Atom: Single-Server Private Information Retrieval with Low Communication and Fast Computation

Baoyu Li[1,2], Binwu Xiang[3], Kang Yang[2], Yu Yu[1,3], and Xiaogang Zhou[4]

[1] Shanghai Jiao Tong University
[2] State Key Laboratory of Cryptology
[3] East China Normal University
[4] China Telecom Quantum Information Technology Group Co., Ltd.

Abstract. Private information retrieval (PIR) enables a client to retrieve a record without revealing the index. Among existing PIR protocols with database-independent preprocessing, for each query, the protocols with low communication often take from several seconds to tens of seconds, while the faster protocols require hundreds of kilobytes for communication.

In this paper, we propose three techniques for different-type ciphertext conversions: (1) the first one is to generate a two-orbit SIMD selector from encrypted bits; (2) the second one is to convert a packed RLWE ciphertext into an aligned monomial RGSW ciphertext; (3) the third one is to produce an arbitrary monomial RGSW ciphertext from encrypted bits. Building on these techniques, we design a new PIR protocol (called Atom), achieving the best of both worlds (i.e., having not only low communication but also fast computation). We implemented Atom and evaluated its performance for 256 B records and databases from 256 MB to 8 GB. Specifically, Atom takes $3.0 \sim 3.8$ KB of online communication (i.e., the total communication, excluding the setup phase that can be run only once and reused for multiple queries), and takes $0.4 \sim 5.0$ seconds per query. Compared to the state-of-the-art KsPIR (CCS'24), Atom reduces the online communication cost by a factor of $40.5\times \sim 51.3\times$, while its running time is comparable to KsPIR ($0.2 \sim 5.2$ seconds per query).

1 Introduction

Private information retrieval (PIR) enables a client to retrieve a record from a database hosted by an untrusted server without revealing the queried index. Since its introduction by Chor et al. [15], PIR has become a fundamental privacy primitive underlying applications such as contact tracing, private web search, and anonymous communication [28,1,25,30,52,48].

PIR protocols broadly follow either a multi-server or a single-server model. Multi-server PIR achieves privacy by assuming that multiple database replicas do not collude, an assumption that can be difficult to enforce in practice.

Single-server PIR removes this assumption, but generally incurs higher communication or server-side computation [4]. Existing single-server constructions are primarily based on either coding-theoretic techniques [20] or homomorphic encryption (HE). We focus on HE-based PIR, which has enabled particularly compact communication and has driven a long line of practical constructions [29,3,49,42,26,33,12,37,16,17,44,40,34].

A central design choice in modern FHE-based single-server PIR is how preprocessing is performed. According to the information on which the preprocessing depends, state-of-the-art protocols can be broadly classified into two paradigms: database-dependent (DB-dependent) PIR and secret-key-dependent (SK-dependent) PIR.

Category 1: DB-dependent PIR. DB-dependent PIR protocols use database-dependent preprocessing to reduce the online computation and achieve high query throughput [26,33,16,44,17,40]. For example, FrodoPIR [16] and SimplePIR [26] use interactive database-dependent hints generated in the setup phase, while Hintless [33], YPIR [44], and InsPIRe [40] perform silent preprocessing. However, this paradigm typically requires large hints or evaluation keys, and database updates may require recomputing the preprocessing material. Stateless variants such as Hintless [33], YPIR [44] and InsPIRe [40] reduce the update burden, but still incur online communication costs on the order of hundreds of kilobytes.

Category 2: SK-dependent PIR. These schemes typically involve pre-sharing some public parameters independent of the secret key during the setup phase, along with sending additional secret-key-dependent ciphertexts as hints to the server [42,12,37,34]. In the online phase, the client transmits compressed query ciphertexts, and the server reconstructs the full query using the pre-shared hints, thereby reducing online communication. The server evaluates the query and returns a compact response ciphertext to the client.

Spiral [42], Respire [12], and KsPIR [37] require only a few tens of kilobytes of communication, while VIA [34] reports communication of only a few kilobytes. Overall, compared with DB-dependent PIR schemes [33,26,44,40,16], these SK-dependent schemes reduce communication by factors ranging from hundreds to thousands. However, their server computation remains substantial. KsPIR achieves a favorable trade-off between server computation and communication.

1.1 Our Contributions

In this work, we present Atom on the two-dimensional evaluation framework of KsPIR and instantiate it in two variants: Atom-s, which minimizes communication, and Atom-f, which prioritizes server-side computation. Unless otherwise specified, Atom denotes Atom-s. Figure 1 compares their online performance with all evaluated SK-dependent and DB-dependent schemes on the largest database in our evaluation. At 8 GB, Atom-s lies on the joint Pareto frontier, combining low online communication with competitive online computation. Specifically, for the first dimension, we develop a two-orbit SIMD query-generation method that represents the query index using compact LWE cipher-

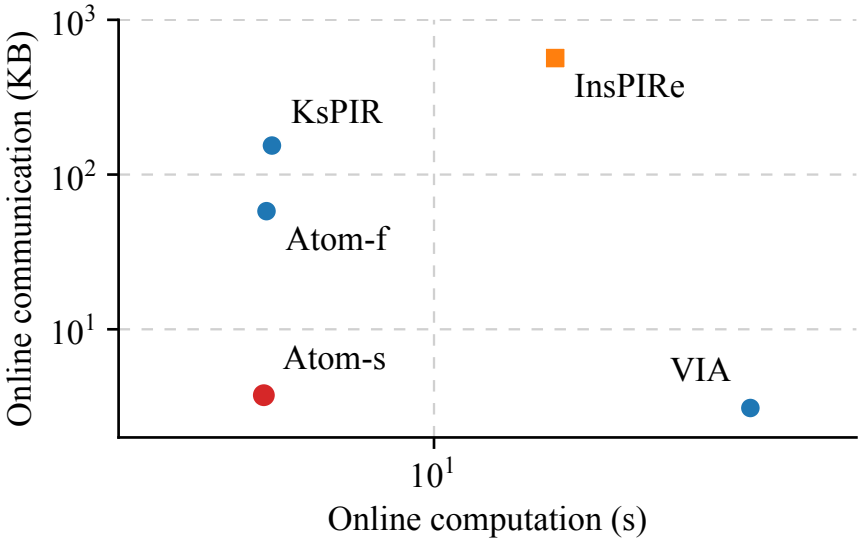


Fig. 1. Online performance comparison for an 8 GB database.

texts and expands them online into two-orbit SIMD RLWE query ciphertexts. This reduces the online query communication from $O(d)$ to $O(\log d)$ bits for ring dimension d. For the second dimension, the two variants adopt different query-generation strategies. In Atom-f, we develop a ciphertext-conversion technique that compresses an aligned monomial RGSW query, whose monomial position is restricted to predetermined block boundaries, into a single packed RLWE ciphertext, enabling fast server-side reconstruction. In contrast, Atom-s introduces BitDecompRGSW to remove this alignment restriction and reconstruct a monomial RGSW query at an arbitrary position from only $O(\log d)$ extracted coefficient ciphertexts. This further reduces query communication at the cost of additional server-side computation.

We implement both variants and evaluate them on databases ranging from 256 MB to 8 GB with 256-byte records. Compared with KsPIR [37], Atom and Atom-f reduce online communication by 40.5×–51.3× and 2.7×, respectively, while maintaining comparable online computation. Against InsPIRe [40], Atom achieves a 1.67×–3.24× speedup and reduces online communication by 65.54×–150.93×. Compared with VIA [34], Atom is 4.94×–7.13× faster while achieving comparable online communication. Figure 1 compares the online performance of all evaluated SK-dependent and DB-dependent schemes on the 8 GB database. Atom-s lies on the joint Pareto frontier, achieving low online communication while maintaining competitive online computation. Our source code will be made publicly available upon acceptance.

1.2 Technical Overview

Before presenting our techniques, we first recall the two-dimensional construction of KsPIR [37]. Let (u, w) denote the row and column indices, respectively, of the requested record in the encoded database.

For the first dimension, KsPIR encodes the row index u as a two-orbit SIMD selection vector and sends $\mathsf{Q}_1 = \mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$, where $\mathbf{e}_u^{\pm}$ denotes the SIMD vector whose u-th slot is selected in both automorphism orbits. The server applies $\mathsf{BSGSMult}$, which uses $O(\sqrt{d})$ rotations. The result is

$$\mathsf{C}_u = \mathsf{RLWE}(\Delta M_u(X), s), \qquad M_u(X) = \sum_j \mathsf{db}[u, j] X^j,$$

where the coefficients of $M_u(X)$ contain the records selected by the first-dimensional index u.

For the second dimension, KsPIR encodes the column index w as a monomial and sends $\mathsf{Q}_2 = \mathsf{RGSW}^{\mathsf{gadget}}(X^{-w}, s)$. The server performs one external product

$$\mathsf{Q}_2 \boxtimes_{\mathsf{gadget}} \mathsf{C}_u = \mathsf{RLWE}(\Delta X^{-w} M_u(X), s),$$

which moves the requested record to the constant coefficient, i.e., $[X^0](X^{-w} M_u(X)) = \mathsf{db}[u, w]$.

Although this two-dimensional evaluation is computationally efficient, its query is expensive to transmit. Let $t = \lceil \log_B q \rceil$ denote the gadget length. As defined in Section 2, an $\mathsf{RGSW}^{\mathsf{gadget}}$ ciphertext consists of two gadget vectors of RLWE ciphertexts,

$$\mathsf{RGSW}^{\mathsf{gadget}}(m, s) = \big(\mathsf{RLWE}'^{\,\mathsf{gadget}}(sm, s), \mathsf{RLWE}'^{\,\mathsf{gadget}}(m, s)\big),$$

and each $\mathsf{RLWE}'^{\,\mathsf{gadget}}$ contains t RLWE ciphertexts. Hence, Q_2 contains $2t$ RLWE ciphertexts, while Q_1 contributes one additional RLWE ciphertext. The complete KsPIR query therefore consists of $2t + 1$ RLWE ciphertexts, making online communication the main bottleneck of this otherwise efficient evaluation framework.

Our Solution

Two-orbit SIMD query generation. We first compress the query for the first dimension. Write the first-dimensional index as $u = \sum_i u_i 2^i$, where $u_i \in \{0, 1\}$. Instead of directly sending the full two-orbit SIMD selector $\mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$, the client sends compact LWE encryptions of the index bits $\{u_i\}$. Using the setup material, the server converts these LWE ciphertexts online into $\{\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s)\}$, which are then used to reconstruct the required SIMD selector.

Specifically, the server initializes an RLWE accumulator $acc = \mathsf{RLWE}(\Delta \mathbf{e}_0^{\pm}, s)$. Here, acc encrypts the current two-orbit SIMD selection vector, initially selecting position zero in both automorphism orbits. For each encrypted bit u_i, the server computes the candidate selector $acc' = \mathsf{Rotate}(acc, 2^i)$, and homomorphically selects between the current and rotated selectors:

$$acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}\big(\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s), acc, acc'\big).$$

Thus, u_i determines whether the current selector is shifted by 2^i positions. Let $\ell = d/2$ denote the number of slots in each SIMD orbit. Since the same automorphism rotates both orbits simultaneously, processing the $\log_2 \ell$ index bits yields $acc = \mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$, which is exactly the first-dimensional query required by KsPIR. We denote this procedure by $\mathsf{OneHotGen}$. It replaces one full RLWE query with only $O(\log d)$ LWE ciphertexts encrypting the index bits.

Aligned monomial RLWE-to-RGSW conversion. For the second dimension, we first consider the computation-oriented variant Atom-f. Recall that KsPIR requires a monomial $\mathsf{RGSW}^{\mathsf{gadget}}$ ciphertext, which contains $2t$ RLWE ciphertexts for gadget length $t = \lceil \log_B q \rceil$. Rather than transmitting these components separately, we pack their gadget-scaled plaintexts into a single RLWE ciphertext.

Let $k \in [t, 2t)$ be a power of two. For a monomial whose exponent is aligned to a block of width k, the client sends

$$ct = \mathsf{RLWE}\left(\sum_{j=0}^{t-1} B^j X^{uk+j}, s\right).$$

The server applies Expand for $\log_2 k$ rounds and extracts $\{\mathsf{RLWE}(B^j X^{uk}, s)\}_{j\in[t]}$, which form $\mathsf{RLWE}'^{\mathsf{gadget}}(X^{uk}, s)$. A subsequent $\mathsf{SchemeSwitch}$ reconstructs the complete $\mathsf{RGSW}^{\mathsf{gadget}}(X^{uk}, s)$ ciphertext.

We denote this conversion by $\mathsf{MonomialRGSW}$. It replaces a monomial RGSW query containing $2t$ RLWE ciphertexts with a single packed RLWE ciphertext, while requiring only $O(t)$ additional server-side computation. Its limitation is the *alignment restriction*: because the gadget components must occupy a power-of-two-sized block for partial expansion, the resulting monomial exponent must be a multiple of the block width k. This aligned conversion is used by Atom-f.

Bit-decomposed generation of arbitrary monomial RGSW ciphertexts. The communication-oriented variant Atom-s removes the above alignment restriction and further compresses the second-dimensional query. For an arbitrary second-dimensional index u, the client sends compact LWE encryptions of its binary representation. The server first converts these LWE ciphertexts online into $\mathsf{RGSW}^{\mathsf{RNS}}$ encryptions of the corresponding index bits.

To generate a monomial at an arbitrary position, we decompose $u = u_1 k + u_2$, where $u_1 \in [d/k]$ identifies a power-of-two block and $u_2 \in [k]$ identifies a position within that block. The server uses these two parts at different stages of the conversion.

The server first initializes an RLWE accumulator $acc = \mathsf{RLWE}\left(\sum_{j=0}^{L-1} g_{\mathsf{RNS},j} X^j, s\right)$. Here, acc encrypts the packed RNS gadget vector placed in the first coefficient block. Throughout the first stage, acc represents the same packed gadget vector, while its block position is determined homomorphically by the processed bits of u_1.

For each encrypted bit $(u_1)_i$, the server computes a candidate that moves the entire gadget block by $2^i k$ coefficients: $acc' = acc \cdot X^{-2^i k}$. It then uses the

reconstructed bit ciphertext to select whether this block shift is applied:

$$acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}\big(\mathsf{RGSW}^{\mathsf{RNS}}((u_1)_i, s), acc, acc'\big).$$

After all high-order bits are processed, acc encrypts the packed gadget vector in the block selected by u_1.

The server then applies partial Expand to separate the gadget components, obtaining $\big\{\mathsf{RLWE}(g_{\mathsf{RNS},j}X^{-u_1 k}, s)\big\}_{j\in[L]}$. The encrypted bits of u_2 are subsequently used to conditionally shift each expanded component inside the selected block. After processing all low-order bits, the server obtains $\{\mathsf{RLWE}(g_{\mathsf{RNS},j}X^{-u}, s)\}_{j\in[L]}$. These ciphertexts form $\mathsf{RLWE}'^{\mathsf{RNS}}(X^{-u}, s)$, and $\mathsf{SchemeSwitch}$ completes the desired $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-u}, s)$.

We denote this procedure by $\mathsf{BitDecompRGSW}$. Its key idea is to select the target *block* using u_1 before expansion and the *within-block position* using u_2 after expansion. This removes the k-alignment restriction of $\mathsf{MonomialRGSW}$ and allows the server to generate a monomial RGSW ciphertext at an arbitrary position. More importantly, the client represents the second-dimensional query using only $O(\log d)$ LWE ciphertexts encrypting the index bits, rather than one packed RLWE ciphertext or a full monomial RGSW ciphertext. The price is additional server-side CMUX computation.

Two communication–computation tradeoffs. The above techniques naturally give the two variants of Atom. The computation-oriented variant Atom-f retains the first-dimensional RLWE selector of KsPIR and compresses only the second-dimensional query. Specifically, the client sends one $\mathsf{RLWE}(\Delta\mathbf{e}_u^{\pm}, s)$ for the first dimension and one packed RLWE ciphertext for the second dimension. The latter is converted by $\mathsf{MonomialRGSW}$ into an aligned monomial RGSW ciphertext. Thus, Atom-f reduces the dominant RGSW communication while introducing relatively little online reconstruction.

In contrast, the communication-oriented variant Atom-s compresses both dimensions into encrypted index bits. For the first dimension, the server converts the uploaded LWE ciphertexts into $\mathsf{RGSW}^{\mathsf{RNS}}$ bit ciphertexts and applies $\mathsf{OneHotGen}$ to reconstruct the two-orbit SIMD RLWE query. For the second dimension, it similarly reconstructs encrypted index bits and applies $\mathsf{BitDecompRGSW}$ to generate a monomial RGSW ciphertext at an arbitrary position. Hence, Atom-s achieves substantially smaller online communication, at the cost of additional server-side query reconstruction. Once the two query ciphertexts are reconstructed, both variants retain the original $\mathsf{BSGSMult}$ and external-product evaluation path of KsPIR.

2 Preliminaries

We denote by λ the security parameter. For an integer n, we write $[n] = \{0, \ldots, n-1\}$. We use lowercase bold letters to denote vectors (e.g., $\mathbf{a}$) and write $\mathbf{a}_i$ to denote the i-th element in $\mathbf{a}$. Similarly, we use uppercase bold letters to denote matrices (e.g., $\mathbf{A}$) and write $\mathbf{A}_{i,j}$ to denote the element in row i

and column j. For an integer x, $|x|$ denotes its bit length. We use the notation $[a]_q = a \bmod q$.

2.1 Homomorphic Encryption

This subsection fixes the lattice notation and homomorphic operations used below. We use q for the ciphertext modulus and p for the plaintext modulus.

Definition 1 (Learning with Errors [50]). *Let $n = n(\lambda)$ be the dimension, $q = q(\lambda)$ be a modulus, and $\mathcal{X} = \mathcal{X}(\lambda)$ denote a discrete Gaussian distribution over $\mathbb{Z}_q$. The LWE assumption states that the following two distributions are computationally indistinguishable:*

$$\{(\mathbf{a}_i, \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i)\}_{i \in [n]} \approx_c \{(\mathbf{a}_i, u_i)\}_{i \in [n]}$$

where $\mathbf{a}_i \leftarrow \mathbb{Z}_q^n$, $\mathbf{s} \leftarrow \mathcal{X}^n$, $e_i \leftarrow \mathcal{X}$, and $u_i \leftarrow \mathbb{Z}_q$.

Definition 2 (Ring Learning with Errors [39]). *Let $d = d(\lambda)$ be the polynomial degree and a power of two. For a modulus q, define the ring $R_q = \mathbb{Z}_q[X]/(X^d + 1)$. Let $\mathcal{X} = \mathcal{X}(\lambda)$ denote a discrete Gaussian distribution over R_q. The RLWE assumption states that the following two distributions are computationally indistinguishable:*

$$\{(a_i, a_i \cdot s + e_i)\}_{i \in [n]} \approx_c \{(a_i, u_i)\}_{i \in [n]}$$

where $a_i \leftarrow R_q$, $s \leftarrow \mathcal{X}$, $e_i \leftarrow \mathcal{X}$, and $u_i \leftarrow R_q$.

Definition 3 (Gadget Decomposition [46]). *Let $q \in \mathbb{Z}$ be a modulus and $B \in \mathbb{Z}$ a gadget decomposition base. Define $t = \lceil \log_B q \rceil$ as the gadget length. When needed, we assume t is a power of two. The* gadget vector *is*

$$\mathbf{g}_{\mathsf{gadget}} = (1, B, B^2, \ldots, B^{t-1}) \in \mathbb{Z}_q^t,$$

whose i-th component is $g_{\mathsf{gadget},i} = B^i$. For $f \in R_q$, its gadget decomposition *is*

$$\mathsf{Decomp}_{\mathsf{gadget}}(f) = \left(\left[\left\lfloor \frac{f}{B^i} \right\rceil \right]_B \right)_{i=0}^{t-1} \in R_B^t,$$

which satisfies $\langle \mathbf{g}_{\mathsf{gadget}}, \mathsf{Decomp}_{\mathsf{gadget}}(f) \rangle = [f]_q$.

Definition 4 (RNS Gadget Decomposition [23]). *Let $q = \prod_{i=0}^{L-1} q_i$, where the moduli q_i are pairwise coprime. For each i, define $\widehat{q}_i = q/q_i$, $g_{\mathsf{RNS},i} = \widehat{q}_i [\widehat{q}_i^{-1}]_{q_i} \bmod q$. The RNS gadget vector is $\mathbf{g}_{\mathsf{RNS}} = (g_{\mathsf{RNS},0}, \ldots, g_{\mathsf{RNS},L-1})$. For $f \in R_q$, define its coefficient-wise RNS decomposition as*

$$\mathsf{Decomp}_{\mathsf{RNS}}(f) = \big([f]_{q_0}, \ldots, [f]_{q_{L-1}}\big).$$

By the Chinese remainder theorem,

$$\langle \mathbf{g}_{\mathsf{RNS}}, \mathsf{Decomp}_{\mathsf{RNS}}(f) \rangle = [f]_q.$$

Ciphertext Structure We adopt three ciphertext types RLWE, RLWE', and RGSW based on the FHE scheme [9,19,22,18,14]. Unless otherwise specified, all ciphertexts are encrypted under a secret key s. For a decomposition basis $\beta \in \{\mathsf{gadget}, \mathsf{RNS}\}$, let $\mathbf{g}_\beta$ denote its gadget vector and let ℓ_β denote its length, where $\ell_{\mathsf{gadget}} = t$ and $\ell_{\mathsf{RNS}} = L$.

- $\mathsf{LWE}(m, \mathbf{s})$: Sample $\mathbf{a} \leftarrow \mathbb{Z}_q^n$, $e \leftarrow \mathcal{X}$, and output $ct = (\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e + m)$.
- $\mathsf{RLWE}(m, s)$: Sample $c_0 \leftarrow R_q$, $e \leftarrow \mathcal{X}$, and output $ct = (c_0, c_0 \cdot s + e + m) = (c_0, c_1)$.
- $\mathsf{RLWE}'^{\beta}(m, s)$: Output $\mathsf{RLWE}'^{\beta}(m, s) = \{\mathsf{RLWE}(g_{\beta,i} \cdot m, s)\}_{i=0}^{\ell_\beta - 1}$.
- $\mathsf{RGSW}^{\beta}(m, s)$: Output $\mathsf{RGSW}^{\beta}(m, s) = \big(\mathsf{RLWE}'^{\beta}(s \cdot m, s), \mathsf{RLWE}'^{\beta}(m, s)\big)$.

We omit the superscript β when the decomposition basis is clear from context.

For a BFV plaintext $m \in R_p$, we write the scaling explicitly as $\mathsf{RLWE}(\Delta m, s)$, where $\Delta = \lfloor q/p \rfloor$. We define BFV decoding as follows.

- $\mathsf{Decrypt}(ct, s)$: Given $ct = (c_0, c_1) = \mathsf{RLWE}(\Delta m, s)$ and secret key s, compute $m = \lfloor \frac{c_1 - c_0 \cdot s}{\Delta} \rceil$.

Decryption is correct if the noise remains below $\Delta/2$.

SIMD Encoding We use the standard SIMD encoding [51] for RLWE-based homomorphic encryption. The 2ℓ SIMD slots split into two automorphism orbits of length ℓ. Let g generate the positive orbit $\mathcal{S}^+ = \{g^j : j \in [\ell]\}$, while $\mathcal{S}^- = \{-g^j : j \in [\ell]\}$ is the corresponding negative orbit. A vector of plaintext values is encoded as a polynomial $m \in R_p$, whose slots can be processed in parallel by homomorphic operations.

Basic Homomorphic Operations We outline standard homomorphic addition, plaintext-ciphertext multiplication, automorphism, and key switching [9,19,10] operations in Appendix A. For details, we refer the reader to [9,19,22,18,14].

Rotate Let g be the generator used above. For an integer i, $\mathsf{Rotate}(m, i) = m(X^{g^i})$ applies the slot rotation to a plaintext polynomial m [51]. For a ciphertext, we define $\mathsf{Rotate}(ct, i) := \mathsf{Auto}(ct, g^i)$.

External Product For $\beta \in \{\mathsf{gadget}, \mathsf{RNS}\}$, the external product [14], denoted by $\boxtimes_\beta$, maps $\mathsf{RGSW}^{\beta}(a, s)$ and $\mathsf{RLWE}(b, s)$ to $\mathsf{RLWE}(a \cdot b, s)$ using decomposition basis β.

CMUX Given $\mathsf{RGSW}^{\beta}(b, s)$ for $b \in \{0, 1\}$ and two RLWE ciphertexts ct_0, ct_1, we can homomorphically select ct_b by computing Eq. 1 [14].

$$\begin{aligned} ct_b &= \mathsf{CMUX}_\beta(\mathsf{RGSW}^{\beta}(b, s), ct_0, ct_1) \\ &= ct_0 + \mathsf{RGSW}^{\beta}(b, s) \boxtimes_\beta (ct_1 - ct_0) \end{aligned} \tag{1}$$

RingSwitch Given a large-ring ciphertext $ct \in \mathsf{RLWE}_{R_{d_1,q}}(\kappa(m), s_1)$ and a ring-switching key $\mathsf{ksk}_{\mathsf{RS}}$, the operation $\mathsf{RingSwitch}(ct, \mathsf{ksk}_{\mathsf{RS}})$ outputs a small-ring ciphertext $ct' \in \mathsf{RLWE}_{R_{d_2,q}}(m, s_2)$, where $d_2 < d_1$ [12]. We describe this process in Alg. 13 of Appendix A.5.

RevPackRlwes Let $m = 2^\ell$ and $T = \log_2(d/m)$. Given RLWE ciphertexts $\{ct_i\}_{i=0}^{m-1}$ whose designated coefficients encrypt $\{\mu_i\}_{i=0}^{m-1}$, $\mathsf{RevPackRlwes}(\{ct_i\}, T)$ outputs one ciphertext encrypting $\sum_{i=0}^{m-1} \mu_i X^{i2^T}$ [31]. We describe this process in Alg. 11 of Appendix A.3.

Expand Given an RLWE ciphertext ct and an integer T, $\mathsf{Expand}(ct, T)$ outputs 2^T RLWE ciphertexts encrypting the selected coefficient classes of the input plaintext [3]. We describe this process in Alg. 12 of Appendix A.4.

SchemeSwitch Given a set of RLWE ciphertexts forming $\mathsf{RLWE}'^{\beta}(m, s)$ and a conversion key csk, $\mathsf{SchemeSwitch}$ outputs $\mathsf{RGSW}^{\beta}(m, s)$ [47]. We describe this process in Alg. 14 of Appendix A.6.

BSGS Multiplication Given an encoded plaintext matrix $\mathsf{M} \in \mathbb{Z}_p^{2\ell \times \ell}$ and a ciphertext encrypting a vector $\mathsf{v} \in \mathbb{Z}_p^{\ell}$ in both SIMD orbits, $\mathsf{BSGSMult}$ outputs a ciphertext encrypting Mv. Equivalently, it evaluates the two $\ell \times \ell$ submatrices in parallel. The baby-step giant-step method reduces the number of homomorphic rotations from $O(\ell)$ to $O(\sqrt{\ell})$ [24,36]. We describe this process in Alg. 15 of Appendix A.7.

2.2 Private Information Retrieval

Definition 5 (Private Information Retrieval [15]). *Let λ be the security parameter and let DB be a database of N records. A single-server PIR protocol consists of the following algorithms.*

- $\mathsf{ParamSet}(1^\lambda)$: On input the security parameter λ, the $\mathsf{ParamSet}$ algorithm outputs public parameters pp and a secret key s.
- $\mathsf{Encode}(pp, \mathsf{DB})$: On input the public parameters pp and the database DB, the Encode algorithm outputs an encoded database db.
- $\mathsf{Query}(pp, idx, s)$: On input the public parameters pp, the query index idx, and the secret key s, the Query algorithm outputs a query qu.
- $\mathsf{Answer}(pp, qu, \mathsf{db})$: On input the public parameters pp, the query qu and the encoded database db, the Answer algorithm outputs a result ciphertext as response ct.
- $\mathsf{Recover}(ct, s)$: On input the response ct and the secret key s, the $\mathsf{Recover}$ algorithm outputs a record m.

Private information retrieval protocols satisfy the following properties:

1. **Correctness**. For every database DB and index $i \in [N]$, let $(pp, s) \leftarrow \mathsf{ParamSet}(1^\lambda)$, $\mathsf{db} \leftarrow \mathsf{Encode}(pp, \mathsf{DB})$, $qu \leftarrow \mathsf{Query}(pp, i, s)$, and $ct \leftarrow \mathsf{Answer}(pp, qu, \mathsf{db})$. Correctness requires

$$\Pr[\mathsf{Recover}(ct, s) = \mathsf{DB}[i]] \geq 1 - \mathrm{negl}(\lambda),$$

where the probability is over the randomness of the five algorithms above.

2. **Query Privacy**. A single-server PIR protocol satisfies query privacy if, for every database DB and any two indices $i, i' \in [N]$, the distributions $\big(pp, \mathsf{db}, \mathsf{Query}(pp, i, s)\big)$ and $\big(pp, \mathsf{db}, \mathsf{Query}(pp, i', s)\big)$ are computationally indistinguishable, where $(pp, s) \leftarrow \mathsf{ParamSet}(1^\lambda)$ and $\mathsf{db} \leftarrow \mathsf{Encode}(pp, \mathsf{DB})$.

3 Ciphertext Conversion

This section presents three ciphertext-conversion techniques used in Atom.

3.1 Two-Orbit SIMD One-Hot Generation

Recall that the SIMD slots form two length-ℓ automorphism orbits $\mathcal{S}^+ = \{g^j\}_{j\in[\ell]}$ and $\mathcal{S}^- = \{-g^j\}_{j\in[\ell]}$. For $u \in [\ell]$, let $\mathbf{e}_u^\pm$ select the u-th slot in both orbits. Our goal is to construct $\mathsf{RLWE}(\Delta\mathbf{e}_u^\pm, s)$ from encrypted bits of u.

Write $u = \sum_{i=0}^{r-1} u_i 2^i$, where $r = \log_2 \ell$ and $u_i \in \{0, 1\}$. We maintain an RLWE accumulator acc encrypting the current two-orbit selector, initialized as $acc \leftarrow (0, \Delta\cdot\mathsf{iNTT}(\mathbf{e}_0^\pm))$. For each encrypted bit $\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s)$, compute $acc' \leftarrow \mathsf{Rotate}(acc, 2^i)$ and update

$$acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}\big(\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s), acc, acc'\big).$$

The same automorphism rotates both SIMD orbits, so each bit simultaneously controls the corresponding shift in both orbits. After processing all bits, acc encrypts $\Delta\mathbf{e}_u^\pm$. The complete procedure is given in Algorithm 1.

Algorithm 1: OneHotGen

Input: $\{\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s)\}_{i\in[r]}$, where $r = \log_2 \ell$
Output: $\mathsf{RLWE}(\Delta\mathbf{e}_u^\pm, s)$

$acc \leftarrow (0, \Delta \cdot \mathsf{iNTT}(\mathbf{e}_0^\pm))$;
for $i = 0, \ldots, r-1$ **do**
  $acc' \leftarrow \mathsf{Rotate}(acc, 2^i)$;
  $acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}(\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s), acc, acc')$;
end
return acc;

Lemma 1. *Let $u = \sum_{i=0}^{r-1} u_i 2^i$, where $r = \log_2 \ell$ and $u_i \in \{0,1\}$. Algorithm 1 transforms $\{\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s)\}_{i\in[r]}$ into an RLWE ciphertext encrypting $\Delta \mathbf{e}_u^{\pm}$.*

Proof. For $i \in \{0, \ldots, r\}$, define $u^{(i)} = \sum_{j=0}^{i-1} u_j 2^j$. After processing the first i bits, the accumulator encrypts $\Delta \mathbf{e}^{\pm}_{u^{(i)}}$. The invariant holds initially for $u^{(0)} = 0$. At iteration i, $\mathsf{Rotate}(acc, 2^i)$ encrypts $\Delta \mathbf{e}^{\pm}_{u^{(i)}+2^i}$. By the definition of $\mathsf{CMUX}_{\mathsf{RNS}}$ in Eq. 1, the update selects the rotated ciphertext exactly when $u_i = 1$. Hence the new accumulator encrypts $\Delta \mathbf{e}^{\pm}_{u^{(i+1)}}$. Since $u^{(r)} = u$, the output encrypts $\Delta \mathbf{e}^{\pm}_u$.

Algorithm 1 uses $\log_2 \ell$ rotations and $\log_2 \ell$ CMUX operations.

3.2 Aligned Monomial RLWE-to-RGSW

We next compress a gadget-based monomial RGSW ciphertext into one packed RLWE ciphertext. Let $t = \lceil \log_B q \rceil$ and choose a power-of-two $k \in [t, 2t)$. For an integer u, consider

$$ct = \mathsf{RLWE}\left(\sum_{j=0}^{t-1} B^j X^{uk+j}, s\right).$$

The t gadget-scaled monomials occupy consecutive coefficients in the block starting at uk. Applying Expand for $\log_2 k$ rounds and keeping the first t outputs yields $\{\mathsf{RLWE}(B^j X^{uk}, s)\}_{j\in[t]}$, which form $\mathsf{RLWE}'^{\mathsf{gadget}}(X^{uk}, s)$. A subsequent $\mathsf{SchemeSwitch}$ produces $\mathsf{RGSW}^{\mathsf{gadget}}(X^{uk}, s)$. The complete procedure is given in Algorithm 2.

Algorithm 2: MonomialRGSW

Input: $ct = \mathsf{RLWE}(\sum_{j=0}^{t-1} B^j X^{uk+j}, s)$, csk, and k
Output: $\mathsf{RGSW}^{\mathsf{gadget}}(X^{uk}, s)$

$\{ct'_j\}_{j\in[k]} \leftarrow \mathsf{Expand}(ct, \log_2 k)$;
$\{ct_j\}_{j\in[t]} \leftarrow \{ct'_j\}_{j\in[t]}$;
return $\mathsf{SchemeSwitch}(\{ct_j\}_{j\in[t]}, \mathsf{csk})$;

Lemma 2. *For a power-of-two $k \in [t, 2t)$, Algorithm 2 maps $\mathsf{RLWE}\left(\sum_{j=0}^{t-1} B^j X^{uk+j}, s\right)$ to $\mathsf{RGSW}^{\mathsf{gadget}}(X^{uk}, s)$.*

Proof. Recall from Appendix A.4 that $\mathsf{Expand}(ct, r)$ separates the plaintext coefficients of an RLWE ciphertext into 2^r coefficient classes. Applying $\mathsf{Expand}(ct, \log_2 k)$ to the packed input therefore recovers $\{\mathsf{RLWE}(B^j X^{uk}, s)\}_{j\in[t]}$, which form $\mathsf{RLWE}'^{\mathsf{gadget}}(X^{uk}, s)$. By the $\mathsf{SchemeSwitch}$ transformation defined in Appendix A.6, this RLWE' ciphertext is converted into $\mathsf{RGSW}^{\mathsf{gadget}}(X^{uk}, s)$.

The conversion replaces the $2t$ RLWE ciphertexts of a standard monomial RGSW ciphertext with one packed RLWE ciphertext and incurs $O(t)$ additional computation. The power-of-two block structure restricts the output to monomials whose exponents are multiples of k.

3.3 Bit-Decomposed Monomial Conversion

We next remove the alignment restriction. For an arbitrary target exponent u, write $u = u_1 k + u_2$, where $u_1 \in [d/k]$ selects a block and $u_2 \in [k]$ selects a position within that block. Let $k = 2^h \geq L$, where L is the length of the RNS gadget vector.

Define $G_{\mathsf{RNS}}(X) = \sum_{j=0}^{L-1} g_{\mathsf{RNS},j} X^j$. The server initializes an RLWE accumulator $acc \leftarrow (0, G_{\mathsf{RNS}}(X))$, which trivially encrypts the packed RNS gadget vector in the first block.

The encrypted bits of u_1 select the target block before expansion. For each bit $(u_1)_i$, compute $acc' \leftarrow acc X^{-2^i k}$ and

$$acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}\big(\mathsf{RGSW}^{\mathsf{RNS}}((u_1)_i, s), acc, acc'\big).$$

After processing all bits of u_1, acc encrypts $X^{-u_1 k} G_{\mathsf{RNS}}(X)$.

A partial Expand then produces $\{ct_j = \mathsf{RLWE}(g_{\mathsf{RNS},j} X^{-u_1 k}, s)\}_{j\in[L]}$. The encrypted bits of u_2 subsequently select the intra-block offset. For each $(u_2)_i$, compute $ct'_j \leftarrow ct_j X^{-2^i}$ and

$$ct_j \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}\big(\mathsf{RGSW}^{\mathsf{RNS}}((u_2)_i, s), ct_j, ct'_j\big).$$

The resulting ciphertexts form $\mathsf{RLWE}'^{\mathsf{RNS}}(X^{-u}, s)$. Applying $\mathsf{SchemeSwitch}$ gives $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-u}, s)$. The complete procedure is given in Algorithm 3.

Lemma 3. *Let $u = u_1 k + u_2$, where $u_1 \in [d/k]$ and $u_2 \in [k]$. Given RNS-based RGSW encryptions of the bits of u_1 and u_2, Algorithm 3 outputs $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-u}, s)$.*

Proof. Write $u_1 = \sum_i a_i 2^i$. The first CMUX sequence conditionally applies the monomial shifts $X^{-2^i k}$ according to the encrypted bits a_i. By Eq. 1, the resulting accumulator therefore encrypts $X^{-u_1 k} G_{\mathsf{RNS}}(X)$, where $G_{\mathsf{RNS}}(X) = \sum_{j=0}^{L-1} g_{\mathsf{RNS},j} X^j$. Recall from Appendix A.4 that partial Expand separates the packed coefficient classes. Applying it to the accumulator yields $\{\mathsf{RLWE}(g_{\mathsf{RNS},j} X^{-u_1 k}, s)\}_{j\in[L]}$. Next write $u_2 = \sum_i b_i 2^i$. The second CMUX sequence conditionally applies the shifts X^{-2^i} to every expanded component according to the encrypted bits b_i. Consequently, the resulting ciphertexts are $\{\mathsf{RLWE}(g_{\mathsf{RNS},j} X^{-(u_1 k + u_2)}, s)\}_{j\in[L]}$, which form $\mathsf{RLWE}'^{\mathsf{RNS}}(X^{-u}, s)$. Finally, the $\mathsf{SchemeSwitch}$ transformation defined in Appendix A.6 converts this RLWE' representation into $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-u}, s)$.

The block-selection stage requires $O(\log(d/k))$ CMUX operations, partial expansion costs $O(k)$, and intra-block selection requires $O(L \log k)$ CMUX operations. The total additional computation is therefore $O(k + L \log k + \log(d/k))$.

The main distinction from $\mathsf{MonomialRGSW}$ is that $\mathsf{BitDecompRGSW}$ selects the block before expansion and the intra-block position afterward, thereby supporting arbitrary monomial exponents.

Algorithm 3: BitDecompRGSW

Input: RNS-based RGSW encryptions of the bits of u_1, u_2, and csk
Output: $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-u}, s)$

$acc \leftarrow (0, \sum_{j=0}^{L-1} g_{\mathsf{RNS},j} X^j)$;
for *each bit position* i *of* u_1 **do**
  $acc' \leftarrow acc X^{-2^i k}$;
  $acc \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}(\mathsf{RGSW}^{\mathsf{RNS}}((u_1)_i, s), acc, acc')$;
end
$\{ct_j\}_{j\in[L]} \leftarrow \mathsf{Expand}(acc, \log_2 k)$;
for *each bit position* i *of* u_2 **do**
  for $j = 0, \ldots, L-1$ **do**
    $ct'_j \leftarrow ct_j X^{-2^i}$;
    $ct_j \leftarrow \mathsf{CMUX}_{\mathsf{RNS}}(\mathsf{RGSW}^{\mathsf{RNS}}((u_2)_i, s), ct_j, ct'_j)$;
  end
end
return $\mathsf{SchemeSwitch}(\{ct_j\}_{j\in[L]}, \mathsf{csk})$;

3.4 RNS-Based Compact Reconstruction

Following [37,6], we use an RNS-based RGSW representation for the encrypted index bits to enable smaller parameters. For completeness, we describe how these $\mathsf{RGSW}^{\mathsf{RNS}}$ ciphertexts are reconstructed from the compact query representation used in our protocol. The reconstruction follows the compact conversion paradigm of prior work [42,12], adapted to the RNS-based representation.

Let $k_{\mathsf{rec}} = 2^{\lceil \log_2 L \rceil}$ and define the spacing $\delta = d/k_{\mathsf{rec}}$. The L RNS gadget components are assigned to the equally spaced coefficient positions $p_j = j\delta$, for $j \in [L]$. When $L < k_{\mathsf{rec}}$, the remaining positions are padded with zeros.

For a bit $b \in \{0, 1\}$, define

$$\widetilde{C}(b) = \mathsf{RLWE}\left(\sum_{j=0}^{L-1} g_{\mathsf{RNS},j} b X^{p_j}, s \right), \qquad p_j = j\delta.$$

Unlike the consecutive layout used in Sections 3.2 and 3.3, the gadget components are equally spaced in the ambient ring.

Communication representation. The full ciphertext $\widetilde{C}(b)$ is not transmitted. For every designated position p_j, the client extracts the corresponding LWE ciphertext and sends

$$Q_b = \left\{ \mathsf{ExtractLWE}_{p_j}(\widetilde{C}(b)) \right\}_{j\in[L]}.$$

Since the pseudorandom components of these LWE ciphertexts are derived from a common PRG seed, Q_b can be represented by a seed ρ and the corresponding coefficients $\{\bar{b}_j\}_{j\in[L]}$.

We define $\mathsf{Repack}(Q_b, \hat{P})$, where $\hat{P} = \{p_j\}_{j\in[L]}$, by regenerating $a'(X) \leftarrow \mathsf{PRG}(\rho)$ and setting $b'(X) = \sum_{j\in[L]} \bar{b}_j X^{p_j}$. The resulting ciphertext $\mathsf{Repack}(Q_b, \hat{P}) = (a'(X), b'(X))$ restores the extracted LWE components to their designated positions.

Reconstruction. Starting from the transmitted representation Q_b, the server uses Repack to place the extracted LWE ciphertexts back at their designated coefficient positions and projects it onto the k_{rec}-dimensional subring generated by $Y = X^\delta$ using the reverse homomorphic trace. We defer the details of $\mathsf{RevHomTrace}$ to Appendix A.2. The resulting plaintext is $\sum_{j=0}^{L-1} g_{\mathsf{RNS},j} b Y^j$. Partial Expand then separates the gadget components into $\{\mathsf{RLWE}(g_{\mathsf{RNS},j} b, s)\}_{j\in[L]}$, forming $\mathsf{RLWE'}^{\mathsf{RNS}}(b, s)$, from which $\mathsf{SchemeSwitch}$ produces $\mathsf{RGSW}^{\mathsf{RNS}}(b, s)$. The complete procedure is given in Algorithm 4.

Algorithm 4: $\mathsf{ReconstructBit}$

Input: Q_b, designated positions $\{p_j\}_{j\in[L]}$, and csk
Output: $\mathsf{RGSW}^{\mathsf{RNS}}(b, s)$

$ct \leftarrow \mathsf{Repack}(Q_b, \{p_j\}_{j\in[L]})$;
$ct \leftarrow \mathsf{RevHomTrace}_{k_{\mathsf{rec}}}(ct)$;
$\{ct'_j\}_{j\in[k_{\mathsf{rec}}]} \leftarrow \mathsf{Expand}(ct, \log_2 k_{\mathsf{rec}})$;
$\{ct_j\}_{j\in[L]} \leftarrow \{ct'_j\}_{j\in[L]}$;
return $\mathsf{SchemeSwitch}(\{ct_j\}_{j\in[L]}, \mathsf{csk})$;

Lemma 4. *For an encrypted bit $b \in \{0, 1\}$, Algorithm 4 transforms its transmitted representation Q_b into $\mathsf{RGSW}^{\mathsf{RNS}}(b, s)$.*

Proof. By definition of Repack, $\mathsf{Repack}(Q_b, \{p_j\}_{j\in[L]})$ places the extracted LWE ciphertexts back at their designated coefficient positions. The reverse homomorphic trace projects this ciphertext onto the subring generated by $Y = X^\delta$ and gives a plaintext of the form $\sum_{j=0}^{L-1} g_{\mathsf{RNS},j} b Y^j$. Recall from Appendix A.4 that partial Expand separates these coefficient classes, yielding $\{\mathsf{RLWE}(g_{\mathsf{RNS},j} b, s)\}_{j\in[L]}$, which form $\mathsf{RLWE'}^{\mathsf{RNS}}(b, s)$. The $\mathsf{SchemeSwitch}$ transformation defined in Appendix A.6 then produces $\mathsf{RGSW}^{\mathsf{RNS}}(b, s)$.

Choice of decomposition basis. Using the RNS basis for the reconstruction keys would introduce the auxiliary RNS moduli into the reconstruction parameters. The resulting modulus growth would require a larger ring dimension for the target security level. We therefore retain gadget-based key switching during $\mathsf{RevHomTrace}$, Expand, and $\mathsf{SchemeSwitch}$, while the reconstructed ciphertext is represented in the RNS basis for the subsequent CMUX and external-product operations. This separation limits parameter expansion during reconstruction while retaining the noise reduction provided by the auxiliary RNS moduli.

4 Our PIR Construction

We now instantiate the ciphertext-conversion techniques of Section 3 into two PIR protocols.

4.1 Setup and Database Encoding

Setup. During $\mathsf{ParamSet}$, the client samples the secret key s and generates the secret-key-dependent evaluation material pp required by the homomorphic operations. Since pp is independent of the database, the same setup material can be reused after database updates as long as s remains unchanged.

Database encoding. Recall that each SIMD orbit contains $\ell = d/2$ slots. We first consider one encoded database block containing $d\ell \log_2 p = d^2 \log_2 p/2$ bits. For each $u \in [\ell]$, let $r_u \in \mathbb{Z}_p^d$ denote the packed plaintext selected by the first query dimension. We arrange these plaintexts as $D \in \mathbb{Z}_p^{d\times\ell}$ and set $D[:,u] = \mathsf{NTT}_{p,d}(r_u)$.

To exploit the two SIMD orbits, we partition D row-wise into $D^+, D^- \in \mathbb{Z}_p^{\ell\times\ell}$. After cyclic diagonalization, corresponding diagonals of D^+ and D^- are encoded into the positive and negative SIMD orbits, respectively, yielding $\mathbf{M} = (M_0, \ldots, M_{\ell-1})$. For BSGS evaluation [24,36], choose a baby-step size $n_1 \mid \ell$ and let $n_2 = \ell/n_1$ be the number of giant steps. For $j \in [n_2]$ and $i \in [n_1]$, the server precomputes

$$\widehat{M}_{j,i} = \mathsf{NTT}_{q,d}\big(\mathsf{Rotate}(M_{jn_1+i}, i)\big).$$

The resulting collection $\{\widehat{M}_{j,i}\}_{j\in[n_2],i\in[n_1]}$ forms the database representation used by $\mathsf{BSGSMult}$.

For convenience, we write $\mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$ for an RLWE encryption of the SIMD encoding of the two-orbit selector $\mathbf{e}_u^{\pm}$. By construction, $\mathsf{BSGSMult}$ with this selector returns the slot vector $\Delta D[:,u]$. Since $D[:,u] = \mathsf{NTT}_{p,d}(r_u)$, SIMD decoding gives $\mathsf{iNTT}_{p,d}(\Delta D[:,u]) = \Delta r_u$. Hence, after the first-dimensional evaluation, the resulting RLWE ciphertext encrypts Δr_u in coefficient representation. The second query dimension can operate directly on the coefficients of r_u.

4.2 The Atom Protocol

Within one encoded block, let (u, v) denote the two-dimensional location of the requested record, where $u \in [\ell]$ selects r_u and $v \in [d]$ specifies a coefficient of r_u.

Let $\mathbf{g}_{\mathsf{RNS}} = (g_{\mathsf{RNS},0}, \ldots, g_{\mathsf{RNS},L-1})$ be the RNS gadget vector defined in Section 2, where L denotes its length. We instantiate $\mathsf{BitDecompRGSW}$ with a power-of-two block width $k_{\mathsf{s}} = 2^{h_{\mathsf{s}}} \geq L$. The second coordinate is decomposed as $v = v^{\mathsf{hi}} k_{\mathsf{s}} + v^{\mathsf{lo}}$, where $v^{\mathsf{hi}} = \lfloor v/k_{\mathsf{s}} \rfloor$ and $v^{\mathsf{lo}} = v \bmod k_{\mathsf{s}}$. We also write $u = \sum_i u_i 2^i$.

Recall from Section 3.4 that Q_b denotes the LWE ciphertext collection transmitted for an encrypted bit b, and $\hat{P}$ denotes the fixed set of coefficient positions associated with this representation. The procedure $\mathsf{ReconstructBit}(Q_b, \hat{P}, \mathsf{csk})$ recovers $\mathsf{RGSW}^{\mathsf{RNS}}(b, s)$. For a collection $\hat{Q} = \{Q_{b_i}\}_i$, we use $\mathsf{ReconstructBits}(\hat{Q}, \hat{P}, \mathsf{csk})$ as shorthand for component-wise application of $\mathsf{ReconstructBit}$.

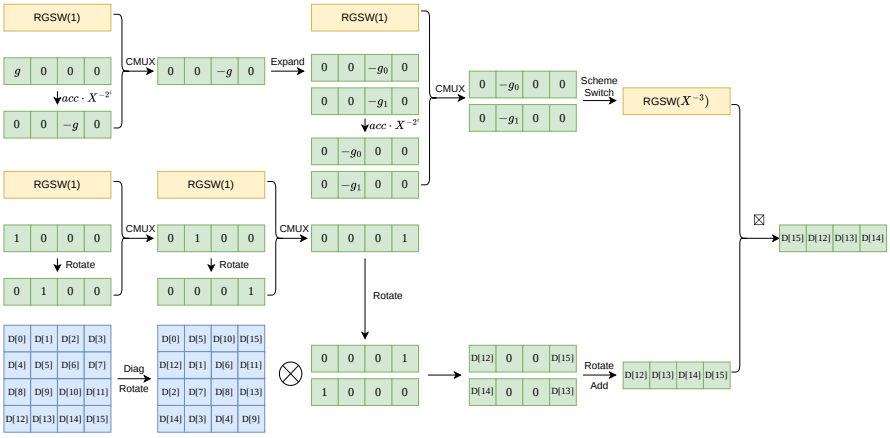


Fig. 2. Overview of Atom. The client transmits LWE ciphertexts representing the query-index bits. The server reconstructs the two query selectors from compact ciphertexts and evaluates the database with BSGS multiplication and one external product.

Query generation. The client generates the LWE query representations for the bits of u, v^{hi}, and v^{lo}.

Algorithm 5: $\mathsf{Query}_{\mathsf{s}}(pp, idx, s)$

Input: pp, query index idx, and secret key s
Output: $qu = (Q_1, Q_2^{\mathsf{hi}}, Q_2^{\mathsf{lo}})$

$(u, v) \leftarrow idx$;
$v^{\mathsf{hi}} \leftarrow \lfloor v/k_{\mathsf{s}} \rfloor,\ v^{\mathsf{lo}} \leftarrow v \bmod k_{\mathsf{s}}$;
$Q_1 \leftarrow \{Q_{u_i}\}_i$;
$Q_2^{\mathsf{hi}} \leftarrow \{Q_{(v^{\mathsf{hi}})_i}\}_i,\ Q_2^{\mathsf{lo}} \leftarrow \{Q_{(v^{\mathsf{lo}})_i}\}_i$;
return $(Q_1, Q_2^{\mathsf{hi}}, Q_2^{\mathsf{lo}})$;

Answer generation. The server first reconstructs the RNS-based RGSW encryptions of the query-index bits. Let $C_1 = \{\mathsf{RGSW}^{\mathsf{RNS}}(u_i, s)\}_i$ denote the reconstructed first-dimensional bits, and let C_2^{hi} and C_2^{lo} denote the corresponding reconstructed bit ciphertexts for v^{hi} and v^{lo}, respectively.

$\mathsf{OneHotGen}(C_1)$ produces $ct_u = \mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$. The BSGS evaluation therefore encrypts Δr_u. Instantiating $\mathsf{BitDecompRGSW}$ with block width k_{s} maps $(C_2^{\mathsf{hi}}, C_2^{\mathsf{lo}})$ to $C_v = \mathsf{RGSW}^{\mathsf{RNS}}(X^{-v}, s)$. The external product then encrypts $\Delta X^{-v} r_u(X)$, placing coefficient v at the designated position.

Multiple database blocks. For a database consisting of B_{blk} encoded blocks $\{\mathsf{db}_b\}_{b \in [B_{\mathsf{blk}}]}$, where B_{blk} denotes the number of blocks, the server reconstructs the query

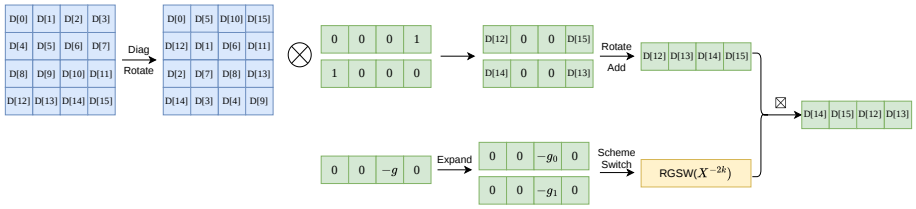


Fig. 3. Overview of Atom-f. The packed second-dimensional query is reconstructed with MonomialRGSW before BSGS multiplication and one external product.

ciphertexts only once. The same first- and second-dimensional selectors are reused for all blocks. The designated outputs are subsequently combined with RevPackRlwes. Thus, query reconstruction is independent of B_{blk}.

Algorithm 6: $\mathsf{Answer}_{\mathsf{s}}(pp, qu, \mathsf{db})$

Input: pp, $qu = (Q_1, Q_2^{\mathsf{hi}}, Q_2^{\mathsf{lo}})$, and db
Output: Response ciphertext ct

$C_1 \leftarrow \mathsf{ReconstructBits}(Q_1, \hat{P}, \mathsf{csk})$;
$C_2^{\mathsf{hi}} \leftarrow \mathsf{ReconstructBits}(Q_2^{\mathsf{hi}}, \hat{P}, \mathsf{csk})$;
$C_2^{\mathsf{lo}} \leftarrow \mathsf{ReconstructBits}(Q_2^{\mathsf{lo}}, \hat{P}, \mathsf{csk})$;
$ct_u \leftarrow \mathsf{OneHotGen}(C_1)$;
$ct \leftarrow \mathsf{BSGSMult}(\mathsf{db}, ct_u)$;
$C_v \leftarrow \mathsf{BitDecompRGSW}(C_2^{\mathsf{hi}}, C_2^{\mathsf{lo}}, \mathsf{csk})$;
$ct \leftarrow C_v \boxtimes_{\mathsf{RNS}} ct$;
return ct;

4.3 The Atom-f Variant

Atom-f reduces query-reconstruction cost by transmitting less compressed ciphertexts. The first-dimensional selector is sent directly as one RLWE ciphertext. The second-dimensional query is represented by one packed RLWE ciphertext and reconstructed with MonomialRGSW.

Let B denote the gadget decomposition base and $t = \lceil \log_B q \rceil$ the corresponding gadget length. As required by MonomialRGSW, choose a power-of-two alignment width $k_{\mathsf{f}} \in [t, 2t)$. The second-dimensional index $v \in [d/k_{\mathsf{f}}]$ identifies the aligned coefficient position $k_{\mathsf{f}} v$. Accordingly, the coefficients associated with the v-th selectable packed record are placed at positions $k_{\mathsf{f}} v, \ldots, k_{\mathsf{f}} v + t - 1$ in r_u.

Query generation. For a query index (u, v), the client sends the two-orbit selector $ct_1 = \mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$ together with $ct_2 = \mathsf{RLWE}\left(\sum_{j=0}^{t-1} B^j X^{-k_{\mathsf{f}} v + j}, s\right)$. Negative exponents are interpreted in $R_q = \mathbb{Z}_q[X]/(X^d + 1)$.

Algorithm 7: $\mathsf{Query}_{\mathsf{f}}(pp, idx, s)$

Input: pp, query index idx, and secret key s
Output: $qu = (ct_1, ct_2)$

$(u, v) \leftarrow idx$;
$ct_1 \leftarrow \mathsf{RLWE}(\Delta \mathbf{e}_u^{\pm}, s)$;
$ct_2 \leftarrow \mathsf{RLWE}(\sum_{j=0}^{t-1} B^j X^{-k_{\mathsf{f}} v + j}, s)$;
return (ct_1, ct_2);

Answer generation. The server first evaluates $\mathsf{BSGSMult}(\mathsf{db}, ct_1)$, which encrypts Δr_u. Instantiating MonomialRGSW with alignment width k_{f} converts ct_2 into $C_- = \mathsf{RGSW}^{\mathsf{gadget}}(X^{-k_{\mathsf{f}} v}, s)$. The external product therefore shifts the selected aligned block to the output positions.

Algorithm 8: $\mathsf{Answer}_{\mathsf{f}}(pp, qu, \mathsf{db})$

Input: pp, $qu = (ct_1, ct_2)$, and db
Output: Response ciphertext ct

$ct \leftarrow \mathsf{BSGSMult}(\mathsf{db}, ct_1)$;
$C_- \leftarrow \mathsf{MonomialRGSW}(ct_2, \mathsf{csk}, k_{\mathsf{f}})$;
$ct \leftarrow C_- \boxtimes_{\mathsf{gadget}} ct$;
return ct;

Multiple database blocks. For B_{blk} blocks $\{\mathsf{db}_b\}_{b \in [B_{\mathsf{blk}}]}$, the server reconstructs C_- once and reuses it for all blocks. Each block produces t designated coefficients after the external product. These $B_{\mathsf{blk}} t$ coefficients are shifted to distinct packing positions and combined into one response with RevPackRlwes. For parameter sets supporting the corresponding packing, let $T = \log_2(d/(B_{\mathsf{blk}} t))$ denote the packing depth.

Small records. For small records, we follow Respire [12] and pack multiple records into one plaintext. After PIR selects the corresponding packed plaintext, an additional homomorphic selection extracts the requested record, followed by RingSwitch to reduce the response dimension.

Algorithm 9: $\mathsf{Answer_f}$ for multiple blocks

Input: pp, $qu = (ct_1, ct_2)$, and $\{\mathsf{db}_b\}_{b\in[B_{\mathsf{blk}}]}$
Output: Response ciphertext ct

$C_- \leftarrow \mathsf{MonomialRGSW}(ct_2, \mathsf{csk}, k_{\mathsf{f}})$;
for $b = 0, \ldots, B_{\mathsf{blk}} - 1$ **do**
  $ct_b \leftarrow \mathsf{BSGSMult}(\mathsf{db}_b, ct_1)$;
  $ct_b \leftarrow C_- \boxtimes_{\mathsf{gadget}} ct_b$;
end
for $b = 0, \ldots, B_{\mathsf{blk}} - 1$ **do**
  for $j = 0, \ldots, t - 1$ **do**
    $\widehat{ct}_{bt+j} \leftarrow ct_b X^{-j}$;
  end
end
$ct \leftarrow \mathsf{RevPackRlwes}(\{\widehat{ct}_i\}_{i\in[B_{\mathsf{blk}}t]}, T)$;
return ct;

4.4 Correctness and Security

Theorem 1 (Correctness). *If the final ciphertext noise is below $\Delta/2$, both Atom and Atom-f correctly recover* $\mathsf{DB}[idx]$.

Proof. Let (u, v) denote the queried location. For Atom, Lemmas 4, 1, and 3 reconstruct the required first- and second-dimensional query ciphertexts. The BSGS evaluation encrypts Δr_u, and the external product with $\mathsf{RGSW}^{\mathsf{RNS}}(X^{-v}, s)$ moves the requested coefficient to the output position.

For Atom-f, Lemma 2 reconstructs $\mathsf{RGSW}^{\mathsf{gadget}}(X^{-k_{\mathsf{f}}v}, s)$, and the aligned database layout ensures that the external product selects the requested record. Response packing and ring switching preserve the selected plaintext. The stated noise bound then guarantees correct decoding.

Query privacy. Query privacy of both Atom and Atom-f follows from Theorem 2.

Theorem 2 (Query Privacy). *Assuming RLWE IND-CPA security and circular/KDM security of the secret-key-dependent evaluation material [7,11], Atom and Atom-f satisfy query privacy against a semi-honest server.*

Proof. For Atom, each transmitted Q_b is obtained by publicly applying $\mathsf{ExtractLWE}$ to a fresh RLWE encryption $\widetilde{C}(b)$; hence, its indistinguishability follows from RLWE IND-CPA security. For Atom-f, both query components are fresh RLWE encryptions and are indistinguishable by the same assumption. Circular/KDM security covers the secret-key-dependent key-switching and conversion ciphertexts. All subsequent reconstruction and database evaluation are efficient post-processing and therefore preserve indistinguishability.

Scheme	Type	256MB		1GB		8GB	
		Comm.	Comp.	Comm.	Comp.	Comm.	Comp.
Spiral [43]	SK	13664.3KB	0.6s	15456.3KB	0.6s	17695.7KB	0.7s
Respire [13]	SK	3980.3KB	17.4s	3980.3KB	70.3s	3980.3KB	655.5s
VIA [35]	SK	15617.0KB	4.7s	15617.0KB	8.4s	15617.0KB	89.0s
KsPIR [38]	SK	9011.2KB	3.2s	9216.0KB	12.7s	9523.2KB	99.9s
Atom-s	SK	27238.4KB	4.1s	41369.6KB	15.7s	41369.6KB	120.6s
Atom-f	SK	11161.6KB	3.2s	11366.4KB	12.8s	11673.6KB	100.1s
SimplePIR [27]	DB	60208.1KB	48.7s	121520.1KB	241.0s	350827.5KB	1982.2s
Hintless [32]	DB	-	53.2s	-	187.0s	-	1615.2s
YPIR+SP [45]	DB	-	7.2s	-	34.1s	-	607.7s
InsPIRe [41]	DB	-	517.6s	-	2045.3s	-	16320.2s

Table 1. Setup costs of the evaluated PIR schemes.

Database	Metric	Spiral [43]	Respire [13]	VIA [35]	KsPIR [38]	Atom-s	Atom-f
256MB	Client Comp.	8.6ms	0.4ms	3.6ms	5.4ms	1.8ms	1.7ms
	Server Comp.	1768.0ms	1187.6ms	2154.0ms	214.6ms	435.2ms	255.3ms
	Online Comp.	**1776.6ms**	**1188.0ms**	**2157.6ms**	**220.0ms**	**437.0ms**	**257.0ms**
	Upload Size	28.0KB	4.1KB	0.7KB	152.0KB	1.0KB	56.0KB
	Download Size	20.0KB	2.0KB	1.4KB	2.0KB	2.0KB	2.0KB
	Online Comm.	**48.0KB**	**6.1KB**	**2.2KB**	**154.0KB**	**3.0KB**	**58.0KB**
1GB	Client Comp.	16.5ms	0.4ms	4.0ms	5.4ms	1.8ms	1.6ms
	Server Comp.	3827.4ms	3037.5ms	5519.0ms	688.5ms	998.1ms	706.6ms
	Online Comp.	**3843.9ms**	**3037.9ms**	**5523.0ms**	**693.9ms**	**999.9ms**	**708.2ms**
	Upload Size	28.0KB	7.7KB	0.8KB	152.0KB	1.4KB	56.0KB
	Download Size	20.5KB	2.0KB	1.4KB	2.0KB	2.0KB	2.0KB
	Online Comm.	**48.5KB**	**9.7KB**	**2.2KB**	**154.0KB**	**3.4KB**	**58.0KB**
8GB	Client Comp.	18.0ms	0.4ms	4.6ms	5.4ms	1.8ms	1.7ms
	Server Comp.	28852.5ms	17545.1ms	35848.0ms	5193.7ms	5030.2ms	5084.4ms
	Online Comp.	**28870.5ms**	**17545.5ms**	**35852.6ms**	**5199.1ms**	**5032.0ms**	**5086.1ms**
	Upload Size	28.0KB	14.8KB	0.9KB	152.0KB	1.8KB	56.0KB
	Download Size	21.0KB	2.0KB	2.2KB	2.0KB	2.0KB	2.0KB
	Online Comm.	**49.0KB**	**16.8KB**	**3.1KB**	**154.0KB**	**3.8KB**	**58.0KB**

Table 2. Performance comparison with SK-dependent PIR.

Database	Metric	SimplePIR [27]	Hintless [32]	YPIR+SP [45]	InsPIRe [41]	Atom-s	Atom-f
256MB	Client Comp.	22.9ms	85.3ms	87.5ms	44.5ms	1.8ms	1.7ms
	Server Comp.	17.1ms	598.3ms	90.5ms	686.2ms	435.2ms	255.3ms
	Online Comp.	**40.0ms**	**683.6ms**	**178.0ms**	**730.7ms**	**437.0ms**	**257.0ms**
	Upload Size	57.7KB	364.0KB	1357.8KB	182.0KB	1.0KB	56.0KB
	Download Size	58.8KB	47104.0KB	12.0KB	12.0KB	2.0KB	2.0KB
	Online Comm.	**116.5KB**	**47468.0KB**	**1369.8KB**	**194.0KB**	**3.0KB**	**58.0KB**
1GB	Client Comp.	46.9ms	158.2ms	158.9ms	48.1ms	1.8ms	1.6ms
	Server Comp.	67.6ms	866.1ms	250.5ms	2627.3ms	998.1ms	706.6ms
	Online Comp.	**114.5ms**	**1024.3ms**	**409.4ms**	**2675.4ms**	**999.9ms**	**708.2ms**
	Upload Size	117.0KB	376.0KB	4045.8KB	182.0KB	1.4KB	56.0KB
	Download Size	118.7KB	47104.0KB	12.0KB	48.0KB	2.0KB	2.0KB
	Online Comm.	**235.7KB**	**47480.0KB**	**4057.8KB**	**230.0KB**	**3.4KB**	**58.0KB**
8GB	Client Comp.	140.5ms	598.7ms	913.9ms	39.5ms	1.8ms	1.7ms
	Server Comp.	577.4ms	40668.3ms	1719.9ms	16269.4ms	5030.2ms	5084.4ms
	Online Comp.	**717.9ms**	**41267.0ms**	**2633.8ms**	**16308.9ms**	**5032.0ms**	**5086.1ms**
	Upload Size	342.2KB	488.0KB	28672.0KB	182.0KB	1.8KB	56.0KB
	Download Size	342.6KB	47104.0KB	12.0KB	384.0KB	2.0KB	2.0KB
	Online Comm.	**684.8KB**	**47592.0KB**	**28684.0KB**	**566.0KB**	**3.8KB**	**58.0KB**

Table 3. Performance comparison with DB-dependent PIR.

5 Implementation and Evaluation

5.1 Experimental Setup and Parameters

We instantiate both variants with polynomial degree $d = 4096$ and a 56-bit ciphertext modulus q. For RNS-based operations, we use a 60-bit auxiliary modulus P and the extended modulus $Q = qP$, with $|Q| = 116$ bits. The plaintext modulus is $p = 65537$, and the noise standard deviations over Q and q are $\sigma_1 = 3.19$ and $\sigma_2 = 0.82$, respectively. We estimate concrete security using the lattice-estimator [2] under the BDGL16 cost model [5]. All Atom parameter sets provide at least 128 bits of computational security and target a decryption failure probability below 2^{-40}. Our implementation uses Intel HEXL [8] for NTT operations. All experiments are single-threaded and run on Ubuntu 22.04 using an Intel(R) Xeon(R) Platinum 8358 CPU at 2.60 GHz with 2 TB of RAM. Unless otherwise stated, each runtime is averaged over 50 executions. Atom-f uses the RNS basis for first-dimensional automorphisms and RevPackRlwes, and the gadget basis for MonomialRGSW and the subsequent external product. Atom-s uses gadget-based compact-query reconstruction and the RNS basis for subsequent key switching, CMUX, and external products. All RNS-based operations are evaluated over the extended modulus $Q = qP$ and subsequently modulus-switched back to q.

5.2 Performance Comparison

We compare Atom-s and Atom-f with the SK-dependent schemes Spiral [43], Respire [13], VIA [35], and KsPIR [38], as well as the DB-dependent schemes SimplePIR [27], Hintless [32], YPIR [45], and InsPIRe [41]. Following Respire [12], all experiments use 256-byte records and databases of 256 MB, 1 GB, and 8 GB.

Tables 1, 2, and 3 report the setup and online performance of the evaluated schemes, while Fig. 4 summarizes their computation-communication trade-offs. Setup and online communication include both upload and download, and online computation includes client and server computation. SK-dependent schemes use reusable secret-key-dependent setup material [42,12,34,37], whereas DB-dependent schemes require database-dependent preprocessing or hints [26,33,44,40]. The released VIA implementation required minor adjustments in our environment. We report the resulting measurements under its original parameter scale, which provides about 110 bits of security, as a favorable reference point rather than a strictly security-matched baseline. Totals are computed from unrounded measurements.

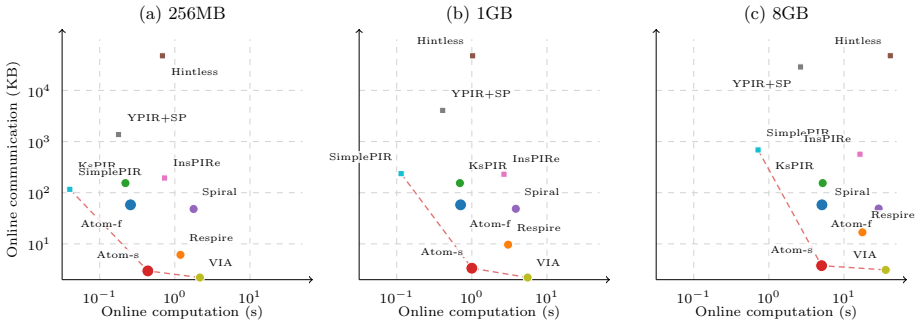


Fig. 4. Pareto comparison with SK-dependent and DB-dependent PIR.

Comparison with SK-dependent PIR. Compared with Spiral [42] and Respire [12], Atom-s improves both online computation and communication across all evaluated database sizes, achieving 2.72×-5.74× speedups and 2.06×-16.22× lower communication. Atom-f further reduces computation, with 3.45×-6.91× speedups, while maintaining 58.0KB of online communication.

Since VIA [34] provides only parameter sets with approximately 110 bits of computational security, we compare directly against this provided configuration, while Atom and the other baselines are instantiated at least 128 bits of security. Atom-s is 4.94×-7.13× faster while requiring only1.22×-1.50× the online communication. Atom-f is 7.05×-8.40× faster, at the cost of higher communication.

KsPIR [37] is the closest baseline structurally, as Atom retains its two-dimensional evaluation framework. Atom-s reduces query upload by 86.86×-158.33×, with additional computation for smaller databases but a 1.03× speedup at 8GB. Atom-f reduces upload by approximately 2.7× while keeping online computation within 17% of KsPIR.

Comparison with DB-dependent PIR. Compared with the DB-dependent baselines [26,33,44,40], Atom-s consistently occupies the low-communication regime,

requiring only 3.0KB-3.8KB across the evaluated database sizes. At 8GB, this corresponds to a 150.93×-12691.20× reduction in online communication. SimplePIR and YPIR+SP achieve lower online computation but require substantially more communication, whereas Atom-s improves both metrics over Hintless and InsPIRe, with speedups of 1.02×-8.20× and 1.67×-3.24×, respectively. Atom-f provides a more computation-oriented operating point while remaining below all evaluated DB-dependent schemes in communication.

Fig. 4 compares the online computation-communication trade-offs of all evaluated SK-dependent and DB-dependent schemes. Panels (a)-(c) correspond to 256MB, 1GB, and 8GB databases, respectively. Circles and squares denote SK-dependent and DB-dependent schemes, while Atom-s is highlighted in red. The dashed line connects SimplePIR, Atom-s, and VIA to illustrate representative Pareto-optimal trade-offs. Both axes are logarithmic, with lower-left indicating better performance. Across all three database sizes, Atom-s lies on the joint Pareto frontier, combining low online communication with competitive computation.

Acknowledgments

We thank ChatGPT for assistance with language editing and coding. All technical details were verified by the authors.

References

1. Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.
2. Martin R. Albrecht, Rachel Player, and Sam Scott. LWE estimator. `https://github.com/malb/lwe-estimator`, 2017. Accessed: 2023-12-01.
3. Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.
4. Hilal Asi, Fabian Boemer, Nicholas Genise, Muhammad Haris Mughees, Tabitha Ogilvie, Rehan Rishi, Kunal Talwar, Karl Tarbe, Akshay Wadia, Ruiyu Zhu, et al. Scalable private search with wally. *arXiv preprint arXiv:2406.06761*, 2024.
5. Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 10–24. SIAM, 2016.
6. Olivier Bernard and Marc Joye. Approximate crt-based gadget decomposition and application to tfhe blind rotation. *Cryptology ePrint Archive*, 2024.
7. John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *Selected Areas in Cryptography*, volume 2595 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2003.
8. Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. Intel HEXL: Accelerating homomorphic encryption with Intel AVX512-IFMA52. Cryptology ePrint Archive, Report 2021/420, 2021. `https://eprint.iacr.org/2021/420`.
9. Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual cryptology conference*, pages 868–886. Springer, 2012.
10. Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014.
11. Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology-CRYPTO 2011*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.
12. Alexander Burton, Samir Jordan Menon, and David J Wu. Respire: High-rate pir for databases with small records. In *ACM CCS*, pages 1463–1477, 2024.
13. Alexander Burton, Samir Jordan Menon, and David J Wu. Respire: High-rate pir for databases with small records. https://github.com/AMACB/respire, 2024.
14. Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *Advances in Cryptology-ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I 22*, pages 3–33. Springer, 2016.
15. Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.
16. Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023.

17. Leo de Castro, Kevin Lewi, and Edward Suh. Whispir: Stateless private information retrieval with low communication. *Cryptology ePrint Archive*, 2024.
18. Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 617–640. Springer, 2015.
19. Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *Cryptology ePrint Archive*, 2012.
20. Adrià Gascón, Yuval Ishai, Mahimna Kelkar, Baiyu Li, Yiping Ma, and Mariana Raykova. Computationally secure aggregation and private information retrieval in the shuffle model. In *Proceedings of the 2024 on ACM SIGSAC conference on computer and communications security*, pages 4122–4136, 2024.
21. Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in BGV-style homomorphic encryption. In *Security and Cryptography for Networks*, pages 19–37. Springer, 2012.
22. 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, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 75–92. Springer, 2013.
23. Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *Topics in Cryptology-CT-RSA 2019*, volume 11405 of *Lecture Notes in Computer Science*, pages 83–105. Springer, 2019.
24. Shai Halevi and Victor Shoup. Faster homomorphic linear transformations in helib. In *Annual International Cryptology Conference*, pages 93–120. Springer, 2018.
25. Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with tiptoe. In *Proceedings of the 29th symposium on operating systems principles*, pages 396–416, 2023.
26. Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast {Single-Server} private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, 2023.
27. Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast {Single-Server} private information retrieval. https://github.com/ahenzinger/simplepir, 2023.
28. Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 239–248. IEEE, 2006.
29. Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings 38th annual symposium on foundations of computer science*, pages 364–373. IEEE, 1997.
30. Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. 2015.
31. Kang Hoon Lee and Ji Won Yoon. Homomorphic field trace revisited: Breaking the cubic noise barrier. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(1):105–134, 2026.
32. Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. `https://github.com/google/hintless_pir`, 2023.

33. Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless single-server private information retrieval. In *Annual International Cryptology Conference*, pages 183–217. Springer, 2024.
34. Chenyang Liu, Xukun Wang, and Zhifang Zhang. Via: Communication-efficient single-server private information retrieval. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 1503–1521. IEEE, 2026.
35. Chenyang Liu, Xukun Wang, and Zhifang Zhang. VIA: Communication-efficient single-server private information retrieval. `https://github.com/owniai/VIA`, 2026.
36. Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. Pegasus: bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1057–1073. IEEE, 2021.
37. Ming Luo, Feng-Hao Liu, and Han Wang. Faster fhe-based single-server private information retrieval. In *ACM CCS*, pages 1405–1419, 2024.
38. Ming Luo, Feng-Hao Liu, and Han Wang. Faster fhe-based single-server private information retrieval. https://github.com/mmingluo/kspir, 2024.
39. Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology-EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30-June 3, 2010. Proceedings 29*, pages 1–23. Springer, 2010.
40. Rasoul Akhavan Mahdavi, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Inspire: Communication-efficient pir with server-side preprocessing. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 1463–1482. IEEE, 2026.
41. Rasoul Akhavan Mahdavi, Sarvar Patel, Joon Young Seo, and Kevin Yeo. InsPIRe: Communication-efficient pir with server-side preprocessing. `https://github.com/google/private-membership/tree/main/research/InsPIRe`, 2026.
42. 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.
43. Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. https://github.com/menonsamir/spiral, 2022.
44. Samir Jordan Menon and David J Wu. {YPIR}:{High-Throughput}{Single-Server}{PIR} with silent preprocessing. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5985–6002, 2024.
45. Samir Jordan Menon and David J Wu. {YPIR}:{High-Throughput}{Single-Server}{PIR} with silent preprocessing. `https://github.com/menonsamir/ypir`, 2024.
46. Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 700–718. Springer, 2012.
47. Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. Cryptology ePrint Archive, Paper 2023/112, 2023.
48. Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. {PIR-Tor}: Scalable anonymous communication using private information retrieval. In *20th USENIX security symposium (USENIX security 11)*, 2011.
49. 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.

50. Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM (JACM)*, 56(6):1–40, 2009.
51. Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71:57–81, 2014.
52. Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *arXiv preprint arXiv:2004.13293*, 2020.

A Details of Homomorphic Encryption

A.1 Basic Homomorphic Operations

All ciphertexts follow the BFV-style RLWE encryption. We use the following standard operations [9,19,10]:

- $\mathsf{Add}(ct_1, ct_2)$: Given $ct_1 = (c_0^1, c_1^1)$ and $ct_2 = (c_0^2, c_1^2)$, return
$$ct = (c_0^1 + c_0^2,\ c_1^1 + c_1^2).$$
- $\mathsf{PCMul}(pt, ct)$: Given $pt \in R_q$ and $ct = (c_0, c_1)$, return
$$ct' = (pt \cdot c_0,\ pt \cdot c_1).$$
- $\mathsf{KeySwitch}(ct, \mathsf{ksk})$: Given $ct = \mathsf{RLWE}(m, s')$ and $\mathsf{ksk} = \mathsf{Rlev}(s', s)$, return $ct' = \mathsf{RLWE}(m, s)$.
- $\mathsf{Auto}(ct, i)$: Given $ct = \mathsf{RLWE}(m, s)$ and $i \in \mathbb{Z}$, return $ct' = \mathsf{RLWE}(\sigma_i(m), s)$, where $\sigma_i : X \to X^i$.

A.2 Reverse HomTrace

We use RevHomTrace as an optimized implementation of the homomorphic trace operation HomTrace used in Section 3.4. Lee and Yoon propose RevHomTrace to remove the phase amplification of the standard homomorphic trace [31]. The main idea is to apply modulus switching by a factor of two at each iteration and evaluate the automorphisms in reverse order.

Let
$$ct = \mathsf{RLWE}(m(X), s),$$
and let n be a power of two dividing d. The goal is to compute
$$\mathsf{RLWE}\Big(\frac{n}{d}\mathsf{Tr}_{K/K_n}(m(X)), s\Big).$$
In particular, for $n = 1$, the output encrypts the constant coefficient of $m(X)$ without the usual multiplicative factor d.

At each level, the modulus switch scales the current phase by one half. Adding the corresponding automorphic image cancels one group of coefficients while preserving those removed in previous levels. The reverse automorphism order keeps the zeroized coefficients fixed in later iterations. After $\log_2(d/n)$ levels, only the coefficients in the n-dimensional subring remain.

The correctness and noise analysis follow from Theorem 4 of [31]. The additional noise variance is bounded by
$$4\log_2(d/n)\, V_{\mathsf{MS}} + \log_2(d/n)\, V_{\mathsf{Auto}}.$$

Algorithm 10: $\mathsf{RevHomTrace}(ct, n)$

Input: An RLWE ciphertext $ct = \mathsf{RLWE}(m(X), s)$ and a power-of-two $n \le d$
Output: $ct' = \mathsf{RLWE}(\frac{n}{d}\mathsf{Tr}_{K/K_n}(m(X)), s)$

$ct' \leftarrow ct$;
for $k = \log_2 n, \ldots, \log_2 d - 1$ **do**
  $\bar{ct} \leftarrow \mathsf{ModRaise}_{q/2\to q}\big(\mathsf{ModSwitch}_{q\to q/2}(ct')\big)$;
  $ct' \leftarrow \mathsf{Add}\big(\bar{ct}, \mathsf{Auto}(\bar{ct}, 2^{k+1}+1)\big)$;
end
return ct';

A.3 Reverse RLWE Packing

Since an RLWE ciphertext is a rank-one GLWE ciphertext, we adapt MS-PackLWEs [31] to RLWE ciphertexts and denote the resulting procedure by RevPackRlwes.

Let $m = 2^\ell$, $T = \log_2(d/m)$, and $ct_i = \mathsf{RLWE}(\mu_i + e_i(X), s)$ for $0 \le i < m$, where μ_i is the constant coefficient to be packed. The goal is to obtain

$$\mathsf{RLWE}\left(\sum_{i=0}^{m-1} \mu_i X^{i2^T}, s\right).$$

Here, $\mathsf{ModSwitch}_{q\to q/2}$ scales a ciphertext from modulus q to $q/2$, and $\mathsf{ModRaise}_{q/2\to q}$ lifts the resulting coefficients back to modulus q. The automorphism keys required by Auto are included in the evaluation material.

Algorithm 11: $\mathsf{RevPackRlwes}(\{ct_i\}_{i=0}^{m-1}, T)$

Input: $m = 2^\ell$ RLWE ciphertexts $\{ct_i\}_{i=0}^{m-1}$, packing depth $T = \log_2(d/m)$
Output: $ct = \mathsf{RLWE}(\sum_{i=0}^{m-1} \mu_i X^{i2^T}, s)$

if $m = 1$ **then**
  return ct_0;
end
$ct_{\mathsf{even}} \leftarrow \mathsf{RevPackRlwes}(\{ct_{2j}\}_{j=0}^{m/2-1}, T+1)$;
$ct_{\mathsf{odd}} \leftarrow \mathsf{RevPackRlwes}(\{ct_{2j+1}\}_{j=0}^{m/2-1}, T+1)$;
$\widetilde{ct} \leftarrow ct_{\mathsf{even}} - X^{2^T} ct_{\mathsf{odd}}$;
$\widetilde{ct} \leftarrow \mathsf{ModRaise}_{q/2\to q}\big(\mathsf{ModSwitch}_{q\to q/2}(\widetilde{ct})\big)$;
$ct \leftarrow \widetilde{ct} + \mathsf{Auto}(\widetilde{ct}, 2^\ell + 1)$;
$ct \leftarrow ct + X^{2^T} ct_{\mathsf{odd}}$;
return ct;

At each recursion level, the automorphism keeps the even component and negates the shifted odd component. After modulus switching, adding its automorphic image recovers the even component, while $X^{2^T} ct_{\mathsf{odd}}$ places the odd

component in the interleaved positions. Repeating this for ℓ levels gives the desired packing.

This procedure is an RLWE specialization of Algorithm 6 in [31], adapted to our notation and zero-based indexing. We refer to [31] for more details.

A.4 Expand

The Expand [3] procedure takes an RLWE ciphertext and produces a set of ciphertexts that encode its coefficient-wise decomposition. Given a ciphertext ct encrypting a polynomial $m(X)$, the algorithm outputs a collection $\{ct_j\}$ such that each ct_j corresponds to a rotated and scaled component of m. This is achieved by iteratively applying automorphisms and combining the results to isolate different coefficient positions. The final normalization step ensures that the outputs collectively preserve the original message. We refer to [3] for the corresponding correctness proof.

Algorithm 12: $\mathsf{Expand}(ct, r)$

Input: Ciphertext ct and $0 \leq r \leq \log_2 d$
Output: Ciphertexts $\{ct_j\}_{j\in[2^r]}$

$ct_0 \leftarrow ct$
for $i = 0, ..., r-1$ **do**
  $\ell \leftarrow d/2^i + 1$
  for $j = 0, ..., 2^i - 1$ **do**
    $ct' \leftarrow \mathsf{Auto}(ct_j, \ell)$
    $ct_{j+2^i} \leftarrow (ct_j - ct') \cdot X^{-2^i}$
    $ct_j \leftarrow ct_j + ct'$
  end
end
$ct_j \leftarrow ct_j \cdot 2^{-r}$
return $\{ct_j\}_{j\in[2^r]}$

A.5 Ring Switch

Let $\iota : R_{d_2,q} \to R_{d_1,q}$ denote the embedding from the smaller ring to the larger ring, and let ι^{-1} denote the induced coefficient map on embedded ciphertext components. We model ring switching [21] as projection to the smaller ring followed by a key-switching step. We refer to [21] for the corresponding correctness proof.

A.6 SchemeSwitch

The SchemeSwitch [47] transformation converts a collection of RLWE ciphertexts into an RGSW ciphertext under the same secret key. Given an RLWE encryption

Algorithm 13: $\mathsf{RingSwitch}(ct, \mathsf{ksk})$

Input: An RLWE ciphertext $ct = (c_0, c_1) \in R_{d_1,q}^2$ and a key-switching key ksk from $\iota^{-1}(s_1)$ to s_2
Output: An RLWE ciphertext $ct' = (c_0', c_1') \in R_{d_2,q}^2$

$(\hat{c}_0, \hat{c}_1) \leftarrow (\iota^{-1}(c_0), \iota^{-1}(c_1))$
$ct' \leftarrow \mathsf{KeySwitch}((\hat{c}_0, \hat{c}_1), \mathsf{ksk})$
return ct'

of a message m represented as a set of ciphertexts $\{ct_j\}$, the algorithm produces an RGSW encryption by decomposing the ciphertext components and combining them with the relevant switching material. Concretely, each RLWE component is decomposed into smaller digits, which are then linearly combined with the corresponding key material to encode the multiplicative structure required by the RGSW representation. We refer to [47] for the corresponding correctness proof.

Algorithm 14: $\mathsf{SchemeSwitch}(\{ct_j\}_{j\in[t]}, \mathsf{ksk})$

Input: A set of ciphertexts $\{ct_j\}_{j\in[t]}$ and the relevant switching material ksk
Output: A RGSW ciphertext $\mathsf{RGSW}(m, s)$

$\mathsf{RLWE}'(m, s) \leftarrow (ct_0, ..., ct_{t-1})$
for $i \leftarrow 0, ..., t-1$ **do**
  $(c_0, c_1) \leftarrow ct_i$
  $\{a_j\} \leftarrow \mathsf{Decompose}(c_0)$
  $\mathsf{RLWE}'(s \cdot m, s)_i = (c_1, 0) + \langle \{a_j\}, \mathsf{ksk} \rangle$
end
return $[\mathsf{RLWE}'(s \cdot m, s), \mathsf{RLWE}'(m, s)]$

A.7 BSGS Multiplication

The $\mathsf{BSGSMult}$ procedure evaluates the encoded plaintext matrix V on the ciphertext ct using a baby-step giant-step decomposition.

We write

$$\ell = n_1 n_2,$$

where n_1 and n_2 are the baby-step and giant-step dimensions, respectively.

The server first precomputes the n_1 baby-step rotations

$$ct_j = \mathsf{Rotate}(ct, j), \qquad j \in [n_1].$$

For each giant step $i \in [n_2]$, these ciphertexts are multiplied by the corresponding plaintext components of V and accumulated into *acc*. The accumulator is then

Algorithm 15: $\mathsf{BSGSMult}(V, ct)$

Input: A ciphertext ct and an encoded plaintext hypercube $V \in \mathbb{Z}^{\ell \times \ell \times 2}$
Output: A ciphertext ct'

$\ell = n_1 * n_2, ct' \leftarrow (0, 0)$
for $i = 0, ..., n_1 - 1$ **do**
  $ct_i \leftarrow \mathsf{Rotate}(ct, i)$
end
for $i = 0, ..., n_2 - 1$ **do**
  $acc \leftarrow (0, 0)$
  for $j = 0, ..., n_1 - 1$ **do**
    $tmp \leftarrow \mathsf{PCMul}(V_{i,j}, ct_j)$
    $acc \leftarrow \mathsf{Add}(acc, tmp)$
  end
  $acc \leftarrow \mathsf{Rotate}(acc, i \cdot n_1)$
  $ct' \leftarrow \mathsf{Add}(ct', acc)$
end
return ct'

rotated by in_1 positions and added to the output ciphertext. Repeating this over all n_2 giant steps produces the desired linear transformation.

Compared with evaluating all ℓ rotations independently, BSGS requires only $n_1 + n_2$ ciphertext rotations. Choosing $n_1 \approx n_2 \approx \sqrt{\ell}$ reduces the rotation complexity from $O(\ell)$ to $O(\sqrt{\ell})$. We refer to [37] for the corresponding correctness proof.