can save a factor of $\log(N)$ by executing the same PIR query over each bucket in the path. As an example, if we know that the block we want is in slot i of bucket j, we can execute a PIR query for slot i over each bucket in the path, ignoring the i^{th} value from all the buckets but bucket j. This is possible, because the other buckets are only queried to maintain obliviousness, their responses are simply discarded. Knowing that these responses are unnecessary, we can even execute a second level of PIR over those responses to return only the single one we are interested in. We can now send two PIR queries: the first selects the i^{th} row from every bucket, and the second selects the response from the j^{th} bucket (out of $\log(N)$ buckets in the path). The overall communication for ReadAndRemove is now $k \cdot \log(N) + l$, which is optimal for any retrieval within the constant factor k. The complexity for ReadAndRemove is particularly important, because it represents the amount of communication that needs to be spent before the user has access to their data. Although Evict can be quite expensive, it can be executed in the background on the server without any user interaction, and the user does not need to "wait" on it.

Lower communication for Evict Path-PIR's default approach to Evict using its PIR-bucket is to simply execute a ReadAndRemove on the parent bucket and two Add operations on the children. This requires three PIR queries and two PIR-writing queries. Since the user knows which child node the block is going to be added to, it can simply execute a "dummy" PIR query over the other child node, where all the encryptions in the request vector are encryptions of zero. The same change value can then be used for both children, but the dummy request will simply result in an entire vector of zeroes and no change to the non-selected child bucket. With this modification, Path-PIR can coalesce the two reads and writes on the child nodes into one, saving a factor of $2 \cdot l$.

Improving overall asymptotic performance Path-PIR already achieves optimal communication complexity for ReadAndRemove and Add, so any improvement in overall communication complexity must come from Evict. In case we would allow $O(n)$ user storage complexity such as Stefanov et al. [16], Path-PIR can also store the bucket map on the user. In addition to t, for each block the user will also store which bucket and which slot inside that bucket the block is currently in. This requires a tripling of user storage, but allows for PIR-bucket operations without querying the server for the map. With this modification, we are no longer constrained by the $n \cdot k$ communication complexity for retrieving the map and can lower communication complexity for our PIR-bucket. However, this will not lead to any improved performance in the overall tree construction, because Evict has a minimum cost of $O(\log^2(N))$, independent of the bucket performance. This is due to the fact that the user must randomly select a constant number of buckets at each level to be evicted. The minimum cost to simply "describe" this choice in an unencrypted way, at each level, is $O(\log(N))$. Therefore, evicting at all $\log(N)$ levels requires $O(\log^2(N))$ complexity.

Fortunately, there is no need to perform Evict in a random way. Randomized eviction means that the expected number of operations that pass before a bucket at level L is evicted is 2^L. We can also realize Evict in a deterministic way, independent of the user access pattern, by simply sweeping through the set of buckets. Using this deterministic Evict process, we are no longer constrained by the $O(\log(N))$ lower-bound and can benefit from more efficient buckets. Using the square-root PIR protocol instead of the linear one immediately achieves an overall complexity of $O(l \cdot \log^{1.5}(N))$. While this approach does not decouple l from the poly-logarithmic factor, it decreases its power.

4.4 Using Somewhat or Fully Homomorphic Encryption

Now that we have a less restrictive lower-bound for Evict, we are free to improve our PIR-bucket further. It is an open question whether PIR-writing can be performed with poly-logarithmic communication complexity and sublinear database growth with only additively-homomorphic encryption. However as shown above, we can get that communication complexity with a somewhat-homomorphic encryption that supports $\log(n)$ multiplications. In our tree construction, $n =$

$\log(N)$, so we only need to support $\log\log(N)$ multiplications, which for all practical database sizes is, e.g., less than six ($2^{2^6} = 2$ exabytes). Naehrig et al. [11] have shown that for small numbers of multiplications, somewhat-homomorphic encryption can actually be quite practical. With such a bucket, the number of encryptions the user needs to send in order to do an operation is also only $\log\log(N)$. This gives us an overall complexity of $k \cdot \log(N) \cdot \log\log(N) + l$.

Fully homomorphic encryption (FHE). Since the eviction process is no longer randomized, we can actually reach a communication complexity of zero given a fully-homomorphic encryption scheme. The user can encode a circuit which evicts one block from a bucket to its children and the server can run it on whichever bucket is queued for eviction at the time. No input from the user is necessary. This would realize an ORAM with optimal communication complexity, since the read/write operations were already optimal and eviction would cost nothing. It is not surprising that one can privately retrieve a block from a database with optimal communication using FHE, since retrieval is equivalent to testing equality over encrypted bit-strings which can be performed quite easily. It is interesting, though, that using this tree construction we can achieve it while only computing over a $\log(N)$ -sized fraction of the database. Any FHE-based approach is likely to be bottlenecked by the expensive ciphertext operations, so it is very helpful that computation only needs to be done over a small portion of the database with each user operation. Unfortunately, fully-homomorphic encryption is still too impractical.

4.5 Summary: Complexity Analysis

Table 1 compares the communication complexity of Path-ORAM [15] with Path-PIR using various techniques of organizing the internal ORAM buckets. Path-ORAM can be realized using $O(N \cdot c)$ client memory complexity for very small c, or with $O(1)$ client memory using a recursive construction. This constant memory version is shown in the last column of Table 1. The "internal bucket organization" column refers to the scheme used for each bucket of the ORAM. For Path-ORAM, this denotes a specific ORAM mechanism, cf., Shi et al. [15], while for Path-PIR this denotes the way PIR is performed on a bucket. In conclusion, Path-PIR reduces the expensive communication complexity that depends on file length l by a factor of $\log(N)$ using a simple "linear" PIR protocol. If somewhat-homomorphic encryption is available, complexity reduces by another factor of $\log(N)$. Although a reduction from $\log^3$ to $\log^2$ may look small, in practice, file size l is usually large and total savings can be substantial – as we will demonstrate in the next section.

5 Evaluation

First, note that, to become deployable in a practical, real-world cloud setting, any ORAM protocol must be parallelizable. The only way to scale up in the cloud is to expand to more nodes and CPUs in the cloud's data center. Fortunately,

Table 1. Asymptotic communication complexity of Path-ORAM and Path-PIR with different bucket ORAM schemes. Here, N is the capacity of the ORAM, e.g., the number of files, and l is the length of each file in bits. Memory complexity is constant.

Related Work: Path-ORAM [15]

Internal bucket organization	Operation			
	ReadAndRemove	Add	Evict	Overall Worst-Case
Linear	$l \cdot \log^2(N)$	$l \cdot \log(N)$	$l \cdot \log^2(N)$	$l \cdot \log^3(N)$
$\sqrt{N}$	$l \cdot \log^{1.5}(N)$	$l \cdot \log^{.5}(N)$	$l \cdot \log^{1.5}(N)$	$l \cdot \log^3(N)$
Hierarchical	$l \cdot \log(N) \cdot \log\log(N)$	$l \cdot \log\log(N)$	$l \cdot \log(N) \cdot \log\log(N)$	$l \cdot \log^3(N)$

This paper: Path-PIR

Linear	$l + \log(N)$	$l + \log(N)$	$\log^2(N) + l \cdot \log(N)$	$l \cdot \log^2(N) + \log^3(N)$
$\sqrt{N}$	$l \cdot \log^{1.5}(N)$	$l + \log(N)$	$l \cdot \log^{1.5}(N)$	$l \cdot \log^{2.5}(N)$
$\log(N)^*$	$l + \log(N)$	$l + \log(N)$	$l + \log(N) \cdot \log\log(N)$	$l \cdot \log(N) + \log^2(N) \cdot \log\log(N)$
FHE[†]	$l + \log(N)$	$l + \log(N)$	—	$l \cdot \log(N) + \log^2(N)$

*Requires somewhat-homomorphic encryption.

†Requires fully homomorphic encryption.

Theoretical Optimum

—	$l + \log(N)$	$l + \log(N)$	—	$l + \log(N)$

PIR as we have described is highly parallelizable. The scalar multiplication on each file can be evaluated independently, so Path-PIR can take advantage of up to $O(log^2(N))$ independent CPUs.

Typically, public cloud providers such as Amazon charge users for both communication/data transfer and CPU time [1]. As Path-PIR imposes additional computational requirements, the question is how the additional computational costs relate to the lower communication costs. Path-PIR's should be cost effective compared to related work that does not require computation, but only implies communication cost. Consequently, we have implemented Path-PIR in Java and run simulations in Amazon's EC2 cloud. We have used the additively-homomorphic encryption scheme from Trostle and Parrish [18], because of its conceptual simplicity and efficient homomorphism (adding two ciphertexts is simply an integer addition). Similar results could be obtained with other efficient additively homomorphic ciphers such as NTRU [6].

Setup. To benchmark our PIR protocols, we have conducted our experiments using a single High-CPU Extra Large instance. One hour of CPU time with such an instance costs \$0.58. To compare, Amazon charges \$0.12 per GB transferred [1] (for the first 12 TByte). We have estimated Path-ORAM's communication usage using a worst-case cost of $\lceil \log(N) \rceil \cdot l$ per bucket operation and an average cost of $l \cdot \lceil \log\log(N) \rceil$. To estimate communication time (download/upload), we have assumed an 88 Mbps connection, in line with the maximum speed one would expect when transferring from Amazon S3 [12]. Computation time for Path-PIR was estimated by benchmarking individual buckets of different sizes and then multiplying by the number of bucket operations that are needed per ORAM read or write.

Figures 3, 4, and 5 show the relative communication, time and monetary cost per read/write operation for Path-ORAM and Path-PIR. We consider databases of 1 MB blocks, with total size between 1 and 16 TB. As the block size increases, our scheme becomes more and more efficient relative to Path-ORAM, but it maintains lower worst-case communication all the way down to blocks of about 500 bytes.

All experiments and communication estimations were done in the linear client memory setting. Figure 7 shows that even under this setting the amount of client memory required is quite low, even for large databases. This means that the number of recursive steps needed to reach constant memory is very small. Up to approximately 300 PB, only one level of recursion is needed when using 1 MB blocks. We also note that when recursion is applied to reach constant client memory, it is simply a multiplicative factor to all communication and computation costs, applied uniformly to Path-ORAM and Path-PIR, therefore the trends we show remain intact.

Latency Figure 6 shows the extremely low cost of ReadAndRemove operations in our scheme. This latency property is important, because it represents the amount of communication necessary before the client has access to their data. The eviction, which takes up most of the communication, can be done in the background without user interaction. We are able to obtain extremely low communication requirements for this operation, between 4 and 10 MB to retrieve a 1 MB file for databases up to 1 Exabyte in size, even using recursion to achieve constant client memory.

Discussion We observe in Path-ORAM that, although the user needs to perform one eviction for each read or write operation, these evictions are not required to be performed immediately after the operation. The contribution of eviction is to keep buckets from overflowing, but the correctness and security of the ORAM remains independent of it. The user can actually conduct $\log(N)$ data accesses without any evictions before the root node will overflow. Since ReadAndRemove and Add are very efficient, and the overwhelming majority of communication is consumed during Evict, this could be very useful when a user's cost on communication may vary in different environments. For instance, a user with a cell phone probably pays significantly more money for cellular data than WiFi data. In a practical implementation of Path-PIR, one could defer evictions while they are on expensive cellular data and choose to perform these operations later when they are on cheap WiFi. This allows for extremely low communication requirements while Evict operations are being deferred. Additionally, the size of the root bucket can be increased by any constant factor to allow for more deferred operations without effecting the overall complexity.

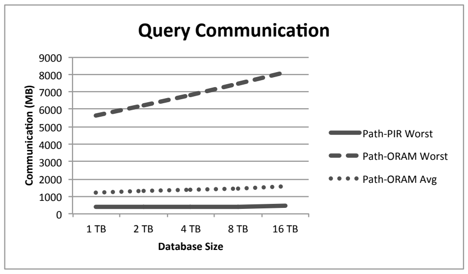

Fig. 3. communication required for one read/write operation.

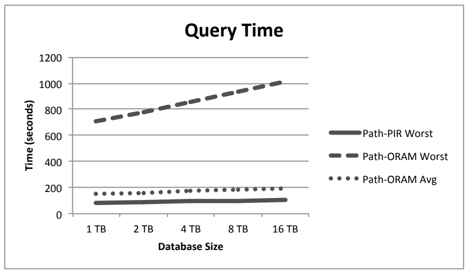

Fig. 4. Time required for one read/write operation.

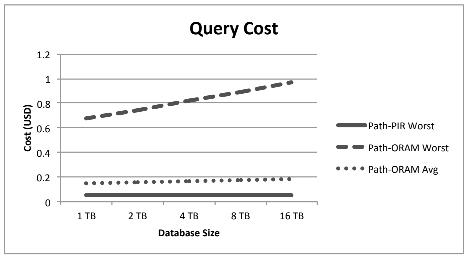

Fig. 5. Monetary cost for one read/write operation, based on Amazon prices.

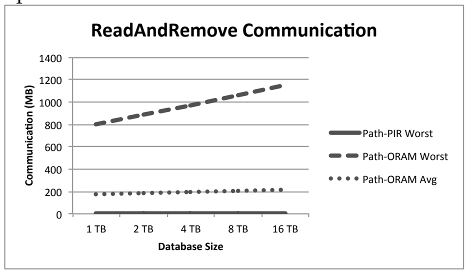

Fig. 6. Communication required for just a ReadAndRemove operation.

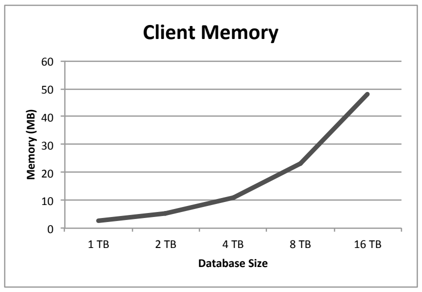

Fig. 7. Client memory required under O(n) client memory. Even with large databases, linear client memory can still be quite small.

6 Conclusion

Outsourcing sensitive data to untrusted clouds implies not only encryption, but also hiding user access patterns in an efficient manner. Path-PIR demonstrates that integrating PIR into recent ORAM mechanisms provides better worst-case communication complexity by a factor of at least $\log N$, without incurring unreasonably large computational burden on the cloud. Through experiments, we are able to verify that Path-PIR's cost savings from the lowered communication complexity are significantly higher than the cost of extra computation. Additionally, Path-PIR benefits from low latency that makes it especially conducive to communication-constrained devices like cell phones or embedded systems.

Bibliography

[1] Amazon. Amazon Elastic EC2 Pricing, 2013. `http://aws.amazon.com/ec2/pricing/`.

[2] D. Boneh, D. Mazieres, and R.A. Popa. Remote Oblivious Storage: Making Oblivious RAM Practical, 2011. URL `http://dspace.mit.edu/handle/1721.1/62006`. Technical Report.

[3] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of Foundations of Computer Science*, pages 97–106, Palm Springs, USA, 2011. ISBN 978-1-4577-1843-4.

[4] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. ISSN 0004-5411.

[5] Google. A new approach to China, 2010. `http://googleblog.blogspot.com/2010/01/new-approach-to-china.html`.

[6] J. Hoffstein, J. Pipher, and J. Silverman. NTRU: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory*, Lecture Notes in Computer Science. Springer Berlin, Heidelberg, 1998.

[7] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *, 38th Annual Symposium on Foundations of Computer Science, 1997. Proceedings*, pages 364 –373, October 1997. doi: 10.1109/SFCS.1997.646125.

[8] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of Symposium on Discrete Algorithms*, pages 143–156, Kyoto, Japan, 2012.

[9] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *Proceedings of Information Security Conference*, pages 314–328, Singapore, 2005. ISBN 978-3-540-29001-8.

[10] H. Lipmaa and B. Zhang. Two New Efficient PIR-Writing Protocols. In *Proceedings of Applied Cryptography and Network Security Conference*, pages 438–455, Beijing, China, 2010. ISBN 978-3-642-13707-5.

[11] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of Workshop on Cloud computing Security*, pages 113–124, Chicago, USA, 2011. ISBN 978-1-4503-1004-8.

[12] Nasuni. State of Cloud Storage Providers Industry Benchmark Report, 2011. `http://cache.nasuni.com/Resources/Nasuni_Cloud_Storage_Benchmark_Report.pdf`.

[13] R. Ostrovsky and W. Skeith. A Survey of Single-Database Private Information Retrieval: Techniques and Applications. In *Proceedings of Public Key Cryptography –*, pages 393–411, Beijing, China, 2007. ISBN 978-3-540-71676-1.

[14] B. Pinkas and T. Reinman. Oblivious RAM Revisited. In *Proceedings of Advances in Cryptology – CRYPTO*, pages 502–519, Santa Barbara, USA, 2010. ISBN 978-3-642-14622-0.

[15] E. Shi, T.-H.H. Hubert Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, volume 7073, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[16] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Proceedings of Network and Distributed System Security Symposium*, San Diego, USA, 2012.

[17] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. `http://techcrunch.com/2010/09/14/google-engineer-spying-fired/`.

[18] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Proceedings of Conference on Information Security*, pages 114–128, Boca Raton, USA, 2010.