Respire: High-Rate PIR for Databases with Small Records

Alexander Burton
UT Austin
amacburton@gmail.com

Samir Jordan Menon
Blyss
samir@blyss.dev

David J. Wu
UT Austin
dwu4@cs.utexas.edu

Abstract

Private information retrieval (PIR) is a key building block in many privacy-preserving systems, and recent works have made significant progress on reducing the concrete computational costs of single-server PIR. However, existing constructions have high communication overhead, especially for databases with small records. In this work, we introduce Respire, a lattice-based PIR scheme tailored for databases of small records. To retrieve a single record from a database with over a million 256-byte records, the Respire protocol requires just 6.1 KB of online communication; this is a 5.9× reduction compared to the best previous lattice-based scheme. Moreover, Respire naturally extends to support batch queries. Compared to previous communication-efficient batch PIR schemes, Respire achieves a 3.4-7.1× reduction in total communication while maintaining comparable throughput (200-400 MB/s). The design of Respire relies on new query compression and response packing techniques based on ring switching in homomorphic encryption.

1 Introduction

A private information retrieval (PIR) protocol [CGKS95] enables a client to retrieve a record from a database without revealing to the database server which record she requested. In recent years, there have been significant advancements in constructing fast and practical PIR protocols and using PIR to realize applications to private certificate transparency auditing [HHC$^+$23], password breach checking [LPA$^+$19, TPY$^+$19, ALP$^+$21], metadata-hiding communication [MOT$^+$11, KLDF16, AS16, ACLS18], private web search [HDCZ23], and many more.

The communication overhead of PIR. While recent works [DPC23, HHC$^+$23, MSR23, LMRS24, ZPSZ24, MW24] have taken great strides in reducing the concrete *computational* costs of (single-server) PIR, existing constructions still incur high communication costs. For instance, retrieving a bit from a 1 GB database using the state-of-the-art SimplePIR protocol [HHC$^+$23] requires 240 KB of online communication. A protocol like Piano [ZPSZ24] requires 32 KB of online communication. These protocols additionally require the client to download a 121 MB hint (SimplePIR) or stream a 1 GB hint (Piano) in an offline phase.

Number-theoretic schemes such as Gentry-Ramzan [GR05] achieve smaller communication overhead (e.g., 2-18 KB of communication to retrieve a 288-byte record from a 1 MB database), but with high computational costs [ALP$^+$21]. In this setting, the server processes the database at a throughput of 20-350 KB/s, whereas the best lattice-based constructions have a server throughput ranging from hundreds of MB per second [MCR21, MW22a, DPC23] to multiple GB per second [HHC$^+$23, LMRS24, MW24].

In settings where records are large (tens of KB), protocols like OnionPIR [MCR21] and Spiral [MW22a] have low communication overhead. However, for many applications of PIR (e.g., anonymous messaging, private DNS, password breach checking, and more), the size of the payload the client is interested in ranges from tens of bytes (e.g., a hash value) to a few hundred bytes. In this regime, we do not have concretely-efficient PIR protocols with high server throughput and low communication costs.

The Respire protocol. This work introduces the Respire protocol, a lattice-based single-server PIR protocol for databases with small records (e.g., 256-byte records). Like recent PIR protocols based on the ring learning with errors (RLWE) problem [MBFK16, ACLS18, AYA$^+$21, ALP$^+$21, MCR21, MW22a, LMRS24, MW24], Respire works over polynomial rings. A key feature in the design of Respire is working over small *subrings* of the main polynomial ring. Working over a subring enables better query compression and response compression compared to all previous lattice-based schemes. For instance, retrieving a record from a database of over a million 256-byte records, Respire only needs 6.1 KB of online communication. Notably, the *total* communication is smaller than the size of even a single RLWE ciphertext in previous schemes. On the same configuration, the previous best lattice-based scheme (Spiral [MW22a]) requires 36 KB of communication (over 5.9× larger). In fact, the required communication in Respire is comparable to those based on group-based or factoring-based assumptions [ALP+21], but with 1000× higher server throughput. The throughput of Respire is 200-400 MB/s, which is just 26% slower than Spiral. Compared to high-throughput schemes like SimplePIR [HHC+23], Respire is 27-50× slower, but has 21-42× less communication; SimplePIR also requires the client to download a (reusable) hint (a few hundred MB) whereas Respire requires the client to *upload* a (reusable) hint (3.9 MB).

Batch queries. Combined with (probabilistic) batch codes [IKOS04, ACLS18] and a new response packing technique, Respire also extends to give a batch PIR protocol. Compared to previous lattice-based batch PIR protocols [MR23], Respire achieves a 3.4-7.1× reduction in total communication, and has higher throughput for small batch sizes (e.g., batch size up to 128 for a database with over 4 million 256-byte records). For larger batch sizes, there is a modest computational overhead ($\approx 2.2\times$) compared to previous protocols. Thus, Respire is well-suited for applications where the client is making a handful of queries simultaneously (e.g., blocklist lookup or DNS queries).

1.1 Our Techniques

The communication overhead in lattice-based PIR is due to the large lattice parameters needed for security. The aforementioned PIR schemes based on RLWE work over polynomial rings with dimension $d \geq 2048$. Typically, each coefficient of the polynomial is an element of $\mathbb{Z}_q$ where $q \geq 2^{32}$. This means that communicating even a single ring element (i.e., as needed to encode or encrypt a query index or a response) already requires 8 KB of communication. When the record size is only a few hundred bytes or smaller, sending even a single element introduces non-trivial communication overhead. For this reason, *all* previous lattice-based PIR schemes have query and response sizes that are over 10 KB. While it is tempting to use rings of smaller dimension to mitigate the communication overhead, this is often infeasible as the modulus q has to be chosen large enough to account for the noise accumulation inherent to lattice-based cryptosystems. In some sense, correctness imposes a minimum modulus q, which for security, translates to a minimal ring dimension d.

Leveraging subrings. The key technical idea underlying the design of Respire is that we can leverage ring switching techniques from homomorphic encryption [BV11, BGV12, GHPS12] to reduce query size and response size with only modest computational overhead. In the context of response compression, the approach we take in Respire is have the server perform most of its computations over the main ring R_1 (of dimension $d_1 = 2048$ and $q \approx 2^{56}$). However, before sending back the response, the server first *projects* the response into a *subring* $R_2 \subseteq R_1$ of much smaller dimension $d_2 = 512$ (and also with respect to a smaller modulus). Critically, this approach only works when the records are small (as the projection operation necessarily *loses* some information about the value encoded over the big ring). Since $d_1/d_2 = 4$, this yields a 4× reduction in response size. Note that we are unable to directly work over the small ring due to the constraints on the dimension d and the modulus q imposed by correctness and security (see Remark 3.1).

Query compression. Working over subrings also provides us a way to achieve query compression. Many RLWE-based PIR protocols leverage the query packing techniques from [ACLS18, CCR19] to pack multiple scalars into a *single* encoded polynomial. In this work, we show that the projection operations used for response compression can also be used for query compression. Namely, while a single element of the big ring R_1 can encode $d_1 = 2048$ scalars, if the protocol only needs $h \ll d_1$ values, then there is again wasted space. In this work, we show that if we embed the h coefficients in a *subring* $R_2 \subset R_1$, then it suffices to send $\approx h$ coefficients to the server. This allows us to reduce the query size from 14 KB to 4 KB. Notably, the query is now *smaller* than even a *single* element in the big ring R_1. All previous RLWE-based PIR schemes required communicating at least one complete ring element in the query. Our work shows that this is *not* essential. We believe this technique will be independently useful in other settings that apply the query packing technique [ACLS18, CCR19] to a small number of inputs.

Starting point: the Spiral protocol. The Respire protocol builds on top of the Spiral protocol [MW22a] (the lattice-based PIR protocol with the best communication). Very briefly, the Spiral protocol arranges the database as a $(1 + \nu_2)$-dimensional hypercube, where the first dimension has size 2^{ν_1} and the remaining dimensions have size 2. A record is indexed by a tuple $(\alpha, \beta_1, \ldots, \beta_{\nu_2})$ where $\alpha \in [2^{\nu_1}]$ and $\beta_1, \ldots, \beta_{\nu_2} \in \{0, 1\}$. The query consists of an RLWE encryption [Reg05, LPR10] of 2^{ν_1} (as a one-hot vector) and encryptions of $\beta_1, \ldots, \beta_{\nu_2}$ using the GSW encryption scheme [GSW13]. The server homomorphically uses the query ciphertexts to select along each dimension and the final output is an RLWE encryption of the record of interest.

The Respire protocol. The Respire protocol integrates our query compression and response compression techniques within Spiral (see Section 3). On a 256 MB database (with a million 256-byte records), our compression techniques reduces the query size by 3.9× and the response size by 10× while maintaining the same server throughput. On larger databases (with the same record size), we reduce the total communication by 4.5× at a cost of an 1.3× increase in server response time. We provide more details in Section 4. Note that the Respire query compression and response compression techniques are tailored for the setting of small database elements. When the database elements are sufficiently large (concretely, on the order of a few KB), then our compression techniques no longer provide any savings and the Respire protocol is equivalent to Spiral (or the SpiralPack variant of Spiral).

Supporting batch queries. When the client seeks to retrieve a batch of T records from the database, we can achieve better communication by *packing* multiple responses into a single ring element. In Section 3.2, we show how the subring embedding and projection machinery we developed can also be used to homomorphically *repack* multiple responses into a single ciphertext. We then compose with probabilistic batch codes [IKOS04, ACLS18] (which allow us to amortize some of the server processing costs). Our scheme is particularly well-suited for small batches of queries (e.g., $T = 16$), whereas previous lattice-based batch PIR schemes [MR23, LLWR24] are more efficient for large batch sizes. We refer to Remark 3.5 (and Section 4.3) for a more detailed comparison.

2 Preliminaries

We write λ to denote the security parameter. For an integer $n \in \mathbb{N}$, we write $[n] := \{1, \ldots, n\}$. For integers $a, b \in \mathbb{N}$, we write $[a, b] := \{a, \ldots, b\}$. For integers $x, y \in \mathbb{N}$, we write $x \mid y$ to denote that x divides y. We use bold lowercase letters to denote vectors (e.g., $\mathbf{u}$, $\mathbf{v}$) and bold uppercase letters (e.g., $\mathbf{A}$, $\mathbf{B}$) to denote matrices. We write $\mathbf{u}_i$ to denote the i^{th} elementary basis vector. For a dimension $d \in \mathbb{N}$, we write $\mathbf{I}_d$ to denote the d-by-d identity matrix. For a distribution $\mathcal{D}$, we write $x \leftarrow \mathcal{D}$ to denote drawing a sample x from $\mathcal{D}$. For a finite set S, we write $x \xleftarrow{\text{R}} S$ to denote a uniform random sample from S.

Private information retrieval. We recall the definition of a single-server two-message private information retrieval (PIR) protocol [CGKS95] in the "client hint" model (where the client uploads a *reusable* query key in an offline phase prior to making queries). We also allow a (silent) server preprocessing step where the server prepares the database so as to be able to efficiently answer queries in the online phase.

Definition 2.1 (Private Information Retrieval [CGKS95, adapted]). Let λ be a security parameter, $N = N(\lambda)$ be the number of records in the database, and $\mathcal{M} = \{\mathcal{M}_\lambda\}$ be the space of possible record values. A two-message single-server private information retrieval (PIR) protocol in the client-hint model is a tuple of efficient algorithms $(\mathsf{Setup}, \mathsf{SetupDB}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$ with the following syntax:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{qk})$: On input a security parameter λ, the setup algorithm outputs parameters pp and a query key qk.
- $\mathsf{SetupDB}(1^\lambda, \{d_i\}_{i \in [N]}) \to \mathsf{db}$: On input the security parameter λ and a collection of records $d_1, \ldots, d_N \in \mathcal{M}_\lambda$, the database setup algorithm outputs a (preprocessed) database db.
- $\mathsf{Query}(\mathsf{qk}, \mathsf{idx}) \to \mathsf{q}$: On input the query key qk and an index idx, the query algorithm outputs a query q.

- $\mathsf{Answer}(\mathsf{pp}, \mathsf{db}, \mathsf{q}) \to \mathsf{a}$: On input the parameters pp, a preprocessed database db, and the query q, the answer algorithm outputs an answer a.
- $\mathsf{Extract}(\mathsf{qk}, \mathsf{a}) \to d_i$: On input the query key qk and the answer a, the extract algorithm outputs a record d_i.

The algorithms must satisfy the following properties:

- **Correctness:** For all security parameters $\lambda \in \mathbb{N}$, all sets of records $d_1, \dots, d_N \in \mathcal{M}_\lambda$, and all indices $\mathsf{idx} \in [N]$,

$$\Pr\left[\mathsf{Extract}(\mathsf{qk}, \mathsf{a}) = d_{\mathsf{idx}} : \begin{array}{c} (\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda) \\ \mathsf{db} \leftarrow \mathsf{SetupDB}(1^\lambda, d_1, \dots, d_N) \\ \mathsf{q} \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx}) \\ \mathsf{a} \leftarrow \mathsf{Answer}(\mathsf{pp}, \mathsf{db}, \mathsf{q}) \end{array}\right] \geq 1 - c.$$

  We refer to c as the correctness error.

- **Query privacy:** For a bit $b \in \{0, 1\}$ and an adversary $\mathcal{A}$, we define the query privacy game as follows:
  - The challenger starts by sampling $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and gives pp to $\mathcal{A}$.
  - Algorithm $\mathcal{A}$ can now make (arbitrarily many) queries on pairs of indices $(\mathsf{idx}_0, \mathsf{idx}_1)$ where $\mathsf{idx}_0, \mathsf{idx}_1 \in [N]$. On each query, the challenger responds with $\mathsf{Query}(\mathsf{qk}, \mathsf{idx}_b)$.
  - When $\mathcal{A}$ is done making queries, it outputs a bit $b' \in \{0, 1\}$, which is the output of the experiment.

  We say that the scheme satisfies query privacy if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$|\Pr[b' = 1 : b = 0] - \Pr[b' = 1 : b = 1]| = \mathsf{negl}(\lambda).$$

Remark 2.2 (Batch PIR). Definition 2.1 considers a setting where the client queries for a *single* record at a time. In batch PIR [BIM00, IKOS04], the client can make a query for a batch of k records, given by indices $(\mathsf{idx}_1, \dots, \mathsf{idx}_k)$, and receives a single answer a from which all k records can be extracted. While this functionality can be realized by running k (parallel) invocations of the single-query protocol, there are many techniques [BIM00, IKOS04, GKL10, AS16, LG15, Hen16, ACLS18, MR23, Yeo23, BPSY24] to reduce the communication and computation costs in the batch setting.

2.1 Lattice Preliminaries

Like many previous lattice-based PIR protocols [MBFK16, ACLS18, GH19, MCR21, MW22a, MR23, MW24], we work over polynomial rings. In this section, we provide a high-level description of the lattice algorithms we use and defer the formal details to Appendix A. Throughout this work, we write R_d to denote the polynomial ring $R_d := \mathbb{Z}[x]/(x^d + 1)$. For an integer $q \in \mathbb{N}$, we write $R_{d,q} := \mathbb{Z}_q[x]/(x^d + 1)$. When $q = 1 \bmod 2d$, we can use the number-theoretic transform (NTT) to efficiently implement polynomial multiplication over $R_{d,q}$ [LMPR08, LN16].

Rounding. We write $\lfloor \cdot \rfloor : \mathbb{R} \to \mathbb{Z}$ to denote the floor function and $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ to denote the function that rounds the input to the nearest integer. For positive integers $q > p$, we write $\lfloor \cdot \rceil_{q,p} : \mathbb{Z}_q \to \mathbb{Z}_p$ to denote the function that takes as input $x \in \mathbb{Z}_q$, lifts it to an integer $x' \in (-q/2, q/2]$, and outputs $\lfloor p/q \cdot x' \rceil$. We extend each of these operations to the ring R_d by component-wise evaluation on the coefficients of the input $r \in R_d$. We also extend the operation to vectors and matrices via component-wise evaluation.

Gadget matrices. For a modulus $q \in \mathbb{N}$ and a decomposition base $z \in \mathbb{N}$, the gadget vector [MP12] is $\mathbf{g}_z := [1, z, z^2, \dots, z^{t-1}] \in \mathbb{Z}_q^t$ where $t = \lfloor \log_z q \rfloor + 1$. For a dimension $n \in \mathbb{N}$, the gadget matrix is $\mathbf{G}_{n,z} := \mathbf{I}_n \otimes \mathbf{g}_z^\mathsf{T} \in \mathbb{Z}_q^{n \times nt}$. We write $\mathbf{g}_z^{-1} : \mathbb{Z}_q \to \mathbb{Z}_q^t$ and $\mathbf{G}_{n,z}^{-1} : \mathbb{Z}_q^n \to \mathbb{Z}_q^{nt}$ to denote the base-z digit decomposition operator that takes the input and expands each component into its base-z representation with each digit in the centered interval $(-z/2, z/2]$. We extend $\mathbf{g}_z^{-1}$ and $\mathbf{G}_{n,z}^{-1}$ to operate on vectors and matrices, respectively, by column-wise evaluation. Both $\mathbf{g}_z$ and $\mathbf{G}_{n,z}$ are defined identically over the ring $R_{d,q}$.

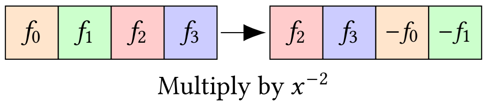


Figure 1: Illustration of the rotation operation over the polynomial ring $R_4 = \mathbb{Z}[x]/(x^4+1)$. We model polynomials $f(x) = \sum_{i \in [0,3]} f_i x^i$ as a vector of coefficients $(f_0, f_1, f_2, f_3) \in \mathbb{Z}^4$.

Ring learning with errors. The security of Respire relies on the ring learning with errors (RLWE) assumption [Reg05, LPR10]:

Definition 2.3 (Ring Learning With Errors [Reg05, LPR10])**.** Let $R_d = \mathbb{Z}[x]/(x^d+1)$ where $d = d(\lambda)$ is a power of two. Let $m = m(\lambda)$ be the number of samples, $q = q(\lambda)$ be the modulus, and $\chi_s, \chi_e = \chi(\lambda)$ be distributions over $R_{d,q}$. The ring learning with errors (RLWE) assumption $\mathsf{RLWE}_{d,m,q,\chi_s,\chi_e}$ states that the following distributions are computationally indistinguishable:

$$\left\{(\mathbf{a}, s\mathbf{a}+\mathbf{e}) : \mathbf{a} \xleftarrow{R} R_{d,q}^m, s \leftarrow \chi_s, \mathbf{e} \leftarrow \chi_e^m\right\} \quad \text{and} \quad \left\{(\mathbf{a}, \mathbf{u}) : \mathbf{a}, \mathbf{u} \xleftarrow{R} R_{d,q}^m\right\}.$$

RLWE encodings. We say that $\mathbf{c} = \left[\begin{smallmatrix} a \\ sa+e+\mu \end{smallmatrix}\right] \in R_{d,q}^2$ is an RLWE encoding of a scalar $\mu \in R_{d,q}$ with respect to a secret key $\mathbf{s} = [-s \mid 1]^\intercal \in R_{d,q}^2$ and error $e \in R_{d,q}$ if $\mathbf{s}^\intercal \mathbf{c} = \mu + e \bmod q$. Under the RLWE assumption, the encoding $\mathbf{c}$ is pseudorandom and hides the encoded value μ. When we write $\mathbf{c} = [c_1, c_2]^\intercal$, we refer to c_1 as the "random" component of the encoding and c_2 as the "message-embedding" component of the encoding. RLWE encodings are additively homomorphic: if $\mathbf{c}_1, \mathbf{c}_2$ are RLWE encodings of μ_1, μ_2 with respect to the same secret key $\mathbf{s}$ and errors e_1, e_2, then $\mathbf{c}_1 \pm \mathbf{c}_2$ is an RLWE encoding of $\mu_1 \pm \mu_2$ with respect to the same secret key $\mathbf{s}$ and error $e_1 \pm e_2$. In many cases, $\mu = \lfloor q/p \rfloor m$ for some value $m \in R_{d,p}$. Given $\lfloor q/p \rfloor m + e$, it is possible to recover m, provided that $|e|$ is small. We state the following theorem adapted from [MW22a]:

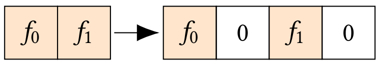
Theorem 2.4 (Message Decoding [MW22a, Theorem 2.11])**.** *Let $R_d = \mathbb{Z}[x]/(x^d+1)$. Suppose $z = \lfloor q/p \rfloor m + e \in R_{d,q}$ where $|m| < p$ and $|e| < \frac{q}{2p} - (q \bmod p)$. Then, $\lfloor z \rceil_{q,p} = m$.*

GSW encodings. Like several recent PIR protocols [GH19, MCR21, MW22a] our construction also relies on the encryption scheme of Gentry, Sahai, and Waters (GSW) [GSW13]. Let $z \in \mathbb{N}$ be a decomposition base, $m = 2(\lfloor \log_z q \rfloor + 1)$, and $\mathbf{G}_{2,z} \in R_{d,q}^{2 \times m}$ be the gadget matrix. We say that $\mathbf{C} \in R_{d,q}^{2 \times m}$ is a GSW encoding of a bit $\mu \in \{0,1\}$ with respect to a secret key $\mathbf{s} \in R_{d,q}^2$, error $\mathbf{e} \in R_{d,q}^m$, and decomposition base $z \in \mathbb{N}$ if $\mathbf{s}^\intercal \mathbf{C} = \mu \mathbf{s}^\intercal \mathbf{G}_{2,z} + \mathbf{e}^\intercal \bmod q$.

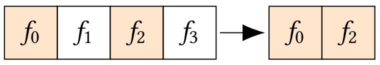
Homomorphic selection. The external product [CGGI18, CGGI20] operation provides a way to homomorphically multiply an RLWE encoding with a GSW encoding. The external product implies a lightweight homomorphic selection operation used in several previous PIR protocols [GH19, MCR21, MW22a]. Specifically, given a GSW encoding of a selection bit $b \in \{0,1\}$ and RLWE encodings of messages $\mu_0, \mu_1 \in R_{d,q}$, the homomorphic selection operation outputs an RLWE encoding of μ_b by homomorphically computing $\mu_b := \mu_0 + b(\mu_1 - \mu_0)$. We model the algorithm as follows and give the full details in Appendix A:

- $\mathsf{Select}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_0, \mathbf{c}_1) \to \mathbf{c}'$: On input a GSW encoding $\mathbf{C}_{\mathsf{GSW}} \in R_{d,q}^{2 \times m}$ and RLWE encodings $\mathbf{c}_0, \mathbf{c}_1 \in R_{d,q}^2$, the selection algorithm outputs an RLWE encoding $\mathbf{c}'$.

Rotations. We associate polynomials $f(x) = \sum_{i \in [0,d-1]} f_i x^i \in R_d$ with their coefficient vector $[f_0, f_1, \ldots, f_{d-1}] \in \mathbb{Z}^d$. Then, multiplication by a monomial x^k corresponds to a (nega)-cyclic left rotation of the coefficient vector. Namely, if $g = x^k f$, then the coefficient vector of g is $[f_k, \ldots, f_{d-1}, -f_0, \ldots, -f_{k-1}]$. We illustrate this procedure in Fig. 1. Thus, given an RLWE encoding $\mathbf{c}$ of a polynomial f, the encoding $x^k \mathbf{c}$ encodes the polynomial $g = x^k f$. Note that this operation does *not* affect the norm of the noise in the resulting encoding.



(a) The subring embedding $\kappa\colon R_2 \to R_4$ from Eq. (3.1).



(b) The dimension-reduction map $\kappa^{-1}\colon R_4 \to R_2$ from Eq. (3.2).

Figure 2: Illustrations of the ring embedding and dimension reduction between $R_4 = \mathbb{Z}[x]/(x^4+1)$ and $R_2 = \mathbb{Z}[x]/(x^2+1)$ We model polynomials $f(x) = \sum_{i\in[0,d-1]} f_i x^i$ as a vector of coefficients $(f_0,\ldots,f_{d-1}) \in \mathbb{Z}^d$.

3 The Design of Respire

In this section we introduce the Respire protocol. We start by providing a high-level overview of the main building blocks we use in Respire. We then give the full Respire protocol in Section 3.1 and a generalization to batch queries in Section 3.2. For ease of exposition, throughout this section, we elect to focus on the syntax and functionality of the main algorithms we use and elide their implementation details. The formal description of these algorithms along with their analysis are provided in Appendices A to C.

Respire design. Like many RLWE-based PIR protocols [MBFK16, ACLS18, AYA⁺21, ALP⁺21, MCR21, MW22a, LMRS24, MW24], Respire works over a power-of-two cyclotomic ring $R_d = \mathbb{Z}[x]/(x^d+1)$. This means that plaintext elements (e.g., a record) and RLWE encodings both have dimension d. The combination of correctness and security constraints limit the choices for the ring dimension, and recent RLWE-based schemes all use rings where $d \geq 2048$. Since the focus of Respire is PIR for databases with short records, a single ring element is generally (much) larger than a single record. For instance, if we use a 4-bit plaintext modulus p, then each element of $R_{d,p} := \mathbb{Z}_p[x]/(x^d+1)$ is at least 1 KB. When the database records are much smaller than 1 KB, it is wasteful for both computation and communication to use a single element of $R_{d,p}$ to represent a single database record. In Respire, we use record packing together with dimension reduction to reduce this overhead.

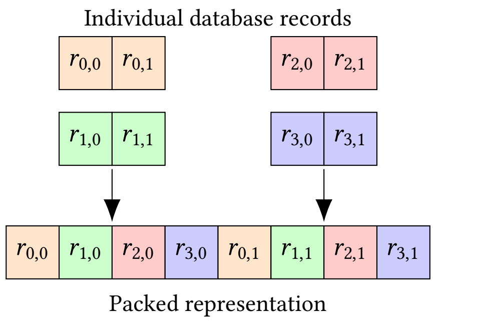
- **Record packing:** In Respire, we consider two different rings: a "large" ring R_{d_1} of dimension d_1 for the RLWE encodings and a subring $R_{d_2} \subset R_{d_1}$ for individual database records. We pack *multiple* database records (specifically, $k = d_1/d_2$ records) into each RLWE encoding.
- **Dimension reduction:** While encoding multiple records in a single RLWE encoding reduces the number of RLWE encodings the server needs to operate on when answering a query, it does *not* help with communication. The encoded response is still an RLWE encoding, which resides in the large ring R_{d_1}. To reduce the communication, we leverage the "ring switching" techniques from [BV11, BGV12, GHPS12] to reduce this overhead. Specifically, the encoded queries (and the bulk of the server processing) occur over R_{d_1}, but before sending the encoded response back to the client, the server switches the response to an RLWE encoding over the subring R_{d_2}. To distinguish this operation from other transformations that translate encodings between rings, we refer to this operation as *dimension reduction.* As we note in Remark 3.1, the constraints on the parameters prevent us from using RLWE encodings over R_{d_2} throughout the protocol; thus, dimension reduction is critical for reducing the communication costs in Respire.

Dimension reduction must lose information about the underlying encoded plaintext (since it projects onto a ring of smaller dimension). Thus, it is essential that the records are packed in a way that still allows efficient recovery of any of the packed records. To discuss how, let $k = d_1/d_2$; we introduce the following two functions:

- **Subring embedding:** The subring embedding function $\kappa\colon R_{d_2} \to R_{d_1}$ is the mapping

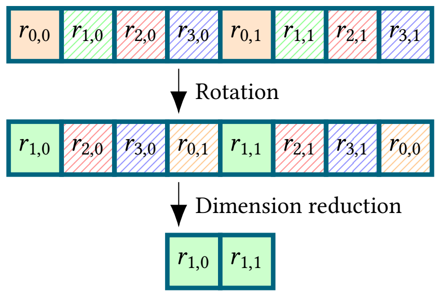
$$\kappa\left(\sum_{i\in[0,d_2-1]} f_i x^i\right) := \sum_{i\in[0,d_2-1]} f_i x^{k\cdot i}. \tag{3.1}$$

  The subring embedding of f sends the ith coefficient of the input polynomial onto the $(k\cdot i)^{\text{th}}$ coefficient of the output polynomial. We illustrate this in Fig. 2a.



Individual database records
$r_{0,0}$ $r_{0,1}$
$r_{2,0}$ $r_{2,1}$
$r_{1,0}$ $r_{1,1}$
$r_{3,0}$ $r_{3,1}$
$r_{0,0}$ $r_{1,0}$ $r_{2,0}$ $r_{3,0}$ $r_{0,1}$ $r_{1,1}$ $r_{2,1}$ $r_{3,1}$
Packed representation

(a) Representation of a single packed database element $\Pi(r_0, r_1, r_2, r_3)$. Each record $r_i = (r_{i,0}, r_{i,1})$ consists of two elements of $\mathbb{Z}_p$. The database records are packed in the *clear* during database preprocessing.



$r_{0,0}$ $r_{1,0}$ $r_{2,0}$ $r_{3,0}$ $r_{0,1}$ $r_{1,1}$ $r_{2,1}$ $r_{3,1}$
Rotation
$r_{1,0}$ $r_{2,0}$ $r_{3,0}$ $r_{0,1}$ $r_{1,1}$ $r_{2,1}$ $r_{3,1}$ $r_{0,0}$
Dimension reduction
$r_{1,0}$ $r_{1,1}$

(b) Extracting the correct record from an RLWE encoding of a packed database element. This is implemented *homomorphically* during Answer. The coefficients in "striped" boxes (i.e. those not in the initial position of the packed representation) are lost during dimension reduction (i.e., response compression). In this example, the client wants record r_1, so the server first applies a (homomorphic) rotation to the encoded coefficient vector followed by dimension reduction. For simplicity, we omit the sign changes from the rotation.

Figure 3: Illustration of how Respire packs small database records in such a way that they can be retrieved homomorphically. In this example, the main ring dimension is $d_1 = 8$, and the reduced dimension/record dimension is $d_2 = 2$. Elements with a bold **blue** border are *encrypted*.

- **Dimension reduction:** We define the dimension-reduction mapping $\kappa^{-1} : R_{d_1} \to R_{d_2}$ to be the mapping

$$\kappa^{-1}\left(\sum_{i \in [0, d_1 - 1]} f_i x^i\right) := \sum_{i \in [0, d_2 - 1]} f_{k \cdot i} x^i. \tag{3.2}$$

  This operation sends the $(k \cdot i)^{\text{th}}$ coefficient of the input to the i^{th} coefficient of the output, and drops all other coefficients. We illustrate the dimension-reduction function in Fig. 2b. Note that κ^{-1} is only a *one-sided* inverse of κ: namely $\kappa^{-1}(\kappa(f)) = f$ for all $f \in R_{d_2}$, the converse does not hold.

Ring packing. The subring embedding gives a natural way to pack multiple records (from the subring R_{d_2}) into a single element of the full ring R_{d_1}. Specifically, we define the ring packing function $\Pi : R_{d_2}^k \to R_{d_1}$ that takes as input a set of k records $r_0, \ldots, r_{k-1} \in R_{d_2}$ and outputs a single element over R_{d_1} as follows:

$$\Pi(r_0, \ldots, r_{k-1}) := \sum_{i \in [0, k-1]} x^i \cdot \kappa(r_i). \tag{3.3}$$

We provide a visual depiction of how we use Π to pack multiple database records into a single record in Fig. 3a.

Manipulating packed ring elements. Given a packed record $\tilde{r} = \Pi(r_0, \ldots, r_{k-1}) \in R_{d_1}$, we refer to r_j as the record at position j within $\tilde{r}$. We refer to record r_0 as the record at the *initial* position. There are two operations we perform on packed encodings:

- **Extracting the record in the initial position:** By construction, the dimension-reduction mapping (Eq. (3.2)) can be applied to a packed element to recover the element at the initial position: namely, $\kappa^{-1}(\Pi(r_0, \ldots, r_{k-1})) = r_0$. This follows by inspection, and we defer to Appendix A.2 for a formal analysis.
- **Rotating the elements:** As noted in Section 2, multiplying a polynomial f by $x^{-\ell}$ implements a nega-cyclic rotation on the coefficients of f (see Fig. 1). In particular, for all $\ell \in [0, k-1]$, we have that $x^{-\ell} \cdot \tilde{r} = \Pi(r_\ell, \ldots, r_{k-1}, -r_0, \ldots, -r_{\ell-1})$.

We can compose rotation and dimension reduction to extract the record at any position $\ell \in [0, k-1]$ from a packed record $\tilde{r}$ (see Fig. 3b). Moreover, if both operations can be implemented homomorphically on an RLWE encoding of the packed record $\tilde{r}$, then the server can homomorphically extract the requested record. Namely, given an RLWE encoding of the packed record $\tilde{r}$ as well as an encoding of the index ℓ, the server homomorphically derives an encoding of $\kappa^{-1}(x^{-\ell} \cdot \tilde{r})$, which is precisely an encoding of the desired record r_ℓ.

Homomorphic rotation. As shown in Fig. 3b, to extract the client's record of interest from an RLWE encoding of a packed record $\tilde{r}$, the server must first homomorphically compute the product $x^{-\ell} \cdot \tilde{r}$. Since the index ℓ is private, the client must provide it in encoded form. One possibility is to have the client send a GSW encoding of $x^{-\ell}$ as part of its query; then the server can use the external product to compute an RLWE encoding of the product $x^{-\ell} \cdot \tilde{r}$. However, GSW encodings are *large* and our query compression technique only gives us the ability to compress GSW encodings of bits (and *not* monomials). Thus, in Respire, we take a different approach. Let $\ell_t \cdots \ell_0$ be the binary representation of ℓ. Then we can write $x^{-\ell} = \prod_{i \in [0,t]} x^{-\ell_i 2^i}$. The client now provides as GSW encodings of the *bits* $\ell_0, \ldots, \ell_t$ in its query. These are GSW encodings of *bits*, so they can be packed into a small number of RLWE encodings (see below and also Appendix B). During the evaluation phase, the server uses the encodings of each ℓ_i to homomorphically select between either $\tilde{r}$ (if $\ell_i = 0$) or $\tilde{r} \cdot x^{-2^i}$ (if $\ell_i = 1$). We can implement the homomorphic rotations using $t+1$ calls to Select and communicating $t+1$ (compressed) GSW encodings (see Construction 3.2).

Homomorphic dimension reduction. Next, we use the ring-switching technique from [BV11, BGV12, GHPS12] to homomorphically apply dimension reduction. Specifically, these works show how to transform an RLWE encoding of a polynomial f_1 over R_{d_1,q_1} (under a key $\mathbf{s}_1 \in R^2_{d_1,q_1}$) to an RLWE encoding of the polynomial $\kappa^{-1}(f_2)$ over R_{d_2,q_2} (under a key $\mathbf{s}_2 \in R^2_{d_2,q_2}$), where κ^{-1} is the dimension-reduction map (Eq. (3.2)). To do so, one essentially publishes an encryption of the components of the source key $\mathbf{s}_1$ under the target key $\mathbf{s}_2$; this is the "key-switching matrix" for translating from R_{d_1,q_1} to R_{d_2,q_2}. It is critical that the target modulus q_2 be much smaller than the source modulus q_1. This is because the key-switching parameters consist of an RLWE encoding over the *smaller* ring R_{d_2}, and security relies on the hardness of RLWE in the smaller ring. Thus, to perform dimension reduction, we apply the following two steps:

1. **Modulus switching:** We first perform modulus reduction to scale the input RLWE modulus from q_1 to q_2. This operations transforms an encoding over R_{d_1,q_1} to one over R_{d_1,q_2}.

2. **Dimension reduction:** Then, we apply dimension reduction to scale the dimension from d_1 to d_2. This operation transforms an encoding over R_{d_1,q_2} to one over R_{d_2,q_2}.

To achieve better compression, we use the "split" modulus switching approach from [MW22a], where the message-embedding component of an RLWE encoding is further scaled to a smaller modulus $q_3 < q_2$. As such, the resulting message-embedding component is an element of R_{d_2,q_3}. Taken together, we refer to the combined modulus switching and dimension reduction procedure as response compression. We give the syntax of our response compression algorithm below, but defer their formal description and analysis to Construction C.3 and Appendix C.2. In the formal description in Appendix C.2, we extend these algorithms to work over RLWE encodings of *vectors* (defined formally in Appendix C.1), as the generalization will be useful when considering batch queries.

Box 1: Response Compression Algorithms

- $\mathsf{CompressSetup}(1^\lambda, \mathbf{s}_1, \mathbf{s}_2) \rightarrow \mathsf{pp}_{\mathsf{comp}}$: On input a security parameter λ, a source key $\mathbf{s}_1 \in R^2_{d_1,q_2}$ and a target key $\mathbf{s}_2 \in R^2_{d_2,q_2}$, the setup algorithm outputs a set of compression parameters $\mathsf{pp}_{\mathsf{comp}}$.
- $\mathsf{Compress}(\mathsf{pp}_{\mathsf{comp}}, \mathbf{c})$: On input the compression parameters $\mathsf{pp}_{\mathsf{comp}}$ and an encoding $\mathbf{c} \in R^2_{d_1,q_1}$, the compression algorithm outputs a new encoding (c'_1, c'_2) where $c'_1 \in R_{d_2,q_2}$ and $c'_2 \in R_{d_2,q_3}$. This algorithm interleaves split modulus switching with dimension reduction. In particular, the message-embedding component c'_2 of the output encoding is over the ring R_{d_2,q_3} where $q_3 \leq q_2 \leq q_1$.
- $\mathsf{CompressRecover}(\mathbf{s}_2, (c'_1, c'_2)) \rightarrow z$: On input a secret key $\mathbf{s}_2$ and a compressed encoding (c'_1, c'_2), the compression algorithm outputs a value $z \in R_{d_2,q_3}$. This algorithm recovers the encoded value from the RLWE encoding.

Remark 3.1 (The Need for Dimension Reduction)**.** An alternative to working over a large ring R_{d_1} and translating the output to the smaller ring R_{d_2} is to just perform all of the operations over the small ring R_{d_2}. However, working over a smaller ring R_{d_2} limits us to a smaller modulus q_2 (since we need to rely on RLWE hardness over the small ring). This is insufficient for correctness (i.e., the noise accumulated from query processing is too high). Indeed, previous PIR schemes based on RLWE (e.g., [ACLS18, MCR21, MW22a]) required a ring dimension of at least $d_1 \geq 2048$. By combining (split) modulus reduction with dimension reduction, we are able to use the larger ring for query processing (e.g., $d_1 = 2048$), but reduce to a smaller ring when communicating the final response (e.g., $d_2 = 512$). This allows us to achieve substantially smaller responses in Respire compared to previous constructions.

Query compression. The queries in Respire consist of RLWE encodings and GSW encodings of the client's desired index. To reduce communication, we need a way to "compress" these query encodings. Here, we rely on the approach of Angel et al. [ACLS18] who showed how to homomorphically expand an RLWE encoding of a polynomial $f(x) = \sum_{i \in [d]} \mu_i x^{i-1} \in R_{d,q}$ into d RLWE encodings of the coefficients $\mu_1, \ldots, \mu_d$. The procedure can be further adapted to "pack" GSW encodings into a small number of RLWE encodings [CCR19, MCR21, MW22a]. This packing procedure avoids the need to communicate large GSW encodings at the expense of a modest amount of computation on the server to unpack the queries. In this work, we abstract out these compression algorithms as follows:

Box 2: Query Packing Algorithms

- $\mathsf{QueryPackSetup}(1^\lambda, \mathbf{s}) \rightarrow \mathsf{pp}_{\mathsf{qpk}}$: On input a security parameter λ and a secret key $\mathbf{s} \in R^2_{d,q}$, the setup algorithm outputs a set of packing parameters $\mathsf{pp}_{\mathsf{qpk}}$.
- $\mathsf{QueryPack}(\mathbf{s}, \mathbf{v}, \boldsymbol{\mu}) \rightarrow \mathsf{enc}$: On input a secret key $\mathbf{s} \in R^2_{d,q}$ and inputs $\mathbf{v} \in \mathbb{Z}^k_q$, $\boldsymbol{\mu} \in \{0,1\}^\ell$, the query packing algorithm outputs a packed encoding enc.
- $\mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{enc}) \rightarrow (\mathbf{c}_1, \ldots, \mathbf{c}_k), (\mathbf{C}_1, \ldots, \mathbf{C}_\ell)$: On input the public parameters $\mathsf{pp}_{\mathsf{qpk}}$, a packed encoding enc, the query unpacking algorithm outputs a tuple of RLWE encodings $\mathbf{c}_1, \ldots, \mathbf{c}_k$ and a tuple of GSW encodings $\mathbf{C}_1, \ldots, \mathbf{C}_\ell$.

If $\mathsf{enc} \leftarrow \mathsf{QueryPack}(\mathbf{s}, \mathbf{v}, \boldsymbol{\mu})$, then the RLWE encodings $\mathbf{c}_1, \ldots, \mathbf{c}_k$ output by $\mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{enc})$ encode $v_1, \ldots, v_k \in \mathbb{Z}_q$ while the GSW encodings $\mathbf{C}_1, \ldots, \mathbf{C}_\ell$ output by $\mathsf{QueryUnpack}$ encode $\mu_1, \ldots, \mu_\ell \in \{0,1\}$. Both sets of encodings are with respect to the secret key $\mathbf{s}$ and have a small error. We refer to Construction B.6 in Appendix B for the details of these algorithms.

Further compression using subrings. The Angel et al. query compression approach can be used to compress d RLWE encodings into a single RLWE encoding. This means the query still consists of at least a single ring element (i.e., an element of $R_{d,q}$).[1] When the number of RLWE encodings h we seek to pack is much smaller than d, then communicating even a single ring element incurs high overhead. In this work we show that it is *not* necessary to send the full RLWE encoding. When $h \ll d$, the client can instead embed h into the coefficients of a polynomial that lives in a *subring* of $R_{d,q}$. Instead of sending a full RLWE encoding of the polynomial, the client can send an smaller encoding that only retains information about the polynomial's coefficients in the subring. Formally, this is the encoding obtained by applying the dimension-reduction mapping (Eq. (3.2)) to the packed encoding. As such, to pack h coefficients into an RLWE encoding, the client now only needs to send roughly h coefficients to the server. For our parameter sets, this yields a substantial concrete reduction in query size: $d/h \approx 3.5$ (e.g., from 14 KB to 4 KB). We defer the formal details of our approach to Appendix B.

3.1 The Respire Protocol

We now describe the Respire protocol. At a high level, Respire combines the general approach from [GH19, MCR21, MW22a] with our improved query compression and response compression techniques to reduce communication (with modest impact to throughput). In particular, Respire uses the approach from [GH19, MCR21, MW22a] to retrieve a packed record (this is the first-dimension processing and folding steps in Construction 3.2), and then applies the homomorphic rotation and dimension reduction techniques to return the desired record.

Database structure. In Respire, the RLWE encodings are over the ring $R_{d_1,q_1} := \mathbb{Z}_{q_1}[x]/(x^{d_1}+1)$ and the database are elements of the ring $R_{d_2,p} := \mathbb{Z}_p[x]/(x^{d_2}+1)$. We require that $d_1 = 2^{\delta_1}$ and $d_2 = 2^{\delta_2}$ and $d_1 \geq d_2$. We view the database as a hypercube with $1 + \nu_2 + \nu_3$ dimensions where the first dimension has size 2^{ν_1}, the remaining $\nu_2 + \nu_3$ dimensions each have size 2, and $\nu_3 = \delta_1 - \delta_2$. In the protocol, we represent the database as $2^{\nu_1+\nu_2}$ elements of $R_{d_1,p} := \mathbb{Z}_p[x]/(x^{d_1}+1)$, where each element of $R_{d_1,p}$ is a packed representation of $2^{\nu_3} = d_1/d_2$ individual database records (see Fig. 3a).

Construction 3.2 (Respire)**.** Let λ be a security parameter. The Respire scheme is parameterized by the following:

- **Lattice parameters:** Let $d_1 = d_1(\lambda)$ and $d_2 = d_2(\lambda)$ denote the full ring dimension and the reduced ring dimension, respectively. We require that $d_1 = 2^{\delta_1}$ and $d_2 = 2^{\delta_2}$ for some $\delta_1, \delta_2 \in \mathbb{N}$ and $d_1 \geq d_2$. Let $q_1 = q_1(\lambda)$, $q_2 = q_2(\lambda)$, and $q_3 = q_3(\lambda)$ be moduli where $q_1 \geq q_2 \geq q_3$. Let $\chi_{1,e} = \chi_{1,e}(\lambda)$ and $\chi_{1,s} = \chi_{1,s}(\lambda)$ be distributions over R_{d_1,q_1}. Let $\chi_{2,e} = \chi_{2,e}(\lambda)$, $\chi_{2,s} = \chi_{2,s}(\lambda)$ be distributions over R_{d_2,q_2}.
- **Plaintext modulus:** Let p be the plaintext modulus. Each database record is an element of $R_{d_2,p}$.
- **Database configuration:** Let $N = 2^{\nu_1+\nu_2+\nu_3}$ where $\nu_1, \nu_2 \in \mathbb{N}$ and $\nu_3 := \delta_1 - \delta_2$ be (a bound on) the number of records in the database. The choices of the initial dimension ν_1 and the folding dimension ν_2 can be arbitrary (we refer to Section 4 for our parameter-selection methodology). We index the database records by a triple (α, β, γ) where $\alpha \in [2^{\nu_1}]$, $\beta \in [2^{\nu_2}]$, and $\gamma \in [2^{\nu_3}]$.
- **Query packing:** Let $(\mathsf{QueryPackSetup}, \mathsf{QueryPack}, \mathsf{QueryUnpack})$ be the query packing algorithms from Box 2 instantiated using Construction B.6 with $\chi_{1,e}$ as the error distribution.
- **Response compression:** Let $(\mathsf{Compress}, \mathsf{CompressRecover})$ be the response compression algorithms from Box 1 instantiated using Construction C.3 with $\chi_{2,e}$ as the error distribution.

We describe how to instantiate the underlying parameters (e.g., decomposition bases, encoding modulus, etc.) for the underlying algorithms in Section 4. We now give the Respire construction:

- $\mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm proceeds as follows:
  - Sample secret keys $\tilde{s}_1 \leftarrow \chi_{1,s}$ and $\tilde{s}_2 \leftarrow \chi_{2,s}$. Define $\mathbf{s}_1 = [-\tilde{s}_1 \mid 1]^\intercal \in R^2_{d_1,q_1}$ and $\mathbf{s}_2 = [-\tilde{s}_2 \mid 1]^\intercal \in R^2_{d_2,q_2}$.
  - Sample parameters $\mathsf{pp}_{\mathsf{qpk}} \leftarrow \mathsf{QueryPackSetup}(1^\lambda, \mathbf{s}_1)$ and $\mathsf{pp}_{\mathsf{comp}} \leftarrow \mathsf{CompressSetup}(1^\lambda, \mathbf{s}_1, \mathbf{s}_2)$.

[1] Technically, an RLWE encoding (see Section 2) consists of two elements of $R_{d,q}$, but one of them is random and can be derived by applying a pseudorandom generator (PRG) to a short seed (and appealing to the random oracle heuristic).

Output the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{s}_2)$ and the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{comp}})$.

- $\mathsf{SetupDB}\big(1^\lambda, \{r_{\alpha,\beta,\gamma}\}_{\alpha \in [2^{\nu_1}], \beta \in [2^{\nu_2}], \gamma \in [2^{\nu_3}]}\big)$: On input the security parameter λ and a collection of N records $r_{\alpha,\beta,\gamma} \in R_{d_2,p}$, the setup algorithm computes for each $\alpha \in [2^{\nu_1}]$ and $\beta \in [2^{\nu_2}]$ the packed record
$$\tilde{r}_{\alpha,\beta} = \Pi\big(r_{\alpha,\beta,1}, \ldots, r_{\alpha,\beta,2^{\nu_3}}\big) \in R_{d_1,p}, \tag{3.4}$$
where $\Pi \colon R_{d_2,p}^{2^{\nu_3}} \to R_{d_1,p}$ is the packing function from Eq. (3.3) (see also Fig. 3a). Output $\mathsf{db} = \big\{\tilde{r}_{\alpha,\beta}\big\}_{\alpha \in [2^{\nu_1}], \beta \in [2^{\nu_2}]}$.
- $\mathsf{Query}(\mathsf{qk}, \mathsf{idx})$: Given the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{s}_2)$ and the index $\mathsf{idx} = (\alpha, \beta, \gamma) \in [2^{\nu_1}] \times [2^{\nu_2}] \times [2^{\nu_3}]$, let $\hat{\alpha}_i = 1$ if $i = \alpha$ and 0 otherwise. Let $\hat{\beta}_1 \cdots \hat{\beta}_{\nu_2}$ be the binary representation of $\beta - 1$ and $\hat{\gamma}_1 \cdots \hat{\gamma}_{\nu_3}$ be the binary representation of $\gamma - 1$. It outputs the query
$$\mathsf{q} \leftarrow \mathsf{QueryPack}\big(\mathbf{s}_1, (\lfloor q_1/p \rfloor \cdot \hat{\alpha}_1, \ldots, \lfloor q_1/p \rfloor \cdot \hat{\alpha}_{2^{\nu_1}}), (\hat{\beta}_1, \ldots, \hat{\beta}_{\nu_2}, \hat{\gamma}_1, \ldots, \hat{\gamma}_{\nu_3})\big). \tag{3.5}$$
- $\mathsf{Answer}(\mathsf{pp}, \mathsf{db}, \mathsf{q})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{comp}})$, a preprocessed database $\mathsf{db} = \big\{\tilde{r}_{i,j}\big\}_{i \in [2^{\nu_1}], j \in [2^{\nu_2}]}$, and the query q, the answer algorithm proceeds as follows:
  1. **Query expansion:** Compute the expanded query
  $$\Big(\big(\mathbf{c}_1^{(1)}, \ldots, \mathbf{c}_{2^{\nu_1}}^{(1)}\big), \big(\mathbf{C}_1^{(2)}, \ldots, \mathbf{C}_{\nu_2}^{(2)}, \mathbf{C}_1^{(3)}, \ldots, \mathbf{C}_{\nu_3}^{(3)}\big)\Big) \leftarrow \mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{q}).$$
  2. **First dimension:** For each $\beta \in [2^{\nu_2}]$, compute $\hat{\mathbf{c}}_\beta^{(1)} = \sum_{\alpha \in [2^{\nu_1}]} \tilde{r}_{\alpha,\beta} \cdot \mathbf{c}_\alpha^{(1)}$.
  3. **Folding:** Let $\hat{\mathbf{c}}_{0,j}^{(2)} = \hat{\mathbf{c}}_j^{(1)}$ for each $j \in [2^{\nu_2}]$. Then, for each $r \in [\nu_2]$ and $j \in [2^{\nu_2 - r}]$, compute
  $$\hat{\mathbf{c}}_{r,j}^{(2)} = \mathsf{Select}\Big(\mathbf{C}_r^{(2)},\ \hat{\mathbf{c}}_{r-1,j}^{(2)},\ \hat{\mathbf{c}}_{r-1,j+2^{\nu_2 - r}}^{(2)}\Big),$$
  where Select is the homomorphic selection algorithm defined in Section 2.
  4. **Rotation:** Let $\hat{\mathbf{c}}_0^{(3)} = \hat{\mathbf{c}}_{\nu_2,1}^{(2)}$. Then, for each $r \in [\nu_3]$, compute
  $$\hat{\mathbf{c}}_r^{(3)} = \mathsf{Select}\Big(\mathbf{C}_r^{(3)},\ \hat{\mathbf{c}}_{r-1}^{(3)},\ x^{-2^{\nu_3 - r}} \cdot \hat{\mathbf{c}}_{r-1}^{(3)}\Big).$$
  Let $\mathbf{c}^{(\mathrm{out})} = \hat{\mathbf{c}}_{\nu_3}^{(3)}$.
  5. **Compression:** Output the (compressed) response $\mathsf{a} \leftarrow \mathsf{Compress}\big(\mathsf{pp}_{\mathsf{comp}}, \mathbf{c}^{(\mathrm{out})}\big)$.
- $\mathsf{Extract}(\mathsf{qk}, \mathsf{a})$: On input the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{s}_2)$ and the answer a, output $\lfloor \mathsf{CompressRecover}(\mathbf{s}_2, \mathsf{a}) \rceil_{q_3,p}$.

Correctness. We formally analyze the correctness error in Respire as a function of the scheme parameters in Appendix D.1. In Section 4.1, we describe how we choose the scheme parameters concretely (to achieve a target error rate of 2^{-40}).

Security. In the Respire protocol (Construction 3.2), the server's view consists of the public parameters along with the client's query. The public parameters consist of RLWE encodings together with key-switching matrices for the underlying transformations, and the query consists of additional RLWE encodings. Assuming the RLWE assumption (Definition 2.3) along with a "circular security" assumption (for the key-switching parameters), we can show that the public parameters and the client's queries are pseudorandom. The circular security assumption we use is similar to those used in many previous PIR protocols [ACLS18, AYA+21, MCR21, MW22a, MR23, LMRS24, MW24]. We give a precise statement of the assumption and a formal security proof in Appendix D.2.

3.2 Extending Respire to the Batch Setting

For security, the dimension d_2 of the smaller ring in Respire cannot be arbitrarily small. Essentially, this means that the client must always download at least d_2 integers, irrespective of the size of a single record. In particular, when the record can be described by d_3 elements of $\mathbb{Z}_p$ where $d_3 \ll d_2$, we again incur a high communication penalty. The problem essentially boils down to the fact that RLWE encodings of short plaintexts have very poor rate. One approach to achieve better rate is in the batch setting where instead of fetching a single record, the client instead fetches k elements (see Remark 2.2). In this section, we describe how to extend Respire (Construction 3.2) to more efficiently support batch queries with low communication overhead. Our packing approach relies on two key ingredients which we describe below: (1) homomorphic repacking and (2) vectorization.

Database configuration. In the multi-query version of Respire, we model each database record as an element of $R_{d_3,p}$ where $d_3 \leq d_2$ is a power-of-two. In the single-query protocol, the record dimension coincides with the small ring dimension d_2, and moreover, we need to choose d_2 to be sufficiently large that the RLWE assumption holds. This in turn limits the range of possible values for d_2, and by extension, the dimension of each plaintext record. In the batched setting, we support any power-of-two $d_3 \leq d_2$. As in Respire, each packed database element contains multiple (i.e., d_1/d_3, where d_1 is the large ring dimension) individual records (see Fig. 3a).

Coefficient projections over R_d. Our homomorphic repacking procedure (described below) relies on way to project away coefficients of a polynomial $f \in R_d$. Specifically, for an integer $j \in [0, \delta]$ where $d = 2^\delta$, we define the coefficient projection map $\pi_j : R_d \to R_d$ to be the mapping

$$\pi_j \left(\sum_{i \in [0, d-1]} f_i x^i \right) := \sum_{i \in [0,d-1] : 2^j \mid i} f_i x^i. \tag{3.6}$$

In words, the projection π_j zeroes out every coefficient f_i of the input polynomial associated with a monomial x^i where i is *not* a multiple of 2^j. For instance, π_1 outputs the polynomial that only contains the even powers of x. We illustrate the coefficient projection operation in Fig. 4a.

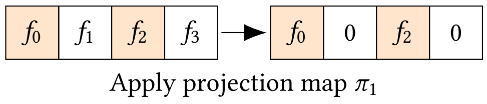
If we work over $R_{d,q}$ for odd q, the projection maps $\pi_0, \ldots, \pi_\delta$ can be efficiently implemented using the Frobenius automorphisms on R_d (i.e., the mapping $x \mapsto x^\ell$ with $\ell \in \mathbb{Z}$). Using techniques to homomorphically evaluate automorphisms on RLWE encodings [BGV12, GHS12a], we obtain an algorithm to homomorphically evaluate the coefficient projection map on RLWE encodings. We model the coefficient projection procedure as follows:

Box 3: Coefficient Projection Algorithms

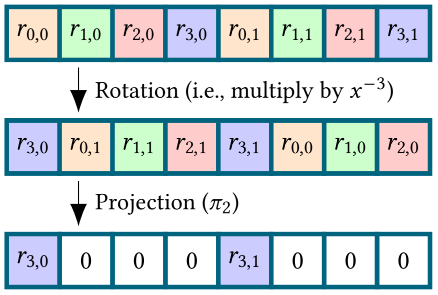
- $\mathsf{ProjectSetup}(1^\lambda, \mathbf{s}) \to \mathsf{pp}$: On input the security parameter λ and an RLWE secret key $\mathbf{s} \in R_{d,q}^2$, the projection setup algorithm outputs a set of projection public parameters $\mathsf{pp}_{\mathsf{proj}}$.
- $\mathsf{Project}(\mathsf{pp}_{\mathsf{proj}}, \mathbf{c}, j) \to \mathbf{c}'$: On input the projection parameters $\mathsf{pp}_{\mathsf{proj}}$, an RLWE encoding $\mathbf{c} \in R_{d,q}^2$, and the projection index $j \in [\delta]$, the projection algorithm outputs a new RLWE encoding $\mathbf{c}'$.

The property we require is that if $\mathbf{c}$ is an RLWE encoding of a polynomial $f \in R_d$ with respect to $\mathbf{s}$, then $\mathsf{Project}(\mathsf{pp}, \mathbf{c})$ outputs an RLWE encoding of the projected polynomial $\pi_j(f)$. We provide the formal description in Appendix A.1.

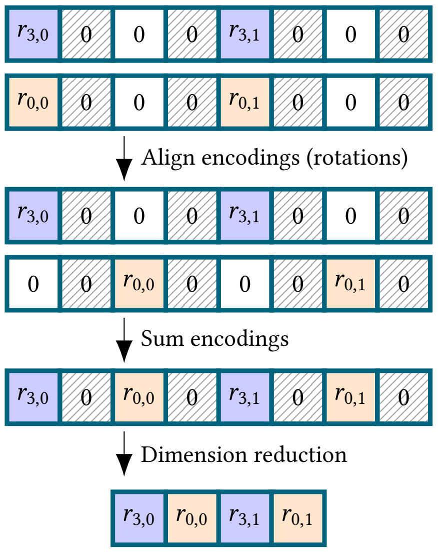
Homomorphic repacking. The main ingredient in the batched construction of Respire is a repacking algorithm. At a high level, the (homomorphic) repacking algorithm takes a batch of $k = d_3/d_2$ responses and compresses them into a *single* response (i.e., a single RLWE encoding). More formally, suppose the client makes a batch of k queries for records $r_1, \ldots, r_k \in R_{d_3,p}$. Let $s_1, \ldots, s_k \in R_{d_1,p}$ be the packed database elements that contain the records $r_1, \ldots, r_k$. The repacking algorithm takes $s_1, \ldots, s_k$ and outputs s' that encodes $\Pi(r_1, \ldots, r_k)$ where $\Pi : R_{d_3,p}^k \to R_{d_2,p}$ is the ring packing function from Eq. (3.3). Essentially, the repacking procedure first extracts the desired record r_i from its



(a) Illustration of the projection operation over the polynomial ring $R_4 = \mathbb{Z}[x]/(x^4+1)$. The projection map π_1 zeroes out all coefficients associated with *odd* powers of x. We model polynomials $f(x) = \sum_{i \in [0,3]} f_i x^i$ as a vector of coefficients $(f_0, f_1, f_2, f_3) \in \mathbb{Z}^4$.


$r_{0,0}$ $r_{1,0}$ $r_{2,0}$ $r_{3,0}$ $r_{0,1}$ $r_{1,1}$ $r_{2,1}$ $r_{3,1}$
Rotation (i.e., multiply by x^{-3})
$r_{3,0}$ $r_{0,1}$ $r_{1,1}$ $r_{2,1}$ $r_{3,1}$ $r_{0,0}$ $r_{1,0}$ $r_{2,0}$
Projection (π_2)
$r_{3,0}$ 0 0 0 $r_{3,1}$ 0 0 0

(b) The rotation step aligns the desired record (r_3) into the initial position, just as in the single-query case. The subsequent projection step zeroes out all components other than the desired one. For simplicity, we ignore the sign changes from the rotation.


$r_{3,0}$ 0 0 0 $r_{3,1}$ 0 0 0
$r_{0,0}$ 0 0 0 $r_{0,1}$ 0 0 0
Align encodings (rotations)
$r_{3,0}$ 0 0 0 $r_{3,1}$ 0 0 0
0 0 $r_{0,0}$ 0 0 0 $r_{0,1}$ 0
Sum encodings
$r_{3,0}$ 0 $r_{0,0}$ 0 $r_{3,1}$ 0 $r_{0,1}$ 0
Dimension reduction
$r_{3,0}$ $r_{0,0}$ $r_{3,1}$ $r_{0,1}$

(c) After applying the rotation and projection mappings to each response, the packing algorithm realigns and sums them together to obtain the final encoding. Finally, we apply the same dimension reduction algorithm (as in vanilla Respire) to obtain the final repacked encoding. Components that are lost after dimension reduction are indicated by the striped pattern.

Figure 4: Illustration of the core operations underlying the Respire (homomorphic) repacking algorithm. In Figs. 4b and 4c, the main ring dimension is $d_1 = 8$, the reduced ring dimension is $d_2 = 4$, and the record dimension is $d_3 = 2$. We pack $k = 2$ records in each output encoding. Each record r_i is a pair $(r_{i,0}, r_{i,1})$.

packed representation s_i and then repacks the extracted records into a single element of $R_{d_1,p}$. We describe the main steps of our approach below and illustrate the key steps in Figs. 4b and 4c.

- **Rotation:** Let $\ell = d_1/d_3$ and let $j_i \in [0, \ell-1]$ be the position of r_i within s_i. The repacking algorithm first computes $x^{-j_i} \cdot s_i$. By construction, r_i is in the initial position within $x^{-j_i} \cdot s_i$.
- **Projection:** Next, the repacking algorithm projects away all records other than the initial record using the projection map $\pi_{\delta_1 - \delta_3}$ where $d_3 = 2^{\delta_3}$ and $d_1 = 2^{\delta_1}$. Namely, the repacking algorithm computes $t_i = \pi_{\delta_1-\delta_3}(x^{-j_i} \cdot s_i)$, where $\pi_{\delta_1-\delta_3}$ is the projection function from Eq. (3.6). This yields a packed encoding with r_i in the initial position and 0 in all other positions.
- **Repacking:** Given $t_1, \ldots, t_k \in R_{d_1,p}$, the algorithm now aggregates the packed encoding by computing $t = \sum_{i \in [0,k-1]} t_i \cdot x^{i \cdot d_1/d_2}$. This is shown in Fig. 4c.

Finally, we observe that each of the underlying operations (rotation, projection, and repacking) can be described in terms of scalar multiplications, additions, and automorphisms; thus, we can implement these homomorphically on RLWE encodings. This yields the homomorphic repacking approach in the batched version of Respire.

Vectorization. With repacking, a single Respire response can encode $k = d_2/d_3$ queries. When the client makes more than k queries, then the response necessarily contains more than a single RLWE encoding. In this setting, we can leverage the response packing approach from Spiral [MW22a] and pack the individual RLWE encodings into a single *vector* encoding. In this way, each of the RLWE encodings in the response share a *common* "random" component. Moreover, with split modulus switching, the modulus associated with the "random" component is much larger than those of the message-embedding component. Concretely, our use of vectorization reduces the response size by a factor of $\approx 2.7\times$ when the batch size is 32.

We start by introducing the notion of a vector RLWE encoding. We say that

$$\mathbf{c} = \begin{bmatrix} a \\ \mathbf{s}a + \mathbf{e} + \boldsymbol{\mu} \end{bmatrix} \in R_{d,q}^{n+1}$$

is an RLWE encoding of a vector $\boldsymbol{\mu} \in R_{d,q}^{n}$ with respect to a secret key $\mathbf{S} = [-\mathbf{s} \mid \mathbf{I}_n]^{\intercal} \in R_{d,q}^{(n+1)\times n}$ and error $\mathbf{e} \in R_{d,q}^{n}$ if $\mathbf{S}^{\intercal}\mathbf{c} = \boldsymbol{\mu}+\mathbf{e}$. Similar to a (scalar) RLWE encoding, when $\mathbf{c} = \left[\begin{smallmatrix} c_1 \\ \mathbf{c}_2 \end{smallmatrix}\right]$, we often refer to c_1 as the "random" component of the encoding and $\mathbf{c}_2$ as the "message-embedding" component of the encoding. Notably, the compression from using vector RLWE encodings comes from the fact that random component is only a *single* ring element. In contrast, n scalar RLWE encodings would include n random components, one for each encoding. We now recall the syntax from [MW22a]:

Box 4: Vectorization Algorithms

- $\mathsf{VecSetup}(1^{\lambda}, \mathbf{s}_1, \mathbf{S}_2) \to \mathsf{pp}_{\mathsf{vec}}$: On input a security parameter λ and two secret keys $\mathbf{s}_1 \in R_{d,q}^{2}$ and $\mathbf{S}_2 \in R_{d,q}^{(n+1)\times n}$, the setup algorithm outputs a set of vectorizing parameters $\mathsf{pp}_{\mathsf{vec}}$.
- $\mathsf{Vectorize}(\mathsf{pp}_{\mathsf{vec}}, (\mathbf{c}_1, \ldots, \mathbf{c}_n)) \to \mathbf{c}'$: On input the vectorization parameters $\mathsf{pp}_{\mathsf{vec}}$ and a tuple of encodings $\mathbf{c}_1, \ldots, \mathbf{c}_n \in R_{d,q}^{2}$, the vectorization algorithm outputs a ciphertext $\mathbf{c}' \in R_{d,q}^{n+1}$.

If $\mathbf{c}_1, \ldots, \mathbf{c}_n$ are RLWE encodings of the scalars $\mu_1, \ldots, \mu_n \in R_{d,q}$ with respect to $\mathbf{s}_1$, then $\mathsf{Vectorize}(\mathsf{pp}_{\mathsf{vec}}, (\mathbf{c}_1, \ldots, \mathbf{c}_n))$ outputs an encoding $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_n)$ with respect to $\mathbf{s}_2$ (and slightly larger noise). We refer to Appendix C.1 for the formal description and correctness analysis.

Batching queries in Respire. We now give the formal description of Respire tailored for the batch setting.

Construction 3.3 (Respire for Batch Queries). Let λ be a security parameter. The batched version of Respire is parameterized by a similar set of components as the base version of Respire (Construction 3.2). We enumerate these below:

- **Lattice parameters:** As in Respire, let $d_1 = d_1(\lambda)$ and $d_2 = d_2(\lambda)$ denote the full ring dimension and the reduced ring dimension, respectively. We require that $d_1 = 2^{\delta_1}$ and $d_2 = 2^{\delta_2}$ where δ_1, δ_2 are non-negative integers and $d_1 \geq d_2$. Let $q_1 = q_1(\lambda)$, $q_2 = q_2(\lambda)$, and $q_3 = q_3(\lambda)$ be moduli where $q_1 \geq q_2 \geq q_3$. Let $\chi_{1,e} = \chi_{1,e}(\lambda)$, $\chi_{1,s} = \chi_{1,s}(\lambda)$, $\chi'_{1,e} = \chi'_{1,e}(\lambda)$, and $\chi'_{1,s} = \chi'_{1,s}(\lambda)$ be distributions over R_{d_1,q_1}. Let $\chi_{2,e} = \chi_{2,e}(\lambda)$, $\chi_{2,s} = \chi_{2,s}(\lambda)$ be distributions over R_{d_2,q_2}.
- **Plaintext dimension modulus:** Let $d_3 = d_3(\lambda)$ be the record dimension and p be the plaintext modulus. Each database record is an element of $R_{d_3,p} \coloneqq \mathbb{Z}_p[x]/(x^{d_3}+1)$. We require that $d_3 = 2^{\delta_3}$ where δ_3 is a non-negative integer and $d_2 \geq d_3$.
- **Database configuration:** Let $N = 2^{\nu_1+\nu_2+\nu_3}$ where $\nu_1, \nu_2 \in \mathbb{N}$ and $\nu_3 \coloneqq \delta_1 - \delta_3$ be (a bound on) the number of records in the database. The choices of the initial dimension ν_1 and the folding dimension ν_2 can be arbitrary.
- **Query packing parameters:** Let ($\mathsf{QueryPackSetup}, \mathsf{QueryPack}, \mathsf{QueryUnpack}$) be the query packing algorithms from Box 2 instantiated using Construction B.6 with $\chi_{1,e}$ as the error distribution.
- **Projection parameters:** Let ($\mathsf{ProjectSetup}, \mathsf{Project}$) be the homomorphic projection algorithms from Box 3 instantiated using Construction A.7 with $\chi_{1,e}$ as the error distribution.
- **Vectorization parameters:** Let ($\mathsf{VecSetup}, \mathsf{Vectorize}$) be the vectorization algorithms from Box 4 instantiated using Construction C.1) with $\chi'_{1,e}$ as the error distribution. Let $n_{\mathsf{vec}} = n_{\mathsf{vec}}(\lambda)$ be the vector length used for vectorization.
- **Response compression parameters:** Let ($\mathsf{Compress}, \mathsf{CompressRecover}$) be the response compression algorithms from Box 1 instantiated using Construction C.3 with $\chi_{2,e}$ as the error distribution.

We describe how to instantiate the underlying parameters (e.g., decomposition bases, encoding modulus) for the underlying algorithms in Section 4. We now describe a scheme that supports a maximum batch size of $T = n_{\mathsf{vec}}(d_2/d_3)$.

- $\mathsf{Setup}(1^\lambda)$: On input the security parameter λ, the setup algorithm proceeds as follows:
  - Sample a source key $\tilde{s}_1 \leftarrow \chi_{1,s}$ and two target keys $\tilde{\mathbf{s}}_1' \leftarrow (\chi_{1,s}')^{n_{\mathsf{vec}}}$ and $\tilde{\mathbf{s}}_2 \leftarrow \chi_{2,s}^{n_{\mathsf{vec}}}$. Define
  $$\mathbf{s}_1 = [-\tilde{s}_1 \mid 1]^{\mathsf{T}} \in R_{d_1,q_1}^2 \quad \text{and} \quad \mathbf{S}_1' = [-\tilde{\mathbf{s}}_1' \mid \mathbf{I}_{n_{\mathsf{vec}}}]^{\mathsf{T}} \in R_{d_1,q_1}^{(n_{\mathsf{vec}}+1)\times n_{\mathsf{vec}}} \quad \text{and} \quad \mathbf{S}_2 = [-\tilde{\mathbf{s}}_2 \mid \mathbf{I}_{n_{\mathsf{vec}}}]^{\mathsf{T}} \in R_{d_2,q_2}^{(n_{\mathsf{vec}}+1)\times n_{\mathsf{vec}}}.$$
  - Next, sample parameters for query packing, projection, vectorization, and response packing:
    * $\mathsf{pp}_{\mathsf{qpk}} \leftarrow \mathsf{QueryPackSetup}(1^\lambda, \mathbf{s}_1)$.
    * $\mathsf{pp}_{\mathsf{proj}} \leftarrow \mathsf{ProjectSetup}(1^\lambda, \mathbf{s}_1)$.
    * $\mathsf{pp}_{\mathsf{vec}} \leftarrow \mathsf{VecSetup}(1^\lambda, \mathbf{s}_1, \mathbf{S}_1')$.
    * $\mathsf{pp}_{\mathsf{comp}} \leftarrow \mathsf{CompressSetup}(1^\lambda, \mathbf{S}_1', \mathbf{S}_2)$.

  The setup algorithm outputs the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{S}_2)$ and the parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{proj}}, \mathsf{pp}_{\mathsf{vec}}, \mathsf{pp}_{\mathsf{comp}})$.

- $\mathsf{SetupDB}\big(1^\lambda, \{r_{\alpha,\beta,\gamma}\}_{\alpha\in[2^{\nu_1}],\beta\in[2^{\nu_2}],\gamma\in[2^{\nu_3}]}\big)$: On input the security parameter λ and a collection of N records $r_{\alpha,\beta,\gamma} \in R_{d_3,p}$, the database preprocessing algorithm constructs the packed records as in Respire (Construction 3.2). Namely, for all $\alpha \in [2^{\nu_1}]$ and $\beta \in [2^{\nu_2}]$, it computes $\tilde{r}_{\alpha,\beta}$ according to Eq. (3.4), except the ring packing function Π now maps $R_{d_3,p}^{2^{\nu_3}}$ to $R_{d_1,p}$. The algorithm then outputs the packed elements $\mathsf{db} = \big\{\tilde{r}_{\alpha,\beta}\big\}_{\alpha\in[2^{\nu_1}],\beta\in[2^{\nu_2}]}$.

- $\mathsf{Query}(\mathsf{qk}, (\mathsf{idx}_1, \ldots, \mathsf{idx}_T))$: On input the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{S}_2)$ and a tuple of T queries $\mathsf{idx}_1, \ldots, \mathsf{idx}_T$ where $\mathsf{idx}_t = (\alpha^{(t)}, \beta^{(t)}, \gamma^{(t)}) \in [2^{\nu_1}] \times [2^{\nu_2}] \times [2^{\nu_3}]$, the query algorithm computes q_t according to Eq. (3.5) for each $t \in [T]$. It outputs the query $\mathsf{q} = (\mathsf{q}_1, \ldots, \mathsf{q}_T)$.

- $\mathsf{Answer}(\mathsf{pp}, \mathsf{db}, \mathsf{q})$: On input the parameters $\mathsf{pp} = (\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{proj}}, \mathsf{pp}_{\mathsf{vec}}, \mathsf{pp}_{\mathsf{comp}})$, a preprocessed database $\mathsf{db} = \big\{\tilde{r}_{i,j}\big\}_{i\in[2^{\nu_1}],j\in[2^{\nu_2}]}$, and a query $\mathsf{q} = (\mathsf{q}_1, \ldots, \mathsf{q}_T)$, the answer algorithm proceeds as follows:
  1. **Run Respire for each query:** For each $t \in [T]$, run Steps 1 to 4 of the Answer algorithm in Respire (Construction 3.2) using the query expansion parameters $\mathsf{pp}_{\mathsf{qpk}}$, the preprocessed database db, and the query q_t. Let $\mathbf{c}_t^{(\mathsf{out})}$ be the output of Step 4 of the Answer algorithm on the tth query.
  2. **Projection:** For each $t \in [T]$, homomorphically project each response:
  $$\mathbf{c}_t^{(\mathsf{proj})} \leftarrow \mathsf{Project}\big(\mathsf{pp}_{\mathsf{proj}}, \mathbf{c}_t^{(\mathsf{out})}, \delta_1 - \delta_3\big).$$
  3. **Repacking:** For each $j \in [n_{\mathsf{vec}}]$, compute the repacked encoding
  $$\mathbf{c}_j^{(\mathsf{repack})} = \sum_{i\in[d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \mathbf{c}^{(\mathsf{proj})}_{(d_2/d_3)\cdot(j-1)+i}.$$
  4. **Vectorizing:** Next, the answer algorithm packs the encodings into a single vector RLWE encoding:
  $$\mathbf{c}^{(\mathsf{vec})} \leftarrow \mathsf{Vectorize}\left(\mathsf{pp}_{\mathsf{vec}}, \left(\mathbf{c}_1^{(\mathsf{repack})}, \ldots, \mathbf{c}_{n_{\mathsf{vec}}}^{(\mathsf{repack})}\right)\right).$$
  5. **Compression:** Output the (compressed) response $\mathsf{a} \leftarrow \mathsf{Compress}\big(\mathsf{pp}_{\mathsf{comp}}, \mathbf{c}^{(\mathsf{vec})}\big)$.

- $\mathsf{Extract}(\mathsf{qk}, \mathsf{a})$: On input the query key $\mathsf{qk} = (\mathbf{s}_1, \mathbf{S}_2)$ and the response a, compute the packed responses
$$\begin{bmatrix} \hat{r}_1 \\ \vdots \\ \hat{r}_{n_{\mathsf{vec}}} \end{bmatrix} \leftarrow \lfloor \mathsf{CompressRecover}(\mathbf{S}_2, \mathsf{a}) \rceil_{q_3,p} \in R_{d_2,p}^{n_{\mathsf{vec}}}$$

  For each $i \in [d_2/d_3]$ and $j \in [n_{\mathsf{vec}}]$, set $r_{(d_2/d_3)\cdot(j-1)+i} = \kappa_{d_3,d_2}^{-1}(x^{-(i-1)} \cdot \hat{r}_j) \in R_{d_3,p}$, where $\kappa_{d_3,d_2}^{-1} : R_{d_2} \to R_{d_3}$ is the dimension reduction mapping from Eq. (3.2). Finally, output the records $r_1, \ldots, r_T$.

Remark 3.4 (Packing Responses from Different Databases)**.** The first step of the Answer algorithm in Construction 3.3 runs T *independent* executions of the Respire protocol to obtain T responses $\mathbf{c}_1^{(\mathrm{out})}, \ldots, \mathbf{c}_T^{(\mathrm{out})}$ which are then packed together. In Construction 3.3, each of these queries were applied to the *same* preprocessed database db. However, this does *not* have to be the case. In particular, each query q_i could be applied over a *different* preprocessed database db_i of the same dimension. The rest of the packing algorithm is agnostic to this choice. This allows us to compose this approach with (probabilistic) batch codes [IKOS04, ACLS18] to reduce the *computational* costs of answering T queries. Instead of needing to make a pass over the full database to answer each query in the batch, the server in this case applies query q_i to a much smaller sub-database db_i. We use this approach to obtain a batch PIR scheme. We refer to Section 4.3 for implementation details and benchmarks.

Remark 3.5 (Comparison with Vectorized BatchPIR)**.** The response packing approach described here may seem similar to other batch PIR schemes such as Vectorized BatchPIR [MR23] and Piranha [LLWR24]. However, there is a critical difference: both Vectorized BatchPIR and Piranha leverage SIMD support in FHE [GHS12a] to support batch queries, and specifically, they use the Brakerski-Fan-Vercauteren (BFV) encryption scheme [Bra12, FV12]. In the BFV scheme, the noise grows exponentially with the multiplicative depth of the computation, leading to larger parameters. Moreover, SIMD packing is not compatible with the query compression techniques from [ACLS18, CCR19], which leads to larger queries and responses. The approach taken in Respire is to first build a communication-efficient single-query PIR scheme (Construction 3.2) that leverages the RLWE-GSW external product [CGGI18, CGGI20] to implement homomorphic multiplication (following [GH19, MCR21, MW22a]). This allows better noise growth (scaling *linearly* with the multiplicative depth) and allows us to leverage techniques for query and response compression. On the flip side, Respire does not support SIMD operations, so the server cost is higher with Respire for *large* batch sizes. For small batch sizes (e.g., issuing 32 queries on a 1 GB database), Respire is 16% faster than Vectorized BatchPIR and requires 4.9× less communication. We refer to Section 4.3 for the full breakdown.

4 Implementation and Evaluation

The Respire protocol is designed for databases with small records. In our evaluation, we focus on the setting where each database record is 256 bytes; this is a typical setting used in applications of PIR to metadata-hiding communication [AS16, ALP+21, AYA+21].

4.1 Parameter Selection

In our evaluation, we use two different sets of parameters. The first set is tailored for the single-query case while the second has better support for batch queries. For our evaluation, we view the single-query Respire (Construction 3.2) as a special case of the batch version of Respire (Construction 3.3), where we set the vectorization length to $n_{\mathrm{vec}} = 1$ and the record dimension d_3 to the reduced ring dimension d_2 (i.e., $d_2 = d_3$). In this case, the batched version of Respire essentially corresponds to the single-query version described in Construction 3.2.

Parameter selection methodology. We choose the scheme parameters to tolerate a correctness error of at most 2^{-40} (based on the formal analysis given in Appendix D.1). Simultaneously, we choose the lattice parameters to ensure that *each* of the underlying RLWE assumptions which we require for security (see Appendix D.2) has 128 bits of classical security. We use the lattice estimator tool [APS15a] for our security estimates.[2] Here, we describe how we select the primary parameters of our scheme and list our parameter choices in Appendix E.

Lattice parameters. In Respire, we rely on three different RLWE assumptions:

- RLWE over the main ring R_{d_1,q_1} (with error distribution $\chi_{1,e}$ and secret key distribution $\chi_{1,s}$). The queries (and many of the key-switching matrices) are encoded over the large ring. We set the ring dimension to be $d_1 = 2048$ and q_1 to be a 56-bit modulus (where $q_1 = 1 \bmod 2d$ to support fast NTT evaluation over R_{d_1,q_1}). Since the noise in the initial GSW encodings (output by the query expansion procedure) scales with the *norm* of the

[2]We used commit `7ea215a4d55f` (April 8, 2024) from [APS15b] for our security estimates.

secret key (as opposed to its variance), we take $\chi_{1,s}$ to be the uniform distribution on the interval $[-7, 7]$. The error distribution $\chi_{1,e}$ is a discrete Gaussian distribution with width parameter $\sigma_{1,e} = 9.9$. We note that using a norm-bounded secret key distribution is common in lattice-based cryptographic systems, and for instance, is used both in standardized lattice-based key-agreement protocols [ABD+21] (specifically, the Kyber protocol uses a binomial distribution on the interval $[-3, 3]$) or FHE schemes (many schemes use a ternary secret key distribution [GHS12b, CKKS17, ACC+18]).

- RLWE over the main ring R_{d_1,q_1} (with error distribution $\chi'_{1,e}$ and secret key distribution $\chi'_{1,s}$). We consider a secondary instantiation of RLWE over the main ring for sampling the vectorization parameters where the secret key and the error are both sampled from a discrete Gaussian distribution with width $\sigma'_1 = 9.9$. Compared to the previous instantiation, we substitute a discrete Gaussian distribution in place of the uniform distribution since the former has a smaller subgaussian width parameter. This allows better control of noise growth in the vectorization step (see Appendix D.1).
- RLWE over the small ring R_{d_2,q_2} (with error distribution $\chi_{2,e}$ and $\chi_{2,s}$). We rely on this assumption to publish the key-switching matrices needed for dimension reduction. In the single-query setting, we take the reduced ring dimension to be $d_2 = 512$ and sample both the secret key and the error from a discrete Gaussian distribution with width $\sigma_2 = 253.6$.

Each of these instantiations provides 128 bits of classical security according to the lattice estimator tool [APS15a].

Database configuration. We choose the dimension of the reduced ring to be $d_2 = 512$. This is the smallest (power-of-two) ring dimension that we could find which provides 128-bits of security and a correctness error of 2^{-40} for the database configurations of interest. We set $p = 16$ so each plaintext element (in $R_{d_2,p}$) can encode 256 bytes of data. Since $d_1 = 2048$, we can pack $v_3 = d_1/d_2 = 4$ records into each ring element. We choose the remaining database dimensions v_1 and v_2 to be roughly equal; this achieves a good balance between the noise growth and the computational costs of the protocol.

Gadget decomposition parameters. The different sub-algorithms in Respire (query packing, projection, vectorization, and compression) are parameterized by different gadget decomposition bases z. Smaller decomposition bases (corresponding to a wider gadget matrix) reduce the noise growth but incurs more computational costs and larger public parameters. In many settings, the noise growth from a sub-algorithm introduces an *additive* increase in the noise rather than a multiplicative factor. As such, we opt to pick the largest gadget decomposition base that does not significantly increase the noise accumulation. This leads to smaller public parameters (and computational overhead). We enumerate the decomposition bases we use in Table 5 in Appendix E.

Modulus choice. Similar to Spiral [MW22a], we choose the main encoding modulus q_1 to be a product of two 28-bit primes: $q_1 = q_{1,1} \cdot q_{1,2}$, where $q_{1,1}, q_{1,2} = 1 \bmod 2d_1$. This allows us to use the (negacyclic) NTT for fast ring multiplication [LMPR08, LN16], which we accelerate using the AVX2 SIMD instructions (c.f., [BKS+21]). We implement arithmetic modulo q_1 using 64-bit native integer arithmetic modulo $q_{1,1}$ and $q_{1,2}$ (with deferred modular reductions), and combine the results using the Chinese remainder theorem. Similarly, we choose $q_2 = 1 \bmod 2d_2$ so we can also use NTTs for polynomial arithmetic over the ring R_{d_2,q_2}. Finally, we choose q_2 and q_3 to be the smallest values possible while still ensuring correctness. Concretely, for the single-query scheme, $q_2 \approx 2^{24}$ and $q_3 = 2^4$.

PRG compression. We use a standard optimization [ALP+21, MCR21, HHC+23, MW22a, MR23, LMRS24, MW24] to reduce the query size, wherein the client sends a PRG seed in place of the random component of the RLWE encodings in the query. We instantiate the PRG using ChaCha20 [Ber08].

Sharing public parameters. Several of the public parameters in Respire (Constructions 3.2 and 3.3) rely on a set of automorphism key-switching matrices (Construction A.4). These include the public parameters $\mathsf{pp}_{\mathsf{coeff,RLWE}}$, $\mathsf{pp}_{\mathsf{coeff,GSW}}$ used for query expansion as well as the public parameters $\mathsf{pp}_{\mathsf{proj}}$ used for projections. When choosing parameters, we use the *same* decomposition base for the GSW query expansion and the projection step; this allows us to use the same set of key-switching matrices for both steps (see Table 5 in Appendix E for the full breakdown). This reduces the size of the public parameters. As noted in Remark B.8, we use different decomposition bases in $\mathsf{pp}_{\mathsf{coeff},\mathsf{RLWE}}$ and $\mathsf{pp}_{\mathsf{coeff},\mathsf{GSW}}$ to balance the noise in the resulting RLWE and GSW encodings.

4.2 Respire Benchmarks and Evaluation

Our implementation of Respire contains roughly 8,000 lines of Rust.[3] We use an AWS EC2 `r7i.8xlarge` instance with 32 vCPUs (Intel Xeon Platinum 8488C @ 2.4GHz), 256 GB of memory, and running Ubuntu 22.04.4 for our experiments. We use `rustc 1.77.0` as our Rust compiler and `gcc 11.4.0` as our C++ compiler. The processor supports the AVX2 and AVX-512 instruction sets and we enable SIMD instruction set support for all schemes. Our implementation of Respire only uses AVX2, and not AVX-512. We use a single-threaded execution environment for all measurements. All measured running times were averaged over at least 5 trials and have a standard deviation of at most 1% of the average value. Throughout, we write KB, MB, GB to denote 2^{10}, 2^{20}, and 2^{30} bytes, respectively.

Comparison schemes. Among the single-query PIR protocols, Respire is most similar to Spiral [MW22a]. Both protocols operate in the model with client-specific public parameters. The Spiral family of protocols represents the current state-of-the-art in this setting. In our evaluation, we benchmark against the reference implementation of Spiral [MW22b], which selects the different Spiral variants (e.g., Spiral, SpiralPack, SpiralStream) depending on the database configuration. For the database configurations we consider (databases with small records), the implementation defaults to SpiralPack. For our evaluation, we focus on databases with small records (e.g., 256-byte records). For databases with larger records, the query compression and response compression techniques in Respire are no longer applicable, and the performance of Respire essentially converges to that of Spiral (or SpiralPack).

To illustrate the new computation/communication trade-offs achieved by Respire, we also report benchmarks against the state-of-the art protocols in other models. This includes the SimplePIR protocol, which operates in a different model where the client first downloads a database-dependent hint in the offline phase. SimplePIR achieves extremely high throughput at the expense of needing a large hint (and larger query/response sizes).[4] Finally, we also compare against HintlessPIR [LMRS24] and YPIR [MW24]. These schemes achieve *silent* preprocessing where there is no client-side or server-side state, but have higher communication costs. We refer to Section 5 for further discussion of other PIR constructions. Finally, for the batch setting, we compare against Vectorized BatchPIR [MR23]. For each of these schemes, we measure their performance using their reference implementations on our benchmarking setup.

Macrobenchmarks. Table 1 compares the performance of Respire to other PIR protocols. On a 256 MB database, a Respire query is just 4.1 KB and the response is 2 KB. This is a 3.9× reduction in query size and 10× reduction in response size compared to Spiral. Compared to protocols like SimplePIR, HintlessPIR, and YPIR, the Respire scheme achieves over a 20× reduction in total communication. Over an 8 GB database, the online communication in Respire is 4.5× smaller than Spiral, and over 40× smaller than SimplePIR. The reduction in query size and response size in Respire is due to the query compression and response compression techniques described in Section 3 (see also Appendices B and C for the formal description of our algorithms). Notably, in Respire, we avoid having to communicate a complete RLWE encoding over the large ring R_{d_1,q_1}, and indeed the *total* online communication in Respire is smaller than the size of a single element of R_{d_1,q_1}. Previous RLWE-based PIR schemes (e.g., [MBFK16, ACLS18, AYA+21, MCR21, MW22a]) all communicated at least one (large) RLWE encoding, which results in larger queries and responses.

In fact, the query and response size of Respire on databases with small records is comparable to those using traditional number-theoretic assumptions [ALP+21] (e.g., schemes based on ElGamal or Gentry-Ramzan [GR05]). The advantage of these traditional number-theoretic schemes has been small communication. For example, the Gentry-Ramzan scheme can have communication as low as 1.8 KB when considering a 1 MB database with 5000 records (288 bytes per record), but at the price of a server throughput of roughly 20 KB/s [ALP+21, Table 5]. By modestly increasing communication to 5.4 KB, the throughput can be increased to roughly 186 KB/s. In contrast, with Respire, we can

[3] Our implementation is available here: `https://github.com/AMACB/respire/`.

[4] There are faster schemes with sublinear server computation [ZPSZ24, MSR23, GZS24], but they require the client to stream the full database in an offline phase. For our comparisons, we focus on schemes whose total communication is sublinear in the database size.

Database	Metric	Spiral	SimplePIR	HintlessPIR	YPIR	Respire
$2^{20} \times$ 256B (256 MB)	**Offline Comm.**	7.8 MB	102.9 MB	—	—	3.9 MB
	Query Size	16.0 KB	32.0 KB	424 KB	574 KB	4.1 KB
	Response Size	20.0 KB	102.0 KB	964 KB	60 KB	2.0 KB
	Computation	1.28 s	0.024 s	0.658 s	0.17 s	1.26 s
	Throughput	200 MB/s	10.4 GB/s	389 MB/s	1.49 GB/s	204 MB/s
$2^{22} \times$ 256B (1 GB)	**Offline Comm.**	7.8 MB	211.1 MB	—	—	3.9 MB
	Query Size	16.0 KB	64.0 KB	488 KB	686 KB	7.7 KB
	Response Size	20.0 KB	211.2 KB	1.71 MB	120 KB	2.0 KB
	Computation	2.94 s	0.093 s	1.242 s	0.40 s	3.48 s
	Throughput	348 MB/s	10.8 GB/s	825 MB/s	2.50 GB/s	295 MB/s
$2^{25} \times$ 256B (8 GB)	**Offline Comm.**	10.0 MB	445.1 MB	—	—	3.9 MB
	Query Size	16.0 KB	256.0 KB	1.35 MB	1.33 MB	14.8 KB
	Response Size	60.0 KB	445.0 KB	1.71 MB	228 KB	2.0 KB
	Computation	15.44 s	0.772 s	3.698 s	1.71 s	20.84 s
	Throughput	530 MB/s	10.4 GB/s	2.16 GB/s	4.69 GB/s	393 MB/s

Table 1: Comparison of Respire to Spiral [MW22a], SimplePIR [HHC+23], HintlessPIR [LMRS24], and YPIR [MW24] for retrieving a single record from databases of various sizes. For each scheme, we report the offline communication (i.e., the public parameters in the case of Spiral and Respire and the server hint in the case of SimplePIR). We define the throughput to be the ratio of the database size to the server's computation time.

achieve comparable communication (6.1 KB), but with a throughput of several hundred MB/s (over 1000× faster than the number-theoretic constructions). Thus, Respire provides a new data point in communication-computation trade-offs.

Compared to other lattice-based PIR schemes, Respire trades off server throughput for smaller queries and responses. Compared to Spiral, Respire is about 26% slower on an 8 GB database (but requires 4.5× less communication and a 2.5× smaller public parameters). Compared to protocols like SimplePIR, HintlessPIR, and YPIR, the Respire protocol is up to 27× smaller on the 8 GB database. These protocols have substantially larger queries (over 90× larger for HintlessPIR) or hints (SimplePIR requires downloading a 445 MB hint).

Server throughput. Fig. 5 shows the query-processing in Respire as a function of the database size. We compare with Spiral, the current state-of-the-art scheme in the model with client-specific parameters. For small databases, the query processing time of Spiral and Respire are quite comparable, but the gap widens with larger database. The difference is likely due to parameter choices: our response compression approach in Respire (Construction C.3) requires using a smaller plaintext modulus compared to Spiral; as such, this increases the cost of the initial linear scan over the database (i.e., the protocol must process more RLWE encodings). Overall, we observe a 1.3× increase in processing time on an 8 GB database (but 4.5× less communication).

Microbenchmarks. Fig. 6 provides a fine-grained breakdown of the server computation time in Respire. The first dimension requires a linear scan over the database and thus, the running time scales linearly with the size of the database. The rest of the cost is split between the folding and the query expansion steps. The peculiar "zig-zag" behavior of query unpacking and folding is due to our parameter selection methodology. As described in Section 4.1, we choose parameters that balance the size of the first dimension 2^{ν_1} and the size of the second dimension 2^{ν_2}. Incrementing ν_1 doubles the number of coefficients that need to be expanded using query expansion while incrementing ν_2 double the amount of work in the folding step. Since we alternate incrementing ν_1 and ν_2, we obtain the behavior shown in Fig. 6. Finally, while the final response compression step is critical for reducing the size of the response (by a 14× factor in the single-query case), it is applied to a fixed number of RLWE encodings (independent of the database size). As such, it constitutes almost a negligible fraction of the overall server computational cost (less than 0.1% in all settings).

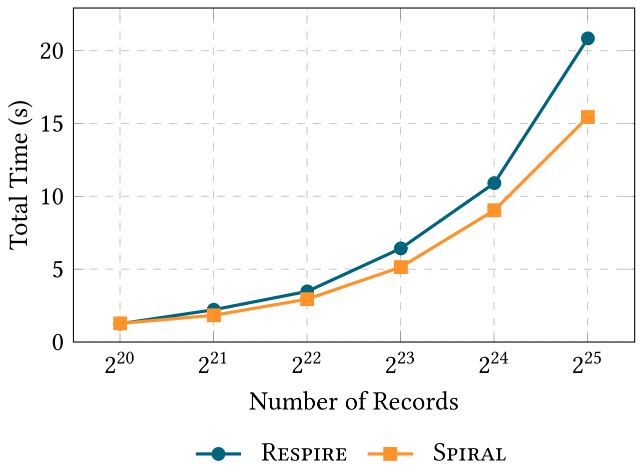


Figure 5: Total query processing time for Respire and Spiral [MW22a] as a function of the number of records in the database. Each record is 256 bytes.

Client computation. The client-side costs in Respire are minimal. In our experiments, the setup time takes a maximum of 80 ms, the query-generation time takes at most 148 ms, and response decoding takes at most 7 ms.

Server preprocessing. In Respire, we allow the server to perform client-independent server preprocessing (i.e. SetupDB). In Respire, this consists of packing the database records into ring elements and applying the NTT transformation to the packed ring elements. The preprocessing cost scales linearly with the size of the database. This precomputation takes 16.8 s for a 256 MB database, and 569 s for an 8 GB database.

4.3 Supporting Batch Queries

In this section, we show how to combine the batched version of Respire (Section 3.2) with probabilistic batch codes [IKOS04, ACLS18] to support small batches of queries. Our goal in this comparison is to show that the base version of Respire readily extends to support batch queries, and composition with more recent PIR-to-batch-PIR transformations [BPSY24] should only offer further improvements. We describe our general methodology and evaluation below.

Cuckoo hashing. To improve server throughput in the batch setting, we use the (probabilistic) batch codes technique from [IKOS04, ACLS18]. In the approach from [ACLS18], the server starts by creating B (empty) buckets and samples h independent hash functions. The server hashes each element of the database into h buckets using the h hash functions. The hash functions are public and known to the client. To query for a batch of T indices $i_1, \ldots, i_T$, the client uses cuckoo hashing [PR01] (with the h hash functions) to associate a distinct bucket index for each index. The client then performs a standard PIR query on each bucket to request the record of interest. The observation is that each of these PIR queries is over an individual bucket, which is significantly smaller than the overall size of the database. Thus, the server no longer needs to perform a linear scan over the full database to respond to each of the i_T queries; instead, it needs to perform a linear scan over the entries in each bucket. Concretely, the work of [ACLS18] shows that when $h = 3$ and the number of buckets is roughly $B \approx 3T/2$, the probability of a cuckoo hashing failure (i.e., that the client is unable to associate a unique index with each desired index) is at most 2^{-40}. With $B \approx 3T/2$ buckets, and modeling the hash functions as *random* (as in [ACLS18]), the expected size of each bucket will be $\approx 2N/T$. Taken together, the server can answer a batch of T queries by performing $3T/2$ vanilla PIR queries, each over a database of size $2N/T$. For simplicity in our implementation, we choose the smallest value of $B \geq 3T/2$ such that every bucket has at most $K \leq 2N/T$, where K is a power-of-two. To support batch queries, we now run the batched version of Respire (Construction 3.3) (with the modification in Remark 3.4) with batch size B (i.e. the number of buckets) and database size K (i.e. the maximum size of each bucket). We provide sample parameters in Table 4 of Appendix E.

Database	Metric	Batch Size $T = 32$		Batch Size $T = 256$	
		VBPIR	RESPIRE	VBPIR	RESPIRE
$2^{20} \times$ 256B (256 MB)	Offline Comm.	9.3 MB	4.6 MB	9.3 MB	4.6 MB
	Query Size	578 KB	67.0 KB	1156 KB	326 KB
	Response Size	128 KB	31.8 KB	1028 KB	234 KB
	Computation	8.83 s	15.02 s	27.59 s	60.04 s
$2^{22} \times$ 256B (1 GB)	Offline Comm.	9.3 MB	4.6 MB	9.3 MB	4.6 MB
	Query Size	578 KB	113 KB	1735 KB	513 KB
	Response Size	128 KB	31.8 KB	771 KB	230 KB
	Computation	32.54 s	28.12 s	44.53 s	86.90 s

Table 2: Comparison of batched RESPIRE and Vectorized BatchPIR [MR23] (denoted "VBPIR") for two different database configurations and batch sizes T. Each database record is 256 bytes. The reference implementation of Vectorized BatchPIR does not report the size of their public parameters, so we report the number from the paper [MR23].

Parameter selection. We follow a similar methodology as described in Section 4.1 to choose parameters for RESPIRE to support batch queries. To allow a common basis of comparison in the batch setting, we choose parameters to ensure the *per-query* correctness error is at most 2^{-40}. We choose the vectorization dimension n_{vec} to balance the public parameter size and the response size. Namely, the size of the vectorization parameters pp_{vec} scales linearly with n_{vec}, whereas the size of the response scales with $\lceil T/(n_{\text{vec}} \cdot (d_2/d_3)\rceil$, where d_2 is the reduced ring dimension and d_3 is the dimension of the record. In the batch setting, we set $d_2 = 2048$ and $d_3 = 512$, and adjust n_{vec} to balance the response size and public parameter size. We refer to Table 4 in Appendix E for a list of our parameter choices.

Macrobenchmarks. Table 2 provides a breakdown of the batch version of RESPIRE to the Vectorized BatchPIR scheme [MR23]. Compared to Vectorized BatchPIR (a scheme *tailored for* batch queries), RESPIRE achieves a 3.4-8.5× reduction in query size and 3.4-4.4× reduction in response size (and 3.4-7.1× reduction in total communication). For small batches of queries and larger databases, RESPIRE is *also* slightly (16%) faster than Vectorized BatchPIR. However, for larger databases and batch sizes, there is about a 2.2× performance overhead with RESPIRE. As mentioned before, the improvements in communication is due to the new query and response compression techniques in RESPIRE and the ability to use smaller parameters due to better control of noise growth (see Remark 3.5).

Fig. 7 compares the computational costs of RESPIRE vs. Vectorized BatchPIR as a function of the database size and the batch size. For small batch sizes, RESPIRE outperforms Vectorized BatchPIR (batch size up to 16 for a 256 MB database and up to 128 for a 1 GB database). In applications where the client only makes a handful of queries at once (e.g., private blocklist checking, private DNS lookups), RESPIRE is preferred in both communication and computation. For large batch sizes, RESPIRE has smaller communication, but larger computational overheads. Schemes like Vectorized BatchPIR or Piranha [LLWR24] are better-suited for large batch sizes (hundreds to thousands in the case of Piranha) because they take advantage of SIMD support in FHE [GHS12a] to process the query. In contrast, RESPIRE starts from a communication-efficient single-query scheme and composes with batch codes and ring packing (for better communication). The batch codes approach allows us to amortize the cost of the linear scan over the database, but not the cost of query expansion (which scales linearly with the batch size). As we show below, the cost of query expansion becomes the dominating factor in our scheme, which makes it less suitable for very large batch sizes.

Microbenchmarks. Fig. 8 provides a breakdown of the server computation costs of batched RESPIRE as a function of the batch size T. The use of batch codes [IKOS04, ACLS18] allows us to amortize the cost of the linear scan (i.e., the first dimension processing) and essentially keeps it fixed as the batch size grows. However, the preprocessing (e.g., query expansion) and post-processing (e.g., folding and response compression) must still be applied individually to each query. As a result, these costs increase with the batch size. As noted above, for large batch sizes, query expansion dominates the

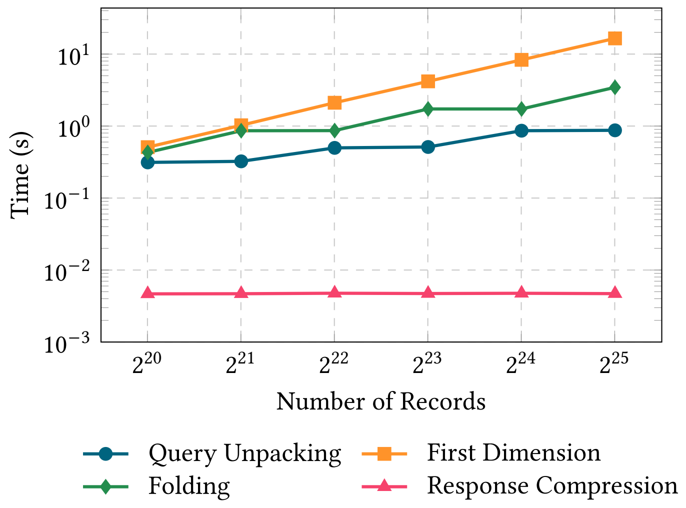


Figure 6: Server computation breakdown for RESPIRE as a function of the number of records. Each record is 256 bytes.

overall cost of the computation. As such, RESPIRE is better-suited for applications with small to moderate batch sizes.

5 Related Work

Chor et al. [CGKS95] first introduced private information retrieval in the *multi-server* setting where the database is replicated across multiple non-colluding servers. The multi-server model allows lightweight information-theoretic constructions [BIKR02, BIK05, WY05, Yek07, Efr09, BIKO12] as well as highly-efficient constructions based on computational assumptions [GI14, BGI16, HH19]. While this model yields schemes with excellent concrete efficiency, the reliance on multiple non-colluding servers raises challenges for deployment. Our focus in this work is on the single-server setting.

Single-server PIR. Starting from the seminal work of Kushilevitz and Ostrovsky [KO97], many works have constructed single-server PIR from different number-theoretic assumptions [CMS99, Cha04, GR05, OI07, DGI+19, BV11, CGH+21, ALP+21, BCM22]. The most concretely-efficient schemes are those based on lattice assumptions [MBFK16, AS16, ACLS18, GH19, PT20, ALP+21, AYA+21, MCR21, MW22a, MR23, DPC23, HHC+23, LMRS24, MW24, dCLS24]. The recent lattice-based schemes can be partitioned into three broad categories: (1) schemes with a client-specific hint [AS16, ACLS18, GH19, PT20, ALP+21, AYA+21, MCR21, MW22a]; (2) schemes with a database-specific hint [DPC23, HHC+23]; and (3) hintless schemes [LMRS24, MW24, dCLS24]. In the first category, clients first upload a small public key (typically, a set of key-switching matrices) to the server. The server uses these parameters for both query expansion and response compression; as such, the communication requirements on these protocols is much smaller than their counterparts. Conversely, in schemes with a database-specific hint, clients first download a (large) database-dependent hint in an offline phase. These schemes support extremely high throughput (comparable to the memory-bandwidth of the system) and are the fastest constructions to date. However, the offline computation in these schemes are often high and moreover, clients will have to refresh their hints whenever the database changes. Recently, several works have shown how to eliminate the hint altogether with a modest cost in communication and throughput. However, these schemes still incur substantial communication overhead (due to the lack of support for query and response compression); see Section 4.2 and Table 1. Several works have also studied augmenting PIR with stronger security in the presence of malicious servers [WZ18, BKP22, CNC+23, DT24, dCL24].

Sublinear PIR. Several recent works [MSR23, GZS24, ZPSZ24] have shown how to construct single-server PIR schemes in the *preprocessing* model where in an offline phase, the client first streams the *entire* database (and precom-

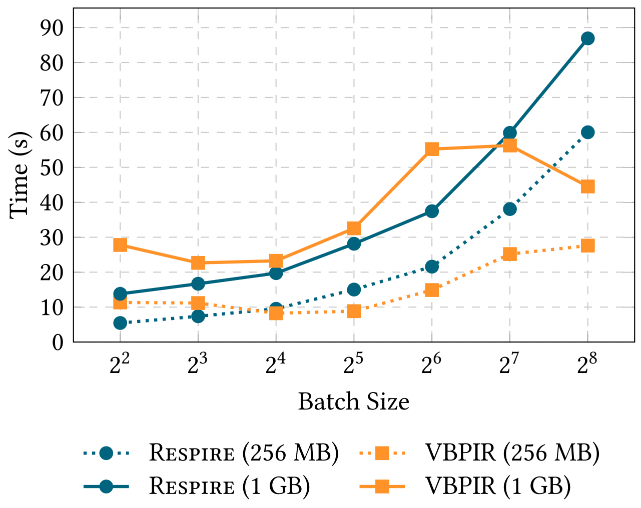


Figure 7: Comparison of batched RESPIRE and Vectorized BatchPIR (VBPIR) end-to-end execution times per query. We fix the size of each record to 256 bytes and the database size to be either 256 MB or 1 GB (indicated in the legend).

putes an $O(\sqrt{N})$-size hint, where N is the size of the database). Then in the online phase, the server can answer queries in sublinear time. More recently, Lin et al. [LMW23] showed how to construct *doubly-efficient* PIR [CHR17, BIPW17] from the RLWE assumption. In this model, the server first encodes the database in a way that allows it to answer queries in sublinear time; impressively, no communication is needed in the offline phase. Doubly-efficient PIR is a powerful primitive, but is still far from being concretely efficient [OPPW23].

Acknowledgments

We thank Steph Cheng for her help in an initial implementation of the RESPIRE protocol. David J. Wu is supported in part by NSF CNS-2140975, CNS-2318701, a Microsoft Research Faculty Fellowship, a Google Research Scholar award, and a grant from Protocol Labs.

References

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-kyber algorithm specifications and supporting documentation (version 3.02). *NIST PQC Round*, 2021.

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, 2018.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE S&P*, 2018.

[ALP+21] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In *USENIX Security Symposium*, 2021.

[APS15a] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3), 2015.

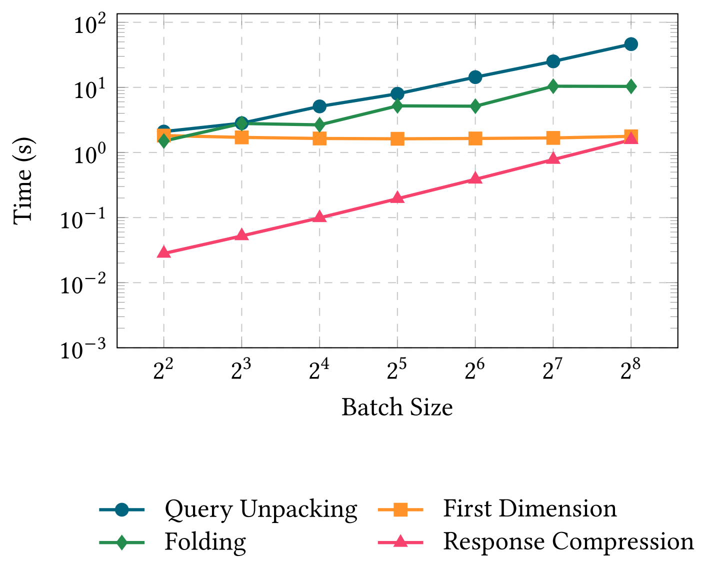


Figure 8: Microbenchmarks for the *batched* Respire protocol as a function of the batch size. We consider a database with 2^{20} records, each 256 bytes.

[APS15b] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors, 2015. https://github.com/malb/lattice-estimator.

[AS16] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, 2016.

[AYA+21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *OSDI*, 2021.

[BCM22] Elette Boyle, Geoffroy Couteau, and Pierre Meyer. Sublinear secure computation from new assumptions. In *TCC*, 2022.

[Ber08] Daniel J. Bernstein. ChaCha, a variant of Salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *ACM CCS*, 2016.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *ITCS*, 2012.

[BIK05] Amos Beimel, Yuval Ishai, and Eyal Kushilevitz. General constructions for information-theoretic private information retrieval. *J. Comput. Syst. Sci.*, 71(2), 2005.

[BIKO12] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. Share conversion and private information retrieval. In *CCC*, 2012.

[BIKR02] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Jean-François Raymond. Breaking the o(n1/(2k-1)) barrier for information-theoretic private information retrieval. In *FOCS*, 2002.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*, 2000.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC*, 2017.

[BKP22] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In *TCC*, 2022.

[BKS$^+$21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe D. M. de Souza, and Vinodh Gopal. Intel HEXL: accelerating homomorphic encryption with intel AVX512-IFMA52. In *WAHC*, 2021.

[BPSY24] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Batch PIR and labeled PSI with oblivious ciphertext compression. In *USENIX Security*, 2024.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *CRYPTO*, 2012.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *FOCS*, 2011.

[CCR19] Hao Chen, Ilaria Chillotti, and Ling Ren. Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE. In *ACM CCS*, 2019.

[CDKS21] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient homomorphic conversion between (ring) LWE ciphertexts. In *ACNS*, 2021.

[CGGI18] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *IACR Cryptol. ePrint Arch.*, 2018.

[CGGI20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1), 2020.

[CGH$^+$21] Melissa Chase, Sanjam Garg, Mohammad Hajiabadi, Jialin Li, and Peihan Miao. Amortizing rate-1 OT and applications to PIR and PSI. In *TCC*, 2021.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[Cha04] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC*, 2017.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT*, 2017.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.

[CNC$^+$23] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. In *USENIX Security Symposium*, 2023.

[dCL24] Leo de Castro and Keewoo Lee. VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers. *USENIX Security Symposium*, 2024.

[dCLS24] Leo de Castro, Kevin Lewi, and Edward Suh. WhisPIR: Stateless private information retrieval with low communication. *IACR Cryptol. ePrint Arch.*, 2024.

[DGI$^+$19] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In *CRYPTO*, 2019.

[DPC23] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *Proc. Priv. Enhancing Technol.*, 2023(1), 2023.

[DT24] Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. In *CRYPTO*, 2024.

[Efr09] Klim Efremenko. 3-query locally decodable codes of subexponential length. In *STOC*, 2009.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012.

[GH19] Craig Gentry and Shai Halevi. Compressible FHE with applications to PIR. In *TCC*, 2019.

[GHPS12] Craig Gentry, Shai Halevi, Chris Peikert, and Nigel P. Smart. Ring switching in BGV-style homomorphic encryption. In *SCN*, 2012.

[GHS12a] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, 2012.

[GHS12b] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[GKL10] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *PKC*, 2010.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO*, 2013.

[GZS24] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing PIR without public-key cryptography. In *EUROCRYPT*, 2024.

[HDCZ23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with Tiptoe. In *SOSP*, 2023.

[Hen16] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *Proc. Priv. Enhancing Technol.*, 2016(4), 2016.

[HH19] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR. *Proc. Priv. Enhancing Technol.*, 2019(4), 2019.

[HHC+23] 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 *USENIX Security Symposium*, 2023.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[KLDF16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proc. Priv. Enhancing Technol.*, 2016(2), 2016.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, 1997.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography and Data Security*, 2015.

[LLWR24] Jian Liu, Jingyu Li, Di Wu, and Kui Ren. PIRANA: Faster multi-query PIR via constant-weight codes. In *IEEE S&P*, 2024.

[LMPR08] Vadim Lyubashevsky, Daniele Micciancio, Chris Peikert, and Alon Rosen. SWIFFT: A modest proposal for FFT hashing. In *Fast Software Encryption*, 2008.

[LMRS24] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz. Hintless single-server private information retrieval. In *CRYPTO*, 2024.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In *STOC*, 2023.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *CANS*, 2016.

[LPA$^+$19] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *ACM CCS*, 2019.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *EUROCRYPT*, 2010.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *Proc. Priv. Enhancing Technol.*, 2016(2), 2016.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server PIR. In *ACM CCS*, 2021.

[MOT$^+$11] Prateek Mittal, Femi G. Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. Pir-tor: Scalable anonymous communication using private information retrieval. In *USENIX Security Symposium*, 2011.

[MP12] Daniele Micciancio and Chris Peikert. Trapdoors for lattices: Simpler, tighter, faster, smaller. In *EUROCRYPT*, 2012.

[MR23] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. In *IEEE S&P*, 2023.

[MSR23] Muhammad Haris Mughees, I Sun, and Ling Ren. Simple and practical amortized sublinear private information retrieval. *Cryptology ePrint Archive*, 2023.

[MW22a] Samir Jordan Menon and David J. Wu. SPIRAL: fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[MW22b] Samir Jordan Menon and David J. Wu. SPIRAL: fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022. Available at `https://github.com/blyssprivacy/sdk/tree/c93fff0`.

[MW24] Samir Jordan Menon and David J. Wu. YPIR: High-throughput single-server PIR with silent preprocessing. In *USENIX Security Symposium*, 2024.

[OI07] Rafail Ostrovsky and William E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *PKC*, 2007.

[OPPW23] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. Towards practical doubly-efficient private information retrieval. *IACR Cryptol. ePrint Arch.*, 2023.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, 2001.

[PT20] Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval. In *ESORICS*, 2020.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC*, 2005.

[TPY+19] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*, 2019.

[WY05] David P. Woodruff and Sergey Yekhanin. A geometric approach to information-theoretic private information retrieval. In *CCC*, 2005.

[WZ18] Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In *ICICS*, 2018.

[Yek07] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. In *STOC*, 2007.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO*, 2023.

[ZPSZ24] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. In *IEEE S&P*, 2024.

A Details on Lattice Algorithms

In this section, we give a formal description and the noise analysis for the different lattice algorithms we use in the construction of Respire.

Terminology. Throughout, we say an algorithm is efficient if it runs in probabilistic polynomial time in the length of its input. We say a function is negligible (denoted $\mathsf{negl}(\lambda)$) if it is $o(\lambda^{-c})$ for all $c \in \mathbb{N}$. We say two families of distributions $\mathcal{D}_1$ and $\mathcal{D}_2$ are computationally indistinguishable if no efficient algorithm can distinguish them except with negligible probability.

Discrete Gaussians. The Gaussian function with width parameter $\sigma > 0$ is the function $\rho_\sigma \colon \mathbb{R} \to \mathbb{R}^+$ where $\rho_\sigma(x) = \exp(-\pi x^2/\sigma^2)$. The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}$ of width σ is defined by the probability mass function $D_{\mathbb{Z},\sigma}(x) = \rho_\sigma(x)/\sum_{y\in\mathbb{Z}} \rho_\sigma(y)$. We say that a random variable X is subgaussian with parameter σ if for all $t \ge 0$, $\Pr[|X| \ge t] \le 2\exp(-\pi t^2/\sigma^2)$. We refer to σ^2 as the *variance* of the subgaussian distribution. If X is sampled from a discrete Gaussian distribution with width σ, then it is subgaussian with parameter σ (and variance σ^2).[5] If X_1, X_2 are independent subgaussian random variables with variances σ_1^2, σ_2^2, respectively, then for all $c_1, c_2 \in \mathbb{R}$, $c_1X_1 + c_2X_2$ is subgaussian with variance $c_1^2\sigma_1^2 + c_2^2\sigma_2^2$.

Polynomial rings. Throughout this section, we write R_d to denote the polynomial ring $R_d = \mathbb{Z}[x]/(x^d + 1)$, where d is a power of two. For a modulus q, we write $R_{d,q} = \mathbb{Z}_q[x]/(x^d + 1)$. For an element $f = \sum_{i\in[d]} \alpha_i x^{i-1} \in R_d$, we write $\|f\|_\infty$ to denote the ℓ_∞ norm of the coefficient vector $[\alpha_1, \dots, \alpha_d]$. When $f \in R_{d,q}$, we write $\|f\|_\infty$ to denote the ℓ_∞ norm of the coefficient vector of f where each coefficient is associated with its integer representative in the interval $(-q/2, q/2]$. Similarly, we write $\|f\|_2$ to denote the ℓ_2 norm of the coefficient vector of f. For all polynomials $f, g \in R_d$, it holds that $\|fg\|_\infty \le d\|f\|_\infty\|g\|_\infty$. For a vector $\mathbf{f} = (f_1, \dots, f_t) \in R_{d,q}^t$, we write $\|\mathbf{f}\|_\infty$ to denote the ℓ_∞ norm of the vector of the concatenation of the coefficient vectors of $(f_1, \dots, f_t)$. We define $\|\mathbf{f}\|_2$ analogously. We define the discrete Gaussian distribution of width σ over R_d to be the distribution that samples each coefficient α_i *independently* from $D_{\mathbb{Z},\sigma}$ and outputting $f = \sum_{i\in[d]} \alpha_i x^{i-1}$. We say f is sampled from a subgaussian distribution with parameter σ if each coefficient of f is sampled from a subgaussian distribution with parameter σ. We say a distribution $\mathcal{D}$ over R_d is B-bounded if $\Pr[\|r\|_\infty \le B : r \leftarrow \mathcal{D}] = 1$. In our analysis, we use the following bound:

Lemma A.1 (Subgaussian Polynomial Product). *Let $R_d = \mathbb{Z}[x]/(x^d + 1)$. Take any vector of polynomials $\mathbf{g} \in R_d^t$. Let $\mathbf{f} = (f_1, \dots, f_t) \in R_d^t$ be a vector where the coefficients of each f_i is sampled independently from a subgaussian distribution with variance σ^2. Then the distribution of each coefficient of $\mathbf{f}^\mathsf{T}\mathbf{g}$ is subgaussian with parameter $\|\mathbf{g}\|_2^2 \cdot \sigma^2$.*

[5]In this context, σ is the *width* of the discrete Gaussian, and not its standard deviation. The standard deviation s of the Gaussian distribution is related to the width parameter by the relation $s = \sigma/\sqrt{2\pi}$. Correspondingly, the "variance" of the Gaussian distribution (as defined in the usual sense by the relation $\mathbb{E}[(X - \mathbb{E}[X])^2]$) is $s^2 = \sigma^2/2\pi$. In this work, we will always write variance to denote the square of the subgaussian width parameter.

Proof. We start with the case where $t = 1$. Let $f = \sum_{i \in [0,d-1]} f_i x^i$ and $g = \sum_{i \in [0,d-1]} g_i x^i$. Let $h = fg = \sum_{i \in [0,d-1} h_i x^i \in R_d$. By definition, for all $i \in [0, d-1]$,

$$h_i = \sum_{j \in [0,d-1]} (-1)^{c_j} f_j g_{i-j \bmod d},$$

for some choice of $c_j \in \{0,1\}$. Since each f_j is independent and subgaussian with variance σ^2, the distribution of h_i is subgaussian with variance $\sum_{j \in [0,d-1]} g_j^2 \sigma^2 = \|g\|_2^2 \cdot \sigma^2$. When $t > 1$, $\mathbf{f}^{\mathsf{T}}\mathbf{g} = \sum_{i \in [t]} f_i g_i$. The coefficients of each $f_i g_i$ is subgaussian with variance $\|g_i\|_2^2 \cdot \sigma^2$. Since each component is independent, the sum is subgaussian with variance $\sum_{i \in [t]} \|g_i\|_2^2 \sigma^2 = \|\mathbf{g}\|_2^2 \cdot \sigma^2$. □

Independence heuristic. Similar to previous lattice-based PIR schemes based on polynomial rings [ACLS18, GH19, MCR21, MW22a, LMRS24, MW24], we rely on the independence heuristic [GHS12b, CGGI18, CGGI20] when analyzing the error accumulated during homomorphic computations. Under the independence heuristic, we model the (subgaussian) error terms arising in the homomorphic operations as being independent. Moreover, instead of bounding the absolute magnitude (i.e., the worst-case error), we analyze the variance of the subgaussian error distribution instead. Since the variance is additive for *independent* subgaussian random variables, bounding the variance yields a square-root improvement in the error analysis compared to the worst-case bound (when considering sums of subgaussian random variables). We stress that the use of the independence heuristic only impacts the correctness error in the protocol (and *not* the security of the protocol). Empirically, we observe that there is still slack between the magnitude of the error predicted based on our analysis (assuming the independence heuristic) and the actual measured noise magnitude. Thus, we believe that our estimates for the correctness error computed under the independence heuristic is still an *overestimate* of the actual correctness error.

External product. We now recall the external product from [CGGI18, CGGI20]. We define the algorithm and state the correctness property below. Our presentation is adapted from that of [MW22a]:

- $\mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_{\mathsf{RLWE}})$: On input a GSW encoding $\mathbf{C}_{\mathsf{GSW}} \in R_{d,q}^{2 \times m}$ with decomposition base $z \in \mathbb{N}$ and an RLWE encoding $\mathbf{c}_{\mathsf{RLWE}} \in R_{d,q}^2$, output $\mathbf{C}_{\mathsf{GSW}} \mathbf{G}_{2,z}^{-1}(\mathbf{c}_{\mathsf{RLWE}}) \in R_{d,q}^2$.

Theorem A.2 (External Product [CGGI18, CGGI20, adapted])**.** *Let $\mathbf{s} \in R_{d,q}^2$ be a secret key. Suppose $\mathbf{C}_{\mathsf{GSW}} \in R_{d,q}^{2\times m}$ is a GSW encoding of a message $\mu \in \{0,1\}$ with respect to $\mathbf{s}$, error $\mathbf{e}_{\mathsf{GSW}} \in R_{d,q}^m$, and decomposition base z. Suppose $\mathbf{c}_{\mathsf{RLWE}} \in R_{d,q}^2$ is an RLWE encoding of a scalar $v \in R_{d,q}$ with respect to the secret key $\mathbf{s}$ and error $e \in R_d$. Let $\mathbf{c} \leftarrow \mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_{\mathsf{RLWE}})$. Then, $\mathbf{c}$ is an RLWE encoding of $\mu v \in R_{d,q}$ with respect to the secret key $\mathbf{s}$ and error $e = \mu e_{\mathsf{RLWE}} + \mathbf{e}_{\mathsf{GSW}}^{\mathsf{T}} \mathbf{G}_{2,z}^{-1}(\mathbf{c}_{\mathsf{RLWE}}) \in R_{d,q}$.*

Homomorphic selection. We now define the homomorphic selection algorithm:

- $\mathsf{Select}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_0, \mathbf{c}_1) \to \mathbf{c}'$: On input a GSW encoding $\mathbf{C}_{\mathsf{GSW}} \in R_{d,q}^{2\times m}$ and RLWE encodings $\mathbf{c}_0, \mathbf{c}_1 \in R_{d,q}^2$, output $\mathbf{c}_0 + \mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_1 - \mathbf{c}_0)$.

Theorem A.3 (Homomorphic Selection)**.** *Let $\mathbf{s} \in R_{d,q}^2$ be a secret key. Let $\mathbf{c}_0, \mathbf{c}_1$ be RLWE encodings of $\mu_0, \mu_1 \in R_{d,q}$ with respect to the secret key $\mathbf{s}$ and errors $e_0, e_1 \in R_d$, respectively. Let $\mathbf{C}_{\mathsf{GSW}}$ be a GSW encoding of a bit $b \in \{0,1\}$ with respect to the secret key $\mathbf{s}$ and error $\mathbf{e} \in R_{d,q}^m$. Suppose e_0, e_1 are subgaussian with variance σ_{RLWE}^2 and the components of $\mathbf{e}$ are subgaussian with variance σ_{GSW}^2. Let $\mathbf{c}' \leftarrow \mathsf{Select}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_0, \mathbf{c}_1)$. Then $\mathbf{c}' \in R_{d,q}^2$ is an RLWE encoding of μ_b with respect to the secret key $\mathbf{s}$ and error e'. Moreover, e' is subgaussian with variance $\sigma^2 \leq \sigma_{\mathsf{RLWE}}^2 + mdz^2\sigma_{\mathsf{GSW}}^2/4$.*

Proof. Let $\hat{\mathbf{c}}' \leftarrow \mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{c}_1 - \mathbf{c}_0)$. By Theorem A.2, $\hat{\mathbf{c}}'$ is an RLWE encoding of $b(\mu_1 - \mu_0)$ with error

$$\hat{e}' = \mu(e_1 - e_0) + \mathbf{e}_{\mathsf{GSW}}^{\mathsf{T}} \mathbf{G}_{2,z}^{-1}(\mathbf{c}_1 - \mathbf{c}_0).$$

By the additively homomorphism of RLWE encodings, we conclude that $\mathbf{c}' = \mathbf{c}_0 + \hat{\mathbf{c}}'$ is an RLWE encoding of $\mu_0 + b(\mu_1 - \mu_0) = \mu_b$ with error $e' = e_0 + \hat{e}' = e_b + \mathbf{e}_{\mathsf{GSW}}^{\mathsf{T}} \mathbf{G}_{2,z}^{-1}(\mathbf{c}_1 - \mathbf{c}_0)$. Since $\|\mathbf{G}_{2,z}^{-1}(\mathbf{c}_1 - \mathbf{c}_0)\|_2^2 \leq mdz^2/4$ and appealing to the independence heuristic, the variance σ^2 of e' satisfies the given bound. □

A.1 Coefficient Projection

In this section, we describe how to homomorphically apply a coefficient projection to an encoded polynomial (i.e., instantiate the algorithms in Box 3). This is an adaptation of the procedure used in [ACLS18, CCR19, CDKS21] for query expansion and packing RLWE encodings. The construction relies on the ability to homomorphically evaluate automorphisms on RLWE encodings [BGV12, GHS12a].

Automorphisms over R_d. For a positive integer $\ell \in \mathbb{N}$, we write $\tau_\ell \colon R_d \to R_d$ to denote the Frobenius automorphism that maps $f(x) \mapsto f(x^\ell)$. For a modulus $q \in \mathbb{N}$, we define the automorphism over $R_{d,q}$ in the same manner, and for ease of notation, write τ_ℓ to denote both automorphisms. We extend τ_ℓ to operate on vectors and matrices by component-wise evaluation. Previously, [BGV12, GHS12a] showed how to homomorphically apply automorphisms to RLWE encodings. We summarize the main algorithms here for the special case of scalar RLWE encodings (following the presentation from [MW22a, MW24]):

Construction A.4 (Automorphisms on RLWE Encodings [GHS12a, BGV12, adapted])**.** Let λ be a security parameter and $d = d(\lambda)$, $q = q(\lambda)$ be lattice parameters where $d = 2^\ell$ is a power of two. Let $R_d = \mathbb{Z}[x]/(x^d + 1)$ and $\chi = \chi(\lambda)$ be an error distribution over R_d. The construction is also parameterized by a decomposition base $z \in \mathbb{N}$. We now define the following algorithms:

- $\mathsf{AutomorphSetup}(1^\lambda, s, \tau)$: On input the security parameter λ, a secret key $\mathbf{s} = [-s \mid 1]^\intercal \in R_{d,q}^2$, and an automorphism $\tau \colon R_{d,q} \to R_{d,q}$, first define $t = \lfloor \log_z q \rfloor + 1$. Then, the setup algorithm samples $\mathbf{a} \xleftarrow{\mathrm{R}} R_{d,q}^t$ and $e \leftarrow \chi^t$ and outputs a key-switching matrix

$$\mathbf{W}_\tau = \begin{bmatrix} \mathbf{a}^\intercal \\ s\mathbf{a}^\intercal + \mathbf{e}^\intercal - \tau(s) \cdot \mathbf{g}_z^\intercal \end{bmatrix} \in R_{d,q}^{2 \times t}. \tag{A.1}$$

- $\mathsf{Automorph}(\mathbf{W}, \mathbf{c}, \tau)$: On input the key-switching matrix $\mathbf{W} \in R_{d,q}^{2\times t}$, an RLWE encoding $\mathbf{c} = (c_0, c_1) \in R_{d,q}^2$, an automorphism $\tau \colon R_{d,q} \to R_{d,q}$, and a decomposition base $z \in \mathbb{N}$, the automorph algorithm outputs

$$\mathbf{c}' = \mathbf{W} \cdot \mathbf{g}_z^{-1}(\tau(c_0)) + \begin{bmatrix} 0 \\ \tau(c_1) \end{bmatrix} \in R_{d,q}^2. \tag{A.2}$$

Theorem A.5 (Homomorphic Evaluation of Automorphisms [GHS12a, BGV12, adapted])**.** *For a positive integer $\ell \in \mathbb{N}$, let $\tau_\ell \colon R_{d,q} \to R_{d,q}$ be the automorphism $p(x) \mapsto p(x^\ell)$ and $z \in \mathbb{N}$ be a decomposition base. Let $\mathbf{s} = [-s \mid 1]^\intercal \in R_{d,q}^2$ be a secret key and $\mathbf{c} \in R_{d,q}^2$ be any encoding. Let $\mathbf{W}_\tau \leftarrow \mathsf{AutomorphSetup}(1^\lambda, s, \tau_\ell)$ and $\mathbf{c}' \leftarrow \mathsf{Automorph}(\mathbf{W}_\tau, \mathbf{c}, \tau_\ell)$. Then, $\mathbf{s}^\intercal \mathbf{c}' = \tau(\mathbf{s}^\intercal \mathbf{c}) + e'$ where e' is subgaussian with variance $(\sigma')^2 \le tdz^2\sigma_\chi^2/4$ and $t = \lfloor \log_z q \rfloor + 1$.*

Coefficient projections. Recall from Section 3.2 (Eq. (3.6)) that the coefficient projection map $\pi_j \colon R_d \to R_d$ takes as input a polynomial $f = \sum_{i \in [0,d-1]} f_i x^i \in R_d$ and outputs $\sum_{i \in [0,d-1] : 2^j | i} f_i x^i$. We define the homomorphic projection map in terms of automorphisms as follows:

Lemma A.6 (Coefficient Projection using Automorphisms)**.** *Let $R_d = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^\ell$ for some $\ell \in \mathbb{N}$. Define $\pi_0(f) = f$. Then, for all $j \in [\ell]$,*

- $2 \cdot \pi_j(f) = \pi_{j-1}(f) + \tau_{d/2^{j-1}+1}(\pi_{j-1}(f))$.
- $2 \cdot \pi_j(f \cdot x^{-2^{j-1}}) = x^{-2^{j-1}} \cdot \left(\pi_{j-1}(f) - \tau_{d/2^{j-1}+1}(\pi_{j-1}(f))\right)$.

Proof. Recall that $\tau_\ell \colon R_d \to R_d$ is the automorphism that maps $f(x) \mapsto f(x^\ell)$. Over the ring $R_d = \mathbb{Z}[x]/(x^d + 1)$, observe that for all $j \in [0, \ell - 1]$ and all integers $c \in \mathbb{N}$,

$$\tau_{d/2^j+1}\left(x^{c\cdot 2^j}\right) = x^{cd + c\cdot 2^j} = (x^d)^c \cdot x^{c\cdot 2^j} = (-1)^c \cdot x^{c \cdot 2^j}. \tag{A.3}$$

Take any polynomial $f = \sum_{i \in [0,d-1]} f_i x^i$. By Eq. (A.3), we have for all $j \in [\ell]$,

$$\tau_{d/2^j+1}\big(\pi_j(f)\big) = \tau_{d/2^j+1}\left(\sum_{i\in[0,d-1]:2^j|i} f_i x^i\right) = \tau_{d/2^j+1}\left(\sum_{c\in[0,d/2^j-1]} f_{c\cdot 2^j} x^{c\cdot 2^j}\right) = \sum_{c\in[0,d/2^j-1]} (-1)^c \cdot f_{c\cdot 2^j} x^{c\cdot 2^j}.$$

We now consider each of the properties:

- For the first property, we have

$$\begin{aligned}\pi_{j-1}(f) + \tau_{d/2^{j-1}+1}\big(\pi_{j-1}(f)\big) &= \sum_{c\in[0,d/2^{j-1}-1]} f_{c\cdot 2^{j-1}} \cdot x^{c\cdot 2^{j-1}} + \sum_{c\in[0,d/2^{j-1}-1]} (-1)^c \cdot f_{c\cdot 2^{j-1}} \cdot x^{c\cdot 2^{j-1}} \\ &= \sum_{c\in[0,d/2^{j-1}-1]:2|c} 2f_{c\cdot 2^{j-1}} \cdot x^{c\cdot 2^{j-1}} \\ &= 2\cdot \pi_j(f).\end{aligned}$$

- The second property follows by a similar calculation:

$$\begin{aligned}\pi_{j-1}(f) - \tau_{d/2^{j-1}+1}\big(\pi_{j-1}(f)\big) &= \sum_{c\in[0,d/2^{j-1}-1]} f_{c\cdot 2^{j-1}} \cdot x^{c\cdot 2^{j-1}} - \sum_{c\in[0,d/2^{j-1}-1]} (-1)^c \cdot f_{c\cdot 2^{j-1}} \cdot x^{c\cdot 2^{j-1}} \\ &= \sum_{c\in[0,d/2^{j}-1]} 2f_{(2c+1)\cdot 2^{j-1}} \cdot x^{(2c+1)\cdot 2^{j-1}}.\end{aligned}$$

  Thus,

$$x^{-2^{j-1}} \cdot \big(\pi_{j-1}(f) - \tau_{d/2^{j-1}+1}(\pi_{j-1}(f))\big) = \sum_{c\in[0,d/2^j-1]} 2f_{(2c+1)\cdot 2^{j-1}} \cdot x^{c\cdot 2^j}.$$

  Finally,

$$\begin{aligned}2\cdot \pi_j\big(f\cdot x^{-2^{j-1}}\big) = \sum_{c\in[0,d/2^j-1]} 2f_{c\cdot 2^j + 2^{j-1}} \cdot x^{c\cdot 2^j} &= \sum_{c\in[0,d/2^j-1]} 2f_{(2c+1)\cdot 2^{j-1}} \cdot x^{c\cdot 2^j} \\ &= x^{-2^{j-1}} \cdot \big(\pi_{j-1}(f) - \tau_{d/2^{j-1}+1}(\pi_{j-1}(f))\big),\end{aligned}$$

  as required. □

Evaluating coefficient projections on RLWE encodings. We now describe the (ProjectSetup, Project) algorithms from Box 3. The construction and analysis are similar to the algorithms from [ACLS18, CCR19, CDKS21].

Construction A.7 (Coefficient Projection on RLWE Encodings)**.** Let λ be a security parameter and $d = d(\lambda)$, $q = q(\lambda)$ be lattice parameters where $d = 2^\ell$ is a power of two. Let $R_d = \mathbb{Z}[x]/(x^d+1)$ and $\chi = \chi(\lambda)$ be an error distribution over R_d. We require that the modulus q satisfy $q = 1 \bmod 2$. The construction is also parameterized by a decomposition base $z \in \mathbb{N}$. The construction relies on the (AutomorphSetup, Automorph) algorithms from Construction A.4 instantiated with the same lattice parameters (d, q, χ) and decomposition base z. We define the algorithms (ProjectSetup, Project) as follows:

- ProjectSetup$(1^\lambda, \mathbf{s})$: On input the security parameter λ and the secret key $\mathbf{s} \in R_{d,q}^2$, the setup algorithm starts by sampling $\mathbf{W}_j \leftarrow \text{AutomorphSetup}(1^\lambda, \mathbf{s}, \tau_{d/2^j+1})$ for each $j \in [0, \ell-1]$. It outputs the projection key $\text{pp}_{\text{proj}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\ell-1})$.
- Project$(\text{pp}_{\text{proj}}, \mathbf{c}, j)$: On input a projection key $\text{pp}_{\text{proj}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\ell-1})$, an RLWE encoding $\mathbf{c} \in R_{d,q}^2$ and an index $j \in [0, \ell]$, the projection algorithm proceeds as follows:
  - Compute $\mathbf{c}_0 = 2^{-j} \cdot \mathbf{c} \in R_{d,q}^2$. Note that $q = 1 \bmod 2$ so 2 is invertible modulo q.

  – For each $i \in [j]$, compute $\mathbf{c}_i = \mathbf{c}_{i-1} + \mathsf{Automorph}(\mathbf{W}_{i-1}, \mathbf{c}_{i-1}, \tau_{d/2^{i-1}+1})$.

At the end of the process, output $\mathbf{c}_j$.

Theorem A.8 (Coefficient Projection on RLWE Encodings). *Let λ be a security parameter and d, ℓ, q, χ, z be the parameters in Construction A.7. Suppose χ is subgaussian with variance σ_χ^2. Let $\mathbf{s} = [-s \mid 1]^\mathsf{T} \in R_{d,q}^2$ be a secret key and $\mathbf{c} \in R_{d,q}^2$ be any encoding. Take any $j \in [\ell]$. Let $\mathsf{pp}_{\mathsf{proj}} \leftarrow \mathsf{ProjectSetup}(1^\lambda, \mathbf{s})$ and $\mathbf{c}' \leftarrow \mathsf{Project}(\mathsf{pp}_{\mathsf{proj}}, \mathbf{c}, j)$. Then $\mathbf{s}^\mathsf{T}\mathbf{c}' = \pi_j(\mathbf{s}^\mathsf{T}\mathbf{c}) + e'$ and under the independence heuristic, e' is subgaussian with variance $(\sigma')^2 = (4^j - 1)/12 \cdot tdz^2\sigma_\chi^2$ and $t = \lfloor \log_z q \rfloor + 1$.*

Proof. Let $\mathsf{pp}_{\mathsf{proj}} \leftarrow \mathsf{ProjectSetup}(1^\lambda, \mathbf{s})$. Then $\mathsf{pp}_{\mathsf{proj}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\ell-1})$ where $\mathbf{W}_j \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}, \tau_{d/2^j+1})$. Let $\mathbf{c}_0, \ldots, \mathbf{c}_j$ be the encodings constructed by $\mathsf{Project}(\mathsf{pp}_{\mathsf{proj}}, \mathbf{c}, j)$. We now show that for all $i \in [0, j]$,

$$\mathbf{s}^\mathsf{T}\mathbf{c}_i = 2^{-j+i}\pi_i(\mathbf{s}^\mathsf{T}\mathbf{c}) + e_i,$$

where $e_0 = 0$ and e_i is subgaussian with variance $\sigma_i^2 = \sum_{k\in[i]} 4^{k-1} tdz^2\sigma_\chi^2/4$. We proceed by induction on i:

- Suppose $i = 0$. By definition, $\mathbf{c}_0 = 2^{-j} \cdot \mathbf{c}$ and the claim holds (since $\pi_0(r) = r$ for all $r \in R_d$).
- For the inductive step, let $\mathbf{c}_i' = \mathsf{Automorph}(\mathbf{W}_i, \mathbf{c}_i, \tau_{d/2^i+1})$. By Theorem A.5 and the inductive hypothesis,

  $$\mathbf{s}^\mathsf{T}\mathbf{c}_i' = \tau_{d/2^i+1}(\mathbf{s}^\mathsf{T}\mathbf{c}_i) + \tilde{e}_{i+1} = \tau_{d/2^i+1}\big(2^{-j+i}\pi_i(\mathbf{s}^\mathsf{T}\mathbf{c}) + e_i\big) + \tilde{e}_{i+1},$$

  where $\tilde{e}_{i+1}$ is subgaussian with variance $tdz^2\sigma_\chi^2/4$. Since $\mathbf{c}_{i+1} = \mathbf{c}_i + \mathbf{c}_i'$, and appealing to Lemma A.6, we have

  $$\begin{aligned}\mathbf{s}^\mathsf{T}\mathbf{c}_{i+1} &= 2^{-j+i}\big(\pi_i(\mathbf{s}^\mathsf{T}\mathbf{c}) + \tau_{d/2^i+1}(\pi_i(\mathbf{s}^\mathsf{T}\mathbf{c}))\big) + e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1} \\ &= 2^{-j+i+1}\pi_{i+1}(\mathbf{s}^\mathsf{T}\mathbf{c}) + e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}.\end{aligned}$$

  Let $e_{i+1} := e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}$. Since e_i is subgaussian with variance σ_i^2 and appealing to the independence heuristic (to argue that the key-switching error $\tilde{e}_{i+1}$ is independent of e_i), we have that e_{i+1} is subgaussian with variance

  $$\sigma_{i+1}^2 = 4\sigma_i^2 + tdz^2\sigma_\chi^2/4 = 4 \cdot \sum_{k\in[i-1]} 4^{k-1} tdz^2\sigma_\chi^2/4 + tdz^2\sigma_\chi^2/4 = \sum_{k\in[i]} 4^{k-1} tdz^2\sigma_\chi^2/4.$$

Finally, the claim follows since $e' = e_j$ which has variance

$$(\sigma')^2 = \sigma_j^2 = tdz^2\sigma_\chi^2/4 \cdot \sum_{k\in[j]} 4^{k-1} = \frac{4^j - 1}{12} tdz^2\sigma_\chi^2. \qquad \square$$

A.2 Subring Embeddings and Dimension Reduction

In this section, we formally define the subring embedding (Eq. (3.1)) and dimension reduction (Eq. (3.2)) mappings from Section 3. We then show some basic algebraic properties on these functions that will be useful in the subsequent analysis.

Definition A.9 (Subring Embedding and Dimension Reduction). Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1} + 1)$ and $R_{d_2} = \mathbb{Z}[x]/(x^{d_2} + 1)$ where d_2 divides d_1. Define the embedding function $\kappa_{d_1,d_2} \colon R_{d_2} \to R_{d_1}$ to be the mapping

$$\sum_{i\in[0,d_2-1]} f_i x^i \in R_{d_2} \mapsto \sum_{i\in[0,d_2-1]} f_i x^{i\cdot d_1/d_2} \in R_{d_1}. \tag{A.4}$$

We also define the dimension-reduction mapping $\kappa_{d_1,d_2}^{-1} \colon R_{d_1} \to R_{d_2}$ to be the mapping

$$\sum_{i\in[0,d_1-1]} f_i x^i \in R_{d_2} \mapsto \sum_{i\in[0,d_2-1]} f_{i\cdot d_1/d_2} x^i \in R_{d_1}. \tag{A.5}$$

When the dimensions d_1, d_2 are clear from context, we simply write κ and κ^{-1} to denote κ_{d_1,d_2} and κ_{d_1,d_2}^{-1}, respectively. We extend κ, κ^{-1} to operate on vectors and matrices in a component-wise manner. For a modulus $q \in \mathbb{N}$, we define $\kappa \colon R_{d_2,q} \to R_{d_1,q}$ and $\kappa^{-1} \colon R_{d_1,q} \to R_{d_2,q}$ in an analogous manner.

Lemma A.10 (Subring Embedding). *Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1}+1)$ and $R_{d_2} = \mathbb{Z}[x]/(x^{d_2}+1)$ where d_2 divides d_1. Let $\kappa\colon R_{d_2} \to R_{d_1}$ and $\kappa^{-1}\colon R_{d_1} \to R_{d_2}$ be the subring embedding and dimension reduction functions from Definition A.9, respectively. Then κ is an injective ring homomorphism (i.e., an embedding function). Specifically, the following properties hold:*

- **One-sided inverse:** *For all $r \in R_{d_2}$, it holds that $\kappa^{-1}(\kappa(r)) = r$. In particular, κ is injective.*
- **Linearity:** *For all $\alpha, \beta \in \mathbb{Z}$ and $r, s \in R_{d_2}$, $\kappa(\alpha r + \beta s) = \alpha\kappa(r) + \beta\kappa(s)$.*
- **Multiplicative homomorphism:** *For all $r, s \in R_{d_2}$, $\kappa(rs) = \kappa(r)\kappa(s)$.*
- **Scaling by rationals:** *For $\alpha \in \mathbb{Q}$ and $r \in R_{d_2}$, $\kappa(\lfloor \alpha r \rceil) = \lfloor \alpha\kappa(r) \rceil$, where the multiplication and rounding operations are performed over the rationals.*

Proof. The one-sided inverse, linearity, and scaling-by-rationals properties of κ follow immediately from the definition of κ (Eq. (A.4)). It suffices to show the multiplicative property. We first show that this hold for products of monomials. The claim then follows by linearity. Take any $i, j \in [0, d_2 - 1]$. Let $c = 0$ if $i + j < d_2$ and $c = 1$ if $i + j \geq d_2$. Then,

$$\kappa\big(x^i x^j\big) = \kappa\big((-1)^c x^{i+j \bmod d_2}\big) = (-1)^c x^{(i+j \bmod d_2)\cdot d_1/d_2} \in R_{d_1}.$$

Similarly,

$$\kappa(x^i) \cdot \kappa(x^j) = x^{i \cdot d_1/d_2} x^{j \cdot d_1/d_2} = (-1)^c x^{(i+j \bmod d_2)\cdot d_1/d_2} = \kappa\big(x^i x^j\big). \tag{A.6}$$

Next, take any $s = \sum_{j \in [0,d-1]} s_j x^j$ where $s_j \in \mathbb{Z}$. By linearity and Eq. (A.6), we have

$$\kappa\big(x^i s\big) = \kappa\left(\sum_{j \in [0,d-1]} s_j x^i x^j\right) = \sum_{j \in [0,d-1]} s_j \kappa(x^i)\kappa(x^j) = \kappa(x^i) \sum_{j \in [0,d-1]} \kappa(s_j x^j) = \kappa(x^i)\kappa(s).$$

Finally, let $r = \sum_{j \in [0,d-1]} r_j x^j$ where $r_j \in \mathbb{Z}$. Again by linearity, we have

$$\kappa(rs) = \sum_{j \in [0,d-1]} \kappa(r_j x^j s) = \sum_{j \in [0,d-1]} \kappa(r_j x^j)\kappa(s) = \kappa(r) \cdot \kappa(s). \qquad \square$$

Lemma A.11 (Subring Projection). *Suppose $d_1 = 2^{\delta_1}$ and $d_2 = 2^{\delta_2}$ for non-negative integers $d_1 \geq d_2$. Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1} + 1)$ and $R_{d_2} = \mathbb{Z}[x]/(x^{d_2} + 1)$ where d_2 divides d_1. Let $\nu = \delta_1 - \delta_2$ and $\pi_\nu\colon R_{d_1} \to R_{d_1}$ be the coefficient projection map from Eq. (3.6). Let $\kappa\colon R_{d_2} \to R_{d_1}$ and $\kappa^{-1}\colon R_{d_1} \to R_{d_2}$ be the subring embedding and dimension reduction mappings from Definition A.9. Then for all $r \in R_{d_1}$, it follows that*

$$\pi_\nu(r) = \kappa(\kappa^{-1}(r))$$

Moreover, the projection mapping π_ν satisfies the following properties:

- **Linearity:** *For all $r, s \in R_{d_1}$, it holds that $\pi_\nu(r + s) = \pi_\nu(r) + \pi_\nu(s)$.*
- **Projection:** *For all $r \in R_{d_2}$, $\pi_\nu(\kappa(r)) = \kappa(r)$. In particular, this means that for all $r \in R_{d_1}$, $\pi_\nu(\pi_\nu(r)) = \pi_\nu(r)$. In addition, for all $r \in R_{d_2}$ and $s \in R_{d_1}$, it holds that $\pi_\nu(\kappa(r)s) = \kappa(r)\pi_\nu(s)$.*
- **Scaling by rationals:** *For $\alpha \in \mathbb{Q}$ and $r \in R_{d_1}$, $\pi_\nu(\lfloor \alpha r \rceil) = \lfloor \alpha\pi_\nu(r) \rceil$ where the multiplication and rounding are performed over the rationals.*

Proof. We first show that $\pi_\nu(r) = \kappa(\kappa^{-1}(r))$. Take any $r = \sum_{i \in [0,d_1-1]} r_i x^i \in R_{d_1}$. Then,

$$\kappa(\kappa^{-1}(r)) = \kappa\left(\sum_{i \in [0,d_2-1]} r_{i\cdot(d_1/d_2)} x^i\right) = \sum_{i \in [0,d_2-1]} r_{i\cdot(d_1/d_2)} x^{i\cdot(d_1/d_2)} = \sum_{i \in [0,d_1-1] : 2^\nu \mid i} r_i x^i = \pi_\nu(i),$$

since $2^\nu = 2^{\delta_1-\delta_2} = d_1/d_2$. Linearity of π_ν and the scaling-by-rationals property now follow by the corresponding properties of κ (Lemma A.10) and the same properties of κ^{-1} (immediate from the definition). It suffices to show that π_ν satisfies the projection property. First, take any $r \in R_{d_2}$. Since $\kappa^{-1}(\kappa(r)) = r$, we have

$$\pi_\nu(\kappa(r)) = \kappa(\kappa^{-1}(\kappa(r))) = \kappa(r).$$

For the additional property, take $r \in R_{d_2}$ and $s \in R_{d_1}$. We start by showing that $\kappa^{-1}(\kappa(r)s) = r\kappa^{-1}(s)$. Since this equation is linear in both r and s, it suffices to prove the case where $r = x^i$ and $s = x^j$ are monomials (and then appeal to linearity of κ^{-1}). Indeed,

$$\kappa^{-1}(\kappa(r)s) = \kappa^{-1}\left(x^{\nu i+j}\right) = \begin{cases} x^{i+j/\nu} & \text{if } \nu \mid j \\ 0 & \text{otherwise} \end{cases} = \begin{cases} r \cdot x^{j/\nu} & \text{if } \nu \mid j \\ 0 & \text{otherwise} \end{cases} = r\kappa^{-1}(s).$$

Since $\kappa^{-1}(\kappa(r)s) = r\kappa^{-1}(s)$, we have that

$$\kappa(\kappa^{-1}(\kappa(r)s)) = \kappa(r\kappa^{-1}(s)).$$

Using the fact that $\pi_\nu(r) = \kappa(\kappa^{-1}(r))$ and linearity of κ, we conclude that

$$\pi_\nu(\kappa(r)s) = \kappa(r\kappa^{-1}(s)) = \kappa(r)\kappa(\kappa^{-1}(s)) = \kappa(r)\pi_\nu(s). \qquad \square$$

Ring packing. We now recall the definition of ring packing from Eq. (3.3). Here, we model the inputs as (arbitrary) ring elements (rather than database records).

Definition A.12 (Ring Packing)**.** Suppose $d_1 \geq d_2$ where d_2 divides d_1. Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1}+1)$ and $R_{d_2} = \mathbb{Z}[x]/(x^{d_2}+1)$. Let $k = d_1/d_2$ and suppose $r_0, \dots, r_{k-1} \in R_{d_2}$. Then we define the ring packing function $\Pi \colon R_{d_2}^k \to R_{d_1}$ as

$$\Pi(r_0, \dots, r_{k-1}) \coloneqq \sum_{i \in [0,k-1]} x^i \cdot \kappa(r_i),$$

where $\kappa \colon R_{d_2} \to R_{d_1}$ is the subring embedding function from Definition A.9.

Lemma A.13 (Ring Packing Extraction)**.** *With the notation of Definition A.12, we have*

$$\kappa^{-1}(x^{-t} \cdot \Pi(r_0, \dots, r_{k-1})) = r_t$$

for any $r_0, \dots, r_{k-1} \in R_{d_2}$ and $t \in [0, k-1]$.

Proof. For $i \in [0, k-1]$, let $r_i = \sum_{j \in [0,d_2-1]} x^j r_{i,j} \in R_{d_2}$ where each $r_{i,j} \in \mathbb{Z}$. Then, we can write

$$\Pi(r_0, \dots, r_{k-1}) = \sum_{i \in [0,k-1]} x^i \kappa(r_i) = \sum_{i \in [0,k-1]} \sum_{j \in [0,d_2-1]} r_{i,j} x^{i+kj} = \sum_{i \in [0,d_1-1]} x^i r_{i \bmod k, \lfloor i/k \rfloor}.$$

Let $s = x^{-t} \cdot \Pi(r_0, \dots, r_{k-1})$ and write $s = \sum_{i \in [0,d_1-1]} s_i$. Then we have

$$x^{-t} \cdot \Pi(r_0, \dots, r_{k-1}) = \sum_{i \in [0,d_1-1]} x^{i-t} r_{i \bmod k, \lfloor i/k \rfloor} = \sum_{i \in [0,d_1-1]} x^i s_i.$$

In particular, this means $s_i = r_{i+t \bmod k, \lfloor (i+t)/k \rfloor}$. Since $t \in [0, k-1]$, we have

$$\kappa^{-1}(s) = \sum_{j \in [0,d_2-1]} x^j s_{kj} = \sum_{j \in [0,d_2-1]} x^j r_{t,j} = r_t. \qquad \square$$

Repacking. Finally, to analyze the repacking step in the batch version of Respire (Construction 3.3 and Appendix D.1), we will rely on the following lemma:

Lemma A.14 (Repacking). *Suppose $d_1 \geq d_2 \geq d_3$ are powers of two, and let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1}+1)$, $R_{d_2} = \mathbb{Z}[x]/(x^{d_2}+1)$, and $R_{d_3} = \mathbb{Z}[x]/(x^{d_3}+1)$. Then for any $r_0, \ldots, r_{d_2/d_3-1} \in R_{d_3}$,*

$$\sum_{i \in [0, d_2/d_3-1]} x^{i \cdot (d_1/d_2)} \kappa_{d_1,d_3}(r_i) = \kappa_{d_1,d_2}\big(\Pi(r_0, \ldots, r_{d_2/d_3-1})\big),$$

where $\kappa_{d_1,d_3} \colon R_{d_3} \to R_{d_1}$ and $\kappa_{d_1,d_2} \colon R_{d_2} \to R_{d_1}$ are the subring embedding functions (Definition A.9) and $\Pi \colon R_{d_3}^{d_2/d_3} \to R_{d_2}$ is the ring packing function (Definition A.12).

Proof. Let $\kappa_{d_2,d_3} \colon R_{d_3} \to R_{d_2}$ be the subring embedding function from R_{d_3} to R_{d_2}. By construction, for all $r = \sum_{i \in [0,d_3-1]} r_i x^i \in R_{d_3}$, it follows that

$$\kappa_{d_1,d_3}(r) = \sum_{i \in [0,d_3-1]} r_i x^{i \cdot (d_1/d_3)} = \sum_{i \in [0,d_3-1]} r_i x^{i \cdot (d_1/d_2)(d_2/d_1)} = \kappa_{d_1,d_2}(\kappa_{d_2,d_3}(r)). \tag{A.7}$$

Now, take any $r_0, \ldots, r_{d_2/d_3-1} \in R_{d_3}$. Then,

$$\begin{aligned}
\sum_{i \in [0,d_2/d_3-1]} x^{i \cdot (d_1/d_2)} \kappa_{d_1,d_3}(r_i) &= \sum_{i \in [0,d_2/d_3-1]} \kappa_{d_1,d_2}(x^i) \cdot \kappa_{d_1,d_2}(\kappa_{d_2,d_3}(r_i)) && \text{by Eq. (A.7)} \\
&= \kappa_{d_1,d_2}\left(\sum_{i \in [0,d_2/d_3-1]} x^i \cdot \kappa_{d_2,d_3}(r_i) \right) && \text{by Lemma A.10} \\
&= \kappa_{d_1,d_2}\big(\Pi(r_0, \ldots, r_{d_2/d_3-1})\big). && \text{by Definition A.12.}
\end{aligned}$$
□

B Query Compression

In this section, we give the formal description of the query packing algorithm used in Respire (i.e., the algorithms in Box 2). The approach we take is a combination of the corresponding procedures in [ACLS18, CCR19] (for packing multiple scalar RLWE encodings into a single RLWE encoding) and in [CCR19, MW22a] (for packing GSW encodings into RLWE encodings).

Coefficient packing. We start by describing the coefficient packing procedure adapted from [ACLS18, CCR19]. As a high level, the coefficient packing procedure from [ACLS18, CCR19] takes an RLWE encoding of a polynomial $f = \sum_{i \in [0,d]} f_i x^i \in R_{d,q}$ and expands it into d RLWE encodings of its coefficients $f_0, \ldots, f_{d-1} \in \mathbb{Z}_q$. Our construction incorporates the dimension-reduction approach described in Section 3 to further reduce the query size. Specifically, if we only need to pack h values into a single RLWE encoding, we would embed the h values into the coefficients of a polynomial that lives in a subring of $R_{d,q}$. The size of the packed encoding then scales with h (rather than the ring dimension d). In exchange for the shorter queries, our approach increases the noise by a modest amount (the variance is higher by a factor of $(d/h)^2$) and a small increase in the size of the public parameters (i.e., we need to communicate $\log d$ key-switching matrices as opposed to $\log h$ key-switching matrices). Practically speaking, the public parameter size difference is not significant (at most 22% for the scenarios considered in Section 4). However, the additional noise growth does introduce some challenges to ensure correctness. To compensate, we have to choose a smaller plaintext modulus (e.g. $p = 16$ instead of the $p = 256$ used in the Spiral system [MW22a]), which in turn decreases throughput (for large databases). We now describe the approach.

Construction B.1 (Coefficient Packing). Let λ be a security parameter and $d_1 = d_1(\lambda), q = q(\lambda)$ be lattice parameters where $d_1 = 2^{\delta_1}$ is a power of two. We require that $q = 1 \bmod 2$. Let $R_{d_1} = \mathbb{Z}_q[x]/(x^{d_1}+1)$. Let $\chi = \chi(\lambda)$ be an error distribution over R_{d_1} and $z \in \mathbb{N}$ be a decomposition base. Let (AutomorphSetup, Automorph) be the algorithms from Construction A.4 with parameters (d_1, q, χ, z). The coefficient packing procedure consists of a tuple of algorithms (CoeffPackSetup, CoeffPack, CoeffUnpack) defined as follows:

- $\mathsf{CoeffPackSetup}(1^\lambda, \mathbf{s})$: On input the security parameter λ and the secret key $\mathbf{s} \in R_{d_1}^2$, the setup algorithm samples

$$\mathbf{W}_j \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}, \tau_{d/2^j+1})$$

for each $j \in [0, \delta_1 - 1]$. It outputs the coefficient packing parameters $\mathsf{pp}_{\mathsf{coeff}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\delta_1 - 1})$.

- $\mathsf{CoeffPack}(\mathbf{s}, (f_0, \ldots, f_{d_2-1}))$: On input the secret key $\mathbf{s} = [-s \mid 1]^\intercal \in R_{d_1}^2$ and a tuple of coefficients $f_0, \ldots, f_{d_2-1} \in \mathbb{Z}_q$ where $d_2 = 2^{\delta_2} \le d_1$ for some non-negative integer δ_2, the packing algorithm defines the following quantities:[6]

  - Let $R_{d_2} = \mathbb{Z}_q[x]/(x^{d_2} + 1)$ and $\nu = \delta_1 - \delta_2$. Let $f(x) = \sum_{i \in [0, d_2-1]} f_i x^i \in R_{d_2}$.
  - Let $\kappa \colon R_{d_2} \to R_{d_1}$ be the subring embedding and $\kappa^{-1} \colon R_{d_1} \to R_{d_2}$ be the dimension-reduction mapping from Definition A.9.

  The packing algorithm now samples $a \xleftarrow{\text{R}} R_{d_1}$ and $e \leftarrow \chi$. It computes the encoding

$$\mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix} = \begin{bmatrix} a \\ sa + e + \kappa(f) \end{bmatrix} \in R_{d_1}^2.$$

  The packing algorithm computes $c_2' = \kappa^{-1}(c_2) \in R_{d_2}$ and outputs (c_1, c_2'). Note that the component c_1 is random and can be compressed by deriving it from a PRG (and appealing to the random oracle heuristic).

- $\mathsf{CoeffUnpack}(\mathsf{pp}_{\mathsf{coeff}}, (c_1, c_2'))$: On input the public parameters $\mathsf{pp}_{\mathsf{coeff}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\delta_1-1})$ and a compressed encoding (c_1, c_2') where $d_2 \in \mathbb{N}$, $c_1 \in R_{d_1}$, and $c_2' \in R_{d_2}$, the unpacking algorithm proceeds as follows

  - Let $\nu = \delta_1 - \delta_2$, where $\delta_2 = \log d_2$. Initialize $\mathbf{c}_0^{(0)} = 2^{-\delta_1} \cdot [c_1 \mid \kappa(c_2')]^\intercal \in R_{d_1}^2$, where $\kappa \colon R_{d_2} \to R_{d_1}$ is the subring embedding.
  - Then, for each $i \in [\nu]$, compute $\mathbf{c}_0^{(i)} = \mathbf{c}_0^{(i-1)} + \mathsf{Automorph}\big(\mathbf{W}_{i-1}, \mathbf{c}_0^{(i-1)}, \tau_{d/2^{i-1}+1}\big)$.
  - For each $i \in [\nu+1, \delta_1]$ and each $j \in [0, 2^{i-\nu} - 1]$, let $\mathbf{c}_{i,j}' = \mathsf{Automorph}\big(\mathbf{W}_{i-1}, \mathbf{c}_j^{(i-1)}, \tau_{d/2^{i-1}+1}\big)$ and compute

$$\begin{aligned} \mathbf{c}_{2j}^{(i)} &= \mathbf{c}_j^{(i-1)} + \mathbf{c}_{i,j}' \\ \mathbf{c}_{2j+1}^{(i)} &= x^{-2^{i-1}} \cdot \big(\mathbf{c}_j^{(i-1)} - \mathbf{c}_{i,j}'\big). \end{aligned}$$

  For a bit-length i and an integer $j \in [0, 2^i - 1]$ with binary representation $b_i b_{i-1} \cdots b_1$, let $\mathrm{rev}_i(j) \in [0, 2^i - 1]$ be the bit-reversal function that outputs the integer whose binary representation is the string $b_1 b_2 \cdots b_i$ (i.e., the bits of j in reverse order). For each $i \in [0, d_2 - 1]$, let $\hat{\mathbf{c}}_i = \mathbf{c}^{(\delta_1)}_{\mathrm{rev}_{\delta_2}(i)}$. The unpacking algorithm outputs the encodings $\hat{\mathbf{c}}_0, \ldots, \hat{\mathbf{c}}_{d_2-1}$.

Theorem B.2 (Coefficient Packing). *Let λ be a security parameter and $d_1, \delta_1, q, \chi, z$ be the parameters in Construction A.7. Suppose χ is subgaussian with variance σ_χ^2. Let $\mathbf{s} = [-s \mid 1]^\intercal \in R_{d_1}^2$ be a secret key. Take any $d_2 \le d_1$ where $d_2 = 2^{\delta_2}$ for some non-negative integer δ_2 and any collection of coefficients $f_0, \ldots, f_{d_2-1} \in \mathbb{Z}_q$. Then, sample the following:*

- $\mathsf{pp}_{\mathsf{coeff}} \leftarrow \mathsf{CoeffPackSetup}(1^\lambda, \mathbf{s})$;
- $(c_1, c_2') \leftarrow \mathsf{CoeffPack}(\mathbf{s}, (f_0, \ldots, f_{d_2-1}))$;
- $(\hat{\mathbf{c}}_0, \ldots, \hat{\mathbf{c}}_{d_2-1}) \leftarrow \mathsf{CoeffUnpack}\big(\mathsf{pp}_{\mathsf{coeff}}, (c_1, c_2')\big)$.

Then, for all $j \in [0, d_2 - 1]$, $\mathbf{s}^\intercal \hat{\mathbf{c}}_j = f_j + \hat{e}_j$, where $\hat{e}_j$ is subgaussian with variance $\hat{\sigma}_j^2 = \sigma_\chi^2\big(1 + t d_1^3 z^2/12\big)$.

[6]Note that the assumption that d_2 is a power of two is not necessary (Remark B.3), and it is possible to generically dispense with this restriction by padding. However, assuming d_2 is a power of two simplifies the description and analysis considerably. We make this simplifying assumption here to streamline the exposition.

Proof. Let $\mathsf{pp}_{\mathsf{coeff}} \leftarrow \mathsf{CoeffPackSetup}(1^\lambda, \mathbf{s})$ and $(c_1, c_2') \leftarrow \mathsf{CoeffPack}(\mathbf{s}, (f_0, \ldots, f_{d_2} - 1))$. By definition, $\mathsf{pp}_{\mathsf{proj}} = (\mathbf{W}_0, \ldots, \mathbf{W}_{\ell-1})$ where $\mathbf{W}_j \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}, \tau_{d/2^j+1})$. Moreover, $c_2' = \kappa^{-1}(c_2) = \kappa^{-1}(sa + e + \kappa(f))$. Consider the encodings $\mathbf{c}_0^{(i)}$ for $i \in [0, \nu]$ computed by CoeffUnpack. We show inductively that these encodings satisfy

$$\mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(i)} = 2^{-\delta_1 + i} \pi_i(u) + e_i, \tag{B.1}$$

where $e_0 = 0$, e_i is subgaussian with variance $\sigma_i^2 = \sum_{k \in [i]} 4^{k-1} t d_1 z^2 \sigma_\chi^2 / 4$, π_i is the projection map from Eq. (3.6), and

$$u = -sa + \pi_\nu(sa + e + \kappa(f)). \tag{B.2}$$

We proceed by induction on i:

- Consider $i = 0$. By definition,

$$\begin{aligned} \mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(0)} = 2^{-\delta_1}(-sc_1 + \kappa(c_2')) &= 2^{-\delta_1}(-sa + \kappa(\kappa^{-1}(sa + e + \kappa(f)))) \\ &= 2^{-\delta_1}(-sa + \pi_\nu(sa + e + \kappa(f))) \qquad \text{by Lemma A.11} \\ &= 2^{-\delta_1} \pi_0(u) + e_0, \end{aligned}$$

  since $e_0 = 0$, $\pi_0(r) = r$ for all $r \in R_{d_1}$, and the definition of u from Eq. (B.2).

- For the inductive step, let $\mathbf{c}_i' = \mathsf{Automorph}(\mathbf{W}_i, \mathbf{c}_0^{(i)}, \tau_{d/2^i+1})$. By Theorem A.5 and the inductive hypothesis (Eq. (B.1)),

$$\mathbf{s}^{\mathsf{T}} \mathbf{c}_i' = \tau_{d/2^i+1}\big(\mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(i)}\big) + \tilde{e}_{i+1} = \tau_{d/2^i+1}\big(2^{-\delta_1+i} \pi_i(u) + e_i\big) + \tilde{e}_{i+1},$$

  where $\tilde{e}_{i+1}$ is subgaussian with variance $t d_1 z^2 \sigma_\chi^2 / 4$. Since $\mathbf{c}_{i+1} = \mathbf{c}_i + \mathbf{c}_i'$, and appealing to Lemma A.6, we have

$$\begin{aligned} \mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(i+1)} &= \mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(i)} + \mathbf{s}^{\mathsf{T}} \mathbf{c}_i' \\ &= 2^{-\delta_1+i}\big(\pi_i(u) + \tau_{d/2^i+1}(\pi_i(u))\big) + e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1} \\ &= 2^{-\delta_1+i+1} \pi_{i+1}(u) + e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}. \end{aligned}$$

  Let $e_{i+1} \coloneqq e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}$. Since e_i is subgaussian with variance σ_i^2 and appealing to the independence heuristic (to argue that the key-switching error $\tilde{e}_{i+1}$ is independent of e_i), we have that e_{i+1} is subgaussian with variance

$$\sigma_{i+1}^2 = 4\sigma_i^2 + t d_1 z^2 \sigma_\chi^2 / 4 = 4 \cdot \sum_{k \in [i-1]} 4^{k-1} t d_1 z^2 \sigma_\chi^2 / 4 + t d_1 z^2 \sigma_\chi^2 / 4 = \sum_{k \in [i]} 4^{k-1} t d_1 z^2 \sigma_\chi^2 / 4. \tag{B.3}$$

Thus, by induction on i, we conclude that $\mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(\nu)} = 2^{-\delta_1+\nu} \pi_\nu(u) + e_\nu$. Using the definition of u from Eq. (B.2) and Lemma A.11, we have

$$\pi_\nu(u) = \pi_\nu(-sa) + \pi_\nu(\pi_\nu(sa + e + \kappa(f))) = \pi_\nu(\kappa(f) + e).$$

This means that

$$\mathbf{s}^{\mathsf{T}} \mathbf{c}_0^{(\nu)} = 2^{-\delta_1+\nu} \pi_\nu\big(\kappa(f) + e\big) + e_\nu. \tag{B.4}$$

For each $i \in [\nu, \delta_1]$ and $j \in [0, 2^{i-\nu} - 1]$, define

$$w_{i,j} \coloneqq x^{-2^\nu \cdot \mathrm{rev}_{i-\nu}(j)} \cdot \big(\kappa(f) + e\big) \tag{B.5}$$

We now show that for each $i \in [\nu, \delta_1]$ and $j \in [0, 2^{i-\nu} - 1]$,

$$\mathbf{s}^{\mathsf{T}} \mathbf{c}_j^{(i)} = 2^{-\delta_1+i} \pi_i(w_{i,j}) + e_i, \tag{B.6}$$

where e_i is subgaussian with variance $\sigma_i^2 = \sum_{k \in [i]} 4^{k-1} t d_1 z^2 \sigma_\chi^2 / 4$. Again, we proceed by induction on i. The base case where $i = \nu$ (and $j = 0$) follows from Eq. (B.4). Consider the inductive step. Take any $j \in [0, 2^{i-\nu} - 1]$ and let $\mathbf{c}_{i,j}' = \mathsf{Automorph}\big(\mathbf{W}_i, \mathbf{c}_j^{(i)}, \tau_{d/2^i+1}\big)$. By Theorem A.5 and the inductive hypothesis (Eq. (B.6)),

$$\mathbf{s}^{\mathsf{T}} \mathbf{c}_{i,j}' = \tau_{d/2^i+1}\big(\mathbf{s}^{\mathsf{T}} \mathbf{c}_j^{(i)}\big) + \tilde{e}_{i+1} = \tau_{d/2^i+1}\big(2^{-\delta_1+i} \pi_i(w_{i,j}) + e_i\big) + \tilde{e}_{i+1},$$

where $\tilde{e}_{i+1}$ is subgaussian with variance $t d_1 z^2 \sigma_\chi^2 / 4$. We now consider the encodings $\mathbf{c}_{2j}^{(i+1)}$ and $\mathbf{c}_{2j+1}^{(i+1)}$:

- First, consider $\mathbf{c}_{2j}^{(i+1)}$. By definition of the bit-reversal function, we have that $\mathrm{rev}_{i-\nu}(j) = \mathrm{rev}_{i+1-\nu}(2j)$. In particular, this means that

  $$w_{i+1,2j} = x^{-2^\nu \cdot \mathrm{rev}_{i+1-\nu}(2j)} \cdot \big(\kappa(f) + e\big) = x^{-2^\nu \cdot \mathrm{rev}_{i-\nu}(j)} \cdot \big(\kappa(f) + e\big) = w_{i,j}.$$

  Now we can write

  $$\begin{aligned} \mathbf{s}^\mathsf{T}\mathbf{c}_{2j}^{(i+1)} &= \mathbf{s}^\mathsf{T}\mathbf{c}_j^{(i)} + \mathbf{s}^\mathsf{T}\mathbf{c}'_{i,j} \\ &= 2^{-\delta_1+i}\big(\pi_i(w_{i,j}) + \tau_{d/2^i+1}(\pi_i(w_{i,j}))\big) + e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1} \\ &= 2^{-\delta_1+i+1}\pi_{i+1}(w_{i,j}) + e_{i+1} \\ &= 2^{-\delta_1+i+1}\pi_{i+1}(w_{i+1,2j}) + e_{i+1}. \end{aligned}$$

  where $e_{i+1} = e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}$. Under the independence heuristic (applied to e_i and $\tilde{e}_{i+1}$), the variance of e_{i+1} satisfies the desired relation via the same calculation as Eq. (B.3).

- Next, consider $\mathbf{c}_{2j+1}^{(i+1)}$. In this case, by definition of the bit-reversal function, $\mathrm{rev}_{i+1-\nu}(2j+1) = 2^{i-\nu} + \mathrm{rev}_{i-\nu}(j)$. This means

  $$w_{i+1,2j+1} = x^{-2^\nu \cdot \mathrm{rev}_{i+1-\nu}(2j+1)} \cdot \big(\kappa(f) + e\big) = x^{-2^i} w_{i,j}.$$

  Now we can write

  $$\begin{aligned} \mathbf{s}^\mathsf{T}\mathbf{c}_{2j+1}^{(i+1)} &= x^{-2^i}\big(\mathbf{s}^\mathsf{T}\mathbf{c}_j^{(i)} - \mathbf{s}^\mathsf{T}\mathbf{c}'_{i,j}\big) \\ &= x^{-2^i}\big(2^{-\delta_1+i}\big(\pi_i(w_{i,j}) - \tau_{d/2^i+1}(\pi_i(w_{i,j}))\big) + e_i - \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}\big) \\ &= 2^{-\delta_1+i+1}\pi_{i+1}\big(w_{i,j} \cdot x^{-2^i}\big) + e_{i+1} \\ &= 2^{-\delta_1+i+1}\pi_{i+1}(w_{i+1,2j+1}) + e_{i+1}, \end{aligned}$$

  where $e_{i+1} = x^{-2^i}\big(e_i + \tau_{d/2^i+1}(e_i) + \tilde{e}_{i+1}\big)$. Under the independence heuristic (applied to e_i and $\tilde{e}_{i+1}$), the variance of e_{i+1} satisfies the desired relation via the same calculation as Eq. (B.3). Note that multiplying a polynomial by x^{-2^i} corresponds to applying a (nega)-cyclic rotation to the coefficients of the polynomial and does *not* change the magnitude of any of the coefficients.

By induction on i, Eq. (B.6) holds for $i = \delta_1$ and all $j \in [0, d_2 - 1]$. Thus, for all $j \in [0, d_2 - 1]$, we have that

$$\mathbf{s}^\mathsf{T}\mathbf{c}_j^{(\delta_1)} = \pi_{\delta_1}(w_{\delta_1,j}) + e_{\delta_1}.$$

By Eq. (B.5) and the fact that $\delta_1 - \nu = \delta_2$, we have

$$\pi_{\delta_1}(w_{\delta_1,j}) = \pi_{\delta_1}\big(x^{-2^\nu \cdot \mathrm{rev}_{\delta_1-\nu}(j)} \cdot \big(\kappa(f) + e\big)\big) = f_{\mathrm{rev}_{\delta_2}(j)} + \pi_{\delta_1}\big(x^{-2^\nu \cdot \mathrm{rev}_{\delta_2}(j)} \cdot e\big).$$

Since $\mathrm{rev}_{\delta_2}(\mathrm{rev}_{\delta_2}(j)) = j$, we have

$$\mathbf{s}^\mathsf{T}\hat{\mathbf{c}}_j = \mathbf{s}^\mathsf{T}\mathbf{c}_{\mathrm{rev}_{\delta_2}(j)}^{(\delta_1)} = f_j + \pi_{\delta_1}\big(x^{-2^\nu \cdot j} \cdot e\big) + e_{\delta_1} = f_j + \hat{e}_j,$$

where $\hat{e}_j = \pi_{\delta_1}\big(x^{-2^\nu \cdot j} \cdot e\big) + e_{\delta_1}$. Since e is sampled independently of e_{δ_1}, we conclude that $\hat{e}_j$ is subgaussian with variance

$$\hat{\sigma}_j^2 = \sigma_\chi^2 + \sum_{k \in [\delta_1]} 4^{k-1} t d_1 z^2 \sigma_\chi^2 / 4 = \sigma_\chi^2 + 4^{\delta_1} t d_1 z^2 \sigma_\chi^2 / 12 = \sigma_\chi^2 \big(1 + t d_1^3 z^2 / 12\big). \qquad \square$$

Remark B.3 (Packing an Arbitrary Number of Coefficients)**.** As defined, Construction B.1 assumes that the packing algorithm CoeffPack takes d_2 coefficients $(f_0, \ldots, f_{d_2-1})$, where d_2 is a power of two. It is straightforward to generalize CoeffPack to take in an arbitrary number of coefficients. One approach is to simply pad the input to the nearest power of two, which incurs at most a 2× overhead. In fact, it is possible to avoid padding altogether by having CoeffPack embed the coefficients $(f_0, \ldots, f_{d_2-1})$ into the polynomial f in bit-reversed order (rather than deferring the bit-reversal to the very end). With this optimization, we avoid the need to expand "unused" coefficients. This is the approach we take in our implementation. In the subsequent description, we will assume that the coefficient-packing algorithm CoeffPack can take an arbitrary number of inputs (up to the ring dimension d_1).

Packing GSW encodings. The second ingredient we use is the approach for packing GSW encodings into a small number of RLWE encodings (which can in turn be further packed using the coefficient packing approach described above). Here, we recall the approach from [CCR19, MW22a]:

Construction B.4 (RLWE-to-GSW [CCR19, MW22a, adapted]). Let λ be a security parameter and $d = d(\lambda)$, $q = q(\lambda)$ be lattice parameters where d is a power of two. Let $R_d = \mathbb{Z}[x]/(x^d + 1)$ and $\chi = \chi(\lambda)$ be an error distribution over R_d. The GSW packing algorithm is parameterized by two decomposition bases: a conversion base $z_{\text{conv}} \in \mathbb{N}$ and the decomposition base $z_{\text{GSW}} \in \mathbb{N}$ for the resulting GSW encodings. Let $t_{\text{GSW}} = \lfloor \log_{z_{\text{GSW}}} q \rfloor + 1$ and $t_{\text{conv}} = \lfloor \log_{z_{\text{conv}}} q \rfloor + 1$. We define the algorithms $(\mathsf{RLWEToGSWSetup}, \mathsf{RLWEToGSW})$ as follows:

- $\mathsf{RLWEToGSWSetup}(1^\lambda, \mathbf{s})$: On input the security parameter λ and the secret key $\mathbf{s} = [-s \mid 1]^{\mathsf{T}} \in R_{d,q}^2$, the setup algorithm samples $\mathbf{a} \xleftarrow{\text{R}} R_{d,q}^{2t_{\text{conv}}}$ and $\mathbf{e} \leftarrow \chi^{2t_{\text{conv}}}$ and output the conversion parameters
$$\mathsf{pp}_{\text{conv}} = \mathbf{V} = \begin{bmatrix} \mathbf{a}^{\mathsf{T}} \\ s\mathbf{a}^{\mathsf{T}} + \mathbf{e}^{\mathsf{T}} - s(\mathbf{s}^{\mathsf{T}} \otimes \mathbf{g}_{z_{\text{conv}}}^{\mathsf{T}}) \end{bmatrix} \in R_{d,q}^{2 \times 2t_{\text{conv}}}.$$

- $\mathsf{RLWEToGSW}(\mathsf{pp}_{\text{conv}}, (\mathbf{c}_1, \ldots, \mathbf{c}_{t_{\text{GSW}}}))$: On input the conversion parameters $\mathsf{pp}_{\text{conv}} = \mathbf{V} \in R_{d,q}^{2 \times 2t_{\text{conv}}}$, and RLWE encodings $\mathbf{c}_1, \ldots, \mathbf{c}_{t_{\text{GSW}}} \in R_{d,q}^2$, let $\hat{\mathbf{c}} = [\mathbf{c}_1 \mid \cdots \mid \mathbf{c}_{t_{\text{GSW}}}] \in R_{d,q}^{2 \times t_{\text{GSW}}}$ and output the GSW encoding
$$\mathbf{C} = [\mathbf{V}\mathbf{g}_{z_{\text{conv}}}^{-1}(\hat{\mathbf{c}}) \mid \hat{\mathbf{c}}] \in R_{d,q}^{2 \times 2t_{\text{GSW}}}.$$

Theorem B.5 (RLWE-to-GSW [CCR19, MW22a, adapted]). *Let λ be a security parameter and $d, q, \chi, z_{\text{conv}}, z_{\text{GSW}}$ be the parameters from Construction B.4. Suppose χ is subgaussian with variance σ_χ^2. Let $\mathbf{s} = [-s, 1]^{\mathsf{T}}$ be a secret key. For each $i \in [t_{\text{GSW}}]$, let $\mathbf{c}_i \in R_{d,q}^2$ be an encoding of μz_{GSW}^{i-1} with error e_i (i.e., $\mathbf{s}^{\mathsf{T}}\mathbf{c}_i = \mu z_{\text{GSW}}^{i-1} + e_i$). Suppose each e_i is subgaussian with variance σ_e^2. Then, sample $\mathsf{pp}_{\text{conv}} \leftarrow \mathsf{RLWEToGSWSetup}(1^\lambda, \mathbf{s})$ and $\mathbf{C} \leftarrow \mathsf{RLWEToGSW}(\mathsf{pp}_{\text{conv}}, (\mathbf{c}_1, \ldots, \mathbf{c}_{t_{\text{GSW}}}))$. Then $\mathbf{C}$ is a GSW encoding of μ with respect to $\mathbf{s}$ with error $\mathbf{e}$ (i.e., $\mathbf{s}^{\mathsf{T}}\mathbf{C} = \mathbf{s}^{\mathsf{T}}\mathbf{G}_{2,z_{\text{GSW}}} + \mathbf{e}^{\mathsf{T}}$) where the components of $\mathbf{e}$ are subgaussian with variance $d\sigma_e^2 \|\mathbf{s}\|_\infty^2 + t_{\text{conv}} d z_{\text{conv}}^2 \sigma_\chi^2 / 2$.*

Query packing. The query packing algorithm in Respire is obtained by composing the coefficient expansion algorithm with the RLWE-to-GSW conversion algorithms. To have finer control over the noise, we allow for *two* separate bases for coefficient packing: one for packing/expanding the RLWE ciphertexts, and one for packing/expanding the GSW ciphertexts. We give the full description of the algorithms from Box 2 below:

Construction B.6 (Query Packing). Let λ be a security parameter and $d_1 = d_1(\lambda)$, $q = q(\lambda)$ be lattice parameters where $d_1 = 2^{\delta_1}$ is a power of two. We require that $q = 1 \bmod 2$. Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1} + 1)$. Let χ be an error distribution over R_{d_1}. The main query packing algorithm uses the coefficient packing and RLWE-to-GSW conversion algorithms from Constructions B.1 and B.4. The scheme is additionally parameterized by four decomposition bases: $z_{\text{coeff,RLWE}}$ for expanding the RLWE encodings, $z_{\text{coeff,GSW}}$ for expanding the GSW encodings, z_{conv} for the RLWE-to-GSW conversion, and z_{GSW} for the GSW decomposition base.

- Let $(\mathsf{CoeffPackSetup}_{\text{RLWE}}, \mathsf{CoeffPack}_{\text{RLWE}}, \mathsf{CoeffUnpack}_{\text{RLWE}})$ be the coefficient packing algorithms from Construction B.1 instantiated with parameters $(d_1, q, \chi, z_{\text{coeff,RLWE}})$.
- Let $(\mathsf{CoeffPackSetup}_{\text{GSW}}, \mathsf{CoeffPack}_{\text{GSW}}, \mathsf{CoeffUnpack}_{\text{GSW}})$ be the coefficient packing algorithms from Construction B.1 instantiated with parameters $(d_1, q, \chi, z_{\text{coeff,GSW}})$.
- Let $(\mathsf{RLWEToGSWSetup}, \mathsf{RLWEToGSW})$ be the RLWE-to-GSW conversion algorithms from Construction B.4 instantiated with parameters $(d_1, q, \chi, z_{\text{conv}}, z_{\text{GSW}})$. Let $t_{\text{GSW}} = \lfloor \log_{z_{\text{GSW}}} q \rfloor + 1$.

We now define the algorithms $(\mathsf{QueryPackSetup}, \mathsf{QueryPack}, \mathsf{QueryUnpack})$:

- $\mathsf{QueryPackSetup}(1^\lambda, \mathbf{s})$: On input a security parameter λ and the secret key $\mathbf{s} \in R_{d,q}^2$, the setup algorithm samples

  - $\mathsf{pp}_{\mathsf{coeff,RLWE}} \leftarrow \mathsf{CoeffPackSetup}_{\mathsf{RLWE}}(1^\lambda, \mathbf{s})$;
  - $\mathsf{pp}_{\mathsf{coeff,GSW}} \leftarrow \mathsf{CoeffPackSetup}_{\mathsf{GSW}}(1^\lambda, \mathbf{s})$; and
  - $\mathsf{pp}_{\mathsf{conv}} \leftarrow \mathsf{RLWEToGSWSetup}(1^\lambda, \mathbf{s})$.

  It outputs the query packing parameters $\mathsf{pp}_{\mathsf{qpk}} = (\mathsf{pp}_{\mathsf{coeff,RLWE}}, \mathsf{pp}_{\mathsf{coeff,GSW}}, \mathsf{pp}_{\mathsf{conv}})$.

- $\mathsf{QueryPack}(\mathbf{s}, \mathbf{v}, \boldsymbol{\mu})$: On input the secret key $\mathbf{s} \in R_{d,q}^2$, a collection of values $\mathbf{v} = (v_1, \dots, v_k) \in \mathbb{Z}_q^k$ where $k \le d_1$, and $\boldsymbol{\mu} = (\mu_1, \dots, \mu_\ell) \in \{0,1\}^\ell$ where $\ell t_{\mathsf{GSW}} \le d_1$, the query packing algorithm computes

$$\mathsf{enc}_{\mathsf{RLWE}} \leftarrow \mathsf{CoeffPack}_{\mathsf{RLWE}}(\mathbf{s}, (v_1, \dots, v_k))$$
$$\mathsf{enc}_{\mathsf{GSW}} \leftarrow \mathsf{CoeffPack}_{\mathsf{GSW}}(\mathbf{s}, (\mu_1, \mu_1 z_{\mathsf{GSW}}, \dots, \mu_1 z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1}, \dots, \mu_\ell, \mu_\ell z_{\mathsf{GSW}}, \dots, \mu_\ell z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1})).$$

  It outputs the packed encoding $\mathsf{enc} = (\mathsf{enc}_{\mathsf{RLWE}}, \mathsf{enc}_{\mathsf{GSW}})$.

- $\mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{enc})$: On input the packing key $\mathsf{pp}_{\mathsf{qpk}} = (\mathsf{pp}_{\mathsf{coeff,RLWE}}, \mathsf{pp}_{\mathsf{coeff,GSW}}, \mathsf{pp}_{\mathsf{conv}})$ and the packed encoding $\mathsf{enc} = (\mathsf{enc}_{\mathsf{RLWE}}, \mathsf{enc}_{\mathsf{GSW}})$, the unpacking algorithm computes

$$(\mathbf{c}_1, \dots, \mathbf{c}_k) \leftarrow \mathsf{CoeffUnpack}_{\mathsf{RLWE}}(\mathsf{pp}_{\mathsf{coeff,RLWE}}, \mathsf{enc}_{\mathsf{RLWE}})$$
$$(\hat{\mathbf{c}}_1, \dots, \hat{\mathbf{c}}_{\ell z_{\mathsf{GSW}}}) \leftarrow \mathsf{CoeffUnpack}_{\mathsf{GSW}}(\mathsf{pp}_{\mathsf{coeff,GSW}}, \mathsf{enc}_{\mathsf{GSW}}).$$

  Then, for each $i \in [\ell]$, it computes $\mathbf{C}_i \leftarrow \mathsf{RLWEToGSW}(\mathsf{pp}_{\mathsf{conv}}, (\hat{\mathbf{c}}_{(i-1)\cdot z_{\mathsf{GSW}}+1}, \dots, \hat{\mathbf{c}}_{i \cdot z_{\mathsf{GSW}}}))$. It outputs the RLWE encodings $(\mathbf{c}_1, \dots, \mathbf{c}_k)$ together with the GSW encodings $(\mathbf{C}_1, \dots, \mathbf{C}_\ell)$.

Theorem B.7 (Query Packing). *Let $\mathbf{s} \in R_{d,q}^2$ be a secret key. Let $z_{\mathsf{coeff,RLWE}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}} \in \mathbb{N}$ be the decomposition bases from Construction B.6. Let*

$$t_{\mathsf{coeff,RLWE}} = \lfloor \log_{z_{\mathsf{coeff,RLWE}}} q \rfloor + 1 \quad , \quad t_{\mathsf{coeff,GSW}} = \lfloor \log_{z_{\mathsf{coeff,GSW}}} q \rfloor + 1 \quad , \quad t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1 \quad , \quad t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1$$

be the corresponding lengths. Suppose the error distribution χ is subgaussian with variance σ_χ^2. Take any vector $\mathbf{v} \in \mathbb{Z}_q^k$ and $\boldsymbol{\mu} \in \{0,1\}^\ell$. Suppose $\mathsf{pp}_{\mathsf{qpk}} \leftarrow \mathsf{QueryPackSetup}(s)$, $\mathsf{enc} \leftarrow \mathsf{QueryPack}(s, \mathbf{v}, \boldsymbol{\mu})$, and $((\mathbf{c}_1, \dots, \mathbf{c}_k), (\mathbf{C}_1, \dots, \mathbf{C}_\ell)) \leftarrow \mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{enc})$. Then, the following hold:

- *For all $i \in [k]$, $\mathbf{c}_i$ is an RLWE encoding of v_i with respect to secret key s and error e_i.*
- *For all $j \in [\ell]$, $\mathbf{C}_j$ is a GSW encoding of μ_j with respect to secret key s and error $\mathbf{e}_j$.*

Under the independence heuristic, the errors $e_1, \dots, e_k$ are subgaussian with variance $\sigma_1^2 = \sigma_\chi^2(1 + t_{\mathsf{coeff,RLWE}} d_1^3 z_{\mathsf{coeff,RLWE}}^2/12)$, and the components of $\mathbf{e}_1, \dots, \mathbf{e}_\ell$ are subgaussian with variance $\sigma_2^2 = \sigma_\chi^2(d_1 \|s\|_\infty^2 (1 + t_{\mathsf{coeff,GSW}} d_1^3 z_{\mathsf{coeff,GSW}}^2/12) + t_{\mathsf{conv}} d_1 z_{\mathsf{conv}}^2/2)$.

Proof. Let $(\mathbf{c}_1, \dots, \mathbf{c}_k)$ and $(\hat{\mathbf{c}}_1, \dots, \hat{\mathbf{c}}_{\ell z_{\mathsf{GSW}}})$ be the results of CoeffUnpack in running QueryUnpack. By Theorem B.2, the following hold:

- For each $i \in [k]$, $\mathbf{c}_i$ is an RLWE encoding of v_i with error e_i where e_i is subgaussian with variance $\sigma_\chi^2(1 + t_{\mathsf{coeff,RLWE}} d_1^3 z_{\mathsf{coeff,RLWE}}^2/12)$.
- For each $i \in [\ell t_{\mathsf{GSW}}]$, the tuple $(\hat{\mathbf{c}}_{(i-1)\cdot z_{\mathsf{GSW}}+1}, \dots, \hat{\mathbf{c}}_{i \cdot z_{\mathsf{GSW}}})$ is an RLWE encoding of $(\mu_i \cdot z_{\mathsf{GSW}}^0, \dots, \mu_i \cdot z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1})$, where the error in each encoding is subgaussian with variance $\sigma_\chi^2(1 + t_{\mathsf{coeff,GSW}} d_1^3 z_{\mathsf{coeff,GSW}}^2/12)$.

By Theorem B.5, we know that for each $i \in [\ell]$, $\mathbf{C}_i$ is a GSW encoding of μ_i with error $\mathbf{e}_i$ and the components of $\mathbf{e}_i$ are subgaussian with variance $\sigma_\chi^2(d_1 \|\mathbf{s}\|_\infty^2 (1 + t_{\mathsf{coeff,GSW}} d_1^3 z_{\mathsf{coeff,GSW}}^2/12) + t_{\mathsf{conv}} d_1 z_{\mathsf{conv}}^2/2)$. □

Remark B.8 (Different Decomposition Bases for Query Unpacking). The QueryUnpack algorithm in Construction B.6 uses two different decomposition bases $z_{\mathsf{coeff,RLWE}}$ and $z_{\mathsf{coeff,GSW}}$ to expand the packed encoding. We use two different decomposition bases because the RLWE-to-GSW conversion procedure (Construction B.4) introduces additional noise. It is advantageous to use a smaller decomposition base when expanding the RLWE encodings that will be assembled into GSW encodings. This way, the noise in the resulting RLWE encodings and the GSW encodings output by QueryUnpack will be on a comparable footing. We illustrate our parameter choices in Table 5.

C Response Compression

In this section, we provide the full details of the (homomorphic) dimension reduction algorithm in Respire (i.e., the algorithms in Box 1). As described in Section 3.2, we compose dimension reduction with vectorization (which is helpful in the batched setting). We recall vectorization (i.e., the algorithms from Box 4) in Appendix C.1 and then give our response compression approach (i.e., the algorithms from Box 1) in Appendix C.2.

C.1 Vectorizing RLWE Encodings

In this section, we describe the approach from [MW22a] for packing multiple scalar RLWE encodings into a single vector RLWE encoding, which we call vectorization. These correspond to the algorithms in Box 4. While [MW22a] shows how to pack scalar encodings into a matrix RLWE encoding, we only consider the case where the target is a *vector* RLWE encoding, since vector encodings yield the best compression. Vectorization yields shorter packed ciphertexts (i.e., achieves higher rate), but requires larger public parameters (proportional to the vector length). Concretely, vectorization packs $2n$ ring elements into $n+1$ ring elements, thus achieving a $\approx 2\times$ reduction in encoding size. Now, we present the (adapted) construction from [MW22a] for vectorizing a collection of scalar RLWE encodings into a vector RLWE encoding (i.e., the algorithms in Box 4) and then state the associated correctness guarantee:

Construction C.1 (Vectorizing RLWE Encodings [MW22a, adapted])**.** Let λ be a security parameter and $d = d(\lambda), q = q(\lambda)$ be lattice parameters where d is a power of two. Let $R_d = \mathbb{Z}[x]/(x^d + 1)$ and $\chi = \chi(\lambda)$ be an error distribution over R_d. The construction is also parameterized by a decomposition base z. Let $t = \lfloor \log_z q \rfloor + 1$. We define the vectorization algorithms $(\mathsf{VecSetup}, \mathsf{Vectorize})$ as follows:

- $\mathsf{VecSetup}(1^\lambda, \mathbf{s}_1, \mathbf{S}_2)$: On input a security parameter λ and two secret keys $\mathbf{s}_1 = [-s_1 \mid 1]^{\mathsf{T}} \in R_{d,q}^2$ and $\mathbf{S}_2 = [-\mathbf{s}_2 \mid \mathbf{I}_n]^{\mathsf{T}} \in R_{d,q}^{(n+1)\times n}$, the setup algorithm samples $\mathbf{a}_i \xleftarrow{\mathrm{R}} R_{d,q}^t$, $\mathbf{E}_i \leftarrow \chi^{n\times t}$, and sets
$$\mathbf{V}_i = \begin{bmatrix} \mathbf{a}_i^{\mathsf{T}} \\ \mathbf{s}_2\mathbf{a}_i^{\mathsf{T}} + \mathbf{E}_i - s_1\mathbf{u}_i\mathbf{g}_z^{\mathsf{T}} \end{bmatrix} \in R_{d,q}^{(n+1)\times t}$$
for each $i \in [n]$ and where $\mathbf{u}_i \in R_{d,q}^n$ denotes the i^{th} canonical basis vector. Finally, the algorithm outputs the parameters $\mathsf{pp}_{\mathsf{vec}} = (\mathbf{V}_1, \ldots, \mathbf{V}_n)$.
- $\mathsf{Vectorize}(\mathsf{pp}_{\mathsf{vec}}, (\mathbf{c}_1, \ldots, \mathbf{c}_n))$: On input the parameters $\mathsf{pp}_{\mathsf{vec}} = (\mathbf{V}_1, \ldots \mathbf{V}_n)$ and a collection of RLWE encodings $\mathbf{c}_1, \ldots, \mathbf{c}_n$ where $\mathbf{c}_i^{\mathsf{T}} = [c_{i,0} \mid c_{i,1}]$, compute and output
$$\mathbf{c} = \sum_{i\in[n]} \left(\mathbf{V}_i \mathbf{g}_z^{-1}(c_{i,0}) + \begin{bmatrix} 0 \\ c_{i,1}\mathbf{u}_i \end{bmatrix} \right) \in R_{d,q}^{n+1}.$$

Theorem C.2 (Vectorizing RLWE Encodings [MW22a, adapted])**.** *Let λ be a parameter and d, q, χ, z be the parameters in Construction C.1. Let $\mathbf{s}_1 = [-s \mid 1]^{\mathsf{T}} \in R_{d,q}^2$ and $\mathbf{S}_2 = [-\mathbf{s} \mid \mathbf{I}_n]^{\mathsf{T}} \in R_{d,q}^{(n+1)\times n}$ be secret keys. Suppose χ is subgaussian with variance σ_χ^2. Take any collection of encodings $\mathbf{c}_1, \ldots, \mathbf{c}_n \in R_{d,q}^2$. Let $\mathsf{pp}_{\mathsf{vec}} \leftarrow \mathsf{VecSetup}(1^\lambda, \mathbf{s}_1, \mathbf{S}_2)$ and $\mathbf{c}' \leftarrow \mathsf{Vectorize}(\mathsf{pp}_{\mathsf{vec}}, (\mathbf{c}_1, \ldots, \mathbf{c}_n))$. Then*
$$\mathbf{S}_2^{\mathsf{T}}\mathbf{c}' = \begin{bmatrix} \mathbf{s}_1^{\mathsf{T}}\mathbf{c}_1 \\ \vdots \\ \mathbf{s}_1^{\mathsf{T}}\mathbf{c}_n \end{bmatrix} + \mathbf{e}',$$
where the components of $\mathbf{e}'$ are subgaussian with variance $(\sigma')^2 \le ntdz^2\sigma_\chi^2/4$.

C.2 Response Compression

We now provide the full details of the response compression scheme in Respire (i.e., the algorithms in Box 1). As outlined in Section 3.2, our approach combines (split) modulus switching [BGV12, MW22a] with ring switching [BGV12, GHPS12].

Construction C.3 (Response Compression)**.** Let $d_1 \geq d_2$ be ring dimensions, and let $k = d_1/d_2$. Let $R_{d_1} = \mathbb{Z}[x]/(x^{d_1}+1)$ and $R_{d_2} = \mathbb{Z}[x]/(x^{d_2}+1)$. Let $\Pi \colon R_{d_2}^k \to R_{d_1}$ be the ring packing function (defined in Eq. (3.3) and Definition A.12). Let $\kappa \colon R_{d_2} \to R_{d_1}$ be the subring embedding and $\kappa^{-1} \colon R_{d_1} \to R_{d_2}$ be the dimension-reduction mappings (defined in Definition A.9). Let χ be an error distribution over R_{d_2}, $q_1 \geq q_2 \geq q_3$ be ring moduli, $z \in \mathbb{N}$ be a decomposition base, n be the input dimension, and $t = \lfloor \log_z q_2 \rfloor + 1$.

- $\mathsf{CompressSetup}(1^\lambda, \mathbf{S}_1, \mathbf{S}_2)$: On input a source key $\mathbf{S}_1 = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n] \in R_{d_1}^{n \times (n+1)}$ and a target key $\mathbf{S}_2 = [-\tilde{\mathbf{s}}_2 \mid \mathbf{I}_n] \in R_{d_2,q_2}^{n\times(n+1)}$, sample $\mathbf{a}_1, \ldots, \mathbf{a}_k \xleftarrow{\text{R}} R_{d_2,q_2}^t$ and $\mathbf{E}_1, \ldots, \mathbf{E}_k \leftarrow \chi^{n \times t}$. Note that we do *not* specify the modulus for $\mathbf{S}_1$. Let

$$\mathbf{a} = \Pi(\mathbf{a}_1, \ldots, \mathbf{a}_k) \in R_{d_1,q_2}^t \quad \text{and} \quad \mathbf{B} = \Pi(\tilde{\mathbf{s}}_2\mathbf{a}_1^{\mathsf{T}} + \mathbf{E}_1, \ldots, \tilde{\mathbf{s}}_2\mathbf{a}_k^{\mathsf{T}} + \mathbf{E}_k) \in R_{d_1,q_2}^{n\times t}.$$

  Output the key-switching matrix

$$\mathbf{W} = \begin{bmatrix} \mathbf{a}^{\mathsf{T}} \\ \mathbf{B} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times t} \\ -(\tilde{\mathbf{s}}_1 \bmod q_2)\cdot \mathbf{g}_z^{\mathsf{T}} \end{bmatrix} \in R_{d_1,q_2}^{(n+1)\times t}.$$

- $\mathsf{Compress}(\mathbf{W}, \mathbf{c})$: On input the key-switching matrix $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^{\mathsf{T}} \\ \mathbf{W}_2 \end{bmatrix}$ where $\mathbf{w}_1 \in R_{d_1,q_2}^t$ and $\mathbf{W}_2 \in R_{d_1,q_2}^{n\times t}$ and an encoding $\mathbf{c} = \begin{bmatrix} c_1 \\ \mathbf{c}_2 \end{bmatrix} \in R_{d_1,q_1}^{n+1}$, compute

$$\hat{c}_1 = \kappa^{-1}\left(\mathbf{w}_1^{\mathsf{T}}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1} c_1 \right\rceil \bmod q_2\right)\right) \in R_{d_2,q_2}$$
$$\hat{\mathbf{c}}_2 = \kappa^{-1}\left(\left\lfloor \tfrac{q_3}{q_1}\mathbf{c}_2 + \tfrac{q_3}{q_2}\mathbf{W}_2\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right\rceil \bmod q_3\right) \in R_{d_2,q_3}^n.$$

  The computations inside "$\lfloor\cdot\rceil \bmod q_i$" are performed over the rationals.[7] Output the encoding $(\hat{c}_1, \hat{\mathbf{c}}_2)$.

- $\mathsf{CompressRecover}(\mathbf{S}_2, (\hat{c}_1, \hat{\mathbf{c}}_2))$: On input a secret key $\mathbf{S}_2 = [-\tilde{\mathbf{s}}_2 \mid \mathbf{I}_n] \in R_{d_2,q_2}^{n\times(n+1)}$ and a compressed encoding $(\hat{c}_1, \hat{\mathbf{c}}_2) \in R_{d_2,q_2} \times R_{d_2,q_3}^n$, output

$$\mathbf{z} = \left\lfloor -\frac{q_3}{q_2}(\tilde{\mathbf{s}}_2 \cdot \hat{c}_1)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2 \in R_{d_2,q_3}^n. \tag{C.1}$$

  The computations inside "$\lfloor\cdot\rceil \bmod q_i$" are performed over the rationals, in the same way as in Compress.

Theorem C.4 (Response Compression Correctness)**.** *Let $q_1 \geq q_2 \geq q_3 \geq p$ be ring moduli. Let $d_1 = 2^{\delta_1} \geq d_2 = 2^{\delta_2}$ be (power-of-two) ring dimensions. Let $k = d_1/d_2$ and let $\nu = \delta_1 - \delta_2$. Let χ be an error distribution over R_{d_2}, $z \in \mathbb{N}$ be a decomposition base, and $n \in \mathbb{N}$ be the vector dimension. Let $t = \lfloor \log_z q_2 \rfloor + 1$. Define the following:*

- *Suppose $\mathbf{c} = \begin{bmatrix} c_1 \\ \mathbf{c}_2 \end{bmatrix} \in R_{d_1,q_1}^{n+1}$ is an RLWE encoding of $\lfloor q_1/p \rfloor\, \mathbf{m}$ for some message $\mathbf{m} \in R_{d_1,p}^n$ with respect to $\mathbf{S}_1$ (when viewed as a secret key over R_{d_1,q_1}) and error $\mathbf{e} \in R_{d_1}^n$.*
- *Suppose $\mathbf{S}_2 = [-\tilde{\mathbf{s}}_2 \mid \mathbf{I}_n] \in R_{d_2,q_2}^{n\times(n+1)}$ is the target key.*
- *Suppose $\mathbf{W} \leftarrow \mathsf{CompressSetup}(\mathbf{S}_1, \mathbf{S}_2)$, $(\hat{c}_1, \hat{\mathbf{c}}_2) \leftarrow \mathsf{Compress}(\mathbf{W}, \mathbf{c})$, and $\mathbf{z} \leftarrow \mathsf{CompressRecover}(\mathbf{S}_2, (\hat{c}_1, \hat{\mathbf{c}}_2))$.*

Then $\mathbf{z} = \lfloor q_3/p \rfloor\, \kappa^{-1}(\mathbf{m}) + \tilde{\mathbf{e}} \in R_{d_2,q_3}^n$, where $\tilde{\mathbf{e}} = \tilde{\mathbf{e}}_1 + \tilde{\mathbf{e}}_2$ and

- $\|\tilde{\mathbf{e}}_1\|_\infty \leq \frac{1}{2}\left(2 + (q_3 \bmod p) + \frac{q_3}{q_1}(q_1 \bmod p)\right)$.

[7] More explicitly, we first lift the quantities inside "$\lfloor\cdot\rceil \bmod q_i$" to the rationals by associating the coefficients of each ring element (i.e., each polynomial) with its unique *integer* representative in the interval $[-q_i/2, q_i/2]$. We then perform all operations over the rationals. After evaluating the arithmetic operations on the polynomials with rational coefficients, rounding yields a polynomial with integer coefficients and taking the result modq_i yields an element of R_{d_1,q_i}.

- *Suppose the components of* $\mathbf{e}$ *are subgaussian with parameter* σ_e, *the components of* $\tilde{\mathbf{s}}_1$ *are subgaussian with parameter* σ_s, *and the distribution* χ *is subgaussian with parameter* σ_χ. *Then, under the independence heuristic, the components of* $\tilde{\mathbf{e}}_2$ *are subgaussian with variance*

$$\tilde{\sigma}^2 = \frac{q_3^2}{q_1^2}\sigma_{\mathbf{e}}^2 + \frac{q_3^2}{4q_2^2}d_1\sigma_s^2 + \frac{q_3^2}{q_2^2}\sigma_\chi^2 B^2,$$

where $B = \left\| \mathbf{g}_z^{-1}(\lfloor {}^{q_2}/_{q_1} \cdot c_1 \rceil \bmod q_2)\right\|_2$. *Note that a trivial bound for* B *is the bound* $B \le \sqrt{td_1} \cdot z/2$.

Proof. Let $\kappa \colon R_{d_2} \to R_{d_1}$ be the subring embedding (Definition A.9), and let $\pi_\nu \colon R_{d_1} \to R_{d_1}$ be the projection map (i.e., the mapping $r \mapsto \kappa(\kappa^{-1}(r))$ from Lemma A.11). We will show that

$$\kappa(\mathbf{z}) = \lfloor q_3/p \rfloor \kappa(\kappa^{-1}(\mathbf{m})) + \kappa(\mathbf{e}') = \lfloor q_1/p \rfloor \pi_\nu(\mathbf{m}) + \kappa(\mathbf{e}').$$

The claim then follows by the fact that κ is an injective ring homomorphism (Lemma A.10). First, write $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^{\mathsf{T}} \\ \mathbf{W}_2 \end{bmatrix}$ where $\mathbf{w}_1 \in R_{d_1,q_2}^t$ and $\mathbf{W}_2 \in R_{d_1,q_2}^{n\times t}$ and let $\mathbf{c} = \begin{bmatrix} c_1 \\ \mathbf{c}_2 \end{bmatrix}$ where $c_1 \in R_{d_1,q_1}$ and $\mathbf{c}_2 \in R_{d_1,q_1}^n$. Let

$$\begin{aligned} \hat{c}_1' &= \mathbf{w}_1^{\mathsf{T}}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1 \right\rceil \bmod q_2\right) \in R_{d_2,q_2} \\ \hat{\mathbf{c}}_2' &= \left\lfloor \tfrac{q_3}{q_1}\mathbf{c}_2 + \tfrac{q_3}{q_2}\mathbf{W}_2\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1 \right\rceil \bmod q_2\right)\right\rceil \bmod q_3 \in R_{d_2,q_3}^n. \end{aligned} \tag{C.2}$$

Then, $\hat{c}_1 = \kappa^{-1}(\hat{c}_1')$ and $\hat{\mathbf{c}}_2 = \kappa^{-1}(\hat{\mathbf{c}}_2')$. By Lemmas A.10 and A.11 and using the definition of $\mathbf{z}$ from Eq. (C.1), we have

$$\begin{aligned} \kappa(\mathbf{z}) = \left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot\kappa(\hat{c}_1)\right)\right\rceil \bmod q_3 + \kappa(\hat{\mathbf{c}}_2) &= \left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot\pi_\nu(\hat{c}_1')\right)\right\rceil \bmod q_3 + \pi_\nu(\hat{\mathbf{c}}_2') \\ &= \pi_\nu\left(\left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot\hat{c}_1'\right)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2'\right) \in R_{d_1,q_3}^n. \end{aligned} \tag{C.3}$$

Substituting in the values of $\hat{c}_1'$ and $\hat{\mathbf{c}}_2'$ from Eq. (C.2) and working over the *rationals*, we have:

$$\begin{aligned} \left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot\hat{c}_1'\right)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2' &= \left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot\mathbf{w}_1^{\mathsf{T}}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right)\right\rceil \\ &\quad + \left\lfloor \tfrac{q_3}{q_1}\mathbf{c}_2 + \tfrac{q_3}{q_2}\mathbf{W}_2\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right\rceil + \boldsymbol{\xi}_1 q_3 \\ &= -\frac{q_3}{q_2}\kappa(\tilde{\mathbf{s}}_2)\cdot\mathbf{w}_1^{\mathsf{T}}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} \\ &\quad + \frac{q_3}{q_1}\mathbf{c}_2 + \frac{q_3}{q_2}\mathbf{W}_2\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \mathbf{e}_{\lfloor\cdot\rceil,2} + \boldsymbol{\xi}_1 q_3 \\ &= \frac{q_3}{q_2}\left(-\kappa(\tilde{\mathbf{s}}_2)\mathbf{w}_1^{\mathsf{T}} + \mathbf{W}_2\right)\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \boldsymbol{\xi}_1 q_3 + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}, \end{aligned} \tag{C.4}$$

where $\mathbf{e}_{\lfloor\cdot\rceil,1}$ and $\mathbf{e}_{\lfloor\cdot\rceil,2}$ are the error terms introduced by the rounding operations, and $\boldsymbol{\xi}_1 \in R_{d_1}^n$. By definition, the norms of $\mathbf{e}_{\lfloor\cdot\rceil,1}$ and $\mathbf{e}_{\lfloor\cdot\rceil,2}$ are bounded by $1/2$. Since $\mathbf{W} \leftarrow \mathsf{CompressSetup}(\mathsf{S}_1, \mathsf{S}_2)$, we know that

$$\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^{\mathsf{T}} \\ \mathbf{W}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{a}^{\mathsf{T}} \\ \mathbf{B} - \tilde{\mathbf{s}}_1 \cdot \mathbf{g}_z^{\mathsf{T}} \end{bmatrix} \in R_{d_1,q_2}^{(n+1)\times t},$$

and by definition of Π, we have $\mathbf{a} = \sum_{i\in[k]} x^{i-1}\kappa(\mathbf{a}_i)$ and $\mathbf{B} = \sum_{i\in[k]} x^{i-1}\kappa(\tilde{\mathbf{s}}_2\mathbf{a}_i^{\mathsf{T}} + \mathbf{E}_i)$ where $\mathbf{a}_1, \ldots, \mathbf{a}_k \in R_{d_2,q_2}^t$ and $\mathbf{E}_1, \ldots, \mathbf{E}_k \leftarrow \chi^{n\times t}$. Since κ is a ring homomorphism (Lemma A.10), we can write

$$-\kappa(\tilde{\mathbf{s}}_2)\mathbf{w}_1^{\mathsf{T}} + \mathbf{W}_2 = -(\tilde{\mathbf{s}}_1 \bmod q_2)\cdot\mathbf{g}_z^{\mathsf{T}} + \sum_{i\in[k]} x^{i-1}\left(-\kappa(\tilde{\mathbf{s}}_2)\kappa(\mathbf{a}_i^{\mathsf{T}}) + \kappa(\tilde{\mathbf{s}}_2\mathbf{a}_i^{\mathsf{T}} + \mathbf{E}_i)\right) = -(\tilde{\mathbf{s}}_1 \bmod q_2)\cdot\mathbf{g}_z^{\mathsf{T}} + \sum_{i\in[k]} x^{i-1}\mathbf{E}_i \in R_{d_1,q_2}^{n\times t}$$

Let $\mathbf{E} = \sum_{i\in[k]} x^{i-1}\mathbf{E}_i$. Then, over the rationals, we have

$$-\kappa(\tilde{\mathbf{s}}_2)\mathbf{w}_1^{\mathsf{T}} + \mathbf{W}_2 = -\tilde{\mathbf{s}}_1\cdot\mathbf{g}_z^{\mathsf{T}} + \mathbf{E} + \Xi q_2,$$

where $\Xi \in R_{d_1}^{n\times t}$. Substituting back into Eq. (C.4), we have over the rationals,

$$\begin{aligned}\left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot \hat{c}_1'\right)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2' &= \frac{q_3}{q_2}\left(-\kappa(\tilde{\mathbf{s}}_2)\mathbf{w}_1^{\intercal} + \mathbf{W}_2\right)\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \boldsymbol{\xi}_1 q_3 + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}\\ &= \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\mathbf{g}_z^{\intercal} + \mathbf{E} + \Xi q_2\right)\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \boldsymbol{\xi}_1 q_3 + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}\\ &= \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \left(\boldsymbol{\xi}_1 + \Xi\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right)q_3\\ &\quad + \frac{q_3}{q_2}\left(\mathbf{E}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}\\ &= \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \left(\boldsymbol{\xi}_1 + \boldsymbol{\xi}_2 + \Xi\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right)q_3\\ &\quad + \frac{q_3}{q_2}\left(\mathbf{E}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2},\end{aligned} \tag{C.5}$$

where $\boldsymbol{\xi}_2 \in R_{d_1}^{n}$. Still working over the rationals, we can write

$$\frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil\right) + \frac{q_3}{q_1}\mathbf{c}_2 = \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left(\tfrac{q_2}{q_1}c_1 + e_{\lfloor\cdot\rceil,3}\right)\right) + \frac{q_3}{q_1}\mathbf{c}_2, \tag{C.6}$$

where $|e_{\lfloor\cdot\rceil,3}| \le 1/2$ is another rounding error. By assumption, $\mathbf{c}$ is a RLWE encoding of $\lfloor q_1/p\rfloor\,\mathbf{m}$ with respect to S_1 and error $\mathbf{e}$. This means $-\tilde{\mathbf{s}}_1 c_1 + \mathbf{c}_2 = \lfloor q_1/p\rfloor\,\mathbf{m} + \mathbf{e} \in R_{d_1,q_1}^{n}$. Equivalently, over R_{d_1}, we can write

$$-\tilde{\mathbf{s}}_1 c_1 + \mathbf{c}_2 = \lfloor q_1/p\rfloor\,\mathbf{m} + \mathbf{e} + \boldsymbol{\xi}_3 q_1,$$

where $\boldsymbol{\xi}_3 \in R_{d_1}^{n}$. Thus, we can rewrite Eq. (C.6) (over the rationals) as

$$\begin{aligned}\frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil\right) + \frac{q_3}{q_1}\mathbf{c}_2 &= \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left(\tfrac{q_2}{q_1}c_1 + e_{\lfloor\cdot\rceil,3}\right)\right) + \frac{q_3}{q_1}\mathbf{c}_2\\ &= \frac{q_3}{q_1}\left(-\tilde{\mathbf{s}}_1 c_1 + \mathbf{c}_2\right) - \frac{q_3}{q_2}\tilde{\mathbf{s}}_1 e_{\lfloor\cdot\rceil,3}\\ &= \frac{q_3}{q_1}\left(\lfloor q_1/p\rfloor\mathbf{m} + \mathbf{e} + \boldsymbol{\xi}_3 q_1\right) - \frac{q_3}{q_2}\tilde{\mathbf{s}}_1 e_{\lfloor\cdot\rceil,3}\\ &= \frac{q_3}{q_1}\lfloor q_1/p\rfloor\mathbf{m} + \frac{q_3}{q_1}\mathbf{e} + \boldsymbol{\xi}_3 q_3 - \frac{q_3}{q_2}\tilde{\mathbf{s}}_1 e_{\lfloor\cdot\rceil,3}\end{aligned}$$

Let $\hat{\boldsymbol{\xi}} = \left(\boldsymbol{\xi}_1 + \boldsymbol{\xi}_2 + \boldsymbol{\xi}_3 + \Xi\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right) \in R_{d_1}^{n}$. Then, Eq. (C.5) becomes (over the rationals)

$$\begin{aligned}\left\lfloor -\tfrac{q_3}{q_3}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot \hat{c}_1'\right)\right\rceil \bmod q_2 + \hat{\mathbf{c}}_2' &= \frac{q_3}{q_2}\left(-\tilde{\mathbf{s}}_1\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil\right) + \frac{q_3}{q_1}\mathbf{c}_2 + \left(\boldsymbol{\xi}_1 + \boldsymbol{\xi}_2 + \Xi\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right)q_3\\ &\quad + \frac{q_3}{q_2}\left(\mathbf{E}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}\\ &= \frac{q_3}{q_1}\lfloor q_1/p\rfloor\mathbf{m} + \hat{\boldsymbol{\xi}} q_3 + \frac{q_3}{q_1}\mathbf{e} - \frac{q_3}{q_2}\tilde{\mathbf{s}}_1 e_{\lfloor\cdot\rceil,3}\\ &\quad + \frac{q_3}{q_2}\left(\mathbf{E}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right)\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}\\ &= \frac{q_3}{q_1}\left\lfloor \frac{q_1}{p}\right\rfloor\mathbf{m} + \hat{\boldsymbol{\xi}} q_3 + \underbrace{\frac{q_3}{q_1}\mathbf{e} + \frac{q_3}{q_2}\left(\mathbf{E}\mathbf{g}_z^{-1}\left(\left\lfloor \tfrac{q_2}{q_1}c_1\right\rceil \bmod q_2\right) - \tilde{\mathbf{s}}_1 e_{\lfloor\cdot\rceil,3}\right) + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}}_{\mathbf{e}_2'}.\end{aligned} \tag{C.7}$$

Finally, we can write $\lfloor q_1/p \rfloor = q_1/p - (q_1 \bmod p)/p$. Then Eq. (C.7) becomes (over the rationals)

$$\begin{aligned}\left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot \hat{c}_1'\right)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2' &= \frac{q_3}{q_1}\left\lfloor \frac{q_1}{p}\right\rfloor \mathbf{m} + \hat{\boldsymbol{\xi}} q_3 + \mathbf{e}_2' + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2} \\ &= \left(\frac{q_3}{q_1}\cdot\frac{q_1}{p} - \frac{q_3}{q_1}\cdot\frac{q_1 \bmod p}{p}\right)\mathbf{m} + \hat{\boldsymbol{\xi}} q_3 + \mathbf{e}_2' + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2} \\ &= \left\lfloor \frac{q_3}{p}\right\rfloor \mathbf{m} + \hat{\boldsymbol{\xi}} q_3 + \underbrace{\left(\frac{q_3 \bmod p}{p} - \frac{q_3(q_1 \bmod p)}{pq_1}\right)\mathbf{m} + \mathbf{e}_{\lfloor\cdot\rceil,1} + \mathbf{e}_{\lfloor\cdot\rceil,2}}_{\mathbf{e}_1'} + \mathbf{e}_2'.\end{aligned} \tag{C.8}$$

Furthermore, the terms in Eq. (C.8) are all in R_{d_1}, so the equation holds over R_{d_1}, and thus over R_{d_1,q_3} as well. Combining Eqs. (C.3) and (C.8) and using linearity of π_ν (Lemma A.11), we have (over R_{d_1,q_3} now)

$$\kappa(\mathbf{z}) = \pi_\nu\left(\left\lfloor -\tfrac{q_3}{q_2}\left(\kappa(\tilde{\mathbf{s}}_2)\cdot \hat{c}_1'\right)\right\rceil \bmod q_3 + \hat{\mathbf{c}}_2'\right) = \left\lfloor \frac{q_3}{p}\right\rfloor \pi_\nu(\mathbf{m}) + \pi_\nu(\mathbf{e}_1') + \pi_\nu(\mathbf{e}_2') \in R^n_{d_1,q_3}.$$

Applying the inversion map κ^{-1} to both sides and appealing to Lemma A.11 (i.e., for all $r \in R_{d_1}$, $\kappa^{-1}(\pi_\nu(r)) = \kappa^{-1}(\kappa(\kappa^{-1}(r))) = \kappa^{-1}(r)$), we have

$$\mathbf{z} = \left\lfloor \frac{q_3}{p}\right\rfloor \kappa^{-1}(\mathbf{m}) + \kappa^{-1}(\mathbf{e}_1') + \kappa^{-1}(\mathbf{e}_2').$$

The claim now holds by setting $\tilde{\mathbf{e}}_1 = \kappa^{-1}(\mathbf{e}_1')$ and $\tilde{\mathbf{e}}_2 = \kappa^{-1}(\mathbf{e}_2')$. By construction of κ^{-1}, for all $r \in R_{d_1}$, the coefficients of the polynomial $\kappa^{-1}(r)$ are a *subset* of the coefficients of r. Since $\|\mathbf{m}\|_\infty \le p/2$, we can bound $\|\tilde{\mathbf{e}}_1\|_\infty$ from Eq. (C.8) by

$$\|\tilde{\mathbf{e}}_1\|_\infty \le \|\mathbf{e}_1'\|_\infty \le \frac{1}{2}\left(q_3 \bmod p + \frac{q_3(q_1 \bmod p)}{q_1} + 2\right).$$

Next, consider the components of $\mathbf{e}_2'$ from Eq. (C.7). Since the components of $\mathbf{E}$ are subgaussian with parameter σ_χ and moreover, $B = \|\mathbf{g}_z^{-1}(\lfloor {}^{q_2}/_{q_1} \cdot c_1\rceil \bmod q_2)\|_2$, we conclude by Lemma A.1 that the components of $\mathbf{E}\mathbf{g}_z^{-1}(\lfloor {}^{q_2}/_{q_1}\cdot c_1\rceil \bmod q_2)$ are subgaussian with variance $\sigma_\chi^2 B^2$. Then, under the independence heuristic, the components of $\mathbf{e}_2'$ are subgaussian with variance

$$\tilde{\sigma}^2 = \frac{q_3^2}{q_1^2}\sigma_e^2 + \frac{q_3^2}{4q_2^2} d_1 \sigma_s^2 + \frac{q_3^2}{q_2^2}\sigma_\chi^2 B^2.$$

Again by construction of κ^{-1}, the same then holds for the coefficients of $\tilde{\mathbf{e}}_2 = \kappa^{-1}(\mathbf{e}_2)$. □

Remark C.5 (Tighter Gadget Bound)**.** In Theorem C.4, the final error variance is stated in terms of the bound $B = \left\|\mathbf{g}_z^{-1}(\lfloor {}^{q_2}/_{q_1}\cdot c_1\rceil \bmod q_2)\right\|_2$. The trivial bound is $B \le \sqrt{td_1}\cdot z/2$. However, in many cases, this bound will be loose since "many" of the coefficients of $\mathbf{g}_z^{-1}(\cdot)$ will be much smaller than $z/2$. Concretely, suppose $z = 2$ and consider the distribution of $\mathbf{u}^\intercal := \mathbf{g}_z^{-1}(y)$ where $y \xleftarrow{R} R_{d_1,q_2}$. First, consider the case where q_2 is a power-of-two. In this case, the coefficients in each component $u_1, \ldots, u_t \in R_{d_1}$ is an independent Bernoulli variable with probability $1/2$. Correspondingly, the ℓ_2 norm of $\mathbf{u}$ is a sum of $d_1 t$ independent Bernoulli random variables (i.e., a binomial random variable). In this case, it is easy to calculate the exact probability that $\|\mathbf{u}\|_2$ exceeds a certain threshold. Concretely, when $d_1 = 2048$, we can show that with probability at least $1 - 2^{-48.4}$, $\|\mathbf{u}\|_2 \le \sqrt{t\cdot 2048}\cdot\eta$ where $\eta = 1200/2048$. Next, observe that in the case where q_2 is *not* a power-of-two, then the coefficients of $y \xleftarrow{R} R_{d_1,q_2}$ can only *decrease*, which can only reduce the probability that $\|\mathbf{u}\|_2$ exceeds the bound $B = \sqrt{t\cdot 2048}\cdot\eta$. We use this tighter bound in the *final* step of our correctness analysis (Appendix D.1). Note that this analysis assumes that the distribution of y (in the scheme, the distribution of $\lfloor q_2/q_1 \cdot c_1\rceil \bmod q_2$) is uniform over $R^n_{d_2,q_2}$. We can ensure this by "re-randomizing" c_1. Namely, we include in the public parameters a fresh encoding c' of 0 (which is pseudorandom under RLWE). The algorithm then applies the compression algorithm to the encoding $c_1 + c'$, which encodes the same underlying value, but whose distribution is computationally indistinguishable from uniform.

D Correctness and Security of Respire

In this section, we prove the correctness and security of the Respire protocol (Construction 3.3). Since the base version of Respire (Construction 3.2) is a special case (in fact, a sub-protocol) of the batched version of Respire (Construction 3.3), we focus exclusively on the batched version in our correctness and security analysis.

D.1 Correctness Analysis for Respire

We use the parameters from Construction 3.3. In the following, for $(i, j) \in \{(1, 2), (1, 3), (2, 3)\}$, we write $\kappa_{d_i,d_j} : R_{d_j} \to R_{d_i}$ and $\kappa^{-1}_{d_i,d_j} : R_{d_i} \to R_{d_j}$ to denote the subring embedding and the dimension-reduction mappings, respectively (Definition A.9). Let N be the number of database records and $T = n_{\mathsf{vec}} \cdot (d_2/d_3)$ be the batch size. Assume the plaintext modulus p divides q_3. Suppose the following properties hold for the error distributions appearing in Construction 3.3:

- Suppose $\chi_{1,e}, \chi'_{1,e}, \chi_{2,e}$ are subgaussian with variances $\sigma^2_{1,e}$, $(\sigma'_{1,e})^2$, and $\sigma^2_{2,e}$, respectively.
- Suppose $\chi_{1,s}$ is $B_{1,s}$-bounded and $\chi'_{1,s}$ is subgaussian with variance $(\sigma'_{1,s})^2$.

We also define the decomposition bases used in each of the underlying algorithms:

- Let z_{GSW} be the GSW decomposition base and $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q_1 \rfloor + 1$.
- Let $z_{\mathsf{coeff,RLWE}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}$ be the decomposition bases for query packing (Construction B.6) and $t_{\mathsf{coeff,RLWE}} = \lfloor \log_{z_{\mathsf{coeff,RLWE}}} q_1 \rfloor + 1$, $t_{\mathsf{coeff,GSW}} = \lfloor \log_{z_{\mathsf{coeff,GSW}}} q_1 \rfloor + 1$, and $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q_1 \rfloor + 1$.
- Let z_{proj} be the decomposition base used for projection (Construction A.7) and $t_{\mathsf{proj}} = \lfloor \log_{z_{\mathsf{proj}}} q_1 \rfloor + 1$.
- Let z_{vec} be the decomposition base used for vectorization (Construction C.1) and $t_{\mathsf{vec}} = \lfloor \log_{z_{\mathsf{vec}}} q_1 \rfloor + 1$.
- Let z_{comp} be the decomposition base used for response compression (Construction C.3) and $t_{\mathsf{comp}} = \lfloor \log_{z_{\mathsf{comp}}} q_2 \rfloor + 1$.

Take any collection of records $r_{\alpha,\beta,\gamma} \in R_{d_3,p}$ where $\alpha \in [2^{\nu_1}]$, $\beta \in [2^{\nu_2}]$, and $\gamma \in [2^{\nu_3}]$. Take any collection of indices $\mathsf{idx}_1, \ldots, \mathsf{idx}_T$, where $\mathsf{idx}_k = (\alpha_k, \beta_k, \gamma_k)$. Suppose we now sample the following:

$$\begin{aligned}
\big((\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{proj}}, \mathsf{pp}_{\mathsf{vec}}, \mathsf{pp}_{\mathsf{comp}}), (\mathbf{s}_1, \mathbf{S}_2)\big) &\leftarrow \mathsf{Setup}(1^\lambda) \\
\mathsf{db} = \big\{\tilde{r}_{\alpha,\beta}\big\}_{\alpha\in[2^{\nu_1}],\beta\in[2^{\nu_2}]} &\leftarrow \mathsf{SetupDB}\big(1^\lambda, \{r_{\alpha,\beta,\gamma}\}_{\alpha\in[2^{\nu_1}],\beta\in[2^{\nu_2}],\gamma\in[2^{\nu_3}]}\big) \\
\mathsf{q} = (\mathsf{q}_1, \ldots, \mathsf{q}_T) &\leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx}_1, \ldots, \mathsf{idx}_b) \\
\mathsf{a} &\leftarrow \mathsf{Answer}(\mathsf{pp}, \mathsf{q}) \\
(\mathsf{resp}_1, \ldots, \mathsf{resp}_T) &\leftarrow \mathsf{Extract}(\mathsf{qk}, \mathsf{a}).
\end{aligned}$$

We now show that with high probability for a given $k \in [T]$, the decoded response resp_k satisfies $\mathsf{resp}_k = r_{\alpha_k,\beta_k,\gamma_k}$. Take any index $k \in [T]$. Our analysis follows the steps of the Answer algorithm in Construction 3.3. Specifically, we analyze the variance of the error in the encodings after each step of the computation (under the independence heuristic). In the following, we will say that an RLWE encoding (resp., a GSW encoding) has *error variance* σ^2 if the error associated with the encoding is distributed according to a subgaussian with variance at most σ^2. Following the definitions in the Query algorithm, let $\hat{\alpha}_i = 1$ if $i = \alpha_k$ and $\hat{\alpha} = 0$ otherwise. Let $\hat{\beta}_1 \cdots \hat{\beta}_{\nu_2}$ be the binary representation of $\beta_k - 1$ and $\hat{\gamma}_1 \cdots \hat{\gamma}_{\nu_3}$ be the binary representation of $\gamma_k - 1$. We now consider each step of the Answer algorithm.

1. **Query expansion:** Let

$$\Big(\big(\mathbf{c}_1^{(1)}, \ldots, \mathbf{c}_{2^{\nu_1}}^{(1)}\big), \big(\mathbf{C}_1^{(2)}, \ldots, \mathbf{C}_{\nu_2}^{(2)}, \mathbf{C}_1^{(3)}, \ldots, \mathbf{C}_{\nu_3}^{(3)}\big)\Big) \leftarrow \mathsf{QueryUnpack}(\mathsf{pp}_{\mathsf{qpk}}, \mathsf{q}_k).$$

   be the output of the query expansion algorithm on query q_k. By Theorem B.7, the following holds:

- For all $i \in [2^{\nu_1}]$, $\mathbf{c}_i^{(1)}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \hat{\alpha}_i$ with error variance

$$\sigma^2_{\mathsf{RLWE}} = \sigma^2_{1,e}(1 + t_{\mathsf{coeff,RLWE}} d_1^3 z^2_{\mathsf{coeff,RLWE}}/12).$$

- Each $\mathbf{C}_i^{(2)}$ is a GSW encoding of $\hat{\beta}_i$ with error variance

$$\sigma^2_{\mathsf{GSW}} = \sigma^2_{1,e}(d_1 B^2_{1,s}(1 + t_{\mathsf{coeff,GSW}} d_1^3 z^2_{\mathsf{coeff,GSW}}/12) + t_{\mathsf{conv}} d_1 z^2_{\mathsf{conv}}/2).$$

- Each $\mathbf{C}_i^{(3)}$ is a GSW encoding of $\hat{\gamma}_i$ with error variance σ^2_{GSW}.

All of these encodings are with respect to the secret key $\mathbf{s}_1$. Unless otherwise noted, all encodings in the subsequent description are with respect to $\mathbf{s}_1$.

2. **First dimension:** The Answer algorithm computes $\hat{\mathbf{c}}_\beta^{(1)} = \sum_{\alpha \in [2^{\nu_1}]} \tilde{r}_{\alpha,\beta} \cdot \mathbf{c}_\alpha^{(1)}$ for each $\beta \in [2^{\nu_2}]$. Since $\|r_{\alpha,\beta}\|_\infty \leq p/2$, this means that $\hat{\mathbf{c}}_\beta^{(1)}$ is an encoding of $\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta}$ with error variance

$$\sigma^2_{\mathsf{first}} = 2^{\nu_1} d_1 (p/2)^2 \sigma^2_{\mathsf{RLWE}}.$$

3. **Folding:** Next, the Answer algorithm sets $\hat{\mathbf{c}}_{0,j}^{(2)} = \hat{\mathbf{c}}_j^{(1)}$ for each $j \in [2^{\nu_2}]$. Then, for each $r \in [\nu_2]$ and $j \in [2^{\nu_2 - r}]$, it computes

$$\hat{\mathbf{c}}_{r,j}^{(2)} = \mathsf{Select}\Big(\mathbf{C}_r^{(2)},\ \hat{\mathbf{c}}_{r-1,j}^{(2)},\ \hat{\mathbf{c}}_{r-1,j+2^{\nu_2-r}}^{(2)}\Big).$$

Thus, the following properties hold:

- From the previous step, we have that $\hat{\mathbf{c}}_{0,j}^{(2)}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta}$ with error variance $\sigma^2_{\mathsf{first}}$.
- From Theorem A.3, for each $r \in [\nu_2]$ and $j \in [2^{\nu_2 - r}]$, we have that $\hat{\mathbf{c}}_{r,j}^{(2)}$ is an RLWE encoding of

$$\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k, j + \sum_{i \in [r]} \hat{\beta}_i 2^{\nu_2 - i}}$$

  with error variance $\sigma^2_{\mathsf{first}} + r \cdot 2 t_{\mathsf{GSW}} d_1 z^2_{\mathsf{GSW}} \sigma^2_{\mathsf{GSW}}/4 = \sigma^2_{\mathsf{first}} + r t_{\mathsf{GSW}} d_1 z^2_{\mathsf{GSW}} \sigma^2_{\mathsf{GSW}}/2$.

Since $\hat{\beta}_1 \cdots \hat{\beta}_{\nu_2}$ is the binary representation of $\beta_k - 1$, we have that $1 + \sum_{i \in [\nu_2]} \hat{\beta}_i 2^{\nu_2 - i} = \beta_k$. This means that $\hat{\mathbf{c}}_{\nu_2,1}^{(2)}$ is an encoding of $\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k}$ with error variance

$$\sigma^2_{\mathsf{fold}} = \sigma^2_{\mathsf{first}} + \nu_2 t_{\mathsf{GSW}} d_1 z^2_{\mathsf{GSW}} \sigma^2_{\mathsf{GSW}}/2.$$

4. **Rotation:** Next, the Answer algorithm sets $\hat{\mathbf{c}}_0^{(3)} = \hat{\mathbf{c}}_{\nu_2,1}^{(2)}$, and for each $r \in [\nu_3]$, it computes

$$\hat{\mathbf{c}}_r^{(3)} = \mathsf{Select}\Big(\mathbf{C}_r^{(3)},\ \hat{\mathbf{c}}_{r-1}^{(3)},\ x^{-2^{\nu_3 - r}} \cdot \hat{\mathbf{c}}_{r-1}^{(3)}\Big).$$

Similar to the previous case, we can appeal to Theorem A.3:

- First, $\hat{\mathbf{c}}_0^{(3)}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k}$.
- From Theorem A.3, for each $r \in [\nu_2]$, we have that $\hat{\mathbf{c}}_r^{(3)}$ is an RLWE encoding of

$$\lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k} \cdot \prod_{i \in [r]} x^{-\hat{\gamma}_i 2^{\nu_3 - i}} = \lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k} \cdot x^{\sum_{i \in [r]} -\hat{\gamma}_i 2^{\nu_3 - i}}$$

  with error variance $\sigma^2_{\mathsf{fold}} + r \cdot 2 t_{\mathsf{GSW}} d_1 z^2_{\mathsf{GSW}} \sigma^2_{\mathsf{GSW}}/4 = \sigma^2_{\mathsf{fold}} + r t_{\mathsf{GSW}} d_1 z^2_{\mathsf{GSW}} \sigma^2_{\mathsf{GSW}}/2$.

Since $\hat{\gamma}_1 \cdots \hat{\gamma}_{\nu_3}$ is the binary representation of $\gamma_k - 1$, we have that $\sum_{i \in [\nu_3]} \hat{\gamma}_i 2^{\nu_3 - i} = \gamma_k - 1$. Thus, we conclude that $\hat{\mathbf{c}}_{\nu_3}^{(3)}$ is an encoding of $x^{-(\gamma_k - 1)} \cdot \lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k}$ with error variance

$$\begin{aligned}\sigma_{\mathsf{rot}}^2 &= \sigma_{\mathsf{fold}}^2 + \nu_3 t_{\mathsf{GSW}} d_1 z_{\mathsf{GSW}}^2 \sigma_{\mathsf{GSW}}^2 / 2 \\ &= \sigma_{\mathsf{first}}^2 + (\nu_2 + \nu_3) t_{\mathsf{GSW}} d_1 z_{\mathsf{GSW}}^2 \sigma_{\mathsf{GSW}}^2 / 2.\end{aligned}$$

5. **Projection:** Thus far, we have established that for all $k \in [T]$, $\mathbf{c}_k^{(\mathsf{out})}$ is an RLWE encoding of $x^{-(\gamma_k-1)} \cdot \lfloor q_1/p \rfloor \cdot \tilde{r}_{\alpha_k,\beta_k}$ with error $e_{\mathsf{single},k}$, where $e_{\mathsf{single},k}$ is subgaussian with variance σ_{rot}^2. In other words, it holds that

$$\mathbf{s}_1^{\mathsf{T}} \mathbf{c}_k^{(\mathsf{out})} = \lfloor q_1/p \rfloor \cdot \mu_k + e_{\mathsf{single},k},$$

where $\mu_k = x^{-(\gamma_k-1)} \cdot \tilde{r}_{\alpha_k,\beta_k} \in R_{d_1,p}$. Now, the projection algorithm computes for each $k \in [T]$,

$$\mathbf{c}_k^{(\mathsf{proj})} \leftarrow \mathsf{Project}\big(\mathsf{pp}_{\mathsf{proj}}, \mathbf{c}_k^{(\mathsf{out})}, \delta_1 - \delta_3\big).$$

By Theorem A.8 and linearity of $\pi_{\delta_1-\delta_3}$ (see Lemma A.11), we conclude that $\mathbf{c}_k^{(\mathsf{proj})}$ is an RLWE encoding of $\pi_{\delta_1-\delta_3}(\mu_k)$ with error $\pi_{\delta_1-\delta_3}(e_{\mathsf{single},k}) + e_{\mathsf{proj},k}$, and $e_{\mathsf{proj},k}$ is subgaussian with variance

$$\sigma_{\mathsf{proj}}^2 = (4^{\delta_1-\delta_3} - 1)/12 \cdot t_{\mathsf{proj}} d_1 z_{\mathsf{proj}}^2 \sigma_{1,e}^2.$$

Define $\tilde{\mu}_k \coloneqq \kappa_{d_1,d_3}^{-1}(\mu_k) \in R_{d_3,p}$. By Lemma A.11, we have

$$\kappa_{d_1,d_3}(\tilde{\mu}_k) = \pi_{\delta_1-\delta_3}(\mu_k). \tag{D.1}$$

By definition of SetupDB and Lemma A.13,

$$\tilde{\mu}_k = \kappa_{d_1,d_3}^{-1}(\mu_k) = \kappa_{d_1,d_3}^{-1}\big(x^{-(\gamma_k-1)} \cdot \tilde{r}_{\alpha_k,\beta_k}\big) = \kappa_{d_1,d_3}^{-1}\big(x^{-(\gamma_k-1)} \cdot \Pi\big(r_{\alpha_k,\beta_k,1}, \ldots, r_{\alpha_k,\beta_k,2^{\nu_3}}\big)\big) = r_{\alpha_k,\beta_k,\gamma_k}. \tag{D.2}$$

We can similarly define $\tilde{e}_{\mathsf{single},k} \coloneqq \kappa_{d_1,d_3}^{-1}(e_{\mathsf{single},k})$ so that $\kappa_{d_1,d_3}(\tilde{e}_{\mathsf{single},k}) = \pi_{\delta_1-\delta_3}(e_{\mathsf{single},k})$. By construction of the projection map, $\tilde{e}_{\mathsf{single},k}$ is also subgaussian with variance σ_{rot}^2.

6. **Repacking:** Next, for each $j \in [n_{\mathsf{vec}}]$, the Answer algorithm computes

$$\mathbf{c}_j^{(\mathsf{repack})} = \sum_{i \in [d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \mathbf{c}_{(d_2/d_3)\cdot(j-1)+i}^{(\mathsf{proj})}.$$

Define $\rho_j \in R_{d_1,p}$ as follows:

$$\rho_j = \sum_{i \in [d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \pi_{\delta_1-\delta_3}\big(\mu_{(d_2/d_3)(j-1)+i}\big) \in R_{d_1,p}.$$

Then, we have the following:

$$\begin{aligned}\rho_j &= \sum_{i \in [d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \pi_{\delta_1-\delta_3}\big(\mu_{(d_2/d_3)(j-1)+i}\big) \\ &= \sum_{i \in [d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \kappa_{d_1,d_3}\big(\tilde{\mu}_{(d_2/d_3)(j-1)+i}\big) \qquad \text{by Eq. (D.1)} \\ &= \kappa_{d_1,d_2}\big(\Pi(\tilde{\mu}_{(d_2/d_3)(j-1)+1}, \ldots, \tilde{\mu}_{(d_2/d_3)(j-1)})\big) \qquad \text{by Lemma A.14.}\end{aligned} \tag{D.3}$$

By the linear homomorphism of RLWE encodings, this means $\mathbf{c}_j^{(\mathsf{repack})}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \rho_j$ with error

$$e_{\mathsf{repack},j} = \underbrace{\sum_{i \in [d_2/d_3]} x^{(i-1)\cdot(d_1/d_2)} \cdot \kappa_{d_1,d_3}\big(\tilde{e}_{\mathsf{single},(d_2/d_3)(j-1)+1}\big)}_{e_{\mathsf{repack},j}^{(1)}} + \underbrace{\sum_{i \in [d_2/d_3]} e_{\mathsf{proj},(d_2/d_3)(j-1)+i}}_{e_{\mathsf{repack},j}^{(2)}}.$$

Again by Lemma A.14, we know that

$$e^{(1)}_{\mathsf{repack},j} = \kappa_{d_1,d_2}\big(\Pi(\tilde{e}_{\mathsf{single},(d_2/d_3)(j-1)+1},\ldots,\tilde{e}_{\mathsf{single},(d_2/d_3)j})\big),$$

so it follows that $e^{(1)}_{\mathsf{repack},j}$ is subgaussian with variance σ^2_{rot}. Also, $e^{(2)}_{\mathsf{repack},j}$ is subgaussian with variance $(d_2/d_3)\sigma^2_{\mathsf{proj}}$. Under the independence heuristic, we conclude that each $\mathbf{c}^{(\mathsf{repack})}_j$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \rho_j$ with error variance

$$\sigma^2_{\mathsf{pack}} = \sigma^2_{\mathsf{rot}} + (d_2/d_3)\sigma^2_{\mathsf{proj}}.$$

7. **Vectorizing:** The Answer algorithm now computes

$$\mathbf{c}^{(\mathsf{vec})} \leftarrow \mathsf{Vectorize}\left(\mathsf{pp}_{\mathsf{vec}}, (\mathbf{c}^{(\mathsf{repack})}_1,\ldots,\mathbf{c}^{(\mathsf{repack})}_{n_{\mathsf{vec}}})\right).$$

Let $\boldsymbol{\rho} := [\rho_1 \mid \cdots \mid \rho_{n_{\mathsf{vec}}}]^{\mathsf{T}} \in R^{n_{\mathsf{vec}}}_{d_1,p}$. By Theorem C.2, $\mathbf{c}^{(\mathsf{vec})}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \boldsymbol{\rho}$ with error variance

$$\sigma^2_{\mathsf{vec}} = \sigma^2_{\mathsf{pack}} + n_{\mathsf{vec}} t_{\mathsf{vec}} d_1 z^2_{\mathsf{vec}} (\sigma'_{1,e})^2/4.$$

Finally, the Answer algorithm sets $\mathsf{a} \leftarrow \mathsf{Compress}(\mathsf{pp}_{\mathsf{comp}}, \mathbf{c}^{(\mathsf{vec})})$. Consider now the value of $(\mathsf{resp}_1,\ldots,\mathsf{resp}_T)$ output by $\mathsf{Extract}(\mathsf{qk},\mathsf{a})$. By construction, the Extract algorithm first computes

$$\hat{\mathbf{r}} = \begin{bmatrix} \hat{r}_1 \\ \vdots \\ \hat{r}_{n_{\mathsf{vec}}} \end{bmatrix} \leftarrow \mathsf{CompressRecover}(\mathsf{S}_2,\mathsf{a}) \in R^{n_{\mathsf{vec}}}_{d_2,p}.$$

Suppose first that

$$\hat{\mathbf{r}} = \kappa^{-1}_{d_1,d_2}(\boldsymbol{\rho}). \tag{D.4}$$

Fix some $k \in [T]$. Since $T = n_{\mathsf{vec}}(d_2/d_3)$, we can write $k = (d_2/d_3)(k_2-1)+k_1$ for some $k_1 \in [d_2/d_3]$ and $k_2 \in [n_{\mathsf{vec}}]$. Then, we have

$$\begin{aligned}
\mathsf{resp}_k &= \kappa^{-1}_{d_2,d_3}\left(x^{-(k_1-1)} \cdot \hat{r}_{k_2}\right) \\
&= \kappa^{-1}_{d_2,d_3}\left(x^{-(k_1-1)} \cdot \kappa^{-1}_{d_1,d_2}(\rho_{k_2})\right) && \text{by Eq. (D.4)} \\
&= \kappa^{-1}_{d_2,d_3}\left(x^{-(k_1-1} \cdot \kappa^{-1}_{d_1,d_2}\left(\kappa_{d_1,d_2}\left(\Pi\left(\tilde{\mu}_{(d_2/d_3)(k_2-1)+1},\ldots,\tilde{\mu}_{(d_2/d_3)k_2}\right)\right)\right)\right) && \text{by Eq. (D.3)} \\
&= \kappa^{-1}_{d_2,d_3}\left(x^{-(k_1-1)} \cdot \Pi\left(\tilde{\mu}_{(d_2/d_3)(k_2-1)+1},\ldots,\tilde{\mu}_{(d_2/d_3)k_2}\right)\right) && \text{by Lemma A.10} \\
&= \tilde{\mu}_{(d_2/d_3)(k_2-1)+k_1} && \text{by Lemma A.13} \\
&= \tilde{\mu}_k = r_{\alpha_k,\beta_k,\gamma_k} && \text{by Eq. (D.2).}
\end{aligned}$$

Thus, when Eq. (D.4) holds, the recovered response resp_k is the desired record.

Bounding the probability of Eq. (D.4). Now, we determine the probability that Eq. (D.4) holds. Let $\mathbf{z} \leftarrow \mathsf{CompressRecover}(\mathsf{S}_2,\mathsf{a}) \in R^{n_{\mathsf{vec}}}_{d_2,q_3}$. First, $\mathbf{c}^{(\mathsf{vec})}$ is an RLWE encoding of $\lfloor q_1/p \rfloor \cdot \boldsymbol{\rho}$ with error variance σ^2_{vec}. Then, by Theorem C.4, $\mathbf{z} = \lfloor q_3/p \rfloor \cdot \kappa^{-1}_{d_1,d_2}(\boldsymbol{\rho}) + \tilde{\mathbf{e}}_1 + \tilde{\mathbf{e}}_2$ where

$$\|\tilde{\mathbf{e}}_1\|_\infty \le \frac{1}{2}\left(2 + (q_3 \bmod p) + \frac{q_3}{q_1}(q_1 \bmod p)\right) = \underbrace{\frac{1}{2}\left(2 + \tfrac{q_3}{q_1}(q_1 \bmod p)\right)}_{B_{\mathsf{final}}},$$

since we assume that p divides q_3. In addition, $\tilde{\mathbf{e}}_2$ is subgaussian with variance

$$\sigma^2_{\mathrm{resp}} = \frac{q_3^2}{q_1^2}\sigma^2_{\mathrm{vec}} + \frac{q_3^2}{4q_2^2} d_1 (\sigma'_{1,s})^2 + \frac{q_3^2}{q_2^2}\sigma^2_{2,e} B^2_{\mathrm{comp}},$$

and B_{comp} is a bound on $\left\| \mathbf{g}^{-1}_{z_{\mathrm{comp}}} (\lfloor q_2/q_1 \cdot c_1^{(\mathrm{vec})} \rceil \bmod q_2) \right\|_2$, where $c_1^{(\mathrm{vec})} \in R_{d_1,q_1}$ is the first component of $\mathbf{c}^{(\mathrm{vec})}$.[8] Finally, by Theorem 2.4, Eq. (D.4) holds as long as $\|\tilde{\mathbf{e}}_1 + \tilde{\mathbf{e}}_2\|_\infty < \frac{q_3}{2p} - (q_3 \bmod p) = \frac{q_3}{2p}$. By the triangle inequality, it suffices to bound the probability that $\|\tilde{\mathbf{e}}_2\| < \frac{q_3}{2p} - B_{\mathrm{final}}$. Since $\tilde{\mathbf{e}}_2 \in R_{d_2}^{n_{\mathrm{vec}}}$ is subgaussian with variance σ^2_{resp}, we use a subgaussian tail bound together with a union bound to conclude that

$$\Pr\left[\forall k \in [T] : \mathrm{resp}_k = r_{\alpha_k,\beta_k,\gamma_k}\right] \le \Pr\left[\|\tilde{\mathbf{e}}_2\|_\infty < \frac{q_3}{2p} - B_{\mathrm{final}}\right] \le 1 - 2d_2 n_{\mathrm{vec}} \exp\left(\frac{-\pi (q_3/2p - B_{\mathrm{final}})^2}{\sigma^2_{\mathrm{resp}}}\right). \tag{D.5}$$

We can also consider the *single-query* correctness error (i.e., the probability that the record for a *specific* index $k^* \in [T]$ is correct). In this case, we only require the d_3 coefficients that determine resp_{k^*} to be correct. Thus, for any $k^* \in [T]$, we have

$$\Pr\left[\mathrm{resp}_{k^*} = r_{\alpha_{k^*},\beta_{k^*},\gamma_{k^*}}\right] \le 1 - 2d_3 \exp\left(\frac{-\pi (q_3/2p - B_{\mathrm{final}})^2}{\sigma^2_{\mathrm{resp}}}\right). \tag{D.6}$$

In our evaluation of Respire for batch queries (Section 4.3), we choose our parameters to target a fixed *single-query* error rate (specifically, a single-query error rate of at most 2^{-40}). This provides a common baseline to compare the performance for instantiations with different batch sizes.

D.2 Security of Respire

Similar to previous PIR protocols [ACLS18, AYA+21, MCR21, MW22a, MR23, LMRS24, MW24] based on the RLWE assumption, the security of Respire relies on a circular security or key-dependent message (KDM) security where RLWE encodings are pseudorandom even given encodings of functions of the secret key. We state the specific assumption we use below (adapted from [MW24]):

Definition D.1 (Key-Dependent Pseudorandomness of RLWE Encodings)**.** Let λ be a security parameter, $d = d(\lambda)$ be a power-of-two, $m = m(\lambda)$ be the number of samples, $q = q(\lambda)$ be an encoding modulus, and $\chi_s = \chi_s(\lambda)$, $\chi_e = \chi_e(\lambda)$ be error distributions over $R_d = \mathbb{Z}[x]/(x^d + 1)$. Let $\mathcal{F}$ be an efficiently-computable set of functions from $R_{d,q}$ to $R_{d,q}$. For a bit $b \in \{0,1\}$ and an adversary $\mathcal{A}$, let

$$W_b := \Pr\left[\mathcal{A}^{O(\cdot)}(1^\lambda, \mathbf{a}, \mathbf{t}_b) : \begin{array}{c} s \leftarrow \chi_s, \mathbf{a} \xleftarrow{\mathrm{R}} R^n_{d,q}, \mathbf{e} \leftarrow \chi_e^m \\ \mathbf{t}_0 = s\mathbf{a} + \mathbf{e}, \mathbf{t}_1 \xleftarrow{\mathrm{R}} R^m_{d,q} \end{array}\right],$$

where the oracle O takes as input a function $f \in \mathcal{F}$ and outputs $(a, sa + e + f(s))$ where $a \xleftarrow{\mathrm{R}} R_{d,q}$ and $e \leftarrow \chi_e$. We say that the key-dependent pseudorandomness of RLWE encodings holds with parameters $(d, m, q, \chi_s, \chi_e)$ if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$, $|W_0 - W_1| = \mathsf{negl}(\lambda)$.

Function families. The security of Respire relies on RLWE with key-dependent pseudorandomness with respect to the family of automorphisms (since the public parameters in Respire consists of encodings of automorphisms of the secret key) as well as the family of quadratic functions (since the RLWE-to-GSW conversion parameters consists of an encoding of an encoding of a quadratic function of the secret key).[9] We define the two function families we consider below:

[8]A trivial bound for B_{comp} is $\sqrt{t_{\mathrm{comp}} d_1} \cdot z_{\mathrm{comp}}/2$. When $c_1^{(\mathrm{vec})}$ is pseudorandom and $z_{\mathrm{comp}} = 2$, we can get a tighter bound on B_{comp}. We refer to Remark C.5 for more details.

[9]Note that we could also modify the scheme to use *different* keys for the RLWE and the GSW encodings. In this case, we would only need key-dependent pseudorandomness against linear functions.

Definition D.2 (Scaled Automorphisms)**.** Let $R_{d,q} = \mathbb{Z}_q[x]/(x^d + 1)$ be a polynomial ring with modulus q and dimension d. We define the family of (scaled) automorphisms over $R_{d,q}$ to be

$$\mathcal{F}_{\mathsf{auto}} \coloneqq \left\{ r \mapsto k \cdot \tau_\ell(r) : k \in \mathbb{Z}_q, \ell \in \mathbb{N} \right\},$$

where $\tau_\ell \colon R_{d,q} \to R_{d,q}$ is the Frobenius automorphism that maps $f(x) \mapsto f(x^\ell)$.

Definition D.3 (Quadratic Functions)**.** Let $R_{d,q} = \mathbb{Z}_q[x]/(x^d + 1)$ be a polynomial ring with modulus q and dimension d. We define the family of quadratic functions over $R_{d,q}$ to be

$$\mathcal{F}_{\mathsf{quad}} \coloneqq \left\{ r \mapsto \alpha_0 + \alpha_1 r + \alpha_2 r^2 : \alpha_0, \alpha_1, \alpha_2 \in \mathbb{Z}_q \right\}.$$

Security of Respire. We now give the formal security proof for the Respire protocol.

Theorem D.4 (Respire Security)**.** *Let $d_1, d_2, q_1, q_2, q_3, \chi_{1,e}, \chi_{1,s}, \chi'_{1,e}, \chi'_{1,s}, \chi_{2,e}, \chi_{2,s}$ be the lattice parameters from Construction 3.3. We also define the decomposition bases used in each of the underlying algorithms:*

- *Let z_{GSW} be the GSW decomposition base and $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q_1 \rfloor + 1$.*
- *Let $z_{\mathsf{coeff,RLWE}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}$ be the decomposition bases for query packing (Construction B.6) and $t_{\mathsf{coeff,RLWE}} = \lfloor \log_{z_{\mathsf{coeff,RLWE}}} q_1 \rfloor + 1$, $t_{\mathsf{coeff,GSW}} = \lfloor \log_{z_{\mathsf{coeff,GSW}}} q_1 \rfloor + 1$, and $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q_1 \rfloor + 1$.*
- *Let z_{proj} be the decomposition base used for projection (Construction A.7) and $t_{\mathsf{proj}} = \lfloor \log_{z_{\mathsf{proj}}} q_1 \rfloor + 1$.*
- *Let z_{vec} be the decomposition base used for vectorization (Construction C.1) and $t_{\mathsf{vec}} = \lfloor \log_{z_{\mathsf{vec}}} q_1 \rfloor + 1$.*
- *Let z_{comp} be the decomposition base use for compression (Construction C.3) and $t_{\mathsf{comp}} = \lfloor \log_{z_{\mathsf{comp}}} q_2 \rfloor + 1$.*

Let n_{vec} be the vector length used for vectorization. Let Q be a bound on the number of queries the adversary makes in the query privacy game. Suppose that the following assumptions hold:

- *Key-dependent pseudorandomness of RLWE with parameters $(d_1, 2Q, q_1, \chi_{1,s}, \chi_{1,e})$ and with respect to the family of automorphisms $\mathcal{F}_{\mathsf{auto}}$ (Definition D.2) and quadratic functions $\mathcal{F}_{\mathsf{quad}}$ (Definition D.3).*
- $\mathsf{RLWE}_{d_1, n_{\mathsf{vec}} t_{\mathsf{vec}}, q_1, \chi'_{1,s}, \chi'_{1,e}}$.
- $\mathsf{RLWE}_{d_2, k t_{\mathsf{comp}}, q_2, \chi_{2,s}, \chi_{2,e}}$*, where $k = d_1/d_2$.*

Then, Construction 3.3 satisfies query privacy for all adversaries making at most Q queries.

Proof. We start by defining a sequence of hybrid experiments, each parameterized by a bit $b \in \{0,1\}$:

- $\mathsf{Hyb}_0^{(b)}$: This is the normal query privacy experiment with bit $b \in \{0,1\}$. Namely, the challenger first samples $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda)$ and gives pp to $\mathcal{A}$. Specifically, the challenger samples $\tilde{s}_1 \leftarrow \chi_{1,s}$ and two target keys $\tilde{\mathbf{s}}'_1 \leftarrow (\chi'_{1,s})^{n_{\mathsf{vec}}}$ and $\tilde{\mathbf{s}}_2 \leftarrow \chi_{2,s}^{n_{\mathsf{vec}}}$. Define

  $$\mathbf{s}_1 = [-\tilde{s}_1 \mid 1]^{\mathsf{T}} \in R_{d_1,q_1}^2 \quad \text{and} \quad \mathbf{S}'_1 = [-\tilde{\mathbf{s}}'_1 \mid \mathbf{I}_{n_{\mathsf{vec}}}]^{\mathsf{T}} \in R_{d_1,q_1}^{(n_{\mathsf{vec}}+1)\times n_{\mathsf{vec}}} \quad \text{and} \quad \mathbf{S}_2 = [-\tilde{\mathbf{s}}_2 \mid \mathbf{I}_{n_{\mathsf{vec}}}]^{\mathsf{T}} \in R_{d_2,q_2}^{(n_{\mathsf{vec}}+1)\times n_{\mathsf{vec}}}.$$

  The challenger then samples parameters for query packing, projection, vectorization, and response packing:

  - $\mathsf{pp}_{\mathsf{qpk}} \leftarrow \mathsf{QueryPackSetup}(1^\lambda, \mathbf{s}_1)$.
  - $\mathsf{pp}_{\mathsf{proj}} \leftarrow \mathsf{ProjectSetup}(1^\lambda, \mathbf{s}_1)$.
  - $\mathsf{pp}_{\mathsf{vec}} \leftarrow \mathsf{VecSetup}(1^\lambda, \mathbf{s}_1, \mathbf{S}'_1)$.
  - $\mathsf{pp}_{\mathsf{comp}} \leftarrow \mathsf{CompressSetup}(1^\lambda, \mathbf{S}'_1, \mathbf{S}_2)$.

  Concretely, the challenger samples the following:

  – $\mathsf{pp}_{\mathsf{qpk}}$: The query compression parameters $\mathsf{pp}_{\mathsf{qpk}}$ consists of three additional sets of public parameters $\mathsf{pp}_{\mathsf{coeff,RLWE}}$, $\mathsf{pp}_{\mathsf{coeff,GSW}}$, and $\mathsf{pp}_{\mathsf{conv}}$. Specifically, for each $j \in [0, \delta_1 - 1]$, the challenger samples $\mathbf{W}_{\mathsf{coeff,RLWE},j} \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}_1, \tau_{d_1/2^j+1})$ and $\mathbf{W}_{\mathsf{coeff,GSW},j} \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}_1, \tau_{d_1/2^j+1})$. The coefficient-expansion parameters are then

$$\begin{aligned} \mathsf{pp}_{\mathsf{coeff,RLWE}} &= (\mathbf{W}_{\mathsf{coeff,RLWE},0}, \ldots, \mathbf{W}_{\mathsf{coeff,RLWE},\delta_1-1}) \\ \mathsf{pp}_{\mathsf{coeff,GSW}} &= (\mathbf{W}_{\mathsf{coeff,GSW},0}, \ldots, \mathbf{W}_{\mathsf{coeff,GSW},\delta_1-1}). \end{aligned}$$

  Finally, the challenger computes

$$\mathsf{pp}_{\mathsf{conv}} = \mathbf{V}_{\mathsf{conv}} = \begin{bmatrix} \mathbf{a}_{\mathsf{conv}}^{\mathsf{T}} \\ \tilde{s}_1 \mathbf{a}_{\mathsf{conv}}^{\mathsf{T}} + \mathbf{e}_{\mathsf{conv}}^{\mathsf{T}} - \tilde{s}_1(\mathbf{s}_1^{\mathsf{T}} \otimes \mathbf{g}_{z_{\mathsf{conv}}}^{\mathsf{T}}) \end{bmatrix} \in R_{d_1,q_1}^{2 \times 2t_{\mathsf{conv}}}.$$

  – $\mathsf{pp}_{\mathsf{proj}}$: For each $j \in [0, \delta_1 - 1]$, the challenger samples $\mathbf{W}_{\mathsf{proj},j} \leftarrow \mathsf{AutomorphSetup}(1^\lambda, \mathbf{s}_1, \tau_{d_1/2^j+1})$ and sets $\mathsf{pp}_{\mathsf{proj}} = (\mathbf{W}_{\mathsf{proj},0}, \ldots, \mathbf{W}_{\mathsf{proj},\delta_1-1})$.
  – $\mathsf{pp}_{\mathsf{vec}}$: For each $i \in [n_{\mathsf{vec}}]$, the challenger sets

$$\mathbf{V}_{\mathsf{vec},i} = \begin{bmatrix} \mathbf{a}_{\mathsf{vec},i}^{\mathsf{T}} \\ \tilde{\mathbf{s}}_1' \mathbf{a}_{\mathsf{vec},i}^{\mathsf{T}} + \mathbf{E}_{\mathsf{vec},i} - \tilde{s}_1 \mathbf{u}_i \mathbf{g}_{z_{\mathsf{vec}}}^{\mathsf{T}} \end{bmatrix} \in R_{d_1,q_1}^{(n_{\mathsf{vec}}+1) \times t_{\mathsf{vec}}}.$$

  – $\mathsf{pp}_{\mathsf{comp}}$: The challenger samples $\mathbf{a}_1, \ldots, \mathbf{a}_k \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{t_{\mathsf{comp}}}$ and $\mathbf{E}_1, \ldots, \mathbf{E}_k \leftarrow \chi_{2,e}^{n_{\mathsf{vec}} \times t_{\mathsf{comp}}}$, where $k = d_2/d_1$. It then sets

$$\mathbf{W}_{\mathsf{comp}} = \begin{bmatrix} \Pi(\mathbf{a}_1^{\mathsf{T}}, \ldots, \mathbf{a}_k^{\mathsf{T}}) \\ \Pi(\tilde{\mathbf{s}}_2 \mathbf{a}_1^{\mathsf{T}} + \mathbf{E}_1, \ldots, \tilde{\mathbf{s}}_2 \mathbf{a}_k^{\mathsf{T}} + \mathbf{E}_k) - (\tilde{\mathbf{s}}_1' \bmod q_2) \cdot \mathbf{g}_{z_{\mathsf{comp}}}^{\mathsf{T}} \end{bmatrix} \in R_{d_1,q_2}^{(n_{\mathsf{vec}}+1) \times t_{\mathsf{comp}}}.$$

  The challenger sets $\mathsf{qk} = (\mathbf{s}_1, \mathbf{S}_2)$ and $\mathsf{pp} = (\mathsf{pp}_{\mathsf{qpk}}, \mathsf{pp}_{\mathsf{proj}}, \mathsf{pp}_{\mathsf{vec}}, \mathsf{pp}_{\mathsf{comp}})$. When algorithm $\mathcal{A}$ makes a query on a pair of indices $(\mathsf{idx}_0, \mathsf{idx}_1)$, the challenger replies with $\mathsf{q} \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx}_b)$.[10] Specifically, the challenger parses $\mathsf{idx}_b = (\alpha, \beta, \gamma) \in [2^{\nu_1}] \times [2^{\nu_2}] \times [2^{\nu_3}]$, let $\hat{\alpha}_i = 1$ if $i = \alpha$ and 0 otherwise. Let $\hat{\beta}_1 \cdots \hat{\beta}_{\nu_2}$ be the binary representation of $\beta - 1$ and $\hat{\gamma}_1 \cdots \hat{\gamma}_{\nu_3}$ be the binary representation of $\gamma - 1$. It sets the query to be

$$\mathsf{q} \leftarrow \mathsf{QueryPack}(\mathbf{s}_1, (\lfloor q_1/p \rfloor \cdot \hat{\alpha}_1, \ldots, \lfloor q_1/p \rfloor \cdot \hat{\alpha}_{2^{\nu_1}}), (\hat{\beta}_1, \ldots, \hat{\beta}_{\nu_2}, \hat{\gamma}_1, \ldots, \hat{\gamma}_{\nu_3})).$$

  The query $\mathsf{q} = (\mathsf{enc}_1, \mathsf{enc}_2)$ where $\mathsf{enc}_1 = \mathbf{c}_1 \in R_{d_1,q_1}^2$ and $\mathsf{enc}_2 = \mathbf{c}_2 \in R_{d_1,q_1}^2$ are RLWE encodings under $\mathbf{s}_1$. After $\mathcal{A}$ finishes making queries, it outputs a bit $b' \in \{0, 1\}$, which is the output of the experiment.

- $\mathsf{Hyb}_1^{(b)}$: Same as $\mathsf{Hyb}_0^{(b)}$, except the challenger samples $\mathbf{W}_{\mathsf{comp}} \xleftarrow{\mathrm{R}} R_{d_1,q_2}^{(n_{\mathsf{vec}}+1) \times t_{\mathsf{comp}}}$.
- $\mathsf{Hyb}_2^{(b)}$: Same as $\mathsf{Hyb}_1^{(b)}$, except the challenger samples $\mathbf{V}_{\mathsf{vec},i} \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{(n_{\mathsf{vec}}+1) \times t_{\mathsf{vec}}}$ for all $i \in [n_{\mathsf{vec}}]$.
- $\mathsf{Hyb}_3^{(b)}$: Same as $\mathsf{Hyb}_2^{(b)}$, except the challenger samples

$$\mathbf{V}_{\mathsf{conv}} \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{2 \times 2t_{\mathsf{conv}}}, \ \mathbf{W}_{\mathsf{coeff,RLWE},i} \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{2 \times t_{\mathsf{coeff,RLWE}}}, \ \mathbf{W}_{\mathsf{coeff,GSW},i} \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{2 \times t_{\mathsf{coeff,GSW}}}, \ \mathbf{W}_{\mathsf{proj},i} \xleftarrow{\mathrm{R}} R_{d_q,q_1}^{2 \times t_{\mathsf{proj}}}.$$

  In response to each query, the challenger also samples $\mathbf{c}_1, \mathbf{c}_2 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^2$.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i^{(b)}(\mathcal{A})$ to denote the output distribution of an execution of $\mathsf{Hyb}_i^{(b)}$ with adversary $\mathcal{A}$. Since the challenger's behavior in $\mathsf{Hyb}_3^{(b)}$ is independent of the bit b, we have that for all adversaries $\mathcal{A}$, $\mathsf{Hyb}_3^{(0)}(\mathcal{A}) \equiv \mathsf{Hyb}_3^{(1)}(\mathcal{A})$. Thus, it suffices to show that each adjacent pair of distributions are computationally indistinguishable:

[10]Technically, in the batch setting, the adversary can specify two lists of T queries. However, since the real scheme generates the batch queries using T independent invocations of the single-query scheme, we can assume without loss of generality that the adversary only queries on one index at a time. The adversary can always simulate a single query on T indices using T individual queries, each on a single index.

Database Size	ν_1	ν_2	ν_3
256 MB	9	9	2
512 MB	9	10	2
1 GB	10	10	2
2 GB	10	11	2
4 GB	11	11	2
8 GB	11	12	2

Table 3: Database dimensions ν_1, ν_2, and ν_3 for Respire (Construction 3.2) as a function of the database size. Each record is 256 bytes. In all of our instantiations, we set $d_1 = 2048$, $d_2 = d_3 = 512$, $p = 2^4$, $q_1 = 268369921 \cdot 249561089 \approx 2^{56}$, $q_2 = 16760833 \approx 2^{24}$, $q_3 = 2^8$, and $\nu_3 = \log_2(d_1/d_3) = 2$.

- First $\mathsf{Hyb}_0^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$ are computationally indistinguishable under the $\mathsf{RLWE}_{d_2,kt_{\mathrm{comp}},q_2,\chi_{2,s},\chi_{2,e}}$ assumption. The only difference between these experiments is the distribution of $\mathbf{W}_{\mathrm{comp}}$. Thus, under the $\mathsf{RLWE}_{d_2,kt_{\mathrm{comp}},q_2,\chi_{2,s},\chi_{2,e}}$ assumption, we have that for $\tilde{s}_{2,i} \leftarrow \chi_{2,s}$, the distributions of

$$\tilde{s}_{2,i}\left[\mathbf{a}_1^{\mathsf{T}} \mid \cdots \mid \mathbf{a}_k^{\mathsf{T}}\right] + \left[\mathbf{e}_1^{\mathsf{T}} \mid \cdots \mid \mathbf{e}_k^{\mathsf{T}}\right],$$

  where $\mathbf{e}_i \leftarrow \chi_{2,e}^{t_{\mathrm{comp}}}$ is pseudorandom. By a hybrid argument over each component of $\tilde{s}_{2,i}$, we conclude that $\tilde{\mathbf{s}}_2\mathbf{a}_i^{\mathsf{T}} + \mathbf{E}_i$ is computationally indistinguishable from uniform for all $i \in [k]$. By definition of the ring packing function Π (Eq. (3.3) and Definition A.12), this means that $\Pi(\tilde{\mathbf{s}}_2\mathbf{a}_1^{\mathsf{T}} + \mathbf{E}_1, \ldots, \tilde{\mathbf{s}}_2\mathbf{a}_k^{\mathsf{T}} + \mathbf{E}_k)$ is computationally indistinguishable from uniform. This is the distribution in $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$.

- Hybrids $\mathsf{Hyb}_1^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ are computationally indistinguishable under the $\mathsf{RLWE}_{d_1,n_{\mathrm{vec}}t_{\mathrm{vec}},q_1,\chi'_{1,s},\chi'_{1,e}}$ assumption. By a hybrid argument (over the n_{vec} components of $\tilde{\mathbf{s}}'_1$), we have that for all $i \in [n_{\mathrm{vec}}]$, $\tilde{\mathbf{s}}'_1\mathbf{a}_{\mathrm{vec},i}^{\mathsf{T}} + \mathbf{E}_{\mathrm{vec},i}$ is computationally indistinguishable from uniform. In this case, the distribution of each $\mathbf{V}_{\mathrm{vec},i}$ is uniform over $R_{d_1,q_1}^{(n_{\mathrm{vec}}+1)\times t_{\mathrm{vec}}}$. This is the distribution in $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$.

- Hybrids $\mathsf{Hyb}_2^{(b)}(\mathcal{A})$ and $\mathsf{Hyb}_3^{(b)}(\mathcal{A})$ are computationally indistinguishable assuming key-dependent pseudorandomness of RLWE with parameters $(d_1, 2Q, q_1, \chi_{1,s}, \chi_{1,e})$ and with respect to the family of automorphisms $\mathcal{F}_{\mathrm{auto}}$ (Definition D.2) and quadratic function $\mathcal{F}_{\mathrm{quad}}$ (Definition D.3), where Q is the number of queries the adversary makes in the query privacy game. First, we observe that the matrices $\mathbf{W}_{\mathrm{coeff,RLWE},i}$, $\mathbf{W}_{\mathrm{coeff,GSW},i}$, and $\mathbf{W}_{\mathrm{proj},i}$ are matrices sampled using AutomorphSetup. From Eq. (A.1), each of these matrices is an RLWE encoding of a scaled automorphism of $\tilde{s}_1$ under $\mathbf{s}_1$. The reduction can simulate these components using the key-dependent pseudorandomness oracle (by querying on functions in $\mathcal{F}_{\mathrm{auto}}$). Next, the encodings $\mathbf{V}_{\mathrm{conv}}$ is an RLWE encoding of a quadratic function of $\tilde{s}_1$ under $\mathbf{s}_1$. Again, this can be simulated using the key-dependent pseudorandomness oracle (by querying on functions in $\mathcal{F}_{\mathrm{quad}}$). Finally, the challenger's response to each of the adversary's queries consists of two RLWE encodings $(\mathbf{c}_1, \mathbf{c}_2)$ under $\mathbf{s}_1$, which can be simulated using the RLWE challenge itself. We conclude that the output of the two distributions are computationally indistinguishable.

Since each pair of adjacent distributions are computationally indistinguishable, query privacy holds. □

E Respire Parameters

In this section, we give the concrete lattice/batching parameters we use in both the single-query version (Table 3) and the batched version (Table 4) of the Respire protocol. We also give the (shared) gadget parameters in Table 5.

Database Size	Batch Size	Hashing Parameters # Buckets B	 Bucket Size K	ν_1	ν_2	ν_3	n_{vec}
256 MB	4	7	128 MB	9	8	2	2
	8	13	64 MB	8	8	2	4
	16	25	32 MB	8	7	2	7
	32	49	16 MB	7	7	2	8
	64	98	8 MB	7	6	2	8
	128	197	4 MB	6	6	2	8
	256	398	2 MB	6	5	2	8
1 GB	4	7	512 MB	10	9	2	2
	8	13	256 MB	9	9	2	4
	16	25	128 MB	9	8	2	7
	32	49	64 MB	8	8	2	8
	64	97	32 MB	8	7	2	8
	128	194	16 MB	7	7	2	8
	256	391	8 MB	7	6	2	8

Table 4: Database dimensions ν_1, ν_2, ν_3, and hashing parameter breakdown for the batched version of Respire (Construction 3.3) as a function of the database size and the batch size. Specifically, for each batch size and database configuration, we partition the database into B buckets, each of size K (see Section 4.3 for more details of the construction). Each of the sub-databases has dimension (ν_1, ν_2, ν_3). The size of each record is fixed to be 256 bytes. In all of our instantiations, we set the lattice parameters as follows: $d_1 = d_2 = 2048$, $d_3 = 512$, $p = 2^4$, $q_1 = 268369921 \cdot 249561089 \approx 2^{56}$, $q_2 = 249857 \approx 2^{18}$, $q_3 = 2^8$, and $\nu_3 = \log_2(d_1/d_3) = 2$.

Parameters	Description	Length (t)	Base (z)
$t_{\text{GSW}}, z_{\text{GSW}}$	GSW encodings (Section 2)	8	127
$t_{\text{coeff,RLWE}}, z_{\text{coeff,RLWE}}$	RLWE encoding packing (Constructions B.1 and B.6)	4	16088
$t_{\text{coeff,GSW}}, z_{\text{coeff,GSW}}$	GSW encoding packing (Constructions B.1 and B.6)	20	7
$t_{\text{conv}}, z_{\text{conv}}$	RLWE to GSW conversion (Constructions B.4 and B.6)	4	16088
$t_{\text{proj}}, z_{\text{proj}}$	Projection (Construction A.7)	20	7
$t_{\text{vec}}, z_{\text{vec}}$	Vectorization (Construction C.1)	2	258794687
$t_{\text{comp}}, z_{\text{comp}}$	Compression (Construction C.3)	$\lfloor \log(q_2) \rfloor + 1$	2

Table 5: Respire decomposition bases (z) and decomposition lengths (t) for the underlying sub-algorithms.