A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost

Changyu Dong[1] and Liqun Chen[2]

[1] Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK
changyu.dong@strath.ac.uk
[2] Hewlett-Packard Laboratories, Bristol, UK
liqun.chen@hp.com

Abstract. Existing single server Private Information Retrieval (PIR) protocols are far from practical. To be practical, a single server PIR protocol has to be both communicationally and computationally efficient. In this paper, we present a single server PIR protocol that has low communication cost and is much faster than existing protocols. A major building block of the PIR protocol in this paper is a tree-based compression scheme, which we call folding/unfolding. This compression scheme enables us to lower the communication complexity to $O(\log \log n)$. The other major building block is the BGV fully homomorphic encryption scheme. We show how we design the protocol to exploit the internal parallelism of the BGV scheme. This significantly reduces the server side computational overhead and makes our protocol much faster than the existing protocols. Our protocol can be further accelerated by utilising hardware parallelism. We have built a prototype of the protocol. We report on the performance of our protocol based on the prototype and compare it with the current most efficient protocols.

Keywords: Private Information Retrieval, Fully Homomorphic Encryption, Privacy

1 Introduction

Private Information Retrieval (PIR) is an important primitive with many applications. A PIR protocol allows a client to retrieve information from a database without revealing what has been retrieved. We have seen PIR being applied in areas such as location-based services [1] and e-commerce [2]. There are two types of PIR protocols: multi-server PIR [3] and single server PIR [4]. In a multi-server PIR protocol, the database is replicated to multiple servers and the queries will be answered jointly by the servers. In a single server PIR protocol, only one server hosts and serves the database. In this paper, we consider single server PIR. It is well-known that designing a non-trivial yet practical single server PIR protocol is a challenging task. For single server PIR, there exists a trivial protocol such that the server simply sends the whole database to the client. Therefore the first design criteria for non-trivial single server PIR protocols is to have sub-linear communication complexity. Traditionally, research in single server PIR focused almost entirely on how to minimise the communication cost [4–12]. However, low communication cost does not mean the protocols are practical. As pointed out by Sion et al [13], due to costly server side computation, single server PIR protocols are often slower than the trivial solution despite that they transmit less bits. The most computationally efficient PIR protocol to date [4] requires n big integer modular multiplications, where n is the size of the database. The computation time of each operation is often much more significant than simply transmitting a bit. Therefore, all single server PIR protocols can be easily beaten by the trivial solution even when the network bandwidth is only a few hundred Kbps (300 in Sion's experiment). How to make the server side computation

faster has become another important consideration.

There has been some work in reducing server side computation time. One approach is to use trusted hardware [14, 15]. Another approach is to base privacy on anonymity by mixing queries from different users [16]. Lipmaa proposed a BDD-based protocol [17] that is very efficient when the database is sparse, but in general case it requires $O(n/\log n)$ modular exponentiations, which is more expensive than n modular multiplications. Those approaches can improve performance but rely on extra assumptions. To the best of our knowledge, the only work that can significantly reduce server side computation time and without extra assumptions is [18]. Unfortunately as we will discuss in section 2, this protocol is not secure.

Contributions. In this paper, we present a fast single server PIR protocol with low communication cost. Our protocol belongs to the homomorphic encryption based PIR family [19]. Namely, we utilise the ring homomorphism provided by the BGV fully homomorphic encryption (FHE) scheme [20] to privately retrieve the bit. Communication wise, the protocol has low communication complexity $O(\log \log n)$. To achieve low communication, we designed a tree-based compression scheme called folding/unfolding. Computation wise, the protocol is much faster than all previous ones. We show how we design the PIR protocol to take advantage of the internal parallelism provided by the BGV FHE scheme, which allows us to amortise the server side computation. Most operations on the server side will be applied to $10^3 - 10^4$ bits in the database simultaneously. The amortised cost per bit is quite low: only around twelve 64-bit modular multiplications at 128-bit security. In contrast, per bit computational cost in previous protocols is one or more big integer modular multiplications (e.g. 3072-bit integers at 128-bit security). So overall, the server side computational overhead in our protocol is much lower. The security of our protocol is based on the security of the BGV FHE scheme, which is based on the well studied Ring Learning with Errors assumption [21].

We have implemented a prototype. We report performance measurements based on this prototype and make comparison with existing protocols. The performance test shows that our protocol consumes only a few hundreds KB bandwidth and is much faster than the previous fastest protocol by Kushilevitz et al [4]. For example, when the database is 4MB, our protocol consumes only 372 KB bandwidth and is 12 times faster than Kushilevitz's protocol; when the database is 4 GB, our protocol consumes only 423 KB bandwidth and is 90 times faster than Kushilevitz's protocol. With some hardware parallelism, our protocol can beat the trivial solution in 100 Mbps LAN.

2 Related Work

There has been abundant research in multi-server PIR, e.g. [3, 22–25]. We will not elaborate them here since our focus is single server PIR. In the single server case, Kushilevitz et al [4] proposed a protocol based on the Goldwasser-Micali homomorphic encrytion with communication complexity $O(n^\epsilon)$ for $\epsilon > 0$. This homomorphic approach is then generalised by Stern [5], Chang [8] and Lipmaa [10, 17] . Stern and Chang uses the Pallier's scheme [26] and the communication complexity is superpolylogarithmic. Lipmaa uses the Damgård-Jurik scheme [27]. The protocol can achieve $O(log^2 n)$ communication complexity. Our protocol follows this line and uses the BGV FHE scheme. Cachin et al proposed a PIR protocol that has polylogarithmic communication complexity ($O(\log^8 n)$) based on the ϕ-hiding assumption. Gentry et al [9]

generalised Cachin et al's approach and proposed a very communication efficient PIR protocol. The total communication cost of the protocol is 3 messages, each of the size of $\Omega(log^{3-o(1)}n)$ bits. Kushilevitz et al [7] showed a single server PIR protocol can also be based on one-way trapdoor permutations (TDPs). The communication complexity is $n - \frac{cn}{k} + O(k^2)$ bits, where c is a constant and k is the security parameter of the one-way trapdoor permutation. Sion et al [13] showed that the trivial single server PIR protocol often out-performed non-trivial ones. To improve computational efficiency, a few approaches have been taken. Williams et al [14] proposed a PIR protocol that has $O(log^2 n)$ server side computational complexity. However it requires trusted temper-resistant hardware. Similarly with trusted hardware, Ding et al [15] developed a protocol that requires $O(n)$ offline computation and constant online computation. Ishai et al [16] showed that anonymous communication could be used as a building block to implement more efficient single serve PIR protocols when there are multiple users. Melchor et al [18] proposed a lattice-based PIR protocol. However a practical attack by Bi et al [28] can be applied to break the security of this protocol. Namely the server can obtain the secret matrixes used to generate the request by constructing a reduced-dimension lattice and then recovers the index being queried.

Our protocol is based on FHE. It has been shown that PIR protocols with low communication can be easily obtained by using FHE. In [11], Brakerski et al proposed a generic PIR protocol that uses an FHE scheme with a symmetric encryption scheme. In the protocol, the client encrypts the index bit-by-bit using a symmetric key and encrypts the key using the FHE scheme. Then with the encrypted index and encrypted key, the server evaluates a circuit homomorphically to retrieve from its database the requested bit. The communication cost is $O(\log n)$ but the computational cost can be quite high because of the deep circuit. Gentry [29] proposed a PIR protocol. In the protocol, the client encrypts the index i bit-by-bit using FHE, then sends the ciphertexts to the server. The server homomorphically evaluates $\sum_{t=1}^{n} x_t \cdot \prod_{j=1}^{\lfloor \log n+1 \rfloor}(t_j - i_j + 1)$, where t_j, i_j are the jth bit of indexes t and i. This approach is also used by Yi et al [12], instantiated using the DGHV FHE scheme [31]. The communication complexity is $O(\log n)$ and the computational complexity is $O(n \log n)$. Our approach is different from previous FHE based PIR protocols, and better both in terms of computation and communication. Yi's paper showed better performance results than ours in their experiments. But in their experiments, γ was set to 2205, which should be at the level of 10^6 to prevent lattice based attack at the targeted security level. If the parameters were set correctly, then the performance of the protocol would be worse than ours.

3 Preliminaries

3.1 Notation

We use bit string and bit vector interchangeably. We use lower case bold face letters to denote vectors, e.g. $\mathbf{q}$. The vector indexes always start at 1. Depending on context, we use bit vectors as plain bit vectors or to represent binary polynomials or vectors of binary polynomials. In the folding/unfolding algorithms, bit vectors are plain bit vectors. On the server side, a query string is viewed as a vector of constant polynomials, i.e. $0 \cdot x^0$ or $1 \cdot x^0$. When encoding server's database, we view a bit vector as a binary polynomial in its coefficient form. Namely, a bit vector $\mathbf{a}$ of size d represents a binary polynomial $\sum_{i=1}^{d} \mathbf{a}_i x^{i-1}$, whose degree is at most $d - 1$. We use capital letters to

denote matrices, We denote the ith row of a matrix M by M_i, its jth column by M^j, and a single element at the ith row and the jth column by M_{ij}. Naturally, each row or column in a matrix can be viewed as a vector (not necessarily binary). The base of $\log$ is 2 throughout the paper.

3.2 Security Definition for Single Server PIR

A single server PIR protocol is between two parties: a server that has an n-bit database $\mathbf{x} = \mathbf{x}_1\mathbf{x}_2...\mathbf{x}_n$, a client that has some index $i \in [1, n]$. The client wants to obtain the ith bit $\mathbf{x}_i$ without revealing i. Any database can be represented in this string form by concatenating all records into a single bit string. The protocol consists of four algorithms:

1. **Init**: Takes as input a security parameter λ and the size n of the database, outputs a set of private parameters $\mathcal{S}$ and a set of public parameters $\mathcal{P}$, denoted as $(\mathcal{S}, \mathcal{P}) = \text{Init}(\lambda, n)$.
2. **QGen**: Takes as input $\mathcal{S}$, $\mathcal{P}$, the size n of the database, and the index i of the bit to retrieve, outputs a query $\mathcal{Q} = \text{QGen}(\mathcal{S}, \mathcal{P}, n, i)$.
3. **RGen**: Takes as input $\mathcal{Q}$, $\mathcal{P}$ and $\mathbf{x}$, outputs a response $\mathcal{R} = \text{RGen}(\mathcal{Q}, \mathcal{P}, \mathbf{x})$.
4. **RExt**: Takes as input $\mathcal{R}$, $\mathcal{S}$, $\mathcal{P}$, the index i and the size of the database n, extracts a bit $b = RExt(\mathcal{R}, \mathcal{S}, \mathcal{P}, i, n)$ such that $b = \mathbf{x}_i$.

In this paper, we consider a PIR protocol to be secure in the sense that it is computationally infeasible for an adversary to distinguish two queries. We say a function $\mu(\cdot)$ is *negligible in* n, or just *negligible*, if for every positive polynomial $p(\cdot)$ and any sufficiently large n it holds that $\mu(n) \leq 1/p(n)$. Formally the security of a single server PIR protocol is defined as follows:

Definition 1. *We say a single server PIR protocol is secure if for any PPT adversary $\mathcal{A}$, the advantage of distinguishing two queries is negligible:*

$$Pr\left[b' = b \;\middle|\; \begin{array}{c} (\mathcal{S}, \mathcal{P}) = \mathit{Init}(\lambda, n), \\ i_0, i_1 \leftarrow \mathcal{A}^{\mathcal{QGen}(\mathcal{S}, \mathcal{P}, \cdot, \cdot)}(\mathcal{P}, \mathbf{x}), \\ b \xleftarrow{R} \{0, 1\}, \\ \mathcal{Q} = \mathcal{QGen}(\mathcal{S}, \mathcal{P}, n, i_b), \\ b' \leftarrow \mathcal{A}^{\mathcal{QGen}(\mathcal{S}, \mathcal{P}, \cdot, \cdot)}(\mathcal{P}, \mathbf{x}, \mathcal{Q}) \end{array}\right] - \tfrac{1}{2} < negl(\lambda)$$

3.3 The BGV Fully Homomorphic Encryption

A homomorphic encryption scheme allows certain operations to be performed on ciphertexts without decrypting the ciphertexts first. In 2009, Gentry [30] developed the first FHE scheme. Following the breakthrough, several FHE schemes based on different hardness assumptions have been proposed, e.g. [31, 20, 32]. In this paper, we use the BGV FHE scheme [20]. We describe it here with improvements introduced in [33, 20, 34]. The security of this scheme is based on the ring-LWE (RLWE) [21] problem.

Let $\Phi_m(x)$ be the m-th cyclotomic polynomial with degree $\phi(m)$, then we have a polynomial ring $\mathbb{A} = \mathbb{Z}[x]/\Phi_m(x)$, i.e. the set of integer polynomials of degree up to $\phi(m) - 1$. Here $\phi(\cdot)$ is the Euler's totient function. The ciphertext space of the BGV encryption scheme consists of polynomials over $\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$, i.e. elements in $\mathbb{A}$ reduced modulo q where q is an odd integer[3]. The plaintext space is usually the ring $\mathbb{A}_2 =$

[3] In the BGV encryption scheme, there are actually a chain of moduli $q_0 < q_1 < \cdots < q_L$ defined for modulus switching. But for simplicity we just use q throughout the paper.

$\mathbb{A}/2\mathbb{A}$, i.e. binary polynomials of degree up to $\phi(m) - 1$. We also have the following distributions that we will use later in the key generation and encryption algorithms:

- $\mathcal{U}_q$: The uniform distribution over $\mathbb{A}_q$.
- $\mathcal{DG}_q(\sigma^2)$: The discrete Gaussian distribution over $\mathbb{A}_q$ with mean and variance $(0, \sigma^2)$.
- $\mathcal{ZO}(p)$: For a probability p, $\mathcal{ZO}(p)$ draws a polynomial in $\mathbb{A}_q$ such that each coefficient is 0 with a probability of $1 - p$, and is ± 1 with a probability of $p/2$ each.
- $\mathcal{HWT}(h)$: Uniformly draws a polynomial in $\mathbb{A}_q$ with exactly h nonzero coefficient and each nonzero coefficient is either 1 or -1.

The BGV encryption scheme has 3 basic algorithms (G, E, D):

- $G(\lambda, L)$: Given λ and L such that λ is the security parameter and L is the depth of the arithmetic circuit to be evaluated, the key generation algorithm chooses $\Phi_m(x), q, \sigma, h$, generates a secret key, the corresponding public key and a set of public parameters. Namely, we sample
$$s \leftarrow \mathcal{HWT}(h), a \leftarrow \mathcal{U}_q, e \leftarrow \mathcal{DG}_q(\sigma^2)$$
Then the secret key is $sk = s$ and the public key is $pk = (a, b) \in \mathbb{A}_q^2$ where $b = a \cdot s + 2e$. The public parameter set $param = \{m, \phi(m), q, \sigma, L, l\}$, where $m, \phi(m), q$ defines $\mathbb{A}_q$ and l is the number of plaintext slots (will explain later).
- $E_{pk}(m)$: Given $pk = (a, b)$, to encrypt an element $m \in \mathcal{A}_2$, we choose one small polynomial and two Gaussian polynomials:
$$v \leftarrow \mathcal{ZO}(0.5), e_0, e_1 \leftarrow \mathcal{DG}_q(\sigma^2)$$
Then we set $d_0 = b \cdot v + 2e_0 + m, d_1 = a \cdot v + 2 \cdot e_1$, the ciphertext is $c = (d_0, d_1)$.
- $D_{sk}(c)$: Given $sk = s$, to decrypt a ciphertext $c = (d_0, d_1)$, we compute $m = (d_0 - s \cdot d_1 \bmod q) \bmod 2$.

We denote homomorphic addition by $\boxplus$ and homomorphic multiplication by $\boxtimes$. At a high level, we can express the homomorphic operations as the following:

- Homomorphic Addition: Given two ciphertexts $c = E_{pk}(m)$ and $c' = E_{pk}(m')$ for $m, m' \in \mathbb{A}_2$, then $c_{add} = c \boxplus c' = E_{pk}(m + m')$.
- Homomorphic Multiplication: Given two ciphertexts $c = E_{pk}(m)$ and $c' = E_{pk}(m')$ for $m, m' \in \mathbb{A}_2$, then $c_{mult} = c \boxtimes c' = E_{pk}(m \cdot m')$.
- Homomorphic Addition (with a plaintext): Given a ciphertext $c = E_{pk}(m)$ and a plaintext m' for $m, m' \in \mathbb{A}_2$, then $c_{add} = c \boxplus m' = E_{pk}(m + m')$.
- Homomorphic Multiplication (with a plaintext): Given a ciphertext $c = E_{pk}(m)$ and a plaintext m' for $m, m' \in \mathbb{A}_2$, then $c_{mult} = c \boxtimes m' = E_{pk}(m \cdot m')$.

Apart from the above operations, the BGV scheme also has two maintenance operations: modulus switching and key switching. These two operations are used to control noise in cihpertexts and to keep ciphertext size down. We do not go into the details of them because they do not change the plaintext encrypted in a ciphertext. They can be viewed as background routines that are invoked automatically when necessary.

Another important feature of the BGV scheme is that it allows packing plaintexts and batching homomorphic computation. It was first observed in [33] that the native plaintext space $\mathbb{A}_2$ can be partitioned into a vector of plaintext slots. The idea is that although the ring polynomial $\Phi_m(x)$ is irreducible modulo q, it can be factorised into distinct factors modulo 2. More specifically, we can factor $\Phi_m(x)$ modulo 2 into l irreducible factors $\Phi_m(x) = F_1(x) \cdot F_2(x) \cdots F_l(x) \bmod 2$, each factor is of degree $d = \phi(m)/l$. So by the Chinese Remainder Theorem, a single element a in $\mathbb{A}_2$ can represent an l-vector $(a \bmod F_1(x), a \bmod F_2(x), ..., a \bmod F_l(x))$. In other words, we

have a mapping $\pi : \mathbb{F}_{2^d}^l \to \mathbb{A}_2$ that packs l elements in field $\mathbb{F}_{2^d}$ into a single element in $\mathbb{A}_2$. Then we can encrypt this packed plaintext as usual. The packed plaintext can be unpacked by the inverse mapping $\pi^{-1} : \mathbb{A}_2 \to \mathbb{F}_{2^d}^l$. For convenience, we use c in normal font to denote a ciphertext that encrypts a native element in $\mathbb{A}_2$, and use $\mathfrak{c}$ in Fraktur font to denote a packed ciphertext that encrypts an l-vector.

A homomorphic operation on packed ciphertexts adds or multiplies componentwise the entire plaintext vectors in an **SIMD** (single instruction multiple data) fashion. Namely, if we have two ciphertexts $\mathfrak{c} = E_{pk}(\pi(\mathbf{p}))$ and $\mathfrak{c}' = E_{pk}(\pi(\mathbf{p}'))$, where $\mathbf{p}$ and $\mathbf{p}'$ are plaintext vectors of size l. Then $\mathfrak{c}_{add} = \mathfrak{c} \boxplus \mathfrak{c}'$ encrypts $\pi(\mathbf{p}^+)$ such that $\mathbf{p}_i^+ = \mathbf{p}_i + \mathbf{p}'_i$, $\mathfrak{c}_{mult} = \mathfrak{c} \boxtimes \mathfrak{c}'$ encrypts $\pi(\mathbf{p}^\times)$ such that $\mathbf{p}_i^\times = \mathbf{p}_i \cdot \mathbf{p}'_i$. Similarly in the multiplication with a plaintext vector case, $\mathfrak{c}_{mult} = \mathfrak{c} \boxtimes \pi(\mathbf{p}')$ encrypts $\pi(\mathbf{p}^\times)$ such that $\mathbf{p}_i^\times = \mathbf{p}_i \cdot \mathbf{p}'_i$.

We can also homomorphically rotate, i.e. circularly shift, a plaintext vector encrypted in a ciphertext. At a high level, we have:

- Homomorphic rotation: Given an integer i such that $1 \leq i < l$ and a ciphertext $\mathfrak{c}$ that encrypts $\pi(\mathbf{p})$ where $\mathbf{p}$ is an l-vector, , the ciphertext $\mathfrak{c}_{rot} = \mathfrak{c} \lhd i$ encrypts $\pi(\mathbf{p}^\lhd)$ such that $\mathbf{p}^\lhd = \mathbf{p} \ll i = (\mathbf{p}_{i+1} \dots \mathbf{p}_l \mathbf{p}_1 \dots \mathbf{p}_i)$

4 The Single Server PIR Protocol

4.1 Some Intuitions

We start from a non-private protocol. The server has an n-bit database $\mathbf{x}$, and the client wants to retrieve the ith bit $\mathbf{x}_i$. Firstly, the server picks an integer $t < n$ and arranges its database into an $n' \times t$ matrix X, where $n' = \lceil \frac{n}{t} \rceil$. Now $\mathbf{x}_i$ becomes X_{jk} for some j and k in the matrix. To retrieve the bit, it is sufficient that the client retrieves the jth row. Each row in X is a bit vector and can be viewed as a binary polynomial of degree at most $t - 1$. To retrieve a row, the client creates an n'-bit query string $\mathbf{q} = \mathbf{q}_1 \mathbf{q}_2 \dots \mathbf{q}_{n'}$ such that all bits are 0 except $\mathbf{q}_j$. This query string can be viewed as a vector of constant binary polynomials. The client sends the query string to the server. The server computes the inner product of $\mathbf{q}$ and X (viewed as an n'-vector of binary polynomials) $\mathbf{q}_1 \cdot X_1 + \mathbf{q}_2 \cdot X_2 + ... + \mathbf{q}_{n'} \cdot X_{n'}$. Here $\cdot$ and $+$ are polynomial multiplication and addition operations. The server sends the inner product to the client. Clearly, since only $\mathbf{q}_j$ is 1, the inner product equals X_j. Given X_j which is the jth row in the server's matrix, the client checks the kth bit. This is the bit it wants to retrieve. If we use an FHE scheme, we can make the above protocol private. However, the communication complexity is too high. To deal with this problem, we use a tree-based compression scheme described in the next section to compress the query string.

4.2 Folding and Unfolding

In this section we show the folding/unfolding compression scheme we designed to compress query strings in the protocol[4]. Without loss of generality, in the following we always assume the parameter $n' = 2^\zeta$ for some ζ that is a positive integer.

Given a query string $\mathbf{q}$, which is n'-bit long and with only one bit at index j set to 1. To fold it, we create a $d_1 \times d_2$ matrix M. Then we fill the query string into the matrix,

[4] As pointed out by a reviewer, functionally the algorithms are equivalent to the encoder and decoder circuits as described in chapter 2 of [35]

starting from the top leftmost cell and wrapping at the end of each row. In the matrix, only one bit $M_{\alpha\beta}$ is 1, and all other bits are 0. We then obtain two strings $\mathbf{u}, \mathbf{v}$ such that $\mathbf{u}$ is d_1-bit and $\mathbf{v}$ is d_2-bit. Both $\mathbf{u}$ and $\mathbf{v}$ have only a single bit set to 1. A toy example is shown in Fig. 1. In this example, $\mathbf{q}$ is 16-bit and $d_1 = d_2 = \sqrt{n'} = 4$. We obtain $\mathbf{u}$ and $\mathbf{v}$ such that in $\mathbf{u}$ the αth ($\alpha = 3$ in the example) bit is 1 and in $\mathbf{v}$ the βth ($\beta = 2$ in the example) bit is 1. To unfold, we create a two-dimensional matrix M', then fill it using $\mathbf{u}$ and $\mathbf{v}$ such that for each $1 \leq a \leq d_1, 1 \leq b \leq d_2$, $M'_{ab} = \mathbf{u}_a \cdot \mathbf{v}_b$. Then we concatenate the rows and get back the original query string q.

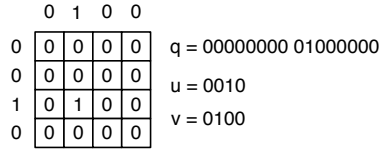


Fig. 1: Fold and unfold a query string

Note that since $\mathbf{u}$ and $\mathbf{v}$ are also two strings with only a single bit set to 1, what we have done to $\mathbf{q}$ can be done to $\mathbf{u}$ and $\mathbf{v}$ in the same way. For each of them, we can fold it into two shorter strings. In the example, both strings can be represented as a 2×2 matrix and folded into two 2-bit strings. The four 2-bit strings can be unfolded and allows us to get back to $\mathbf{u}$ and $\mathbf{v}$. In general, for any bit string of size n' with only one bit set to 1, we can always fold it into $\log n'$ strings that each one is only 2-bit long.

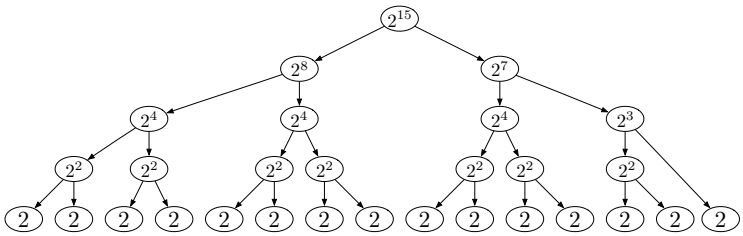


Fig. 2: A folding tree for a 2^{15}-bit query string. The number in each node is the length of the string to be folded/unfolded at the node.

Folding a string can be done in different ways if we choose different dimensions for the matrix in each step. In the example in Fig. 1, we can also use a 2×8 matrix to fold $\mathbf{q}$. To be deterministic, we define a tree structure and associated algorithms. The tree is an auxiliary structure that directs how to fold and unfold a string recursively. To build such a tree, the only information we need is the length of the query string. More formally, we define a *folding tree* to be a binary tree, such that each non-leaf node has exactly two children (referred to as the left child and the right child). Each node in the folding tree stores a number that is the length of the string to be folded or unfolded at this node. The algorithm to build a folding tree is given in Algorithm 1 and a folding tree built from the algorithm is shown in Fig. 2. It is easy to prove that a folding tree built from the algorithm has height $\log \log n'$ and has $\log n'$ leaf nodes.

What Algorithm 1 does is to build a tree structure. At each node, it checks the input number n' which is always a power of 2, if $n' > 2$ then n' can always be factored into

$n' = 2^{\zeta_1} \cdot 2^{\zeta_2}$. To make it deterministic, we choose ζ_1 such that ζ_1 is an integer and $2^{\zeta_1} < n' \leq 2^{2\zeta_1}$. After we find ζ_1, we set $\zeta_2 = \log n' - \zeta_1$. That means we can write the n'-bit string into an $2^{\zeta_1} \times 2^{\zeta_2}$ matrix, thus the string can be folded into two strings of 2^{ζ_1}-bit and 2^{ζ_2}-bit long. Then we invoke the next level recursions with 2^{ζ_1} and 2^{ζ_2}. The recursions will end when the input number is 2.

Algorithm 1: buldTree(node, n')

input : A tree node node and an integer $n' = 2^{\zeta}$
output: A folding tree for a query string of length n'

1 **if** *node* == *NULL* **then** node = new node; // a new tree
2 node.$strLen = n'$;
3 **if** n' == 2 **then** return; // end condition of the recursion
4 $\zeta_1 = \lceil \log n'/2 \rceil, \zeta_2 = \log n' - \zeta_1$; // determine the dimensions
5 left = new node, right = new node;
6 node.$left$ = left, node.$right$ = right;
7 buldTree(left, 2^{ζ_1}); // recursion
8 buldTree(right, 2^{ζ_2});
9 return node;

Algorithm 2: fold(T, $\mathbf{q}$)

input : A folding tree T and a query string $\mathbf{q}$ of length $n' = 2^{\zeta}$
output: A folded representation of $\mathbf{q}$, which is a string of $2 \log n'$ bits

1 **if** *T is a leaf node* **then** return $\mathbf{q}$;
2 d_1=T.$left.strLen$, d_2= T.$right.strLen$;
3 j = the index of the 1 bit in $\mathbf{q}$, $\alpha = \lfloor (j-1)/d_2 \rfloor + 1$, $\beta = ((j-1) \bmod d_2) + 1$;
4 $\mathbf{l}$ = new bit string of length d_1, all bits are initialized to 0;
5 $\mathbf{r}$ = new bit string of length d_2, all bits are initialized to 0;
6 set the αth bit in $\mathbf{l}$ to 1;
7 set the βth bit in $\mathbf{r}$ to 1;
8 $\mathbf{a}$ = fold(T.$left, \mathbf{l}$);
9 $\mathbf{b}$ = fold(T.$right, \mathbf{r}$);
10 return $\mathbf{a}||\mathbf{b}$;

After we have built the folding tree, we can use it to fold and unfold the query string. Each folding tree is built with an input n' and can only fold/unfold query strings of length n'. The algorithm to fold a query string is shown in Algorithm 2. In the algorithm, we do not really need to fill the string into a matrix. As we can see in line 2, the dimensions of the matrix are stored in the folding tree: the number stored in the left child node is the number of rows and the number stored in the right child node is the number of columns. With this information, then in line 3, given the index j of the 1 bit in the input string, we can convert the index into a row index α and a column index β in the matrix. Then we can generate two strings, one with the αth bit set to 1 and one with the βth bit set to 1. The strings will be passed to the next recursions. At a leaf node, the recursion ends. At the end of the algorithm, the 2-bit strings at all leaf nodes are concatenated and returned. Since we have $\log n'$ leaf nodes, the query string is folded into a string of $2 \log n'$ bits. Unfolding is essentially the inverse process. The folded query string is broken into $\log n'$ strings each of 2 bits long and assigned to the leaf nodes.

Then starting from the leaf nodes, the strings held by sibling nodes are unfolded into a longer string by multiplying the bits. Eventually at the root the original query string is fully unfolded.

Algorithm 3: unfold(T, s)

input : A folding tree T and a folded query string $\mathbf{s}$
output: A query string $\mathbf{q}$ of n' bit, unfolded from $\mathbf{s}$

1 **if** *T is a leaf node* **then** return $\mathbf{s}$;
2 d_1=T.$left.strLen$, d_2= T.$right.strLen$, $\zeta_1 = 2\log d_1$, $\zeta_2 = 2\log d_2$;
3 Split $\mathbf{s}$ into two strings such that $\mathbf{s} = \mathbf{sl}||\mathbf{sr}$, $\mathbf{sl}$ is ζ_1-bit and $\mathbf{sr}$ is ζ_2-bit;
4 $\mathbf{l}$ = unfold(T.$left$, $\mathbf{sl}$);
5 $\mathbf{r}$ = unfold(T.$right$, $\mathbf{sr}$);
6 $\mathbf{q}$ = new bit string of length $d_1 \times d_2$;
7 **for** $a = 1$ *to* d_1 **do**
8 **for** $b = 1$ *to* d_2 **do**
9 $i = (a-1) \cdot d_2 + b$;
10 $\mathbf{q}_i = \mathbf{l}_a \cdot \mathbf{r}_b$;
11 **end**
12 **end**
13 return $\mathbf{q}$;

The unfolding algorithm can work perfectly with an FHE scheme. Now all strings in the algorithm are replaced by vectors of ciphertexts that encrypt the strings bit-by-bit. The input ciphertext vector is of size $2\log n'$ and the output ciphertext vector is of size n'. The process is almost identical to the plaintext case. The only difference is that in line 10 of Algorithm 3, the multiplication operation will be the homomorphic multiplication operation.

4.3 The PIR protocol

Now we are ready to describe our PIR protocol. The protocol is the parallelised version of the protocol described in Section 4.1. Recall that in BGV, we can pack an l-vector of elements in $\mathbb{F}_{2^d}$ in a single ciphertext and process the elements in an SIMD fashion. We will use this feature in our protocol to run l instances of the protocol in section 4.1 simultaneously. On the server side, the server represents its database as an $n' \times l$ matrix, each element in the matrix is a d-bit binary vector that can be viewed as an element in $\mathbb{F}_{2^d}$. Later, homomorphic operations will be applied to all elements in the l-vector simultaneously. That is, we can process $l \cdot d = \phi(m)$ bits each time. In this way, we can amortise server side computation. On the client side, the client needs to send the query string $\mathbf{q}$ to the server. It uses the folding algorithm to fold $\mathbf{q}$ into $\mathbf{s}$. The folded query string $\mathbf{s}$ is short, only $2\log n'$ bits. We can always find BGV parameters such that $2\log n' \leq l$. Therefore the client can pack $\mathbf{s}$ into one single ciphertext and sends it to the server. The protocol is as follows and we will explain why this is correct after the description:

1. **Init**: Given a security parameter λ and the size of the database n, the client chooses the maximum depth of circuit L, and invokes $G(\lambda, L)$ to generate a BGV key pair (pk, sk) and public parameters $param$. The private parameter set $\mathcal{S} = \{sk\}$, the public parameter set $\mathcal{S} = \{pk, param\}$. Given $\phi(m)$ and the number of plaintext

slots l, the server arranges its database into an $n' \times l$ matrix X, where $n' = \lceil \frac{n}{\phi(m)} \rceil$. Each element in the matrix is a bit vector of length d, where $d = \phi(m)/l$.

2. **QGen**: The client does the following to generate a query:
   (a) The client converts i into (α, β, γ), i.e. the bit $\mathbf{x}_i$ is the γth bit of the element at $X_{\alpha\beta}$. Then the client creates a query string $\mathbf{q}$ of length n', which contains all 0 bits except the αth bit set to 1. The client creates a folding tree with n' as the input. Then the client folds $\mathbf{q}$ into $\mathbf{s}$ using the folding algorithm.
   (b) The client pads $\mathbf{s}$ with 0 to l bits. Then the client circularly right shifts $\mathbf{s}$ to get a new string $\mathbf{s}' = \mathbf{s} \gg (\beta - 1)$ so that in $\mathbf{s}'$ the βth bit is the first bit in $\mathbf{s}$. Here $\mathbf{s}'$ can be viewed as an l-vector of constant binary polynomials. Then the client uses the packing feature: maps $\mathbf{s}'$ to an element in $\mathbb{A}_2$ and encrypts it. The result $\mathfrak{s} = E_{pk}(\pi(\mathbf{s}'))$ is the query $\mathcal{Q}$ and is sent to the server.
3. **RGen**: Given $\mathcal{Q} = \mathfrak{s}$, the server generates a response as follows:
   (a) The server generates a vector of ciphertext $\mathbf{c}$ that contains $2 \log n'$ ciphertexts after receiving $\mathcal{Q}$, such that $\mathbf{c}_1 = \mathfrak{s}$ and for each $2 \leq k \leq 2 \log n'$, $\mathbf{c}_k = \mathfrak{s} \lhd (k-1)$. The server generates a folding tree with n' as input. Then the server runs the unfolding algorithm homomorphically with the folding tree and $\mathbf{c}$ as input. The result $\mathbf{c}'$ is a vector of n' ciphertexts.
   (b) The server then computes a single ciphertext $\mathfrak{r} = (\mathbf{c}'_1 \boxtimes \pi(X_1)) \boxplus (\mathbf{c}'_2 \boxtimes \pi(X_2)) \ldots \boxplus (\mathbf{c}'_{n'} \boxtimes \pi(X_{n'}))$. Then the server returns the resoponse $\mathcal{R} = \mathfrak{r}$ to the client.
4. **RExt**: Given $\mathcal{R} = \mathfrak{r}$, the client decrypts $\mathfrak{r}$ and obtains an l-vector. The γth bit in the βth element in the vector is the bit $\mathbf{x}_i$ it wants to retrieve.

As we said earlier, in this protocol we work with packed ciphertexts. Homomorphic operations involving packed ciphertexts are component-wise. So one major change in this protocol compared to the protocol described in Section 4.1 is that instead of just one single binary polynomial in a row of the matrix X, now in each row we have a vector of l binary polynomials. The client's goal is to retrieve the βth polynomial in the αth row that contains the bit. In step 2a, the client generates a query string $\mathbf{q}$ that can be used to retrieve the αth row in the server's matrix and folds it into $\mathbf{s}$. In step 2b, the client circularly right shifts the string $\mathbf{s}$ by $\beta - 1$ positions. The reason is that $\mathbf{s}$ contains only information about the row index of the element the client wants to retrieve, by shifting it the result $\mathbf{s}'$ contains information about both the row index and the column index. This becomes clearer in step 3a. The server generates a vector of ciphertext $\mathbf{c}$ by rotating the ciphertext from the client. The first ciphertext $\mathbf{c}_1$ encrypts $\mathbf{s}'$, and the βth bit in $\mathbf{s}'$ is $\mathbf{s}_1$, the second ciphertext encrypts $\mathbf{s}' \ll 1$, and the βth bit in $\mathbf{s}' \ll 1$ is $\mathbf{s}_2$, and so on and so forth. In fact we can view $\mathbf{c}$ as encrypting a bit matrix S of size $2 \log n' \times l$. The βth column S^β is the folded query string $\mathbf{s}$ generated in step 2a by the client. The server can unfold $\mathbf{s}$ back to $\mathbf{q}$ by running the unfolding algorithm. This is because the batched homomorphic operations are component-wise. Therefore by running the algorithm with packed ciphertexts, the server actually runs l instances of unfolding simultaneously, each with the same folding tree and a distinct column from S as input. The input string to the βth instance is $\mathbf{s}$ and thus $\mathbf{q}$ can be unfolded. For the other unfolding instances, it does not matter what the unfolding results are because the client is only interested in the αth element in the βth column of the server's database matrix. So as long as the βth instance is correct then it is fine. The result $\mathbf{c}'$ obtained

from running the unfolding algorithm can also be viewed as encrypting an $n' \times l$ matrix such that the βth column is $\mathbf{q}$ and the other columns contain useless bits. Then in step 3b, the server again uses batched homomorphic operations to run l instances of inner product evaluation. The input to each instance is a column in the unfolding result matrix and the corresponding column in X. The βth instance computes the inner product of $\mathbf{q}$ and X^{β}. The result is $X_{\alpha\beta}$ which is the element the client wants to retrieve. The element is encrypted in the βth slot in $\mathfrak{r}$ and by decrypting the ciphertext, the client can obtain the element. Then by examining the γth bit in the element, the client knows the bit it wants to retrieve. Fig. 3 shows a toy example. In the example $n = 32, l = 4, d = 2$ and $n' = 4$, so the server's database is organised as a 4×4 matrix. The bit the client wants to retrieve is the first bit in$X_{3,2}$.

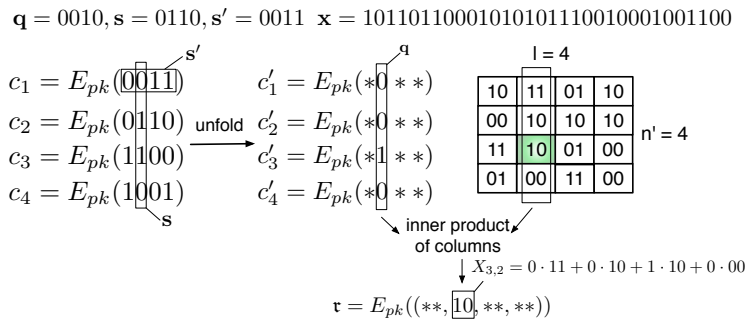


Fig. 3: An Example of the PIR Protocol (* means a bit we do not care).

Extensions. With some modifications, we can extend the PIR protocol into a PBR protocol [3] or a symmetric PIR protocol [4]. In a PBR protocol, the client retrieves a block rather than a single bit. In a symmetric PIR protocol, the client retrieves just one bit and learns nothing about the other bits in the server's database. Due to limited space, we do not discuss them here. They will be presented in the full version of the paper.

4.4 Efficiency Analysis

Communication. In the protocol, the client sends a request that is a single ciphertext and the server returns a single ciphertext. The size of the ciphertexts depends on $\mathbb{A}_q$. For each element in $\mathbb{A}_q$, it is a polynomial of degree at most $\phi(m) - 1$. Therefore the size of a ring element is at most $\phi(m) \cdot \log q$ bits. The parameters $\phi(m)$ and q are interdependent. For simplicity, in our protocol we choose a large enough and fixed $\phi(m)$ and therefore q becomes a variable independent of $\phi(m)$. Then we have $\log q = a + b \cdot L$ where a, b are small constants and L is the depth of circuit to be evaluated. Since $L = \log\log n'$, the bit length of q is $O(\log\log n')$. Then overall, the communication cost is $O(\log\log n') = O(\log\log \frac{n}{\phi(m)}) = O(\log\log n)$.

Server side computation. The server side computation consists of three parts: homomorphic rotations, unfolding and homomorphic inner product computation. The complexity of the rotation operation is $O(\phi(m) log \phi(m))$ multiplications modulo q. We need in total $2\log n' - 1$ rotations. To unfold the query string, the server needs $\sum_{i=0}^{\log\log n'} 2^i \sqrt[2^i]{n'} < n' + 3\sqrt{n'}$ homomorphic multiplications. The computational cost of the inner product part is dominated by the n' homomorphic multiplications. To un-

derstand the cost of the protocol, we need to understand the cost of homomorphic multiplication operations. We have two different homomorphic multiplication operations: raw multiplications and full multiplications. A raw multiplication simply computes the tensor product of the parts in the ciphertexts, so the cost is 4 (2 if one operand is a plaintext) multiplications over $\mathbb{A}_q$. A full multiplication is a raw multiplication followed by a modulus switching and a key switching on the product. The maintenance operations are necessary to ensure correctness and maintain the size of the ciphertext. The cost of a multiplication over $\mathbb{A}_q$ is $\phi(m)$ multiplications modulo q. The complexity of modulus switching and key switching is $O(\phi(m)log\phi(m))$ multiplications modulo q. Therefore a full multiplication is more costly than a raw multiplication.

Our observation is that in our protocol most homomorphic multiplications can be raw multiplications. Namely, we mean the n' multiplications required by the last step of the unfolding algorithm and the n' multiplications required by the inner product computation. The total cost of this part is $4 \cdot n' \cdot \phi(m) + 2 \cdot n' \cdot \phi(m) = 6n$ multiplication modulo q (q is less than 64-bit because of previous modulus switching operations). We only need less than $3\sqrt{n'}$ full multiplications. The total cost of this part is $O(\sqrt{n'}\phi(m)log\phi(m))$ modular multiplication operations. Each modular multiplication here can be implemented by 1 or a few 64-bit modular multiplications.

As we can see, the overall computational complexity is $O(\log n' + \sqrt{n'} + n') = O(n)$. For sufficiently large n, the computational cost of the homomorphic rotation part is insignificant compared to the the other two parts. Moreover, when n is sufficiently large, $\sqrt{n'}$ will be much smaller than n'. That means the number of total operations required by full multiplication part is smaller than the raw multiplication part. Then in this case, the server side computation is bounded by $12n$ 64-bit modular multiplication operations.

Client side Computation. The client side computation in our protocol is very light. The client needs to do 1 encryption and 1 decryption. The cost of encryption or decryption is $O(\phi(m))$ multiplications modulo q. In practice, each encryption/decryption needs only a couple of milliseconds.

4.5 Security Analysis

In this section, we analyse the security of our PIR protocol. We have the following theorem:

Theorem 1. *If the BGV FHE is semantically secure, then our PIR protocol is a secure single server PIR protocol.*

Proof. We show that if a PPT adversary $\mathcal{A}$ can distinguish two queries with a non-negligible advantage, then an adversary $\mathcal{A}'$ can use $\mathcal{A}$ as a subroutine to win the BGV CPA game with a non-negligible advantage. The BGV CPA game is a standard public key encryption CPA game, in which $\mathcal{A}'$ needs to distinguish two ciphertexts encrypted under a BGV public key. The game is in the appendix. $\mathcal{A}'$ does the following:

- $\mathcal{A}'$ chooses n and generates a database $\mathbf{x}$, chooses λ and L, then receives the BGV public key and parameters $(pk, param)$. It then invokes $\mathcal{A}$ with $\mathbf{x}, pk, param$.
- For any index i, $\mathcal{A}$ can generate the query by itself using the public key. At some point of time, $\mathcal{A}$ outputs two indexes i_0, i_1 and sends them to $\mathcal{A}'$.
- $\mathcal{A}'$ generates m_0 using i_0. $\mathcal{A}'$ first generates a query string $\mathbf{q}$ from i_0, folds $\mathbf{q}$ into $\mathbf{s}$ and then pads and shifts to get $\mathbf{s}'$ from $\mathbf{s}$. The message $m_0 = \pi(\mathbf{s}')$.

– $\mathcal{A}'$ generates m_1 in the same way as above but using i_1 as input.
– $\mathcal{A}'$ sends m_0 and m_1 to the challenger in the BGV CPA game, then receives $\mathbf{c}_b$.
– $\mathcal{A}'$ sends $\mathbf{c}_b$ to $\mathcal{A}$, and outputs whatever $\mathcal{A}$ outputs.

It is clear that the probability of $\mathcal{A}'$ winning the BGV game is the same as the probability of $\mathcal{A}$ outputting $b' = b$. Since the BGV encryption is semantically secure, the probability of $\mathcal{A}'$ winning the game is $\frac{1}{2} + \eta$, where η is negligible. Then the advantage of $\mathcal{A}$ is also negligible.

5 Implementation and Performance

5.1 Implementation

We have implemented a prototype in C++. The implementation is based on HElib [36], an open source implementation of the BGV FHE scheme. Currently in the prototype the client and the server run in the same process. This does not affect the evaluation result. To measure network communication, we output the ciphertexts to files and measure the file size. We have done a few optimisations:

Delayed Unfolding. We delay the last unfolding step. Instead of fully unfolding the query string, we combine this step with the inner product computation step. The main reason is that if we fully unfold the query string, we need to store n' ciphertexts. Because n' can be large, we need enormous memory to store the ciphertexts. If we stop at the two children of the root, then we only need to store two vectors of approximate $\sqrt{n'}$ ciphertexts. When we compute the inner product, we can unfold the bit we need on the fly using the two vectors.

Tree Pruning. We can also prune the folding tree to lower the communication cost. The idea is that if we do not fully fold the query string, we will end up with a longer folded string, but we might still be able to pack it into one ciphertext. For space reason, we do not formally present the tree pruning algorithm but use an example to explain the idea. Consider without pruning, the client and the server use the folding tree in Fig. 2, so the client fully folds its query string into a 30-bit string and the server can unfold the query string. If we prune the tree to have only 3 nodes: the root node and the two children of it, then with this tree, the client can fold the the query string into a $2^8+2^7 = 384$-bit string. As long as the number of plaintext slot $l \geq 384$, the client can pack the string into one ciphertext. With this packed ciphertext, the server can obtain $\mathbf{c}$ by 383 rotations, and then breaks $\mathbf{c}$ into two vectors, the first one with 2^8 ciphertexts and second one with 2^7 ciphertexts. The encrypted query string can be unfolded from these two vectors, and the server can then compute the inner product. The tree pruning algorithm takes a folding tree and l as input, scans from the root, once it finds a level such that the sum of $strLen$ of all nodes at this level is smaller than l, it prunes all nodes below this level. Tree pruning requires only a minor modification to the unfolding algorithm. Tree pruning can reduce communication cost because lower tree height means lower circuit depth, and then smaller q.

Multithreading. Conceptually, the server side computation in our protocol can be easily parallelised. Each step in unfolding requires $d_1 \cdot d_2$ independent homomorphic multiplications, and the inner product computation step requires n' independent homomorphic multiplications. We can parallelise those multiplications without much effort. However, multithreading is not easy with HElib because it depends on the NTL library that is not thread safe. After analysing the source code of HElib, we managed

to make the raw multiplication and addition operations independent of the NTL library and make the prototype partially multithreaded. This enables our implementation to take advantage of multicore hardware.

5.2 Performance

In this section, we report the performance based on our prototype implementation. All experiments were conducted on a MacBook Pro laptop, with an Intel 2720QM quad-core 2.2 GHz CPU and 16 GB RAM. The choice of the BGV parameters is based on the formula given in [37]: $\phi(m) \geq \frac{\log(q/\sigma)(\lambda+110)}{7.2}$, where σ is the noise variance of the discrete Gaussian distribution and λ is the security parameter. The variance $\sigma = 3.2$ in HElib. We chose $m = 8191$ thus $\phi(m) = 8190$ and the number of plaintext slot $l = 630$. The modulus q is an odd integer of $40 + 20L$ bits, where L is the height of the folding tree. When $\lambda = 128$, the largest L supported by the chosen m is 7. In other words, the parameters ensure 128-bit security as long as the database is less than $2^{2^7} = 2^{128}$-bit, which is more than enough in any practical settings.

We first show the communication cost (Table 1). One thing to be noticed is that HElib outputs ciphertexts as textual strings of decimal numbers, so the measured size is bigger than the raw bit size. We used database of size 2^{25} bits (4 MB), 2^{30} bits (128 MB), and 2^{35} bits (4 GB) in our experiments. As we can see, the communication cost is low, only a few hundred KB. The response is only one ciphertext and the size is fixed across all cases. Most time the request is larger than the response despite that it is also just a single ciphertext. The reason is that q is not fixed in the BGV FHE scheme. We use modulus switching to switch to smaller q during the homomorphic operations. So the ciphertext in the response uses a smaller q and in consequence the size of the ciphertext is smaller. Another fact about the response is that has 3 ring elements[5] because we omitted the key switching operations in the last unfolding step. This explains why in the first experiment with pruning (database size = 2^{25}), the request is smaller than the response. We can also see that tree pruning does help reduce the request size.

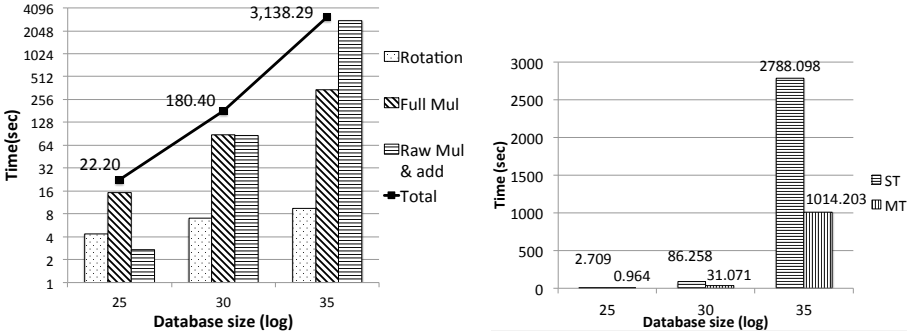
	Without Pruning			With Pruning		
	L	Request	Response	L	Request	Response
2^{25}	5	336 KB	192 KB	2	180 KB	192 KB
2^{30}	6	389 KB	192 KB	3	231 KB	192 KB
2^{35}	6	389 KB	192 KB	3	231 KB	192 KB

Table 1: Communication cost with different database size

We then show the server side computation time (Fig. 4). In Fig. 4a, we show the computation time for each step as well as the total time. The columns show the time for rotation, full multiplications (unfolding except the last step) and raw multiplications and additions (the last unfolding step plus inner product computation). The line shows the total computation time. As we can see, the rotation step is always fast. When the size of the database is small, the computation is dominated by the full multiplications. But when the size of the database increases, the raw multiplication and addition step starts to become dominant. From the total time, we can estimate the minimal bandwidth needed

[5] The 3-part ciphertext can still be decrypted correctly, so this does not affect the correctness of our protocols.

to make the trivial solution faster. When the database is 4 MB, 128 MB and 4 GB, the minimal bandwidth is 1.25 Mbps, 5.65 Mbps and 10.44 Mbps. With a more powerful CPU (our experiments were done on a laptop), the minimal bandwidth would be higher. We can make our protocol more practical by utilising hardware parallelism. In Fig. 4b, we show the performance of our multithreaded implementation versus single threaded one. The time compared in the diagram is the raw multiplication and addition step, which is the only step we can currently implement in parallel. The experiments were done with a quad-core CPU, and the performance improvement was about 2.7 - 3 times. If with a fully thread safe BGV implementation and 2 or 3 more CPUs, the performance of our protocol can compete with the trivial solution in 100 Mbps LAN.



(a) Computation Time (b) Hardware Parallelisation

Fig. 4: Server Side Computation Time of Our PIR Protocol

5.3 Performance Comparison

Communication. We compare the communication cost with the current most efficient protocols: Lipmaa's protocol [10] and Gentry's protocol [9]. The result is plotted in Fig. 5. Note we use size of ciphertext in the raw bit representation to draw the line for each protocols, so the numbers for our protocol are different from the numbers in Table 1. Lipmaa's protocol assumes that the n-bit database has n' entries each is t bits. The smallest modulus size is $2k$-bit. When $t = k$, the total cost is $\alpha \cdot ((s + \alpha + 1)\zeta/2)(n'^{1/\alpha} - 1) \cdot k$ bit. In the figure, we set $k = 3072$ for 128-bit security and let $\alpha = \log n', s = 1, \zeta = 1$. Then as we can see, our protocol (without pruning) incurs more communication when the database is small, but would be better when the database is sufficiently large. This is due to the large ciphertext size in the BGV scheme. Gentry's protocol is very communication efficient. The communication cost is $3 \log^{3-o(1)} n$-bit integers. With any practical database size, it would be always more efficient than our protocol in terms of communication. However, the difference is less than 200 KB, which is not significant.

Computation. Here we do not compare with Melchor's protocol [18] because it is not secure. We do not compare with other FHE based protocols [11, 12] because they are obviously less efficient. Among all other existing protocols:

- Kushilevitz's protocol [4] requires n modular multiplications.
- Kushilevitz's TDP-based protocol [7] uses interactive hashing to protect the client's privacy against the server. It requires n TDP evaluations on the server side. Each TDP evaluation requires at least one modular multiplication.

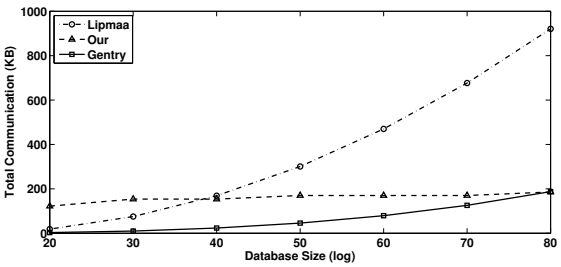


Fig. 5: Communication Cost Comparison

- Gentry's protocol [9] requires only one modular exponentiation but the exponent is $2n$-bit. The computational cost is approximately n modular multiplications.
- Cachin's protocol [6] requires n modular exponentiations. The computational cost is approximately $l_e n/2$ modular multiplications, where l_e is the bit length of the exponent.
- Lipmaa's protocol [10] requires for each $2 \leq j \leq \log n'$, $2^{\log n' - j}$ exponentiations. The computational cost is at least $c{\cdot}n$ modular multiplications for some c depending on $\log n'$.
- Lipmaa's BDD based protocol [17] requires $O(n/(\log n))$ modular exponentiations. The computational cost is approximately $l_e n/(2 \log n)$ modular multiplications. Because the exponent size l_e is larger than $2 \log n$, the total cost is larger than n modular multiplications.
- Chang's protocol [8] requires n modular multiplications and $2 \log n$ modular exponentiations.

The most efficient protocol of all above ones is Kushilevitz's protocol [4] that requires n modular multiplications. Although the number of operations in Gentry's protocol is similar to Kushilevitz's protocol, in practice it would be less efficient due to the large exponent which is twice as big as the database. Another factor that makes Kushilevitz's protocol the most efficient one is the modulus size. Some protocols, e.g. Cachin's, Lipmaa's (and the BDD-based), and Chang's, require larger moduli. So the modular multiplication operation is slower in those protocols than in Kushilevitz's protocol.

We then compare our protocol with Kushilevitz's protocol. For 128-bit security, the modulus size needs to be at least 3072-bit. We measured time for a 3072-bit modular multiplication using the GMP library [38]. This is done by averaging the time for 1 million operations. The time for a single operation is 8.269×10^{-6} second. Thus, when the database size is 2^{25}, 2^{30} and 2^{35} bits, Kushilevitz's protocol would need 277.46, 8878.72 and 284119.04 seconds respectively. That is 12.5, 49.2 and 90.5 times slower than our protocol in single threaded mode.

6 Conclusion

In this paper, we presented a single server PIR protocol based on the BGV FHE scheme. The protocol is efficient both in terms of communication and computation. We have analysed its efficiency and security. We validated its practicality by a prototype implementation. The test results show that the total communication cost is as low as a few hundreds KB and the server side computation is much faster than existing single server PIR protocols.

In future work, we will test and improve performance over large data. We will extend the protocol to multi-query PIR [39]. Namely to further amortise the server-side computation complexity of PIR over multiple queries performed by a single client.

References

1. Khoshgozaran, A., Shahabi, C.: Private information retrieval techniques for enabling location privacy in location-based services. In: Privacy in Location-Based Applications. (2009) 59–83
2. Henry, R., Olumofin, F.G., Goldberg, I.: Practical PIR for electronic commerce. In: ACM Conference on Computer and Communications Security. (2011) 677–690
3. Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: FOCS. (1995) 41–50
4. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: Single database, computationally-private information retrieval. In: FOCS. (1997) 364–373
5. Stern, J.P.: A new efficient all-or-nothing disclosure of secrets protocol. In: ASIACRYPT. (1998) 357–371
6. Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: EUROCRYPT. (1999) 402–414
7. Kushilevitz, E., Ostrovsky, R.: One-way trapdoor permutations are sufficient for non-trivial single-server private information retrieval. In: EUROCRYPT. (2000) 104–121
8. Chang, Y.C.: Single database private information retrieval with logarithmic communication. In: ACISP. (2004) 50–61
9. Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: ICALP. (2005) 803–815
10. Lipmaa, H.: An oblivious transfer protocol with log-squared communication. In: ISC. (2005) 314–328
11. Brakerski, Z., Vaikuntanathan, V.: Efficient fully homomorphic encryption from (standard) LWE. In: FOCS. (2011) 97–106
12. Yi, X., Kaosar, M.G., Paulet, R., Bertino, E.: Single-database private information retrieval from fully homomorphic encryption. IEEE Trans. Knowl. Data Eng. **25**(5) (2013) 1125–1134
13. Sion, R., Carbunar, B.: On the practicality of private information retrieval. In: NDSS. (2007)
14. Williams, P., Sion, R.: Usable PIR. In: NDSS. (2008)
15. Ding, X., Yang, Y., Deng, R.H., Wang, S.: A new hardware-assisted PIR with o(n) shuffle cost. Int. J. Inf. Sec. **9**(4) (2010) 237–252
16. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Cryptography from anonymity. In: FOCS. (2006) 239–248
17. Lipmaa, H.: First CPIR protocol with data-dependent computation. In: ICISC. (2009) 193–210
18. Melchor, C.A., Gaborit, P.: A fast private information retrieval protocol. In: ISIT. (2008) 1848–1852
19. Ostrovsky, R., III, W.E.S.: A survey of single-database private information retrieval: Techniques and applications. In: Public Key Cryptography. (2007) 393–411
20. Brakerski, Z., Gentry, C., Vaikuntanathan, V.: (leveled) fully homomorphic encryption without bootstrapping. In: ITCS. (2012) 309–325
21. Lyubashevsky, V., Peikert, C., Regev, O.: On ideal lattices and learning with errors over rings. In: EUROCRYPT. (2010) 1–23
22. Chor, B., Gilboa, N.: Computationally private information retrieval (extended abstract). In: STOC. (1997) 304–313
23. Ishai, Y., Kushilevitz, E.: Improved upper bounds on information-theoretic private information retrieval (extended abstract). In: STOC. (1999) 79–88

24. Beimel, A., Ishai, Y., Kushilevitz, E., Raymond, J.F.: Breaking the O(n1/(2k-1)) barrier for information-theoretic private information retrieval. In: FOCS. (2002) 261–270
25. Goldberg, I.: Improving the robustness of private information retrieval. In: IEEE Symposium on Security and Privacy. (2007) 131–148
26. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: EUROCRYPT. (1999) 223–238
27. Damgård, I., Jurik, M.: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system. In: Public Key Cryptography. (2001) 119–136
28. Bi, J., Liu, M., Wang, X.: Cryptanalysis of a homomorphic encryption scheme from ISIT 2008. In: ISIT. (2012) 2152–2156
29. Gentry, C.: A fully homomorphic encryption scheme. PhD thesis, Stanford University (2009)
30. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: STOC. (2009) 169–178
31. van Dijk, M., Gentry, C., Halevi, S., Vaikuntanathan, V.: Fully homomorphic encryption over the integers. In: EUROCRYPT. (2010) 24–43
32. Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: CRYPTO (1). (2013) 75–92
33. Smart, N.P., Vercauteren, F.: Fully homomorphic SIMD operations. IACR Cryptology ePrint Archive **2011** (2011) 133
34. Gentry, C., Halevi, S., Smart, N.P.: Fully homomorphic encryption with polylog overhead. In: EUROCRYPT. (2012) 465–482
35. Savage, J.E.: Models of Computation: Exploring the Power of Computing. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
36. Halevi, S., Shoup, V.: Algorithms in HElib. IACR Cryptology ePrint Archive **2014** (2014)
37. Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the aes circuit. In: CRYPTO. (2012) 850–867
38. Granlund, T., the GMP development team: GNU MP: The GNU Multiple Precision Arithmetic Library. 5.1.3 edn. (2013) http://gmplib.org/.
39. Ishai, Y., Kushilevitz, E., Ostrovsky, R., Sahai, A.: Batch codes and their applications. In: STOC. (2004) 262–271

A The CPA game

The security of the BGV scheme is captured by the following CPA game between an adversary and a challenger:

1. The adversary chooses L, then given a security parameter λ, the challenger runs $G(\lambda, L)$ to generate the secret key sk, the public key pk and the public parameters $param$. The challenger retains sk and gives the adversary pk and $param$.
2. The adversary may choose a polynomially bounded number of plaintexts and encrypts them using the public key.
3. Eventually, the adversary submits two chosen plaintexts m_0, m_1 to the challenger.
4. The challenger selects a bit $b \in \{0, 1\}$ uniformly at random, and sends the challenge ciphertext $c = E_{pk}(m_b)$ back to the adversary.
5. The adversary is free to perform a polynomially bounded number of additional computations or encryptions. Finally, it outputs a guess b'.

The adversary wins the game if $b' = b$. Under the RLWE assumption, the BGV scheme is secure which means the probability of any PPT adversary winning this game is $\frac{1}{2} + \eta$ for some negligible η.