Vectorized Batch Private Information Retrieval

Muhammad Haris Mughees   Ling Ren
University of Illinois at Urbana-Champaign
{*mughees2, renling*}*@illinois.edu*

Abstract—**This paper studies Batch Private Information Retrieval (BatchPIR), a variant of private information retrieval (PIR) where the client wants to retrieve multiple entries from the server in one batch. BatchPIR matches the use case of many practical applications and holds the potential for substantial efficiency improvements over PIR in terms of amortized cost per query. Existing BatchPIR schemes have achieved decent computation efficiency but have not been able to improve communication efficiency at all. In this paper, we present the first BatchPIR protocol that is efficient in both computation and communication for a variety of database configurations. Specifically, to retrieve a batch of 256 entries from a database with one million entries of 256 bytes each, the communication cost of our scheme is 7.2∼75x better than state-of-the-art solutions.**

I. Introduction

User privacy is a critical challenge in cloud-based applications. To protect user privacy, various cryptographic primitives and protocols have been proposed. Private information retrieval (PIR) is one such primitive that allows a client to download an entry from a public database on a server without revealing the entry of interest to the server [1]. An efficient PIR scheme can enable many privacy-preserving applications such as DNS lookup [2], password check [3], [4], anonymous communication [5].

Unfortunately, even after decades of study [3], [6]–[15], single-server PIR schemes still come with high costs for many applications. Existing schemes achieve decent performance only when database entries are large (e.g., KiloBytes) [3], [12], [15], [16]. But many applications have small entries; for example, in password check, contact discovery, and DNS lookup, each entry is often a hash digest or an IP address (e.g., 128 or 256 bits). When the entry size is small, existing single-server PIR schemes suffer from very high communication overhead.

It has been observed that in many applications, a client wants to retrieve multiple entries from the same database [5], [12], [17], [18]. For example, a user of an anonymous messaging system fetches multiple messages directed to her [5], a user's browser downloads multiple ads relevant to her interests [14], a user checks which of her contacts signed up for a service [19], or a user checks all her passwords at once against a database of breached passwords [4]. These scenarios motivate BatchPIR [12], [17] (also called Amortized PIR [17] or Multi-query PIR [12]) as a promising alternative to PIR where the client wants to download multiple entries from the server at once.

SealPIR [12]	$2,500\times$
Angel *et al.* BatchPIR [12]	$1,872\times$
MulPIR [3]	$982\times$
OnionPIR [16]	$384\times$
Spiral [15]	$140\times$
This work	$19.2\times$

Table I: Communication overhead of recent single-server PIR and BatchPIR schemes as well as our work. Each entry in the database is 256 bytes. For Angel *et al.* BatchPIR and our BatchPIR schemes, we assume that the client wants to retrieve 256 entries from the server.

The BatchPIR approach has been proposed to reduce the amortized computational cost over PIR. For example, in the state-of-the-art BatchPIR scheme [5], for a database with one million entries each of 288 bytes, it takes 20 seconds of server computation to retrieve a batch of 256 entries, which works out to be only 78 milliseconds amortized per query. However, when it comes to communication, the BatchPIR approach has not been able to provide any benefit.

In Table I, we show the communication overhead of recent single-server PIR schemes and the state-of-the-art BatchPIR scheme [12]. We assume the entry size is 256 Bytes. For BatchPIR schemes, we assume the client wants to retrieve a batch of 256 entries. It can be seen that even for the most efficient scheme, the communication overhead is still more than 100x. The key reason behind the high communication overhead is that these schemes are based on the Ring Learning With Error (RLWE) encryption. A RLWE ciphertext is quite large (tens of KiloBytes), no matter how small the underlying database entry is. Existing BatchPIR schemes do not address this issue and still require at least two ciphertexts to be sent (one in each direction) for *each* query in the batch.

In summary, the state-of-the-art BatchPIR scheme nicely amortizes the computation cost over the batch. But the communication cost is not amortized and remains high for databases with small entries. We believe this is currently the main limitation of BatchPIR for practical applications.

Main contribution. In this paper, we present the first BatchPIR scheme that achieves both low communication and low computation for a wide range of database parameters.

Our key observation is that we can save communication by using a single ciphertext to retrieve many database entries. To achieve this goal, we use a vectorized variant of RLWE homomorphic encryption and design a method to merge ciphertexts encrypting independent entries. As shown in Table I, to download a batch of 256 entries from a database with one million entries where each entry is 256 bytes, the amortized communication overhead of our scheme is 19.2x over the insecure baseline, which is 7.2x better than the state-of-the-art PIR and two orders of magnitude better than existing BatchPIR. Our amortized computation cost is 2x higher to the state-of-the-art BatchPIR, at about only 153 milliseconds.

II. Preliminary and Background

A. Somewhat Homomorphic Encryption

Fully homomorphic encryption (FHE) is a special kind of encryption scheme that allows arbitrary computation over ciphertexts. FHE for arbitrary computation is still very expensive. To achieve practical performance, somewhat homomorphic encryption (SHE, also called leveled FHE) is often used, which only supports computation up to a fixed depth.

We focus on SHE schemes based on the Ring Learning with Errors (RLWE) problem. Many RLWE homomorphic encryption schemes, such as Regev [20], BFV [21], BGV [22], and CKKS [23] share the following common structure. Let $R = \mathbb{Z}[x]/(x^n + 1)$ be a polynomial ring where n is the degree of the polynomial (also called ciphertext dimension) and is usually a power of two. A plaintext message m is a polynomial in $R_t = R \mod t$. The secret key s is a polynomial sampled from a distribution of "small" (e.g., with binary coefficients) polynomials in R. A ciphertext consists of two polynomials in $R_q = R \mod q$ and is given as $(c_0, c_1) = (a, a \cdot s + e + m)$ where a is sampled uniformly at random from R_q and e is a noise polynomial with coefficients sampled from a bounded Gaussian distribution. To decrypt, one computes $\mu = c_1 - c_0 \cdot s = e + m$. As long as the noise e is small, rounding μ recovers m.

RLWE-based SHE schemes support the following homomorphic operations.

- $\mathsf{CtCtAdd}(c_1, c_2)$: This operation takes as input two ciphertexts $c_1 \in R_q^2$ and $c_2 \in R_q^2$, and outputs a ciphertext encrypting the sum of two plaintexts.
- $\mathsf{CtPtMul}(c, p)$: This operation takes as input a plaintext $p \in R_t$ and a ciphertext $c \in R_q^2$ encrypting $m \in R_t$, and outputs a ciphertext encrypting $p \cdot m$.
- $\mathsf{CtCtMul}(c_1, c_2)$: This operation takes as input two ciphertexts $c_1 \in R_q^2$ and $c_2 \in R_q^2$ and outputs a ciphertext encrypting the product of two plaintexts.

Each homomorphic operation increases the noise level in the resulting ciphertext, which is why only a limited number of operations can be performed.

Operation	Time cost (milliseconds)	Noise added (bits)
CtCtAdd	0.07	≈0
CtPtMul	0.09	22
CtCtMul	12.1	29
CtRotate	3.6	≈0

Table II: Experimental computation cost and noise growth of each BFV homomorphic operation. The polynomial degree n is 8,192, the ciphertext modulus q has 150 bits, and the plaintext modulus t has 20 bits. Time costs are measured with the SEAL library [27] version 3.7.2 on a single core in AWS t2.2xlarge instances.

Trade-offs in RLWE parameter selection. Parameter selection for a RLWE encryption scheme provides a delicate balance between computation depth, cost, and security. For a fixed security level and a plaintext text modulus t, a larger ciphertext modulus q requires a larger polynomial degree n, allows higher computation depth, but increases the ciphertext size and per-operation computation cost. We will pick parameters that provide a good balance between cost and computation depth and give a widely accepted security level of 128 bits.

B. Vectorized Homomorphic Encryption

If the plaintext modulus t is a prime, a polynomial in R_t can be used to encode a vector in Z_t^n [24]. This transforms the above multiplication and addition operations into component-wise (also called *slots* in the literature [25]) operations between vectors in $\mathbb{Z}_t^n$.

Vector rotation. With proper parameter choices, an automorphism can be used to move plaintext data across different slots in the vector [26]. This can be abstracted as the following ciphertext rotation operation.

- $\mathsf{CtRotate}(c, r)$: This operation takes as input a ciphertext c encrypting a plaintext vector $v = [v_1, v_2, \cdots, v_n]$ and a value $r \in [0, n)$. It outputs a ciphertext encrypting $v' = [v_{n-r+1}, v_{n-r+2}, \cdots, v_n, v_1, v_2, \cdots, v_{n-r}]$, i.e., v rotated by r slots.

We will extensively use this operation in our scheme.

C. Noise Growth and Computation Costs of Homomorphic Operations

Different homomorphic operations have drastically different noise growth and computation costs. This will significantly impact our design decisions. We summarize the noise growth and computation cost of relevant homomorphic operations in Table II. To be concrete, we use the BFV scheme as an example [28].

CtCtxtAdd is fast and adds little noise. CtPtxtMul is also fast but adds a lot more noise than addition. On the other hand, CtRotate does not add much noise but it is quite slow due to an expensive key-switching step [27]. Lastly, CtCtMul is quite expensive in both computation and noise growth. It is slow because it involves multiple expensive steps: base expansion, quantization, and relinearization [27].

D. (Batch) Private Information Retrieval

Private Information Retrieval (PIR). Given an index i and a database DB of N entries, the client wants to privately download i-th entry in the database, i.e., DB_i. A PIR protocol should satisfy the following properties.

- **Correctness:** If the client and the server correctly execute the protocol, then the client recovers the requested entry.
- **Privacy of client:** The server learns *nothing* about which entry the client is requesting.

Batch Private Information Retrieval (BatchPIR). In BatchPIR, instead of a single entry, the client wants to privately download a batch of entries corresponding to indices $\{i_1, i_2, \cdots, i_b\}$. That is to say, the output of BatchPIR is $\{\mathsf{DB}_{i_1}, \mathsf{DB}_{i_2}, \cdots, \mathsf{DB}_{i_b}\}$. A BatchPIR protocol should satisfy the following properties.

- **Correctness:** If the client and server correctly execute the protocol, then the client recovers the requested set of entries.
- **Privacy of client:** The server learns *nothing* about the batch of indices the client is requesting.

E. Previous Batch Private Information Retrieval

Figure 1 gives a high-level overview of the BatchPIR framework of Angel *et al.* [12]. The server database consists of N entries and the client batch has b indices. The example in Figure 1 has $N = 6$ and $b = 3$. There is a one-time setup stage in which the server hashes database entries into buckets. In more detail, the server picks w independent hash functions $h_1, \cdots, h_w$; typically $w = 3$. Then, the server creates B buckets; typically $B = 1.5b$. For i-th entry a_i in the database, the server picks buckets $h_1(i), \cdots, h_w(i)$ and copy a_i into each of these buckets. It is common to add a nonce to the hash functions to make sure the w buckets are distinct. This results in exactly wN entries in total across all buckets.

To generate a query, the client assigns the batch of b indices into the B buckets as well. To do that, the client uses cuckoo hashing with a maximum bucket size set to one. In more detail, for each index i in the batch, the client computes the candidate buckets $h_1(i), \cdots, h_w(i)$ and places i in one empty candidate bucket. If none of the candidate buckets is empty, put i into one of the random candidate buckets, evict the index (call it j) currently in that bucket, and try to re-insert j. If reinserting j causes another index to be evicted, the process continues recursively for a predetermined maximum number of times. After the cuckoo hashing step finishes for all the b indices in the batch, the client assigns a dummy index (usually 0) to each of the remaining empty buckets. Each bucket now holds a single index.

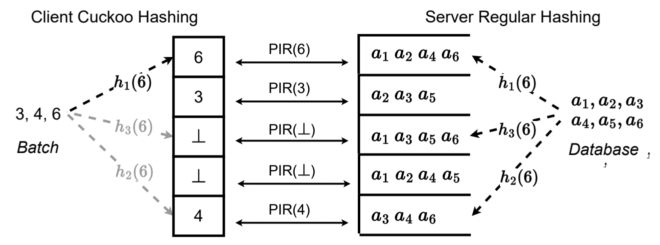

Figure 1: The BatchPIR protocol of Angel *et al.*. The server assigns and copies each element into three buckets and the client assigns each index to one of those three buckets. The client and server run an independent PIR for each bucket.

It is important to note that the server adds each entry to all the candidate buckets and the client assigns each index to one of the candidate buckets. Therefore, if a particular index i is assigned to bucket j on the client side, then bucket j on the server side is guaranteed to contain the i-th entry of the database DB_i. Therefore, the client and the server perform B PIR instances, one for each bucket, to retrieve all the desired entries. Angel *et al.* used SealPIR which is proposed in the same paper [12], but their BatchPIR framework is compatible with any PIR scheme listed in Table I.

Convert database index to bucket index. In the above explanation, one subtle issue is left to address. For each bucket, the client only knows the assigned entry index in the database. To make a bucket PIR query, the client must also know the location/index of entry within that bucket. To address the issue, Angel *et al.* has proposed several solutions. The client can directly acquire a map from each index in the database to all indices within each server bucket. This map can be compressed using techniques like bloom filter [29]. Another option is for the client to construct this map locally. This will require the server to share with the client the source of randomness. The client can then simulate the server hashing procedure on the N indices.

Cuckoo hashing failure. There is a chance that the eviction and re-insertion process exceeds the maximum number of insertions. If that happens, we say the cuckoo hashing step has failed. This failure probability depends on the number of hash functions w, the batch size b, and the number of buckets B. Analyzing the failure probability of cuckoo hashing is an open problem. Most previous works experimentally verify it for the parameter configurations they are interested in [30], [31]. Angel *et al.* experimentally estimated with $w = 3$ and $B = 1.5b$, the failure probability is less than 2^{-40} for a batch size of 200 and above, and less than 2^{-20} for a batch size of 32. For small batch sizes, one can set B to be larger than $1.5b$ to reduce the failure probability. Alternatively, an application can also choose to divide a batch into smaller batches when a cuckoo hashing failure does occur.

Low computational cost. The B PIR instances dominate the computation cost. The computation cost of a PIR is roughly proportional to the number of entries in the database. Since the total number of entries across all buckets is $3N$, the total computation cost of these B PIR instances is proportional to only $3N$ where N is the size of the server's original database. Hence, the amortized computation cost per entry is quite small.

High communication overhead. Unfortunately, Angel *et al.*'s BatchPIR framework does not improve communication at all. The client and the server exchange at least two RLWE ciphertexts (one for query and one for response) for each PIR instance. The ciphertext size could range from 21 KB to 128 KB, even if the plaintext entry is small. Thus, the communication overhead to retrieve a batch of small entries could get very high. Concretely, even if we plug in the most communication efficient PIR scheme [15], to retrieve a batch of 256 entries where the entry size is 256 bytes, the total communication is 13.4 MB, which is around 214x the plaintext entry size.

As discussed in Section II, RLWE ciphertext size depends on the ciphertext modulus q and the polynomial degree n. To reduce the communication overhead, one may be tempted to simply reduce q and n. However, we cannot reduce q because q needs to be sufficiently large to accommodate the noise growth; similarly, to maintain security we cannot reduce the polynomial degree n [32].

III. A Vectorized PIR Protocol

Our BatchPIR protocol follows the template of Angel *et al.* framework described in the previous section where the client and server first distribute their inputs (the requested batch and the database respectively) into B buckets. But instead of running independent PIRs for each bucket, our protocol merges the request and response ciphertexts across buckets.

Towards this goal, we will first present a new PIR protocol (not batched) whose request and response ciphertexts are *vectorized*. Naturally, we will rely on vectorized RLWE SHE introduced in Section II. To elaborate, the response ciphertext will encrypt a vector that contains the desired entry at one of the slots and zeroes in the remaining slots. We will then find ways to merge many vectorized responses into a single ciphertext in our BatchPIR protocol later.

We remark that if used as a standalone PIR protocol, our vectorized PIR has no advantage over the state-of-art. It is designed solely to serve as a building block to our BatchPIR protocol in the next section.

A. A Warm-up Protocol

In this section, we give a warm-up protocol to help build intuition. The warm-up protocol has a high computational cost and will not be used as is. In the next subsection, we will present a technique to improve the computational cost.

In our protocol, the client query consists of vectorized ciphertexts, and the server database is also encoded as an array of plaintext vectors. We use the standard *hierarchical query* technique [6] to reduce the request size. In this technique, a database with N entries are represented as a d-dimensional hypercube, where each dimension is of size $N' = \sqrt[d]{N}$.

The number of dimensions d plays a key role in performance. A larger d means a smaller request size but a higher multiplicative depth, which requires less efficient RLWE parameters and hence higher computation cost. We found that setting $d = 3$ provides a decent trade-off between request size and computation in most of our experiments. Therefore, we will describe the protocol for $d = 3$, i.e., the database is represented as a cube with each dimension having size $N' = \sqrt[3]{N}$.

We think of the cube as having N' slices, where each slice is a $N' \times N'$ matrix. For now, we assume that within each slice, each column is a separate plaintext vector of size $n = N'$. In this cube, any desired entry can be accessed by successively selecting the correct row (first dimension), then column (second dimension), and eventually slice (third dimension). We will now describe how we perform these selections privately.

To access a database entry, the client represents the index as $d = 3$ one-hot query vectors each of size N' (responsible for selecting the row, the column, and the slice, respectively). The client then encrypts each query vector into a separate ciphertext. ① The server performs *ciphertext-plaintext multiplication* between the first query ciphertext and each plaintext vector. ② As a result, each slice now contains N' *encrypted* columns each containing a single non-zero entry at the same slot. The server then merges the encrypted columns within each slice after homomorphic rotations. Specifically, within each slice, the server rotates the encrypted columns in increments of one so that their non-zero entries are now in distinct slots; the server then sums them up. ③ The output is a matrix of $N/{N'}^2$ encrypted columns where each column holds the selected row from one slice.

The server then processes the second dimension, this time using *ciphertext-ciphertext multiplication* between the second query vector and the previous output. ④ The same rotate-then-sum step is performed. This results in a single ciphertext having one entry from each slice. ⑤ The server will multiply this with a third query vector. ⑥ the output ciphertext has a single non-zero slot containing the desired entry.

Figure 2 illustrates the warm-up vectorized PIR protocol. The server database coonsisting of 27 entries is represented in $d = 3$ dimensions each of size 3 and the client query consists of $d = 3$ query ciphertexts. The plaintext vector size n is 3.

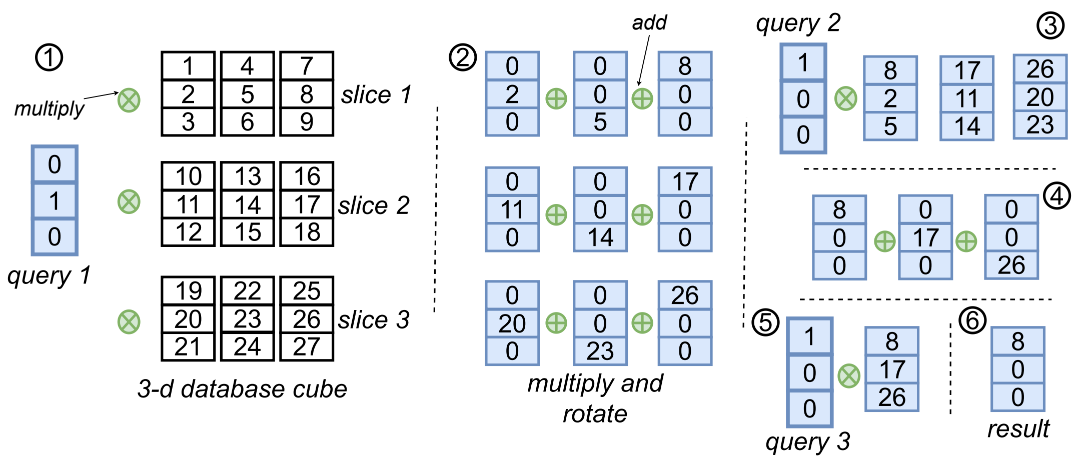

Figure 2: A basic vectorized PIR protocol in three dimensions. The database consists of 27 entries and each dimension is of size three. The query consists of three RLWE ciphertexts.

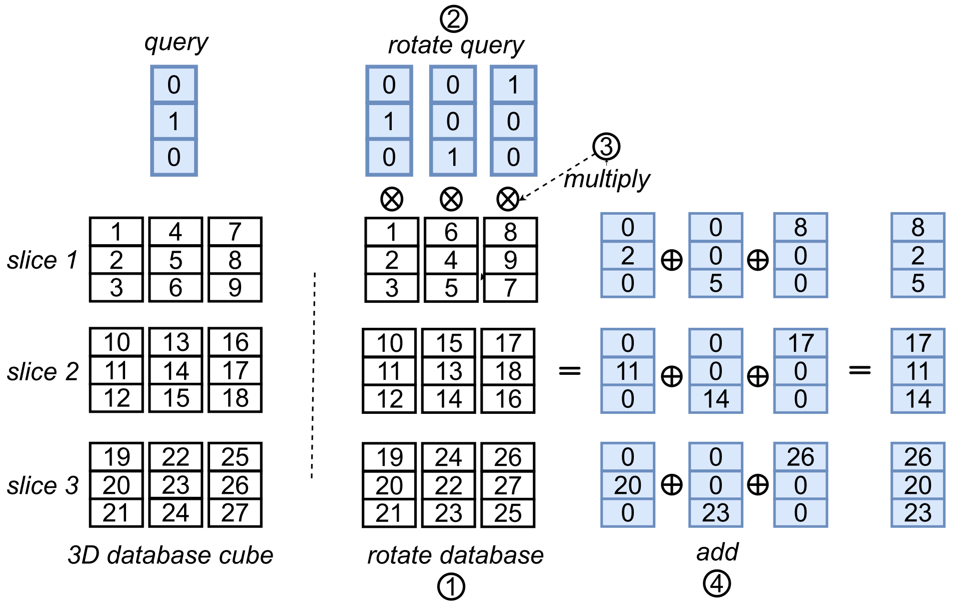

Figure 3: A method to reduce the number of rotations in the first dimension of vectorized PIR. The database of 27 entries is encoded in three dimensions. In each 2 dimensional slice, the server rotates the second column by one and the third column by two during initialization. The server also copies and rotates the encrypted query ciphertexts. The server performs three encrypted rotations (instead of 9 with the method in Figure 2).

B. Rotating Query to Reduce Computation

The main limitation of the above approach is the high computational cost. As shown in Table II, ciphertext rotation is an expensive operation. Therefore, the N/N' ciphertext rotations in the first dimension will be the computation bottleneck. In this subsection, we observe that if the server rotates the first query (instead of the first dimension's output), the number of rotations can be greatly reduced from N/N' to N'.

To achieve this, we make two changes to the warm-up protocol, shown in Figure 3. ① At initialization, after representing the database into a cube, for every slice, the server rotates the plaintexts column in increments of one. These cheap plaintext rotations will help avoid expensive ciphertext rotations at the time of the query. ② At the time of query, the server copies the first query ciphertext N' times and then rotates each copy in increments of one, in total N' ciphertext rotations. ③ Then within each slice, multiply i-th

Algorithm 1: VecPIRSetup algorithm

```
Input: DB, Input database with $N = |\mathsf{DB}|$ entries
Output: DB′, A database with plaintext vectors
1  $N' = \sqrt[d]{N}$
2  for $i = 0 : \lceil N/N' \rceil - 1$ do
3      $\mathsf{DB}'[i] = \mathsf{EncodeToPt}(\mathsf{DB}[iN' : (i+1)N' - 1])$
4      Rotate $\mathsf{DB}'[i]$ by $i \cdot n/N' \mod n$
5  end
6  outputs DB′

7  Function $\mathsf{EncodeToPt}(v)$
       Output: $p$, Plaintext vector of size $n$
8      $N' = |v|$
9      $g = n/N'$
10     for $j = 0 : N' - 1$ do
11         $p[j \cdot g] = v[j]$
12     end
13     Output $p$
```

Algorithm 2: VecPIRQueryGen algorithm.

```
Input: idx, Desired index; $N'$, dimension size
Output: $\tilde{Q}$, A list of $d$ ciphertext query vectors
1  Represent idx as $d$-dimensional coordinates of the
     hypercube $(\mathsf{idx}_0, \cdots \mathsf{idx}_{d-1})$
2  Compute $(\mathsf{idx}'_0, \cdots \mathsf{idx}'_{d-1})$ where $\mathsf{idx}'_j = \sum_{k=0}^{j} \mathsf{idx}_k$
     mod $N'$
3  for $i = 0 : d - 1$ do
4      Generates vector $b$ as one-hot encoding of $\mathsf{idx}'_i$
5      $\tilde{Q}[i] = \mathsf{EncodeToPt}(b)$
6  end
7  Output $\tilde{Q}$
```

plaintext with the i-th rotated copy of the query ciphertext and sum the resulting encrypted columns. The overall effect of this approach is the same as the warm-up protocol, i.e. the output of the first dimension is a matrix with N/N'^2 encrypted columns. But the number of rotations performed at the time of query is now N'. Recall that $N' = \sqrt[3]{N}$. Thus, we have $N' < N/N'$.

C. Full Vectorized PIR Protocol

Setup. The server uses Algorithm 1 to encode database with N entries into $\lceil N/N' \rceil$ plaintexts. Specifically, the algorithm iterates over groups of N' consecutive entries, encoding each group into an individual plaintext vector of size n. For most RLWE SHE schemes, the number of plaintext slots n is several thousand. As we set $d = 3$ so $N' = \sqrt[3]{N}$. Therefore, for all practical databases $N' < n$. The algorithm places N' entries at equal distances in a plaintext vector.

Algorithm 3: VecPIRResp algorithm.

```
Input:
- $\tilde{Q}$, Set of PIR query ciphertexts
- DB′, Encoded plaintext PIR database
- $N'$, Dimension size
Output:
- $R$, a single ciphertexts encrypting PIR response.

1  $g = n/N'$
   ▷ Query rotate and Ct-Pt multiply for first dimension
2  for $k = 0 : N' - 1$ do
3      $\tilde{Q}_{\text{rot.}}[k] = \mathsf{CtRotate}(\tilde{Q}[0], k \cdot g)$
4  end
5  for $j = 0 : |\mathsf{DB}|/N' - 1$ do
6      $\mathsf{DB}'[j] = \sum_{k=0}^{N'-1} \mathsf{CtPtMul}(\tilde{Q}_{\text{rot.}}[k], \mathsf{DB}[jN' + k]))$
7  end
               ▷ Ct-Ct multiply for latter dimensions
8  for $i = 1 : d - 1$ do
9      $\mathsf{C}[0 : |\mathsf{DB}'|/N' - 1] = 0$
10     for $j = 0 : |\mathsf{DB}'|/N' - 1$ do
11         for $k = 0 : N' - 1$ do
12             $C' = \mathsf{CtCtMul}(\tilde{Q}[i], \mathsf{DB}'[jN' + k])$
               $\mathsf{C}[j] = \mathsf{C}[j] + \mathsf{CtRotate}(C', k \cdot g))$
13         end
14     end
15     $\mathsf{DB}' = \mathsf{C}$    ▷ Encrypted intermediate database
16 end
17 Output $R = \mathsf{DB}'$     ▷ Final encrypted response
```

In other words, within a plaintext vector, entries are placed at multiples of g slots. where $g = n/N'$. Encoding plaintext entries this way will help us merge the requests ciphertexts in our BatchPIR protocol later in Section IV. The encoded database consists of $\lceil N/N' \rceil$ plaintexts. The server then rotates the database as discussed in Section III-C.

Client query generation. The client uses Algorithm 2 to generate a query. The algorithm first represents idx as the d-dimensional coordinates of the hypercube, where the i-th coordinate is the position of the desired entry in the i-th dimension. Because of rotations in the server computation, the position of the desired entry will shift, but these shifts are public and hence can be easily calculated by the client (Line 2). Finally, the client represents these shifted coordinates as d one-hot query vectors each of size N'. Similar to database encoding, query vectors are encoded at equal distances in the plaintext vector. Each encoded query vector is then encrypted as a separate ciphertext.

Response generation. The response generation algorithm is given in Algorithm 3. The server uses the client query and the encoded database to generate a single response ciphertext. For the first dimension, the server follows the above approach using rotated copies of the first query ciphertext and the rotated database. We observe that for the second and higher dimensions, query rotation does not reduce the number of ciphertext rotations. For these dimensions, we stick with rotating and summing the resulting ciphertexts after multiplying the query vector with the encrypted output of the previous dimension, as in the warm-up protocol.

Recall that the output after the final dimension is a single ciphertext that contains the desired entry in one of the slots. If the protocol is used as a standalone PIR, then this ciphertext is returned to the client. But in the next section, we will further process this ciphertext at the server to lower the communication of our BatchPIR protocol, so there is no need to send it back to the client.

IV. Vectorized Batch Private Information Retrieval

As mentioned earlier, our BatchPIR protocol is built atop Angel *et al.* framework given in Section II-E. The server divides the database into $B = 1.5b$ buckets using $w = 3$ independent hash functions, the average size of each bucket is $N_B \approx wN/B$. The client divides a batch of b indices into B buckets using cuckoo hashing. Next, we will invoke our vectorized PIR protocol for each bucket. Each bucket is represented as a $d = 3$ dimensional hypercube with each dimension size $N'_B = \sqrt[3]{N_B}$. The client request consists of 3 ciphertexts per bucket. The response consists of B vectorized ciphertexts (one per bucket) where each ciphertext has at most one non-zero slot containing the desired entry.

We next introduce mechanisms to reduce communication. As long as $B < n$ (which is usually the case since n is quite large), the server will merge the B response ciphertexts into a single ciphertext. Observe that if all the desired entries reside in different slots, the server can merge these responses into a single ciphertext simply using vectorized homomorphic additions. Unfortunately, response ciphertexts from different buckets may collide, so the server could not directly merge them. We propose a mechanism that publicly aligns these entries. This directly reduces the response size. We then propose a mechanism to pack requests for multiple buckets together, which results in significant savings in request size.

A. Merging Response Ciphertext

We illustrated our technique for $B = 4$ response ciphertexts in Figure 4. Recall that after running vectorized PIR for each bucket the server holds B response ciphertexts, each having at most a single entry. Each desired entry could be at any arbitrary index within the respective bucket. As a result, there may be collisions in the slots occupied by desired entries from different buckets. For example, Figure 4, entries b and d collide.

Therefore, we can not add them directly. We do not want the client to send extra encrypted material to guide the server. Because that will increase the request size. The key challenge is to enable the server to merge these ciphertexts publicly without having additional interaction with the client.

Given B response ciphertexts, our merging technique works as follows. $\textcircled{1}$ The server first copies the non-zero value to all slots. This is achieved with a *rotate-and-sum* approach. The server first rotates the ciphertext by one position and adds the rotated and the original ciphertext. This copies the non-zero value to an adjacent slot. This step is then repeated $\log n - 1$ times, each time, rotating the output of the previous step by the next power of two and adding rotated ciphertext to the previous ciphertext. The result is a ciphertext that contains the same value at all slots. $\textcircled{2}$ Then the server publicly selects a *distinct* slot from each ciphertext by multiplying it with a plaintext binary one-hot mask that is 1 only at the selected slot. Now each ciphertext has a distinct non-zero slot. $\textcircled{3}$ The server finally merges these ciphertexts by homomorphically adding them together.

This technique allows us to merge the responses from up to n buckets, where n is a polynomial degree. In our implementation, n is set to $8,192$. Thus, even for a batch with thousands of indices, the final response of our BatchPIR protocol is a single ciphertext.

B. Packing Request Ciphertexts

Each query vector is of size $N'_B = \sqrt[3]{N_B}$. As mentioned in the previous section, each query vector is much smaller than the ciphertext dimension n. Therefore, we can pack query vectors from multiple buckets into a single ciphertext. Care is needed to ensure that the packing technique is compatible with the rotation and summation that the server performs for each dimension of the Vectorized PIR.

We illustrated an example of our packing technique in Figure 5. In the figure, the client has two batches of queries, each containing two query vectors. The server database is divided into two buckets, each containing four elements represented as a 2×2 matrix.

For each batch, the client assigns query vectors to alternate slots of the request ciphertext $\textcircled{1}$. The server encodes the buckets in the same fashion. Entries from two buckets are encoded into alternating plaintext slots $\textcircled{2}$. Note that for both batches, rotating the second column by two slots (independent of the query) avoids collisions of non-zero entries and allows the results to be merged together $\textcircled{3}$.

More generally request packing works as follows: Given query vectors from g_B buckets, where each bucket is of size N_B and $g_B = n/N'_B$. Note that in practice buckets are of unequal sizes. But the server will extend them to the size of the largest bucket (denoted as N_B from here on) by padding zero entries. For the first dimension, take the first query vectors of all the buckets and pack them into alternating slots

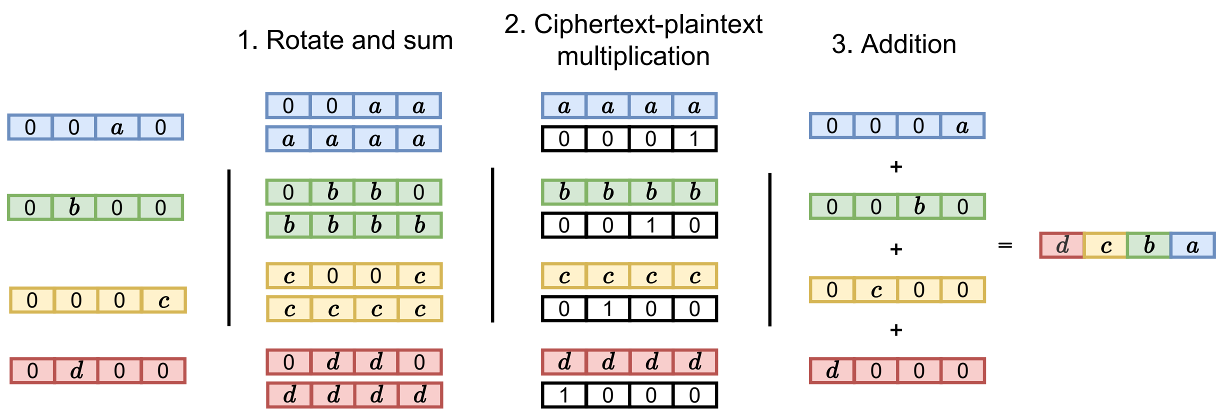

Figure 4: Response merging across the buckets. Entries within different buckets overlap. To align entries into distinct slots first each entry is copied to all slots using rotate and sum. After that distinct slot for each entry is picked.

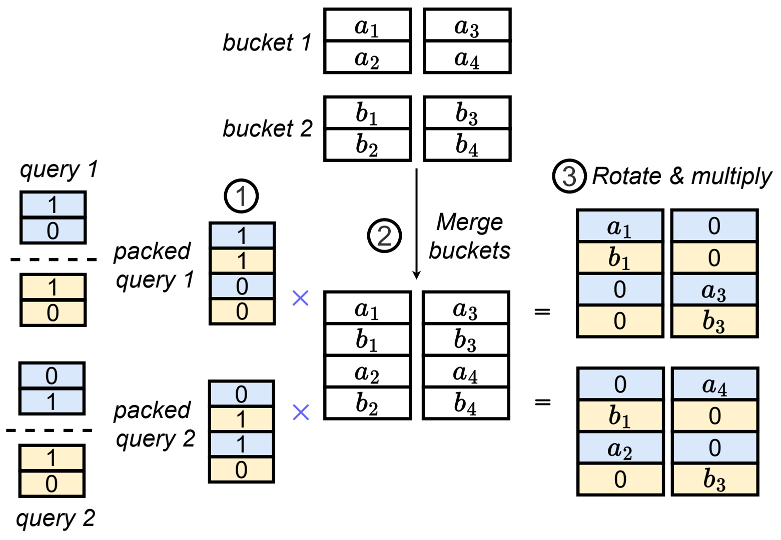

Figure 5: Examples of query packing. Query vectors from different buckets are assigned to alternative slots of the ciphertext. Similarly, entries from different buckets are assigned to alternate slots of the plaintexts.

of an independent plaintext. Specifically, assign values of k-th first query vector to plaintext slots congruent with k mod g_B. Repeat this step for the second and third dimensions. The output will be $d = 3$ packed query vectors one for each dimension. The server encodes the buckets' data in the same way. Pack groups of consecutive N'_B entries from all the g_B buckets into an independent plaintext vector. The output consists of N_B/N'_B plaintexts. Concretely, for buckets $B = 256$, largest bucket size $N_B \approx 8192$, and dimension size $N'_B = 32$, for each dimension query vectors from all the buckets can be packed into a single ciphertext. Therefore the client request would only be $d = 3$ ciphertexts.

Note that because of request packing each response ciphertext now consists of g_B desired entries. Therefore, the rotate and sum step of response merging is repeated $\log n/g_B - 1 = \log N'_B - 1$ times (instead of $\log n - 1$), and each time response ciphertext is rotated in multiples of g_B.

C. *Putting it all together*

We have introduced all the components of our vectorized BatchPIR protocol separately in previous sections. Now we describe our full vectorized BatchPIR protocol by putting together all the techniques. The final protocol is given in Algorithm 7.

Our protocol has a setup phase that the server has to perform only once for all the clients. The server first runs setup using Algorithm 4. In the setup phase, the server divides the database into B buckets using the server hashing technique of Angel *et al.* framework as given Section II-E. After that the server extends the size of each bucket to the size of largest bucket by appending zero entries, and then independently encodes each bucket as d dimensional

Algorithm 4: BatchPIRSetup algorithm

```
Input:
  - DB, Database of $N$ entries
  - System parameters given in Algorithm 7
Output:
  - $\widetilde{\mathsf{DB}}_0, \cdots, \widetilde{\mathsf{DB}}_{\lceil B/g_B \rceil - 1}$, Merged plaintext buckets
1 For each entry $\mathsf{DB}_i$, copy it to buckets
    $h_1(i), \cdots, h_w(i)$   ▷ server hashing, average size of bucket is $wN/B$
2 Pad each bucket $\mathsf{B}_i$ to size $N_B$ with dummy entries
3 for $i = 0 : B - 1$ do
4     $\mathsf{B}'_i \leftarrow \mathsf{VecPIRSetup}(\mathsf{B}_i)$
5 end
6 for $i = 0 : \lceil B/g_B \rceil - 1$ do
7     $\widetilde{\mathsf{DB}}_i = \sum_{k=0}^{g_B-1} \mathsf{PtsRotate}(\mathsf{B}'_{i \cdot g_B + k}, k)$
8 end
9 Output $\widetilde{\mathsf{DB}}_0, \cdots, \widetilde{\mathsf{DB}}_{\lceil B/g_B \rceil - 1}$
10 $\mathsf{PtsRotate}(P, k)$: A function takes as input a list of
    plaintext vectors $P$ and a value $k$ and rotates each
    plaintext vector in the list by $k$ slots.
```

Algorithm 5: BatchPIRQueryGen algorithm

```
Input:
  - $I = \{i_1, i_2, \cdots, i_b\}$, Client query batch
  - System parameters given in Algorithm 7
Output:
  - $\tilde{Q}_0, \cdots, \tilde{Q}_{B/g-1}$, A list of encrypted queries, each
    consisting of $d$ RLWE ciphertexts
1 Apply cuckoo hashing to each index in $I$ using hash
    functions $h_1, \cdots, h_w$
2 for $i = 0 : B - 1$ do
3     $Q_i \leftarrow \mathsf{VecPIRQueryGen}(j_i, N'_B)$ ▷ $j_i$ is the index
      in $i$-th bucket the client wants to retrieve
4 end
5 for $i = 0 : \lceil B/g_B \rceil - 1$ do
6     ${Q'}_i = \sum_{k=0}^{g-1} \mathsf{PtsRotate}(Q_{i \cdot g_B + k}, k)$
7 end
8 Encrypt $Q'_0, \cdots, Q'_{B/g_B-1}$ to get $\tilde{Q}_0, \cdots, \tilde{Q}_{B/g_B-1}$
```

hypercube using algorithm 1. After that, the server merges g_B buckets. For this take the j-th plaintext of each bucket, rotate each in increasing order, and sum together. In this way, the result is $\lceil B/g_B \rceil$ merged buckets.

The client generates a query using Algorithm 5. The client first divides the batch into B buckets using cuckoo hashing. For each bucket, the client will map the database index to the bucket index using one of the techniques discussed in Section II-E. Then, for each bucket, the client generates d query vectors using Algorithm 2. Then the client packs query vectors of g_B buckets together and encrypts each packed query vector into an independent ciphertext. The result is a list of $\lceil B/g_B \rceil$ PIR queries.

Algorithm 6: BatchPIRRespMerge algorithm.

```
Input:
  - $\{R_i\}_{i \in [k]}$, List of response ciphertexts, assuming
    $k \cdot g_B \leq n$
  - System parameters given in Algorithm 7
Output:
  - $T$, Merged response ciphertext
1 Set $R' = 0$ and $T = 0$
2 for $i = 0 : k - 1$ do
3     for $j = 0 : \log(n/g_B) - 1$ do
                          ▷ copying to all slots
4         $R' = \mathsf{CtRotate}(R_i, g_B \cdot 2^j)$
5         $R_i = \mathsf{CtxtCtxtAdd}(R_i, R')$
6     end
7     Set plaintext vector $p = 0$.
8     $l = i \cdot g_B$          ▷ selecting distinct slots
9     $p[l : l + (g_B - 1)] = 1$
10    $R' = \mathsf{CtxtPtxtMul}(R_i, p)$
11    $T = \mathsf{CtxtCtxtAdd}(R', T)$
12 end
13 Output $T$
```

For each merged buckets, the server calls Algorithm 3 and collects all the response ciphertexts. Each response ciphertext contains g_B non-zero slots.

After that, the server calls Algorithm 6 to merge these response ciphertexts. For each response ciphertext, the server first copies non-zero values to all ciphertext slots by using rotate-and-sum. The server then selects g_B distinct slots from each response ciphertext by multiplying it with a plaintext binary mask and then merges the resulting ciphertexts using homomorphic addition. As long as number of buckets B is less than ciphertext dimensions n, the result will be a single ciphertext, containing all the desired entries.

D. Efficiency Analysis

Communication between client and the server. We now calculate how many ciphertexts the client and the server exchange in the BatchPIR protocol. The communication from the client to the server includes packed requests for underlying vectorized PIR and the response ciphertexts. Concretely, the client sends $\lceil B/g_B \rceil$ packed PIR queries to the server. The query for each PIR contains d ciphertexts. Due to merging the response consists of $\lceil B/n \rceil$ ciphertexts. So in total, $d \cdot \lceil B/g_B \rceil + \lceil B/n \rceil$ ciphertexts are exchanged between the client and the server.

Recall that g_B represents the ratio between plaintext size

Algorithm 7: Full Vectorized BatchPIR Protocol.

```
Input:
  - DB, Server database of size $N$
  - $I = \{i_1, i_2, \cdots, i_b\}$, Indices of entries client wants to retrieve
System Parameters:
  - $b$, Client query batch size $|I|$
  - $B$, Number of buckets, usually set to $1.5b$
  - $h_1, \cdots, h_w$, Hash functions
  - $N_B$, Size of the largest bucket
  - $N'_B$, Size of first two dimensions, set to a power of two larger than $\sqrt[3]{N_B}$
  - $n$, Number of slots per ciphertext
  - $g_B = n/N'_B$
1 Server prepares database $\mathsf{DB}'_0, \cdots, \mathsf{DB}'_{\lceil B/g_B \rceil - 1}$ by calling BatchPIRSetup(DB) from Algorithm 4
2 Client creates queries $\tilde{Q}_0, \cdots, \tilde{Q}_{\lceil B/g_B \rceil - 1}$ by calling BatchPIRQueryGen($I$) from Algorithm 5 and sends them to the server.
3 for $i = 0 : \lceil B/g_B \rceil - 1$ do
4     Server computes $R_i = \mathsf{VecPIRResp}(\tilde{Q}_i, \mathsf{DB}'_i, N'_B)$
5 end
6 Server computes $T = \mathsf{BatchPIRRespMerge}(\{R_i\}_{i \in \lceil B/g_B \rceil})$ and sends it to the client
7 Client decrypts $T$ to get the entries corresponding to $I$
```

and the dimension size of bucket data n/N'_B. Therefore, increasing the size of N'_B will increase the request size.

Server computational cost. In the setup phase, the server performs one-time hashing of the database. The server will use the same hashed database for subsequent queries and across all the clients. As discussed in Section II, additions in SHE are quite cheap. Therefore, we focus on computational cost due to multiplications and rotations.

The most computationally demanding step of BatchPIR is B/g_B calls to Algorithm 3. In each call, the input database consists of $\approx N_B/N'_B$ plaintexts, where $g_B = n/N'_B$. The algorithm performs dimension-wise computation. Specifically, for the first dimension, the server performs N_B/N'_B CtPtMul operations. For the second dimension $N_B/{N'_B}^2$ CtCtMul operations are required and for the third dimension, only single CtCtMul operation is needed. Before multiplication of the first dimension, the server also performs N'_B CtRotate operations and for the second dimension, the CtRotate operations are N_B/N'_{B^2}. The third dimension does not CtRotate operation. Additionally, $\log N'_B - 1$ CtRotate operations are needed for response merging.

In total, our algorithm requires BN_B/n CtPtMul operations, $B/n(N_B/N'_B + N'_B)$ CtCtMul operations and $B/n({N'_B}^2 + N_B/N'_B + N'_B(\log N'_B - 1))$ CtRotate operations. For a small value of N'_B, CtCtMul operations are the computational bottleneck. However, if we raise the value of N'_B, then CtRotate operations consume the most computation. In our experiments, N'_B is set such that CtCtMul operations take up the most computation.

E. Additional Details and Extensions

Dimension size. A subtle issue is that the rotation operations as described in query packing work only if the size of each query vector is a power of two (or divides the polynomial degree n). To make this step work, we set the size of the first two dimensions to the same power of two. Since no rotations are needed for the third (and last) dimension, we can set its value to what naturally remains, usually

Modulus switching to reduce response size. At the end of the BatchPIR protocol, the server sends response ciphertexts to the client who will decrypt them. In other words, the response ciphertexts will not be used for further computation. We can then use the modulus switching technique to reduce the size of the response ciphertexts. This technique was first applied to PIR in [33]. Recall that RLWE ciphertexts are elements in R_q^2. Suppose a response ciphertext c has noise $\mathsf{Err}(c)$. Modulus switching transforms c to $R_{q'}^2$ with noise $\max(\sqrt{n}, \mathsf{Err}(c) \cdot q'/q)$, where n is the polynomial degree. To ensure correctness of decryption, we set $q' \approx \sqrt{n}t$. This optimization reduces the response size by $\approx \log q / \log \sqrt{n}t$.

Short random seed to reduce request size. We can further reduce the request size by using a simple optimization given in [3]. Recall that in a fresh RLWE ciphertext, the first component c_0 is sampled uniformly randomly from R mod q. Thus, instead of a sending truly random c_0, the client can send a short random seed to generate a pseudorandom c_0. This optimization reduces the request size by half.

Handling larger items. Recall that our output ciphertext contains B non-zero slots. For many practical scenarios, B is smaller than n, so the output ciphertext has space for more data. We can exploit the space to handle entries larger than t bits. Specifically, the server splits each entry into chunks of t bits. One can see it as multiple sub-databases $\mathsf{DB}^1, \mathsf{DB}^2, \cdots, \mathsf{DB}^k$, where DB^i corresponds to i-th chunk of all entries. The server then applies the same client query to all sub-databases and obtains multiple output ciphertexts each containing B non-zero slots. Lastly, the server can merge these output ciphertexts using the rotate-and-sum approach again.

Note that the request will remain the same, i.e. $d\lceil B/g_B \rceil$ ciphertexts. But the response will be $\lceil kB/n \rceil$ ciphertexts. Similarly, the server computation will be k times larger than compute given in previous section. With one exception that the term $B{N'_B}^2/n$ term does not get multiplied with k this is because we can use the same rotated query for all the chunks.

V. Implementation

We have implemented our scheme in C++ atop of the Microsoft SEAL Library version 3.7.2 [27]. Following SEAL, we use the BFV [21], [28] encryption scheme, though our protocol could easily be implemented using other encryption schemes such as BGV. We implement the client and server hashing scheme of Angel *et al.* scheme using SHA256 and the crypto++ library [34]. For cuckoo hashing, we set the maximum number of iterations to 200.

Noise analysis of Vectorized BatchPIR. We will analyse noise in our protocol using noise estimates of BFV encryption given in Section II. In BFV rotations and additions adds insignificant amount of noise. Therefore, it is sufficient to estimate noise growth due to multiplications. Algorithm 3 sequentially performs 1 ciphertext-plaintext multiplication followed by $d-1$ ciphertext-ciphertext multiplications (on same ciphertext). Additionally, to merge responses across the buckets, one extra ciphertext-plaintext multiplication is used. Total multiplicative depth of the protocol is $d+1$. Therefore the expected noise in the BatchPIR response ciphertext will be $O(\sqrt{n}(t^{d+1}\mathsf{Err}(c) + \mathsf{Err}(c)^d))$. In other words the response will correctly decrypt as long as coefficient modulus q is greater than $O(\sqrt{n}(t^{d+2}\mathsf{Err}(c) + t\mathsf{Err}(c)^d))$. In our implementation we set q slightly higher than the required size, to ensure high correctness.

BFV parameters. The performance of our scheme depends on our choice of BFV parameters. We test our scheme with different parameters to find the best choices. We set the polynomial degree n to 8192 and the ciphertext modulus q to 200 bits. We use SEAL's default configurations for standard deviation error and secret key distribution. This gives us more than 128 bits of computational security. SEAL implements the RNS variant of BFV, in q is broken up into several smaller co-primes $q_1, q_2, q_3, \cdots$. Ciphertexts are represented in Chinese remainder theorem (CRT) form, so operations on ciphertexts are performed on the smaller rings $R \mod q_i$. An extra RNS component of q is required for the key material to reduce noise in the relinearization and key switching steps for multiplications and rotations. This extra component is not involved in the client query or the server response. We pick this extra component to be of 50 bits, leaving the modulus of the query and response ciphertexts to be 150 bits.

As mentioned above, we use a modulus reduction in the end to reduce the response. Concretely, the reduced modulus q' is only 30 bits, giving a 4x reduction in response size.

Plaintext encoding in BFV. In SEAL's implementation of BFV, a plaintext is represented as a $n/2 \times 2$ matrix (instead of length-n vector). We note that this is not an issue for us because we can divide buckets into two equal-sized groups and then encode one group per column. To elaborate, given g buckets that can all fit into a plaintext vector of size n, we first divide them into two groups each of size $g/2$ and then encode the two groups into the two columns of the plaintext matrix. All our algorithms work directly on this encoding with the only minor change that we will rotate till $n/2$ (instead of n).

VI. Evaluation

Experimental setup. We run our experiments on Amazon EC2 instances. Specifically, we used a t2.2xlarge instance with 32 GB RAM. The instance has 8 CPU cores but our implementation is single-threaded for ease of comparison with prior works. All the results are obtained by running each experiment 10 times and taking the average.

Benchmarking BatchPIR. In Table III we demonstrate the performance of our scheme when retrieving a batch of 32 to 1024 entries. For all these experiments, the server database consists of one million entries and each entry is 256 bits. As mentioned, 256-bit entries are common in many applications.

Observe that both request and response sizes increase with the batch size. The request size increases because a larger batch corresponded to more buckets and hence more ciphertext query vectors. The response size increases because more response ciphertexts are needed to hold all the entries in a larger batch.

It is worth noting that the request size accounts for a larger proportion in total communication. This is because, for each group of g buckets, Algorithm 7 needs three query ciphertexts (one for each dimension), but only one response ciphertext. Furthermore, the modulus reduction at the end helps reduce response size (more substantial than the effect of short seed in request).

Our protocol inherits the efficient computation from the Angel *et al.* paradigm. As shown in Table III, it takes 10 to 18 milliseconds (amortized per query) for different batch sizes. Other than a batch of size 32, the computational cost is dominated by the second dimension. This is because the second dimension involves $N_B/{N'_B}^2$ expensive ciphertext-ciphertext multiplications (recall N'_B is the size of the first two dimensions). This cost could be reduced by increasing N'_B but that will increase the size of each query vector (which is the communication bottleneck).

Comparison with prior work. We demonstrated the performance of our protocol by comparing it with Angel *et al.*'s BatchPIR scheme [12] which uses SealPIR as the underlying PIR. To the best of our knowledge, this is the only practical BatchPIR in the literature. Unfortunately, the publicly available implementation of this scheme [36] currently gives compilation errors, and the authors acknowledged that the implementation is no longer functional. Thus, for computation cost, we can only compare with the single data point given in their paper [12], i.e. for a batch of size 256 and a database with one million entries where each entry is 288 bytes. Fortunately, the communication cost of their scheme can be easily calculated and compared with.

	$b = 32$ $B = 48$	$b = 64$ $B = 96$	$b = 128$ $B = 192$	$b = 256$ $B = 384$	$b = 512$ $B = 768$	$b = 1,024$ $B = 1,536$
First two dimensions size	128	64	64	64	32	32
Third dimension size	5	9	5	3	5	3
Request (MB)	0.45	0.45	0.90	1.35	1.35	2.70
Response (MB)	0.06	0.06	0.06	0.06	0.12	0.18
Total Communication (MB)	0.51	0.51	0.96	1.41	1.47	2.88
First Dimension (Sec)	1.33	1.06	1.03	0.93	0.77	0.76
Second Dimension (Sec)	0.75	1.63	1.63	1.28	2.47	2.47
Total Computation (Sec)	2.50	3.17	3.32	2.94	4.10	4.52

Table III: Performance of our scheme for batch b sizes 32, 64, 128, 512, 1024. For all the cases the server database consists of one million entries and each entry is of 32 Bytes.

	Our work	Angel *et al.*
Initialization (Sec)	22.2	14.2
Communication (MB)	1.26	120
Computation (Sec)	41.6	20

Table IV: Performance comparison between our scheme and Angel *et al.* [12] batch PIR scheme for a batch of size 256 entries, where each entry is of 288 bytes and the server database consists of one million entries.

Entry size	Communication cost (MB)	
(Byte)	Our work	Angel *et al.*
4	0.93	120
16	0.96	120
64	1.02	120
256	1.20	120

Table V: Communication overhead (in MegaBytes) of our scheme and Angel *et al.* [12] BatchPIR scheme. For all experiments, we assume that the client batch consists of 256 entries and the server database has one million entries. Each database entry size is set to 4, 16, 64, or 256 bytes.

As depicted in Table V, the communication in our scheme is 75$\sim$129x smaller than the Angel *et al.* scheme for different entry sizes. In terms of computation, Table IV shows that the Angel *et al.* scheme is slightly better. Concretely, the computation during initialization of our scheme is about 1.5x higher than theirs and the computation at query time is about 2x higher than theirs. When amortized per query, the computation cost of our scheme is 162 milliseconds compared with 78 milliseconds for their scheme.

To provide one more comparison point, we also compared with Cong *et al.* [35] Labeled Private Set Intersection (LPSI) scheme, even though it is not a fair comparison. LPSI is a stronger primitive than BatchPIR in that it also protects the server's data privacy, i.e. the client should not be able to learn any information about the server data not in the intersection. LPSI can be directly used as a BatchPIR if the client and the server do not perform blinding of their respective inputs using an oblivious pseudorandom function; thus, we exclude the cost of OPRF evaluation from their server initialization time. The results are given in Table VI. We consider a server database consisting of one million entries and a client batch of 256 indices. We test three entry sizes: 4, 16, 64, and 256 Bytes.

The LPSI scheme has significantly higher server initialization than our scheme. The high initialization is because the server has to perform expensive polynomial interpolations. This makes their scheme undesirable for applications where the database updates frequently. The communication of our scheme beats the LPSI scheme in all cases. Specifically, for a small entry size of 4 bytes, our scheme demonstrates 2.5x improvement in computation and 3.7x improvement in communication. We observe that the communication in the LPSI scheme increases rapidly with the entry size. For an entry size of 256 bytes, their scheme has 9.2x times more communication than our scheme. We also note as the entry size increases, the computational costs of the two schemes become comparable. In conclusion for fixed batch size, our scheme has a better computational performance when the entry size is small. While for bigger entries (>256 bytes), the LPSI scheme has a lower computational overhead. Nevertheless, in both cases, our scheme has better communication performance.

VII. Related Work

Private Information Retrieval (PIR) is first introduced by Chor *et al.* [1]. There is an extensive list of works that rely on multiple non-colluding servers. Since the focus of our paper is a single server, we focus on single-server PIR schemes in this section.

Early single-server PIR schemes. Kushilevitz and Ostrovsky proposed the first single-server PIR protocol [6]. Their scheme is based on *additively homomorphic encryption*. The database is represented as a d dimensional hypercube, which results in a request size of $O(N^{1/d}K)$ and a response size of $O(N^{1/d}K^{d-1})$, where K is the ciphertext

	4 Bytes		16 Bytes		64 Bytes		256 Bytes	
	Our work	LPSI	Our work	LPSI	Our work	LPSI	Our work	LPSI
Server's Init. (Sec)	11.6	2038.8	13.4	3359.6	16.7	7364.9	20.2	4438.7*
Communication (MB)	0.93	3.7	0.96	4.1	1.47	5.6	1.20	11.08
Computation (Sec)	1.8	4.6	3.9	5.6	7.08	7.8	39.7	7.6*

Table VI: Comparison of our scheme with Labelled PSI (LPSI) scheme [35]. For all experiments, we assume that the client batch consists of 256 entries and the server database consists of one million entries. Each database entry size is 4, 16, 64, or 256 bytes. For 256-byte entries, we run LPSI with 8 threads because their initialization phase is prohibitively slow on a single thread.

expansion factor. After their work, several works further improved the asymptotic communication cost using various techniques and assumptions [7]–[9], [37]. But Sion and Carbunar [38] observed that these schemes in practice often perform slower than downloading the entire database when the network bandwidth is a few hundred Kbps. The poor practical performance of these schemes is due to the fact that the server needs to perform at least N big-integer modular multiplications or modular exponentiations. The computation cost of these operations is often higher than simply sending the data to the client.

Recent practical single-server PIR schemes. Recent practical single-server PIR constructions use lattice-based cryptography. In particular, they use Somewhat Homomorphic Encryption (SHE) schemes based on the Ring learning with error (RLWE) assumption. At a high level, these schemes followed the hierarchical PIR blueprint of Kushilevitz and Ostrovsky and represent the database as a d-dimensional hypercube. For the first dimension, the server performs a dot-product between the encrypted client query and the plaintext database. For subsequent dimensions, the dot-product is between the ciphertext output of the previous dimension and the encrypted client query. Ciphertext-ciphertext multiplication in RLWE encryption is expensive and is the bottleneck. Hence, these schemes mainly differ in how they handle multiplications in the second and higher dimensions.

These schemes only achieve low communication overhead when the database entry is large (tens of KiloBytes). For databases with small entries, the communication overhead is very high.

Aguilar-Melchor *et al.* [11] proposed the first such scheme called XPIR. XPIR significantly improved the computation cost over earlier schemes but its communication overhead is prohibitively high. For example, to retrieve a 256-Byte entry from a database with one million entries, its total communication is more than 17 MB, about $70,000$x of the plaintext entry. This is mostly due to the large request size but the response size is also quite large.

SEALPIR [12] addresses the request size bottleneck by introducing the query compression technique. This results in a significant reduction in the request size (to 72 KB) at the cost of a slight increase in computation. But the response size of SEALPIR is similar to XPIR, and still results in an overall communication overhead of around $2,500$x under the previous example.

The large response size of XPIR and SEALPIR is due to how they handle the multiplications in the second (and higher) dimension. Instead of performing a regular ciphertext-ciphertext multiplication, they re-interpret one of the two ciphertexts as multiple plaintexts and then multiply each of such plaintext with the other ciphertext using ciphertext-plaintext multiplication. The client will need all the resulting ciphertexts to recover the result.

Ali *et al.* [3] improve upon SEALPIR's response size to achieve a total communication overhead of around 982x using the same example. Their key technique is to use ciphertext-ciphertext multiplication directly in the second and higher dimensions, followed by a modulus switching step to reduce the response size. This strategy results in higher noise growth and forces their protocol to adopt less efficient RLWE parameters. This in turn increases the computation cost.

The next breakthrough comes from a new type of homomorphic multiplication that composes RLWE ciphertexts with RGSW ciphertexts. This new multiplication, first introduced by Chillotti *et al.* [39], only adds an additive (rather than multiplicative) amount of noise after each operation. But using this new multiplication for PIR requires some extra care. Its low noise growth directly improves communication since we no longer need ciphertext splitting or large RLWE parameters. But this low-noise multiplication is even more expensive in terms of computation than RLWE ciphertext-ciphertext multiplications. Thus, a straightforward design using this low-noise multiplication will improve communication but severely worsen computation [40].

ONIONPIR avoids this computation bottleneck by adding base decomposition and sticking with RLWE ciphertext-plaintext multiplication in the first dimension. Their scheme achieves a 288x communication overhead for the aforementioned scenario of retrieving one entry from a database of one million entries each of 256 Bytes. SPIRAL adds modulus switching and matrix RRLWE encryption to ONIONPIR, and further reduces the communication overhead to about 128x in the above concrete scenario.

Batch Private Information Retrieval (BatchPIR). Ishai *et al.* [17] proposed the first BatchPIR scheme (named as Amortized PIR) using batch codes. In their scheme to retrieve a batch of size b, the server computation is proportional to $O(N(3/2)^{\log b})$ and the protocol communication is proportional to $O(3^{\log b})$. As the performance of their scheme depends on the size of the batch, the scheme is getting highly inefficient even for a small batch size of 32.

The state-of-the-art BatchPIR scheme called Multi-query PIR is by Angel *et al.* [12]. We have reviewed it in detail in Section II-E, so we do not repeat it here. If the client wants to retrieve a batch of 256 entries from a database with one million 256-Byte entries, its per-query computation is just 80 milliseconds but its communication overhead is still $1,872$x.

Orthogonal directions to improve PIR. The Stateful PIR approach improves the computation when the client has many entries to retrieve over time. The high-level idea is that the client retrieves some helper data (state) in the offline phase and uses it to make cheaper queries in the online phase.

Patel *et al.* [41] introduced the first stateful PIR construction. In the online phase, the server performs a linear number of cheaper symmetric-key operations and a sublinear number of expensive RLWE homomorphic operations. Corrigan-Gibbs and Kogan proposed a stateful PIR scheme [42] in which the server performs only a sublinear amount of computation. Their work does not provide an implementation or performance evaluation.

The offline phase of both of these protocols involves the client downloading subset sums of database entries. Mughees *et al.* [16] gives a construction for this problem based on batch PIR and copy networks. It is only efficient when each database entry is big (around 30 KB). Hence, concretely efficient construction for small entry sizes remains open.

Two independent works Henzinger *et al.* [43] and Davidson *et al.* [44] proposed stateful PIR based on the Learning with errors (LWE) assumption. Their key observation is that in LWE, the bulk of server computation is independent of the client query and can be performed in the offline phase. The downside of this scheme is that the client needs to download a large state offline.

VIII. Conclusion

In this paper, we have proposed the first BatchPIR protocol that is efficient in both computation and communication. Our protocol is based on vectorized homomorphic encryption and is especially suitable for applications with small entry sizes. The response overhead of our scheme is 7.2∼75x less than the previous BatchPIR scheme.

Acknowledgements This work is funded in part by a gift from Google.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *36th Annual Symposium on Foundations of Computer Science*, 1995, pp. 41–50.

[2] F. Zhao, Y. Hori, and K. Sakurai, "Two-servers PIR based DNS query scheme with privacy-preserving," in *International Conference on Intelligent Pervasive Computing (IPC)*, 2007.

[3] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, "Communication-computation trade-offs in PIR," in *30th USENIX Security Symposium*, 2021.

[4] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *30th USENIX Security Symposium*, 2021.

[5] S. Angel and S. T. V. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, K. Keeton and T. Roscoe, Eds., 2016.

[6] E. Kushilevitz and R. Ostrovsky, "Replication is NOT needed: SINGLE database, computationally-private information retrieval," in *38th Annual Symposium on Foundations of Computer Science, FOCS*, 1997, pp. 364–373.

[7] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *Advances in Cryptology - EUROCRYPT, International Conference on the Theory and Application of Cryptographic Techniques*, 1999, pp. 402–414.

[8] Y. Chang, "Single database private information retrieval with logarithmic communication," in *Information Security and Privacy: 9th Australasian Conference, ACISP*, 2004.

[9] C. Gentry and Z. Ramzan, "Single-database private information retrieval with constant communication rate," in *Automata, Languages and Programming, 32nd International Colloquium, ICALP*, 2005, pp. 803–815.

[10] H. Lipmaa, "An oblivious transfer protocol with log-squared communication," in *Information Security, 8th International Conference, ISC*, vol. 3650. Springer, 2005, pp. 314–328.

[11] C. A. Melchor, J. Barrier, L. Fousse, and M. Killijian, "XPIR: Private information retrieval for everyone," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, pp. 155–174, 2016.

[12] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *2018 IEEE Symposium on Security and Privacy*. San Francisco, California, USA: IEEE Computer Society, 2018, pp. 962–979.

[13] J. Park and M. Tibouchi, "SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval," in *Computer Security - ESORICS 2020 - 25th European Symposium on Research in Computer Security*. Guildford, UK: Springer, 2020, pp. 86–106.

[14] M. H. Mughees, G. Pestana, A. Davidson, and B. Livshits, "Privatefetch: Scalable catalog delivery in privacy-preserving advertising," *CoRR*, vol. abs/2109.08189, 2021.

[15] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server pir via fhe composition," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[16] M. H. Mughees, H. Chen, and L. Ren, "OnionPIR: Response efficient single-server PIR," in *CCS '21: ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea*, 2021.

[17] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004.* ACM, 2004, pp. 262–271.

[18] R. Henry, "Polynomial batch codes for efficient IT-PIR," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 4, pp. 202–218, 2016.

[19] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *28th USENIX Security Symposium, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds., 2019.

[20] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005.* ACM, 2005, pp. 84–93.

[21] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2012, p. 144, 2012.

[22] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *ACM Transactions on Computation Theory*, 2014.

[23] J. H. Cheon, A. Kim, M. Kim, and Y. S. Song, "Homomorphic encryption for arithmetic of approximate numbers," in *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017*, 2017.

[24] S. Halevi and V. Shoup, "Algorithms in HElib," in *Annual Cryptology Conference*, 2014, pp. 554–571.

[25] N. P. Smart and F. Vercauteren, "Fully homomorphic simd operations," *Designs, codes and cryptography*, vol. 71, pp. 57–81, 2014.

[26] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, 2012.

[27] H. Chen, K. Laine, and R. Player, "Simple encrypted arithmetic library - SEAL v2.1," in *Financial Cryptography and Data Security – International Workshops.* Springer, 2017.

[28] S. Halevi, Y. Polyakov, and V. Shoup, "An improved RNS variant of the BFV homomorphic encryption scheme," in *Topics in Cryptology - CT-RSA - The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA*, 2019.

[29] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, 1970.

[30] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on OT extension," *ACM Transactions on Privacy and Security (TOPS)*, 2018.

[31] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[32] C. Peikert *et al.*, "A decade of lattice cryptography," *Foundations and trends in theoretical computer science*, vol. 10, pp. 283–424, 2016.

[33] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) lwe," *SIAM Journal on computing*, vol. 43, pp. 831–871, 2014.

[34] "Crypto++ library 8.7," Online: https://www.cryptopp.com/, Apr. 2022.

[35] K. Cong, R. C. Moreno, M. B. da Gama, W. Dai, I. Iliashenko, K. Laine, and M. Rosenberg, "Labeled psi from homomorphic encryption with reduced computation and communication," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1135–1150.

[36] "Mpir," Online: https://github.com/sga001/mpir, Apr. 2022.

[37] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC*, vol. 1992, 2001, pp. 119–136.

[38] R. Sion and B. Carbunar, "On the computational practicality of private information retrieval," in *Proceedings of the Network and Distributed Systems Security Symposium,San Diego, California, USA*, 2007.

[39] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds," in *Advances in Cryptology - ASIACRYPT-22nd International Conference on the Theory and Application of Cryptology and Information Security*, 2016.

[40] C. Gentry and S. Halevi, "Compressible FHE with applications to PIR," in *Theory of Cryptography - 17th International Conference, TCC 2019, Nuremberg, Germany, December 1-5, 2019, Proceedings, Part II*, 2019.

[41] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS*, 2018.

[42] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, May 30 - June 3, 2022, Proceedings, Part II*, 2022.

[43] A. Henzinger, M. M. Hong, H. Corrigan-Gibbs, S. Meiklejohn, and V. Vaikuntanathan, “One server for the price of two: Simple and fast single-server private information retrieval,” Cryptology ePrint Archive, Paper 2022/949, 2022.

[44] A. Davidson, G. Pestana, and S. Celi, “Frodopir: Simple, scalable, single-server private information retrieval,” Cryptology ePrint Archive, Paper 2022/981, 2022.