SkrrtPIR: Doubly-Stateless Batch PIR from Single-Key RLWE Unpacking

Keewoo Lee
Ethereum Foundation

Yongha Son
Sungshin Women's University

Abstract

Batch Private Information Retrieval (Batch PIR) allows a client to privately retrieve multiple entries from a public database while amortizing query costs. However, concretely efficient schemes typically rely on per-client server state, such as client-specific evaluation keys, which makes queries linkable across sessions and complicates deployment. In this work, we initiate the study of doubly-stateless Batch PIR, where the server stores no per-client state and the client maintains no database-dependent hint. For this model, we construct a communication-efficient scheme that substantially reduces communication compared with the natural baseline of adapting off-the-shelf Batch PIR schemes by sending fresh evaluation keys with each batch query. The core technical ingredient is a single-key RLWE unpacking procedure that replaces the traditional tree-based approach, which requires logarithmically many Galois keys, with a linear walk over a cyclic Galois subgroup requiring only a single Galois key.

1 Introduction

A (single-server) Private Information Retrieval (PIR) scheme [18, 40] allows a client to retrieve a database entry without revealing to the server which entry it requests. A long line of work has since significantly improved PIR's concrete efficiency. However, most of these works rely on *statefulness*, often an obstacle in practice, as we detail below.

Client-stateful PIR. In a *client-stateful* scheme, the client maintains a database-dependent *hint*. For instance, LWE-based schemes such as SimplePIR and DoublePIR [35], as well as FrodoPIR [24], have the server precompute a global hint that each client downloads once at onboarding, yielding high server throughput. A separate line of work instead attains sublinear server computation per online query [22, 23, 36, 55, 58]. In this line, however, the hint is client-specific rather than global: it cannot be precomputed solely by the server and shared across clients.

In either case, the hint is an offline cost amortized only over many queries, so an ephemeral client issuing few queries barely benefits. The burden is heaviest for the sublinear schemes above: in practice, their client-specific hints are built by having the client stream the entire database.[1] Furthermore, since the hint is database-dependent, every update makes it stale and entails per-client interaction and work, making the approach ill-suited to any frequently updated database.

Server-stateful PIR. Conversely, *server-stateful* schemes have the server store client-specific key material and reuse it across that client's queries. In particular, many PIR designs based on homomorphic encryption require a sizable server-side evaluation key, which the client is typically assumed to have uploaded in an offline phase [4, 13, 48, 51]. As in the client-stateful case, a client issuing few queries benefits little, while the server's storage grows with the client population. More fundamentally, per-client key material undermines anonymity: since each query must indicate which key material to use, all queries from one client are linkable to one another, which itself leaks metadata. Even atop a network-level anonymity tool such as Tor, the linkage persists, degrading the very privacy PIR is meant to provide.

Batch PIR. Orthogonally to statefulness, retrieving several entries from the same database at once is a common and useful pattern. This is the setting of *batch PIR* (a.k.a. multi-query PIR) [4, 38], which aims to beat the naive repetition of a single-query scheme by amortizing computation and communication across the batch. We highlight two specific applications. (1) Anonymous messaging systems [5, 30, 41, 44] often involve clients privately fetching multiple messages addressed to them through PIR. (2) Private Merkle proof retrieval [14] is itself a batch query, retrieving the sibling hashes along a root-to-leaf path in a Merkle tree. While sometimes a goal in its own right, as in Zcash, where a client must privately fetch

[1] Some works [29, 57] delegate this hint construction to the server via FHE, avoiding the client-side streaming, but remain expensive in practice.

the Merkle proof needed to spend its note [56], it is also a natural approach to PIR with *integrity* [40],[2] where the proof authenticates the retrieved record.

1.1 Our Contribution

In this work, we initiate the study of *doubly-stateless batch PIR*, i.e., batch PIR without any client or server state. By keeping neither party stateful, it sidesteps the practical drawbacks described above and can be deployed behind a clean, stateless API. There are two natural baselines, but both fall short.

The first is a trivial batch PIR that runs an existing doubly-stateless single-query scheme [26, 34, 42, 43, 47, 49] once for every query in the batch. Since no amortization is applied, the per-query cost remains that of a standalone single-query invocation. The second is to take an existing server-stateful batch PIR [4, 8, 13, 43, 52] and make it server-stateless by having the client resend its key material with each batch. Since this key material is uploaded once and reused across many queries, its size has drawn little attention and remains unoptimized, typically far larger than a query and its response. Resending it with every batch therefore inflates the upload.

To this end, we propose a new doubly-stateless batch PIR scheme. Observing that, in existing batch PIR schemes, the bulk of the key material is required for *unpacking* the queries, we design a *single-key unpacking* technique that drastically shrinks this key material. Around it, we assemble a complete scheme, revisiting each standard building block for the doubly-stateless setting. We detail both below.

Single-key RLWE unpacking. A central ingredient of our scheme is *unpacking* (a.k.a. *expansion*), a homomorphic transform first introduced in SealPIR [4]: it turns a single RLWE ciphertext, whose plaintext polynomial packs many values across its coefficients, into a collection of ciphertexts, one per coefficient. Packing values into a single ciphertext and later unpacking them is a versatile pattern that reaches well beyond the batch PIR we study: the same primitive underlies single-query PIR [48, 51], ORAM [15, 21], and oblivious message retrieval (OMR) [44, 45].

The standard way to unpack, from SealPIR [4], is *tree-based* (§3.1), recursively applying Galois automorphisms and needing $O(\log d)$ distinct keys for ring dimension d. A server-stateful scheme uploads this large key material once and reuses it, adding substantially to per-client storage, whereas the doubly-stateless setting keeps no such state, so it must be resent with every batch, dominating the upload.

We start from a simple algebraic observation (§3.2): unpacking recovers each coefficient by a *trace* over the ring's Galois group Gal, and a trace is *linear*, a sum of automorphisms. Stacking these traces over all coefficients, we cast unpacking as a single linear map on the automorphism images of the input, i.e., a matrix-vector multiplication. Restricted to a cyclic subgroup of Gal, the trace needs only the orbit of one generator, so a single Galois key suffices. Furthermore, the resulting matrix is DFT-like, so a Cooley–Tukey butterfly can evaluate it in $O(d \log d)$ ring operations.

Our approach does increase the output noise variance, by about a factor of d, and halves the per-ciphertext packing capacity. However, in the doubly-stateless setting, this is a highly favorable trade, since replacing the tree's $O(\log d)$ keys with one cuts the dominant upload while the added noise and halved capacity add only modest overhead. As unpacking recurs across the applications above, our single-key technique may be of independent interest.

Doubly-stateless batch PIR. With single-key unpacking as our main ingredient, we assemble a complete doubly-stateless batch PIR scheme (§4) from standard building blocks. We stress, though, that this is not a plug-and-play combination. These building blocks were designed for the server-stateful regime, so we reconsider whether each is worthwhile in the doubly-stateless setting before adopting it. Our evaluation (§6) shows that, with optimizations (§5), the scheme answers a batch of 32 queries on a 1 GB database of 64 B entries at an amortized per-query cost of 11.4 KB of communication. Against the two natural baselines, it cuts communication by more than 20× versus repeating a doubly-stateless single-query scheme, InsPIRe [47], and by 13× versus a server-stateful batch PIR scheme, Respire-B [13], with its key material resent per batch.

2 Preliminaries

2.1 Notations

For $n \in \mathbb{N}$ we write $[n] := \{0, 1, \ldots, n-1\}$. For a vector $\mathbf{v}$, $\mathbf{v}[i]$ denotes its i-th entry and $\mathbf{v}[i{:}j]$ its subvector on indices $i, \ldots, j-1$; similarly, for $x \in \mathbb{N}$, $x[i]$ its i-th bit in binary (with $x[0]$ the least significant), and for a polynomial $f(X)$, $f[i]$ its degree-i coefficient. We often write a vector by listing its entries as a family $(v_i)_{i\in[n]}$. For a power-of-two cyclotomic ring $\mathcal{R} = \mathbb{Z}[X]/(X^d+1)$, we denote its quotients $\mathcal{R}_q := \mathcal{R}/q\mathcal{R}$ and $\mathcal{R}_p := \mathcal{R}/p\mathcal{R}$. For $f(X) \in \mathcal{R}$, we define its Euclidean and infinity norms $\|f(X)\|_2 := \left(\sum_i f[i]^2\right)^{1/2}$ and $\|f(X)\|_\infty := \max_i |f[i]|$. For $f(X) \in \mathcal{R}_p$, we let $\hat{f}(X) \in \mathcal{R}$ denote its *balanced representative*, with coefficients in $[-p/2, p/2)$.

2.2 Batch Private Information Retrieval

We review the definition of batch PIR [38]. A batch PIR scheme is a tuple of algorithms $(\mathsf{Preprocess}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$, all taking the public parameters as an implicit input, which fix at least the database size N, the entry size w, and

[2] We note that this provides a weaker security guarantee than recent authenticated/verifiable PIR [6, 20, 25, 54], though it is often sufficient in practice.

the batch size n_{Q}. A raw database is then of the form $\mathsf{DB}_{\mathrm{raw}} \in (\{0,1\}^w)^N$, and a client queries a batch of n_{Q} distinct indices.

- $\mathsf{DB}_{\mathrm{prep}} \leftarrow \mathsf{Preprocess}(\mathsf{DB}_{\mathrm{raw}})$: the server preprocesses the raw database $\mathsf{DB}_{\mathrm{raw}}$ into $\mathsf{DB}_{\mathrm{prep}}$;
- $(\mathsf{qst}, \mathsf{qry}) \leftarrow \mathsf{Query}(\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1})$: the client generates a query qry for its batch of n_{Q} indices, along with a secret query state qst;
- $\mathsf{ans} \leftarrow \mathsf{Answer}(\mathsf{DB}_{\mathrm{prep}}, \mathsf{qry})$: the server answers the query qry using the preprocessed database $\mathsf{DB}_{\mathrm{prep}}$;
- $(\mathtt{m}_0, \ldots, \mathtt{m}_{n_{\mathsf{Q}}-1}) \leftarrow \mathsf{Extract}(\mathsf{qst}, \mathsf{ans})$: the client uses qst to recover the n_{Q} queried entries from ans.

Note that the syntax above already implies that the scheme is *doubly-stateless*: Query takes no database-dependent input other than the queried indices, and Answer takes no client-specific input other than the query qry. The algorithms should satisfy the following correctness and privacy.

- *Correctness:* For any raw database $\mathsf{DB}_{\mathrm{raw}} \in (\{0,1\}^w)^N$ and any batch of distinct indices $(\mathsf{idx}_i)_{i\in[n_{\mathsf{Q}}]} \in [N]^{n_{\mathsf{Q}}}$, let $\mathsf{DB}_{\mathrm{prep}} \leftarrow \mathsf{Preprocess}(\mathsf{DB}_{\mathrm{raw}})$, $(\mathsf{qst}, \mathsf{qry}) \leftarrow \mathsf{Query}(\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1})$, and $\mathsf{ans} \leftarrow \mathsf{Answer}(\mathsf{DB}_{\mathrm{prep}}, \mathsf{qry})$. Then, the following holds with high probability:

$$\mathsf{Extract}(\mathsf{qst}, \mathsf{ans}) = \left(\mathsf{DB}_{\mathrm{raw}}[\mathsf{idx}_i]\right)_{i\in[n_{\mathsf{Q}}]}.$$

- *Privacy:* For any two batches of distinct indices $(\mathsf{idx}_i)_{i\in[n_{\mathsf{Q}}]}, (\mathsf{idx}'_i)_{i\in[n_{\mathsf{Q}}]} \in [N]^{n_{\mathsf{Q}}}$, the resulting queries qry and qry$'$ are computationally indistinguishable, where $(\mathsf{qst}, \mathsf{qry}) \leftarrow \mathsf{Query}(\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1})$ and $(\mathsf{qst}', \mathsf{qry}') \leftarrow \mathsf{Query}(\mathsf{idx}'_0, \ldots, \mathsf{idx}'_{n_{\mathsf{Q}}-1})$.

2.3 Cuckoo-Hashing

Cuckoo-hashing [53] is a hashing scheme that stores each item in one of a few hash-chosen buckets, with at most one item per bucket. More precisely, the public parameters consist of the number of items n_{Q}, the number of buckets n_{B}, the number of hash functions n_{H} along with the functions $H_0, \ldots, H_{n_{\mathsf{H}}-1}$ mapping an item to a bucket in $[n_{\mathsf{B}}]$, and the maximum number of steps T per insertion. The n_{H} candidate buckets of an item x are $H_0(x), \ldots, H_{n_{\mathsf{H}}-1}(x)$. Given n_{Q} items, Cuckoo inserts them one at a time into their candidate buckets: an item takes a uniformly random free candidate if one exists, otherwise it evicts the occupant of a uniformly random candidate and re-inserts that occupant, terminating after at most T steps. An item still unplaced after T steps fails. The output is the placement $\psi : [n_{\mathsf{Q}}] \to [n_{\mathsf{B}}] \cup \{\perp\}$, sending each item to its bucket, or to $\perp$ if it failed.

2.4 RLWE and RGSW Encryptions

Ring LWE. We define the Ring Learning with Errors (RLWE) assumption [46] as follows. Given a power-of-two ring degree d, a modulus q, a secret-key distribution χ_{sk} over $\mathcal{R} = \mathbb{Z}[X]/(X^d+1)$, and an error distribution χ_{err} over $\mathcal{R}$, the $(d, q, \chi_{\mathsf{sk}}, \chi_{\mathsf{err}})$-RLWE assumption states that the following two distributions are computationally indistinguishable:

- $(a(X), a(X)\cdot s(X) + e(X))$, where $a(X) \xleftarrow{\$} \mathcal{R}_q$, $s(X) \xleftarrow{\$} \chi_{\mathsf{sk}}$, and $e(X) \xleftarrow{\$} \chi_{\mathsf{err}}$.
- $(a(X), u(X))$, where $a(X) \xleftarrow{\$} \mathcal{R}_q$ and $u(X) \xleftarrow{\$} \mathcal{R}_q$.

RLWE encryption. We define RLWE (noisy) encryption and decryption of a plaintext $\mu(X) \in \mathcal{R}$ under a secret key $s(X) \xleftarrow{\$} \chi_{\mathsf{sk}}$ as follows. Its security is based on the $(d, q, \chi_{\mathsf{sk}}, \chi_{\mathsf{err}})$-RLWE assumption.

- $\mathsf{RLWE.Enc}(s(X), \mu(X))$: Sample $a(X) \xleftarrow{\$} \mathcal{R}_q$, $e(X) \xleftarrow{\$} \chi_{\mathsf{err}}$. Compute $b(X) \leftarrow a(X)\cdot s(X) + \mu(X) + e(X)$ and output $c := (a(X), b(X))$.
- $\mathsf{RLWE.Dec}(s(X), c)$: For $c = (a(X), b(X))$, output $b(X) - a(X)\cdot s(X)$.

More generally, we call any pair $c = (a(X), b(X)) \in \mathcal{R}_q^2$ an RLWE ciphertext of $\mu(X)$ under $s(X)$ if

$$b(X) - a(X)\cdot s(X) = \mu(X) + e(X)$$

for some small $e(X)$, which we call the *noise* of c.

Note that the above scheme is noisy: decryption returns $\mu(X) + e(X)$ rather than $\mu(X)$. One can instead restrict the plaintext to $\mathcal{R}_p$ for some modulus p and scale it by $\Delta := \lfloor q/p \rfloor$, following the BFV encoding [11, 28]: encrypting $\mu(X) = \Delta \cdot \hat{m}(X)$ for the balanced representative $\hat{m}(X) \in \mathcal{R}$ (§2.1) of a message $m(X) \in \mathcal{R}_p$, one recovers $m(X)$ exactly from $\mu(X) + e(X)$ by the rounding $\lfloor (\mu(X) + e(X))/\Delta \rceil \bmod p$, which succeeds whenever $\|e(X)\|_\infty < \Delta/2$.

Noise model. We will apply homomorphic operations (§2.5, §2.6) to RLWE ciphertexts, each accumulating noise that must be tracked to ensure that results remain decryptable. We analyze this noise under the *independence heuristic* [32], standard in the homomorphic encryption literature: it treats each noise coefficient as an independent, zero-mean Gaussian, so that we track only the *per-coefficient noise variance*.

Plaintext-ciphertext multiplication. The following lemma bounds the noise growth of the plaintext-ciphertext multiplication pattern that appears in this work, where the RLWE ciphertext encrypts a Δ-scaled bit.

Lemma 1 (Plaintext-ciphertext multiplication)**.** *Under the independence heuristic (§2.4), let c be an* RLWE *ciphertext encrypting $\Delta \cdot m$ for a bit $m \in \{0,1\}$, with per-coefficient noise variance σ_c^2, and let $f(X) \in \mathcal{R}_p$ with balanced representative $\hat{f}(X) \in \mathcal{R}$ (§2.1). Then $\hat{f}(X) \cdot c \in \mathcal{R}_q^2$ encrypts $\Delta \cdot m \cdot \hat{f}(X)$ with per-coefficient noise variance at most $d \cdot p^2 \cdot \sigma_c^2/4$.*

Gadget decomposition. For a modulus q, an integer base $B \geq 2$, and length $\ell := \lceil \log_B q \rceil$, we define a *gadget vector* [50] as a row vector $\mathbf{g} := (1, B, \ldots, B^{\ell-1}) \in \mathcal{R}_q^{1\times\ell}$. A *gadget decomposition* $\mathbf{g}^{-1} : \mathcal{R}_q \to \mathcal{R}^\ell$ splits its input into ℓ ring elements, where the k-th element collects the k-th balanced base-B digit (each in $[-B/2, B/2)$) of every coefficient of the input. That is, $\mathbf{g} \cdot \mathbf{g}^{-1}(a(X)) = a(X)$ holds for every $a(X) \in \mathcal{R}_q$.

RGSW encryption. An RGSW [33] encryption of $\mu(X) \in \mathcal{R}$ under a secret $s(X)$, at modulus q with a gadget parameter (B, ℓ), is a matrix $C \in \mathcal{R}_q^{2\times 2\ell}$ whose 2ℓ columns are RLWE ciphertexts under $s(X)$: the first ℓ encrypt $-B^k \cdot \mu(X) \cdot s(X)$ and the last ℓ encrypt $B^k \cdot \mu(X)$, for $k \in [\ell]$. In our construction, such ciphertexts are produced by scheme-switching (§2.5) and consumed by the *external product* (§2.6), which incurs only additive noise growth.

2.5 Automorphisms and Scheme-switching

Key-switching. Given suitable key material, *key-switching* transforms an RLWE ciphertext c encrypting $\mu(X)$ under a source key $s_1(X)$ into another RLWE ciphertext c' encrypting the same plaintext under a target key $s_2(X)$. For a gadget parameter (B, ℓ), we define:

- $\mathsf{KSKGen}(s_1(X), s_2(X))$: Output $\mathsf{ksk} \in \mathcal{R}_q^{2\times\ell}$ whose k-th column, for $k \in [\ell]$, is

$$\mathsf{ksk}[k] \leftarrow \mathsf{RLWE.Enc}(s_2(X), B^k \cdot s_1(X)).$$

- $\mathsf{KeySwitch}(\mathsf{ksk}, c)$: For $c = (a(X), b(X))$, output

$$c' \leftarrow (0, b(X)) - \mathsf{ksk} \cdot \mathbf{g}^{-1}(a(X)).$$

Lemma 2 (Key-switching)**.** *Let* $\mathsf{ksk} \leftarrow \mathsf{KSKGen}(s_1(X), s_2(X))$ *with errors drawn from χ_{err} of variance σ_{err}^2. Under the independence heuristic (§2.4), for any $a(X) \in \mathcal{R}_q$ the product $\mathsf{ksk} \cdot \mathbf{g}^{-1}(a(X))$ is an* RLWE *encryption of $a(X) \cdot s_1(X)$ under $s_2(X)$ with per-coefficient noise variance at most σ_{KS}^2, where*

$$\sigma_{\mathsf{KS}}^2 := \frac{B^2 \ell d\, \sigma_{\mathsf{err}}^2}{4}.$$

Consequently, $\mathsf{KeySwitch}(\mathsf{ksk}, c)$ *increases the per-coefficient noise variance of c by at most σ_{KS}^2.*

Homomorphic automorphism. The power-of-two cyclotomic ring $\mathcal{R} = \mathbb{Z}[X]/(X^d + 1)$ has a group of automorphisms Gal of order d, consisting of $\tau_i : f(X) \mapsto f(X^i)$ for $i \in \mathbb{Z}_{2d}^\times$. Our main technique, namely single-key unpacking (§3), rests on the fact that, for $d \geq 8$, τ_g with $g = 5$ generates an order-$(d/2)$ cyclic subgroup of Gal [37, §4.1].

These automorphisms can be evaluated homomorphically using key-switching. More precisely, given a *Galois key* for τ_i, $\mathsf{gk}_i \leftarrow \mathsf{KSKGen}(\tau_i(s(X)), s(X))$, one can evaluate τ_i on an RLWE ciphertext c encrypting $\mu(X)$ to obtain an RLWE ciphertext encrypting $\tau_i(\mu(X))$ as follows:

- $\mathsf{Galois}(\mathsf{gk}_i, c)$: For $c = (a(X), b(X))$, output

$$\mathsf{KeySwitch}\big(\mathsf{gk}_i, (\tau_i(a(X)), \tau_i(b(X)))\big).$$

Since τ_i is a ring homomorphism, $(\tau_i(a(X)), \tau_i(b(X)))$ encrypts $\tau_i(\mu(X))$ under $\tau_i(s(X))$, which the key-switching brings back to $s(X)$, at the cost of σ_{KS}^2 in noise (Lemma 2).

RLWE-to-RGSW Scheme-switching. Given a *scheme-switching key* $\mathsf{ssk} \leftarrow \mathsf{KSKGen}(s(X)^2, s(X))$, one can convert RLWE ciphertexts $\mathbf{c} \in (\mathcal{R}_q^2)^\ell$ under a secret $s(X)$, where $\mathbf{c}[k]$ encrypts $B^k \cdot \mu(X)$ for $k \in [\ell]$, into an RGSW ciphertext encrypting $\mu(X)$ under the same key [27]:

- $\mathsf{SchemeSwitch}(\mathsf{ssk}, \mathbf{c})$: Writing $\mathbf{c}[k] = (a_k(X), b_k(X))$, output $C \in \mathcal{R}_q^{2\times 2\ell}$ whose columns, for each $k \in [\ell]$, are

$$C[k] \leftarrow (b_k(X), 0) + \mathsf{ssk} \cdot \mathbf{g}^{-1}(a_k(X)) \quad \text{(left half)},$$
$$C[\ell + k] \leftarrow \mathbf{c}[k] \quad \text{(right half)}.$$

Lemma 3 (Scheme-switching)**.** *With* ssk *and* $\mathbf{c}$ *as above, write σ_{in}^2 for the maximal per-coefficient noise variance among the $\mathbf{c}[k]$, with σ_{KS}^2 as in Lemma 2. Then, under the independence heuristic (§2.4), $C = \mathsf{SchemeSwitch}(\mathsf{ssk}, \mathbf{c})$ has per-coefficient noise variance at most*

- $\sigma_{\mathsf{L}}^2 := \|s(X)\|_2^2 \cdot \sigma_{\mathsf{in}}^2 + \sigma_{\mathsf{KS}}^2$ *in the left half,*
- $\sigma_{\mathsf{R}}^2 := \sigma_{\mathsf{in}}^2$ *in the right half.*

2.6 External Product and Multiplexers

External product. Given an RGSW ciphertext C encrypting $\beta \in \{0,1\}$ and an RLWE ciphertext $c = (a(X), b(X))$ encrypting $\mu(X)$, both under a secret $s(X)$, their *external product* [17] is denoted and defined as

$$C \boxdot c := C \cdot \big(\mathbf{g}^{-1}(a(X)),\ \mathbf{g}^{-1}(b(X))\big) \in \mathcal{R}_q^2,$$

an RLWE ciphertext under $s(X)$ encrypting $\beta \cdot \mu(X)$.

Lemma 4 (External product)**.** *With C and c as above, suppose C has per-coefficient noise variance σ_{L}^2 in its left half and σ_{R}^2 in its right half, and c has per-coefficient noise variance σ_c^2.*

Algorithm 1 $\mathsf{DMuxTree}(\mathbf{C}, c)$

Input: RGSW ciphertexts $\mathbf{C} \in (\mathcal{R}_q^{2\times 2\ell})^{\nu}$, each $\mathbf{C}[t]$ encrypting $\beta_t \in \{0,1\}$; RLWE ciphertext $c \in \mathcal{R}_q^2$.

Output: RLWE ciphertexts $\mathbf{c} \in (\mathcal{R}_q^2)^{2^\nu}$, where $\mathbf{c}[\sum_t \beta_t \cdot 2^t]$ encrypts the plaintext of c and the others zero.

$\mathbf{c} \leftarrow (c)$
for $t \in [\nu]$ **do**
  for $j \in [2^t]$ **do**
    $(\mathbf{c}'[j], \mathbf{c}'[j+2^t]) \leftarrow \mathsf{DMux}(\mathbf{C}[t], \mathbf{c}[j])$
  $\mathbf{c} \leftarrow \mathbf{c}'$
return $\mathbf{c}$.

Then, under the independence heuristic (§2.4), $C \boxdot c$ has per-coefficient noise variance at most $\beta \cdot \sigma_c^2 + \sigma_{\boxdot}^2$, where

$$\sigma_{\boxdot}^2 := \frac{B^2 \ell d}{4}\left(\sigma_{\mathsf{L}}^2 + \sigma_{\mathsf{R}}^2\right).$$

DMux. From the external product, we can build the *demultiplexer* [43], which routes an RLWE ciphertext c encrypting $\mu(X)$ to one of two output positions, controlled by an RGSW ciphertext C encrypting a bit $\beta \in \{0,1\}$:

$$\mathsf{DMux}(C, c) := \left(c - C \boxdot c, C \boxdot c\right).$$

This produces two RLWE ciphertexts, such that the ciphertext at position β encrypts $\mu(X)$ and the other is an encryption of zero. By iterating this over ν selector RGSW ciphertexts (Alg. 1, DMuxTree), we can route an input RLWE ciphertext c to one of 2^ν output positions.

Lemma 5 (DMuxTree)**.** *For a ν-level* DMuxTree *(Alg. 1), let σ_{in}^2 be the per-coefficient noise variance of its input* RLWE *ciphertext, with $\sigma_{\boxdot}^2$ as in Lemma 4. Then, under the independence heuristic (§2.4), every output ciphertext carries per-coefficient noise variance at most $\sigma_{\mathsf{in}}^2 + \nu \cdot \sigma_{\boxdot}^2$.*

CMux. Similarly, the *controlled multiplexer* [17] selects one of two RLWE ciphertexts c_0 and c_1, controlled by an RGSW ciphertext C encrypting a bit $\beta \in \{0,1\}$:

$$\mathsf{CMux}(C, c_0, c_1) := c_0 + C \boxdot (c_1 - c_0).$$

This produces an RLWE ciphertext encrypting the same plaintext as c_β, using a single external product. By iterating this over ν selector RGSW ciphertexts (Alg. 2, CMuxTree), we can select, from 2^ν RLWE ciphertexts $\mathbf{c} \in (\mathcal{R}_q^2)^{2^\nu}$, the one at the index determined by the selector bits.

Lemma 6 (CMuxTree)**.** *For a ν-level* CMuxTree *(Alg. 2), let σ_{in}^2 be the maximal per-coefficient noise variance among its input* RLWE *ciphertexts, with $\sigma_{\boxdot}^2$ as in Lemma 4. Then, under the independence heuristic (§2.4), the output ciphertext carries per-coefficient noise variance at most $\sigma_{\mathsf{in}}^2 + \nu \cdot \sigma_{\boxdot}^2$.*

Algorithm 2 $\mathsf{CMuxTree}(\mathbf{C}, \mathbf{c})$

Input: RGSW ciphertexts $\mathbf{C} \in (\mathcal{R}_q^{2\times 2\ell})^{\nu}$, each $\mathbf{C}[t]$ encrypting $\beta_t \in \{0,1\}$; RLWE ciphertexts $\mathbf{c} \in (\mathcal{R}_q^2)^{2^\nu}$.

Output: RLWE ciphertext with the plaintext of $\mathbf{c}[\sum_t \beta_t \cdot 2^t]$.

for $t \in [\nu]$ **do**
  for $j \in [2^{\nu-1-t}]$ **do**
    $\mathbf{c}'[j] \leftarrow \mathsf{CMux}\big(\mathbf{C}[t], \mathbf{c}[2j], \mathbf{c}[2j+1]\big)$
  $\mathbf{c} \leftarrow \mathbf{c}'$
return $\mathbf{c}[0]$.

3 RLWE Unpacking

RLWE unpacking transforms a single RLWE ciphertext encrypting $f(X) \in \mathcal{R}$ into a collection of ciphertexts, each encrypting a single coefficient of $f(X)$. We formulate unpacking algebraically below and review the standard tree-based method (§3.1). Our main contribution is a new method that requires only a single Galois key (§3.2). We compare its cost with that of the tree-based approach in §3.3.

Algebraic view. Recall that $\mathrm{Gal} \cong \mathbb{Z}_{2d}^\times$ consists of the automorphisms $\tau_i : X \mapsto X^i$ (§2.5). The trace $\mathrm{Tr}(\cdot) := \sum_{\tau_i \in \mathrm{Gal}} \tau_i(\cdot)$ is fixed by every automorphism (any τ_i merely reindexes the sum by $\tau_i \circ \tau$), and the constants are the only such polynomials, so $\mathrm{Tr}(\cdot)$ returns a constant. Since each of the d automorphisms fixes the constant term,

$$\mathrm{Tr}(f(X)) = d \cdot f[0].$$

Applied to the shift $X^{-k} \cdot f(X)$, whose constant term is $f[k]$,

$$\mathrm{Tr}\big(X^{-k} \cdot f(X)\big) = d \cdot f[k]. \tag{1}$$

That is, one route to RLWE unpacking is *homomorphic trace evaluation* of $X^{-k} \cdot f(X)$ for every k, up to the factor d.[3]

3.1 Review: SealPIR's Tree-based Unpacking

SealPIR's *tree-based unpacking* (a.k.a. expansion) [4] is based on the following factorization of the trace over $\mathcal{R}$:

$$\mathrm{Tr} = (\mathrm{id} + \tau_3) \circ \cdots \circ (\mathrm{id} + \tau_{d/2^j+1}) \circ \cdots \circ (\mathrm{id} + \tau_{d+1}). \tag{2}$$

To see why, follow a monomial X^i through the product. Since $X^d = -1$, the automorphism τ_{d+1} sends $X^i \mapsto (-1)^i X^i$, so $(\mathrm{id} + \tau_{d+1})$ doubles X^i when i is even and cancels it otherwise. The next automorphism $\tau_{d/2+1}$ acts the same way one level down, sending an even X^i to $(-1)^{i/2} X^i$, so $(\mathrm{id} + \tau_{d/2+1})$ doubles X^i when i is a multiple of 4 and cancels it otherwise. Each later factor doubles the terms of degree a multiple of

[3] We later absorb this scaling factor into pre-scaling (§4.3).

the next power of 2, so after all $\log d$ factors only the constant term survives, doubled $\log d$ times and hence scaled by d.

Let φ_k denote multiplication by X^{-k}. The full shift φ_k decomposes into its per-bit pieces:

$$\varphi_k = \varphi_{k[\log d-1]\cdot d/2} \circ \cdots \circ \varphi_{k[i]\cdot 2^i} \circ \cdots \circ \varphi_{k[0]}.$$

We thread these shifts into the trace factorization (2). Abbreviating $\tilde{\tau}_a := (\mathrm{id} + \tau_a)$, the operator that produces the k-th output is

$$\begin{aligned}\mathrm{Tr} \circ \varphi_k &= (\tilde{\tau}_3 \circ \cdots \circ \tilde{\tau}_{d+1}) \circ (\varphi_{k[\log d-1]\cdot d/2} \circ \cdots \circ \varphi_{k[0]}) \\ &= (\tilde{\tau}_3 \circ \varphi_{k[\log d-1]\cdot d/2}) \circ \cdots \circ (\tilde{\tau}_{d+1} \circ \varphi_{k[0]}).\end{aligned}$$

The second equality holds because, for $j < i$, the shift $\varphi_{k[i]\cdot 2^i}$ commutes with the factor $(\mathrm{id} + \tau_{d/2^j+1})$: the automorphism $\tau_{d/2^j+1}$ fixes every monomial of degree a multiple of 2^{j+1}, and 2^i is such a multiple.

Since each factor depends only on bit $k[i]$, outputs that agree on their low bits share a suffix of this factorization. A binary-tree structure can exploit this, reusing the shared computation rather than evaluating $\mathrm{Tr}(\cdot)$ separately.

Cost analysis. The factorization uses the $\log d$ distinct automorphisms, so a straightforward implementation of tree-based unpacking needs $\log d$ Galois keys. Each internal node performs one key-switching, so unpacking needs $d-1$ key-switchings in total. The additions and monomial multiplications total $O(d)$.

Lemma 7 (Tree-based unpacking)**.** *Write σ^2_{KS} for the per-switch noise variance (Lemma 2). Under the independence heuristic (§2.4), tree-based unpacking of an input with per-coefficient noise variance σ^2_{in} outputs ciphertexts with per-coefficient noise variance at most*

$$d \cdot \sigma^2_{\mathsf{in}} + (d-1) \cdot \sigma^2_{\mathsf{KS}}.$$

Proof. Write V_r for the per-coefficient noise variance after r rounds. Each round doubles the accumulated variance through $(\mathrm{id} + \tau)$ and adds one key-switching of variance at most σ^2_{KS}, so $V_{r+1} \le 2V_r + \sigma^2_{\mathsf{KS}}$. With $V_0 = \sigma^2_{\mathsf{in}}$, unrolling gives $V_r \le 2^r \cdot \sigma^2_{\mathsf{in}} + (2^r - 1) \cdot \sigma^2_{\mathsf{KS}}$, so after all $\log d$ rounds the variance is at most $d \cdot \sigma^2_{\mathsf{in}} + (d-1) \cdot \sigma^2_{\mathsf{KS}}$. □

3.2 Our Single-key Unpacking

Key idea. Tree-based unpacking evaluates the trace through its factorization into a *composition* of automorphisms (2). Instead, we return to its defining *sum* over the automorphism group, $\mathrm{Tr}(\cdot) = \sum_{\tau_i \in \mathrm{Gal}} \tau_i(\cdot)$. Each τ_i is a ring homomorphism with $\tau_i(X^{-k}) = X^{-k\cdot i}$, so

$$\begin{aligned}\mathrm{Tr}\big(X^{-k} \cdot f(X)\big) &= \sum_{\tau_i \in \mathrm{Gal}} X^{-k\cdot i} \cdot \tau_i(f(X)) \\ &= \big\langle (X^{-k\cdot i})_{\tau_i \in \mathrm{Gal}}, (\tau_i(f(X)))_{\tau_i \in \mathrm{Gal}} \big\rangle.\end{aligned}$$

Plugging this into (1) and stacking over all $k \in [d]$ gives a representation of unpacking as a matrix-vector multiplication,

$$\big(d \cdot f[k]\big)_{k\in[d]} = M \cdot \big(\tau_i(f(X))\big)_{\tau_i \in \mathrm{Gal}}. \tag{3}$$

Note that M is DFT-like: its entries are powers of X, a $2d$-th root of unity in $\mathcal{R}$, which we exploit for fast evaluation.

Trace over a cyclic subgroup. Evaluating (3) requires the orbit $\{\tau_i(f(X))\}_i$. Generating this orbit homomorphically takes at least two Galois keys, since $\mathbb{Z}^\times_{2d} \cong \mathbb{Z}_2 \times \mathbb{Z}_{d/2}$ needs two generators and no single automorphism generates Gal. A single key does, however, generate a cyclic subgroup of index two. We therefore use a single-key variant of (3) with the trace over the cyclic subgroup $\langle \tau_g \rangle$ where $g = 5$ (§2.5):

$$\mathrm{Tr}_{\langle \tau_g \rangle}(\cdot) = \sum_{i=0}^{d/2-1} \tau_g^i(\cdot).$$

The price of a single Galois key is that this trace retains two of the d coefficients rather than one, at positions 0 and $d/2$, since $\langle \tau_g \rangle$ fixes the two monomials 1 and $X^{d/2}$ (Lemma 8). We therefore pack only the lower half, where the partial trace cleanly extracts a single coefficient, and restrict inputs to

$$\mathcal{R}_q^{\mathrm{low}} := \mathrm{span}_{\mathbb{Z}_q}\{X^j : 0 \le j < d/2\}.$$

Lemma 8 (Partial trace)**.** *For every $f(X) \in \mathcal{R}_q^{\mathrm{low}}$ and $0 \le k < d/2$,*

$$\mathrm{Tr}_{\langle \tau_g \rangle}\big(X^{-k} \cdot f(X)\big) = \tfrac{d}{2} \cdot f[k].$$

Proof. The generator τ_g sends $X^j \mapsto X^{j\cdot g}$, so $\langle \tau_g \rangle$ fixes X^j iff $(g-1)\cdot j \equiv 0 \pmod{2d}$. With $g = 5$, the only fixed monomials are 1 and $X^{d/2}$. Since $\mathrm{Tr}_{\langle \tau_g \rangle}(\cdot)$ is τ_g-invariant, it is supported only on these two degrees. Moreover, each τ_g^i preserves the corresponding coefficients, so the trace returns $d/2$ times the degree-0 and degree-$(d/2)$ coefficients of its input. For $f(X) \in \mathcal{R}_q^{\mathrm{low}}$ and $0 \le k < d/2$, the exponents of $X^{-k} \cdot f(X)$ lie in $(-d/2, d/2)$, so its degree-0 coefficient is $f[k]$ and its degree-$(d/2)$ coefficient vanishes, giving $\mathrm{Tr}_{\langle \tau_g \rangle}(X^{-k} \cdot f(X)) = \frac{d}{2} \cdot f[k]$. □

The full algorithm. Putting the preceding components together yields Unpack (Alg. 3). It proceeds in two stages. First, the *orbit generation* stage repeatedly applies τ_g using the single key gk_g, producing $\mathbf{c} \in (\mathcal{R}_q^2)^{d/2}$ such that $\mathbf{c}[i]$ encrypts $\tau_{g^i}(f(X))$. Second, the *linear map evaluation* stage combines these ciphertexts as

$$\mathbf{c}_{\mathsf{out}}[k] = \sum_{i=0}^{d/2-1} X^{-k\cdot g^i} \cdot \mathbf{c}[i], \tag{4}$$

where the monomial $X^{-k\cdot g^i}$ equals $\tau_{g^i}(X^{-k})$, so $\mathbf{c}_{\mathsf{out}}[k] = \mathrm{Tr}_{\langle \tau_g \rangle}(X^{-k} \cdot f(X))$ decrypts to $(d/2) \cdot f[k]$ by Lemma 8. Equivalently, this evaluates the matrix M from (3) restricted to $\langle \tau_g \rangle$, giving a $(d/2) \times (d/2)$ transform.

Algorithm 3 $\mathsf{Unpack}(\mathsf{gk}_g, c_{\mathsf{in}})$

Input: Galois key gk_g (§2.5); ciphertext $c_{\mathsf{in}} \in \mathcal{R}_q^2$ encrypting $f(X) \in \mathcal{R}_q^{\mathrm{low}}$.

Output: $\mathbf{c}_{\mathsf{out}} \in (\mathcal{R}_q^2)^{d/2}$, each $\mathbf{c}_{\mathsf{out}}[k]$ encrypts $(d/2) \cdot f[k]$.

// Stage 1: Orbit generation
$\mathbf{c}[0] \leftarrow c_{\mathsf{in}}$
for $j \in [d/2-1]$ **do**
  $\mathbf{c}[j+1] \leftarrow \mathsf{Galois}(\mathsf{gk}_g, \mathbf{c}[j])$

// Stage 2: Linear map evaluation
for $r \in [\log(d/2)]$ **do**
  $\mathrm{len} \leftarrow (d/2)/2^r$
  for $i \in [2^r]$ **do**
    $W \leftarrow X^{-2^r}$
    for $j \in [\mathrm{len}/2]$ **do**
      $t \leftarrow i \cdot \mathrm{len} + j$
      $(U, V) \leftarrow (\mathbf{c}[t], \mathbf{c}[t + \mathrm{len}/2])$
      $(\mathbf{c}[t], \mathbf{c}[t + \mathrm{len}/2]) \leftarrow (U + V, W \cdot (U - V))$
      $W \leftarrow W^g$
for $k \in [d/2]$ **do**
  $\mathbf{c}_{\mathsf{out}}[k] \leftarrow \mathbf{c}[\mathsf{bitrev}_{\log(d/2)}(k)]$
return $\mathbf{c}_{\mathsf{out}}$.

A direct evaluation of this linear map costs $O(d^2)$ ring operations. However, the transform admits a radix-2 Cooley–Tukey evaluation: since $g^{d/4} \equiv d+1 \pmod{2d}$, we have $X^{-g^{i+d/4}} = -X^{-g^i}$. The resulting butterfly algorithm (Alg. 3), using only monomial twiddles, evaluates the transform in $O(d \log d)$ ring operations.

Cost analysis. Unpacking uses a single Galois key throughout. Its only key-switchings are the $d/2-1$ applications of τ_g in orbit generation. Evaluating the linear map takes $O(d \log d)$ ring operations across $\log(d/2)$ rounds.

Lemma 9 (Single-key unpacking)**.** *Write σ^2_{KS} for the per-switch noise variance of gk_g (Lemma 2). Under the independence heuristic (§2.4), Unpack (Alg. 3) maps an input with per-coefficient noise variance σ^2_{in} to output ciphertexts with per-coefficient noise variance at most σ^2_{unp}, where*

$$\sigma^2_{\mathsf{unp}} := \tfrac{d}{2} \cdot \sigma^2_{\mathsf{in}} + \tfrac{d(d-2)}{8} \cdot \sigma^2_{\mathsf{KS}}.$$

Proof. Each output is the sum $\mathbf{c}_{\mathsf{out}}[k] = \sum_{i=0}^{d/2-1} X^{-k \cdot g^i} \cdot \mathbf{c}[i]$ of (4), in which each $\mathbf{c}[i]$ appears once. The butterfly only re-associates this sum, so its output equals (4) and carries the same noise. The sweep produces $\mathbf{c}[i]$ by i key-switchings under gk_g, each adding fresh error of variance at most σ^2_{KS}, so $\mathbf{c}[i]$ has variance at most $\sigma^2_{\mathsf{in}} + i \cdot \sigma^2_{\mathsf{KS}}$. Since multiplying by a power of X does not amplify the noise, treating the $d/2$ summands as independent gives output variance at most

$$\sum_{i=0}^{d/2-1} \left(\sigma^2_{\mathsf{in}} + i \cdot \sigma^2_{\mathsf{KS}}\right) = \tfrac{d}{2} \cdot \sigma^2_{\mathsf{in}} + \tfrac{d(d-2)}{8} \cdot \sigma^2_{\mathsf{KS}}. \qquad \square$$

Table 1: Per-ciphertext cost of the two unpacking methods.

	Tree-based [4]	Ours
#(Galois Key)	$\log d$	1
#(Packed coeff.)	d	$d/2$
#(Key-switching)	$d-1$	$d/2-1$
Output noise variance	$O(d) \cdot \sigma^2_{\mathsf{KS}}$	$O(d^2) \cdot \sigma^2_{\mathsf{KS}}$

3.3 Comparison

Table 1 compares tree-based unpacking (§3.1) with our single-key unpacking (§3.2) on a single packed ciphertext. Tree-based unpacking needs $\log d$ Galois keys, whereas ours needs only a single key. In return, ours unpacks only the lower $d/2$ coefficients of a ciphertext rather than all d, so a given payload occupies twice as many ciphertexts. Key-switchings dominate the running time, and both spend about one per unpacked coefficient; the remaining ring operations ($O(d)$ for the tree, $O(d \log d)$ for our butterfly) are monomial multiplications and additions, negligible beside them. Finally, the output noise variance is $O(d^2)$ for ours against $O(d)$ for the tree (Lemmas 7 and 9). This trade-off is favorable whenever key material outweighs ciphertext material.

4 Doubly-Stateless Batch PIR

In this section, we present our doubly-stateless batch PIR scheme. The construction largely follows the established framework for (server-stateful) batch PIR schemes developed in SealPIR [4], Respire [13], and VIA-B [43]. In the doubly-stateless setting, however, key size becomes the primary design constraint: each query must carry all evaluation keys required by the underlying HE scheme, and this key material can easily dominate communication. The most consequential choice is the single-key RLWE unpacking technique in §3.2, which expands a query using a single Galois key rather than the logarithmically many keys required by the prior expansion method in SealPIR. The remaining techniques are adopted selectively from Respire and VIA-B, with some deliberately omitted when their key-size cost outweighs their benefit.

Cuckoo-hashing layer. At the outermost level, the framework reduces batch PIR to multiple single-query PIR instances using SealPIR's probabilistic batch code approach [4], instantiated via cuckoo-hashing (§2.3).[4] When preprocessing the database, the server replicates each entry into all of

[4] We note that VIA-B [43] omits SealPIR's cuckoo-hashing layer.

Table 2: Public parameters.

N	the number of database entries
w	the bit-width of each entry
n_{Q}	the batch size (number of queries)
d	ring degree, a power of two
q	ciphertext modulus
p	plaintext modulus
χ_{sk}	the secret-key distribution
χ_{err}	the error distribution
Δ	the scaling factor $\lfloor q/p \rfloor$
B	the gadget base
ℓ	the gadget length
n_{B}	the number of buckets
n_{H}	the number of hash functions
H_i	public hash functions $[N] \to [n_{\mathsf{B}}]$, $i \in [n_{\mathsf{H}}]$
N'	bucket capacity, a power of two, $N' = 2^{\nu_0+\nu_1+\nu_2}$
ν_0, ν_1	define the plaintext matrix layout: $2^{\nu_0} \times 2^{\nu_1}$
ν_2	defines the number of entries per plaintext: 2^{ν_2}
n_{C}	the number of packed query ciphertexts, $n_{\mathsf{C}} = \lceil n_{\mathsf{B}} \ell(\nu_0+\nu_1+\nu_2)/(d/2) \rceil$

its candidate buckets, one per hash function. The client then cuckoo-hashes its target indices into the buckets so that each bucket holds at most one target index; since every entry sits in all of its candidate buckets, the bucket a query lands in already holds the entry it seeks. Client and server then run single-query PIR per bucket, all in parallel. Running PIR per bucket over these small buckets is usually cheaper than the naive alternative of running full-database PIR per query.

Public parameters. The construction is parameterized by public parameters, taken implicitly as input by all algorithms in this section. Table 2 lists these parameters for reference.

Scope. For clarity, here we describe a streamlined version of the scheme. Taken literally, this version is not concretely competitive; §5 describes the remaining techniques that close the gap. For the fully detailed optimized construction, we refer the reader to our implementation.

4.1 Plaintext Encoding

We assume an unspecified embedding $\mathsf{emb} : \{0,1\}^w \to \mathbb{Z}_p^{d/2^{\nu_2}}$. A single plaintext then holds 2^{ν_2} entries, encoded by Ecd and recovered one at a time by Dcd. $\mathsf{Ecd}(\mathtt{m}_0, \ldots, \mathtt{m}_{2^{\nu_2}-1})$, with each $\mathtt{m}_i \in \{0,1\}^w$, interleaves the 2^{ν_2} embeddings into a plaintext $f(X) \in \mathcal{R}_p$, placing the coefficients of the i-th entry at the degrees congruent to i modulo 2^{ν_2}:

$$f[2^{\nu_2} j + i] = \mathsf{emb}(\mathtt{m}_i)[j], \qquad i \in [2^{\nu_2}],\ j \in [d/2^{\nu_2}].$$

Algorithm 4 $\mathsf{Preprocess}(\mathsf{DB}_{\mathrm{raw}})$

Input: raw database $\mathsf{DB}_{\mathrm{raw}} \in (\{0,1\}^w)^N$.
Output: preprocessed database $\mathsf{DB}_{\mathrm{prep}}$.

```
// Stage 1: Bucket assignment
for β ∈ [n_B] do
    DB_raw^(β) ← ()
for i ∈ [N], j ∈ [n_H] do
    β ← H_j(i)
    DB_raw^(β) ← DB_raw^(β) ‖ DB_raw[i]
// Stage 2: Encoding
for β ∈ [n_B] do
    DB_raw^(β) ← DB_raw^(β) ‖ (0^w)^{N' − |DB_raw^(β)|}
    for u ∈ [2^{ν0+ν1}] do
        f_u(X) ← Ecd((DB_raw^(β)[u + v·2^{ν0+ν1}])_{v∈[2^{ν2}]})
    for i ∈ [2^{ν0}], j ∈ [2^{ν1}] do
        DB_prep^(β)[i, j] ← f̂_{i+j·2^{ν0}}(X)
DB_prep ← (DB_prep^(β))_{β∈[n_B]}
return DB_prep.
```

$\mathsf{Dcd}(f(X), i)$ inverts Ecd: it reads off $(f[2^{\nu_2} j + i])_{j \in [d/2^{\nu_2}]}$ and applies emb^{-1} to recover the i-th entry.

4.2 Preprocess

Preprocess (Alg. 4) proceeds in two stages. First, bucket assignment: each of the N entries is replicated into all n_{H} of its candidate buckets. Entries are processed in order, and hash indices are processed in order within each entry, yielding a fixed public order that the client replays in §4.3. Second, encoding: each bucket is padded to the common size N' with all-zero entries (see Remark 1) and encoded with Ecd (§4.1) into a plaintext matrix.

Remark 1 (Bucket overflow)**.** *Buckets exceeding the default capacity N' do not cause failure: the server resizes them to the next power of two. Since bucket loads are determined by N and the public hashes, the client can detect overflows locally (see Remark 2) and apply the same adjustment.*

4.3 Query

Query (Alg. 5) proceeds in the three stages described below. Note that the output qry includes the evaluation keys.

1. *Bucket assignment.* This stage places the n_{Q} query indices into buckets and recovers each query's *within-bucket index* (Remark 2). Cuckoo-hashing (§2.3) assigns at most one query to each bucket; the assignment failure is discussed in §4.6. Replaying the bucket assignment from Preprocess (§4.2) then identifies the within-bucket

index $\mathsf{idx}^{(\beta)} \in [N']$ for each placement, with unused buckets assigned the dummy index 0.

2. *Key generation.* This stage draws a fresh per-batch secret and the key-switching keys (§2.5) that drive the server's pipeline: gk_g is for Unpack (§3.2), ssk for SchemeSwitch (§2.5), and the projection keys pjk_i for the response-packing stage (§4.4).

3. *Encryption.* This stage encodes and encrypts the within-bucket indices into packed query ciphertexts. Each of the n_{B} indices $\mathsf{idx}^{(\beta)}$ has $\log N'$ index bits $\mathsf{idx}^{(\beta)}[t]$, and each bit is encoded as the ℓ gadget multiples $B^j \cdot \mathsf{idx}^{(\beta)}[t]$. These multiples are pre-scaled by $(d/2)^{-1} \bmod q$ to cancel the $d/2$ factor that Unpack (§3.2, §4.4) introduces. They are then packed as the coefficients of plaintext polynomials and encrypted as RLWE ciphertexts. Only the lower-half coefficients (degree $< d/2$) are utilized, matching the subspace that Unpack acts on. Two additional notes for comparison with prior work:

   - Every index bit is encoded bit-wise, following VIA-B [43], and expanded server-side (§4.4), whereas Respire [13] and OnionPIR [16, 51] transmit the low-order bits as a one-hot vector. The bit-wise encoding is smaller, favoring communication.
   - Unlike VIA-B, which encrypts the bits as LWE ciphertexts and then *ring-packs* them [7], we encrypt directly into RLWE: ring-packing requires key material that, in our setting, outweighs any saving.

Remark 2 (Linear-time within-bucket indexing)**.** *After* Cuckoo *(§2.3) assigns each query to a bucket, the client still needs the entry's within-bucket index to run the per-bucket PIR. Stage 1 of* Query *recovers this index, replaying* Preprocess*'s bucket assignment over all N entries. This replay makes* Query *linear in N, but this cost is small in practice (see Remark 4): it is only hashing and counting with n_{B} memory, and involves no cryptographic work. Alternatively, an implementation can instantiate the per-bucket PIR as a keyword PIR rather than an index PIR, using the generic cuckoo-hashing-based conversion of [2]. This eliminates the replay step entirely and is worth considering when client-side cost is the primary optimization target. See [4, §5] for a broader discussion of how clients can recover within-bucket indices.*

4.4 Answer

At the framework level, Answer (Alg. 6) follows the structure of the batch version of Respire [13], which in turn builds on the single-query PIR approach of OnionPIR [16, 51]. OnionPIR expands a compact query into an encrypted one-hot selection vector together with bit encryptions for a CMux tree (§2.6). The one-hot vector is multiplied by the database matrix, and the resulting products are compressed by the tree.

Algorithm 5 $\mathsf{Query}(\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1})$

Input: distinct indices $\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1} \in [N]$.
Output: query state qst; query qry.

// Stage 1: Bucket assignment
$\psi \leftarrow \mathsf{Cuckoo}(\mathsf{idx}_0, \ldots, \mathsf{idx}_{n_{\mathsf{Q}}-1})$
for $\beta \in [n_{\mathsf{B}}]$ **do**
  $(\mathsf{idx}^{(\beta)}, \mathsf{ctr}^{(\beta)}) \leftarrow (0,0)$
for $i \in [N]$, $j \in [n_{\mathsf{H}}]$ **do**
  $\beta \leftarrow H_j(i)$
  if $\psi(k) = \beta$ and $\mathsf{idx}_k = i$ for some k **then**
    $\mathsf{idx}^{(\beta)} \leftarrow \mathsf{ctr}^{(\beta)}$
  $\mathsf{ctr}^{(\beta)} \leftarrow \mathsf{ctr}^{(\beta)} + 1$

// Stage 2: Key generation
$s(X) \xleftarrow{\$} \chi_{\mathsf{sk}}$
$\mathsf{gk}_g \leftarrow \mathsf{KSKGen}(\tau_g(s(X)), s(X))$
$\mathsf{ssk} \leftarrow \mathsf{KSKGen}(s(X)^2, s(X))$
for $i \in [\nu_2]$ **do**
  $\mathsf{pjk}_i \leftarrow \mathsf{KSKGen}(\tau_{d/2^i+1}(s(X)), s(X))$

// Stage 3: Encryption
$\mathbf{m} \leftarrow \mathbf{0} \in \mathbb{Z}_q^{n_{\mathsf{C}} \cdot d/2}$
for $\beta \in [n_{\mathsf{B}}]$, $t \in [\nu_0+\nu_1+\nu_2]$ **do**
  $\kappa \leftarrow \beta\ell(\nu_0+\nu_1+\nu_2) + t\ell$
  for $j \in [\ell]$ **do**
    $\mathbf{m}[\kappa + j] \leftarrow (d/2)^{-1} B^j \cdot \mathsf{idx}^{(\beta)}[t]$
for $i \in [n_{\mathsf{C}}]$ **do**
  $m_i(X) \leftarrow \sum_{j=0}^{d/2-1} \mathbf{m}[i \cdot d/2 + j] \cdot X^j$
  $c_i \leftarrow \mathsf{RLWE.Enc}(s(X), m_i(X))$

$\mathsf{qst} \leftarrow (s(X), \psi)$
$\mathsf{qry} \leftarrow ((c_i)_{i \in [n_{\mathsf{C}}]}, \mathsf{gk}_g, \mathsf{ssk}, (\mathsf{pjk}_i)_{i \in [\nu_2]})$
return $(\mathsf{qst}, \mathsf{qry})$.

Respire batches this process over the cuckoo-hashing layer of SealPIR [4], running one retrieval per bucket while turning query expansion into a single batch-wide step and adding a packing step that merges the per-bucket answers into a small number of response ciphertexts.

The principal distinction from prior work is our use of single-key unpacking (§3.2). The other design choices draw selectively on Respire [13] and VIA-B [43], with the details given stage by stage below.

1. *Query expansion.* This stage expands packed RLWE query ciphertexts into the RGSW selectors used during retrieval, one for each per-bucket index bit. Unpack (§3.2), the single-key unpacking we introduce, unpacks the packed query ciphertexts, and SchemeSwitch (§2.5) turns the unpacked RLWE ciphertexts into the RGSW selectors $\mathbf{C}^{(\beta)}[t]$. Whereas OnionPIR and Respire use external-product-based scheme-switching [15], we follow VIA-B and use the key-switching-based scheme-

switching of [27], which is lighter in both key size and server-side computation.

2. *Per-bucket retrieval.* This stage processes each bucket independently, retrieving the entry addressed by that bucket's selector bits. A one-hot vector built from part of the selectors is multiplied by the database matrix, and CMuxTree (§2.6) folds the result with the remaining selectors into one ciphertext $c^{(\beta)}$. Whereas OnionPIR and Respire transmit the one-hot vector as packed RLWE ciphertexts, we follow VIA-B [43] in generating it server-side via DMuxTree (§2.6), reducing query size.

3. *Response packing.* At this point each bucket has produced a single ciphertext $c^{(\beta)}$ encrypting 2^{ν_2} entries, the queried one together with $2^{\nu_2}-1$ others. This stage isolates the queried entry from $c^{(\beta)}$ and packs groups of 2^{ν_2} queried entries into the final response ciphertexts. A *blind rotation* [17], implemented as ν_2 selector-driven CMux steps, multiplies $c^{(\beta)}$ by an appropriate power of X. (Since the entries are encoded in the polynomial coefficients, this multiplication acts as a shift, bringing the queried entry to the target position.) The irrelevant entries are then projected away via the algebraic trace relation, evaluated using the projection keys pjk_i (see Respire [13, §A]). Finally, the per-bucket ciphertexts are shifted into distinct slots, so that each group of 2^{ν_2} can be summed into one final packed response ciphertext. Our response packing largely follows Respire, but omits ring-switching for simplicity (§B). We avoid VIA-B's HERMES-style ring-packing approach [7], which requires substantially larger key material.

Algorithm 6 $\mathsf{Answer}(\mathsf{DB}_{\mathrm{prep}}, \mathsf{qry})$

Input: preprocessed database $\mathsf{DB}_{\mathrm{prep}}$; query qry.
Output: $\mathsf{ans} \in (\mathcal{R}_q^2)^{\lceil n_B/2^{\nu_2}\rceil}$.

$(\mathsf{DB}^{(\beta)}_{\mathrm{prep}})_{\beta\in[n_B]} \leftarrow \mathsf{DB}_{\mathrm{prep}}$
$((c_i)_{i\in[n_C]}, \mathsf{gk}_g, \mathsf{ssk}, (\mathsf{pjk}_i)_{i\in[\nu_2]}) \leftarrow \mathsf{qry}$

// Stage 1: Query expansion
for $i \in [n_C]$ **do**
  $\mathbf{c}[i\cdot d/2 : (i+1)\cdot d/2] \leftarrow \mathsf{Unpack}(\mathsf{gk}_g, c_i)$
for $\beta \in [n_B], t \in [\nu_0+\nu_1+\nu_2]$ **do**
  $\kappa \leftarrow \beta\ell(\nu_0+\nu_1+\nu_2)+t\ell$
  $\mathbf{C}^{(\beta)}[t] \leftarrow \mathsf{SchemeSwitch}(\mathsf{ssk}, \mathbf{c}[\kappa : \kappa+\ell])$

// Stage 2: Per-bucket retrieval
for $\beta \in [n_B]$ **do**
  $\mathbf{c}_{\mathrm{in}} \leftarrow \mathsf{DMuxTree}\big(\mathbf{C}^{(\beta)}[0{:}\nu_0], (0,\Delta)\big)$
  for $j \in [2^{\nu_1}]$ **do**
    $\mathbf{c}_{\mathrm{out}}[j] \leftarrow \sum_{i\in[2^{\nu_0}]} \mathbf{c}_{\mathrm{in}}[i]\cdot \mathsf{DB}^{(\beta)}_{\mathrm{prep}}[i,j]$
  $c^{(\beta)} \leftarrow \mathsf{CMuxTree}\big(\mathbf{C}^{(\beta)}[\nu_0{:}\nu_0+\nu_1], \mathbf{c}_{\mathrm{out}}\big)$

// Stage 3: Response packing
for $\beta \in [n_B]$ **do**
  for $i \in [\nu_2]$ **do**
    $c^{(\beta)} \leftarrow \mathsf{CMux}\big(\mathbf{C}^{(\beta)}[\nu_0+\nu_1+i], c^{(\beta)}, X^{-2^i}c^{(\beta)}\big)$
  $c^{(\beta)} \leftarrow 2^{-\nu_2}\cdot c^{(\beta)}$
  for $i \in [\nu_2]$ **do**
    $c^{(\beta)} \leftarrow c^{(\beta)} + \mathsf{Galois}(\mathsf{pjk}_i, c^{(\beta)})$
for $\gamma \in [\lceil n_B/2^{\nu_2}\rceil]$ **do**
  $\bar{c}^{(\gamma)} \leftarrow \sum_{i\in[2^{\nu_2}]} X^i \cdot c^{(\gamma\cdot 2^{\nu_2}+i)}$

return $(\bar{c}^{(0)}, \ldots, \bar{c}^{(\lceil n_B/2^{\nu_2}\rceil-1)})$.

4.5 Extract

Extract (Alg. 7) decrypts each response ciphertext and decodes the queried entry from its packing slot, recovering $\mathtt{m}_i = \mathsf{DB}_{\mathrm{raw}}[\mathsf{idx}_i]$ for every placed query $i \in [n_Q]$ unless a decryption failure occurs (§A).

4.6 Security Analysis

Privacy. Under the per-batch secret $s(X) \xleftarrow{\$} \chi_{\mathsf{sk}}$, every component of qry is, by inspection, either an RLWE ciphertext (c_i) or an RLWE key-switching key (gk_g, ssk, pjk_i) with parameters $(d, q, \chi_{\mathsf{sk}}, \chi_{\mathsf{err}})$. Hence, qry is pseudorandom under the $(d, q, \chi_{\mathsf{sk}}, \chi_{\mathsf{err}})$-RLWE assumption and hides the queried indices. Note that each key-switching key encrypts a function of the secret under the secret itself. Thus, we additionally require a circular-security assumption.

Cuckoo-hashing failure. When Cuckoo (§2.3) cannot place all n_Q queries, some query j is left unplaced, with $\psi(j) = \perp$. Unlike a decryption failure, this failure is not *silent*: the client observes it while constructing the query, before anything is sent. It nevertheless has privacy implications, since whether a batch fails depends on the queried indices. We must therefore choose the cuckoo-hashing parameters so that placement failures are sufficiently rare. The appropriate failure probability is application-dependent. If the leakage must be ruled out, the parameters should make failure cryptographically negligible, so that the server effectively never observes it. If the application can tolerate the residual leakage, or can recover from $\perp$ obliviously, it may use cheaper parameters and degrade gracefully, for example by issuing the placed queries and deferring the rest. We estimate the failure probability by simulating Cuckoo; see Remark 3.

5 Optimizations

In this section, we describe the optimizations applied to the base construction of §4 to arrive at our final scheme. We begin by tuning the gadget and key-switching machinery (§5.1), which is central to the overall efficiency. We then group the remaining optimizations by the primary cost they reduce:

Algorithm 7 $\mathsf{Extract}(\mathsf{qst}, \mathsf{ans})$

Input: query state qst; answer ans.
Output: entries $(\mathtt{m}_0, \ldots, \mathtt{m}_{n_\mathsf{Q}-1}) \in (\{0,1\}^w)^{n_\mathsf{Q}}$.

$(s(X), \psi) \leftarrow \mathsf{qst}$
$(\mathsf{ans}^{(\gamma)})_{\gamma \in [\lceil n_\mathsf{B}/2^{\nu_2} \rceil]} \leftarrow \mathsf{ans}$
for $\gamma \in [\lceil n_\mathsf{B}/2^{\nu_2} \rceil]$ **do**
  $\mu_\gamma(X) \leftarrow \mathsf{RLWE.Dec}(s(X), \mathsf{ans}^{(\gamma)})$
  $m_\gamma(X) \leftarrow \lfloor \mu_\gamma(X)/\Delta \rceil \bmod p$
for $i \in [n_\mathsf{Q}]$ **do**
  $\beta \leftarrow \psi(i)$
  $\gamma \leftarrow \lfloor \beta / 2^{\nu_2} \rfloor$
  $\mathtt{m}_i \leftarrow \mathsf{Dcd}\big(m_\gamma(X), \beta \bmod 2^{\nu_2}\big)$
return $(\mathtt{m}_0, \ldots, \mathtt{m}_{n_\mathsf{Q}-1})$.

communication (§5.2) or server computation (§5.3). We also discuss the optimizations we chose not to adopt in §B.

5.1 Gadget and Key-Switching Tuning

Decoupling gadgets. The construction uses gadgets in six places: the unpacking key gk_g, the scheme-switching key ssk, the three selector families ($\mathbf{C}_0$ for DMux, $\mathbf{C}_1$ for CMux, and $\mathbf{C}_2$ for response packing), and the projection keys pjk_i. We decouple these six by assigning each its own base and length, rather than a single shared gadget. This exposes a larger design space: we use longer gadgets where noise growth is the bottleneck and shorter gadgets where there is slack, reducing communication and computation.

Approximate gadget decomposition. The exact decomposition of §2.4 uses $\ell = \lceil \log_B q \rceil$ digits for every coefficient. In contrast, the approximate decomposition [17] keeps only the $\ell' \le \ell$ most significant digits. The truncated gadget vector is

$$\mathbf{g} = (B^{\ell-\ell'}, B^{\ell-\ell'+1}, \ldots, B^{\ell-1}),$$

and $\mathbf{g}^{-1}(\cdot)$ first rounds each coefficient to the nearest multiple of $B^{\ell-\ell'}$ before extracting its digits. Consequently, the identity of §2.4 holds only up to a rounding residual:

$$\mathbf{g} \cdot \mathbf{g}^{-1}(a(X)) = a(X) + \varepsilon, \qquad \|\varepsilon\|_\infty \le B^{\ell-\ell'}/2.$$

The exact gadget is the special case $\ell' = \ell$.

The size of a key or selector, as well as the cost of applying it in a key-switching or external product, scales with ℓ'. Thus, reducing ℓ' makes the operation more efficient, but at the cost of increased noise. This added noise matters only once it becomes comparable to what the input ciphertext to the key-switching or external product already carries, so cutting ℓ' to that level is essentially free.

Hybrid key-switching. The noise analysis in §A shows that the key-switching error is a major bottleneck, especially for gk_g, whose per-switch error accumulates during unpacking. The natural lever is the gadget length ℓ, but larger ℓ eventually yields diminishing returns while increasing the key size. A more balanced approach is to increase the gadget length only to a moderate level and seek the remaining headroom from a larger modulus q. In gadget-based key-switching (§2.5), however, the key and ciphertexts share the same modulus q, so enlarging q inflates both.

Hybrid key-switching [32, 39] avoids this by placing an *auxiliary modulus* P on the key side: the switch is performed at the momentarily enlarged modulus $P \cdot q$ and then rescaled back to q, which reduces the key-switching noise by a factor of P. Thus, only the key material grows from modulus q to $P \cdot q$, while ciphertexts remain at modulus q. A caveat is that security then relies on RLWE over this larger modulus $P \cdot q$, rather than over q alone.

5.2 Reducing Communication

Randomness compression. The $a(X)$ component of a fresh RLWE ciphertext is uniformly random and unstructured. Hence, the client can send only a short seed, allowing each party to locally derive all $a(X)$ components. Applied to both the query ciphertexts and the key material, this roughly halves the upload. This technique is standard in lattice-based cryptography [3, 10].

Response truncation. The response RLWE ciphertexts can be compressed before transmission by simply truncating the least-significant bits of their coefficients, provided that the decrypted message remains decodable. Note that the $b(X)$ component can be truncated more aggressively than the $a(X)$ component: during decryption, a truncation error of $a(X)$ is amplified by the secret $s(X)$, whereas one of $b(X)$ is not. This technique is standard in lattice-based cryptography [10].

Partial projection. Each plaintext packs 2^{ν_2} entries, and the response-packing stage of Answer (Alg. 6) projects away all but the queried entry, thereby resolving all ν_2 bits that index an entry within a plaintext. Instead, the server can resolve only $\nu_2' \le \nu_2$ of these bits, leaving $2^{\nu_2 - \nu_2'}$ candidate entries for the client to disambiguate after decryption. This yields a trade-off between key size and response size: only ν_2' projection keys are required, but the response expands to $\lceil n_\mathsf{B}/2^{\nu_2'} \rceil$ ciphertexts. We choose ν_2' to minimize total communication; this optimization is most beneficial when ν_2 is large, i.e., when the entry size w is small.

Collision-free candidate buckets. For simplicity, we have so far assumed that each item's n_H candidate buckets are sampled using n_H independent hashes (§2.3), so collisions may occur. Sampling a distinct n_H-subset of $[n_\mathsf{B}]$ instead (rejecting repeats) prevents an item from collapsing onto fewer than n_H

buckets. For a fixed target failure probability, this allows a smaller n_{B}, especially when n_{Q} is small, reducing both query and response size, since the numbers of packed query ciphertexts and response ciphertexts both grow with n_{B}.

5.3 Reducing Server Computation

Precomputing the database NTT. Multiplication in $\mathcal{R}_q$ is performed using the *number-theoretic transform* (NTT). By mapping each operand to its evaluation representation, polynomial multiplication reduces to coefficient-wise multiplication, lowering the cost from $O(d^2)$ to $O(d \log d)$. Since multiplication by the database matrix dominates the server's work, transforming the database into the NTT domain on the fly would itself become a computational bottleneck. We therefore have Preprocess store the database directly in NTT form, so that these transforms are paid once during preprocessing rather than once per query. This is a standard optimization in similar RLWE-based PIR schemes [13, 16, 43, 48]. The tradeoff is increased memory usage: each database coefficient is stored as a $\mathbb{Z}_q$ element rather than a $\mathbb{Z}_p$ element, inflating the database size by roughly a factor of $\log q / \log p$.

Modulus-switching before DB multiplication. *Modulus-switching* [12] rescales a ciphertext to a smaller modulus q' while leaving the noise-to-modulus ratio nearly unchanged. We apply one immediately before the dominant database multiplication, so that the database is stored and multiplied at q', speeding up all subsequent operations and shrinking the memory blowup. The RGSW selectors $(\mathbf{C}_1, \mathbf{C}_2)$ and projection keys pjk_i used afterward must then be built at q'; this also shrinks the projection keys, reducing the key upload.

Skipping padded entries. Buckets are padded to the nearest power-of-two size (§4.2). Since padding entries are zero, any all-padding plaintext is the zero plaintext in $\mathcal{R}_p$ and can be skipped, along with the parts of the surrounding DMuxTree and CMuxTree that operate only on it.

6 Evaluation

We implement our scheme in C++, using HEXL [9] for AVX-512 acceleration of the NTT and modular arithmetic, xxHash [19] for the non-cryptographic hash functions in cuckoo-hashing, and OpenSSL for seed expansion and randomness, with no other dependencies. All benchmarks run single-threaded on an Intel Ice Lake Xeon at 2.60 GHz with 512 GiB of memory, running Ubuntu 26.04 LTS. §6.1 reports the parameters used throughout the evaluation and §6.2 compares the scheme against two baselines, InsPIRe [47] and Respire-B [13]. §C then provides microbenchmarks. For runtimes, we report the median of 5 trials, except InsPIRe's Preprocess, which we measure once.

6.1 Parameters

Throughout all configurations, we use ring degree $d = 4096$, with χ_{sk} uniform over the polynomials whose coefficients lie in $\{-1, 0, 1\}$ and χ_{err} a discrete Gaussian of standard deviation $\sigma_{\mathsf{err}} = 3.19$. The ciphertext modulus is $q = 2^{62} - 2^{16} + 1$, the switched modulus (§5.3) is $q' = 2^{32} - 2^{17} - 2^{13} + 1$, and the plaintext modulus is $p = 16$. The remaining parameters vary across configurations, and we report them in Table 5; all are chosen for 128-bit security under the lattice estimator [1] (commit `53da598`) and a per-batch decryption failure probability below 2^{-40} (§A).

Remark 3 (Cuckoo-hashing failure)**.** *Respire [13] sets $n_{\mathsf{H}} = 3$ and $n_{\mathsf{B}} \approx 1.5 n_{\mathsf{Q}}$, following the rule of thumb of SealPIR [4] for a 2^{-40} failure probability, and we adopt the same parameters for a fair comparison against it (§6.2). However, this rule does not hold at small n_{Q}: our simulation estimates the failure probability at $2^{-15.9}$ for $n_{\mathsf{Q}} = 32$ with $(n_{\mathsf{B}}, n_{\mathsf{H}}) = (48, 3)$. For the other parameters we use, the simulation bounds the failure probability by 2^{-30}; for the comparison against InsPIRe [47], for instance, we use $(n_{\mathsf{B}}, n_{\mathsf{H}}) = (48, 4)$ at $n_{\mathsf{Q}} = 32$.*

6.2 Comparison

We benchmark against the two natural baselines for doubly-stateless batch PIR. The first simply repeats a doubly-stateless single-query PIR scheme, InsPIRe [47], once per index (Table 3). The second takes a server-stateful batch PIR scheme, Respire-B [13], and makes it doubly-stateless by re-shipping its evaluation keys with each batch (Table 4).

Versus InsPIRe [47]. We report amortized per-query cost, which isolates the benefit of batching: repeating single-query InsPIRe n_{Q} times leaves its per-query cost unchanged, whereas our scheme amortizes its per-batch cost over the n_{Q} queries. At a fixed database, InsPIRe can trade the server's online runtime (Answer) against communication; for completeness we include both extremes reported in its paper, labeled *min-comm* and *min-time* in Table 3. We benchmark at the 1 GB and 8 GB databases with 64 B entries reported in the InsPIRe paper, reusing its parameter sets, while our scheme runs at $n_{\mathsf{Q}} \in \{32, 256\}$. InsPIRe uses no cuckoo-hashing, so only decryption failure applies, and its paper fixes that probability at 2^{-40} per query; our scheme holds the same 2^{-40} bound on the probability that any query in the batch fails to decrypt, and additionally bounds the cuckoo-hashing failure probability by 2^{-30}, set experimentally. At the 1 GB database, our scheme reduces total communication by $20\times$ for $n_{\mathsf{Q}} = 32$ and $68\times$ for $n_{\mathsf{Q}} = 256$, even relative to InsPIRe's min-comm configuration. At this database, our server's online runtime is below InsPIRe's min-comm configuration at both batch sizes, and below its min-time configuration at $n_{\mathsf{Q}} = 256$; only at $n_{\mathsf{Q}} = 32$ against min-time do we spend more, by $3\times$. Communication is in any case the more precious resource in practice, since

Table 3: Comparison with InsPIRe.

	InsPIRe		Ours (amortized*)	
	min-comm	min-time	$n_Q = 32$	$n_Q = 256$
$1\,\text{GB} = 2^{24} \times 64\,\text{B}$				
Upload (KB)	201	1090	9.5	2.3
Download (KB)	25	12	1.9	1.0
Total (KB)	225	1102	11.4	3.3
Preprocess (s)	1193	100	41	33
Query (ms)	42.45	62.28	12.84	1.27
Answer (ms)	1267	281	843	225
Extract (ms)	0.82	0.38	0.03	0.01
$8\,\text{GB} = 2^{27} \times 64\,\text{B}$				
Upload (KB)	229	3842	9.7	2.5
Download (KB)	98	12	1.9	1.1
Total (KB)	328	3854	11.6	3.6
Preprocess (s)	4916	673	323	263
Query (ms)	42.67	118.23	93.26	7.96
Answer (ms)	5098	1110	5182	619
Extract (ms)	2.98	0.56	0.02	0.01

* Except Preprocess, which is per database, not per index.

Table 4: Comparison with Respire-B.

	$n_Q = 32$		$n_Q = 256$	
	Respire-B	Ours	Respire-B	Ours
$256\,\text{MB} = 2^{20} \times 256\,\text{B}$				
Upload (MB)	4.86	0.28	5.13	0.47
Download (MB)	0.03	0.07	0.24	0.52
Total (MB)	4.89	0.35	5.37	0.98
Preprocess (s)	56.9	7.2	55.1	7.4
Query (ms)	111	56	251	78
Answer (s)	15.4	8.3	62.6	30.0
Extract (ms)	1.3	0.8	8.0	6.9
$1\,\text{GB} = 2^{22} \times 256\,\text{B}$				
Upload (MB)	4.91	0.30	5.32	0.53
Download (MB)	0.03	0.06	0.24	0.49
Total (MB)	4.94	0.37	5.55	1.03
Preprocess (s)	229.7	29.1	225.3	29.5
Query (ms)	116	109	254	130
Answer (s)	27.5	23.0	90.2	52.8
Extract (ms)	1.3	0.8	8.1	6.7

server runtime is the easier of the two to buy back with more cores. We also note that InsPIRe's Preprocess cost exceeds ours.

Versus Respire-B [13]. We report per-batch totals, not amortized per-query costs, so that the evaluation keys re-shipped by the doubly-stateless port remain visible as an upload the client pays for every batch. We benchmark at the 256 MB and 1 GB databases with 256 B entries and batch sizes $n_Q \in \{32, 256\}$, reusing the parameter sets reported for Respire-B. At the 1 GB database, our scheme reduces total communication by $13\times$ for $n_Q = 32$ and $5\times$ for $n_Q = 256$, with the server's online runtime (Answer) $1.2\times$ and $1.7\times$ faster respectively. Seen another way, Respire-B's key material is large enough that even registering it once, rather than resending it per batch, leaves our total communication the lower one for a client that does not query too often. At the 1 GB database with $n_Q = 32$, the crossover is 22 batches.

Remark 4 (Cost of bucket assignment)**.** *The* Query *comparison in Table 4 favors Respire-B, which precomputes the bucket assignment (about* 51 *MB at the* 256 *MB database and* 205 *MB at* 1 *GB), whereas our* Query *(§4.3) computes it online. Even so, computing it online remains practical: although linear in the database size N, it takes only* 0.02 *seconds at the* 256 *MB database and* 0.07 *seconds at* 1 *GB.*

Remark 5 (Comparison with VIA-B)**.** *VIA-B [43], a recent communication-efficient batch PIR, incurs an even larger doubly-stateless penalty than Respire-B: its public parameters are* 14.8 *MB, against Respire-B's* 4.8 *MB, so re-shipping them with each batch dominates the upload. We do not give it a separate table, since VIA-B has no public implementation, leaving communication as the only comparable axis.*

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015. doi:10.1515/jmc-2015-0016.

[2] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 1811–1828. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/ali.

[3] Erdem Alkim, Léo Ducas, Thomas Pöppelmann, and Peter Schwabe. Post-quantum key exchange - A new hope. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 327–343. USENIX Association, August 2016. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/alkim.

[4] 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. IEEE Computer Society Press, May 2018. doi:10.1109/SP.2018.00062.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[6] Pranav Shriram Arunachalaramanan and Ling Ren. Single-server stateful PIR with verifiability and balanced efficiency. In *2026 IEEE Symposium on Security and Privacy*, pages 1385–1402. IEEE Computer Society Press, May 2026. doi:10.1109/SP63933.2026.00019.

[7] Youngjin Bae, Jung Hee Cheon, Jaehyung Kim, Jai Hyun Park, and Damien Stehlé. HERMES: Efficient ring packing using MLWE ciphertexts and application to transciphering. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 37–69. Springer, Cham, August 2023. doi:10.1007/978-3-031-38551-3_2.

[8] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Batch PIR and labeled PSI with oblivious ciphertext compression. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/bienstock.

[9] 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 Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC)*, pages 57–62, 2021. doi:10.1145/3474366.3486926.

[10] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy*, pages 353–367. IEEE Computer Society Press, April 2018. doi:10.1109/EuroSP.2018.00032.

[11] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886. Springer, Berlin, Heidelberg, August 2012. doi:10.1007/978-3-642-32009-5_50.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325. ACM, January 2012. doi:10.1145/2090236.2090262.

[13] Alexander Burton, Samir Jordan Menon, and David J. Wu. Respire: High-rate PIR for databases with small records. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 1463–1477. ACM Press, October 2024. doi:10.1145/3658644.3690328.

[14] Quang Cao, Son Hoang Dau, Rinaldo Gagiano, Duy Huynh, Xun Yi, Phuc Lu Le, Quang-Hung Luu, Emanuele Viterbo, Yu-Chih Huang, Jingge Zhu, Mohammad M. Jalalzai, and Chen Feng. TreePIR: Efficient private retrieval of merkle proofs via tree colorings with fast indexing and zero storage overhead. In Marina Blanton, William Enck, and Cristina Nita-Rotaru, editors, *2025 IEEE Symposium on Security and Privacy*, pages 4476–4494. IEEE Computer Society Press, May 2025. doi:10.1109/SP61157.2025.00032.

[15] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, November 2019. doi:10.1145/3319535.3354226.

[16] Yue Chen and Ling Ren. OnionPIRv2: Efficient single-server PIR. Cryptology ePrint Archive, Report 2025/1142, 2025. URL: https://eprint.iacr.org/2025/1142.

[17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, January 2020. doi:10.1007/s00145-019-09319-x.

[18] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995. doi:10.1109/SFCS.1995.492461.

[19] Yann Collet. xxHash: Extremely fast non-cryptographic hash algorithm. https://github.com/Cyan4973/xxHash, 2012. Version 0.8.3.

[20] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3835–3851. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/colombo.

[21] Kelong Cong, Debajyoti Das, Georgio Nicolas, and Jeongeun Park. Panacea: Non-interactive and stateless oblivious RAM. In *2024 IEEE European Symposium on Security and Privacy*, pages 790–809. IEEE Computer Society Press, July 2024. doi:10.1109/EuroSP60621.2024.00049.

[22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 3–33. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07085-3_1.

[23] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Cham, May 2020. doi:10.1007/978-3-030-45721-1_3.

[24] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *PoPETs*, 2023(1):365–383, January 2023. doi:10.56553/popets-2023-0022.

[25] Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/de-castro.

[26] Leo de Castro, Kevin Lewi, and Edward Suh. WhisPIR: Stateless private information retrieval with low communication. Cryptology ePrint Archive, Report 2024/266, 2024. URL: https://eprint.iacr.org/2024/266.

[27] Gabrielle De Micheli, Duhyeong Kim, Daniele Micciancio, and Adam Suhl. Faster amortized FHEW bootstrapping using ring automorphisms. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part II*, volume 14604 of *LNCS*, pages 322–353. Springer, Cham, April 2024. doi:10.1007/978-3-031-57728-4_11.

[28] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. URL: https://eprint.iacr.org/2012/144.

[29] Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. ThorPIR: Single server PIR via homomorphic thorp shuffles. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 1448–1462. ACM Press, October 2024. doi:10.1145/3658644.3690326.

[30] Ben Fisch, Zeyu Liu, Eran Tromer, and Yunhao Wang. UnifOMR: Oblivious message retrieval with near-optimal concrete efficiency. Cryptology ePrint Archive, Report 2026/910, 2026. URL: https://eprint.iacr.org/2026/910.

[31] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in BGV-style homomorphic encryption. In Ivan Visconti and Roberto De Prisco, editors, *SCN 12*, volume 7485 of *LNCS*, pages 19–37. Springer, Berlin, Heidelberg, September 2012. doi:10.1007/978-3-642-32928-9_2.

[32] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 850–867. Springer, Berlin, Heidelberg, August 2012. doi:10.1007/978-3-642-32009-5_49.

[33] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Berlin, Heidelberg, August 2013. doi:10.1007/978-3-642-40041-4_5.

[34] 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 (SOSP)*, 2023. doi:10.1145/3600006.3613134.

[35] 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 Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 3889–3905. USENIX Association, August 2023. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger.

[36] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: Single-server PIR with efficient updates via invertible PRFs. In Serge Fehr and Pierre-Alain Fouque, editors, *EUROCRYPT 2025, Part VI*, volume 15606 of *LNCS*, pages 3–33. Springer, Cham, May 2025. doi:10.1007/978-3-031-91095-1_1.

[37] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory*, volume 84 of *Graduate Texts in Mathematics*. Springer, 2 edition, 1990.

[38] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004. doi:10.1145/1007352.1007396.

[39] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Mehdi Tibouchi and Huaxiong Wang, editors, *ASIACRYPT 2021, Part III*, volume 13092 of *LNCS*, pages 608–639. Springer, Cham, December 2021. doi:10.1007/978-3-030-92078-4_21.

[40] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997. doi:10.1109/SFCS.1997.646125.

[41] Keewoo Lee and Yongdong Yeo. SophOMR: Improved oblivious message retrieval from SIMD-aware homomorphic compression. Cryptology ePrint Archive, Report 2024/1814, 2024. URL: https://eprint.iacr.org/2024/1814.

[42] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz. Hintless single-server private information retrieval. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part IX*, volume 14928 of *LNCS*, pages 183–217. Springer, Cham, August 2024. doi:10.1007/978-3-031-68400-5_6.

[43] Chenyang Liu, Xukun Wang, and Zhifang Zhang. VIA: Communication-efficient single-server private information retrieval. In *2026 IEEE Symposium on Security and Privacy*, pages 1503–1521. IEEE Computer Society Press, May 2026. doi:10.1109/SP63933.2026.00086.

[44] Zeyu Liu and Eran Tromer. Oblivious message retrieval. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part I*, volume 13507 of *LNCS*, pages 753–783. Springer, Cham, August 2022. doi:10.1007/978-3-031-15802-5_26.

[45] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: Oblivious message retrieval with reduced communication and computation. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/liu-zeyu.

[46] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Berlin, Heidelberg, May / June 2010. doi:10.1007/978-3-642-13190-5_1.

[47] 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*, pages 1463–1482. IEEE Computer Society Press, May 2026. doi:10.1109/SP63933.2026.00061.

[48] 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*, pages 930–947. IEEE Computer Society Press, May 2022. doi:10.1109/SP46214.2022.9833700.

[49] Samir Jordan Menon and David J. Wu. YPIR: High-throughput single-server PIR with silent preprocessing. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024. URL: https://www.usenix.org/conference/usenixsecurity24/presentation/menon.

[50] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 700–718. Springer, Berlin, Heidelberg, April 2012. doi:10.1007/978-3-642-29011-4_41.

[51] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2292–2306. ACM Press, November 2021. doi:10.1145/3460120.3485381.

[52] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *2023 IEEE Symposium on Security and Privacy*, pages 437–452. IEEE Computer Society Press, May 2023. doi:10.1109/SP46215.2023.10179329.

[53] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms – ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[54] Mayank Rathee, Keewoo Lee, and Raluca Ada Popa. Verifiable PIR with small client storage. In *2026 IEEE Symposium on Security and Privacy*, pages 1423–1442. IEEE Computer Society Press, May 2026. doi:10.1109/SP63933.2026.00024.

[55] Ling Ren, Muhammad Haris Mughees, and I Sun. Simple and practical amortized sublinear private information retrieval using dummy subsets. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 1420–1433. ACM Press, October 2024. doi:10.1145/3658644.3690266.

[56] Karl Wüst, Sinisa Matetic, Moritz Schneider, Ian Miers, Kari Kostiainen, and Srdjan Capkun. ZLiTE: Lightweight clients for shielded zcash transactions using trusted execution. In Ian Goldberg and Tyler Moore, editors, *FC 2019*, volume 11598 of *LNCS*, pages 179–198. Springer, Cham, February 2019. doi:10.1007/978-3-030-32101-7_12.

[57] Zhou Zhang, Song Bian, Zhenyu Guan, and Giuseppe Persiano. RotPIR: Sublinear single-server PIR with efficient pre-processing via oblivious homomorphic rotation. Cryptology ePrint Archive, Report 2026/1336, 2026. URL: https://eprint.iacr.org/2026/1336.

[58] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: Extremely simple, single-server PIR with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy*, pages 4296–4314. IEEE Computer Society Press, May 2024. doi:10.1109/SP54263.2024.00055.

A Noise Analysis

We analyze the noise of the base construction of §4 and bound the resulting decryption failure probability.

Failure probability. Extract (§4.5) first recovers the noisy encoding $\Delta m(X)+e(X)$ with RLWE.Dec, then extracts $m(X)$ from it by coefficient-wise rounding (§2.4). A message coefficient is successfully recovered if and only if the corresponding coefficient of $e(X)$ has magnitude less than $\Delta/2$. With each coefficient of $e(X)$ a zero-mean Gaussian of variance σ^2, it fails with probability at most $2\exp(-(\Delta/2)^2/(2\sigma^2))$. The response is $\lceil n_{\mathsf{B}}/2^{\nu_2}\rceil$ ciphertexts of d coefficients each. Assuming all of them have the same noise variance σ^2_{ans}, a union bound over all the coefficients bounds the per-batch failure probability

$$P_{\mathsf{fail}} \le 2\cdot\lceil n_{\mathsf{B}}/2^{\nu_2}\rceil\cdot d\cdot\exp\left(-\frac{(\Delta/2)^2}{2\sigma^2_{\mathsf{ans}}}\right).$$

Per-stage noise analysis. We now estimate the response noise variance σ^2_{ans}. Write σ^2_{err} for the variance of χ_{err}. Because every key-switching key shares the gadget parameters (B,ℓ) at the modulus q, one key-switching adds the same variance $\sigma^2_{\mathsf{KS}} = B^2\ell d\,\sigma^2_{\mathsf{err}}/4$ throughout (Lemma 2). The stages of Answer (Alg. 6) then contribute as follows.

1. *Query expansion.* Unpacking gives each output coefficient variance $\sigma^2_{\mathsf{unp}} \approx (d^2/8)\cdot\sigma^2_{\mathsf{KS}}$ (Lemma 9). The scheme-switching then assembles the RGSW selectors, whose right half carries per-coefficient noise variance σ^2_{unp} and whose left half carries it multiplied by $\|s(X)\|_2^2$ (Lemma 3). One external product against such a selector composes these two variances (Lemma 4). Neglecting a smaller key-switching term that the scheme-switching adds to the left half,

$$\sigma^2_{\boxdot} \approx \frac{B^2\ell d}{4}\cdot\left(1+\|s(X)\|_2^2\right)\cdot\sigma^2_{\mathsf{unp}}.$$

2. *Per-bucket retrieval.* The ν_0-level DMuxTree leaves every input to the matrix product at variance at most $\nu_0\cdot\sigma^2_{\boxdot}$ (Lemma 5), each encrypting a Δ-scaled bit. The matrix multiplication sums 2^{ν_0} products of these with database entries in $\mathcal{R}_p$, each scaling the input variance by at most $d\cdot p^2/4$ (Lemma 1):

$$\sigma^2_{\mathsf{mat}} := 2^{\nu_0}\cdot\frac{d\cdot p^2}{4}\cdot\nu_0\cdot\sigma^2_{\boxdot}.$$

   The CMuxTree then adds one $\sigma^2_{\boxdot}$ for each of its ν_1 levels (Lemma 6), leaving each bucket's retrieved ciphertext at

$$\sigma^2_{\mathsf{bkt}} := \sigma^2_{\mathsf{mat}} + \nu_1\cdot\sigma^2_{\boxdot}.$$

3. *Response packing.* The blind rotation likewise adds one $\sigma^2_{\boxdot}$ for each of its ν_2 levels. The projection's ν_2 doublings each scale the ciphertext by two, hence the variance by four. The $2^{-\nu_2}$ normalization cancels them, so the projection leaves its input unamplified. Each of the ν_2 key-switchings it injects is amplified only by the doublings that follow it: the one with k doublings remaining contributes $4^k\cdot\sigma^2_{\mathsf{KS}}$, summing to $\sum_{k=0}^{\nu_2-1}4^k\cdot\sigma^2_{\mathsf{KS}} = \frac{1}{3}(4^{\nu_2}-1)\cdot\sigma^2_{\mathsf{KS}}$. The packing sum of 2^{ν_2} independent

per-bucket ciphertexts then multiplies the variance by 2^{ν_2}, so the response carries noise variance of at most

$$\sigma_{\mathsf{ans}}^2 := 2^{\nu_2}\left(\sigma_{\mathsf{bkt}}^2 + \nu_2 \cdot \sigma_{\boxdot}^2 + \tfrac{1}{3}\left(4^{\nu_2}-1\right)\cdot \sigma_{\mathsf{KS}}^2\right),$$

which collapses to $\sigma_{\mathsf{ans}}^2 \approx 2^{\nu_2}\cdot\sigma_{\mathsf{bkt}}^2 \approx 2^{\nu_2}\cdot\sigma_{\mathsf{mat}}^2$ once the terms subdominant under typical settings are dropped.

Expanding σ_{ans}^2 down to σ_{err}^2 gives the closed-form approximation

$$\sigma_{\mathsf{ans}}^2 \approx \frac{1}{512}\,2^{\nu_0+\nu_2}\,\nu_0\left(1+\|s(X)\|_2^2\right)B^4\ell^2 d^5 p^2\,\sigma_{\mathsf{err}}^2.$$

B Optimizations Not Adopted

We collect here the optimizations we considered for the base construction of §4 but did not adopt (§5).

Ring-switching. *Ring-switching* [31] maps a ciphertext to a smaller ring dimension $d' < d$, at the cost of one key-switching and the corresponding key material. Respire [13] applies ring-switching as a final step, shrinking a response ciphertext whose payload occupies fewer than d coefficients. In our batch setting, this yields little benefit, since most response ciphertexts are fully packed with 2^{ν_2} entries (§4.4). Applied instead before the projection step, ring-switching would allow the projection keys pjk_i to live in the smaller ring and shrink accordingly. We do not adopt this optimization, for simplicity.

Preprocessing Unpack in the CRS model. Each query in our scheme carries a fresh seed for the $a(X)$ components. In the common reference string (CRS) model, where a single seed can instead be fixed and reused across queries and clients, the entire $a(X)$ side of Unpack becomes query-independent. The server can therefore precompute it once for all clients, leaving only the $b(X)$ side to be computed per query, similar to the optimization used by InsPIRe [47] for ring-packing. Since the bottleneck lies in the database multiplication, however, this optimization yields only a limited speedup, and we omit it for simplicity.

A finer modulus schedule. Our single modulus-switching to q' could be generalized to a finer-grained schedule that passes through several intermediate moduli, progressively reducing the modulus as the computation proceeds and thereby lowering server computation and communication costs. We forgo it to keep the implementation simple.

C Microbenchmarks

We microbenchmark our scheme to see where its server runtime and communication go. We report at a representative operating point: 64-byte entries, databases of 1 and 8 GB, and batch sizes $n_{\mathsf{Q}} \in \{32, 256\}$.

Table 6 records how the server's online runtime divides among the operations of Answer. We can observe that Unpack is not the bottleneck: its cost is set by the number of packed query ciphertexts, not by the database size, so its share falls as the database grows. The dominant operation, the matrix multiplication, does not grow with the batch size, so a larger batch improves the amortized cost.

Table 7 likewise divides the communication into its components, all of them under the randomness-compression optimization of §5.2: the a-component of each ciphertext is expanded from a shared seed, so only the b-component is transmitted and measured. The key material is what the doubly-stateless setting charges per batch, and it barely moves across the four configurations. A larger batch therefore shifts the upload from the keys to the query ciphertexts, and lowers the amortized cost.

Table 5: Per-configuration parameters.

		$2^{24}\times 64$ B = 1 GB		$2^{27}\times 64$ B = 8 GB		$2^{20}\times 256$ B = 256 MB		$2^{22}\times 256$ B = 1 GB	
		n_{Q}=32	n_{Q}=256	n_{Q}=32	n_{Q}=256	n_{Q}=32	n_{Q}=256	n_{Q}=32	n_{Q}=256
See Table 2.	$(n_{\mathsf{B}}, n_{\mathsf{H}})$	(48,4)	(384,3)	(48,4)	(384,3)	(48,3)	(384,3)	(48,3)	(384,3)
	(ν_0,ν_1,ν_2)	(7,9,5)	(6,7,5)	(9,10,5)	(7,9,5)	(6,8,3)	(5,6,3)	(7,9,3)	(6,7,3)
Number of resolved intra bits (§5.2)	ν_2'	3	4	3	4	3	3	3	3
Auxiliary modulus (§5.1)	$\lceil \log P_{\mathsf{gk}} \rceil$	26	29	29	30	30	29	26	31
	$\lceil \log P_{\mathsf{ssk}} \rceil$	40	43	43	44	44	43	40	44
Approximate gadget parameters (§5.1)	$(B_{\mathsf{gk}}, \ell_{\mathsf{gk}}, \ell'_{\mathsf{gk}})$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$	$(2^{31},2,2)$
	$(B_{\mathsf{ssk}}, \ell_{\mathsf{ssk}}, \ell'_{\mathsf{ssk}})$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$	$(2^{62},1,1)$
	(B_0, ℓ_0, ℓ'_0)	$(2^5,13,7)$	$(2^8,8,4)$	$(2^7,9,5)$	$(2^8,8,4)$	$(2^9,7,3)$	$(2^9,7,3)$	$(2^5,13,7)$	$(2^8,8,4)$
	(B_1, ℓ_1, ℓ'_1)	$(2^8,4,3)$	$(2^8,4,3)$	$(2^8,4,3)$	$(2^8,4,2)$	$(2^8,4,2)$	$(2^8,4,2)$	$(2^8,4,3)$	$(2^8,4,2)$
	(B_2, ℓ_2, ℓ'_2)	$(2^8,4,3)$	$(2^8,4,2)$	$(2^8,4,2)$	$(2^8,4,3)$	$(2^8,4,2)$	$(2^8,4,3)$	$(2^8,4,3)$	$(2^8,4,3)$
	$(B_{\mathsf{pjk}}, \ell_{\mathsf{pjk}}, \ell'_{\mathsf{pjk}})$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$	$(2^{11},3,2)$
Number of truncated bits (§5.2)	(ρ_a, ρ_b)	(19,25)	(19,24)	(19,25)	(18,23)	(18,24)	(19,24)	(19,25)	(19,25)

Table 6: Server's Online Runtime Breakdown (s).

	1 GB		8 GB	
	n_{Q}=32	n_{Q}=256	n_{Q}=32	n_{Q}=256
Unpack	2.7	13.4	2.7	14.9
SchemeSwitch	2.1	11.0	2.0	11.4
DMuxTree	3.6	9.2	10.9	18.5
Modulus switching	0.9	3.6	3.5	7.1
Matrix multiplication	12.7	11.3	136.8	79.5
CMuxTree	5.0	8.3	9.8	26.2
Response packing	0.1	0.7	0.1	0.8
Total	27.0	57.6	165.8	158.4
Amortized	0.84	0.22	5.18	0.62

Table 7: Communication Breakdown (KB).

	1 GB		8 GB	
	n_{Q}=32	n_{Q}=256	n_{Q}=32	n_{Q}=256
$(c_i)_{i\in[n_{\mathsf{C}}]}$	63.5	317.4	63.5	349.2
gk_g	90.1	93.2	93.2	94.2
ssk	52.2	53.8	53.8	54.3
$(\mathsf{pjk}_i)_{i\in[\nu_2']}$	98.3	131.1	98.3	131.1
Upload	304.1	595.5	308.8	628.8
Download	61.4	258.0	61.4	282.6
Total	365.6	853.5	370.2	911.4
Amortized	11.42	3.33	11.57	3.56