OnionPIRv2: Efficient Single-Server PIR

Yue Chen
yuec12@illinois.edu

Ling Ren
renling@illinois.edu

School of Computing and Data Science
University of Illinois at Urbana-Champaign

Abstract

This paper presents OnionPIRv2, an efficient implementation of OnionPIR incorporating standard orthogonal techniques and engineering improvements. OnionPIR is a single-server PIR scheme that improves response size and computation cost by utilizing recent advances in somewhat homomorphic encryption (SHE) and carefully composing two lattice-based SHE schemes to control the noise growth of SHE. OnionPIRv2 achieves 2.5x-3.6x response overhead for databases with moderately large entries (around 4 KB or above) and up to 1600 MB/s server computation throughput.

1 Introduction

Protecting user privacy is becoming a critical concern for cloud applications and service providers. *Private information retrieval* (PIR) is a fundamental and important cryptographic primitive that allows a user to retrieve an entry from a public database without revealing which entry is retrieved. In this paper, we will focus on single-server PIR.

In PIR, there are three well-established efficiency metrics: *request size* (data uploaded from client to server), *response size* (data downloaded from server to client), and *server computation*. Following recent work [19], we measure the server computation cost using *throughput*, which is defined as the plaintext database size in bytes divided by the server computation time. Another metric that has recently been pointed out [19] is *extra* server storage. Many recent PIR schemes require the server to store helper values (key materials) per client. [1]

SealPIR [3] is a milestone in single-server PIR. It has a reasonably small request size, but is still very expensive in response size and server computation. Its response overhead is at least 100 times larger than the plaintext entry being retrieved. Its server computation throughput is around 150 MB/s, that is, it needs about 20 seconds of server computation for a 3 GB database. (Server computation is roughly proportional to the database size.) Park and Tibouchi [22] and Ali et al. [2] presented techniques to reduce response overhead but had to worsen server computation even more.

OnionPIR. In 2021, Mughees et al. designed OnionPIR [20] to address the response bottleneck of single-server PIR, while keeping the server computation cost similar to that of SealPIR. The main technique is to control the noise growth by replacing regular RLWE ciphertext multiplication with a special homomorphic multiplication called *external product* [10], which composes two homomorphic encryption schemes for much superior noise control. However, directly applying external

[1] There are also recent works that require the client to store a large amount of hints or perform heavy computation. We do not use those techniques and do not compare with those schemes in this article.

products will not lead to a practical scheme. Several supporting techniques were invented, most notably a new query packing algorithm and nonuniform database dimensions.

The paradigm of OnionPIR has since been adopted in follow-up works such as Spiral [19], WhisPIR [13], Respire [8]. These subsequent works reported improved performance numbers over the original OnionPIR through a combination of algorithmic changes and engineering refinements. However, the original OnionPIR paper focused on presenting algorithmic ideas, and its open-source prototype had poor performance. This makes it difficult to evaluate the algorithmic changes proposed in subsequent works.

This article. This article presents an improved implementation of OnionPIR incorporating standard orthogonal techniques and our best engineering efforts. We call the implementation OnionPIRv2. OnionPIRv2 achieves 2.5x-3.6x response overhead and 1100-1600 MB/s server computation throughput, which are up to 40% reduction and 11x improvements over the original OnionPIR prototype, respectively. We believe OnionPIRv2 is the most all-around practical single-server PIR protocol to date.

For readers' convenience, we repeat the necessary background and OnionPIR protocol details for readers' convenience. The pseudocode in this article reflects additional techniques and has improved clarity.

2 Background and Preliminary

2.1 Somewhat Homomorphic Encryption

State-of-the-art single-server PIR schemes use lattice-based leveled Fully Homomorphic Encryption, also called *Somewhat Homomorphic Encryption* (SHE), whose security relies on the hardness of Learning With Errors (LWE) or its variant on the polynomial ring (RLWE). We will compose two SHE schemes: BFV [6, 14] and RGSW [16, 11]. As its name suggests, a SHE scheme supports a limited number of homomorphic addition and multiplication operations on the ciphertexts. All known SHE schemes use *noisy* ciphertexts. Homomorphic operations on the ciphertexts increase the noise level in the resulting ciphertext. After a certain number of operations, the noise would become too large, and the ciphertext could no longer be decrypted.

BFV encryption. The BFV scheme is defined over a fixed polynomial ring $R = \mathbb{Z}[x]/(x^n + 1)$. Here, n is the degree of the polynomial and is usually a power of two. The secret key s is a polynomial sampled from a distribution of "small" (e.g., with ternary coefficients) polynomials in R. Let q and t denote the coefficient modulus for the ciphertext and plaintext, respectively. A plaintext message m is a polynomial in $R \mod t$. Each ciphertext consists of two polynomials in $R \mod q$ and is given as $(c_0, c_1) = (-as + e + \Delta m, a)$. Here, a is sampled uniformly at random from $R \mod q$, m is the plaintext polynomial, e is a noise polynomial with coefficients sampled from a bounded *Gaussian* distribution, and $\Delta = \lfloor q/t \rfloor$ is a scaling factor to put the message at the most significant bits. When $\|e\|_\infty < \Delta/2$, a ciphertext can be decrypted by computing

$$\left\lfloor \frac{c_0 + c_1 s}{\Delta} \right\rceil \mod t.$$

RGSW encryption. The RGSW scheme is parameterized by two parameters: the base B and the gadget vector length l. The two parameters give a trade-off between efficiency and noise growth.

Table 1: Comparison of computation costs and noise growths of homomorphic operations. Typically, l is set to 5.

Operation	Cost	Noise Growth
BFV ciphertext addition	≈ 0	additive
BFV plaintext-ciphertext multiplication	2	multiplicative
BFV ciphertext-ciphertext multiplication	$4 + 2l$	multiplicative
External product	$4l$	additive (for PIR)

The RGSW scheme has a gadget matrix:

$$\mathbf{G} = \mathbf{I}_2 \vee g = \begin{bmatrix} g & 0 \\ 0 & g \end{bmatrix} \in R^{(2l \times 2)}$$

where the *gadget vector* $g^{(l \times 1)} = (B^{\log q / \log B - 1}, B^{\log q / \log B - 2}, \cdots, B^{\log q / \log B - l})$.

A RGSW encryption of a plaintext polynomial $m \in R$ is

$$\mathsf{C} = \mathbf{Z} + m \cdot \mathbf{G}$$

where each row of $\mathbf{Z} \in R^{(2l \times 2)}$ is a BFV encryption of 0. Notice that the message m is not multiplied by the scaling factor Δ.

2.2 Noise Growth and Computational Cost of Homomorphic Operations

As mentioned, each homomorphic operation in SHE increases the noise in the output ciphertext. Different operations result in drastically different computation costs (measured by the number of polynomial multiplications) and noise growths. These will significantly impact our design decisions. We discuss below the approximate noise growth and computation costs of different operations and summarize them in Table 1.

BFV **ciphertext addition.** This operation adds two BFV ciphertexts and outputs a BFV ciphertext encrypting the plaintext sum. This operation is very efficient (does not involve polynomial multiplication) and has very small noise growth.

BFV **plaintext-ciphertext multiplication.** This operations takes as input a plaintext polynomial m and a BFV ciphertext encrypting m'. The output is a BFV ciphertext encrypting the product $m \cdot m'$. The noise term is multiplied by the plaintext [14]. This operation requires two polynomial multiplications.

BFV **ciphertext-ciphertext multiplication.** This operations takes as input two BFV ciphertexts and outputs a BFV ciphertext encrypting the plaintext product. This operation increases the noise by t (the plaintext modulus). This operation also requires an expensive relinearization step that takes $4 + 2l$ polynomial multiplications, where l is typically the same as the decomposition factor l in RGSW.

It is important to note that BFV multiplications (of either type) make the noise multiply and hence blow up exponentially if performed several times in a row. This is the primary source of inefficiency in prior PIR schemes.

External product. The external product operation takes as input a BFV ciphertext d encrypting m_d and a RGSW ciphertext C encrypting m_C, with respect to the same secret key s. The output is a BFV ciphertext encrypting the plaintext product $m_d \cdot m_C$. The noise growth of external product is $O\left(B \cdot \mathsf{Err}(\mathsf{C}) + |m_{\mathsf{C}}| \cdot \mathsf{Err}(\mathsf{d})\right)$ [10] where $\mathsf{Err}(\cdot)$ denotes the variance of the noise term in a ciphertext. In the context of PIR, m_{C} is usually a single bit, making the noise growth $O\left(B \cdot \mathsf{Err}(\mathsf{C}) + \mathsf{Err}(\mathsf{d})\right)$, which is *additive*.

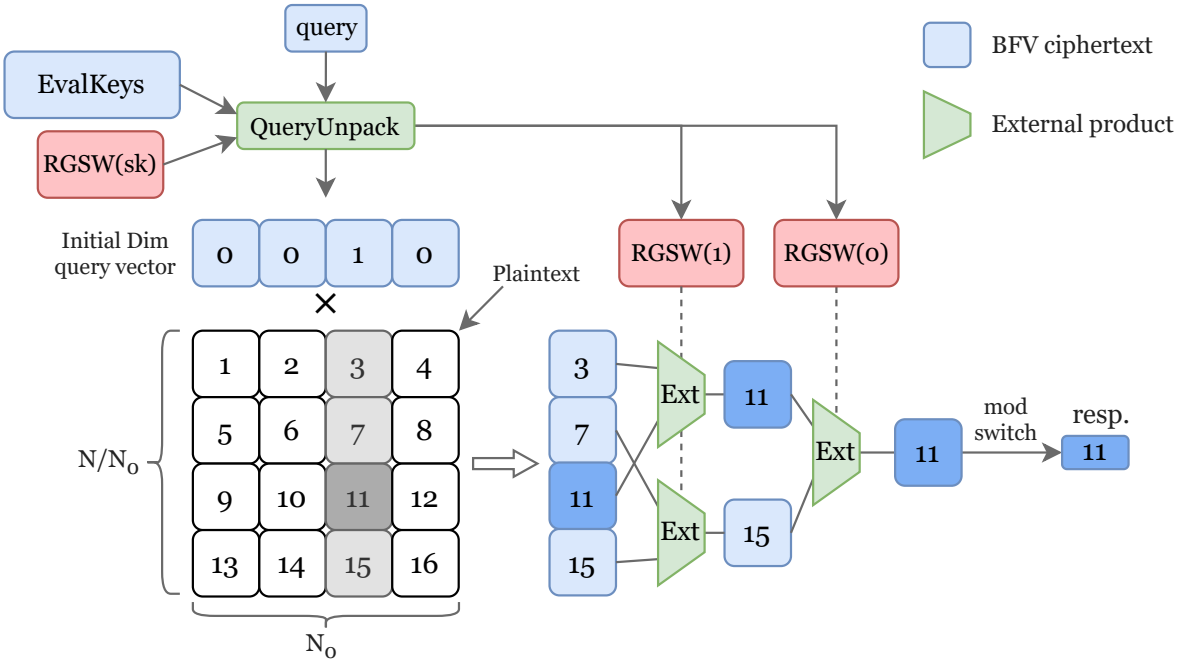

Figure 1: An illustration of the OnionPIR protocol. The QueryUnpack algorithm uses key materials stored on the server. The initial dimension uses plaintext-ciphertext multiplication. The remaining dimensions use external products. The initial dimension is larger, so that it dominates the computation cost.

2.3 Background on Single-Server PIR Protocols

The most basic SHE-based single-server PIR scheme is a simple dot product between the plaintext database and a one-hot query vector of ciphertexts sent by the client. The ciphertext corresponding to the target entry encrypts 1 and all the others encrypt 0. This basic scheme has a request size that is linear in the number of database entries. To reduce the request size, the database is represented as a multi-dimensional hypercube [12, 24]. To access a database entry, the client now sends d query vectors, each consisting of $\sqrt[d]{N}$ ciphertexts, where d is the number of dimensions.

SealPIR [3] proposes a technique called *query compression*, which allows the client to pack many encrypted bits within a single (BFV) ciphertext. The server can unpack this ciphertext into an array of ciphertexts, each encrypting a single bit. The unpacking process requires some client-specific key material, leading to a per-client extra storage on the server. With the help of another standard technique of pseudorandom seed [10] (cf. Section 10), SealPIR would achieve a request size of 32 KB for databases up to a few Gigabytes.

However, as mentioned earlier, SealPIR still suffers from large response overhead (100x) and server computation (150 MB/s). Both issues are related to the large noise increase from homomorphic multiplications. Anticipating a large noise increase, SealPIR had to use large ciphertexts to encrypt small plaintexts. This clearly hurts the response overhead. It also hurts server computation because each homomorphic operation performs very little actual work on the underlying plaintext.

3 The OnionPIRv2 Protocol

3.1 A Warm-up Protocol

We first describe a warm-up protocol that reduces the response size at the expense of higher computation. The idea is to use RGSW ciphertexts for client query vectors and use *external product* to evaluate the dot product in every dimension. Thanks to the additive noise growth of the external products in PIR, we can now afford to use more dimensions. In fact, one can set each dimension to 2 and use $\log n$ dimensions. This is essentially the protocol proposed in [22, 15].

In slightly more detail, the database with N entries is regarded as a $2 \times 2 \times \ldots \times 2$ hypercube of $\log N$ dimensions. The client sends RGSW to encrypt a selection bit for each dimension. The server uses *external products* to recursively and homomorphically select the correct half of the database at each dimension.

The warm-up protocol significantly reduces the response size over SealPIR due to the smaller noise growth of external products. Unfortunately, the warm-up protocol would incur much higher server computation than SealPIR. This is because the external product is more computationally intensive than plaintext-ciphertext multiplication (cf. Section 2), roughly by a factor of $2l = 10$.

3.2 OnionPIR: Optimizing the Computation

A key idea in OnionPIR is a simple trick to keep the server computation low: stay with BFV plaintext-ciphertext multiplication in the initial dimension and make the initial dimension larger than the remaining dimensions. Let N_i denote the size of the i-th dimension. We set the initial dimension size to $N_0 = 512$, and subsequent dimension sizes to $N_1 = N_2 = \ldots = 2$. This way, the total server computation cost is again dominated by the initial dimension, which uses BFV plaintext-ciphertext multiplication like prior works. To elaborate, the total number of polynomial multiplications across all dimensions is about $2N + 4l \cdot (\frac{N}{2N_0} + \frac{N}{4N_0} + \frac{N}{8N_0} + \cdots)$. With $N_0 = 512$, the $2N$ polynomial multiplications in the initial dimension dominate the computational cost.

Modulus choice. The choice of the ciphertext moduli has a big impact on computation. The total noise growth roughly dictates $\log q - \log t$. Thus, a large ciphertext q decreases the ciphertext expansion factor (i.e., $\log q / \log t$) and improves the server computation throughput. However, a larger q increases the request size and the key material size. Due to this trade-off, we recommend two sets of parameters and evaluate them in section 4.4.

3.3 Query Compression

Sending one ciphertext per query bit leads to a large request size. SealPIR introduced the query compression technique [3, 9] to reduce the query size. The high-level idea is that the client can pack many *independent* bits into a single ciphertext. The server then unpacks this ciphertext into individual ciphertexts, each encrypting a single bit. SealPIR's query compression is for BFV ciphertexts. We adapt the query compression algorithm for RGSW ciphertexts from the Onion Ring ORAM protocol [9].

Query packing. Algorithm 1 is the query packing algorithm in OnionPIR. For the initial dimension, we pack N_0 values where N_0 is the size of the initial dimension. For each subsequent dimension, we pack l values, corresponding to the first l rows of the single RGSW ciphertext for that dimension (see Section 3.4). With all subsequent dimensions having size 2, the number of dimensions $d = 1 + \lceil \log_2(N/N_0) \rceil$. This gives a total of $N_0 + l(d-1)$ values to pack. A BFV

Algorithm 1: QueryPack algorithm of OnionPIR

Input:

- $m_0 \in \{0, \ldots, N_0 - 1\}$, plaintext query index for initial dimension.
- $\{m_i\}_{i=1}^{d-1}$, plaintext query bits, one for each higher dimension.

Output: $\tilde{c}$, a single BFV ciphertext packing the query.

Notation:

- N_0, size of the initial dimension.
- d, the number of dimensions.
- l, the RGSW decomposition factor.
- w, smallest power of two no less than $N_0 + l(d-1)$.
- $1/w$, the inverse of w in the polynomial ring.
- $g = (B^{\log q/\log B - 1}, \cdots, B^{\log q/\log B - l})$, the RGSW gadget.
- $[\cdot]$ accessing an element of a vector.
- $\langle\cdot\rangle$, accessing a coefficient of a polynomial.
- $\mathsf{BitRev}(x)$, bit reversal of integer x in $(\log w)$-bit representation.

Create a zero plaintext pt
$\mathsf{pt}\langle\mathsf{BitRev}(m_0)\rangle = 1/w$
$\tilde{c} = (\tilde{c}_0, \tilde{c}_1) = \mathsf{BFV}(\mathsf{pt})$
$g' = g/w$
for $i = 1 : d-1$ **do**
  if $m_i == 1$ **then**
    for $k = 0 : l-1$ **do**
      $p = \mathsf{BitRev}(N_0 + l(i-1) + k)$
      $\tilde{c}_0\langle p\rangle = (\tilde{c}_0\langle p\rangle + g'[k]) \mod q$
    end
  end
end
Outputs $\tilde{c}$.

ciphertext in our implementation has $n = 2048$ or $n = 4096$ plaintext slots. With $N_0 = 512$ and $l = 5$, we can pack all the query bits into a *single* BFV ciphertext for all realistic databases.

Query unpacking. Algorithm 3 unpacks a single BFV query into individual ciphertexts. It first calls Algorithm 2 to extract the coefficients of the underlying polynomial of the packed query. A "substitution" operation $\mathsf{Subs}(c, j)$ maps $c = \mathsf{BFV}(\sum m_i x^i)$ to $\mathsf{BFV}(\sum m_i x^{ij})$ for any odd j (see Algorithms 8 of [9]). Setting $j = n + 1$ exploits the relation $x^{ij} = (x^n)^i \cdot x^i = (-1)^i \cdot x^i$ in the ring R. Then, we can use

$$c + \mathsf{Subs}(c, n+1) = \mathsf{BFV}(\sum 2m_{2i} \cdot x^{2i})$$
$$c + x^{-1}(c - \mathsf{Subs}(c, n+1)) = \mathsf{BFV}(\sum 2m_{2i+1} \cdot x^{2i})$$

to extract the coefficients at even and odd positions, respectively. By recursively extracting even and odd coefficients at each level, Algorithm 2 extracts all coefficients from the packed query into individual ciphertexts, each encrypting a constant.

Algorithm 2: $\mathsf{ExpandBFV}$ algorithm adapted from Algorithm 3 in [9].

Input: $\tilde{c}$ = Output of Algorithm 1

Output: $\mathsf{c}[i]$, $0 \leq i < u$, an array of BFV ciphertexts, each encrypting a constant polynomial corresponding to a value packed by Algorithm 1.

Notation: $u = N_0 + l(d-1)$, number of packed values. See Algorithm 1 for other notation.

$\mathsf{c}[1] = \tilde{c}$
for $i = 1$ **to** $w - 1$ **do**
  $k = 2^{\lfloor \log i \rfloor}$
  if $iw/k - w < u$ **then**
    $c' = \mathsf{Subs}(\mathsf{c}[i],\, w/k + 1)$
    $\mathsf{c}[2i] = \mathsf{c}[i] + c'$
    $\mathsf{c}[2i+1] = (\mathsf{c}[i] - c')\, x^{-k}$
  end
end
Outputs $\mathsf{c}[w, \ldots, w + u - 1]$

Algorithm 3: $\mathsf{QueryUnpack}$ algorithm, adapted from Algorithm 4 in [9].

Input: $(\tilde{c})$, a single BFV ciphertext packing the query; $A = \mathsf{RGSW}(s)$, RGSW encryption of the client's secret key, provided by client during initialization.

Output: $\left(\{\mathsf{C}_{\mathsf{BFV}}[j]\}_{j=0}^{N_0-1}, \{\mathsf{C}_{\mathsf{RGSW}}[i]\}_{i=1}^{d-1}\right)$, unpacked encrypted query vectors. $\mathsf{C}_{\mathsf{BFV}}$ is the vector of BFV ciphertexts for the first dimension and $\mathsf{C}_{\mathsf{RGSW}}[i]$ is the single RGSW ciphertext for the i-th dimension.

Notation: $\mathsf{C}_{\mathsf{RGSW}}[i]_k$ denotes the k^{th} row of the $\mathsf{C}_{\mathsf{RGSW}}[i]$ ciphertext.

$\mathsf{c} = \mathsf{ExpandBFV}(\tilde{c})$
$\mathsf{C}_{\mathsf{BFV}}[0, \ldots, N_0 - 1] = \mathsf{c}[0 \ldots, N_0 - 1]$
for $i = 1 : d - 1$ **do**
  $base = N_0 + l(i-1)$
  for $k = 0 : l - 1$ **do**
    $\mathsf{C}_{\mathsf{RGSW}}[i]_k = \mathsf{c}[base + k]$
    $\mathsf{C}_{\mathsf{RGSW}}[i]_{k+l} = \mathsf{ExternalProduct}(A, \mathsf{c}[base + k])$
  end
end
Outputs $\left(\{\mathsf{C}_{\mathsf{BFV}}[j]\}_{j=0}^{N_0-1}, \{\mathsf{C}_{\mathsf{RGSW}}[i]\}_{i=1}^{d-1}\right)$.

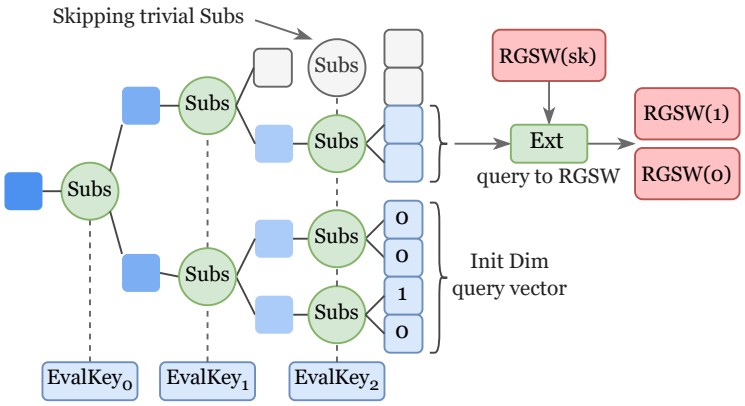

Figure 2: An illustration of Algorithm 3. One evaluation key is used for Subs in each level.

At this point, the first N_0 ciphertexts are used as the encrypted query vector for the initial dimension. For RGSW selection ciphertexts for subsequent dimensions, Algorithm 2 outputs only the top l rows of each RGSW ciphertext (because Algorithm 1 packed only those rows). Algorithm 3 then derives the bottom l rows by performing external products between the top rows and an RGSW encryption of the client secret key; see section 4.3 of [9] for a more detailed explanation.

More details in query packing. At a high level, query packing (Algorithm 1) adds different values to distinct coefficients of a BFV(0) ciphertext, and ExpandBFV (Algorithm 2) shifts each packed value to the constant term of its corresponding ciphertext. Concretely, packing a value is done by inserting the monomial μx^i to the first polynomial of BFV(0), producing $c = (\mu x^i, 0) + (-as + e, a)$. After running algorithm 2 on c, the resulting ciphertext at position BitRev(i) is exactly $(-a's + e' + 2^h\mu, a')$, where h is the height of the expansion tree, a' and e' are new random polynomial and noise term, respectively.

Pseudorandom component in BFV. Recall that a BFV ciphertext consists of two components, and one of them is sampled uniformly randomly from $R \mod q$. So, the client can generate it pseudorandomly from a short random seed and send the seed to the server. This trick is proposed in [10] and cuts the request size in half.

3.4 Standard Techniques Incorporated by OnionPIRv2

OnionPIRv2 incorporates many techniques overlooked in the original OnionPIR. Some are known techniques in the literature (this subsection), while others are new observations (next subsection).

Modulus Switching. *Modulus switching* [7, 2] is a standard technique to reduce the size of a RLWE ciphertext. It is often applied when no further computation needs to occur on the ciphertext. For our context, it is applied to the final response to the client. Modulus switching takes a ciphertext with modulus q and scales it to a ciphertext with a smaller modulus $q' < q$, while maintaining correct decryption with high probability. This is possible because the noise norm of the ciphertext after modulus switching is also rescaled to approximately q'/q of the original noise.

Modulus switching also supersedes a design in the original OnionPIR: the plaintext splitting in the initial dimension was aimed to reduce the response size, but it is less effective than modulus switching and increases computation.

Reducing the number of external products. Starting from the second dimension, we select one half of the database in each dimension. A simple and standard trick will allow us to use only one RGSW ciphertext for 1-out-of-2 selection in each higher dimension [10, 9, 22]. Let us think of the database as two halves x and y. Then, for a choice bit b, we can select the correct half by computing $b \cdot (y - x) + x$.

Delayed modular reduction. By default, homomorphic operations in mainstream FHE libraries apply the modular reduction after each homomorphic addition or multiplication. However, since we need to perform many homomorphic additions sequentially, it is more efficient to skip the modular reductions in between and only perform the modular reduction once at the very end.

Barrett reduction. In modular arithmetic, a naive way to compute $r = x \mod n$ is using $r = x - \lfloor x/n \rfloor \cdot n$. However, divisions are slow in practice. If we know the modulus n ahead of time, with some pre-computation, Barrett reduction replaces divisions with cheap bit-shifting operations. We refer readers to [4, 17] for the original ideas and detailed implementations.

Separate RGSW parameters for key material and query ciphertexts. Recall that a RGSW ciphertext has two parameters: l and B. A larger l leads to higher computation cost for external products but permits a smaller noise growth bound B. In OnionPIR, external products are used for both query unpacking and subsequent dimension data retrieval. Query unpacking is computed only once but contributes more to the overall noise growth. Menon and Wu [19] suggest using a smaller B for the key material to better control noise in the query unpacking step and a smaller l for query ciphertexts to speed up the dot products in the subsequent dimensions.

3.5 New Optimizations in OnionPIRv2

Tree trimming in query unpacking. Conceptually, the query unpacking process in Algorithm 2 has a tree structure. Each Subs call splits a node into two children, until reaching the leaf nodes that correspond to the $N_0 + l(d-1)$ BFV ciphertexts. In the original OnionPIR, the query unpacking algorithm always produces a perfect binary tree. This is wasteful when $N_0 + l(d-1)$ is not a power of 2. In that case, many leaf nodes are useless zero ciphertexts, and they are the results of splitting intermediate nodes that are also guaranteed to be zero ciphertexts. We skip the Subs calls at all those wasteful nodes (line 4 of Algorithm 2). To do so, we need to pack the values in a bit-reversed fashion (hence the use of $\mathsf{BitRev}(\cdot)$).

Pseudorandom components in key materials The pseudorandom component idea can also reduce the key material size by half. The trick directly applies to the key-switching keys used in the Subs automorphism in Algorithm 2. The $\mathsf{RGSW}(s)$ in Algorithm 3 requires extra care.

Recall that $\mathsf{RGSW}(s) = \mathbf{Z} + s \cdot \mathbf{G}$, where s is the encryption secret key itself, $\mathbf{Z}$ is $2l$ rows of $\mathsf{BFV}(0)$, and $\mathbf{G} = \mathbf{I}_2 \vee g$ is the gadget matrix. Let g_k denote the k^{th} gadget values, $k \in \{0, \ldots, l-1\}$. We first observe that the top l rows of $\mathsf{RGSW}(s)$ have the form $(-a \cdot s + e + s \cdot g_k, a)$. The second term can be easily generated pseudorandomly using a seed, so the server does not have to store it. The bottom l rows of $\mathsf{RGSW}(s)$ have the form $c = (-a \cdot s + e, a + s \cdot g_k)$. We can equivalently write it as $c' = (-(a - s \cdot g_k) \cdot s + e, a)$. The correctness can be verified by observing that c and c' decrypt to the same result, namely, $c_0 + c_1 \cdot s = c_0{}' + c_1{}' \cdot s$. Another way to interpret the equivalence is to think of $a - s \cdot g_k$ as a'. Since the polynomial a in each row is (pseudo-)random, applying a shift $-s \cdot g_k$ to a does not change its distribution. We can now use a single short seed to generate the second components of all $2l$ rows of $\mathsf{RGSW}(s)$. This cuts the server storage for $\mathsf{RGSW}(s)$ in half.

Algorithm 4: OnionPIR Protocol.

Input: DB server database of size N; idx, the index of the client's desired entry.

Notation:

- N, database size.
- $\mathsf{DB}[i]$, i-th entry.
- DB', intermediate database.
- All the notations defined in Algorithm 1 and 3.
- shaded part is executed by server.

Client converts idx into a vector $(i_1, \cdots i_d)$, where i_j is the position of idx entry in j-th dimension of the hypercube.

Client computes $\tilde{c} = \mathsf{QueryPack}((i_1, \cdots i_d))$, and sends $\tilde{c}$ to the server.

Server computes $(\mathsf{C_{BFV}}, \{\mathsf{C_{RGSW}}[i]\}_{i=1}^{d-1}) = \mathsf{QueryUnpack}(\tilde{c})$

for $j = 0 : N/N_0 - 1$ **do**

  $\mathsf{DB}'_j = \sum_{k=0}^{N_0-1} \mathsf{C_{BFV}}[k] \cdot \mathsf{DB}[k(N/N_0) + j]$

end

for $i = 1 : d - 1$ **do**

  $half = |\mathsf{DB}'|/2$

  for $j = 0 : half - 1$ **do**

    $\mathsf{DBdiff} = \mathsf{DB}'[half + j] - \mathsf{DB}'[j]$

    $\mathsf{DB}'[j] = \mathsf{ExternalProduct}(\mathsf{C_{RGSW}}[i], \mathsf{DBdiff}) + \mathsf{DB}'[j]$

  end

  $\mathsf{DB}' = \mathsf{DB}'[0, \ldots, half - 1]$

end

Server computes response $r = \mathsf{ModSwitch}(\mathsf{DB}')$ and sends r to the client.

Client decrypts r to get the content of entry idx.

Using standard matrix multiplication. The initial dimension computation can be viewed as a generalized matrix multiplication between the database, represented as a matrix of size $(N/N_0) \times N_0$, and the query BFV vector of size $N_0 \times 2$, where each element is a polynomial of degree n.

The polynomial multiplications are performed using Number Theoretic Transformation (NTT). After applying NTT to the plaintext database, each polynomial multiplication becomes an element-wise vector multiplication. Then, we can reinterpret the computation as n separate instances of standard matrix multiplication, each corresponding to one component after NTT. This way, each instance (which we call a *slice*) becomes a standard integer matrix multiplication between an integer matrix of size $(N/N_0) \times N_0$ and an integer matrix of size $N_0 \times 2$. The latter matrix is small enough to fit in cache, so the runtime of each slice is dominated by memory access time for the larger matrix. Ideally, the computation of the initial dimension can approach the bandwidth limit of reading the NTT-preprocessed database from the main memory.

3.6 OnionPIRv2 Full Protocol

The final OnionPIRv2 protocol is given in Algorithm 4. We have introduced all the components of the algorithm separately in previous sections. Below we describe the protocol putting together all the techniques.

The database is represented as a hypercube of d dimensions. The size of the initial dimension is $N_0 = 512$, and each of the remaining dimensions is of size 2. The total number of dimensions is thus $d = 1 + \lceil \log_2(N/N_0) \rceil$.

The client converts the desired index idx into d query vectors, one for each dimension of the hypercube. The client then packs all the query bits into a single BFV ciphertext using Algorithm 1 and sends the ciphertext to the server. The server unpacks this single ciphertext using Algorithm 3 into a vector of BFV ciphertexts for the initial dimension and a single RGSW ciphertext for each subsequent dimension.

For the initial dimension, the server performs a dot product between the BFV ciphertext vector and the plaintext database hypercube. The output is a BFV ciphertext hypercube of one fewer dimension. The server then continues to process higher dimensions similarly but uses external products. After the dot product at each dimension, the output is an intermediate hypercube of one fewer dimension and is used as the input to the next dot product. The output after the last dot product is a single BFV ciphertext encrypting the desired entry. The server applies modulus switching to this ciphertext and sends it to the client. The client decrypts it to obtain the desired database entry.

4 Implementation and Evaluation

4.1 Parameter Choices

The OnionPIR algorithm can use a wide range of RLWE parameter choices. We focus on the following two sets of parameters:

- Polynomial degree $n = 2048$, 60-bit ciphertext modulus q, 16-bit plaintext modulus t, 27-bit ciphertext modulus q' after modulus switching, $l = 10$ for query unpacking, and $l = 5$ for subsequent dimension.
- $n = 4096$, 120-bit q, 46-bit t, 57-bit q', $l = 8$ for query unpacking, and $l = 4$ for subsequent dimensions.

In both settings, we use $N_0 = 512$. (If the database is very small, one should use a smaller N_0.)

We use SEAL's default values for standard deviation error and secret key distribution. The LWE estimator by Albrecht et al. [1] suggests these parameters yield about 113 bits of computational security in both settings.

We use SEAL's noise budget estimator to select the plaintext modulus. Each parameter set leaves a noise budget of 1 to 3 bits after client decryption. We have tested these parameters in more than 5000 runs and all results are correct.

4.2 Implementation Details

We implemented OnionPIRv2 atop the SEAL Homomorphic Encryption Library version 4.1. Our implementation is available at `https://github.com/chenyue42/OnionPIRv2`.

In 2021, Intel released the HEXL library [5] for accelerating common computations used in homomorphic encryption schemes, including faster NTT using AVX512 instructions. The SEAL version we use has incorporated the faster NTT implementation. But SEAL only provides a BFV encryption scheme. So we implemented RGSW and external products using the SEAL library.

	OnionPIRv1	Spiral	KsPIR	OnionPIRv2	
DB Entry Size	30 KB	8 KB	8 KB	3.75 KB	22.5 KB
Server Storage	6.3 MB	15–17 MB	9–9.3 MB	0.63 MB	2.9 MB
DB Size	0.9 GB	1 GB	1 GB	0.9 GB	1.4 GB
Request Size	64 KB	14 KB	140 KB	16 KB	64 KB
Response Size	128 KB	20.5 KB	26 KB	13.5 KB	57 KB
Resp. Overhead	4.27×	2.56×	3.25×	3.6×	2.53×
Throughput	122 MB/s	247 MB/s	1251 MB/s	1109 MB/s	1098 MB/s
DB Size	7.5 GB	8 GB	8 GB	7.5 GB	11.3 GB
Request Size	64 KB	14 KB	140 KB	16 KB	64 KB
Response Size	128 KB	21 KB	26 KB	13.5 KB	57 KB
Resp. Overhead	4.27×	2.625×	3.25×	3.6×	2.53×
Throughput	149 MB/s	320 MB/s	1366 MB/s	1271 MB/s	1641 MB/s

Table 2: Performance comparison of OnionPIRv1, SpiralPIR, KsPIR, and OnionPIRv2. For each metric, we mark the best scheme in green, and schemes close to the best in lighter green.

4.3 Evaluation Setup

We test the performance of our OnionPIRv2 implementation on an AWS EC2 c5n.9xlarge instance with 96 GB RAM and Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz. Our implementation uses a single thread with AVX2 and AVX512 enabled. We use Ubuntu 22.04 and GCC 11.0.4 compiler for all our implementations.

We compare with OnionPIRv1 [20], SpiralPIR [19], and KsPIR [18] in Table 2. We evaluate all the schemes using two database sizes. We use each scheme's "native" (most preferred) entry size, which is why the database size in each experiment matches roughly but not exactly. We report the per-client server storage size, request size, response size (along with the multiplicative response overhead), and server computation throughput.

4.4 Evaluation Results

Communication cost. In OnionPIR, the request size is fixed for any practical database, which is $n \log q + |\mathsf{seed}|$ bits long, where $|\mathsf{seed}|$ is the size of a short pseudorandom seed. The size of a response ciphertext is $2n \log q'$, where q' is the smaller modulus after modulus switching. If the database exceeds the plaintext size, multiple response ciphertexts are needed. With the $n = 2048$ parameter choice, the request size is 16 KB and the response size is 13.5 KB, giving a response blowup of $13.5/3.75 = 3.6$. With the $n = 4096$ parameter choice, the request size is 64 KB and the response size is 57 KB, giving a response blowup of $57/22.5 \approx 2.53$.

Key material size. The key material size is 0.63 MB for the smaller parameter setting and 2.9 MB for the larger parameter setting. This can be stored on the server. Alternatively, it can be sent as part of the request if a system prefers to avoid per-client extra server storage.

Server Computation. For both variants of OnionPIRv2, the server mainly performs three tasks: (i) query unpacking, (ii) the initial-dimension dot products between the query vector and the database, and (iii) the multiplexer in the remaining dimensions. The time for query unpacking is logarithmic, and thus insensitive, to the database size.

The initial dimension is the server computation bottleneck. We have expressed the initial dimension as integer matrix multiplication (see Section 4.2). There are still two major factors affecting the initial dimension throughput. First, to use delayed modular reduction, we must use integer multiplications with 64-bit source operands but 128-bit destination operands to avoid integer overflow. With our best efforts, this slightly non-standard integer matrix multiplication achieves roughly 50% of peak memory bandwidth, which is roughly 12 GB/s in our machine. Second, after server preprocessing, the database is stored in NTT form and becomes larger by roughly a factor of $\log q / \log t$. As a result of these two factors, the throughput for our initial dimension is ≈ 1.6 GB/s when $\log q = 60$ and ≈ 2.3 GB/s when $\log q = 120$.

The remaining dimensions are slower for the 120-bit modulus q because the Residue Number System (RNS) is employed to represent large integers. Linear operations can be performed in the RNS form, but non-linear operations, such as NTT and gadget decomposition in external products, require transformation from RNS back to the standard integer representation. This is why the overall throughput of OnionPIRv2 is close to the throughput of the initial dimension when q is 60-bit, but there is a noticeable gap when q is 120-bit. Better engineering of RNS may speed up external products in both query unpacking and remaining dimensions and bring the overall throughput closer to that of the initial dimension for larger q.

5 Conclusion

In this paper, we have proposed OnionPIRv2, an efficient single-server PIR scheme with a response overhead of 2.5-3.6x of an insecure baseline while offering over 1600 MB/s throughput.

Despite all our algorithmic and engineering improvements, we believe there is room for further improvements. The bottleneck of OnionPIRv2 lies in the initial dimension, which we have expressed as standard matrix multiplication with integers. In theory, the performance of matrix multiplication could reach the memory bandwidth limit. Our implementation does not reach that limit. Our external product with a large modulus is also slow, likely due to inefficient RNS implementation. Another avenue is to use hardware acceleration. Recent works have demonstrated that GPU and FPGA can significantly speed up polynomial multiplications [23].

Acknowledgment. We thank Muhammad Haris Mughees for pointing us to the delayed modulus technique. We thank Zhikun Wang for suggesting the pseudorandom component in key materials. We thank Pranav Arunachalaramanan and Hao Chen for helpful discussions.

This work is funded in part by the National Science Foundation Award #2246386 and the Google Research Scholar Program. This work used AWS through the CloudBank project [21], which is supported by National Science Foundation grant #1925001.

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[2] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. *IACR Cryptol. ePrint Arch.*, 2019:1483, 2019.

[3] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979, San Francisco, California, USA, 2018. IEEE Computer Society.

[4] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In Andrew M. Odlyzko, editor, *Advances in Cryptology — CRYPTO' 86*, pages 311–323, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[5] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D.M. de Souza, and Vinodh Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '21, page 57–62, New York, NY, USA, 2021. Association for Computing Machinery.

[6] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual cryptology conference*, pages 868–886. Springer, 2012.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, 2011.

[8] Alexander Burton, Samir Jordan Menon, and David J Wu. Respire: High-rate pir for databases with small records. *Cryptology ePrint Archive*, 2024.

[9] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 345–360, London, UK, 2019. ACM.

[10] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika 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*, pages 3–33, Hanoi, Vietnam, 2016. eprint.

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[12] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[13] Leo de Castro, Kevin Lewi, and Edward Suh. Whispir: Stateless private information retrieval with low communication. *Cryptology ePrint Archive*, 2024.

[14] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012.

[15] Craig Gentry and Shai Halevi. Compressible fhe with applications to pir. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography*, pages 438–464, Cham, 2019. Springer International Publishing.

[16] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference*, pages 75–92, Santa Barbara, CA, USA, 2013. Springer.

[17] Rémi Géraud, Diana MaimuŢ, and David Naccache. *Double-Speed Barrett Moduli*, pages 148–158. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[18] Ming Luo, Feng-Hao Liu, and Han Wang. Faster fhe-based single-server private information retrieval. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 1405–1419, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022.

[20] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 2292–2306, 2021.

[21] Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, et al. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*, pages 1–4. 2021.

[22] Jeongeun Park and Mehdi 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*, pages 86–106, Guildford, UK, 2020. Springer.

[23] M. Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. HEAX: an architecture for computing on encrypted data. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems*, pages 1295–1309, Lausanne, Switzerland, 2020. ACM.

[24] Julien P. Stern. A new efficient all-or-nothing disclosure of secrets protocol. In *Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and Applications of Cryptology and Information Security*, volume 1514 of *Lecture Notes in Computer Science*, pages 357–371, Beijing, China, 1998. Springer.