

Efficient Private File Retrieval by Combining ORAM and PIR

Abstract—Recent research results on tree-based Oblivious RAM by Shi et al. [14] obtain communication complexity of $O(l \cdot \log^3(N))$ in the worst-case for an N -capacity storage with blocks size l . The individual nodes in the tree, however, are constructed using traditional ORAMs which have worst-case communication complexity linear in their capacity and block size. PIR protocols are able to provide better worst-case bounds (decoupling capacity from block size), but have traditionally been less practical than ORAM due to the fact that they require $O(N)$ computational complexity on the server. This paper presents Path-PIR, a hybrid ORAM construction, using techniques from PIR, that overcomes the individual weaknesses of each. Path-PIR significantly reduces communication complexity when the block size of the ORAM is large. Compared to existing work, this leads to smaller data transfer costs by orders of magnitude for practical sized databases and, achieves asymptotic communication of $O(l \cdot \log^2(N))$ for large block sizes. Additionally, the typically high computational cost of PIR is negated by the tree structure of the ORAM, which requires only a small fraction of the database to be operated on for each query. We also investigate the concept of an ORAM’s *latency*, which is the amount of communication required before users receive the result of their query. We show that Path-PIR achieves lower latency than any existing scheme, only about four times the block size. Using Amazon EC2 as an example, we demonstrate that even with the additional cost on PIR computation, Path-PIR provides a significant monetary saving compared to related work.

I. 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. Seeing news of high-profile hacking incidents involving data theft has become commonplace [5, 18]. 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 their data. We call this information the user’s “access pattern”. As a motivating example, consider a hospital that outsources 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 patient 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/cloud): Oblivious RAM (ORAM) [3] and Private Information Retrieval (PIR) [7]. The traditional approach taken by ORAM is to arrange the data in such a way that the user never touches the same piece twice, without an intermediate “shuffle” which erases the correlation between block locations. ORAMs have historically featured low amortized communication complexity and did not require any computation on the server, but occasionally the user was required to download and reshuffle the entire database. This could 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 encrypted requests and sends them to the server. 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.

Recently, there has been a flurry of research on ORAMs which provide a sublinear worst-case complexity. This goal was achieved by, e.g., Shi et al. [14] and independently by Kushilevitz et al. [8], but with very different approaches. Since then, several additional schemes have been proposed that achieve better communication complexity, but at the cost of increasing *client memory* from constant to logarithmic [15] or polynomial [16, 17] in N . As we will discuss below, constant client memory is a desirable and an important property expected for many applications, and even logarithmic client memory can be a significant drawback when it depends on the block size l . Because of this, we focus our work based on the constant client memory complexity model and compare our performance with ORAM techniques that use the same model.

In this paper, we present Path-PIR, a new ORAM con-

struction combining ORAM and PIR, thereby overcoming the individual drawbacks of each of the two approaches. Path-PIR’s strategy is to augment the recently proposed ORAM by Shi et al. [14] using techniques from PIR. It has been shown that PIR can be quite efficient when the block size is large in relation to the number of elements in the database [10]. Since the tree-based ORAM of Shi et al. is composed of many buckets, each of which has only a small number of elements, we are able to take advantage of PIR’s better worst-case communication guarantees, while at the same time using the tree structure to limit the portion of the database which is subject to expensive PIR operations.

Additionally, we explore the notion of an ORAM’s *latency*, the amount of communication required before the client has access to their data. This is important because many modern ORAM constructions involve an initial query, which returns the data, and a more expensive “book keeping” protocol which insures the integrity and obliviousness of the data structure. Low latency is a highly desirable feature, since the reorganizing and shuffling of the data structure can happen in the background after the client receives their data. We also note that low latency can be very useful in certain client settings with restrictive data limits, i.e. smart phones.

Our Path-PIR scheme is especially suited to databases with large block sizes, an important setting in the real world that has not been thoroughly explored. For example, in medical applications, the size of each patient record (block size) can be quite large, due to medical images, test results, etc.

This paper makes the following major **contributions**:

- 1) Path-PIR, an ORAM construction which uses a combination of PIR and tree-based ORAM techniques to achieve good performance while maintaining constant client memory requirement. Specifically, in a database that stores a total of N files (entries), and each file is of bit length l , Path-PIR reduces communication from $O(l \cdot \log^3(N))$ to $O(\log^3(N) + l \cdot \log^2(N))$. Path-PIR is especially efficient in practical real-world settings where $l \geq 100$ KB and $N < 2^{35}$, i.e., total databases of up to 3 PB size.
- 2) 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 $l > O(\log^2(N))$.
- 3) 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 significantly *faster* and *cheaper* operations than previous constant client memory constructions. The source code is available for download [12].

II. RELATED WORK

There exists a large body of work on improving Oblivious RAM since the original concept introduced by Goldreich and Ostrovsky [3]. For example, Pinkas and Reinman [13] 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. However, it only achieves this complexity for extremely large databases. For N where $\log(N) < 7 \log \log(N)$, this scheme actually degrades to $\Omega(l \cdot \log^3(N))$. Since we are considering practical sized databases in the order of several hundred petabytes, that is, $N \approx 2^{37}$, Kushilevitz et al. [8] construction achieves only $O(l \cdot \log^3(N))$ performance.

In contrast, as we will see, Path-PIR only approaches $l \cdot O(\log^2(N))$ for very large databases and is, in practice much closer to $O(\log(N))$. Our evaluation will show that this leads to significant performance improvements, and Path-PIR clearly outperforms Kushilevitz et al. [8] at the targeted database sizes.

Shi et al. [14] have proposed another ORAM with worst case bounds $O(l \cdot \log^3(N))$ using an entirely new construction. Instead of deamortizing previous schemes, authors 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. Using a recursive access technique, this scheme can achieve constant client memory. Path-PIR is based upon this approach, overcoming its (still) expensive individual bucket accesses with PIR as presented in Section III.

Stefanov et al. [16] have shown that a bucket-based construction can achieve $O(l \cdot \log(N))$ amortized *and* worst-case complexity. However, this is achieved only with either linear client memory complexity or with square-root client memory complexity at the cost of additional communication complexity. Achieving constant client memory is an important requirement, because it allows ORAM applications for constrained devices such as smart phones and embedded systems. Unlike Shi et al. [14], the client memory required by Stefanov et al. [16] is dependent on the block size; for example, a 1 TB database of 1 MB files consumes approximately 800 MB of client memory in the square-root construction of Stefanov et al. [16]. It is unrealistic to expect 800 MB of free client memory for many real-world applications. Even in the linear client memory setting, Shi et al. [14] requires only 4 MB. This difference is caused by the fact that the Shi et al. [14] scheme has client memory independent of the block size, while Stefanov et al. [16] needs a block cache which can be very large for large block sizes.

For tree-based schemes, the size of the client memory in the linear setting is important, because it will govern how many recursive steps are needed in the constant-memory setting. Since linear-memory requirements are, for practical sizes, very low, we will see later that reducing to constant-memory requires only a small number of recursive steps (typically one).

TABLE I

COMMUNICATION COMPLEXITY OF PATH-PIR AND RELATED *constant-memory* SCHEMES. HERE, N IS THE ORAM CAPACITY, E.G., THE NUMBER OF FILES, l IS THE BIT-LENGTH OF EACH FILE, AND k IS THE SECURITY PARAMETER. LATENCY IS THE AMOUNT OF COMMUNICATION BEFORE THE CLIENT HAS ACCESS TO DATA. THE “PRACTICAL” SETTING IS $l \geq 100$ KB AND $N < 2^{35}$.

	Latency	Worst-Case	Practical Worst-Case
Shi et al. [14]	$O(l \cdot \log^2(N))$	$O(l \cdot \log^3(N))$	$O(l \cdot \log^2(N))$
Kushilevitz et al. [8]	$O(\frac{l \cdot \log^2(N)}{\log \log(N)})$	$O(\frac{l \cdot \log^2(N)}{\log \log(N)})$	$O(l \cdot \log^3(N))$
Path-PIR Linear	$O(k \cdot \log(N) + l)$	$O(k \cdot \log^3(N) + l \cdot \log^2(N))$	$O(l \cdot \log(N))$
Path-PIR FHE	$O(k + l)$	$O(k \log(N) + l \cdot \log(N))$	$O(k + l)$
Optimal	$O(\log(N) + l)$	$O(\log(N) + l)$	$O(\log(N) + l)$

Stefanof et al. [15] were also able to improve upon the original tree-based construction to achieve communication costs of $l \cdot \log^2(N)/\log \chi$ in a scheme they call Path-ORAM (where $\chi = \log(N)/l$). However, to achieve this, the authors again give up constant client memory. Although this construction achieves relatively low complexity, its memory still contains a block cache which means it is dependent on l and therewith expensive.

One significant down-side to schemes which contain a block cache [14] [16] [15] is that the cache load can only be estimated *empirically*. If the size of the cache is chosen to be too small, and the actual cache needed during execution exceeds that size, then ORAM can only abort and lose its integrity. Even in devices with enough memory to allow for such schemes, it may be a significant burden on the client compared with a scheme that requires only a small constant memory.

Additionally, in order to make an ORAM available to multi-users, the client memory must be stored on the server between data accesses [4]. In such a situation, the client must include the size of their local memory in the communication cost of each request, significantly hampering the efficiency of schemes with non-constant memory requirements.

III. 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 now define the security of an ORAM protocol in the standard way.

Definition (Obliviousness). Let

$$\vec{y} := \{(\text{op}_1, a_1, \text{data}_1), \dots, (\text{op}_M, a_M, \text{data}_M)\} \quad (1)$$

be a sequence of data requests of length M , where op_i is either read or write, a_i is the address targeted by that operation and data_i is either the data to be written, if $\text{op}_i = \text{write}$, or null if $\text{op}_i = \text{read}$.

Let $A(\vec{y})$ be the sequence of accesses induced on the remote storage by the client access pattern \vec{y} . An Oblivious RAM protocol is said to be secure if, for any two data access patterns \vec{y} and \vec{z} , $A(\vec{y})$ and $A(\vec{z})$ are computationally indistinguishable.

A. Shi et al. [14] ORAM

Traditionally, ORAMs support two operations: **Read(x)**, which reads the block with identifier x , and **Write(x, y)**, which writes value y to the block identified by x .

However, **Read(x)** and **Write(x, y)** can be emulated with 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 done by calling **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 that block in order to disassociate future accesses to it.

B. Tree Construction

Assume for simplicity that N is a power of two. In order to amortize the cost of shuffling, Shi et al. [14] 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**, and (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 $t \in \{0, \dots, N-1\}$, 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 $O(N)$ user memory, Shi et al. [14] show how this map can itself be recursively stored in an ORAM to achieve $O(1)$ client memory. However, for the sake of clarity, we will assume $O(N)$ user memory when presenting Path-PIR. The recursive technique can be applied equally to our construction since it does not depend on the makeup of the individual buckets. This adds a $\log(N)$ factor to each query because there are at most $\log(N)$ recursive ORAMs to store that map.

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 $\text{ReadAndRemove}(x)$ 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 $t \leftarrow \{0, \dots, N-1\}$ 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 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 [14].

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 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

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 resulting in $O(l \cdot \log^3(N))$.

IV. PATH-PIR'S HYBRID CONSTRUCTION

Although the complexity of constant client memory ORAM is only $O(l \cdot \log^3(N))$, a large value l will render a ORAM scheme impractical in practice. Our aim is to modify the scheme so that it is practical for all large value l , while maintaining constant memory complexity on the client side.

A. Oblivious Outsourced Storage

Our hybrid construction will be different from traditional ORAM in one important way: it requires computation on the server side. Because of this, we require a slightly different security definition for a construct we call Oblivious Outsourced Storage.

Definition. Let

$$\vec{y} := \{(op_1, a_1, data_1), \dots, (op_M, a_M, data_M)\}$$

be a sequence of data requests of length M , where op_i is either read or write, a_i is the address targeted by that operation, and $data_i$ is either the data to be written, if $op_i = \text{write}$, or null if $op_i = \text{read}$.

Let $\text{Trace}(\vec{y})$ be the transcript of all messages sent between the client and server during the execution of the protocol on \vec{y} . An Oblivious Outsourced Storage scheme is said to be secure if, for any two data access patterns \vec{y} and \vec{z} , $\text{Trace}(\vec{y})$ and $\text{Trace}(\vec{z})$ are computationally indistinguishable.

This definition is compatible with the traditional ORAM security definition, i.e., any secure ORAM will also be a secure OOS (since the data accesses induced by the server are part of the trace). However, it also incorporates the notion of server computation by stipulating that the entire trace must be indistinguishable instead of simply the access pattern on the server. The traditional ORAM definition does not need this requirement because there are no other interactions in the trace besides the access pattern. In our case, however, the trace includes PIR vectors and additional ciphertexts, so the definition must be modified slightly.

B. Private Information Retrieval

Since PIR has communication complexity which can be very efficient for retrieving large blocks, our goal will be to replace the bucket ORAMs in the previously described tree-based scheme with PIR queries to obtain better overall performance. Our approach in Path-PIR is to create a “PIR-bucket” to replace 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 called “PIR-writing”. To begin, we will briefly define PIR and discuss relevant details of the PIR protocol we will be using.

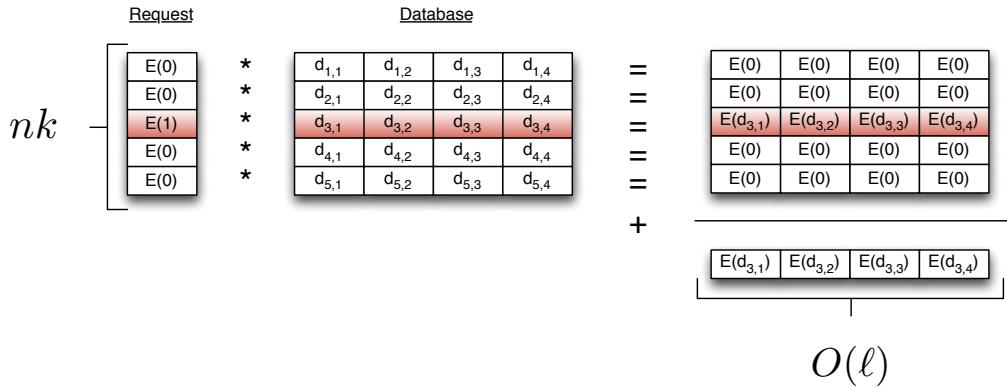


Fig. 1. PIR using the linear scheme. The dot product of the request vector (size n) and the database is computed. The result has size l .

Definition 2. A *Private Information Retrieval protocol* is a set of interactions between a user and a server comprised of the following functions:

PrepareQuery(x): Given a private input $x \in \{1, 2, \dots, N\}$, the user generates a query which is designed to retrieve the block with index x from the server.

ExecuteQuery(q): The server receives query q prepared by the user and executes it over the database. The response, consisting of the encoded requested block, is sent back to the user.

DecodeResponse(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 define security (“Privacy”) for PIR.

Definition 3 (Privacy). A *Private Information Retrieval protocol* is secure, iff for any PPT adversary and any two indices y and z , the corresponding queries $Q = \text{PrepareQuery}(y)$ and $Q' = \text{PrepareQuery}(z)$ are computationally indistinguishable.

We consider only single-server, computationally-secure PIR protocols, as multi-server schemes would not be appropriate for our setting. Though cloud providers may make use of many servers, they are all controlled by the same entity and may freely collude. It may be an interesting problem to consider multiple, competing cloud providers as a multi-server setting, but that is outside the scope of this paper.

In contrast with 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 causes 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.

1) **Linear PIR:** 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) **PrepareRead(x)** – The user generates a vector $Q = \{q_0, \dots, q_n\}$ where $\forall i \neq x : q_i = \mathcal{E}(0)$ and $q_x = \mathcal{E}(1)$.
- 2) **ExecuteRead(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, $\mathcal{E}(d_x)$.
- 3) **DecodeResponse(r)** – The user computes $m = \mathcal{D}(r)$.

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. The communication cost for this protocol is $O(l + k \cdot N)$, where k is the size of the block cipher used. The overall communication is a $\frac{1}{N} + \frac{k}{l}$ fraction of the database. If l is large in relation to N , this protocol is actually very efficient, because l is independent of N in the complexity. For instance, if $l > \sqrt{N}$, then the overall communication is only $O(l)$ which is optimal. This is important because no existing ORAM can obtain optimal communication under any setting of parameters (except the trivial case where $N = 1$). In our use case, N will be very small (actually $\log(N)$ in the notation of the overall ORAM), so it is highly likely that l will be large in comparison to it, particularly in our motivating case of large block sizes. This is where our savings in communication comes from, because retrieval requires only one full sized block to be transferred, while all the “indexing” information is in small ciphertexts.

C. PIR-Writing

We define PIR-writing [9] as follows:

Definition 4. A *PIR-writing protocol* is a set of interactions between a user and a server comprised of the following functions:

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

ExecuteWrite(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.

1) *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 $C = \{\mathcal{E}(d_1), \dots, \mathcal{E}(d_N)\}$, the protocol runs as follows:

- 1) **PrepareQuery(x, y)** – The user generates a vector $Q = \{q_1, \dots, q_n\}$ 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) **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. One might try to set $q_i = y' - y$ to get around this, but then the size of each encryption becomes $O(l)$ and we lose any benefit from using PIR. 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 the two ciphertexts. This is equivalent to seeing two ciphertexts $((c_1, c_2 \oplus c_1) \Leftrightarrow (c_1, c_2))$, which gives the adversary no information under an indistinguishable encryption.

D. 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. This is because the bucket will be storing “sparse” identifiers, i.e., there are N possible block identifiers and a random $O(\log(N))$ subset of them will be in any given bucket. 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 N the capacity of the entire ORAM. Let us assume the user has an IND-CPA additively homomorphic encryption scheme $(\mathcal{E}, \mathcal{D}, \mathcal{K})$, e.g., Paillier, and an IND-CPA symmetric encryption scheme $(\mathcal{E}', \mathcal{D}', \mathcal{K}')$, e.g., AES-CBC with random IVs. We will first show how to construct a basic PIR bucket, then discuss additional improvements that can be made and interesting properties that arise from it.

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 $\{0, \dots, 2^m\}$ with $m > n$. This is required for the tree construction, because there are, overall, N 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.

Note 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). We construct a store for the internal ORAM buckets meeting the above conditions for n blocks as follows:

1) *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 the 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.

2) *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 $V = v_1, \dots, v_n$, 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

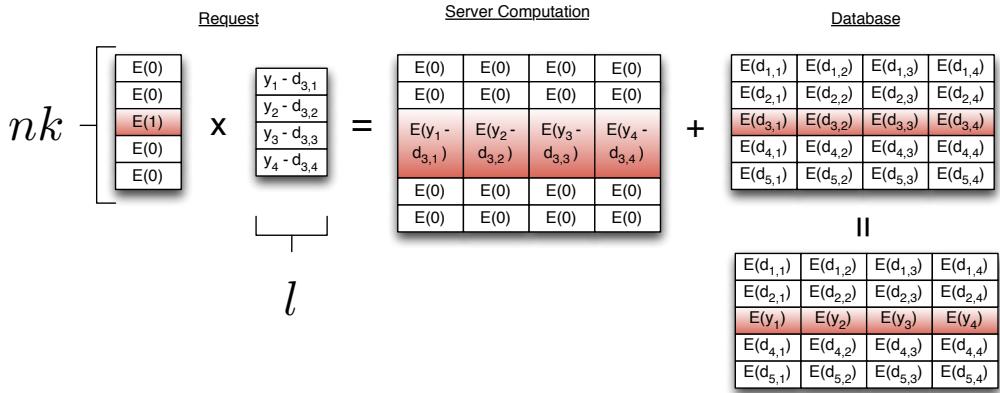


Fig. 2. PIR-Writing using the linear scheme. The cross product of the request vector (size n) and the change value (size l) is computed. And added to the database.

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 the map $U = \{u_1, \dots, u_n\}$ from the server
- $i \leftarrow 0$
- $exists \leftarrow false$
- for** $j \in \{1, \dots, n\}$ **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
- $Q \leftarrow \text{PrepareRead}(i)$
- $R \leftarrow \text{ExecuteRead}(Q)$
- if** $exists$ **then**
- return** $\mathcal{D}'(\mathcal{D}(\text{DecodeResponse}(R)))$
- else**
- return** \perp
- end**

end

value.

3) **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.

4) **Complexity Analysis:** The communication complexity for Path-PIR's "PIR-bucket" is $O(n \cdot k + P(n))$, where k is

Algorithm 2: Add

Input: Identifier x of block to add and value y of said block
Output:
begin

- Read and decrypt $U = \{u_1, \dots, u_n\}$ from the server
- $i \leftarrow 0$
- /* First, find an empty block in the bucket */
- for** $j \in \{1, \dots, n\}$ **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 */
- $Q \leftarrow \text{PrepareRead}(i)$
- $R \leftarrow \text{ExecuteRead}(Q)$
- /* Calculate the difference between the old and new values */
- $oldValue \leftarrow \mathcal{D}(\text{DecodeResponse}(R))$
- $changeValue \leftarrow \mathcal{E}'(y) - oldValue$
- /* Write the change back to the bucket */
- $Q \leftarrow \text{PrepareWrite}(i, changeValue)$
- $\text{ExecuteQuery}(Q)$

end

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 of the bucket is just $O(n \cdot k + l)$. 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 VI.

E. Improvements to the basic scheme

1) *Lower latency*: An interesting property to consider for any ORAM is its data latency, that is the amount of data that is transferred before the client has access to the requested information. In our scheme, the client has access immediately after **ReadAndRemove**. 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. 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 $k \cdot \log^2(N) + l \cdot \log(N)$. We can then save a factor of $\log(N)$ by executing another PIR query over the results from 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 retrieve it with cost only $O(\log^2(N) + l)$. We can 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 latency is now $k \cdot \log^2(N) + l$, which is optimal for any retrieval within the constant factor k . This leads to very low latency in practical situations (lower than any other previous work, even allowing for non-constant client memory).

2) *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 $2 \cdot l$.

Fully homomorphic encryption (FHE). An interesting extension to our scheme would be to use a more powerful homomorphic encryption (FHE). The most communication intensive part of our scheme, which maintains a dependence between N and l , is **Evict**. Unfortunately, since **Evict** is randomized, the user is required to send at least the random choices of buckets to the server, of which there may be many. Fortunately, there is no need to perform **Evict** in a random way. Randomized eviction is only used originally to minimize client memory (the client need not remember which buckets have been evicted recently). All we need is a scheme where 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. The server remembers which bucket it has evicted last on each level, and simply evicts the next one in line. We can see that this does not

depend on the user’s access pattern because it is deterministic, independent of any action by the client.

Since the eviction process is deterministic, 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 very close to optimal communication complexity ($l \cdot \log(N)$), since the read/write operations were already close to optimal and eviction would cost nothing. It is not surprising that one can privately retrieve a block from a database with good 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 to be used in this situation.

F. Summary: Complexity Analysis

Table I compares the communication complexity of related work with Path-PIR. The last column shows the performance of each scheme in a setting with moderately large blocks ($l > 100$ KB) and practical sized databases of up to 3 PB. The two-bucket based schemes perform better in this setting because the depth of recursion is limited to one (they lose a $\log(N)$ factor). Additionally, Path-PIR performs especially well since all the ciphertexts that need to be transferred are significantly less than the size of one block, so the communication is dominated by the $O(\log(N))$ blocks which are transferred during **Evict**.

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. Although a reduction from \log^3 to \log^2 may look small, in practice, the total savings can be substantial – as we will demonstrate by our experimental results presented in section VI.

a) *Latency*: We observe in Path-PIR 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 $O(\log(N))$ 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 may pay 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.

V. SECURITY

It is relatively simple to show the security of our scheme. At a high level, we are just composing PIR and ORAM techniques which are secure individually, and we shall see that they remain secure together. The PIR elements that we use result in the transfer of many ciphertexts between the client and the server. However, these ciphertexts are freshly generated by the client for each query, so under IND-CPA security they should not give any information to the server.

First, we show that the individual bucket, created using PIR techniques, is secure.

Lemma 1. *Path-PIR’s construction of a PIR-bucket is a secure oblivious operation.*

Proof: There are three parts of each PIR-bucket operation: retrieval of the map, execution of the PIR query and execution of the PIR-writing query.

The map is oblivious because it is encrypted and retrieved in its entirety for every operation. At the conclusion of an operation, the entire map is freshly reencrypted and sent back to the server. In that way, it is actually a “trivial” ORAM and therefore meets our security definition.

Next, we show that Path-PIR is broken only if the IND-CPA security of either the symmetric encryption or the additively homomorphic encryption is broken. Let $\mathcal{Q}(\vec{y}) = \{Q_1, \dots, Q_n\}$, where $Q_i = \{q_{i,0}, \dots, q_{i,m}\}$, denote the set of all PIR and PIR-writing queries involved in the execution of a set of data accesses \vec{y} . Each $q_{i,j}$ is an encryption of the plaintext $p_{i,j}$ under our additively homomorphic IND-CPA encryption scheme. Similarly, let $\mathcal{P}(\vec{y}) = \{P_1, \dots, P_n\}$ denote the set of all plaintexts used in the construction of $\mathcal{Q}(\vec{y})$. Assume that Path-PIR is broken, that is, there exists an adversary, \mathcal{A} , that can distinguish between two access patterns, \vec{y} and \vec{z} . We will construct an adversary, \mathcal{A}' which can break the IND-CPA security of our homomorphic encryption.

\mathcal{A}' runs \mathcal{A} and obtains the two access patterns, \vec{y} and \vec{z} , which \mathcal{A} can distinguish. \mathcal{A}' then simulates Path-PIR on the two access patterns, obtaining $X_0 = \mathcal{P}(\vec{y})$ and $X_1 = \mathcal{P}(\vec{z})$. \mathcal{A}' submits X_0 and X_1 to the IND-CPA challenger and receives back C , an encryption of one of the two plaintexts. He then sends C to \mathcal{A} as the trace and \mathcal{A} returns either y or z as the chosen access pattern. If \mathcal{A} returns y , then \mathcal{A}' returns 0 to the challenger, otherwise returns 1. Thus, if \mathcal{A} is able to distinguish between the two access patterns with probability $1/2 + \epsilon$, then \mathcal{A}' will be able to distinguish between the two encryptions with the same probability.

Note that the trace will also include responses from the server, but those cannot reveal any additional information

because they are deterministically computed from the client’s requests.

We have now established that the construction of a PIR-bucket is a secure oblivious operation. ■

Theorem 1. *The basic version of Path-PIR is a secure Oblivious Outsourced Storage protocol.*

Proof: The security proof by Shi et al. [14] shows that the sequence of bucket accesses will be oblivious. As stated above, the standard definition of ORAM security is compatible with our OOS security definition, therefore the security of our scheme depends only on the PIR-bucket itself being oblivious, which is established by Lemma 1. ■

Theorem 2. *The improved latency version of Path-PIR is a secure Oblivious Outsourced Storage protocol.*

Proof: The only difference between the improved latency scheme and the original one is that a single PIR vector is “reused” for every bucket on a path in the tree. However, since PIR vectors are themselves indistinguishable for any two indices being accessed, “reusing” them does not provide any additional information and therefore, the protocol remains oblivious. If an adversary could distinguish in the case with just one request vector, he could equally distinguish that same vector in the original scheme by ignoring all other vectors. ■

Based on the oblivious secure protocol properties of PIR and ORAM, we have established that Path-PIR satisfies the Oblivious Outsourced Security properties.

VI. 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, 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. We have implemented Path-PIR in Java and run simulations in Amazon’s EC2 cloud. We have used the PIR scheme from Trostle and Parish [19], because of its conceptual simplicity and efficient server computation (adding two PIR values is simply an integer addition). Similar results could be obtained with efficient additively homomorphic ciphers such as NTRU [6]. We chose security parameter $k = 2048$ as recommended by the authors.

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). The worst-case commu-

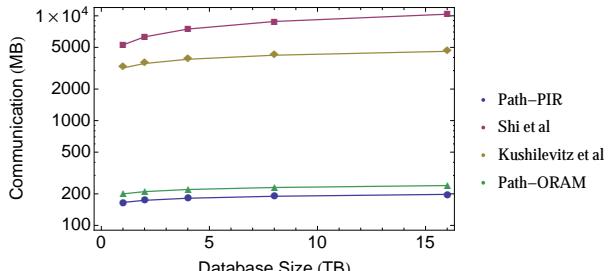


Fig. 3. Communication for one read/write

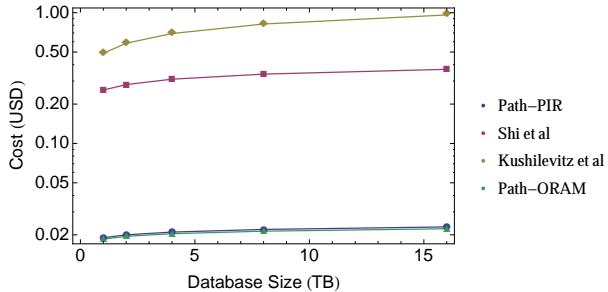


Fig. 5. Monetary cost for one read/write

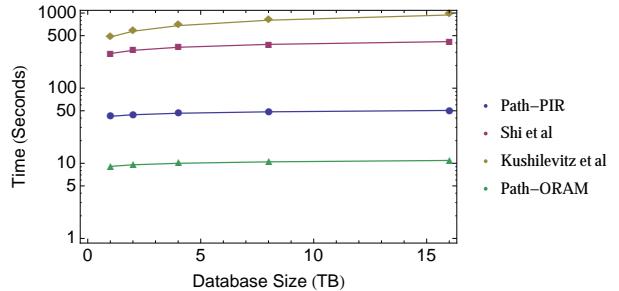


Fig. 4. Time for one read/write

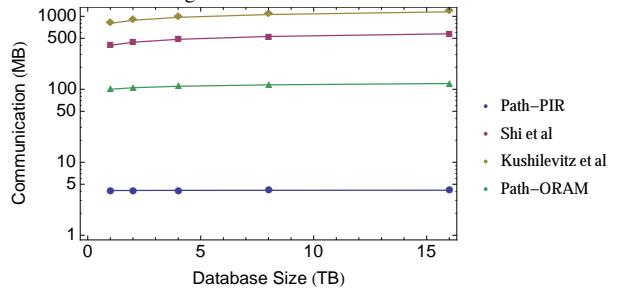


Fig. 6. Communication ReadAndRemove

nication cost of Shi et al. [14] and Path-PIR can be exactly computed based on N and l . Related work requires no server computation, so we modeled cost based on communication alone. To estimate communication time (download/upload), we assume an 88 Mbps connection, in line with the maximum speed one would expect when transferring from Amazon S3 [11]. This is a very generous estimate, and our scheme compares even more favorably in bandwidth constrained environments. Computation time for Path-PIR is calculated by benchmarking on buckets of various sizes (using EC2).

Path-ORAM We include comparison with Path-ORAM [15] (even though it is not constant-memory) since it is currently the most efficient known ORAM protocol. We show that our scheme is very competitive with Path-ORAM while requiring only constant client memory.

Figures 3, 4, and 5 show relative communication, time and monetary cost per read/write operation for related work 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 efficient relative to related work. Note that our scheme has even lower bandwidth requirements than Path-ORAM and very competitive monetary cost.

b) 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, since it requires transmitting only one full block. Even compared to the best related work in any memory setting, Path-PIR obtains vastly better latency.

VII. CONCLUSION

Outsourcing sensitive data to untrusted clouds requires 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 communication 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.

REFERENCES

- [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] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. ISSN 0004-5411.
- [4] Michael T Goodrich, Michael Mitzenmacher, Olga Ohri menko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–167. SIAM, 2012.
- [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 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.
- [10] Travis Mayberry, Erik-Oliver Blass, and Agnes Hui Chan. Pixmap: Efficient private information retrieval for mapreduce. *IACR Cryptology ePrint Archive*, 2012:398, 2012.
- [11] Nasuni. State of Cloud Storage Providers Industry Benchmark Report, 2011. http://cache.nasuni.com/Resources/Nasuni_Cloud_Storage_Benchmark_Report.pdf.
- [12] Path-PIR. Source Code, 2013. Anonymized.
- [13] 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.
- [14] 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.
- [15] Emil Stefanov, Martin van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyou Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol, 2013. <http://eprint.iacr.org/2013/280.pdf>.
- [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] Emil Stefanov and Elaine Shi. Oblivystore: High performance oblivious cloud storage. In *Proc. of IEEE Symposium on Security and Privacy*, 2013.
- [18] Techcrunch. Google Confirms That It Fired Engineer For Breaking Internal Privacy Policies, 2010. <http://techcrunch.com/2010/09/14/google-engineer-spying-fired/>.
- [19] 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.