PIRMAP: Efficient Private Information Retrieval for MapReduce

Abstract. Private Information Retrieval (PIR) allows for retrieval of bits from a database in a way that hides a user's access pattern from the server. However, its practicality in a real-world cloud computing setting has recently been questioned. In such a setting, PIR's enormous computation and communication overhead is expected to outweigh any cost saving advantages of cloud computing. This paper presents PIRMAP, a practical, highly efficient protocol for PIR in MapReduce, a widely supported cloud computing API. PIRMAP focuses especially on the retrieval of large files from the cloud, where it achieves close to optimal communication complexity with query times significantly faster than previous schemes. To achieve this, PIRMAP extends Lipmaa's PIR protocol to allow for optimal parallel computation during the "Map" phase of MapReduce, and homomorphic aggregation in the "Reduce" phase. To improve computational cost, we employ a new, faster "somewhat homomorphic" encryption, making our scheme practical for databases of useful size while still keeping communication costs low. PIRMAP has been implemented and tested in Amazon's public cloud with total database sizes of up to 1 TByte. Our performance evaluation shows that non-trivial PIR such as PIRMAP can be more than one order of magnitude cheaper and faster than trivial PIR in the real-world. PIRMAP adds only 20% overhead to a theoretical optimal PIR indicating its efficiency.

1 Introduction

Cloud computing has recently been identified as an important strategic technology [9], as it offers the advantage of greater flexibility and potentially reduced costs to companies outsourcing their data and computation. The cost advantage of cloud computing comes from the fact that cloud users do not need to maintain their own, expensive data center, but instead can pay a cloud provider for hosting. However, despite the hype, hesitations remain among major organizations. Lack of security and privacy guarantees has been identified as a major adoption obstacle for both large enterprise [19] and governmental organizations [14].

Privacy problems stem from the fact that, when a cloud user hands over his information to the cloud, control of that data is relinquished to a third party. Public clouds are threatened by hackers due to the much larger target they present. Insiders such as data center administrative staff can also easily access private data that has been outsourced. As multiple cloud users are hosted on the same data center ("multi tenancy"), even other cloud users might try to illegally access data. Finally, as cloud providers place data centers abroad in countries with unclear privacy laws, local authorities become an additional threat to outsourced data. Such attacks are realistic and have already been reported [10, 22]. This shows that users need to take special precautions to ensure the privacy and integrity of their cloud data.

In this paper, we envision an application scenario where a cloud user stores large amounts of data (files) with a cloud provider and subsequently wishes to retrieve a subset of that data. Such a scenario is realistic; for example, many hospitals are currently looking to take advantage of cloud storage by outsourcing patients' sensitive health records [20]. One can imagine that doctors upload their patients' health records to a public cloud provider for long term storage and occasionally retrieve those records in the course of daily operations. In the face of an untrusted cloud provider, the privacy of patients has to be protected.

While encryption of data at rest helps to protect data confidentiality, it is often not sufficient: the subsequent data access patterns can leak information about patients. For example, if the outsourced data in question contains encrypted patient records, the cloud provider might learn that a patient has been diagnosed with cancer when it sees that the patient's records have been retrieved by an oncologist. In general, it is hard to assess which information may be leaked, what inferences an adversary may make, and what risks the leakage may incur.

Private Information Retrieval (PIR) offers a solution to the information leakage problem by hiding access patterns [5, 12], independent of an encryption mechanism. In PIR, a database stores n strings each of size l bits, and a user can query for one l bit string without leaking *which* string to the database. A trivial way to accomplish this is to simply transmit the entire database back to the user. However, this is *communication* inefficient, and research focuses on achieving better communication bounds. In contrast, *computational* cost of every cPIR scheme is necessarily $O(n \cdot l)$, because the server must touch every piece of the database if it is to remain oblivious of the requested piece.

In existing work, the server's computation is comprised of expensive cryptographic operations over the entire database. Because of the significant overhead this imposes, it has recently been questioned whether PIR will ever become *practical* in a real-world cloud computing setting, cf. Sion and Carbunar [18]. Cloud providers such as Amazon charge their customers for both data transfer *and* CPU hours [2]. Due to the necessary condition that PIR protocols compute over the entire database for each query, it has been argued that trivial PIR (retrieving the whole $(l \cdot n)$ bit database) is not only faster, but also *cheaper* for the cloud customer compared to a PIR query that involves lengthy computation [4, 16, 18].

Another open question is how to perform PIR in a real-world cloud computing environment. One of the biggest challenges in cloud computing is writing application code, so it can scale easily to the large distributed systems which are characteristic in such a setting. In order to alleviate this difficulty, major cloud providers (e.g., Amazon, Google, IBM, Microsoft) offer an interface to the prominent MapReduce [7] API for distributed computing to their users. MapReduce comprises not only parallelization ("Map") of work, but an aggregation ("Reduce") of individual results to keep computational burden on the user side low. Existing PIR schemes with low communication costs require an iterative, round based evaluation on the server, which is not conducive to MapReduce because of the relatively large amount of time it takes to start computation. MapReduce is most efficient in parallel settings that require little synchronization.

This paper considers the single-server, computationally-private information retrieval setting. This is appropriate, because, although a cloud provider may allow access to many servers, they must all be considered a single "trust entity": they are under the control of the same organization. We view the cloud as a single, large server with many distributed CPUs. In the single-server setting, it is known that unconditionally secure PIR cannot be more efficient than transferring the entire database [6], so we are concerned instead with a computationally secure protocol. Further use of the term PIR will be in reference to computationally-secure PIR, unless otherwise noted.

This paper presents PIRMAP, an efficient single-server cPIR protocol suited for MapReduce clouds. PIRMAP especially targets retrieval of relatively large files, a more specific setting than considered by previous work. In a scenario with n files each of size l bits and $l \gg n$, PIRMAP achieves communication complexity linear in l with low constants. PIRMAP is designed for and leverages MapReduce parallelization and aggregation. We have implemented PIRMAP in Hadoop MapReduce, and its performance will be presented in Section 5.

Our contributions in this paper are:

1.) PIRMAP, an efficient PIR scheme for cloud computing with optimal communication complexity ($O(l)$) when retrieving an l bit file for $l \gg n$. Additionally, our scheme has asymptotic computational complexity as good or better than prominent related work. PIRMAP runs on top of standard MapReduce, not requiring changes to the underlying cloud infrastructure.

2.) An implementation of PIRMAP that is usable in real-world MapReduce clouds today, e.g., Amazon. We evaluate PIRMAP, first, in our own (tiny) local cloud and, second, with Amazon's cloud. We verify its practicality by scaling up to 1 TByte of data in Amazon's cloud setting. Compared to the previously largest single database PIR experiment with up to 28 GByte of data [16], this demonstrates the efficiency and practicality of PIRMAP in the real-world. PIRMAP is more than one order of magnitude *cheaper* and *faster* than trivial PIR, and – in our particular scenario – we can significantly outperform related work. Compared to a theoretical, but unrealistic lower bound PIR, PIRMAP adds only 20% overhead on Amazon. PIRMAP's source code is available for download [17].

2 Problem Statement

PIR: A PIR protocol is a series of interactions between a user and a server that results in the user retrieving one file out of n, of his choice, while the server does not learn which file this was. More formally, with n files the server cannot guess with probability greater than $1/n$ which file was queried. Note that this probability does not increase over multiple queries. This effectively hides the user's access pattern as each query is computationally indistinguishable from the others. The only information leaked is the number of records queried.

A $CPIR_l^n$ protocol ($\mathsf{Query}, \mathsf{Transfer}, \mathsf{Recover}$) is a computationally-secure private information retrieval protocol over a database of n elements, each of bit length l. The Query function generates the query and sends it to the server. Transfer is run on the server and involves transforming the received query into the results of the query and sending it back to the client. Finally, Recover is run by the client to transform the response from the server into the correct plaintext query result.

MapReduce: With the trend towards more computers and more cores rather than faster individual processors, it is important that any practical PIR implementation be deployable in a way that can take full advantage of parallel and cluster computing environments. Perhaps the most widely adopted architecture for scaling parallel computation in public clouds today is Google's MapReduce [7]. Its design allows for a set of computations, or "job", to be deployed across many nodes in a cloud cloud. Its biggest advantage is that it scales transparently to the programmer. That is, once an implementation is written using MapReduce, it can run on any number of nodes in the data center, from one up to hundreds or thousands, without changes in the code. This is managed by splitting computation into two phases, each of which can be run in parallel on many computing nodes.

The first phase is called the "Map" phase. MapReduce will automatically split the input to the computation equally among available nodes in the cloud data center, and each node will then run a function called *map* on their respective pieces (called *InputSplits*). It is important to note that the splitting actually occurs when the data is uploaded into the cloud (in our case when the patient record/files are uploaded) and not when the job is run. This means that each "mapper" node will have local access to its InputSplit as soon as computation has started and you avoid a lengthy copying and distributing period. The map function runs a user defined computation on each InputSplit and outputs a number of *key-value* pairs that go into the next phase.

The second phase, "Reduce", takes as input all of the key-value pairs emitted by the mappers and sends them "reducer" nodes in the data center. Specifically, each reducer node receives a single key, along with the sequence of values output by the mappers which share that key. The reducers then take each set and combine it in some way, outputting a single value for each key.

Despite being widely used, MapReduce is a very specific computational model and not all algorithms can be easily adapted to it. A practical PIR such as PIRMAP has to take its specifics into account – as we will see below.

3 Related Work

Despite the large amount of theoretical research, there has not been much investigation into practical PIR. Recently, Trostle and Parrish [21] compared the state of the art and found that at least it took 8 minutes to retrieve a file of size only 3 MB, out of a data set of 3 GB, on commodity hardware. Additionally, existing schemes have significant communication overhead for files of that size and larger. For example, Lipmaa [13], enjoying the most efficient asymptotic communication complexity today, requires 30 MB of communication in the previous case.

One of the first cPIR schemes to achieve sublinear communication is that of Kushilevitz and Ostrovsky [12]. This is achieved using an additively homomorphic cipher $\mathcal{E}$ as follows:

1. Arrange n elements in a $\sqrt{n} \times \sqrt{n}$ matrix
2. $\mathsf{Query}(x)$ – Client generates a vector $\boldsymbol{v}$ of length $\sqrt{n}$ where $\boldsymbol{v}_x = \mathcal{E}(1)$ and $\boldsymbol{v}_i = \mathcal{E}(0), \forall i \neq x$

3. $\mathsf{Transfer}(\boldsymbol{v})$ – The server multiplies the $\boldsymbol{v}$ by the database matrix and returns the result as $\boldsymbol{v}'$
4. $\mathsf{Recover}(\boldsymbol{v}')$ – $\boldsymbol{v}'$ now consists of $\sqrt{n}$ encrypted elements, one of which is x. The client chooses this element and decrypts it, discarding the rest.

The scheme is sound, because the client sends a vector, $\boldsymbol{v}$ which "zeroes out" all the rows in the matrix but the row requested. The communication costs is $O(l \cdot \sqrt{n})$, which is still quite high for large values of n. They also show that this protocol can be repeated recursively to achieve communication less than $O(n^c)$ for any $c > 0$, but at the cost of worse constants and computational complexity.

Lipmaa [13] reduces communication complexity to $O(l \cdot \log n + k \cdot \log^2 n)$, where k is a security parameter. This is accomplished by generalizing the Kushilevitz and Ostrovsky [12] scheme into a "family" of protocols, parameterized by a dimension α. The Kushilevitz and Ostrovsky [12] scheme can be seen as two-dimensional, because the database elements are arranged in a $\sqrt{n} \times \sqrt{n}$ matrix. When $\alpha = \log(n)$, the database is viewed as a $\log(n)$ dimensional $2 \times 2 \times \ldots \times 2$ matrix. In this manner, the client may send a sequence of $\log(n)$ vectors, each only length 2, which "zero out" half of the remaining database elements at each step. It is conceptually similar to specifying a path in a binary tree, and when the leaf node is reached the only element that will remain non-zero is the one corresponding to that node.

The down side of this scheme is that it requires computing over the entire database *twice* ($\sum_{i=1}^{\log n} \frac{1}{2^i} = 2$). While not a significant problem asymptotically, in practice we will see that cPIR schemes are usually bottlenecked by their computational cost. Increasing this cost by 100% dramatically effects the efficiency of the scheme, especially given the cloud setting where twice the computation corresponds to twice the cost. Another problem is that the computational complexity of Lipmaa [13] is $O(n \cdot l \cdot k^2)$, because it requires modular exponentiations on ciphertexts of size k. This is a significant overhead that we can avoid by using a more efficient homomorphism.

Although Lipmaa [13] requires only one round of communication with the user, it is somewhat iterative in that it requires $\log(n)$ rounds of computation on the server, each round depending on the output of the previous round and operating on a smaller set of data. As we will see, a large part of the cost for MapReduce is in initializing the parallel computation, so restarting $log(n)$ times would add a significant overhead.

More recently, Aguilar-Melamine and Gaborit [1] have proposed a scheme using lattice-based homomorphic encryption (similar to Hoffstein et al. [11]) which has much better computational complexity, both asymptotically and in practice. Their scheme also takes advantage of the fact that modern GPUs can solve large parallel problems very quickly. They are able to achieve very fast query response times, but at the cost of a relatively large communication cost. Additionally, their protocol is tuned very specifically to the requirements of GPUs and would not scale well in a distributed environment like MapReduce.

Using a different encryption scheme based on the Trapdoor Group Assumption, Trostle and Parrish [21] propose a more traditional PIR scheme based on Kushilevitz and Ostrovsky [12]. They also achieve much faster query response, but again, with large communication costs.

Comparison of existing work To understand how existing work compares and is suited to the application targeted in this paper, we start by examining asymptotic computation and communication complexities in Table 1. We can see that Lipmaa [13] has very good communication complexity with relatively bad computational complexity, while Aguilar-Melamine and Gaborit [1] have the opposite. However, this does not adequately describe the costs involved with PIR, because we are concerned primarily with a concrete, practical setting.

Consequently, in Table 2, we present analytically calculated communication costs for specific database and file sizes. Aguilar-Melamine and Gaborit [1] state that their requests are of size $25kb \cdot n$ and the responses are $6 \cdot l$. For databases with $l > nk$, Lipmaa [13] requires transmitting $\log^2(n)$ elements of size k in the request and receiving a response of size $\log(n) \cdot l$. We chose to use $k = 2048$, as suggested by Lipmaa [13].

In Table 2, we can see that even though Lipmaa [13] has good asymptotic communication, there is room for improvement in a concrete setting, especially for databases composed of large files. Aguilar-Melamine and Gaborit [1], similarly to PIRMAP, is designed specifically for these types of databases but it also has relatively high communication costs due to large constants.

Approach In conclusion, we would like to achieve, in real world settings, the small communication cost of Lipmaa [13] with the fast query response times of Aguilar-Melamine and Gaborit [1] and Trostle and Parrish [21]. Lipmaa notes that, when $\alpha = 1$, the PIR scheme is particularly efficient for databases where $l > n \cdot k$. We choose this case as a base for our new protocol PIRMAP, because it requires only one iteration of the database during Transfer and has communication complexity of $O(l + n \cdot k)$. When $l > n \cdot k$, this is optimal, since the server response must always be of size l. In short, PIRMAP can be thought of as a combination of this protocol with the fast encryption scheme used by Trostle and Parrish [21]. This new “somewhat” homomorphic encryption scheme requires only modular multiplications rather than exponentiations, significantly improving the asymptotic and real world computational requirements.

	Communication	Computation
Kushilevitz and Ostrovsky [12]	$O(\sqrt{n \cdot l})$	$O(l \cdot n \cdot k^2)$
Lipmaa [13]	$O(l \cdot \log(n)^2)$	$O(l \cdot n \cdot k^2)$
Aguilar-Melamine and Gaborit [1]	$O(l + n \cdot k)$	$O(l \cdot n \cdot \log(k) \cdot \log(\log(k)))$
PIRMAP	$O(l + n \cdot k)$	$O(l \cdot n \cdot \log(k) \cdot \log(\log(k)))$

Table 1: Communication and computational complexities of related work.

4 PIRMAP

PIRMAP extends the PIR protocol by Lipmaa [13], specifically addressing the shortcomings perceived in regards to retrieval of large files in a parallelization-aggregation computation framework such as MapReduce. We start by giving an overview of PIRMAP which can be used with *any* additively homomorphic encryption scheme.

	1 MB Files			5 MB Files			10 MB Files		
	1 GB	10 GB	100 GB	1 GB	10 GB	100 GB	1 GB	10 GB	100 GB
[13]	10	13	17	38	55	71	66	100	133
[1]	29	240	2350	35	77	499	62	83	295
PIRMAP	3	6	38	11	12	18	22	22	26

Table 2: Real communication costs of related work in MB for different database sizes.

Upload. In the following, we assume that the cloud user has already uploaded its files into the cloud using the interface provided to them by the cloud provider.

Query. In keeping with standard PIR notation, our data set holds n files, each of which is l bits in length. There is also an additional parameter k which is the block size of the chosen cipher. For ease of presentation, we consider the case where all files are the same length, but PIRMAP can easily be extended to accommodate variable length files by using padding. Our scheme can be summarized as follows:

1. Query – If the user wishes to retrieve file $1 \leq x \leq n$ out of the n files, it creates a vector $\boldsymbol{v} = (v_1, \ldots, v_n)$, where $v_x = \mathcal{E}(1)$ and $v_i = \mathcal{E}(0) \forall i \neq x$. Here, $\mathcal{E}$ is any probabilistic, indistinguishable, additively homomorphic encryption scheme. The user sends $\boldsymbol{v}$ to the cloud.
2. Multiply – The cloud arranges files into a table $\mathcal{T}$ similar to Table 3. The cloud divides each file i into $\frac{l}{k}$ blocks $\{B_{i,1}, \ldots, B_{i,\frac{l}{k}}\}$ and multiplies each block by v_i, i.e., $B'_{i,j} = v_i \cdot B_{i,j}$. Here, “$\cdot$” denotes scalar multiplication.
3. Sum – The cloud adds column-wise to create one result vector $\boldsymbol{r} = (r_1, \ldots, r_{\frac{l}{k}})$. Each element r_i of that vector is of size k, so the total bit length of $\boldsymbol{r}$ is l bits. Vector $\boldsymbol{r}$ is an encryption of file x. Vector $\boldsymbol{r}$ is returned to the user who decrypts it.

	1	2	$\cdots$	$\frac{l}{k}$
$file_1$	$B_{1,1}$	$B_{1,2}$	$\cdots$	$B_{1,\frac{l}{k}}$
$file_2$	$B_{2,1}$	$B_{2,2}$	$\cdots$	$B_{2,\frac{l}{k}}$
$\cdots$		$\cdots$		
$file_n$	$B_{n,1}$	$B_{n,2}$	$\cdots$	$B_{n,\frac{l}{k}}$

Table 3: Cloud splits files into pieces

The cloud effectively performs the matrix-vector multiplication $\boldsymbol{r} = \mathcal{T}^T \cdot \boldsymbol{v}$, where $\mathcal{T}^T$ is $\mathcal{T}$ transposed. In practice the cloud does not need to create a table, but just needs to perform the block wise scalar multiplications and additions.

The computation consists of multiplying each of the $n \cdot \frac{l}{k}$ blocks by a value in the request vector and then performing $(n-1) \cdot \frac{l}{k}$ additions to obtain the final sum. The computational complexity of this scheme is $O(n \cdot l \cdot M(k) + (n-1) \cdot l \cdot A(k)$, where $M(k)$

is the cost of performing one scalar multiplication in the additively homomorphic crypto system chosen and $A(k)$ is the cost of one addition. The communication complexity can be broken down into two parts: user to cloud and cloud to user. The user sends a vector of size n containing ciphertexts of size k, so the bandwidth complexity user-cloud is $O(n \cdot k)$. The cloud sends back a vector of $\frac{l}{k}$ entries, each of size $O(k)$, making that complexity $O(l)$. Consequentially, the overall communication complexity is $O(n \cdot k + l)$.

While PIRMAP appears to be relatively basic compared to other PIR schemes [3, 13], it achieves better overhead in practice because of the specific values of n and l that we will be using. Existing work in PIR is efficient for data sets that have large values for n and comparatively small values for l. PIRMAP would do poorly in that setting, because the complexity would be dominated by n. However, PIRMAP's cloud-to-user communication is *optimal* at $O(l)$ because the cloud must send back a message at least the size of the file the user queries for. For values of l larger than $n \cdot k$, PIRMAP allows for complexity of $O(l)$ with small constants. We argue that in practice this is often true. For example, if a user has a 1 TB data set of 10 MB files, $\frac{l}{n} \approx 800$. In contrast, under the same conditions, arranging the files in a $\sqrt{n} \times \sqrt{n}$ matrix would result in a download cost of $1024 \cdot 10MB \approx 10GB$. Even if $n > l$, the actual communication costs are low for practical choices of the parameters, see Section 5. For many clouds, the user is not charged for uploads, only downloads. This is an additional benefit of our scheme because the query result (communication which the user is charged for) is always very close to optimal, even when the overall communication is not.

We believe that our assumption of $l > n$ is realistic and useful for many applications. Medical records may contain one or more images (x-ray, MRI, etc) which would make them several megabytes at the very least.

Optimization: Although the cloud is doing most of the computation in this scheme, the user is still required to generate a vector of ciphertexts of length n and then decrypt the resulting response. Encryption is relatively expensive for ciphers that would be used in this scheme, so it is a non-trivial amount of computation if users are low-powered devices such as smartphones. A way to alleviate this problem is to have a moderately powerful trusted server pre-generate vectors of ciphertexts and upload them to the cloud for the users to use. Optionally, it could be the low-powered device itself during an overnight downtime. This machine would generate m vectors of size n such that $V_{i,j} = \mathcal{E}(1)$ where $j = \mathrm{HMAC}(k, i)$ and $V_{i,j} = \mathcal{E}(0)$ otherwise. This allows the user to use one of these "disposable" vectors at query time and permute it so that the single $\mathcal{E}(1)$ is at the index of the file it wishes to retrieve. If k is a key shared between the user and trusted server, the user can efficiently locate $\mathcal{E}(1)$. The user then generates a description of a permutation which moves the $\mathcal{E}(1)$ value to the correct position and randomly shuffles all other locations. A description of this permutation is of size $n \cdot \log(n)$, which is smaller than the size of the vector for $k > 30$, so this also effectively front loads the upload cost of the query and makes response time faster.

Although the encryption scheme we use (see Section 4.2 below) can perform encryptions very quickly and does not require the use of this optimization, we point it out as a general improvement that could be used with our scheme if the homomorphic cipher was changed.

4.1 PIRMAP Specification

In our protocol, the cloud performs two operations: multiplication of each block by the corresponding value in the "PIR vector" $\boldsymbol{v}$, and column-wise addition to construct the encrypted file chosen by the user. These two stages translate exactly to map and reduce implementations respectively. The files will be distributed evenly over all participating nodes where the map function will split each file into blocks and multiply the blocks by the correct encrypted value. The output of these mappers is a set of key-value pairs where the key is the index of the block and the value is the product of the block and encrypted PIR value. These values are all passed on to the reducers, which take a set of values for each key (block position or column) and add them together to get the final value for each block.

Being interested in $file_x$, the user executes the above step Query to compute $\boldsymbol{v}$ which is sent to the MapReduce cloud. There, each mapper node evaluates Multiply on its locally stored file and generates key-values pairs for the reducer. The reducer simply computed the Sum step by adding all values with the same key and sends them back to the user. The user receives $\frac{l}{k}$ values of size k from the reducers that he decrypts to get $file_x$.

User

```
function GenQuery($n, x$)
    $v := \{\}$
    for $i = 1$ to $n$ do
        if $i = x$ then
            $v_i := \mathcal{E}(1)$
        else
            $v_i := \mathcal{E}(0)$
        end if
    end for
end function
```

Cloud

```
function Map($file, v$)
    for $i = 1$ to $n$ do
        $c := B_i \cdot v_i$
        $Emit(i, c)$
    end for
end function
function Reduce($key, v$)
    $total := 0$
    for $i = 1$ to $n$ do
        $total := total + v_i$
    end for
    $Emit(key, total)$
end function
```

4.2 Encryption Scheme

Since the map phase of our protocol involves multiplying every piece of the data set by an encrypted PIR value, it is important that we choose an efficient cryptosystem. Traditional additively homormorphic cryptosystems, such as Paillier's, use some form of multiplication as their homomorphism. That is, for elements a and b, $\mathcal{E}(a) \cdot \mathcal{E}(b) = \mathcal{E}(a + b)$. Since our map phase consists of multiplying ciphertexts by unencrypted scalars, we would actually have to do exponentiation of a ciphertext. Our scheme, and all PIR schemes, must compute on the whole data set, so it would be quite computationally intensive.

Our solution to this problem is to use the *somewhat homomorphic* encryption scheme introduced by Trostle and Parrish [21] which relies on the hardness of the trapdoor group assumption. True homomorphic encryption schemes support an unlimited number of computations without increasing the size of the ciphertexts. In contrast, this scheme results in ciphertexts which grow in size by $O(\log_2 n)$ bits for n additions. In return for this size increase, we can have an encryption scheme where the additive homomorphism is actually addition itself. This scheme to encrypt n bits with some security parameter $k > n$ is as follows:

$\mathsf{KeyGen}(1^k)$: Generate a prime m of k bits and a random $b < m$.
Encrypt $\mathcal{E}(x) = b \cdot (r \cdot 2^n + x) \mod m$, for a random r
Decrypt $\mathcal{D}(c) = b^{-1} \cdot c \mod m \mod 2^n$

This encryption has the desired homomorphic property $\mathcal{E}(a) + \mathcal{E}(b) = \mathcal{E}(a + b)$. This scheme is *somewhat homomorphic*, because it cannot support an unlimited number of additions. When two ciphertexts c_1 and c_2 are added, you can express the sum as

$$b \cdot (r_1 \cdot 2^n + x_1) + b \cdot (r_2 \cdot 2^n + x_2) = b \cdot ([r_1 + r_2] \cdot 2^n + x_1 + x_2).$$

If the inside term $(r_1 + r_2) \cdot 2^n + x_1 + x_2$ exceeds m and "wraps around", then it will not be decrypted correctly, because application of the modulus will cause a loss of information. The modulus m must be chosen large enough to support the number of additions expected to occur. To support t additions, m should be increased by $\log_2 t$ bits. Additionally, each scalar multiplication can be thought of as up to 2^n additions, meaning that the size of m must be doubled for each supported scalar product. For our PIR scheme, m must be chosen to be $O(2k + \log_2(n))$ to support the required homomorphic operations.

In return for the reasonable increase in ciphertext size caused by the larger modulus (about 300% in our evaluations in Section 5), we are able to do very efficient computations over the encrypted data. Additionally, encryption is equivalent to only two multiplications, an addition and a modular reduction, while decryption is one multiplication and a reduction. This compares very favorably with other homomorphic encryption schemes, such as Paillier, requiring a modular exponentiation.

With our encryption scheme defined, we can now express more precise computational complexities for the protocol. Our previous complexity was parameterized over $M(k)$ and $A(k)$, the complexity of scalar multiplication and addition respectively in the cryptosystem used. For our encryption, addition is simply regular integer addition. Since each cipher text is at most $2k + \log_2(n)$ bits long, addition is $O(2k + \log_2(n))$. We can do scalar multiplication as integer multiplication as well. Integer multiplication can be done for m bits in $O(m \log(m) \log(\log(m)))$ [8], so $M(k) = O(2k + \log_2(n) \log(2k + \log_2(n)) \log(\log(2k + \log_2(n))))$. The complexity is then dominated by the multiplication cost and results in:

$$O(n \cdot \frac{l}{k} \cdot (k + \log(n)) \log(k + \log(n)) \log(\log(k + \log(n)))) \tag{1}$$

$$= O((n \cdot l + n \cdot \log(n)) \log(k + \log(n)) \log(\log(k + \log(n)))) \tag{2}$$

If $l > n$, then $l > \log(n)$, and we can simplify to $O(n \cdot l \log(k + \log(n)) \cdot \log(\log(k + \log(n))))$. Additionally, k has to be much larger than $\log(n)$ (otherwise we are implying that the server will have the resources to find the key by brute force), allowing us to simplify further to $O(n \cdot l \cdot \log(k) \cdot \log(\log(k)))$.

4.3 Privacy Analysis

PIRMAP inherits privacy properties of the work it is based on, i.e., Lipmaa [13] and the PIR variant by Trostle and Parrish [21]. In the following, we sketch our privacy rationale.

PIRMAP is privacy-preserving, **iff** an adversary (the cloud) cannot guess, after each query, with probability greater than $1/n$, which file was retrieved by the user after an invocation of the protocol. There are two pieces of information that the adversary has access to: the set of uploaded files and the vector $\boldsymbol{v}$ of PIR values. If the files are each individually encrypted with an IND-CPA cipher, e.g., AES-CBC, they are computationally indistinguishable from random to the adversary. That means any attack that could break PIRMAP using the information in the database could also break it when run with a simulator that generates a set of random files. Therefore, privacy depends only on $\boldsymbol{v}$.

Vector $\boldsymbol{v}$ contains many encryptions of “0” and one encryption of “1”. The problem of determining which file was selected is then equivalent to *distinguishing* between encryptions of “0” and encryptions of “1” in the underlying encryption. However, the scheme we use is provably secure against distinguishing under the Trapdoor Group Assumption [21]. Consequently, PIRMAP preserves user privacy.

5 Evaluation

We have evaluated the performance of our scheme in three contexts: a local “cloud” (a single server with multiple CPUs), a commodity laptop, and Amazon’s EC2 cloud using Elastic MapReduce [2]. We have implemented PIRMAP in Java for standard Hadoop MapReduce version 1.0.3 and the source code is available for download [17].

5.1 Setup

Local We used a local server to prototype and debug our application and to do detailed timing analysis requiring many runs of MapReduce. This server, running Arch Linux 2011.08.19, has dual 2.4 GHz quad-core Xeon E-5620 processor and 48 GB of memory. Based on specs and benchmark results, our local server is closest to an “EC2 Quadruple Extra Large” instance, which has dual 2.9 GHz quad-core Xeon X-5570 processors and 24 GB of memory.

We have measured the time needed for PIR queries, i.e., the time to upload PIR vector $\boldsymbol{v}$ plus the time to process the query and download the result. Using Amazon’s standard cost model, we have calculated the price of each PIR query as the amount of money required to run the query on one of the above EC2 instances [2] (for the same amount of time it took to run locally) plus the bandwidth cost of downloading the results [2]. Since uploading data to Amazon is free, that did not add any additional cost. To put our measurements into perspective, we have also included the time and cost of two other, hypothetical, PIR protocols. We have implemented a *Baseline*, which does not perform *any* cryptographic operations and merely “touches” each piece of data through the MapReduce API. This measure shows the theoretical *lower bound* of computation and time required for *any* PIR scheme that uses MapReduce, independent of the encryption and exact PIR method used.

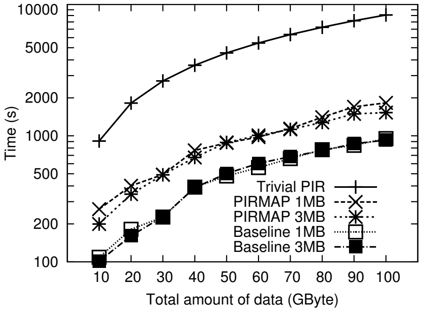

Fig. 1: Time per query, local server

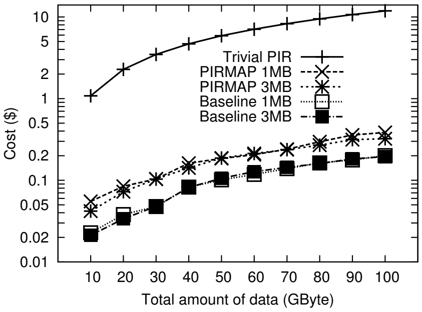

Fig. 2: Cost per query, local server

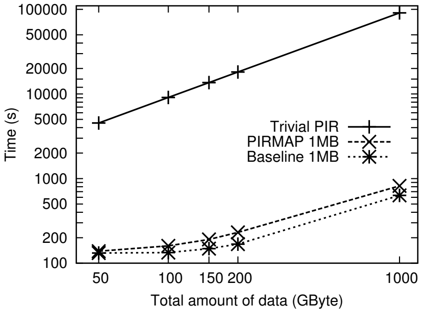

Fig. 3: Time per query, Amazon Elastic MapReduce

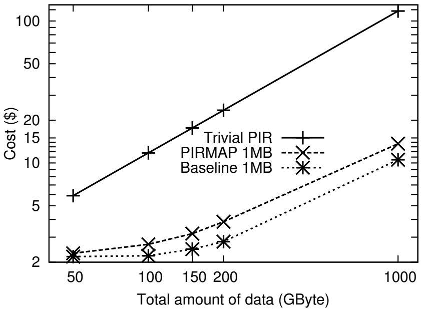

Fig. 4: Cost per query, Amazon Elastic MapReduce

To highlight the advantage of computational PIR, we have also included the time and cost required for the “trivial” PIR scheme. The trivial scheme is one where the user downloads the entire data set and simply discards the files he is not interested in. This is very bandwidth intensive, but computationally lightweight. Sion and Carbunar [18] conjecture this trivial PIR to be the most cost effective in the real-world. We have calculated the cost based on the amount that Amazon charges to download the corresponding amount of data and the time based on a 11.28 Mbps connection, an average as reported by Nasuni [15]. Note that we generally do not count the cost for long-term storage of data at Amazon. Although potentially significant for large amounts of data, the user has to pay for this regardless of whether he wants queries to be privacy-preserving or not. PIRMAP does not increase the amount of storage in Amazon.

Laptop We also ran our implementation on a 2012 MacBook Pro with a 2.3 Ghz i7, 8 GB of RAM and an NVIDIA GeForce 650M. This test was to compare with related work and the results are shown in table 4. The Aguilar scheme takes advantage of GPU resources so we chose a machine with comparable graphics and CPU resources (unlike the above server, which had no discrete graphics). This represents the performance you would get on a commodity machine and also shows that query generation is not very taxing.

We obtained our timing information for PIRMAP and the Aguilar scheme by running our implementation and their respectively on the above machine. To compare with Lipmaa [13], we used `openssl speed rsa` to time determine how many RSA private key operations could be computed per second. Each scalar multiplication in that scheme is equivalent to one modular exponentiation, or one RSA private key operation. This estimation is quite generous because it does not include disk access or homomorphic additions (modular multiplication), but we believe it is close because the time will be largely dominated by the exponentiations.

	5 GB	10 GB	15 GB	20 GB	25 GB	30 GB
Lipmaa [13]	1852	3704	5508	7312	9116	10920
Aguilar-Melamine and Gaborit [1]	4	7	11	14	17	20
PIRMAP	1	2	3	3	4	4

Table 4: Query time in minutes, including generation, evaluation, and decryption, for databases of varying sizes, composed of 5 MB files.

Amazon Besides the local experiments, to demonstrate the scalability of our scheme, we have also evaluated it on Amazon's Elastic MapReduce cloud. Amazon imposes a maximum limit of 20 instances per MapReduce job by default. In keeping with this restriction, we used 20 "Cluster Compute Eight Extra Large" instances which each having dual eight-core Xeon E5-2670 processors and 64 GB of RAM.

5.2 Results

Time and total cost: Figures 1 to 4 show our evaluation results. Figures 1 and 2 show the local evaluation, while figures 3 and 4 show evaluation with Amazon. In each figure, the x-axis shows the total amount of data stored at the cloud, i.e., number of files n times file size l. The y-axis shows either the time elapsed (for the whole query from the time the user submits the query until MapReduce returns the result back) or the cost implied with the query. In all four graphs, we scale the y-axis logarithmically, and in figures 3 and 4 we also scale the x-axis logarithmically. Each data point represents the average of at least 3 runs. Relative standard deviation was low at $\approx 5\%$.

To verify the impact of varying file sizes l, for our local evaluation, we show results with file sizes of 1 MB and 3 MB, attaining approximately equal runtime in both cases. This is to be expected because, in each data point, we have fixed the size of the database so varying retrieval sizes merely reshapes the matrix without changing the number of elements in it. The execution is dominated by the scalar multiplications that occur during the map phase, and the same number of those is required no matter the dimension of the matrix.

Our evaluation shows that PIRMAP outperforms trivial PIR in both time and cost by one order of magnitude. Compared to the theoretical, yet unrealistic optimum Baseline, PIRMAP introduces only 20% of overhead in the case of Amazon. Locally, we experience slightly larger overhead of 100%. This is because executing on Amazon has a much higher "administrative" cost due to the higher number of nodes and more distributed setting. These results indicate not only PIRMAP's efficiency over Trivial PIR and Baseline, but also its real-world practicality: in a small database comprising 10,000 patient records of size 1 MB each (10 GByte), a doctor can retrieve a single record in $\approx$ 3 min for only $\approx \$0.03$. In a huge data set with $1,000,000$ files, a single file can be retrieved in $\approx$ 13 min for $\approx \$14$. In the case where it is necessary to retrieve data in a fully privacy-preserving manner, we conjecture this to be acceptable.

Although a comparison with related research is not straightforward (as PIRMAP targets a very special scenario), we put our results into perspective with those of Aguilar and Lipmaa. We show that, while Lipmaa's scheme has very good communication complexity in all cases, it is completely impractical due to the enormous amount of computation needed to respond to queries. We also show that our scheme is comparable with that of Aguilar, in terms of computation, and beats it in communication cost by a significant margin.

Query Generation and Decryption: Due to the efficiency of the encryption in PIRMAP, PIR query generation is very fast. One ciphertext (element of $\boldsymbol{v}$) is generated for each file in the cloud, so the generation time is directly proportional to the number of files. We omit in-depth analysis, but in our trials on a commodity Macbook running on a single roce it takes about 2.5 seconds per 100,000 files in the cloud. Decryption is slightly more expensive than encryption, but we still managed, on the same machine, to decrypt approximately 3 MB per second. We conclude this overhead to be feasible for the real-world.

Bandwith: PIRMAP introduces bandwith overhead, 1.) to *upload* $\boldsymbol{v}$, and 2.) to *download* the encrypted version of the file. For security, we set $k = 2048$ bit, so each of the n elements of vector $\boldsymbol{v}$ has size 2048 bit. For a data set with 10,000 files (10 GByte), this requires the user to upload ≈ 2.5 MByte. As this can become significant with larger number of files, we suggest to then use the optimization in Section 4, especially for constrained devices.

6 Conclusion

Retrieval of previously outsourced data in a privacy-preserving manner is an important requirement in the face of an untrusted cloud provider. PIRMAP is the first practical PIR mechanism suited to real-world cloud computing. In the case, where a cloud user wishes to privately retrieve large files from the untrusted cloud, PIRMAP is communication efficient. Designed for prominent MapReduce clouds, it leverages their parallelism and aggregation phases for maximum performance. Our analysis shows that PIRMAP is an order of magnitude more efficient than trivial PIR and introduces acceptable overhead over non-privacy-preserving data retrieval. Additionally, we have shown that our scheme can scale to cloud stores of up to 1 TB on Amazon's Elastic MapReduce.

Bibliography

[1] C. Aguilar-Melamine and P. Gaborit. A Lattice-Based Computationally-Efficient Private Information Retrieval Protocol, 2007. http://eprint.iacr.org/2007/446.pdf.

[2] Amazon. Elastic MapReduce, 2010. http://aws.amazon.com/elasticmapreduce/.

[3] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Advances in Cryptology, EUROCRYPT*, pages 402–414, Prague, Czech Republic, 1999.

[4] Y. Chen and R. Sion. On securing untrusted clouds with cryptography. In *Workshop on Privacy in the Electronic Society*, pages 109–114, Chicago, USA, 2010.

[5] B. Chor, O. Goldreich, and E. Kushilevitz. Private Information Retrieval. In *Proceedings of Symposium on Foundations of Computer Science*, 1995.

[6] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 41 –50, oct 1995. doi: 10.1109/SFCS.1995.492461.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of Symposium on Operating System Design and Implementation*, pages 137–150, San Francisco, USA, 2004.

[8] M. Fürer. Faster integer multiplication. In *Proceedings of Symposium on Theory of Computing*, 1997.

[9] Gartner. Gartner Identifies the Top 10 Strategic Technologies for 2011, 2010. http://www.gartner.com/it/page.jsp?id=1454221.

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

[11] Jeffrey Hoffstein, Jill Pipher, and Joseph Silverman. Ntru: A ring-based public key cryptosystem. In Joe Buhler, editor, *Algorithmic Number Theory*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1998.

[12] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings of Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[13] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. In *Conference on Information Security*, pages 314–328, Taipei, Taiwan, 2005.

[14] D. McClure. GSA's role in supporting development and deployment of cloud computing technology, 2010. http://www.gsa.gov/portal/content/159101.

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

[16] F.G. Olumofin and I. Goldberg. Revisiting the Computational Practicality of Private Information Retrieval. In *Proceedings of Financial Cryptography*, 2011.

[17] PIRMAP. Source Code, 2012. Anonymized for blind submission.

[18] R. Sion and B. Carbunar. On the Computational Practicality of Private Information Retrieval. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 1–10, San Diego, USA, 2007.

[19] Symantec. State of Cloud Survey, 2011. http://www.symantec.com.

[20] The Telegraph. Patient records go online in data cloud, 2011. http://www.telegraph.co.uk/.

[21] 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.

[22] Z. Whittaker. Microsoft admits Patriot Act can access EU-based cloud data, 2011. http://www.zdnet.com/.