SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition*

Samir Jordan Menon
Unaffiliated
menon.samir@gmail.com

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

Abstract

We introduce the SPIRAL family of single-server private information retrieval (PIR) protocols. SPIRAL relies on a composition of two lattice-based homomorphic encryption schemes: the Regev encryption scheme and the Gentry-Sahai-Waters encryption scheme. We introduce new ciphertext translation techniques to convert between these two schemes and in doing so, enable new trade-offs in communication and computation. Across a broad range of database configurations, the basic version of SPIRAL *simultaneously* achieves at least a 4.5× reduction in query size, 1.5× reduction in response size, and 2× increase in server throughput compared to previous systems. A variant of our scheme, SPIRALSTREAMPACK, is optimized for the *streaming* setting and achieves a server throughput of 1.9 GB/s for databases with over a million records (compared to 200 MB/s for previous protocols) and a rate of 0.81 (compared to 0.24 for previous protocols). For streaming large records (e.g., a private video stream), we estimate the monetary cost of SPIRALSTREAMPACK to be only 1.9× greater than that of the no-privacy baseline where the client directly downloads the desired record.

1 Introduction

A private information retrieval (PIR) [CGKS95] protocol enables a client to download an element from a public database *without* revealing to the database server which record is being requested. Beyond its direct applications to private database queries, PIR is a core building block in a wide range of privacy-preserving applications such as anonymous messaging [MOT+11, KLDF16, AS16, ACLS18], contact discovery [BDG15, DRRT18], private contact tracing [TSS+20], private navigation [FKP15, WZPM16], and safe browsing [KC21].

Private information retrieval protocols fall under two main categories: (1) multi-server protocols where the database is replicated across multiple servers [CGKS95]; and (2) single-server protocols where the database lives on a single server [KO97]. We refer to [Gas04, OS07] for excellent surveys of single-server and multi-server constructions. In many settings, multi-server constructions have reduced computational overhead compared to single-server constructions and can often achieve information-theoretic security. The drawback, however, is their reliance on having multiple *non-colluding* servers; this assumption can be challenging to realize in practice.

Conversely, single-server PIR protocols do not assume non-colluding servers. Instead, existing single-server PIR implementations have significantly higher computational costs compared to multi-server constructions. Indeed, it was believed that single-server PIR would never outperform the "trivial PIR" of simply having the client download the entire database [SC07]. While this assumption applied to earlier number-theoretic PIR schemes [KO97, CMS99, Cha04, GR05], recent lattice-based constructions [MBFK16, ACLS18, GH19, PT20, AYA+21, MCR21] have made significant strides in concrete efficiency and are much faster than the trivial PIR in many settings.

When studying PIR protocols, we are primarily interested in the (1) *rate*, which is the ratio of the response size to the size of the retrieved record; and (2) the *server throughput*, which is the ratio of the database size to the server's computation time. The rate measures the overhead in the server-to-client communication while the throughput measures how fast the server can answer a PIR query as a function of the database size. A third quantity of interest is the query size. Recent constructions are able to achieve relatively compact queries (e.g., 32–64 KB queries in the case of [ACLS18, MCR21] for databases with millions of records and tens of gigabytes of data).

*This is the extended version of a paper by the same title that appeared at IEEE Security & Privacy 2022 [MW22].

The current state-of-the-art single-server PIR, OnionPIR [MCR21], achieves a rate of 0.24 and a throughput of 149 MB/s. In contrast, the fastest two-server PIR scheme can achieve an essentially optimal rate of ≈1 and a throughput of 5.5 GB/s [HH19]. Thus, there remains a large gap between the performance of the best single-server PIR and the best two-server PIR protocols.

This work. In this work, we introduce Spiral, a new family of lattice-based single-server PIR schemes that enables new trade-offs in communication and computation. The basic instantiation of Spiral simultaneously achieves a 4.5× reduction in query size, a 1.5× increase in the rate, and a 2× increase in the server throughput compared to OnionPIR [MCR21] (see Table 2).

Like previous PIR protocols [ACLS18, GH19, PT20, ALP+21, MCR21, AYA+21], the Spiral protocol works in the model where the client starts by sending the server a set of *query-independent* public parameters. The server uses these parameters along with the client's query to compute the response. Since these parameters can be *reused* for an arbitrary number of queries and they are *independent* of the query, the client can transmit these parameters to the server in a separate "offline" phase. For this reason, we often distinguish between the offline cost of generating and communicating the public parameters and the online cost of generating the query and computing the response.

We also introduce several variants of Spiral that achieve higher server throughput and rates (i.e., reduced online cost) in exchange for larger queries and/or public parameters:

- SpiralStream: The SpiralStream protocol variant is optimized for the *streaming* setting. In the streaming setting, the client's query is *reused* across multiple databases, so we can amortize the cost of query generation and communication over multiple PIR responses. The SpiralStream protocol has larger queries (30 MB), but achieves a rate of 0.49 (2× higher than OnionPIR) and an effective server throughput of up to 1.5 GB/s (roughly 10× higher than OnionPIR). We provide more detailed benchmarks in Section 5.3 and Table 4.

- SpiralPack: The SpiralPack protocol leverages a new response packing technique that reduces the online costs of Spiral (for databases with large records) at the expense of requiring a larger set of (reusable) public parameters. As we show in Section 5.3 and Table 3, when database records are large, SpiralPack can achieve a 30% higher rate compared to Spiral while simultaneously providing a similar or higher server throughput.

The two optimizations we describe above can also be combined and we refer to the resulting protocol as SpiralStreamPack. Compared to the other Spiral variants, SpiralStreamPack has the largest public parameter and query sizes, but is able to simultaneously achieve a high rate (0.81) and a high server throughput (1.9 GB/s) on databases with over a million records. This represents a 2.1× increase in rate and 5.5× increase in throughput compared to the base version of Spiral. However, the size of the public parameters is 4.2× higher and the query size is over 2000× higher. In absolute terms, the public parameter size increases from 30 MB to 125 MB and the query size increases from 14 KB to 30 MB. We believe these remain reasonable for many streaming applications. Overall, for settings where both the public parameters and the query will be reused for a large number of queries, SpiralStreamPack likely offers the most competitive performance.

We note that for databases with sufficiently-large records (≥ 30 KB), the server throughput of our streaming constructions is 2–4× higher than that of full database encryption using a *software-based* AES implementation. We believe that this is the first single-server PIR where the server throughput is faster than applying a *symmetric* cryptographic primitive over the full database. Although this is still 2.9× slower than the best two-server PIR using *hardware-accelerated* AES [HH19], hardware acceleration for the lattice-based building blocks underlying our construction could help bridge this gap (e.g., [SFK+21]).

A limitation of Spiral is that it generally requires larger *public parameters* compared with previous schemes. To compare, the public parameters in SealPIR [ACLS18], FastPIR [AYA+21], and OnionPIR [MCR21] are 3.4 MB, 1.4 MB, and 4.6 MB, respectively. In Spiral, they range from 14 to 18 MB and for SpiralStream, they range from 344 KB to 3 MB. The larger parameters in Spiral are needed to enable our new ciphertext translation procedures (Sections 1.2 and 3) that are critical for reducing the online costs of our protocol. The SpiralPack variant requires public parameters that range from 14 to 47 MB (in order to support ciphertext packing).

1.1 Background on Lattice-Based PIR

The most efficient single-server PIR protocols [AS16, MBFK16, ACLS18, GH19, PT20, ALP+21, MCR21] use lattice-based fully homomorphic encryption (FHE) schemes [Gen09, BV11, Bra12, FV12, BGV12, GSW13].[1] These protocols follow the general paradigm of constructing PIR from homomorphic encryption [KO97]. In these protocols, the database is represented as a hypercube, and the client sends encryptions of basis vectors selecting for each dimension of the hypercube. To compute the response, the server either relies on *multiplicative homomorphism*, where the server iteratively multiplies the response for each dimension with the client's query vectors, or by using a *recursive composition* approach that only needs additive homomorphism. While earlier PIR protocols [AS16, MBFK16, ACLS18] relied on recursive composition and additive homomorphism, more recent protocols [GH19, PT20, ALP+21, MCR21] have shown how to leverage multiplicative homomorphism for better efficiency.

The challenge: ciphertext noise management. A key challenge when working with lattice-based FHE schemes is managing noise growth. In these schemes, the ciphertexts are *noisy* encodings of the plaintext messages, and homomorphic operations increase the magnitude of the noise in the ciphertext. If the noise exceeds a predetermined bound, then it is no longer possible to recover the message. The lattice parameters are chosen to ensure that the scheme can support the requisite number of operations and achieve the target level of security. Most lattice-based PIR constructions [MBFK16, AS16, ACLS18, GH19, ALP+21, MCR21] are based on either the Regev encryption scheme [Reg05], which is additively homomorphic, or its generalization, the Brakerski/Fan-Vercauteren (BFV) scheme [Bra12, FV12], which additionally supports homomorphic multiplication. In the BFV scheme, the ciphertext noise scales *exponentially* in the multiplicative depth of the computation.[2] Consequently, initial lattice-based PIR schemes did not use multiplicative homomorphism [MBFK16, AS16, ACLS18].

A solution: FHE composition. Recently, Chillotti et al. [CGGI18, CGGI20] introduced an "external product" operation to homomorphically multiply ciphertexts from two *different* schemes. They specifically show how to multiply a ciphertext encrypted under Regev's encryption scheme [Reg05] with a ciphertext encrypted under the encryption scheme of Gentry, Sahai, and Waters (GSW) [GSW13]. The requirement is that the two Regev and GSW ciphertexts are encrypted with respect to the *same* secret key.

The advantage of the GSW encryption scheme is its *asymmetric noise growth* for homomorphic multiplication. Specifically, in the setting of PIR, one of the inputs to each homomorphic multiplication is a "fresh" ciphertext (i.e., a query ciphertext). In this case, the noise growth after k sequential multiplications increases *linearly* with k rather than exponentially with k (as would be the case with BFV). The drawback of GSW ciphertexts is their poor rate: encrypting a scalar requires a large matrix. Conversely, Regev ciphertexts have much better rate; over polynomial rings, the amortized version [PVW08] can encrypt an $n \times n$ plaintext elements with a ciphertext of size $n \times (n+1)$.

The external product operation from [CGGI18, CGGI20] enables us to get the best of both worlds. Namely, if each homomorphic multiplication is between a Regev ciphertext and a *fresh* GSW ciphertext, then the noise scales additively in the number of multiplications, and the result is still a high-rate Regev ciphertext. This is the basis of the theoretical PIR construction of Gentry and Halevi [GH19] and the recently-implemented OnionPIR protocol [MCR21]. Our approach further builds upon and expands this technique of composing Regev encryption with GSW encryption to get a better handle on noise growth while enabling fast computation.

1.2 Our Contributions and Construction Overview

In this work, we present the Spiral family of single-server PIR protocols that leverages the combination of matrix Regev and GSW encryption schemes to *simultaneously* reduce the query size, response size, and the server computation time compared to all previous implemented protocols (see Section 5). Here, we provide an overview of our techniques.

[1] Technically, these constructions (including Spiral) only require *leveled* homomorphic encryption, which support a bounded number of computations. For ease of exposition, we will still write FHE to refer to leveled schemes.

[2] While it is possible to use bootstrapping [Gen09] to reduce the noise, the concrete cost of bootstrapping in the BFV encryption scheme remains high (e.g., a few minutes to refresh a *single* ciphertext) [VJH21].

High rate via ciphertext amortization. To achieve higher rate, we take the Gentry-Halevi [GH19] approach of using the amortized version of Regev encryption [PVW08] (over rings [LPR10]) as our base encryption scheme. Here, the rate of the encryption scheme (i.e., the ratio of plaintext size to ciphertext size) scales with $n^2/(n^2+n)$ where n is the plaintext dimension. Higher dimensions enable a better rate at the cost of higher server computation. For example, by using the high-rate version of Regev encryption, the base version of Spiral is able to achieve a rate that is 1.5×–6× better than OnionPIR (Table 2) on a broad range of database configurations.

Ciphertext translation and query compression. To take advantage of the Regev-GSW homomorphism, the client would have to include GSW ciphertexts as part of their query. Even with the query compression techniques of [ACLS18, CCR19], Gentry and Halevi estimate that the size of the queries in their construction to be 30 MB, which is more than 450× worse compared to existing schemes. The reason for this blowup is that the Angel et al. query compression technique [ACLS18] relies on the ability to homomorphically compute automorphisms; while this is possible on matrix Regev ciphertexts, the same does not seem to hold for GSW ciphertexts. As such, in the Gentry-Halevi construction, the client has to send multiple large GSW ciphertexts as part of its query. The OnionPIR scheme avoids this issue by observing that in the 1-dimensional case, a GSW ciphertext can be viewed as a BFV ciphertext, in which case, they can use the same type of packing approach from [ACLS18, CCR19].

In this work, we describe a general technique for translating between matrix Regev ciphertexts (of *any* dimension) and GSW ciphertexts (Section 3). Our transformations leverage the similar algebraic structure shared by Regev ciphertexts and GSW ciphertexts, and can be viewed as a particular form of key switching between two different encryption schemes. We then compose our translation algorithms with the query-packing approach from [ACLS18, CCR19], and compress our query into a single *scalar* Regev ciphertext of just 14 KB. Our query expansion procedure expands this single Regev ciphertext into a collection of *matrix* Regev ciphertexts and GSW ciphertexts encoding the client's query along each dimension of the database hypercube. More generally, our ciphertext translation protocols can be viewed as a way to "compress" GSW ciphertexts (Remark 3.3), and may be useful in other settings where users are sending/receiving GSW ciphertexts.

The Spiral family of PIR protocols. The Spiral family of PIR protocol follows a similar high-level structure as previous lattice-based PIR protocols (Section 1.1). We describe the main steps here and also visually in Fig. 1:

- **Query generation:** The client's query consists of a single *scalar* Regev ciphertext that encodes the record index the client wants to retrieve. We structure the database of $N = 2^{\nu_1 \times \nu_2}$ records as a $2^{\nu_1} \times 2 \times \cdots \times 2$ hypercube. A record index can then be described by a tuple $(i, j_1, \ldots, j_{\nu_2})$ where $i \in \{0, \ldots, 2^{\nu_1} - 1\}$ and $j_1, \ldots, j_{\nu_2} \in \{0, 1\}$. The query consists of an encoding of the vector $(i, j_1, \ldots, j_{\nu_2})$, which we can pack into a single scalar Regev ciphertext using the Angel et al. [ACLS18] technique.
- **Query expansion:** Upon receiving the client's query, the server expands the query ciphertext as follows:
  - **Initial expansion:** The server starts by applying the expansion technique from [ACLS18] to expand the query into a collection of (scalar) Regev ciphertexts that encode the queried index $(i, j_1, \ldots, j_{\nu_2})$. This yields two collections of Regev ciphertexts, which we will denote by C_{Reg} and C_{GSW}.
  - **First dimension expansion:** Next, the server uses C_{Reg} to expand the ciphertexts into a collection of 2^{ν_1} *matrix* Regev ciphertexts that "indicate" index i: namely, the i^{th} ciphertext is an encryption of 1 while the remaining ciphertexts are encryptions of 0. We can view this collection of ciphertexts as an encryption of the i^{th} basis vector. This step relies on a scalar-to-matrix algorithm ScalToMat that takes a Regev ciphertext encrypting a bit $\mu \in \{0, 1\}$ and outputs a matrix Regev ciphertext that encrypts the matrix $\mu \mathbf{I}_n$, where $\mathbf{I}_n$ is the n-by-n identity matrix. We describe this construction in Section 3.1.
  - **GSW ciphertext expansion:** The server then uses C_{GSW} to construct GSW encryptions of the indices $j_1, \ldots, j_{\nu_2} \in \{0, 1\}$. This step relies on a Regev-to-GSW translation algorithm RegevToGSW that we describe in Section 3.2.

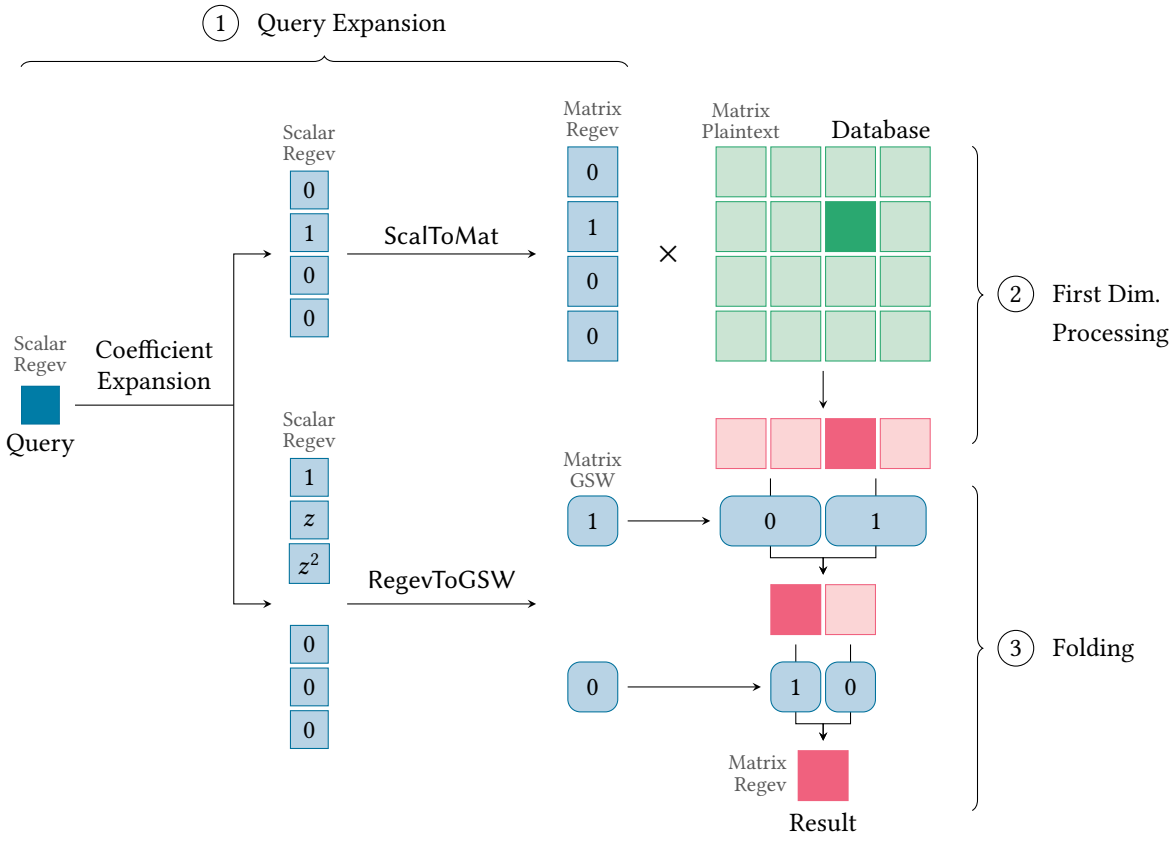

Figure 1: Server processing for a single SPIRAL query. The parameter z here is a decomposition base and is used for the translation between Regev ciphertexts and GSW ciphertexts. We refer to Section 3.2 for more details.

- **Query processing:** After expanding the query into matrix Regev encryptions of the first dimension and GSW encryptions of the subsequent dimension, the server follows the Gentry-Halevi blueprint [GH19] and homomorphically computes the response as follows:
  - **First dimension processing:** First, it uses the matrix Regev encryptions of the i^{th} basis vector to project the database onto the sub-database of records whose first index is i. This step only requires linear homomorphisms since the database records are available in the clear while the query is encrypted. At the end of this step, the server has matrix Regev encryptions of the projected database.
  - **Folding in subsequent dimensions:** Next, the server uses the Regev-GSW external product to homomorphically multiply in the GSW ciphertexts encrypting the subsequent queries. Each GSW ciphertext selects for one of two possible dimensions. Since each multiplication involves a "fresh" GSW ciphertext derived from the original query, we can take advantage of the asymmetric noise growth property of Regev-GSW multiplication. The result is a single matrix Regev ciphertext encrypting the desired record.
- **Response decoding:** At the conclusion of the above protocol, the server replies with a single *matrix* Regev ciphertext encrypting the desired record.

We provide the full protocol description in Section 4 and a high-level illustration in Fig. 1.

SPIRALPACK: New trade-offs via response packing. Using matrix Regev ciphertexts for the bulk of the computation improves the rate of the protocol but does incur some computational overhead (from the need to operate on

matrices rather than scalars). The SpiralPack protocol is a variant of Spiral that allows the server to simultaneously operate on *scalar* Regev ciphertexts while retaining the high rate benefits of using matrix Regev ciphertexts. In particular, we show how to adapt our ciphertext translation techniques to *pack* multiple *scalar* Regev ciphertexts into a single *matrix* Regev ciphertext. The server processing then operates on 1-dimensional ciphertexts, while the response consists of n-dimensional ciphertexts. The main cost of this packing procedure is it requires a larger set of public parameters. We describe the construction details in Section 4.1.

Automated parameter selection. Our design introduces multiple tunable parameters that allow us to explore new trade-offs between server computation, query size, and response size. Since the overall server computation and communication of our PIR protocol is a complex function of the underlying parameters of our scheme, we introduce an automatic parameter selection procedure that takes as input a database configuration (i.e., number of records and record size), and systematically searches through the space of possible parameters to minimize the server cost. A similar approach was also used in the XPIR system [MBFK16]. We describe our parameter selection methodology in Section 5.1. Our system allows choosing parameters that either minimizes the estimated cost of the protocol (based on the costs associated with server computation and communication), or focuses solely on maximizing either the server throughput or rate. The system also supports selecting parameters that satisfy a constraint on the query size or the public parameter size. The parameter selection tool searches over candidate parameter sets for all of the Spiral variants described in this paper and selects the system that best achieves the target objective.

Performance evaluation and trade-offs. Finally, we provide a complete implementation of Spiral and a detailed experimental analysis and comparison with previous PIR protocols. We provide the full evaluation and accompanying microbenchmarks in Section 5.3. We also estimate the concrete monetary costs of applying the Spiral family of protocols to support several privacy-preserving applications. For instance, based on current cloud computing costs, we show that SpiralStreamPack enables a user to privately stream a 2 GB movie from a library of 2^{14} movies with a server cost of \$0.34, which is just 1.9× higher than the no-privacy baseline (where the client directly downloads the movie of interest). This is a 9× reduction in cost compared to the previous state of the art, OnionPIR [MCR21].

2 Preliminaries

We write λ to denote the security parameter. For a positive integer $n \in \mathbb{N}$, we write $[n]$ to denote the set $\{1, \ldots, n\}$. For integers $a, b \in \mathbb{Z}$, we write $[a, b]$ to denote the set $\{a, a+1, \ldots, b\}$. For a positive integer $q \in \mathbb{N}$, we write $\mathbb{Z}_q$ to denote the integers modulo q. We use bold uppercase letters to denote matrices (e.g., $\mathbf{A}, \mathbf{B}$) and bold lowercase letters to denote vectors (e.g., $\mathbf{u}$, $\mathbf{v}$). We write $\mathsf{poly}(\lambda)$ to denote a function that is $O(\lambda^c)$ for some $c \in \mathbb{N}$ and $\mathsf{negl}(\lambda)$ to denote a function that is $o(\lambda^{-c})$ for all $c \in \mathbb{N}$. An algorithm is efficient if it runs in probabilistic polynomial time in its input length. We say that two families of distributions $\mathcal{D}_1 = \{\mathcal{D}_{1,\lambda}\}_{\lambda \in \mathbb{N}}$ and $\mathcal{D}_2 = \{\mathcal{D}_{2,\lambda}\}_{\lambda \in \mathbb{N}}$ are computationally indistinguishable if no efficient algorithm can distinguish them with non-negligible probability. We denote this by writing $\mathcal{D}_1 \overset{c}{\approx} \mathcal{D}_2$. We say a distribution $\mathcal{D}$ is B-bounded if $\Pr[|x| \leq B : x \leftarrow \mathcal{D}] = 1$.

Discrete Gaussians and tail bounds. We recall the definition of the discrete Gaussian distribution and basic facts about subgaussian distributions (see [Pei16] for more details). For a real value $\sigma > 0$, the Gaussian function $\rho_\sigma : \mathbb{R} \to \mathbb{R}^+$ with width σ is the function $\rho_\sigma(x) := \exp(-\pi x^2/\sigma^2)$. The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}$ with mean 0 and width σ is the distribution where

$$\Pr[X = x : X \leftarrow D_{\mathbb{Z},\sigma}] = \frac{\rho_\sigma(x)}{\sum_{y \in \mathbb{Z}} \rho_\sigma(y)}. \tag{2.1}$$

Definition 2.1 (Subgaussian Random Variable)**.** A real random variable X is *subgaussian* with parameter σ if for every $t \geq 0$, $\Pr[|X| > t] \leq 2\exp(-\pi t^2/\sigma^2)$.

Fact 2.2 (Subgaussian Random Variables)**.** Subgaussian random variables satisfy the following properties:

- If X is subgaussian with parameter σ and $a \in \mathbb{R}$, then aX is subgaussian with parameter $|a|\,\sigma$.

- If $X_1, \ldots, X_m$ are independent subgaussian random variables with parameters $\sigma_1, \ldots, \sigma_m$, respectively, $\sum_{i \in [m]} X_i$ is subgaussian with parameter $\|\boldsymbol{\sigma}\|_2$ where $\boldsymbol{\sigma} = (\sigma_1, \ldots, \sigma_m)$.

Private information retrieval. We now recall the standard definition of a two-message single-server PIR protocol [KO97]. Like most lattice-based PIR schemes [ACLS18, GH19, PT20, AYA+21, ALP+21, MCR21], we allow for an initial *query-independent* and *database-independent* setup protocol that outputs a query key qk (known to the client) and a set of public parameters pp (known to both the client and the server). The same pp and qk can be reused by the client and server for multiple queries, so we can *amortize* the cost of the setup phase over many PIR queries. Note that we can also obtain a standard 2-message PIR protocol *without* setup by having the query algorithm generate qk and pp and including pp as part of its query.

Definition 2.3 (Two-Message Single-Server PIR [KO97, adapted])**.** A two-message single-server private information retrieval (PIR) scheme $\Pi_{\mathsf{PIR}} = (\mathsf{Setup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$ is a tuple of efficient algorithms with the following properties:

- $\mathsf{Setup}(1^\lambda, 1^N) \to (\mathsf{pp}, \mathsf{qk})$: On input the security parameter λ and a bound on the database size N, the setup algorithm outputs a query key qk and a set of public parameters pp.
- $\mathsf{Query}(\mathsf{qk}, \mathsf{idx}) \to (\mathsf{st}, \mathsf{q})$: On input the query key qk and an index idx, the query algorithm outputs a state st and a query q.
- $\mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q}) \to \mathsf{r}$: On input the public parameters pp, a database $\mathcal{D} = \{d_1, \ldots, d_N\}$, and a query q, the answer algorithm outputs a response r.
- $\mathsf{Extract}(\mathsf{qk}, \mathsf{st}, \mathsf{r}) \to d_i$: On input the query key qk, the state st, and a response r, the extract algorithm outputs a database record d_i.

The algorithms should satisfy the following properties:

- **Correctness:** For all $\lambda \in \mathbb{N}$, all polynomials $N = N(\lambda), \ell = \ell(\lambda)$, and all databases $\mathcal{D} = \{d_1, \ldots, d_N\}$ where each $d_i \in \{0,1\}^\ell$, and all indices $\mathsf{idx} \in [N]$,

$$\Pr[\mathsf{Extract}(\mathsf{qk}, \mathsf{st}, \mathsf{r}) = d_i] = 1,$$

  where $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^N)$, $(\mathsf{st}, \mathsf{q}) \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx})$, and $\mathsf{r} \leftarrow \mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q})$.

- **Query privacy:** For all polynomials $N = N(\lambda)$ and all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left| \Pr\left[\mathcal{A}^{\mathcal{O}_b(\mathsf{qk}, \cdot, \cdot)}(1^\lambda, \mathsf{pp}) = b \right] - \frac{1}{2} \right| = \mathsf{negl}(\lambda),$$

  where $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^N)$, $b \xleftarrow{\mathrm{R}} \{0,1\}$, and the oracle $\mathcal{O}_b(\mathsf{qk}, \mathsf{idx}_0, \mathsf{idx}_1)$ outputs $\mathsf{Query}(\mathsf{qk}, \mathsf{idx}_b)$. This definition captures *reusability* of pp and qk.

CPA-secure encoding. We also recall the notion of CPA security. While this notion is typically defined for encryption schemes, we define it in the simpler setting of an encoding scheme (i.e., an encryption scheme without an explicit decryption algorithm).

Definition 2.4 (CPA-Secure Encoding)**.** A (secret-key) encoding scheme for a message space $\mathcal{M}$ is a pair of two efficient algorithms $(\mathsf{KeyGen}, \mathsf{Encode})$ with the following properties:

- $\mathsf{KeyGen}(1^\lambda) \to \mathsf{sk}$: On input the security parameter $\lambda \in \mathbb{N}$, the key-generation algorithm outputs a secret encoding key sk.

- $\mathsf{Encode}(\mathsf{sk}, m) \to c_m$: On input the secret encoding key sk and a message $m \in \mathcal{M}$, the encode algorithm outputs an encoding c_m.

We say an encoding scheme is CPA-secure if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr\left[\mathcal{A}^{\mathcal{O}_b(\mathsf{sk},\cdot,\cdot)}(1^\lambda) = b\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda), \tag{2.2}$$

where $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$, $b \xleftarrow{R} \{0,1\}$, and $\mathcal{O}_b(\mathsf{sk}, m_0, m_1)$ outputs $\mathsf{Encode}(\mathsf{sk}, m_b)$. We say that the scheme is Q-query secure if Eq. (2.2) holds against all efficient adversaries making up to Q queries.

Key-dependent message security. Like all FHE schemes that support key switching or bootstrapping [Gen09, BV11, BGV12, Bra12, FV12, GHS12a, GSW13], we require CPA security to hold even given encodings of (certain) functions of the secret key. We formally capture this using the following notion of key-dependent message (KDM) security, adapted from [BRS02].

Definition 2.5 (Key-Dependent Message Security [BRS02, adapted])**.** Let $\Pi = (\mathsf{KeyGen}, \mathsf{Encode})$ be an encoding scheme over a message space $\mathcal{M}$. Let $\mathcal{F}$ be a set of efficiently-computable functions with output space $\mathcal{M}$. We say that Π satisfies $\mathcal{F}$-key-dependent message security ($\mathcal{F}$-KDM security) if for all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that for all $\lambda \in \mathbb{N}$,

$$\left|\Pr\left[\mathcal{A}^{\mathcal{O}_b(\mathsf{sk},\cdot)}(1^\lambda) = b\right] - \frac{1}{2}\right| = \mathsf{negl}(\lambda),$$

where $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$, $b \xleftarrow{R} \{0,1\}$, and on input $f \in \mathcal{F}$, oracle $\mathcal{O}_b$ samples $r \xleftarrow{R} \mathcal{M}$ and responds with $\mathsf{Encode}(\mathsf{sk}_i, f(\mathsf{sk}))$ if $b = 0$ and $\mathsf{Encode}(\mathsf{sk}_i, r)$ if $b = 1$.

2.1 Lattice-Based Homomorphic Encodings

Like previous lattice-based PIR protocols [MBFK16, ACLS18, GH19, ALP+21, MCR21, AYA+21], Spiral operates over cyclotomic rings $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. For a positive integer $q \in \mathbb{N}$, we write $R_q = R/qR$. For a polynomial $f \in R$, we write $\|f\|_\infty$ to denote the ℓ_∞ norm of the vector of coefficient of f over $\mathbb{Z}^d$. We say that $f \in R$ is sampled from a subgaussian distribution with parameter σ if the coefficients of f are sampled from independent subgaussian distributions, each with parameter σ. We write γ_R to denote the ring expansion constant associated with the ring R. Namely, for all $f, g \in R$, we have that $\|fg\|_\infty \leq \gamma_R \|f\|_\infty \|g\|_\infty$. When $R = \mathbb{Z}[x]/(x^d + 1)$ is a power-of-two cyclotomic ring, $\gamma_R = d$ [DPSZ12]. When the coefficients of one of the polynomials f are sampled from a subgaussian distribution and the coefficients of the other polynomial g are sampled from an independent B-bounded distribution, then the distribution of coefficients in the product polynomial $fg \in R$ is also subgaussian:

Lemma 2.6 (Polynomials with Subgaussian Coefficients)**.** *Let $R = \mathbb{Z}[x]/(x^d+1)$. Let $f = \sum_{i\in[0,d-1]} f_i x^i$ be a polynomial where each f_i is independently sampled from a subgaussian distribution with parameter σ and $g = \sum_{i\in[0,d-1]} g_i x^i$ be a polynomial where each g_i is independently sampled from a B-bounded distribution. Let $h = fg = \sum_{i\in[0,d-1]} h_i x^i \in R$. Then, for all $i \in [0, d-1]$, the distribution of h_i is subgaussian with parameter $\sqrt{d}B\sigma$. In addition, if the polynomial g has at most k non-zero coefficients, where each non-zero coefficient is sampled from independent B-bounded distributions, then the distribution of h_i is subgaussian with parameter $\sqrt{k}B\sigma$.*

Proof. Over the ring R, we have that

$$h_i = \sum_{j,k:\ j+k=i \bmod d} (-1)^{c_{j,k}} f_j g_k = \sum_{j\in[0,d-1]} (-1)^{c'_j} f_j g_{i-j \bmod d},$$

for some $c_{j,k}, c'_j \in \{0,1\}$. Let $z_j = f_j g_{i-j \bmod d}$. By construction, z_j is either 0 (if $g_{i-j \bmod d} = 0$) or subgaussian with parameter $B\sigma$. Since the f_j's and g_k's are sampled independently, the z_j's are also independent. Thus, if g has k non-zero coefficients, then each h_i is subgaussian with parameter $\sqrt{k}B\sigma$. The main statement corresponds to the case $k = d$. □

Ring learning with errors. We now recall the ring learning with errors problem [LPR10]. We state the assumption in normal form where the RLWE secret is sampled from the error distribution. Applebaum et al. [ACPS09] showed that this version is as hard as the standard version of the RLWE assumption where the secret is sampled from the uniform distribution.

Definition 2.7 (Ring Learning with Errors (RLWE) [LPR10, ACPS09])**.** Let λ be a security parameter, and let $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = d(\lambda)$ is a power of 2. Let $m = m(\lambda)$ be the number of samples, $q = q(\lambda)$ be a ring modulus, and $\chi = \chi(\lambda)$ be an error distribution over R_q. The decisional ring learning with errors (RLWE) assumption $\mathsf{RLWE}_{d,m,q,\chi}$ in Hermite normal form states that for $\mathbf{a} \xleftarrow{R} R_q^m$, $s \leftarrow \chi$, $\mathbf{e} \leftarrow \chi^m$, and $\mathbf{u} \xleftarrow{R} R_q^m$, the following two distributions are computationally indistinguishable:

$$(\mathbf{a}, s\mathbf{a} + \mathbf{e}) \text{ and } (\mathbf{a}, \mathbf{u}).$$

Gadget matrices. In this work, we use gadget matrices with *different* bases [MP12]. Fix a dimension $n \in \mathbb{N}$, and a base $z \in \mathbb{N}$. Let $\mathbf{g}_z^\mathsf{T} = [1, z, z^2, \ldots, z^{\lfloor \log_z q \rfloor}] \in R_q^t$ where $t = \lfloor \log_z q \rfloor + 1$. We define the gadget matrix $\mathbf{G}_{n,z} = \mathbf{I}_n \otimes \mathbf{g}_z^\mathsf{T} \in R_q^{n \times m}$, where $m = nt$. We write $\mathbf{g}_z^{-1} \colon R_q \to R_q^t$ to denote the function that expands the input into a base-z representation where each digit is in the range $[-z/2, z/2]$.[3] We write $\mathbf{G}_{n,z}^{-1} \colon R_q^n \to R_q^m$ to denote the function that applies $\mathbf{g}_z^{-1}$ to each component of the input vector, and extend $\mathbf{G}_{n,z}^{-1}$ to operate on matrices $\mathbf{M}$ by applying $\mathbf{G}_{n,z}^{-1}$ to each column of $\mathbf{M}$.

Regev encoding scheme for matrices. Spiral uses the matrix version of Regev encryption over rings [Reg05, PVW08, LPR10]. When describing our construction, it is more convenient to view Regev encryption as a *noisy encoding* scheme over R_q, which does *not* support decryption for all encoded values. If we apply a *redundant* encoding of a message (i.e., scaling the message as in standard Regev encryption) then it is possible to recover the encoded value (see Definition 2.10 and Theorem 2.11).

Construction 2.8 (Matrix Regev Encoding [Reg05, PVW08, LPR10, adapted])**.** Let $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = d(\lambda)$ is a power of two. Let $n = n(\lambda)$ be the dimension, $\chi = \chi(\lambda)$ be an error distribution over R_q, and $q = q(\lambda)$ be the encoding modulus. The matrix encoding scheme for matrices in $R_q^{n \times n}$ is defined as follows:

- $\mathsf{KeyGen}(1^\lambda, 1^n)$: On input the security parameter λ and the message dimension n, sample $\tilde{\mathbf{s}} \leftarrow \chi^n$ and output $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1) \times n}$.
- $\mathsf{Encode}(\mathbf{S}, \mathbf{M})$: On input the secret key $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T}$ and a matrix $\mathbf{M} \in R_q^{n \times n}$, sample $\mathbf{a} \xleftarrow{R} R_q^n$, $\mathbf{E} \leftarrow \chi^{n \times n}$ and output the encoding[4]
$$\mathbf{C} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}\mathbf{a}^\mathsf{T} + \mathbf{E} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1 \times n} \\ \mathbf{M} \end{bmatrix} \in R_q^{(n+1) \times n}.$$

In addition, we define the following homomorphic operations supported by the Regev encoding scheme:

- $\mathsf{Add}(\mathbf{C}_1, \mathbf{C}_2)$: On input two encodings $\mathbf{C}_1, \mathbf{C}_2 \in R_q^{(n+1) \times n}$, output $\mathbf{C}_1 + \mathbf{C}_2 \in R_q^{(n+1) \times n}$.
- $\mathsf{ScalarMul}(\mathbf{C}, \mathbf{T})$: On input an encoding $\mathbf{C} \in R_q^{(n+1) \times n}$ and a matrix $\mathbf{T} \in R_q^{n \times n}$, output $\mathbf{CT} \in R_q^{(n+1) \times n}$.

Definition 2.9 (Regev Encoding Structure)**.** Let $\mathbf{S} \in R_q^{(n+1) \times n}$ be a Regev secret key. We say that $\mathbf{C} \in R_q^{(n+1) \times n}$ is a Regev encoding of $\mathbf{M} \in R_q^{n \times n}$ with error $\mathbf{E} \in R^{n \times n}$ if $\mathbf{S}^\mathsf{T}\mathbf{C} = \mathbf{M} + \mathbf{E}$. Note that an encoding $\mathbf{C}$ can be associated with many possible message/error pairs $(\mathbf{M}, \mathbf{E})$.

[3] One way to obtain this representation is to first write the input in the standard base-z representation, and then observe that we can always rewrite $k \cdot z^i$ as $z^{i+1} - (z-k)z^i$. At least one of k and $z-k$ will be in the interval $[-z/2, z/2]$. In our settings, $2q < z^{\lfloor \log_z q \rfloor + 1}$, so this approach does not increase the dimension of the decomposition.

[4] We can alternatively define Encode to encode a single vector in R_q^n. In this case, we sample $a \xleftarrow{R} R_q$ and $\mathbf{e} \leftarrow \chi^n$ and output an encoding $\mathbf{c} \in R_q^{n+1}$. Then, an encoding of a matrix $\mathbf{M} \in R_q^{n \times n}$ is just the concatenation of n independent encodings, one for each column of $\mathbf{M}$. In this way, we can generalize this encoding algorithm to encode messages in $R_q^{n \times k}$ for arbitrary k. Our key-switching matrices in Section 3 will rely on encoding rectangular matrices.

Definition 2.10 (Regev Decoding)**.** Let $p = p(\lambda)$ be a plaintext modulus. We define the following decoding algorithm:

- $\mathsf{Decode}(\mathbf{Z})$: On input $\mathbf{Z} \in R_q^{n\times n}$, output $\lfloor p/q \cdot \mathbf{Z} \rceil$, where $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. Both the scaling and rounding are performed over the rationals.

Theorem 2.11 (Decoding Correctness)**.** *Suppose* $\mathbf{Z} = \lfloor q/p \rfloor \cdot \mathbf{M} + \mathbf{E} \in R_q^{n\times n}$ *where* $\mathbf{M} \in R_p^{n\times n}$ *and* $\mathbf{E} \in R_q^{n\times n}$*. If* $\|\mathbf{E}\|_\infty + (q \bmod p) < q/2p$*, then* $\mathsf{Decode}(\mathbf{Z}) = \mathbf{M}$.

Proof. Since $\|\mathbf{M}\|_\infty < p$ and $\|\mathbf{E}\|_\infty < q/2p$, the relation $\mathbf{Z} = \lfloor q/p \rfloor \cdot \mathbf{M} + \mathbf{E}$ holds over R (not just R_q). Then, over the rationals,

$$\frac{p}{q} \cdot \mathbf{Z} = \frac{p}{q}\left(\left\lfloor \frac{q}{p} \right\rfloor \mathbf{M} + \mathbf{E}\right) = \mathbf{M} - \underbrace{\frac{q \bmod p}{q}\mathbf{M} + \frac{p}{q}\mathbf{E}}_{\mathbf{E}'}.$$

since $\lfloor q/p \rfloor = q/p - (q \bmod p)/p$. Since $\|\mathbf{M}\|_\infty < p$,

$$\|\mathbf{E}'\|_\infty = \left\|\frac{p}{q}\mathbf{E} - \frac{q \bmod p}{q}\mathbf{M}\right\|_\infty < \frac{p}{q}\left(\|\mathbf{E}\|_\infty + (q \bmod p)\right) < \frac{1}{2}.$$

Since $\|\mathbf{E}'\|_\infty < 1/2$, we have that $\lfloor p/q \cdot \mathbf{Z} \rceil = \lfloor \mathbf{M} + \mathbf{E}' \rceil = \mathbf{M}$. □

Theorem 2.12 (Homomorphic Operations for Matrix Encodings)**.** *Let* $R = \mathbb{Z}[x]/(x^d + 1)$ *where* d *is a power of two. Suppose that* $\mathbf{C}_1, \mathbf{C}_2 \in R_q^{(n+1)\times n}$ *are Regev encodings of* $\mathbf{M}_1, \mathbf{M}_2 \in R_q^{n\times n}$ *under* $\mathbf{S} \in R_q^{(n+1)\times n}$ *with errors* $\mathbf{E}_1, \mathbf{E}_2 \in R^{n\times n}$*, respectively. Then, the following hold:*

- ***Addition:*** *Let* $\mathbf{C} \leftarrow \mathsf{Add}(\mathbf{C}_1, \mathbf{C}_2)$*. Then,* $\mathbf{C}$ *is an encoding of* $\mathbf{M}_1 + \mathbf{M}_2$ *with error* $\mathbf{E}$ *where* $\|\mathbf{E}\|_\infty \le \|\mathbf{E}_1\|_\infty + \|\mathbf{E}_2\|_\infty$*. If* $\mathbf{E}_1$ *and* $\mathbf{E}_2$ *are independent and subgaussian with parameters* σ_1 *and* σ_2*, respectively, then* $\mathbf{E}$ *is subgaussian with parameter* $\sigma = \sqrt{\sigma_1^2 + \sigma_2^2}$.
- ***Scalar multiplication:*** *Take any* $\mathbf{T} \in R_q^{n\times n}$ *and let* $\mathbf{C} \leftarrow \mathsf{ScalarMul}(\mathbf{C}_1, \mathbf{T})$*. Then,* $\mathbf{C}$ *is an encoding of* $\mathbf{M}_1\mathbf{T} \in R_q^{n\times n}$ *with error* $\mathbf{E}$ *where* $\|\mathbf{E}\|_\infty = \|\mathbf{E}_1\mathbf{T}\|_\infty \le dn\,\|\mathbf{T}\|_\infty \|\mathbf{E}_1\|_\infty$*. If* $\mathbf{E}_1$ *is subgaussian with parameter* σ_1*, then* $\mathbf{E}$ *is subgaussian with parameter* $\sigma = \sqrt{nd}\,\|\mathbf{T}\|_\infty \sigma_1$.

  Moreover, if every element of $\mathbf{T}$ *is a monomial (i.e., an element of the form* $t_{ij} \cdot x^u$ *for some* $t_{ij} \in \mathbb{Z}_q$ *and* $u \in \mathbb{Z}$*), then* $\|\mathbf{E}\|_\infty \le n\,\|\mathbf{T}\|_\infty \|\mathbf{E}_1\|_\infty$*. Likewise, if* $\mathbf{E}_1$ *is subgaussian with parameter* σ_1*, then* $\mathbf{E}$ *is subgaussian with parameter* $\sqrt{n}\,\|\mathbf{T}\|_\infty \sigma_1$.

Proof. By definition, $\mathbf{S}^\mathsf{T}\mathbf{C}_i = \mathbf{M}_i + \mathbf{E}_i$ for $i \in \{1, 2\}$. We analyze each operation individually:

- **Addition:** By definition, $\mathbf{C} = \mathbf{C}_1 + \mathbf{C}_2$. Then, $\mathbf{S}^\mathsf{T}\mathbf{C} = (\mathbf{M}_1 + \mathbf{M}_2) + (\mathbf{E}_1 + \mathbf{E}_2)$, so $\mathbf{C}$ is an encoding of $\mathbf{M}_1 + \mathbf{M}_2$ with error $\mathbf{E}_1 + \mathbf{E}_2$.
- **Scalar multiplication:** By definition, $\mathbf{C} = \mathbf{C}_1\mathbf{T}$ so $\mathbf{S}^\mathsf{T}\mathbf{C} = \mathbf{M}_1\mathbf{T} + \mathbf{E}_1\mathbf{T}$. Thus, $\mathbf{C}$ is an encoding of $\mathbf{M}_1\mathbf{T}$ with error $\mathbf{E}_1\mathbf{T}$. The claim (and the special case) follow from Lemma 2.6. □

Theorem 2.13 (CPA Security [Reg05, PVW08, LPR10])**.** *Fix a security parameter* λ *and take any polynomial* $Q = Q(\lambda)$*. Let* $R = \mathbb{Z}[x]/(x^d + 1)$ *where* $d = d(\lambda)$ *is a power of two. Let* $n = n(\lambda)$*,* $\chi = \chi(\lambda)$*, and* $q = q(\lambda)$ *be scheme parameters. Then, under the* $\mathsf{RLWE}_{d,nQ,q,\chi}$ *assumption, the Regev encoding scheme (Construction 2.8) is* Q*-query CPA-secure.*

GSW encoding scheme. We now describe the encryption scheme of Gentry, Sahai, and Waters [GSW13]. Since we do not require the ability to decrypt ciphertexts, we again present it as an encoding scheme. We refer to [GSW13] for details on how to implement decryption.

Construction 2.14 (GSW Encoding [GSW13])**.** Let $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = d(\lambda)$ is a power of two. Let $n = n(\lambda)$ be the dimension and $q = q(\lambda)$ be the encoding modulus. Let $z \in \mathbb{N}$ be a decomposition base, let $t = \lfloor \log_z q \rfloor + 1$ and let $m = (n + 1)t$. The GSW encoding scheme for messages in R_q is defined as follows:

- KeyGen(1^λ): Sample $\tilde{\mathbf{s}} \leftarrow \chi^n$ and output $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$.
- Encode($\mathbf{S}, \mu$): On input a secret key $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T}$ and a message $\mu \in R_q$, sample $\mathbf{a} \xleftarrow{\text{R}} R_q^m$, $\mathbf{E} \leftarrow \chi^{n\times m}$ and output the encoding

$$\mathbf{C} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}\mathbf{a}^\mathsf{T} + \mathbf{E} \end{bmatrix} + \mu \cdot \mathbf{G}_{n+1,z} \in R_q^{(n+1)\times m}$$

Definition 2.15 (GSW Encoding Error)**.** Let $z \in \mathbb{N}$ be a decomposition base, $t = \lfloor \log_z q \rfloor + 1$, and $m = (n+1)t$. Let $\mathbf{S} \in R_q^{(n+1)\times n}$ be a GSW secret key. We say that $\mathbf{C} \in R_q^{(n+1)\times m}$ is a GSW encoding of $\mu \in R_p$ with error $\mathbf{E} \in R_q^{n\times m}$ if $\mathbf{S}^\mathsf{T}\mathbf{C} = \mu\mathbf{S}^\mathsf{T}\mathbf{G}_{n+1,z} + \mathbf{E}$.

Remark 2.16 (Complementing GSW Encodings)**.** If $\mathbf{C} \in R_q^{(n+1)\times m}$ is a GSW encoding of a bit $b \in \{0,1\}$ with respect to error $\mathbf{E}$ and a decomposition base z, then we define $\text{Complement}(\mathbf{C}) \coloneqq \mathbf{G}_{n+1,z} - \mathbf{C}$. By construction, $\text{Complement}(\mathbf{C})$ is a GSW encoding of $1-b \in \{0,1\}$ with error $-\mathbf{E}$.

Theorem 2.17 (CPA Security [GSW13])**.** *Fix a security parameter λ and take any polynomial $Q = Q(\lambda)$. Let $R = \mathbb{Z}[x]/(x^d+1)$ where $d = d(\lambda)$ is a power of two. Let $n = n(\lambda)$, $\chi = \chi(\lambda)$, and $q = q(\lambda)$ be scheme parameters. Under the $\mathsf{RLWE}_{d,mQ,q,\chi}$ assumption, the GSW encoding scheme (Construction 2.14) is Q-query CPA-secure.*

Independence heuristic. When choosing lattice parameters for homomorphic encryption, there is often a large amount of slack between the noise level observed in practice compared to those predicted by a worst-case noise analysis. To obtain better concrete efficiency, a common heuristic is to appeal to an *independence heuristic* and model the noise components that arise in homomorphic operations as independent subgaussian variables. We describe this heuristic in the following remark:

Remark 2.18 (Independence Heuristic)**.** Similar to existing lattice-based PIR constructions [ACLS18, GH19, MCR21] and other systems based on homomorphic encryption [GHS12b, CGGI18, MCR21], we model the noise components introduced by key switching and other homomorphic operations as *independent* subgaussian variables instead of bounding the noise magnitude with a worst-case bound when setting concrete parameters. This allows us to bound the *variance* of the noise vector (i.e., σ^2 where σ is the subgaussian width parameter associated with the noise) rather than the ℓ_∞ norm of the noise vector. Since the variance is additive for independent subgaussian variables, bounding the variance typically yields a square root improvement to some of the noise components compared to the worst-case bounds. In our theorem statements, we provide both a worst-case characterization of the noise as well as a bound on the noise variance under an independence assumption.

When setting concrete parameters (Section 5.1), we use the bounds obtained under the independence heuristic. Importantly, we only consider parameter sets that provide the target level of security, and select among these the smallest set that achieves a target level of correctness (estimated under the independence heuristic). The use of the independence heuristic does *not* lead to selecting parameter sets that provide lower security, only parameter sets that increase the probability of a correctness error. In Section 5.3, we empirically compare the actual noise magnitude in our encodings with those predicted by our noise model under the independence heuristic. The empiric results indicate that even with the independence heuristic, there still remains several bits of slack between the predicted noise level and that observed in practice. In these cases, there is no degradation in the correctness or security of the protocol.

Regev-GSW homomorphism. A critical homomorphism we rely on in our construction (and also used in a number of recent works [CGGI18, CCR19, GH19, CGGI20, PT20, MCR21]) is the ability to multiply a Regev encoding with a GSW encoding when the two encodings are encoded with respect to the *same* key. We describe this multiplication operation (sometimes called an "external product" [CGGI18, CCR19, CGGI20, MCR21]) below:

- Multiply($\mathbf{C}_{\text{GSW}}, \mathbf{C}_{\text{Regev}}$): On input a GSW encoding $\mathbf{C}_{\text{GSW}} \in R_q^{(n+1)\times m}$ with decomposition base $z \in \mathbb{N}$ and $m = (n+1)(\lfloor \log_z q \rfloor + 1)$ and a Regev encoding $\mathbf{C}_{\text{Regev}} \in R_q^{(n+1)\times n}$, output $\mathbf{C}_{\text{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\text{Regev}})$.

Theorem 2.19 (Regev-GSW Homomorphic Multiplication)**.** *Let $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$ be a secret key. Suppose $\mathbf{C}_{\text{GSW}}$ and $\mathbf{C}_{\text{Regev}}$ are encodings under $\mathbf{S}$ with the following properties:*

- *Suppose* $\mathbf{C}_{\mathsf{GSW}} \in R_q^{(n+1)\times n}$ *is a GSW encoding of a message* $\mu \in R_p$ *with decomposition base* $z \in \mathbb{N}$ *and error* $\mathbf{E}_{\mathsf{GSW}} \in R_q^{n\times m}$ *where* $m = (n+1)(\lfloor \log_z q \rfloor + 1)$.
- *Suppose* $\mathbf{C}_{\mathsf{Regev}} \in R_q^{(n+1)\times n}$ *is a Regev encoding of* $\mathbf{M} \in R_q^{n\times n}$ *with error* $\mathbf{E}_{\mathsf{Regev}} \in R_q^{n\times n}$.

Let $\mathbf{C} \leftarrow \mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{C}_{\mathsf{Regev}})$. *Then,* $\mathbf{C}$ *is a Regev encoding of* $\mu\mathbf{M} \in R_q$ *with error* $\mathbf{E}$ *where the following hold:*

- $\|\mathbf{E}\|_\infty \le d \|\mu\|_\infty \left\|\mathbf{E}_{\mathsf{Regev}}\right\|_\infty + md \|\mathbf{E}_{\mathsf{GSW}}\|_\infty z/2$. *If* μ *is a monomial, then* $\|\mathbf{E}\|_\infty \le \|\mu\|_\infty \left\|\mathbf{E}_{\mathsf{Regev}}\right\|_\infty + md \|\mathbf{E}_{\mathsf{GSW}}\|_\infty z/2$.
- *If the components of* $\mathbf{E}_{\mathsf{GSW}}$ *and* $\mathbf{E}_{\mathsf{Regev}}$ *are subgaussian with parameters* σ_{GSW} *and* σ_{Regev}, *respectively, then under the independence heuristic (Remark 2.18), the components of* $\mathbf{E}$ *are subgaussian with parameter* σ *where* $\sigma^2 = d \|\mu\|_\infty^2 \sigma_{\mathsf{Regev}}^2 + mdz^2\sigma_{\mathsf{GSW}}^2/4$ *(and* $\sigma^2 = \|\mu\|_\infty^2 \sigma_{\mathsf{Regev}}^2 + mdz^2\sigma_{\mathsf{GSW}}^2/4$ *if* μ *is a monomial).*

Proof. By definition, $\mathbf{S}^\mathsf{T}\mathbf{C}_{\mathsf{GSW}} = \mu\mathbf{S}^\mathsf{T}\mathbf{G}_{n+1,z} + \mathbf{E}_{\mathsf{GSW}}$ and $\mathbf{S}^\mathsf{T}\mathbf{C}_{\mathsf{Regev}} = \mathbf{M} + \mathbf{E}_{\mathsf{Regev}}$. Then,

$$\begin{aligned}\mathbf{S}^\mathsf{T}\mathbf{C} &= \mathbf{S}^\mathsf{T}\mathbf{C}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}}) \\ &= \mu\mathbf{S}^\mathsf{T}\mathbf{C}_{\mathsf{Regev}} + \mathbf{E}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}}) \\ &= \mu\mathbf{M} + \mu\mathbf{E}_{\mathsf{Regev}} + \mathbf{E}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}}).\end{aligned}$$

This is a Regev encoding of $\mu\mathbf{M}$ with error $\mathbf{E} = \mu\mathbf{E}_{\mathsf{Regev}} + \mathbf{E}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}})$. Thus,

$$\|\mathbf{E}\|_\infty \le d \|\mu\|_\infty \left\|\mathbf{E}_{\mathsf{Regev}}\right\|_\infty + md \|\mathbf{E}_{\mathsf{GSW}}\|_\infty z/2.$$

When μ is a monomial (and $R = \mathbb{Z}[x]/(x^d+1)$), then $\left\|\mu\mathbf{E}_{\mathsf{Regev}}\right\|_\infty = \|\mu\|_\infty \left\|\mathbf{E}_{\mathsf{Regev}}\right\|_\infty$. For the case where $\mathbf{E}_{\mathsf{GSW}}$ and $\mathbf{E}_{\mathsf{Regev}}$ are subgaussian, by Lemma 2.6, we have that $\mu\mathbf{E}_{\mathsf{Regev}}$ is subgaussian with parameter $\sqrt{d}\|\mu\|_\infty \sigma_{\mathsf{Regev}}$ (and $\|\mu\|_\infty \sigma_{\mathsf{Regev}}$ when μ is a monomial). Assuming independence of $\mathbf{E}_{\mathsf{GSW}}$ and $\mathbf{C}_{\mathsf{Regev}}$, the components of $\mathbf{E}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}})$ is subgaussian with parameter $\sqrt{md}\sigma_{\mathsf{GSW}}z/2$. The claim now follows by appealing to the independence heuristic. □

3 Encoding Compression and Translation

Similar to previous PIR protocols based on homomorphic encryption, we view our database as a hypercube. A PIR query consists of a collection of encodings encrypting 0/1 indicator vectors that select for the desired index along each dimension (see Section 1.1). A naïve implementation would require at least one encoding for each dimension of the hypercube in the query. Previously, Angel et al. [ACLS18] and Chen et al. [CCR19] introduced a query compression algorithm to pack the ciphertexts for the different dimensions into a *single* RLWE ciphertext (specifically, a BFV ciphertext [Bra12, FV12]).

To achieve higher rate and reduce noise growth, Spiral follows the Gentry-Halevi approach [GH19] of encoding the index along the first dimension using a matrix Regev encoding and the subsequent dimensions using GSW encodings (see Section 4). In this section, we introduce new building blocks to enable an analogous query compression approach as [ACLS18, CCR19] that allows us to compress the query encodings into a single Regev encoding of a *scalar*. Using our transformations, a PIR query in Spiral consists of a single RLWE ciphertext, which precisely matches schemes like SealPIR [ACLS18] or OnionPIR [MCR21]. However, due to better control of noise growth, Spiral can be instantiated with smaller lattice parameters, thus resulting in *smaller* queries (see Section 5.3). Our approach relies on four main ingredients which we describe in this section:

- In Section 3.1, we show how to expand a Regev encoding of a scalar $\mu \in R_q$ into a matrix Regev encoding of $\mu\mathbf{I}_n$ for any $n > 1$. In the Spiral protocol, this is used to obtain the matrix Regev encoding of the query index along the first database dimension.
- In Section 3.2, we show how to take Regev encodings of the components of $\mu \cdot \mathbf{g}_z$ to obtain a GSW encoding of μ with respect to the gadget matrix $\mathbf{G}_{n+1,z}$ for any $n \ge 1$ (Section 3.2). In the Spiral protocol, this is used to derive the GSW encodings of the query index along the subsequent dimensions of the database.

- To compress the query itself, we rely on previous techniques [ACLS18, CCR19] to pack multiple Regev encodings of *scalars* into a *single* Regev encoding (of a polynomial).
- Finally, after server processing, we apply modulus switching [BV11, BGV12] to the output Regev encoding to reduce the encoding size. Here, we describe a simple variant of modulus switching that rescales the Regev encoding by *two* different scaling factors to achieve a higher rate (Section 3.4). This is especially beneficial when working with matrix Regev encodings.

We believe that our transformations are also useful in other settings that combine Regev and GSW encodings. Overall, they allow us to take advantage of the high rate of matrix-Regev encodings and the slower (asymmetric) noise growth of GSW homomorphic operations, but without needing to communicate low-rate GSW encodings.

The scalar-to-matrix and Regev-to-GSW transformations we develop here are very similar to "key switching" transformations used in FHE [BV11, BGV12]. Much like key switching in FHE, the client needs to publish additional key-switching components (as part of the public parameters of the PIR scheme). The key-switching matrices are essentially encryptions of the secret key for the encoding scheme, so security relies on a key-dependent message security assumption (e.g., a circular security assumption). We note that previous query expansion algorithms [ACLS18, CCR19] also require publishing key-switching matrices (to support automorphisms), which also necessitate making a circular security assumption.

3.1 Expanding a Scalar Regev Encoding to a Matrix Regev Encoding

First, we describe a method to expand a Regev encoding of a scalar $\mu \in R_q$ into a matrix Regev encoding of $\mu \mathbf{I}_n \in R_q^{n \times n}$. The conversion procedure consists of a setup algorithm that samples a conversion key (i.e., a key-switching matrix):

- $\mathsf{ScalToMatSetup}(\mathbf{s}_0, \mathbf{S}_1, z)$: On input the source key $\mathbf{s}_0 = [-\tilde{s}_0 \mid 1]^\mathsf{T} \in R_q^2$, the target key $\mathbf{S}_1 = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$, and a decomposition base $z \in \mathbb{N}$, sample $\mathbf{a} \xleftarrow{\mathrm{R}} R_q^m$ and $\mathbf{E} \leftarrow \chi^{n \times m}$, where $m = n(\lfloor \log_z q \rfloor + 1)$. Then, output the key
$$\mathbf{W} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}_1 \mathbf{a}^\mathsf{T} + \mathbf{E} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times m} \\ -\tilde{s}_0 \cdot \mathbf{G}_{n,z} \end{bmatrix} \in R_q^{(n+1)\times m}.$$
- $\mathsf{ScalToMat}(\mathbf{W}, \mathbf{c})$: On input a key $\mathbf{W} \in R_q^{(n+1)\times m}$ and an encoding $\mathbf{c} = (c_0, c_1) \in R_q^2$, output $\mathbf{W}\mathbf{G}_{n,z}^{-1}(c_0 \mathbf{I}_n) + [\, \mathbf{0}^n \mid c_1 \mathbf{I}_n \,]^\mathsf{T}$.

Theorem 3.1 (Scalar to Matrix Conversion). *Let $\mathbf{c} = (c_0, c_1)$ be a Regev encoding of a scalar $\mu \in R_q$ with respect to a secret key $\mathbf{s}_0 = [-\tilde{s}_0 \mid 1]^\mathsf{T} \in R_q^{2\times 1}$ and error e. Take any secret key $\mathbf{S}_1 = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$ and decomposition base $z \in \mathbb{N}$. Let $t = \lfloor \log_z q \rfloor + 1$ and $m = nt$. Let $\mathbf{W} \leftarrow \mathsf{ScalToMatSetup}(\mathbf{s}_0, \mathbf{S}_1, z)$, $\mathbf{C} \leftarrow \mathsf{ScalToMat}(\mathbf{W}, \mathbf{c})$. Then, $\mathbf{C}$ is a Regev encoding of $\mu \mathbf{I}_n$ under $\mathbf{S}_1$ and error $\mathbf{E}$ where $\mathbf{E}$ satisfy the following properties:*

- *If χ in* $\mathsf{ScalToMatSetup}$ *is B-bounded, then $\|\mathbf{E}\|_\infty \le \|e\|_\infty + dtBz/2$; and*
- *If e is subgaussian with parameter σ_e and χ is subgaussian with parameter σ_χ, then under the independence heuristic, the components of $\mathbf{E}$ are subgaussian with parameter σ_E where $\sigma_E^2 = \sigma_e^2 + dtz^2\sigma_\chi^2/4$.*

Proof. By definition, $\mathbf{s}_0^\mathsf{T}\mathbf{c} = \mu + e$ and
$$\mathbf{W} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}_1 \mathbf{a}^\mathsf{T} + \tilde{\mathbf{E}} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times m} \\ -\tilde{s}_0 \cdot \mathbf{G}_{n,z} \end{bmatrix} \in R_q^{(n+1)\times m},$$
where $\tilde{\mathbf{E}} \leftarrow \chi^{n\times m}$ and for some $\mathbf{a} \in R_q^m$. Then,
$$\begin{aligned} \mathbf{S}_1^\mathsf{T}\mathbf{C}_1 = \mathbf{S}_1^\mathsf{T}\mathbf{W}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) + \mathbf{S}_1^\mathsf{T}\begin{bmatrix} \mathbf{0}^{1\times n} \\ c_1\mathbf{I}_n \end{bmatrix} &= \tilde{\mathbf{E}}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) - \tilde{s}_0 c_0 \mathbf{I}_n + c_1\mathbf{I}_n \\ &= \tilde{\mathbf{E}}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) + \mathbf{I}_n(\mathbf{s}_0^\mathsf{T}\mathbf{c}) \\ &= \mu\mathbf{I}_n + e\mathbf{I}_n + \tilde{\mathbf{E}}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n). \end{aligned}$$

Thus, $\mathbf{C}$ is an encoding of $\mu\mathbf{I}_n$ with error $\mathbf{E} = e\mathbf{I}_n + \tilde{\mathbf{E}}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n)$. For the case where χ is B-bounded, then the entries of $\tilde{\mathbf{E}}$ are B-bounded, so $\|\tilde{\mathbf{E}}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n)\|_\infty \le \gamma_R tBz/2 = dtBz/2$,[5] and the claim follows. When the components of e and $\tilde{\mathbf{E}}$ are independent subgaussians, and appealing also to the independence heuristic, then the components of $\mathbf{E}$ are also subgaussian with parameter σ_E where $\sigma_E^2 = \sigma_e^2 + dtz^2\sigma_\chi^2/4$. □

3.2 Converting Regev Encodings into GSW Encodings

Next, we describe an approach to construct a GSW encoding of a message $\mu \in R_q$ (with decomposition base z_{GSW}) from a collection of *scalar* Regev encodings of $\mu\mathbf{g}_{z_{\mathsf{GSW}}} = [\mu, \mu \cdot z_{\mathsf{GSW}}, \ldots, \mu \cdot z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1}]$ where $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1$. Chen et al. [CCR19] previously showed an approach for the special case where $n = 1$ that builds up the GSW ciphertext row by row using homomorphic multiplications. It is not clear how to extend this approach to higher dimensions (e.g., to allow homomorphic multiplication with matrix Regev encodings). Here, we describe a general transformation for arbitrary n.

To have finer control over noise growth, we introduce an additional decomposition base z_{conv} used for the conversion algorithm. The decomposition base z_{conv} for conversion does *not* have to match the decomposition base z_{GSW} for the GSW encoding. As we show in Theorem 3.2, the noise introduced by the encoding conversion step depends only on the decomposition base z_{conv} and *not* on the decomposition base z_{GSW} associated with the GSW encodings. This will enable more flexibility in parameter selection (see Section 5.1) and better concrete efficiency.

- $\mathsf{RegevToGSWSetup}(\mathbf{s}_{\mathsf{Regev}}, \mathbf{S}_{\mathsf{GSW}}, z_{\mathsf{GSW}}, z_{\mathsf{conv}})$: On input the Regev secret key $\mathbf{s}_{\mathsf{Regev}} = [-\tilde{s}_{\mathsf{Regev}} \mid 1]^\mathsf{T} \in R_q^2$, the GSW secret key $\mathbf{S}_{\mathsf{GSW}} = [-\tilde{\mathbf{s}}_{\mathsf{GSW}} \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$, and decomposition bases $z_{\mathsf{GSW}}, z_{\mathsf{conv}} \in \mathbb{N}$, proceed as follows:
  - Define $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1$, $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1$, and $m_{\mathsf{GSW}} = (n+1)t_{\mathsf{GSW}}$.
  - Sample $\mathbf{W} \leftarrow \mathsf{ScalToMatSetup}(\mathbf{s}_{\mathsf{Regev}}, \mathbf{S}_{\mathsf{GSW}}, z_{\mathsf{conv}})$.
  - Sample $\mathbf{a} \xleftarrow{\mathrm{R}} R_q^{2t_{\mathsf{conv}}}$ and $\mathbf{E} \leftarrow \chi^{n\times 2t_{\mathsf{conv}}}$ and construct the matrix
$$\mathbf{V} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}_{\mathsf{GSW}}\mathbf{a}^\mathsf{T} + \mathbf{E} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times 2t_{\mathsf{conv}}} \\ -\tilde{\mathbf{s}}_{\mathsf{GSW}} \cdot (\mathbf{s}_{\mathsf{Regev}}^\mathsf{T} \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\mathsf{T}) \end{bmatrix} \in R_q^{(n+1)\times 2t_{\mathsf{conv}}}. \tag{3.1}$$
  - Define the permutation matrix $\mathbf{\Pi} \in \{0,1\}^{m_{\mathsf{GSW}}\times m_{\mathsf{GSW}}}$ such that
$$\begin{bmatrix} \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} & \mathbf{0}^{1\times nt_{\mathsf{GSW}}} \\ \mathbf{0}^{n\times t_{\mathsf{GSW}}} & \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} \otimes \mathbf{I}_n \end{bmatrix} \mathbf{\Pi} = \begin{bmatrix} \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} & \mathbf{0}^{1\times nt_{\mathsf{GSW}}} \\ \mathbf{0}^{n\times t_{\mathsf{GSW}}} & \mathbf{I}_n \otimes \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} \end{bmatrix} = \mathbf{G}_{n+1,z_{\mathsf{GSW}}} \in R_q^{(n+1)\times m_{\mathsf{GSW}}}.$$

  Output the conversion key $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$.
- $\mathsf{RegevToGSW}(\mathsf{ck}, \mathbf{c}_1, \ldots, \mathbf{c}_{t_{\mathsf{GSW}}})$: On input the conversion key $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$ and Regev encodings $\mathbf{c}_1, \ldots, \mathbf{c}_{t_{\mathsf{GSW}}} \in R_q^2$, compute $\mathbf{C}_i \leftarrow \mathsf{ScalToMat}(\mathbf{W}, \mathbf{c}_i)$ for each $i \in [t_{\mathsf{GSW}}]$. Then, output
$$\mathbf{C} = [\ \mathbf{V}\mathbf{g}_{z_{\mathsf{conv}}}^{-1}(\hat{\mathbf{C}}) \mid \mathbf{C}_1 \mid \cdots \mid \mathbf{C}_{t_{\mathsf{GSW}}}\ ] \cdot \mathbf{\Pi}$$
  where $\hat{\mathbf{C}} = [\ \mathbf{c}_1 \mid \cdots \mid \mathbf{c}_{t_{\mathsf{GSW}}}\ ] \in R_q^{2\times t_{\mathsf{GSW}}}$.

Theorem 3.2. *Fix decomposition bases $z_{\mathsf{GSW}}, z_{\mathsf{conv}} \in \mathbb{N}$. Let $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1$ and $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1$. Let $\mathbf{c}_1, \ldots, \mathbf{c}_{t_{\mathsf{GSW}}}$ be Regev encodings of $\mu, \mu z_{\mathsf{GSW}}, \ldots, \mu z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1} \in R_q$ with errors $e_1, \ldots, e_{t_{\mathsf{GSW}}} \in R_q$, respectively. Take any secret key $\mathbf{S}_{\mathsf{GSW}} = [-\tilde{\mathbf{s}}_{\mathsf{GSW}} \mid \mathbf{I}_n]^\mathsf{T}$ and let $\mathsf{ck} \leftarrow \mathsf{RegevToGSW}(\mathbf{s}_{\mathsf{Regev}}, \mathbf{S}_{\mathsf{GSW}}, z_{\mathsf{GSW}}, z_{\mathsf{conv}})$, $\mathbf{C} \leftarrow \mathsf{RegevToGSW}(\mathsf{ck}, \mathbf{c}_1, \ldots, \mathbf{c}_{t_{\mathsf{GSW}}})$. Then, $\mathbf{C}$ is a GSW encoding of μ with respect to secret key $\mathbf{S}$, decomposition base z_{GSW}, and error $\mathbf{E}$ where $\mathbf{E}$ satisfies the following properties:*

- *If the noise distribution χ in* $\mathsf{ScalToMatSetup}$, $\mathsf{RegevToGSWSetup}$ *is B-bounded,*[6] *then $\|\mathbf{E}\|_\infty \le de_{\max}\|\tilde{\mathbf{s}}_{\mathsf{GSW}}\|_\infty + dt_{\mathsf{conv}}Bz_{\mathsf{conv}}$ where $e_{\max} = \max_{i\in[t_{\mathsf{GSW}}]}\|e_i\|_\infty$.*

[5] In particular, note that each column of $\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n)$ only contains at most t non-zero entries.

[6] While we could use different error distributions χ for ScalToMatSetup and RegevToGSWSetup, this level of generality is unnecessary in this work. We present the analysis for the setting where the *same* error distribution is used for both the ScalToMat and the RegevToGSW transformations.

- *If the noise distribution χ in* ScalToMatSetup *and* RegevToGSWSetup *is subgaussian with parameter σ_χ, and the errors $e_1, \ldots, e_t$ are subgaussian with parameter σ_e, then under the independence heuristic (Remark 2.18), the components of $\mathbf{E}$ are subgaussian with parameter σ where $\sigma^2 = d\sigma_e^2 \|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty^2 + dt_{\mathrm{conv}}\sigma_\chi^2 z_{\mathrm{conv}}^2/2$.*

Proof. First, $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$, where $\mathbf{W} \leftarrow \mathsf{ScalToMatSetup}(\mathbf{s}_{\mathrm{Regev}}, \mathbf{S}_{\mathrm{GSW}}, z_{\mathrm{conv}})$ and $\mathbf{V}$ is given in Eq. (3.1). Let $\hat{\mathbf{C}} = [\ \mathbf{c}_1 \mid \cdots \mid \mathbf{c}_{t_{\mathrm{GSW}}}\ ]$ and $\hat{\mathbf{e}} = [\ e_1 \mid \cdots \mid e_{t_{\mathrm{GSW}}}\ ]^\mathsf{T}$. By definition, $\mathbf{C} = [\ \mathbf{V}\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}}) \mid \mathbf{C}_1 \mid \cdots \mid \mathbf{C}_{t_{\mathrm{GSW}}}\ ] \cdot \mathbf{\Pi}$, where $\mathbf{C}_i \leftarrow \mathsf{ScalToMat}(\mathbf{W}, \mathbf{c}_i)$ for each $i \in [t_{\mathrm{GSW}}]$. We now consider the components of $\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}\mathbf{C}$:

- By definition $\mathbf{s}_{\mathrm{Regev}}^\mathsf{T}\hat{\mathbf{C}} = \mu\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} + \hat{\mathbf{e}}^\mathsf{T}$. Now, by definition of $\mathbf{V}$ from Eq. (3.1),

$$\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}\mathbf{V}\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}}) = \mathbf{E}_V\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}}) - \tilde{\mathbf{s}}_{\mathrm{GSW}}\mathbf{s}_{\mathrm{Regev}}^\mathsf{T}\hat{\mathbf{C}} = \mathbf{E}_V\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}}) - \tilde{\mathbf{s}}_{\mathrm{GSW}}(\mu\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} + \hat{\mathbf{e}}^\mathsf{T}),$$

  for some $\mathbf{E}_V \leftarrow \chi^{n \times 2t_{\mathrm{conv}}}$.

- By Theorem 3.1, for $i \in [t_{\mathrm{GSW}}]$, we have $\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}\mathbf{C}_i = z_{\mathrm{GSW}}^{i-1} \cdot \mu\mathbf{I}_n + \mathbf{E}_i$. Letting $\tilde{\mathbf{E}} = [\ \mathbf{E}_1 \mid \cdots \mid \mathbf{E}_{t_{\mathrm{GSW}}}\ ]$, we can then write

$$\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}[\ \mathbf{C}_1 \mid \cdots \mid \mathbf{C}_{t_{\mathrm{GSW}}}\ ] = \mu\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} \otimes \mathbf{I}_n + \tilde{\mathbf{E}}.$$

Let $\tilde{\mathbf{E}}' = [\ \mathbf{E}_V\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}}) - \tilde{\mathbf{s}}_{\mathrm{GSW}}\hat{\mathbf{e}}^\mathsf{T} \mid \tilde{\mathbf{E}}\ ] \cdot \mathbf{\Pi} \in R_q^{n \times m_{\mathrm{GSW}}}$. Now, putting everything together, we can write

$$\begin{aligned}\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}\mathbf{C} &= \left[-\mu\tilde{\mathbf{s}}_{\mathrm{GSW}}\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} \mid \mu\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} \otimes \mathbf{I}_n\right]\mathbf{\Pi} + \tilde{\mathbf{E}}' = \mu[-\tilde{\mathbf{s}}_{\mathrm{GSW}} \mid \mathbf{I}_n]\begin{bmatrix}\mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} & \mathbf{0}^{1 \times nt_{\mathrm{GSW}}} \\ \mathbf{0}^{n \times t_{\mathrm{GSW}}} & \mathbf{g}_{z_{\mathrm{GSW}}}^\mathsf{T} \otimes \mathbf{I}_n\end{bmatrix}\mathbf{\Pi} + \tilde{\mathbf{E}}' \\ &= \mu\mathbf{S}_{\mathrm{GSW}}^\mathsf{T}\mathbf{G}_{n+1, z_{\mathrm{GSW}}} + \tilde{\mathbf{E}}'.\end{aligned}$$

Thus, $\mathbf{C}$ is a GSW encoding of μ with error $\tilde{\mathbf{E}}'$. We now bound the components of $\tilde{\mathbf{E}}'$ as follows:

- Since χ is B-bounded, by Theorem 3.1, $\|\tilde{\mathbf{E}}\|_\infty \le e_{\max} + dt_{\mathrm{conv}}Bz_{\mathrm{conv}}/2$. Similarly, by Lemma 2.6, $\|\mathbf{E}_V\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}})\|_\infty \le 2t_{\mathrm{conv}}dBz_{\mathrm{conv}}/2 = dt_{\mathrm{conv}}Bz_{\mathrm{conv}}$.
- Next, $\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\hat{\mathbf{e}}^\mathsf{T}\|_\infty \le de_{\max}\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty$.
- Finally, multiplying by a permutation matrix $\mathbf{\Pi}$ does not change the error magnitude.

The overall error is thus bounded by

$$\begin{aligned}\|\tilde{\mathbf{E}}'\|_\infty &\le \max(e_{\max} + dt_{\mathrm{conv}}Bz_{\mathrm{conv}}/2, de_{\max}\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty + dt_{\mathrm{conv}}Bz_{\mathrm{conv}}) \\ &= de_{\max}\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty + dt_{\mathrm{conv}}Bz_{\mathrm{conv}}.\end{aligned}$$

For the subgaussian case, we have the following:

- By Theorem 3.1, the components of $\tilde{\mathbf{E}}$ are subgaussian with parameter σ_E where $\sigma_E^2 = \sigma_e^2 + dt_{\mathrm{conv}}z_{\mathrm{conv}}^2\sigma_\chi^2/2$.
- By Lemma 2.6, the components of $\mathbf{E}_V\mathbf{g}_{z_{\mathrm{conv}}}^{-1}(\hat{\mathbf{C}})$ are subgaussian with parameter $\sqrt{2t_{\mathrm{conv}}d}\sigma_\chi z_{\mathrm{conv}}/2$. Similarly, the components of $\tilde{\mathbf{s}}_{\mathrm{GSW}}\hat{\mathbf{e}}^\mathsf{T}$ are subgaussian with parameter $\sqrt{d}\sigma_e\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty$.

Under the independence heuristic, the components of $\tilde{\mathbf{E}}'$ are thus subgaussian with parameter σ where $\sigma^2 = \max(\sigma_E^2, t_{\mathrm{conv}}d\sigma_\chi^2 z_{\mathrm{conv}}^2/2 + d\sigma_e^2\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty^2) = t_{\mathrm{conv}}d\sigma_\chi^2 z_{\mathrm{conv}}^2/2 + d\sigma_e^2\|\tilde{\mathbf{s}}_{\mathrm{GSW}}\|_\infty^2$. □

Remark 3.3 (Compressing GSW Encodings)**.** The RegevToGSW algorithm takes t_{GSW} Regev encodings (consisting of $2 \cdot t_{\mathrm{GSW}}$ elements of R_q) and outputs a single GSW encoding with $(n+1)m_{\mathrm{GSW}} = (n+1)^2 t_{\mathrm{GSW}}$ elements of R_q. Thus, our Regev-to-GSW transformation can be viewed as a way to achieve a $(n+1)^2/2$ factor compression on GSW encodings at the cost of a small amount of additional noise and needing to store a (large) conversion key ck. However, ck can be generated in a separate offline phase and reused across multiple protocol invocations. This provides an effective way to reduce the *online* communication costs of sending GSW encodings.

3.3 Coefficient Extraction on Regev Encodings

The next ingredient we require is the coefficient expansion algorithm by Angel et al. [ACLS18] and extended by Chen et al. [CCR19]. The algorithm takes a polynomial $f = \sum_{i\in[0,2^r-1]} f_i x^i \in R_q$ as input and outputs a (scaled) vector of coefficients $2^r \cdot (f_0, \ldots, f_{2^r-1}) \in \mathbb{Z}_q^{2^r}$. This algorithm relies on the fact that we can homomorphically evaluate automorphisms on Regev-encoded polynomials. We use the same approach from [ACLS18, CCR19], so we defer the description to Appendix A.

3.4 Modulus Switching

Modulus switching [BV11, BGV12] reduces the size of Regev-based encodings by rescaling the encoding down into a smaller ring while preserving the encoded message. This allows performing homomorphic operations over a larger ring R_q (which accommodates more homomorphic operations) and then rescaling the final encoding (e.g., the PIR response) to a smaller ring $R_{q'}$ to obtain a more compact representation.

While previous approaches [BV11, BGV12, GHS12b, GH19] rescale all of the ciphertext components from R_q to $R_{q'}$ for some $q' < q$, we can achieve further compression by re-scaling some of the components of the Regev ciphertext to one modulus q_1 and the remaining components to a different modulus q_2. Decoding then requires a modified procedure (see Theorem 3.4). The advantage of this variant is that we can use a very small value of q_1 (e.g., $q_1 = 4p$) and still ensure correctness. Importantly, the message-embedding components of the ciphertext (i.e., an $n^2/(n^2+n)$ fraction of the ciphertext components) can be rescaled to the smaller modulus q_1. This allows us to achieve a higher rate when using larger plaintext dimensions n. With our modulus switching approach, we achieve a ciphertext rate of

$$\text{rate} = \frac{\text{plaintext size}}{\text{ciphertext size}} = \frac{n^2 \log p}{n^2 \log q_1 + n \log q_2}.$$

This allows us to achieve considerably higher rates (e.g., up to 0.81 compared to a maximum of 0.34 if we were to use a single modulus; see Section 5.3) while maintaining low computational costs. We now describe our variant of the modulus switching procedure ModulusSwitch along with an encoding-recovery procedure Recover that takes a rescaled encoding $(\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$ (as output by ModulusSwitch) and the secret key $\mathbf{S}$ as input, and outputs an encoding $\mathbf{Z}$ (over R_{q_1}) satisfying $\mathbf{Z} = \lfloor q_1/p \rfloor \mathbf{M} + \mathbf{E}'$. If $\mathbf{E}'$ is sufficiently small, we can recover $\mathbf{M}$ from $\mathbf{Z}$ using the procedure from Definition 2.10. Both the ModulusSwitch and Recover algorithms are parameterized by a pair of moduli $q_1, q_2 \in \mathbb{N}$.

- $\mathsf{ModulusSwitch}_{q_1,q_2}(\mathbf{C})$: On input an encoding $\mathbf{C} = \begin{bmatrix} \mathbf{c}_1^\mathsf{T} \\ \mathbf{C}_2 \end{bmatrix}$, where $\mathbf{c}_1 \in R_q^n$ and $\mathbf{C}_2 \in R_q^{n\times n}$, let $\hat{\mathbf{c}}_1 = \lfloor \mathbf{c}_1 \cdot q_2/q \rceil \in R_{q_2}^n$ and $\hat{\mathbf{C}}_2 = \lfloor \mathbf{C}_2 \cdot q_1/q \rceil \in R_{q_1}^{n\times n}$. Both the division and rounding are performed over the rationals. Output $(\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$
- $\mathsf{Recover}_{q_1,q_2}(\mathbf{S}, (\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2))$: On input the secret key $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n] \in R_q^{n\times(n+1)}$, and an encoding $(\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$ where $\hat{\mathbf{c}}_1 \in R_{q_2}^n$, and $\hat{\mathbf{C}}_2 \in R_{q_1}^{n\times n}$, compute $\mathbf{Z} = \left\lfloor (q_1/q_2)(-\tilde{\mathbf{s}}\hat{\mathbf{c}}_1^\mathsf{T}) \right\rceil + \hat{\mathbf{C}}_2$, and output $\mathbf{Z} \bmod q_1$.

Theorem 3.4 (Modulus Switching)**.** *Fix positive integers $q > q_2 > q_1 > p$. Let $\mathbf{C} \in R_q^{(n+1)\times n}$ be a Regev encoding of $\lfloor q/p \rfloor \mathbf{M} \in R_q^{n\times n}$ for some message $\mathbf{M} \in R^{n\times n}$ (with $\|\mathbf{M}\|_\infty \le p/2$) and with respect to a secret key $\mathbf{S} = [-\tilde{\mathbf{s}} \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$ and error $\mathbf{E} \in R_q^{n\times n}$. Let $(\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2) \leftarrow \mathsf{ModulusSwitch}_{q_1,q_2}(\mathbf{C})$ and $\mathbf{Z} \leftarrow \mathsf{Recover}_{q_1,q_2}(\mathbf{S}, (\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2))$. Then, $\mathbf{Z} = \lfloor q_1/p \rfloor \mathbf{M} + \mathbf{E}' \pmod{q_1}$, where $\mathbf{E}' = \mathbf{E}_1' + \mathbf{E}_2'$ and*

$$\left\|\mathbf{E}_1'\right\|_\infty \le \frac{1}{2}\left(2 + (q_1 \bmod p) + \frac{q_1}{q}(q \bmod p)\right) \quad \text{and} \quad \left\|\mathbf{E}_2'\right\|_\infty \le \frac{q_1}{q}\|\mathbf{E}\|_\infty + \frac{q_1}{2q_2} d \|\tilde{\mathbf{s}}\|_\infty .$$

When the components of $\mathbf{E}$ are subgaussian with parameter σ_e and the components of $\tilde{\mathbf{s}}$ are subgaussian with parameter σ_s, then under the independence heuristic (Remark 2.18), the components of $\mathbf{E}_2'$ are subgaussian with parameter $\sigma = \sqrt{(q_1/q)^2\sigma_e^2 + (q_1/q_2)^2\sigma_s^2 d/4}$.

Proof. Write $\mathbf{C}$ as $\mathbf{C} = \begin{bmatrix} \mathbf{c}_1^\mathsf{T} \\ \mathbf{C}_2 \end{bmatrix}$, where $\mathbf{c}_1 \in R_q^n$ and $\mathbf{C}_2 \in R_q^{n \times n}$. Let $k = \lfloor q/p \rfloor = (q - q \bmod p)/p$. By definition, $\mathbf{S}^\mathsf{T}\mathbf{C} = k \cdot \mathbf{M} + \mathbf{E} \pmod q$, so we can write $\mathbf{S}^\mathsf{T}\mathbf{C} = -\tilde{\mathbf{s}}\mathbf{c}_1^\mathsf{T} + \mathbf{C}_2 = k \cdot \mathbf{M} + \mathbf{E} + q\mathbf{D} \in R^{n \times n}$ for some $\mathbf{D} \in R^{n \times n}$. Over the rationals, we can write $\hat{\mathbf{c}}_1 = (q_2/q)\mathbf{c}_1 + \mathbf{e}_1$, $\hat{\mathbf{C}}_2 = (q_1/q)\mathbf{C}_2 + \mathbf{E}_2$, and $\mathbf{Z} = (q_1/q_2)(-\tilde{\mathbf{s}}\hat{\mathbf{c}}_1^\mathsf{T}) + \hat{\mathbf{C}}_2 + \mathbf{e}_3$, where the rounding errors $\mathbf{e}_1, \mathbf{E}_2, \mathbf{e}_3$ satisfy $\|\mathbf{e}_1\|_\infty, \|\mathbf{E}_2\|_\infty, \|\mathbf{e}_3\|_\infty \le 1/2$. Again, working over the rationals,

$$
\begin{aligned}
\mathbf{Z} &= (q_1/q_2)(-\tilde{\mathbf{s}}\hat{\mathbf{c}}_1^\mathsf{T}) + \hat{\mathbf{C}}_2 + \mathbf{e}_3 \\
&= (q_1/q_2)\big((q_2/q)(-\tilde{\mathbf{s}}\mathbf{c}_1^\mathsf{T}) - \tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T}\big) + (q_1/q)\mathbf{C}_2 + \mathbf{E}_2 + \mathbf{e}_3 \\
&= (q_1/q)(-\tilde{\mathbf{s}}\mathbf{c}_1^\mathsf{T} + \mathbf{C}_2) - (q_1/q_2)\tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T} + \mathbf{E}_2 + \mathbf{e}_3 \\
&= (kq_1/q)\mathbf{M} + (q_1/q)\mathbf{E} - (q_1/q_2)\tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T} + \mathbf{E}_2 + \mathbf{e}_3 + q_1\mathbf{D} \\
&= \frac{q_1}{p}\mathbf{M} - \frac{q_1(q \bmod p)}{pq}\mathbf{M} + \frac{q_1}{q}\mathbf{E} - \frac{q_1}{q_2}\tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T} + \mathbf{E}_2 + \mathbf{e}_3 + q_1\mathbf{D} \\
&= \left\lfloor \frac{q_1}{p} \right\rfloor \mathbf{M} - \underbrace{\frac{q_1 \bmod p}{p}\mathbf{M} - \frac{q_1(q \bmod p)}{pq}\mathbf{M} + \mathbf{E}_2 + \mathbf{e}_3}_{\mathbf{E}_1'} + \underbrace{\frac{q_1}{q}\mathbf{E} - \frac{q_1}{q_2}\tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T}}_{\mathbf{E}_2'} \pmod{q_1}.
\end{aligned}
$$

The bound now follows from the fact that $\|\mathbf{e}_1\|_\infty, \|\mathbf{E}_2\|_\infty, \|\mathbf{e}_3\|_\infty < 1/2$ and $\|\mathbf{M}\|_\infty \le p/2$. For the subgaussian case, we apply the independence heuristic to conclude that the components of $-\tilde{\mathbf{s}}\mathbf{e}_1^\mathsf{T}$ are subgaussian with parameter $\sqrt{d}\sigma_s/2$. The claim now follows from the independence heuristic. □

4 The Spiral Protocol

The structure of the basic Spiral protocol follows recent constructions of PIR based on composing Regev-based encryption with GSW encryption [GH19, MCR21]. The primary difference is that it uses the techniques from Section 3 for query compression. Very briefly, the database of $N = 2^{\nu_1+\nu_2}$ records is arranged as a hypercube with dimensions $2^{\nu_1} \times 2 \times 2 \times \cdots \times 2$. Processing the initial (large) dimension only requires scalar multiplication (since the database is known in the clear) and is implemented using matrix Regev encodings. After processing the first dimension, the server has a $(2 \times 2 \times \cdots \times 2)$-hypercube containing 2^{ν_2} matrix-Regev encodings. The client's index for each of the subsequent dimensions is encoded using GSW, so using ν_2 rounds of the Regev-GSW homomorphic multiplication, the server can "fold" the remaining elements into a single matrix Regev encoding. We refer to Section 1.2 and Fig. 1 for a general overview.

Construction 4.1 (Spiral)**.** Let λ be a security parameter, and $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = d(\lambda)$ is a power of two. Let $p = p(\lambda)$ be the plaintext modulus and $n = n(\lambda)$ be the plaintext dimension.

Database structure. Each database record d_i is an element of $R_p^{n \times n}$, where $\|d_i\|_\infty \le p/2$. We represent a database $\mathcal{D} = \{d_1, \ldots, d_N\}$ of $N = 2^{\nu_1+\nu_2}$ records as a $(\nu_2 + 1)$-dimensional hypercube with dimensions $2^{\nu_1} \times 2 \times 2 \times \cdots \times 2$. In the following description, we index elements of $\mathcal{D}$ using either the tuple $(i, j_1, \ldots, j_{\nu_2})$ where $i \in [0, 2^{\nu_1} - 1]$ and $j_1, \ldots, j_{\nu_2} \in \{0, 1\}$, or the tuple (i, j) where $i \in [0, 2^{\nu_1} - 1]$ and $j \in [0, 2^{\nu_2} - 1]$.

Scheme parameters. A notable feature of our PIR is that it relies on several additional parameters that will be helpful for enabling new communication/computation trade-offs:

- Let $q = q(\lambda)$ be an encoding modulus (for the query) and $q_1 = q_1(\lambda), q_2 = q_2(\lambda)$ be the smaller moduli associated with the PIR response. We require that q is odd.
- Let $\chi = \chi(\lambda)$ be an error distribution. We use the *same* error distribution for all sub-algorithms.
- Let $z_{\mathsf{coeff}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}} \in \mathbb{N}$ be different decomposition bases that will be used for query expansion and homomorphic evaluation:

– z_{coeff} is the decomposition base for evaluating the automorphisms in the coefficient expansion algorithm (Section 3.3 and Algorithm 1);

– z_{conv} is the decomposition base used to translate scalar Regev encodings into matrix Regev encodings (Section 3.1); and

– z_{GSW} is the decomposition base used in GSW encodings.

The decomposition bases are chosen to balance the server computational costs with the total communication costs (see Section 5.1 for details on how we choose these parameters). For ease of notation, in the following, we will write $\mathbf{G}_{\mathsf{GSW}}$ to denote the gadget matrix $\mathbf{G}_{n+1,z_{\mathsf{GSW}}} \in R_q^{(n+1)\times m_{\mathsf{GSW}}}$ associated with GSW encodings, where $m_{\mathsf{GSW}} = (n+1)\cdot t_{\mathsf{GSW}}$ and $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1$.

We now describe the PIR scheme below:

- $\mathsf{Setup}(1^\lambda, 1^N)$: On input the security parameter λ and the database size N, the setup algorithm proceeds as follows:

  1. **Key-generation:** Sample two secret keys $\mathbf{S} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^n)$ and $\mathbf{s} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^1)$ that are used for response encoding and query encoding, respectively.
  2. **Regev-to-GSW conversion keys:** Compute $\mathsf{ck} \leftarrow \mathsf{RegevToGSWSetup}(\mathbf{s}, \mathbf{S}, z_{\mathsf{conv}})$.
  3. **Automorphism keys:** Let $\rho = 1 + \max(\nu_1, \lceil \log t_{\mathsf{GSW}}\nu_2 \rceil)$. For each $i \in [0, \rho - 1]$, compute $\mathbf{W}_i \leftarrow \mathsf{AutomorphSetup}(\mathbf{s}, \tau_{2^{\rho-i}+1}, z_{\mathsf{coeff}})$.

  Output the public parameters $\mathsf{pp} = (\mathsf{ck}, \mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1})$ and the querying key $\mathsf{qk} = (\mathbf{s}, \mathbf{S})$.

- $\mathsf{Query}(\mathsf{qk}, \mathsf{idx})$: On input the querying key $\mathsf{qk} = (\mathbf{s}, \mathbf{S})$ and an index $\mathsf{idx} = (i^*, j_1^*, \ldots, j_{\nu_2}^*)$ where $i^* \in [0, 2^{\nu_1} - 1]$ and $j_1^*, \ldots, j_{\nu_2}^* \in \{0, 1\}$, the query algorithm does the following:

  1. **Encoding the first dimension:** Define the polynomial $\mu_{i^*}(x) = \lfloor q/p \rfloor \cdot x^{i^*} \in R_q$.
  2. **Encoding subsequent dimensions:** Define the polynomial $\mu_{j^*} = \sum_{\ell \in [\nu_2]} \mu_{j_\ell^*}$ where for each $\ell \in [\nu_2]$,

  $$\mu_{j_\ell^*}(x) = j_\ell^* \sum_{k \in [t_{\mathsf{GSW}}]} (z_{\mathsf{GSW}})^{k-1} x^{(\ell-1)t_{\mathsf{GSW}}+k}.$$

  3. **Query packing:** Define the "packed" polynomial

  $$\mu(x) := 2^{-r_1}\mu_{i^*}(x^2) + 2^{-r_2} x \mu_{j^*}(x^2) \in R_q, \tag{4.1}$$

  where $r_1 = 1 + \nu_1$ and $r_2 = 1 + \lceil \log(t_{\mathsf{GSW}}\nu_2) \rceil$.
  4. **Query encryption:** Compute the encrypted query $\mathbf{c} \leftarrow \mathsf{Regev.Encode}(\mathbf{s}, \mu) \in R_q^2$. Output the query $\mathsf{q} = \mathbf{c}$ and an empty query state $\mathsf{st} = \perp$.

- $\mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q})$: On input the database $\mathcal{D}$, the public parameters $\mathsf{pp} = (\mathsf{ck}, \mathbf{W}_1, \ldots, \mathbf{W}_\rho)$, and a query $\mathsf{q} = \mathbf{c}$, the server response algorithm parses $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$ and proceeds as follows:

  1. **Query expansion:** The server expands the query ciphertext $\mathbf{c}$ into 2^{ν_1} matrix Regev encodings (for the first dimension) and ν_2 GSW encodings (for the subsequent dimensions) as follows:
     (a) **Initial expansion:** Homomorphically evaluate a single iteration of the coefficient expansion algorithm (Algorithm 1) on $\mathbf{c}$. Let $\mathbf{c}_{\mathsf{Reg}}, \mathbf{c}_{\mathsf{GSW}} \in R_q^2$ be the output encodings.
     (b) **First dimension expansion:** Continue homomorphic evaluation of Algorithm 1 for ν_1 additional iterations on $\mathbf{c}_{\mathsf{Reg}}$ to obtain encodings $\mathbf{c}_1^{(\mathsf{Reg})}, \ldots, \mathbf{c}_{2^{\nu_1}}^{(\mathsf{Reg})} \in R_q^2$. For each $i \in [0, 2^{\nu_1} - 1]$, let $\mathbf{C}_i^{(\mathsf{Reg})} \leftarrow \mathsf{ScalToMat}(\mathbf{W}, \mathbf{c}_i^{(\mathsf{Reg})})$.

(c) **GSW ciphertext expansion:** Continue homomorphic evaluation of Algorithm 1 for $\lceil \log(t_{\mathsf{GSW}} \nu_2) \rceil$ additional iterations on $\mathbf{c}_{\mathsf{GSW}}$ to obtain encodings $\mathbf{c}_1^{(\mathsf{GSW})}, \ldots, \mathbf{c}_{t_{\mathsf{GSW}}\nu_2}^{(\mathsf{GSW})} \in R_q^2$. Discard any additional encodings output by Algorithm 1 whenever $t_{\mathsf{GSW}}\nu_2$ is not a power of two. For each $j \in [\nu_2]$, compute $\mathbf{C}_j^{(\mathsf{GSW})} \leftarrow \mathsf{RegevToGSW}\big(\mathsf{ck}, \mathbf{c}_{(j-1)t_{\mathsf{GSW}}+1}^{(\mathsf{GSW})}, \ldots, \mathbf{c}_{jt_{\mathsf{GSW}}}^{(\mathsf{GSW})}\big)$.

Note that the above invocations of Algorithm 1 will use the automorphism keys $\mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1}$.

2. **Processing the first dimension:** For every $j \in [0, 2^{\nu_2} - 1]$, the server does the following:
   (a) Initialize $\mathbf{C}_j^{(0)} \leftarrow \mathsf{ScalarMul}\big(\mathbf{C}_0^{(\mathsf{Reg})}, d_{0,j}\big)$.
   (b) For each $i \in [2^{\nu_1} - 1]$, update $\mathbf{C}_j^{(0)} \leftarrow \mathsf{Add}\big(\mathbf{C}_j^{(0)}, \mathsf{ScalarMul}\big(\mathbf{C}_i^{(\mathsf{Reg})}, d_{i,j}\big)\big)$.
3. **Folding in the subsequent dimensions:** For each $r \in [\nu_2]$, and each $j \in [0, 2^{\nu_2 - r} - 1]$, compute

$$\mathbf{C}_j^{(r)} = \mathsf{Add}\left(\mathsf{Multiply}\left(\mathsf{Complement}(\mathbf{C}_r^{(\mathsf{GSW})}), \mathbf{C}_j^{(r-1)}\right), \mathsf{Multiply}\left(\mathbf{C}_r^{(\mathsf{GSW})}, \mathbf{C}_{2^{\nu_2 - r}+j}^{(r-1)}\right)\right), \tag{4.2}$$

   where the Complement operation is as defined in Remark 2.16.
4. **Modulus switching:** Output the rescaled response $\mathsf{r} \leftarrow \mathsf{ModulusSwitch}_{q_1,q_2}(\mathbf{C}_0^{(r)})$.

- $\mathsf{Extract}(\mathsf{qk}, \mathsf{st}, \mathsf{r})$: On input the query key $\mathsf{qk} = (\mathbf{s}, \mathbf{S})$, an (empty) query state st, and the server response r, the extraction algorithm first computes $\mathbf{Z} \leftarrow \mathsf{Recover}_{q_1,q_2}(\mathbf{S}, \mathsf{r}) \in R_{q_1}^{n \times n}$ and outputs $\mathbf{C} \leftarrow \mathsf{Decode}(\mathbf{Z}) \in R_p^{n \times n}$.

Remark 4.2 (Plaintext Dimension $n = 1$)**.** For the particular case where the plaintext dimension n satisfies $n = 1$ in Construction 4.1, we can simplify the protocol by taking the query encoding key $\mathbf{s}_0$ to be the *same* as the response encoding key $\mathbf{s}_1$. By setting $\mathbf{s}_0 = \mathbf{s}_1$, we no longer need to apply the scalar-to-matrix translation ScalToMat from $\mathbf{s}_0$ to $\mathbf{s}_1$ when expanding the first dimension. While the base version of Spiral always considers plaintext dimensions $n > 1$ to achieve higher rates and throughput (see Section 5), the $n = 1$ case is useful when combined with a "packing" approach we describe in Section 4.1.

Remark 4.3 (Decomposition Bases in Query Expansion)**.** As noted in Remark A.5, we can reduce noise growth from the expansion steps in Steps 1b and 1c of the Answer algorithm by using different decomposition bases. Since $2^{\nu_1} \gg t_{\mathsf{GSW}}\nu_2$ in all of the scenarios we consider, we opt to use a larger decomposition base to expand the Regev encodings for processing the first dimension and a smaller decomposition base to expand the GSW encodings for the folding steps. Using a larger decomposition base enables a faster expansion algorithm but adds more noise to the output encoding. We denote these expansion bases by $z_{\mathsf{coeff,Reg}}$ and $z_{\mathsf{coeff,GSW}}$.

Remark 4.4 (Query Size Trade-off and the SpiralStream Protocol)**.** To reduce noise growth in Construction 4.1, the client can directly upload the Regev encodings $\mathbf{c}_i^{(\mathsf{Reg})}$ and $\mathbf{c}_j^{(\mathsf{GSW})}$ for $i \in [2^{\nu_1}]$ and $j \in [t_{\mathsf{GSW}}\nu_2]$ as part of its query rather than compress them into a single encoding. This yields larger queries, but eliminates the noise growth from query expansion. As we discuss in Section 5, this setting is appealing for streaming scenarios where the *same* query is reused for a large number of consecutive requests. Note that it still remains *advantageous* to use our Regev-to-GSW transformation (Section 3.2 and Remark 3.3) rather than send GSW encodings directly. This is because GSW encodings are much larger than Regev encodings and the expansion process is fast and only introduces a small amount of noise. We refer to this variant of Spiral as SpiralStream.

Correctness and security analysis. We now state the formal correctness and security theorems for Construction 4.1, but defer their formal proofs to Appendix B.

Theorem 4.5 (Correctness)**.** *Let $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = d(\lambda)$ is a power of two. Let p be the plaintext modulus and n be the message dimension. Take any database $\mathcal{D}$ with up to $N = 2^{\nu_1 + \nu_2}$ records from $R_p^{n \times n}$, and arranged as a $(\nu_2 + 1)$-dimensional hypercube as described in Construction 4.1. Let $z = \max(z_{\mathsf{coeff}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}})$ be the largest decomposition base. Then, if χ is B-bounded and $q = \Omega\big(d^2 npBz \log q (2^{2\nu_1} p + \nu_2^2 Bdz \log^2 q)\big)$, $q_2 = \Omega(dBp)$, and $q_1 = \Omega(p^2)$, then Construction 4.1 is correct.*[7]

[7]For this setting of parameters, correctness holds *without* needing to appeal to the independence heuristic (Remark 2.18).

Theorem 4.6 (Security). *Let $\mathcal{F}_{\mathsf{aff}}$ and $\mathcal{F}_{\mathsf{auto}}$ be the following family of functions over R_q:*

- *Let $\mathcal{F}_{\mathsf{aff}}$ be the family of affine functions on $R_q^{(n+1)\times n}$.*
- *Let $\mathcal{F}_{\mathsf{auto}} = \{r \mapsto k \cdot \tau_\ell(r) \mid k \in \mathbb{Z}, \ell \in \mathbb{Z}\}$ be the family of scaled automorphisms over R_q, where for an integer $\ell \in \mathbb{Z}$, $\tau_\ell \colon R_q \to R_q$ is the ring automorphism $r(x) \mapsto r(x^\ell)$.*

Suppose the Regev encoding scheme with message space R_q^n is $\mathcal{F}_{\mathsf{aff}}$-KDM-secure and the Regev encoding scheme with message space R_q is $\mathcal{F}_{\mathsf{auto}}$-KDM-secure. Then Construction 4.1 satisfies query privacy.

4.1 SpiralPack: Higher Rate via Encoding Packing

In this section, we describe a variant of Spiral (called SpiralPack) that enables a higher rate and a higher throughput (for large records) at the expense of larger public parameters. As we discuss in greater detail in Section 5.1, the plaintext dimension n in Spiral directly affects the rate and the throughput. A larger value of n yields a higher rate (i.e., the rate scales with $n^2/(n^2+n)$). However, the cost of processing the first dimension scales *quadratically* with n.

Here, we describe an encoding packing approach that allows us to enjoy the "best of both worlds." At a high level, our approach takes n^2 Regev encodings of *scalars* and packs them into a *single* matrix Regev encoding of an $n \times n$ matrix. To leverage this to achieve higher rate, we modify Spiral as follows:

- Break each record in the database into n^2 blocks of equal length. This yields a collection of n^2 different databases, where the i^{th} database contains the i^{th} block of each record. To process a query, the server applies the query to each of the n^2 databases.
- The query consists of packed Regev encodings of *scalar* values (i.e., 1-dimensional values). As noted above, this minimizes the server's computational cost when processing the first dimension.
- After applying the query to each of the n^2 databases, the server has n^2 Regev encodings of *scalars*. Transmitting these back to the client would yield a protocol with a low rate (at best, the rate is $1/2$, and typically, it is much lower). Instead, the server now applies a "packing" technique to pack the n^2 Regev encodings into a single $n \times n$ *matrix* Regev encoding. The rate now scales with $n^2/(n^2+n) > 1/2$ whenever $n > 1$.

The packing transformation used here requires publishing an additional set of translation matrices in the public parameters. Thus, this approach provides a trade-off between the size of the public parameters and the online costs of the protocol (measured in terms of server throughput and rate). Since the public parameters can be reused over many queries, SpiralPack is better-suited for settings where a client will perform many database queries and the server is able to store the client's public parameters.

We note that our *encoding* packing approach is different from and incomparable to the *message* packing technique from [ACLS18, CCR19] described in Section 3.3. Our encoding packing technique packs scalar Regev encodings into a matrix Regev encoding while the message packing technique from [ACLS18, CCR19] packs multiple messages into a single scalar Regev encoding. Encoding packing helps reduce *response size* while message packing helps reduce *query size*. The SpiralPack protocol uses *both* packing techniques. We now describe our encoding packing approach.

Packing Regev encodings. Similar to the other encoding translation algorithms from Section 3, our packing technique relies on a set of public parameters. In the following description, we write $\mathbf{u}_i \in R_q^n$ to denote the i^{th} elementary basis vector (i.e., a vector with a 1 in the i^{th} component and 0 elsewhere).

- $\mathsf{PackSetup}(\mathbf{s}_0, \mathbf{S}_1, z)$: On input the source key $\mathbf{s}_0 = [-\tilde{s}_0 \mid 1]^\mathsf{T} \in R_q^2$, the target key $\mathbf{S}_1 = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1)\times n}$, and a decomposition base $z \in \mathbb{N}$, let $t = \lfloor \log_z q \rfloor + 1$. For each $i \in [n]$, sample $\mathbf{a}_i \xleftarrow{\text{R}} R_q^t$ and $\mathbf{E}_i \leftarrow \chi^{n\times t}$, and compute the key
$$\mathbf{P}_i = \begin{bmatrix} \mathbf{a}_i^\mathsf{T} \\ \tilde{\mathbf{s}}_1 \mathbf{a}_i^\mathsf{T} + \mathbf{E}_i \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times t} \\ -\tilde{s}_0 \mathbf{u}_i \mathbf{g}_z^\mathsf{T} \end{bmatrix} \in R_q^{(n+1)\times t}.$$
Output the packing key $\mathsf{pk} = (\mathbf{P}_1, \ldots, \mathbf{P}_n)$.

- $\mathsf{Pack}(\mathsf{pk}, \mathbf{c}_1 \dots \mathbf{c}_{n^2})$: On input the packing key $\mathsf{pk} = (\mathbf{P}_1, \dots, \mathbf{P}_n)$ and a collection of Regev encodings $\mathbf{c}_1, \dots, \mathbf{c}_{n^2} \in R_q^2$, write $\mathbf{c}_i = (c_{i,1}, c_{i,2})$ for all $i \in [n^2]$. Output the encoding

$$\mathbf{C} = \sum_{i \in [n]} \sum_{j \in [n]} \left(\mathbf{P}_i \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) + [0 \mid c_{i \cdot n+j,2} \mathbf{u}_i^\mathsf{T}]^\mathsf{T} \right) \mathbf{u}_j^\mathsf{T} \in R_q^{(n+1) \times n}.$$

Theorem 4.7 (Packing). *For each $i \in [n^2]$, let $\mathbf{c}_i \in R_q^2$ be a Regev encoding of $\mu_i \in R_q$ with respect to a secret key $\mathbf{s}_0 = [-\tilde{s}_0 \mid 1]^\mathsf{T}$ and error e_i. Take any secret key $\mathbf{S}_1 = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\mathsf{T} \in R_q^{(n+1) \times n}$ and decomposition base $z \in \mathbb{N}$. Let $t = \lfloor \log_z q \rfloor + 1$. Let $\mathsf{pk} \leftarrow \mathsf{PackSetup}(\mathbf{s}_0, \mathbf{S}_1, z)$ and $\mathbf{C} \leftarrow \mathsf{Pack}(\mathsf{pk}, \mathbf{c}_1, \dots, \mathbf{c}_{n^2})$. Then, $\mathbf{C}$ is a Regev encoding of the matrix $\mathbf{M} \in R_q^{n \times n}$ with error $\mathbf{E}$, where $\mathbf{M} = \sum_{i \in [n]} \sum_{j \in [n]} \mu_{i \cdot n+j} \mathbf{u}_i \mathbf{u}_j^\mathsf{T}$ and $\mathbf{E}$ satisfies the following properties:*

- *If χ in* $\mathsf{PackSetup}$ *is B-bounded and setting $e_{\max} = \max_{i \in [n^2]} \|e_i\|_\infty$, then $\|\mathbf{E}\|_\infty \le e_{\max} + dntBz/2$; and*
- *If each e_i is subgaussian with parameter σ_e and χ is subgaussian with parameter σ_χ, then under the independence heuristic (Remark 2.18), the components of $\mathbf{E}$ are subgaussian with parameter σ_E where $\sigma_E^2 = \sigma_e^2 + dntz^2\sigma_\chi^2/4$.*

Proof. By assumption, for each $i \in [n^2]$, $\mathbf{s}_0^\mathsf{T} \mathbf{c}_i = -\tilde{s}_0 c_{i,1} + c_{i,2} = \mu_i + e_i$. In addition, for all $i \in [n]$,

$$\mathbf{P}_i = \begin{bmatrix} \mathbf{a}_i^\mathsf{T} \\ \tilde{\mathbf{s}}_1 \mathbf{a}_i^\mathsf{T} + \mathbf{E}_i \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1 \times t} \\ -\tilde{s}_0 \mathbf{u}_i \mathbf{g}_z^\mathsf{T} \end{bmatrix},$$

where $\mathbf{E}_i \leftarrow \chi^{n \times t}$ and for some $\mathbf{a}_i \in R_q^t$. Then,

$$\begin{aligned}
\mathbf{S}_1^\mathsf{T} \mathbf{C} &= \sum_{i \in [n]} \sum_{j \in [n]} \mathbf{S}_1^\mathsf{T} \left(\mathbf{P}_i \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) + [0 \mid c_{i \cdot n+j,2} \mathbf{u}_i^\mathsf{T}]^\mathsf{T} \right) \mathbf{u}_j^\mathsf{T} \\
&= \sum_{i \in [n]} \sum_{j \in [n]} \mathbf{E}_i \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) \mathbf{u}_j^\mathsf{T} - \tilde{s}_0 c_{i \cdot n+j,1} \mathbf{u}_i \mathbf{u}_j^\mathsf{T} + c_{i \cdot n+j,2} \mathbf{u}_i \mathbf{u}_j^\mathsf{T} \\
&= \sum_{i \in [n]} \sum_{j \in [n]} \mathbf{E}_i \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) \mathbf{u}_j^\mathsf{T} + (\mu_{i \cdot n+j} + e_{i \cdot n+j}) \mathbf{u}_i \mathbf{u}_j^\mathsf{T} \\
&= \mathbf{M} + \underbrace{\sum_{i \in [n]} \sum_{j \in [n]} e_{i \cdot n+j} \mathbf{u}_i \mathbf{u}_j^\mathsf{T} + \mathbf{E}_i \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) \mathbf{u}_j^\mathsf{T}}_{\mathbf{E}}.
\end{aligned}$$

Thus, $\mathbf{C}$ is an encoding of $\mathbf{M}$ with error $\mathbf{E}$. To bound the error components, we use the fact that $\mathbf{u}_j$ is a basis vector so $\left\| \sum_{j \in [n]} \mathbf{g}_z^{-1}(\mathbf{c}_{i \cdot n+j}) \mathbf{u}_j^\mathsf{T} \right\|_\infty \le z/2$. Next, since χ is B-bounded, the entries of $\mathbf{E}_i$ are correspondingly B-bounded, so

$$\|\mathbf{E}\|_\infty \le e_{\max} + \left\| \sum_{i \in [n]} \mathbf{E}_i \left(\sum_{j \in [n]} \mathbf{g}_z^{-1}(c_{i \cdot n+j,1}) \mathbf{u}_j^\mathsf{T} \right) \right\|_\infty \le e_{\max} + \gamma_R ntBz/2.$$

The claim now follows since $\gamma_R = d$. When the components of e and $\mathbf{E}_i$ are independent subgaussians, and appealing also to the independence heuristic, then by the same calculation as above, we conclude that the components of $\mathbf{E}$ are subgaussian with parameter σ_E where $\sigma_E^2 = \sigma_e^2 + dntz^2\sigma_\chi^2/4$. □

The SpiralPack protocol. We now describe the SpiralPack variant of Spiral that leverages the above packing transformation to achieve higher throughput and rates at the expense of larger public parameters and a larger minimum database element size.

Construction 4.8 (SpiralPack). SpiralPack use the same setup as Spiral (Construction 4.1), except we first split each record in the database $\mathcal{D}$ into T_{pack}^2 blocks of equal size, where $T_{\mathsf{pack}} \in \mathbb{N}$ is the packing dimension. We construct T_{pack}^2 sub-databases $\mathcal{D}_1 \dots \mathcal{D}_{T_{\mathsf{pack}}^2}$, where $\mathcal{D}_i$ contains the i^{th} block of each record in $\mathcal{D}$. Let (Spiral.Setup, Spiral.Query, Spiral.Answer, Spiral.Extract) be the algorithms from Construction 4.1. The SpiralPack protocol is defined as follows:

- $\mathsf{Setup}(1^\lambda, 1^N)$: On input the security parameter $\lambda \in \mathbb{N}$ and the number of records in the database, run $(\mathsf{pp}_{\textsc{Spiral}}, \mathsf{qk}_{\textsc{Spiral}}) \leftarrow \textsc{Spiral}.\mathsf{Setup}(1^\lambda, 1^N)$ with plaintext dimension $n = 1$. Using the optimization from Remark 4.2, the querying key $\mathsf{qk}_{\textsc{Spiral}}$ in this case consists of a *single* encoding key $\mathbf{s} \in R_q^2$. Next, sample a matrix encoding key $\mathbf{S} \xleftarrow{R} \mathsf{KeyGen}(1^\lambda, 1^{T_{\mathsf{pack}}})$ that will be used for packing, and construct the packing key $\mathsf{pk} \leftarrow \mathsf{PackSetup}(\mathbf{s}, \mathbf{S}, z_{\mathsf{conv}})$. Output $\mathsf{pp} = (\mathsf{pp}_{\textsc{Spiral}}, \mathsf{pk})$ and $\mathsf{qk} = (\mathsf{qk}_{\textsc{Spiral}}, \mathbf{S}) = (\mathbf{s}, \mathbf{S})$.
- $\mathsf{Query}(\mathsf{qk}, \mathsf{idx})$: On input the querying key $\mathsf{qk} = (\mathsf{qk}_{\textsc{Spiral}}, \mathbf{S})$ and the index $\mathsf{idx} \in [N]$, output the query $\mathsf{q} \leftarrow \textsc{Spiral}.\mathsf{Query}(\mathsf{qk}_{\textsc{Spiral}}, \mathsf{idx})$.
- $\mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q})$: On input the public parameters $\mathsf{pp} = (\mathsf{pp}_{\textsc{Spiral}}, \mathsf{pk})$, the database $\mathcal{D} = (\mathcal{D}_1, \ldots, \mathcal{D}_{T_{\mathsf{pack}}^2})$, and the query q, the answer algorithm essentially runs $\textsc{Spiral}.\mathsf{Answer}$ on each of the T_{pack}^2 databases $\mathcal{D}_1, \ldots, \mathcal{D}_{T_{\mathsf{pack}}^2}$ and then packs the T_{pack}^2 responses together into a single matrix Regev encoding. Note that the query expansion only needs to be done *once*. More precisely, the answer algorithm operates as follows:
  1. **Query expansion:** This is the same procedure as in $\textsc{Spiral}.\mathsf{Answer}(\mathsf{pp}_{\textsc{Spiral}}, \cdot, \cdot)$.
  2. **Response generation:** For each $i \in [T_{\mathsf{pack}}^2]$, process the first dimension and the subsequent dimensions on database $\mathcal{D}_i$ according to the specification in $\textsc{Spiral}.\mathsf{Answer}$. The same (expanded) query is used to process each $\mathcal{D}_i$. This yields a collection of encodings $\mathbf{c}_1, \ldots, \mathbf{c}_{T_{\mathsf{pack}}^2} \in R_q^2$.
  3. **Packing:** Compute the packed encoding $\mathbf{C} \leftarrow \mathsf{Pack}(\mathsf{pk}, \mathbf{c}_1 \ldots \mathbf{c}_{T_{\mathsf{pack}}^2}) \in R_q^{(T_{\mathsf{pack}}+1) \times T_{\mathsf{pack}}}$.
  4. **Modulus switching:** Given the final packed encoding $\mathbf{C}$, output the response $\mathsf{r} \leftarrow \mathsf{ModulusSwitch}_{q_1,q_2}(\mathbf{C})$.
- $\mathsf{Extract}(\mathsf{qk}, \mathsf{st}, \mathsf{r})$: On input the query key $\mathsf{qk} = (\mathbf{s}, \mathbf{S})$, an (empty) query state st, and the server response r, output $\mathsf{Decode}(\mathsf{Recover}_{q_1,q_2}(\mathbf{S}, \mathsf{r}))$.

Correctness. Correctness of SpiralPack essentially follows from the same analysis as in the proof of Theorem 4.5. We provide a sketch of the argument here. The only difference in SpiralPack is the additional packing transformation applied at the very end of the answer algorithm. Let $\mathcal{D} = (d_1, \ldots, d_N)$ and let $\mathsf{idx} \in [N]$ be an arbitrary index. Suppose $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^N)$, $\mathsf{q} \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx})$. Consider the behavior of $\mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q})$. By the same analysis as in the proof of Theorem 4.5, each encoding $\mathbf{c}_i$ output by the response-generation step is a Regev encoding of $\lfloor q/p \rfloor\, d_{\mathsf{idx},i}$ with error $\mathbf{e}_i$, where $d_{\mathsf{idx},i}$ denotes the i^{th} block of record d_{idx} and

$$\|\mathbf{e}_i\|_\infty = O\left(2^{2\nu_1} d^2 pBz \log q + \nu_2^2 B^2 d^3 z^2 \log^3 q\right).$$

Note here that we are instantiating the base protocol with $n = 1$. Now, by Theorem 4.7, the packing step introduces an additional noise to the final encoding that is bounded by $O(dT_{\mathsf{pack}} Bz \log q)$. Then, by Theorem 3.4, after modulus switching, the noise $\mathbf{E}$ in the final encoding $\mathbf{Z} \leftarrow \mathsf{Recover}(\mathbf{S}, \mathsf{r})$ is bounded by

$$\|\mathbf{E}\|_\infty = O\left(\frac{q_1}{q}\left(2^{2\nu_1} d^2 pBz \log q + \nu_2^2 B^2 d^3 z^2 \log^3 q + dT_{\mathsf{pack}} Bz \log q + p\right) + \frac{q_1}{q_2} dB + p\right).$$

By Theorem 2.11, $\mathsf{Decode}(\mathbf{Z}) = d_{\mathsf{idx}}$ as long as $\|\mathbf{E}\|_\infty + (q_1 \bmod p) \le q_1/2p$. Thus, it suffices to set

$$\Omega\big(dpBz \log q (T_{\mathsf{pack}} + 2^{2\nu_1} dp + \nu_2^2 Bd^2 z \log^2 q)\big) \quad \text{and} \quad q_2 = \Omega(dBp) \quad \text{and} \quad q_1 = \Omega(p^2).$$

Security. Security follows by an analogous argument as the proof of Theorem 4.6. In this case, the only additional component added to the public parameters pp is the packing key pk. By construction, pk consists of encodings of the *source* key $\mathbf{s}$ under the destination key $\mathbf{S}$. Since $\mathbf{s}$ and $\mathbf{S}$ are sampled independently, we can appeal to CPA security of the Regev encoding scheme to argue that the components in pk are computationally indistinguishable from encodings of random matrices. The remaining components of pp are common to Spiral and those are computationally indistinguishable from encodings of random values assuming KDM-security for Regev encodings (see Theorem 4.6).

5 Implementation and Evaluation

In this section, we describe the implementation of the Spiral system as well as our automated parameter selection procedure. We conclude with a detailed experimental evaluation.

5.1 Automatic Parameter Selection

To select parameters for Construction 4.1, we start by analyzing the noise growth from the homomorphic operations. As discussed in Remark 2.18, when selecting concrete parameters, we rely on the independence heuristic and bound the *variance* of the noise rather than rely on the worst-case bounds from Theorem 4.5. This allows us to use smaller lattice parameters in our instantiation at the expense of a possibly larger correctness error. In Section 5.3, we validate the independence heuristic by comparing the actual noise magnitude to the noise level predicted under our heuristic noise model. Next, we describe the computation/communication trade-offs associated with the system parameters. We empirically measure the costs of the main steps in the response-generation as a function of the different underlying parameters and use this to construct a heuristic model for the computational cost for a given parameter instantiation. We then search over the candidate parameter sets to identify the most suitable setting for a target database.

Noise analysis. Suppose the error distribution χ in Construction 4.1 is B-bounded and subgaussian with parameter σ. Under the independence heuristic (Remark 2.18), we model the error in the server's response as a subgaussian random variable. In the following, we provide a precise characterization of the error in the response as opposed to the asymptotic one from Theorem 4.5 since we will use this analysis to select the concrete parameters for Spiral.

- **Query:** The initial error $\mathbf{e}$ in the client's encoding $\mathbf{q}$ is sampled from χ, so it is subgaussian with parameter σ.
- **Query expansion:** We consider the Regev and GSW encodings separately. As described in Remark 4.3, we use two different decomposition bases $z_{\mathsf{coeff,Reg}}$ and $z_{\mathsf{coeff,GSW}}$ for expanding the components for the Regev encodings and those for the GSW encodings, respectively.
  - By Theorem A.4, the error $\mathbf{e}_i^{(\mathrm{Reg})}$ associated with each encoding $\mathbf{c}_i^{(\mathrm{Reg})}$ is subgaussian with parameter $\hat{\sigma}^{(\mathrm{Reg})}$ where $\left(\hat{\sigma}^{(\mathrm{Reg})}\right)^2 = 4^{\nu_1+1}\sigma^2(1+dt_{\mathsf{coeff,Reg}}z_{\mathsf{coeff,Reg}}^2/3)$, and $t_{\mathsf{coeff,Reg}} = \lfloor \log_{z_{\mathsf{coeff,Reg}}} q \rfloor + 1$. By Theorem 3.1, each $\mathbf{E}_i^{(\mathrm{Reg})}$ is subgaussian with parameter $\sigma^{(\mathrm{Reg})}$ where
$$\left(\sigma^{(\mathrm{Reg})}\right)^2 = \left(\hat{\sigma}^{(\mathrm{Reg})}\right)^2 + dt_{\mathsf{conv}}z_{\mathsf{conv}}^2\sigma^2/4 = \sigma^2\left(4^{\nu_1+1}(1+dt_{\mathsf{coeff,Reg}}z_{\mathsf{coeff,Reg}}^2/3) + dt_{\mathsf{conv}}z_{\mathsf{conv}}^2/4\right),$$
    and $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1$.
  - Again by Theorem A.4, the error $\mathbf{e}_i^{(\mathrm{GSW})}$ associated with each encoding $\mathbf{c}_i^{(\mathrm{GSW})}$ is subgaussian with parameter $\hat{\sigma}^{(\mathrm{GSW})}$ where
$$\left(\hat{\sigma}^{(\mathrm{GSW})}\right)^2 \leq 4(t_{\mathsf{GSW}}\nu_2+1)^2\sigma^2(1+dt_{\mathsf{coeff,GSW}}z_{\mathsf{coeff,GSW}}^2/3),$$
    and $t_{\mathsf{coeff,GSW}} = \lfloor \log_{z_{\mathsf{coeff,GSW}}} q \rfloor + 1$. By Theorem 3.2, each $\mathbf{E}_i^{(\mathrm{GSW})}$ is subgaussian with parameter $\sigma^{(\mathrm{GSW})}$ where
$$\begin{aligned}\left(\sigma^{(\mathrm{GSW})}\right)^2 &= \left(\hat{\sigma}^{(\mathrm{GSW})}\right)^2 dB^2 + t_{\mathsf{conv}}d\sigma^2 z_{\mathsf{conv}}^2/2 \\ &= d\sigma^2\left(4B^2(t_{\mathsf{GSW}}\nu_2+1)^2(1+dt_{\mathsf{coeff,GSW}}z_{\mathsf{coeff,GSW}}^2/3) + t_{\mathsf{conv}}z_{\mathsf{conv}}^2/2\right).\end{aligned}$$
- **Processing the first dimension:** Under the independence heuristic and Theorem 2.12, the noise $\mathbf{E}_j^{(0)}$ in each encoding $\mathbf{C}_j^{(0)}$ is subgaussian with parameter $\sigma^{(0)}$ where $\left(\sigma^{(0)}\right)^2 = 2^{\nu_1}nd(p/2)^2\left(\sigma^{(\mathrm{Reg})}\right)^2$.

- **Folding in the subsequent dimensions:** Under the independent heuristic, Theorem 2.19, and appealing to a similar argument as in the proof of Theorem 4.5, the noise $\mathbf{E}_j^{(r)}$ in each encoding $\mathbf{C}_j^{(r)}$ is subgaussian with parameter $\sigma^{(r)}$ where $\left(\sigma^{(r)}\right)^2 = \left(\sigma^{(r-1)}\right)^2 + dm_{\mathsf{GSW}} z_{\mathsf{GSW}}^2 \left(\sigma^{(\mathsf{GSW})}\right)^2/2$. Correspondingly, the noise in the final encoding $\mathbf{E}_0^{(\nu_2)}$ is subgaussian with parameter $\sigma^{(\nu_2)}$ where

$$\left(\sigma^{(\nu_2)}\right)^2 = 2^{\nu_1} nd(p/2)^2 \left(\sigma^{(\mathsf{Reg})}\right)^2 + \nu_2 dm_{\mathsf{GSW}} z_{\mathsf{GSW}}^2/2 \cdot \left(\sigma^{(\mathsf{GSW})}\right)^2$$

- **Modulus switching**: By Theorem 3.4, the final error $\mathbf{E}$ can be written as $\mathbf{E} = \mathbf{E}_1 + \mathbf{E}_2$ where $\|\mathbf{E}_1\|_\infty \leq ((q_1 \bmod p) + (q_1/q)(q \bmod p) + 2)/2$ and the components of $\mathbf{E}_2$ are subgaussian with parameter σ_2 where $\sigma_2^2 = (q_1/q)^2 \left(\sigma^{(\nu_2)}\right)^2 + (q_1/q_2)^2 \sigma^2 d/4$.

Let $C > 0$ be a correctness parameter. By a union bound over the entries of $\mathbf{E}_2$, with probability $1 - 2dn^2 \exp(-\pi C^2)$, $\|\mathbf{E}_2\|_\infty \leq C\sigma_2$ and correspondingly,

$$\|\mathbf{E}\|_\infty \leq \frac{1}{2}\left((q_1 \bmod p) + \frac{q_1}{q}(q \bmod p) + 2\right) + C\sigma_2. \tag{5.1}$$

To ensure correct decoding (Theorem 2.11), we now choose the parameters such that $\|\mathbf{E}\|_\infty + (q_1 \bmod p) < q_1/2p$. Finally, we define the correctness error $\varepsilon_{\mathsf{corr}}$ to be $\varepsilon_{\mathsf{corr}} = 2dn^2 \exp(-\pi C^2)$.

Parameter selection trade-offs. We now describe our general methodology for selecting parameters to support a database $\mathcal{D}$ with up to N records, where each record is at most S bits. The parameters of interest in Construction 4.1 are the lattice parameters d, q, χ, the plaintext modulus p, the plaintext dimension n, the database configuration ν_1, ν_2, the decomposition bases $z_{\mathsf{coeff,Reg}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}}$ (see Remark 4.3), and the correctness parameter C. A single invocation of the PIR protocol yields an element of $R_p^{n \times n}$, which encodes $dn^2 \log p$ bits. When the record size S satisfies $S > dn^2 \log p$, we break each record into $T \geq S/(dn^2 \log p)$ blocks, each of size $dn^2 \log p$. We then construct T databases where the i^{th} database contains the i^{th} block of each record and run the Answer protocol T times to compute the response. Importantly, the query expansion step only needs to be computed *once* in this case since the same query is applied to each of the T databases. The subsequent homomorphic evaluation is performed over each of the T databases. Our goal is to choose parameters that minimize the estimated cost of the protocol (estimated based on current AWS computing costs and the total computation and communication required of the protocol; see Section 5.2). We choose parameters to tolerate a correctness error $\varepsilon_{\mathsf{corr}} = 2dn^2 \exp(-\pi C^2)$ of at most 2^{-40} and a security level of 128 bits of (classical) security.

- **Lattice parameters.** Throughout this work, we work over a power-of-two cyclotomic ring, so the ring dimension d is a power of two. Since the computational cost of multiplications in $R_q = \mathbb{Z}_q[x]/(x^d + 1)$ is super-linear in the ring dimension, and the size of the modulus also increases with d, we choose the minimal value of d that suffices for correctness and security. The modulus q must be large enough to ensure correctness. This in turn translates to a lower bound on d to ensure security. In our setting, we set $d = 2^{11} = 2048$, which allows us to choose a 56-bit modulus q. This is sufficient to efficiently support databases with up to 2^{22} records. We use a standard discrete Gaussian noise distribution χ with mean 0 and width $\sigma = 6.4$. This ensures 128 bits of classical security, as estimated by the LWE estimator [APS15] (based on the algorithms from [CN11, AG11, BDGL16]).
- **Database configuration.** When the records are large (i.e., $S \geq dn^2 \log p$), each plaintext element can encode at most one record. In this case, we require $2^{\nu_1+\nu_2} \geq N$. When the database records are small (i.e., $S < dn^2 \log p$), we can pack multiple records into a single plaintext.[8] In this case, it suffices to choose ν_1 and ν_2 so that $2^{\nu_1+\nu_2} dn^2 \log p \geq NS$. The size of the first dimension ν_1 determines the number of rounds of query expansion and the noise growth in the query expansion scales exponentially with ν_1. The number of subsequent dimensions ν_2 determines the number of rounds of folding needed during query processing (Step 3 of the Answer algorithm in Construction 4.1).

[8] Of course, the overall rate is smaller with packing. The highest rate we can achieve corresponds to the case where the size of each record is at least as large as an element of $R_p^{n \times n}$.

- **Plaintext dimension.** A larger plaintext dimension n translates to a higher rate. For sufficiently large records ($S \geq dn^2 \log p$) and leveraging modulus switching (with reduced moduli q_1 and q_2), the ratio of the record size to the response size is

$$\text{rate} = \frac{\text{record size}}{\text{response size}} = \frac{n^2 \log p}{n \log q_2 + n^2 \log q_1}. \tag{5.2}$$

  However, the per-record computational cost also scales linearly with the dimension n. In particular, each homomorphic operation in the first dimension processing pass (Step 2 of the Answer algorithm in Construction 4.1) requires computing $N \cdot O(n^2(n+1))$ ring operations for a database of size $N \cdot O(n^2 \log p)$.

  For the base SPIRAL protocol, we set $n = 2$ which provides a compelling trade-off between computation and communication. Compared to the case where $n = 1$, this yields a 33% reduction in communication at the expense of a 33% increase in computation. Under our cost model based on a deployment over AWS, the communication cost is often higher than the compute cost, especially for streaming applications.

  As discussed in Section 4.1, we also consider a packed variant SPIRALPACK where we use $n = 1$ for query processing and a much larger dimension $n > 2$ for response packing.

- **Plaintext and encoding modulus.** From our noise analysis above (Eq. (5.1); see also Theorem 4.5), the error in the final set of encodings depends on $(q_1 \bmod p)$ and $q_1/q \cdot (q \bmod p)$, where p is the plaintext modulus and q_1 is one of the moduli used for modulus switching. As mentioned above, we set $q = 2^{56}$ for all of our instantiations. In our construction, we choose $q_1 = 4p$ so $q_1 \bmod p = 0$. We then search over a range of possible values for p (and compute the noise variance according to Eq. (5.1)). Using a large p leads to higher noise growth (since the noise in the first dimension is scaled by p) but enables a higher rate and higher throughput (since more plaintext data is processed per operation on the encoded data).

  We set q_1 and q_2 as *small* as possible to maximize the rate (see Eq. (5.2)). Throughout this work, we set $q_1 = 4p$ which is the smallest value that ensures correctness (for our choice of parameters) and has the added benefit that $q_1 \bmod p = 0$. Finally, we choose q_2 to be the smallest value that satisfies the correctness requirement.

- **Decomposition bases.** Using larger decomposition bases $z_{\text{coeff,Reg}}$ and z_{conv} increases the amount of noise introduced by the associated transformation, but reduces the amount of computation in the query expansion step (i.e., the dimensions of $\mathbf{G}_{n,z}$ scales with $n \log_z q$ so larger values of z corresponds to a smaller gadget matrix $\mathbf{G}_{n,z}$). We additionally observe that the decomposition base $z_{\text{coeff,GSW}}$ used to expand the GSW encodings has a minimal impact on computation, since $\nu_2 t_{\text{GSW}} \ll 2^{\nu_1}$. For this reason, we fix $z_{\text{coeff,GSW}} = 2$ which minimizes the noise growth from the GSW expansion step.

- **GSW gadget base.** Using a larger decomposition base z_{GSW} for the GSW encodings increases the noise growth from homomorphic multiplications (in the folding steps), but reduces the computational cost in both the query expansion and the folding steps. Moreover, to pack the query into a *single* scalar Regev encoding, we require that $2^{\nu_1} + \nu_2 t_{\text{GSW}} \leq d$. Otherwise, the client cannot pack all of the components of the query into a single encoding and would have to send multiple Regev encodings on each query.

Automatic parameter selection. Balancing the different scheme parameters is important for obtaining a good trade-off between computational costs and communication. Similar to XPIR [MBFK16], we introduce a heuristic search algorithm for parameter selection based on a given database configuration (i.e., the number of records N and the record size S). As described above, we set the ring dimension $d = 2048$ and use a 56-bit encoding modulus q. This ensures 128 bits of security and suffices to support databases of size $N \leq 2^{22}$. As noted above, for the base SPIRAL protocol, we set the plaintext dimension to $n = 2$. It then suffices to choose the plaintext modulus p, the decomposition dimensions $t_{\text{coeff,Reg}}$, $t_{\text{coeff,GSW}}$, t_{conv}, t_{GSW}, database configuration ν_1, ν_2, and the number of executions T. For each of these parameters, there is a small number of reasonable values, and we can quickly search over *all* of the candidate configurations.

We set the plaintext modulus p to be a power of two with maximum value 2^{30}. Using larger p would require using a larger modulus q and ring dimension $d \geq 4096$. We consider $t_{\text{coeff,Reg}}, t_{\text{conv}} \in \{2, 4, 8, 16, 32, 56\}$ and $t_{\text{GSW}} \in [2, 56]$.[9] Recall from above that we fix the decomposition base $z_{\text{coeff,GSW}} = 2$, which fixes the dimension $t_{\text{coeff,GSW}} = 56$. Finally, we consider all database configurations $\nu_1, \nu_2 \in [2, 11]$.[10] With these constraints, there are ≈3 million candidate parameter sets for each database setting. After pruning out parameter settings where the correctness error exceeds the threshold (2^{-40}), we are left with ≈700,000 parameter sets. This initial pruning step takes 3 minutes on our benchmarking platform, and the pruned set of feasible parameters can be cached in a single 40 MB file.

We now need a way to estimate the concrete performance (e.g., server computation time) for each set of candidate parameters. We do so by developing an empiric model based on concrete measurements for the different components of the server computation:

- For typical configurations, the cost of the coefficient expansion (Algorithm 1) is dominated by the cost of expanding the encodings for the first dimension (since $2^{\nu_1} \gg t_{\text{GSW}}\nu_2$). To model the computational cost, we just focus on the cost of expanding the first dimension, which is a function of ν_1 and t_{coeff} only.[11] There are only 60 possible combinations for ν_1 and $t_{\text{coeff,Reg}}$, so we precompute a look-up table with the running times for each candidate parameter setting.
- During the encoding translation steps, we call ScalToMat 2^{ν_1} times and RegevToGSW $t_{\text{GSW}}\nu_2$ times. Asymptotically, the running time scales linearly with the variables 2^{ν_1}, $t_{\text{GSW}}\nu_2$, t_{conv}, and t_{GSW}. We fit a linear model based on the measured running times for a small set of candidate parameters to estimate the concrete running time.
- For the cost of processing the first dimension (Step 2 of the Answer algorithm in Construction 4.1), the cost is a linear function of the database configuration 2^{ν_1}, 2^{ν_2}, and $2^{\nu_1+\nu_2}$. We again use a linear model to estimate the concrete running time as a function of ν_1 and ν_2.
- For the folding step (Step 3 of the Answer algorithm in Construction 4.1), the running time of a single folding step is linear in t_{GSW}. We fit a linear model to predict the concrete running time of a single folding step and scale the result by the total number of folding steps ν_2.
- Finally, we compute the minimum number of executions $T = \lceil S/n^2 d \log p \rceil$ needed to serve a record of size at least S. This yields an estimate on the overall server running time.

Using the above models, we can efficiently estimate the concrete running time for each parameter setting. Applying the AWS monetary cost model (see Section 5.2) for CPU time and network download, we then select the parameter setting that minimizes the server's *total cost* to answer a query. The search process takes about 10 seconds on our platform. For all of the parameter sets selected using this approach, the estimated server computation time is within 10% of the actual measured running time. We provide sample parameters chosen by our procedure in Table 1. We use an almost identical procedure to select parameters for SpiralPack. In this setting, we set the initial plaintext dimension to $n = 1$ and introduce an additional packing dimension parameter T_{pack} to the search procedure. We then estimate the extra noise introduced by packing using the bounds from Theorem 4.7, and then select parameters that minimize the server cost while ensuring the target level of correctness.

Remark 5.1 (Other Optimization Objectives)**.** By default, we configure our parameter-selection method to minimize the total cost on an AWS-based deployment. However, the system naturally supports optimizing other objectives such as minimizing the estimated server computation time or to maximize the rate. We also support selecting parameter sets with a size constraint on the public parameter size or the query size. This provides a way to systematically explore different trade-offs in the final protocol. We elaborate on some of these trade-offs in Section 5.3.

[9] While we could also consider the full range of values for $t_{\text{coeff,Reg}}, t_{\text{conv}}$, this would increase the size of our search space by ≈ 100×. In our experiments, we did not observe a significant benefit to the overall system efficiency with the expanded search space.

[10] Our vectorized implementation for processing the first dimension requires that $\nu_1 > 1$. We exclude $\nu_2 = 1$ because this settings makes it infeasible to pack all of the query coefficients into a small number of ciphertexts for even a moderate-size database with just a few thousand records.

[11] For cases where ν_1 is small and $t_{\text{GSW}}\nu_2 > 2^{\nu_1}$, this model *underestimates* the cost of coefficient expansion. However, even with this underestimation, the parameter selection tool does not favor such configurations, as they lead to an imbalanced, and thereby suboptimal, configuration. Thus, for modeling simplicity, we focus solely on the cost of expanding the components in the first dimension.

Database	$\log p$	$\log q_2$	$t_{\text{coeff,Reg}}$	t_{conv}	t_{GSW}	(ν_1, ν_2)	T	Rate	Estim. CPU	Actual CPU
$2^{20} \times 256$B	8	21	8	4	9	(9, 6)	1	0.0122	1.68 s	1.69 s
$2^{14} \times 100$KB	9	21	16	4	10	(9, 5)	11	0.4129	5.03 s	4.92 s

Table 1: Parameter sets for SPIRAL chosen by our search procedure for two different database configurations. We use a plaintext dimension $n = 2$, a ring dimension $d = 2048$, an encoding modulus q where $\log q = 56$, a discrete Gaussian error distribution χ with mean 0 and width $\sigma = 6.4$, and decomposition dimension $t_{\text{coeff,GSW}} = 56$ for all settings. As described in Section 5.1, p is the plaintext modulus, q_1, q_2 are the reduced moduli (with $q_1 = 4p$), $t_{\text{coeff,Reg}}$, t_{conv}, t_{GSW} are the decomposition dimensions for the various ciphertext translation algorithms, ν_1, ν_2 is the database configuration, and T is the number of executions. The parameters are chosen to provide 128 bits of classical security [APS15] and ensure a correctness error of at most 2^{-40}. The "Rate" is the ratio of the PIR response size to the database record size. The "Estim. CPU" column gives the estimated time in seconds to process a single query according to our model described in Section 5.1 and the "Actual CPU" column gives the actual computation time taken as measured on our experimental setup.

5.2 Implementation and Experimental Setup

We now describe some system optimizations used in our implementation as well as our experimental setup.

SPIRAL configurations. The vanilla version of SPIRAL is designed to be a general-purpose PIR protocol. However, in a *streaming* setting, the SPIRALSTREAM variant of SPIRAL (Remark 4.4) can achieve even better performance. In our experimental evaluation (Section 5.3), we consider both a static setting and a streaming setting:

- **Static setting:** This is the basic setting where the client privately retrieves a single record from a database. For this setting, we choose the parameters to balance query size, response size, and the server computation time. This is the default operating mode of SPIRAL (and its packed version, SPIRALPACK).
- **Streaming setting:** In the streaming setting, a client uploads a single query that is *reused* across many databases. This captures two general settings: (1) applications with large records that we want to consume progressively (e.g., a private video streaming service like Popcorn [GCM+16]); and (2) metadata-hiding messaging systems where a user is repeatedly reading from a "mailbox" (e.g. Pung [AS16] or Addra [AYA+21]). Since the query can be *reused* in the streaming setting, we can amortize the cost of transmitting the query over the lifetime of the stream. Systems like FastPIR [AYA+21] are designed specifically for the streaming setting and as such, achieve higher server throughput compared to SealPIR [ACLS18], but require larger queries. We can easily adapt SPIRAL to the setting using the approach from Remark 4.4. Namely, in SPIRALSTREAM, the client uploads all of the Regev encodings directly without using the query packing approach from [ACLS18]. As we show in Section 5.3, SPIRALSTREAM has larger queries, but achieves a much better rate and server throughput. We define the streaming version of SPIRALPACK analogously and refer to the resulting scheme as SPIRALSTREAMPACK.

We use our automatic parameter selection tool (Section 5.1) to select parameters for all of the SPIRAL variants.

Compressing Regev encodings. In SPIRAL (and all of its variants), the PIR query consists of one or more scalar Regev encodings. A scalar Regev encoding $\mathbf{c}$ is a pair $\mathbf{c} = (c_0, c_1)$, where $c_0 \in R_q$ is uniformly random. Instead of sending c_0, the client can instead send a seed s for a pseudorandom generator (PRG) and derive c_0 by evaluating the PRG on the seed s. Security holds if we model the PRG as a random oracle. This is a standard technique to compress Regev encodings [Gal13, BCD+16, ISW21].

Modulus choice. In our implementation, we use a 56-bit modulus q that is a product of two 28-bit primes α, β. By the Chinese remainder theorem (CRT), $R_q \cong R_\alpha \times R_\beta$. We implement arithmetic operations in R_α and R_β using native 64-bit arithmetic. We choose $\alpha, \beta = 1 \bmod 2d$ so $\mathbb{Z}_\alpha$ and $\mathbb{Z}_\beta$ have a subgroup of size $2d$ (i.e., the $(2d)^{\text{th}}$ roots of unity). Polynomial multiplication in R_α and R_β can be efficiently implemented using a standard "nega-cyclic" fast Fourier

transform (also called the number-theoretic transform (NTT)) [LMPR08, LN16]. To allow faster modular reduction, we also choose α, β to be of the form $2^i - 2^j + 1$ for integers i, j where $2^i > 2^j > 2d$.

Database representation. Database elements in our system are elements of $R_p^{n \times n}$. We represent all ring elements in their evaluation representation (i.e., the FFT/NTT representation). This enables faster homomorphic operations during query processing.

SIMD operations. Like previous constructions [MCR21], we take advantage of the Intel Advanced Vector Extensions (AVX) to accelerate arithmetic operations in R_α and R_β (recall $R_q \cong R_\alpha \times R_\beta$). In particular, we use the AVX2 and AVX-512 instructions when computing the scalar multiplications and homomorphic additions for the first dimension processing in Construction 4.1.

Code. Our implementation consists of roughly 4,000 lines of C++.[12] We adapt the procedure from the SEAL homomorphic encryption library [SEA19] to implement the FFTs for homomorphic evaluation. We use the Intel HEXL library [BKS+21] to implement FFTs in the response decoding procedures.

Experimental setup. We compare our PIR protocol against the public implementations of SealPIR [ACLS18], FastPIR [AYA+21], and OnionPIR [MCR21]. Since the memory requirements vary between protocols, we use an implicit representation of the database across all of our measurements to ensure a consistent comparison. To minimize any variance in running time due to cache accesses, we set the minimal size of the implicitly-represented database to be 1 GB. Based on our measurements, using this implicit database representation only has a small effect on the measurements (at most a 1% difference in server compute time).

We measure the performance of our system on an Amazon EC2 `c5n.2xlarge` instance running Ubuntu 20.04. The machine has 8 vCPUs (Intel Xeon Platinum 8124M @ 3 GHz) and 21 GB of RAM. We use the same benchmarking environment for all experiments, and compile all of the systems using `Clang 12`. The processor supports the AVX2 and AVX-512 instruction sets, and we enable SIMD instruction set support for all systems. We use a single-threaded execution for *all* of our experiments and report running times averaged over a minimum of 5 trials.

Metrics. For each database configuration, we measure the total computation and communication for the client and the server, as well as the size of the public parameters. Similar to previous works [ACLS18, MCR21, AYA+21], we assume the public parameters have been generated and transmitted in a separate offline phase, and focus exclusively on the online computation and communication. This is often justified since the public parameters only needs to be generated once and can be *reused* for many PIR queries.

We also estimate the server's monetary cost to respond to a single query. This is the sum of the server's CPU cost and the cost of the network communication. We estimate these costs based on the current rates for a long-term Amazon EC2 instance: \$0.0195/CPU-hour and \$0.09/GB of outbound traffic at the time of writing [AWS21]. Finally, we report the *rate* of the protocol (i.e., the ratio of the record size to the response size), and the server's throughput (i.e., the number of database bytes the server can process each second). We generally do not report the response-decoding times, since they are very small (Fig. 5).

5.3 Evaluation Results for Spiral

We start by comparing the performance of Spiral and SpiralStream to existing systems on three different database configurations in Table 2:

- A database with many small records (2^{20} records of size 256 B). This is a common baseline for PIR [AS16, ALP+21, AYA+21].
- A database with moderate-size records (2^{18} records of size 30 KB). This is the optimal configuration for OnionPIR [MCR21].

[12] Our implementation is available here: `https://github.com/menonsamir/spiral`.

Database	Metric	SealPIR	FastPIR	MulPIR*	OnionPIR	Spiral	SpiralStream
	Param. Size	3 MB	**1 MB**	-	5 MB	14–18 MB	344 KB–3 MB
$2^{20} \times 256$B (268 MB)	**Query Size**	66 KB	33 MB	122 KB	63 KB	**14 KB**	8 MB
	Response Size	328 KB	66 KB	119 KB	127 KB	21 KB	**20 KB**
	Computation	3.19 s	1.44 s	-	3.31 s	1.69 s	**0.85 s**
	Rate	0.0008	0.0039	0.0024	0.0020	0.0122	**0.0125**
	Throughput	84 MB/s	186 MB/s	-	81 MB/s	159 MB/s	**314 MB/s**
	Server Cost	$0.000047	$0.000014	-	$0.000029	$0.000011	**$0.000006**
$2^{18} \times 30$KB (7.9 GB)	**Query Size**	66 KB	8 MB	-	63 KB	**14 KB**	15 MB
	Response Size	3 MB	262 KB	-	127 KB	84 KB	**62 KB**
	Computation	74.91 s	50.52 s	-	52.73 s	24.46 s	**8.99 s**
	Rate	0.0092	0.1144	-	0.2363	0.3573	**0.4803**
	Throughput	105 MB/s	156 MB/s	-	149 MB/s	322 MB/s	**875 MB/s**
	Server Cost	$0.000701	$0.000297	-	$0.000297	$0.000140	**$0.000054**
$2^{14} \times 100$KB (1.6 GB)	**Query Size**	66 KB	524 KB	-	63 KB	**14 KB**	8 MB
	Response Size	11 MB	721 KB	-	508 KB	242 KB	**208 KB**
	Computation	19.03 s	23.27 s	-	14.38 s	4.92 s	**2.38 s**
	Rate	0.0092	0.1387	-	0.1969	0.4129	**0.4811**
	Throughput	86 MB/s	70 MB/s	-	114 MB/s	333 MB/s	**688 MB/s**
	Server Cost	$0.001076	$0.000191	-	$0.000124	$0.000048	**$0.000032**

* To date, there is not a public implementation of the MulPIR system. Here, we report the query and response sizes on a similar database of size $2^{20} \times 288$B from [ALP+21].

Table 2: Comparison of Spiral and SpiralStream with recent PIR protocols (SealPIR [ACLS18], FastPIR [AYA+21], MulPIR [ALP+21], OnionPIR [MCR21]) on different database configurations. All measurements are collected on the same computing platform using a single-threaded execution. SealPIR and OnionPIR provide 115 and 111 bits of security, respectively. All other schemes provide at least 128 bits of security. The public parameter size ("Param. Size" column) for Spiral (and SpiralStream) varies depending on database configuration and we report the range here. The rate is the ratio of the record size to the response size, the throughput is the ratio of the server's computation time to database size, and the server cost is the estimated monetary cost needed to process a single query based on current AWS prices (see Section 5.2).

- A database with a small number of large records (2^{14} records of size 100 KB).

When the record size is small, all of the lattice-based PIR schemes have low rate. This is because lattice ciphertexts encode a *minimum* of a few KB of data, so there is a significant amount of unused space for small records. When the record size is comparable or greater than the amount of data that can be packed into a lattice ciphertext, the rate essentially becomes the inverse of the ciphertext expansion factor. Due to better control of noise growth, the use of matrix Regev encodings, and improved modulus switching, Spiral and SpiralStream achieve a higher rate than previous implementations of single-server PIR.

In all three settings, Spiral has the smallest query size. For the databases with 30 KB and 100 KB records, Spiral's throughput is at least 2.2× higher than competing schemes (while achieving a higher rate and smaller queries). In the small record case, Spiral's server throughput is only outperformed by FastPIR, which is optimized for the streaming setting and requires a query that is over 2400× larger. The main limitation of Spiral is its larger public parameter size. This is due to the additional keys needed for the query compression approach from Section 3. Note though that these public parameters are *reusable* and the cost of communicating them can be amortized over multiple queries.

Turning next to SpiralStream, we see that it achieves a higher rate and server throughput compared to all previous schemes. For instance, on the database with moderate-size records, SpiralStream achieves a throughput of over 800 MB/s, which is 5.6× higher than the previous state-of-the-art; SpiralStream simultaneously achieves a 2× increase in rate as well. Measured in terms of monetary cost, SpiralStream is 5.4× less expensive compared to OnionPIR for this database configuration. The trade-off is SpiralStream requires larger queries, though this is a less significant factor in streaming settings where the same query is reused across multiple requests.

Database	Metric	Best Previous	Spiral	SpiralStream	SpiralPack	SpiralStreamPack
$2^{20} \times 256$B (268 MB)	**Param. Size**	1 MB	14 MB	**344 KB**	14 MB	16 MB
	Query Size	34 MB	**14 KB**	8 MB	**14 KB**	15 MB
	Response Size	66 KB	21 KB	**20 KB**	**20 KB**	71 KB
	Computation	1.44 s	1.68 s	0.86 s	1.37 s	**0.42 s**
	Rate	0.0039	0.0122	**0.0125**	**0.0125**	0.0036
	Throughput	186 MB/s	159 MB/s	312 MB/s	196 MB/s	**635 MB/s**
$2^{18} \times 30$KB (7.9 GB)	**Param. Size**	5 MB	18 MB	**3 MB**	18 MB	16 MB
	Query Size	63 KB	**14 KB**	15 MB	**14 KB**	30 MB
	Response Size	127 KB	84 KB	**62 KB**	86 KB	96 KB
	Computation	52.99 s	24.52 s	9.00 s	17.69 s	**5.33 s**
	Rate	0.2363	0.3573	**0.4803**	0.3488	0.3117
	Throughput	148 MB/s	321 MB/s	874 MB/s	444 MB/s	**1.48 GB/s**
$2^{14} \times 100$KB (1.6 GB)	**Param. Size**	5 MB	17 MB	**1 MB**	47 MB	24 MB
	Query Size	63 KB	**14 KB**	8 MB	**14 KB**	30 MB
	Response Size	508 KB	242 KB	208 KB	188 KB	**150 KB**
	Computation	14.35 s	4.92 s	2.40 s	4.58 s	**1.21 s**
	Rate	0.1969	0.4129	0.4811	0.5307	**0.6677**
	Throughput	114 MB/s	333 MB/s	683 MB/s	358 MB/s	**1.35 GB/s**

Table 3: Comparison for all four Spiral variants with the best alternative system: FastPIR [AYA+21] for the database with small records ($2^{20} \times 256$B) and OnionPIR otherwise [MCR21].

Packing. In Table 3, we compare the packed versions of Spiral and SpiralStream with the vanilla versions on each of the main benchmarks. As shown in Table 3, packing enables higher rates and throughput, but requires larger public parameter for the packing keys (Section 4.1). For instance, the size of the public parameters ranges from 14–18 MB for Spiral and increases to 14–47 MB for SpiralPack. On the flip side, when considering larger databases, SpiralPack achieves a 30% increase in the rate with comparable or higher server throughput. If we consider the streaming variant (which optimizes for throughput and rate at the expense of public parameter size and query size), the packed variant achieves substantially higher throughput compared to previous PIR schemes and the other Spiral variants. On the larger databases, SpiralStreamPack achieves 10× higher throughput compared to previous systems (1.5 GB/s) and a 1.7× improvement over the non-packed scheme SpiralStream.

System scaling. Fig. 2 shows how the server's computation time for different PIR schemes scales with the number of records N in the database. When the database consists of relatively small records (10 KB), Spiral achieves similar performance as existing systems when the numbers of records is small, but is up to 2× faster for databases with a million records. When considering databases with larger records (100 KB), Spiral is always 1.8–3× faster for all choices of N we considered. The server computation time of SpiralPack is generally comparable to that of Spiral. Packing is most beneficial when the number of records is large; in these cases SpiralPack achieves up to a 1.5× reduction in server computation time. As we discuss next, packing makes the most difference in the *streaming* setting.

Throughput in the streaming setting. As noted in Section 5.2, we also consider using PIR in a streaming setting, where the same query is *reused* across multiple PIR invocations (on different databases). In this case, query expansion only needs to happen once and its cost can be amortized over the lifetime of the stream. Thus, when considering the streaming setting, we measure the server's processing time *without* the query expansion process. We apply the same methodology to all Spiral variants, SealPIR, OnionPIR, and FastPIR. The effective server throughput for different schemes is shown in Fig. 3 and Table 4. When choosing the parameters for the streaming protocol variants SpiralStream and SpiralStreamPack, we impose a maximum query size of 33 MB to ensure a balanced comparison with the FastPIR protocol [AYA+21] which have queries of the same size. FastPIR is a PIR protocol tailored for the

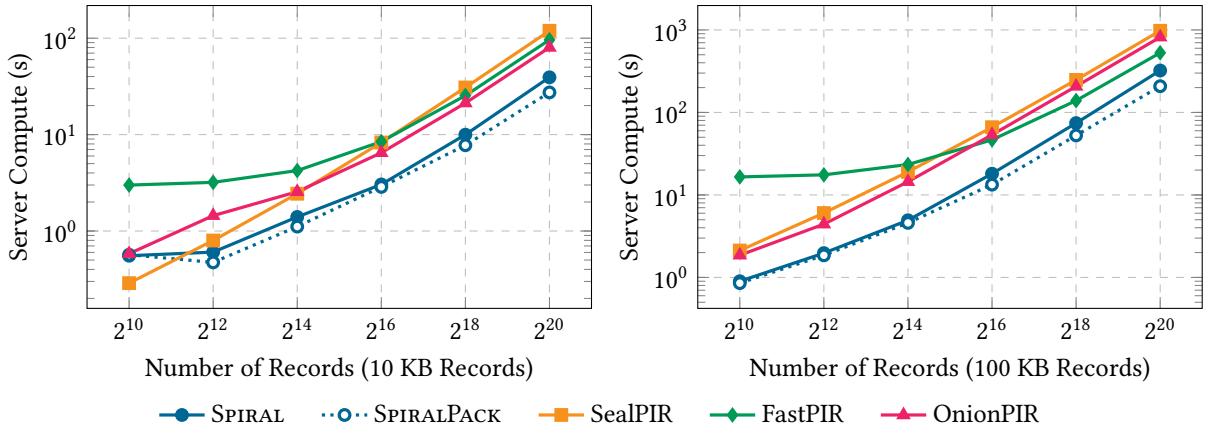

Figure 2: Server computation time as a function of database size for different PIR protocols.

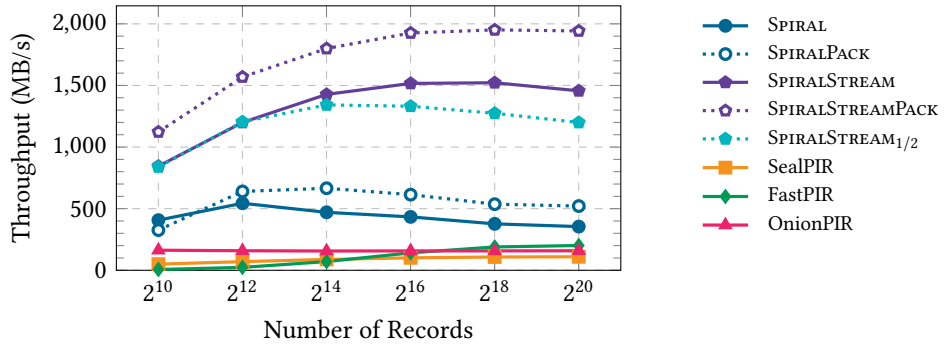

Figure 3: Server throughput in the streaming setting as a function of the number of database records. In the streaming setting, we ignore the query expansion costs (if present) and use the optimal record size for each system. The query sizes for SealPIR, FastPIR, SPIRAL/SPIRALPACK, SPIRALSTREAM/SPIRALSTREAMPACK and SPIRALSTREAM$_{1/2}$, are 65 KB, 33 MB, 14 KB, 33 MB, and 16 MB, respectively. In particular, we choose parameters for SPIRALSTREAM and SPIRALSTREAMPACK so as to match the query size from the FastPIR system (a PIR protocol tailored for the streaming setting).

streaming setting that leverages a large query size to achieve better server throughput. We note that increasing the query size in SPIRALSTREAM and SPIRALSTREAMPACK beyond 33 MB can enable further improvements to the server throughput and the rate, and we explore these trade-offs in more detail in Fig. 4.

For the database configurations we considered, the base version of SPIRAL achieves a 1.7–3.7× higher throughput in the streaming setting compared to previous systems. The packed version SPIRALPACK achieves higher throughput with the same query size, but at the expense of larger public parameters. The streaming-optimized systems SPIRALSTREAM and SPIRALSTREAMPACK achieve significantly higher throughput; on databases with roughly a million records, the server throughput of SPIRALSTREAMPACK is 1.9 GB/s, which is 9.7× higher than FastPIR. The rate is also 5.8× higher than that of FastPIR (i.e., the number of bits the client has to download is 5.8× smaller with SPIRALSTREAMPACK).

From Fig. 3, we observe that for each SPIRAL configuration, the throughput peaks at a certain number of records, and then starts decreasing as the number of records increases. Based on our microbenchmarks (Fig. 5), we observe that the throughput is highest when the server's work is evenly distributed between the first dimension processing and the folding steps (Steps 2 and 3 of the Answer algorithm in Construction 4.1, respectively). For small database sizes, we do not achieve an even split, resulting in lower throughput. Moreover, we cannot increase the size of the first dimension indefinitely, as the noise accumulation scales linearly with the size of the first dimension. In the case of SPIRAL and SPIRALPACK, the limiting factor is that the first dimension can have size at most 2^9 before the

N	Metric	FastPIR	OnionPIR	Spiral	SpiralPack	SpiralStream	SpiralStreamPack
2^{12}	**Param. Size**	**1 MB**	5 MB	31 MB	156 MB	3 MB	125 MB
	Query Size	131 KB	63 KB	**14 KB**	**14 KB**	15 MB	15 MB
	Rate	0.1392	0.2419	0.4348	0.7143	0.4918	**0.8057**
	Throughput*	23 MB/s	159 MB/s	544 MB/s	640 MB/s	1.20 GB/s	**1.57 GB/s**
2^{16}	**Param. Size**	**1 MB**	5 MB	30 MB	31 MB	5 MB	125 MB
	Query Size	2 MB	63 KB	**14 KB**	**14 KB**	30 MB	30 MB
	Rate	0.1392	0.2419	0.4000	0.7013	0.4918	**0.8057**
	Throughput*	142 MB/s	157 MB/s	433 MB/s	614 MB/s	1.52 GB/s	**1.93 GB/s**
2^{20}	**Param. Size**	**1 MB**	5 MB	30 MB	91 MB	5 MB	125 MB
	Query Size	34 MB	63 KB	**14 KB**	**14 KB**	30 MB	30 MB
	Rate	0.1392	0.2419	0.3902	0.6857	0.4918	**0.8057**
	Throughput*	201 MB/s	158 MB/s	355 MB/s	521 MB/s	1.46 GB/s	**1.94 GB/s**

* This throughput measurement does not include query expansion costs, since these are amortized away in the streaming scenario.

Table 4: Performance of FastPIR [AYA+21], OnionPIR [MCR21], and the different Spiral variants in the streaming setting as a function of the number of records N in the database. In the streaming setting, we ignore all query expansion costs (if present) and use the optimal record size for each system.

Database	Best System	Rate	Throughput	Param. Size	Query Size
$2^{20} \times$ **256B**	SpiralStream	**0.0227**	130 MB/s	344 KB	15 MB
$2^{20} \times$ **256B**	SpiralStreamPack	0.0025	**1.03 GB/s**	16 MB	15 MB
$2^{18} \times$ **30KB**	SpiralStream	**0.4883**	326 MB/s	5 MB	3 MB
$2^{18} \times$ **30KB**	SpiralStreamPack	0.1723	**1.85 GB/s**	24 MB	30 MB
$2^{14} \times$ **1MB**	SpiralStreamPack	**0.7750**	1.35 GB/s	88 MB	8 MB
$2^{14} \times$ **1MB**	SpiralStreamPack	0.6532	**1.65 GB/s**	16 MB	30 MB

Table 5: Maximum-rate and maximum-throughput Spiral configurations for different database configurations. For each database configuration, we use the automatic parameter selection algorithm (Section 5.1) to choose the best Spiral variant and parameters that yields the highest rate or the highest server throughput (including query expansion costs). We report the rate, throughput, public parameter size, and query size for the chosen scheme. In all cases, we impose a maximum query size of 33 MB (i.e., the query size in the FastPIR system [AYA+21]). Fig. 4 shows how the maximum rate and throughput scales more generally as a function of the query size (and the public parameter size).

noise from the coefficient expansion process is too high to ensure correctness (without moving to a larger set of lattice parameters). Increasing the number of records requires fixing the size of the first dimension and increasing the number of folding rounds. This also leads to lower throughput.

For SpiralStream and SpiralStreamPack, the limiting factor on how we can split the database between the first dimension and the subsequent dimensions is the query size. The query size scales linearly with the size of the first dimension, and once this maxes out, the throughput starts to decrease. To illustrate this, we consider a version of SpiralStream where the limit on the query size is halved to 16 MB (denoted $\text{SpiralStream}_{1/2}$). As shown in Fig. 3, the throughput peaks at a much smaller database (2^{14} records as opposed to 2^{16} records). To scale to larger databases while maintaining high throughput, we could either increase the query size (to reduce the number of expansion steps needed) or increase the response size (by running the protocol in parallel with a fixed query).

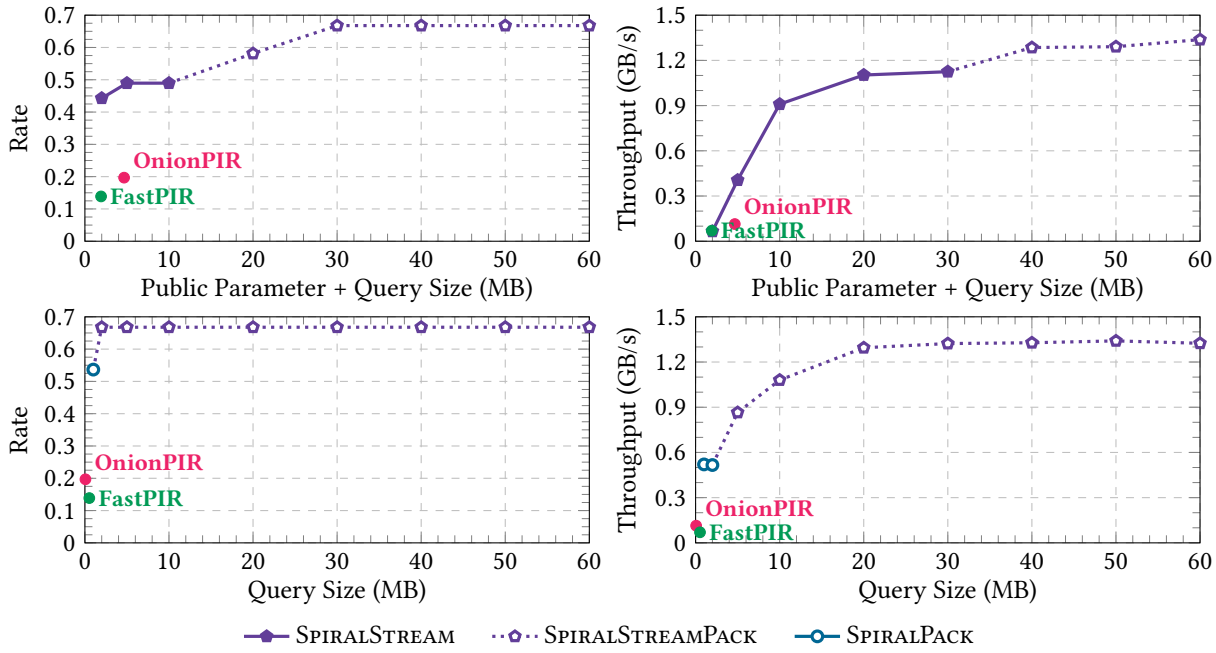

Figure 4: Maximum achievable rate and throughput with a constraint on the public parameter size and/of the query size across all Spiral variants for a $2^{14} \times 100$ KB database. We set the automatic parameter selection algorithm (Section 5.1) to choose the best Spiral variant and parameters that yield the highest rate or the highest server throughput subject to the restriction on the size of the public parameters and/or the size of the query. The performance of OnionPIR and FastPIR are plotted as points, since we cannot vary their query sizes.

Optimizing for rate or throughput. Next, we measure the highest possible throughput and rate achieved by the Spiral family of protocols on different database configurations. Specifically, for each database configuration, we set our parameter selection tool to select parameters that maximize either the rate or the server throughput (recall that our default optimization objective is the estimated server cost). We summarize our results in Table 5.

When considering *small* database records (e.g., 256 bytes), SpiralStream achieves a higher rate compared to the packed version SpiralStreamPack. This is because SpiralStreamPack relies on a large packing dimension T_{pack} to achieve high rate. However, using larger values of T_{pack} translates to a larger minimum record size (i.e., each response decodes to $T_{\mathsf{pack}}^2 \log p$ bits). When the number of bits in the database record is *smaller* than $T_{\mathsf{pack}}^2 \log p$, the extra bits in the response are effective wasted and reduce the rate of the resulting protocol. When the record size is sufficiently large (e.g., 1 MB), the packed version of SpiralStream outperforms the vanilla version. On the flip side, when maximizing throughput, SpiralStreamPack generally outperforms SpiralStream, since the first dimension processing step (Step 2 of the Answer algorithm in Construction 4.1) is applied to encodings of 1-dimensional values rather than 2-dimensional ones. Recall that the computational cost of processing the first dimension scales quadratically with the message dimension n, and this step is a significant portion of the total server computation.

We also note that for the database configurations considered in Table 5, the streaming variants of Spiral and SpiralPack achieve the best rates and throughput. This is because of the extra ciphertext expansion needed in the the vanilla versions of Spiral (and SpiralPack). On the flip side, achieving higher rates and throughput using the streaming variants of Spiral requires communicating larger public parameters and queries. In Fig. 4 we explore the trade-offs between the public parameter size (and query size) and the rate or throughput of the system. Specifically, we report the system performance subject to a bound on either the total query and public parameter size or just the query size. As Fig. 4 shows, it can be advantageous to use the non-streaming variants of Spiral when the goal is to minimize the online query size. However, as we allow the query size (and public parameter size) to grow, the streaming and packed variants of Spiral will allow for higher rates and server throughput. We also observe rapidly

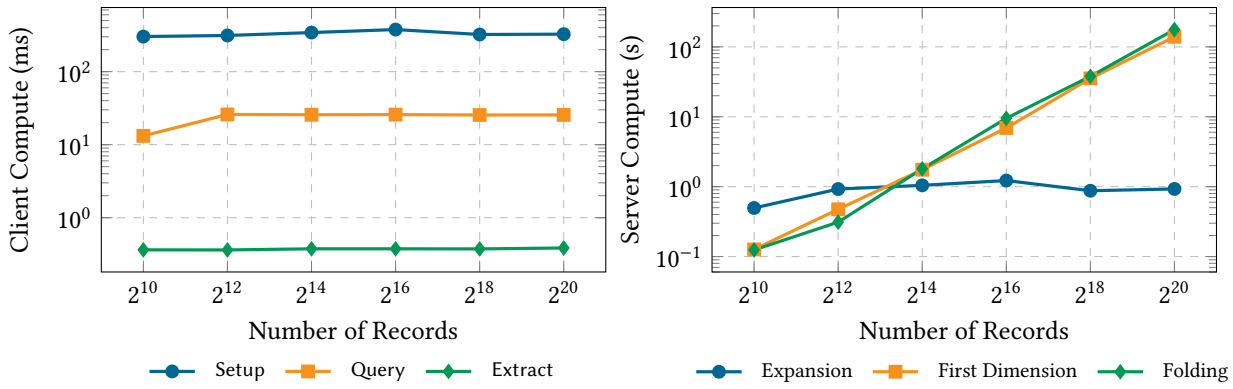

Figure 5: Microbenchmarks for client and server computation in SPIRAL for processing databases with 100 KB records. The client computation consists of the Setup, Query, and Extract algorithms while the server computation consists of the Answer algorithm from Construction 4.1. We separately measure the costs of the query expansion (Step 1), first dimension processing (Step 2), and ciphertext folding (Step 3) in the Answer algorithm.

diminishing returns in the achievable rate and throughput as a function of the parameter sizes.

Microbenchmarks. Finally, we provide a more fine-grained breakdown of the different components of the client's and server's computation in Fig. 5. The client's cost is dominated by the key-generation procedure (which samples the key-switching matrices needed for the query generation algorithm). While this cost is non-trivial ($\approx$ 700 ms), this only needs to be generated once and can be reused for *arbitrarily* many queries. The query-generation completes in under 30 ms, and the response-decoding completes in under 1 ms.

For server computation, the cost of query expansion is mostly fixed, while the cost of processing the initial dimension and the subsequent folding steps (Steps 2 and 3 of the Answer algorithm in Construction 4.1, respectively) both scale linearly with the size of the database. The parameters chosen by our parameter generation algorithm favor those that balance the cost of the initial dimension processing and the cost of the subsequent folding operations.

CRT/SIMD optimizations. As noted in Section 5.2, we choose the 56-bit modulus q to be a product of two 28-bit primes and use the Chinese remainder theorem (CRT) in conjunction with the AVX instruction set to accelerate the integer arithmetic. Choosing a modulus q that splits into 32-bit primes is important for concrete efficiency. We observe that using the AVX instruction sets, we can compute four 32-bit-by-32-bit integer multiplications in the same time it takes to compute a *single* 64-bit-by-64-bit integer multiplication. Thus, using CRT with AVX gives us a factor of 2× speed-up for arithmetic operations. Note that this is helpful primarily when processing the first dimension and less so for the subsequent GSW folding operations. Indeed, if we compare against a modified implementation where we use 64-bit-by-64-bit integer multiplications, we observe a 2.1× slowdown in the time it takes to process the first dimension. As a function of the overall computation time, using CRT provides a 1.3–1.4× speed-up (since the first dimension processing accounts for slightly less than half of the total server computation).

We also note that our implementation uses AVX-512, whereas previous systems only used AVX2. However, AVX-512 is not the main source of speedup in our implementation. If we disable AVX-512, we only observe moderate slowdowns of 6–14%. AVX2 is more critical to our system's performance; for large databases, disabling AVX2 results in a 2× slowdown.

Heuristic noise analysis. As mentioned in Remark 2.18, we set our lattice parameters under an independence heuristic where we model the noise introduced by various homomorphic operations as independent subgaussian distributions (see Section 5.1). While this is a standard heuristic in many previous lattice-based systems (e.g., [GHS12b, CGGI18, MCR21]), we validate the heuristic by comparing the actual error magnitude in the lattice encodings with the magnitude predicted by our heuristic model. We compare the two in Fig. 6. As the plot shows,

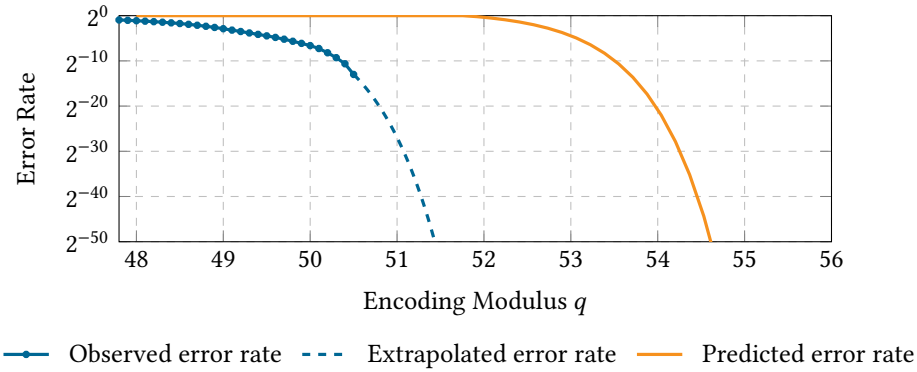

Figure 6: Observed error rates for SPIRAL on a $2^{20} \times 256$B database as a function of the encoding modulus q. The observed error rate is the fraction of error components that exceed the correctness threshold for a particular encoding modulus. These are estimated from a set of 163,000 error coefficients obtained from 20 independent protocol executions. We extend the measured values with a Gaussian function (with width parameter estimated based on the final observed value). Finally, we also plot the error rate predicted by our heuristic noise analysis from Section 5.1.

there is still a decent margin between the measured error magnitude and the predicted error magnitude. This means that while we set our correctness target to 2^{-40}, the actual scheme should satisfy a much higher correctness guarantee.

Application scenarios. We now estimate the concrete cost of using SPIRAL to support various privacy-preserving applications based on PIR:

- **Private video streaming.** Suppose a user is interested in privately streaming a 2 GB movie from a library of 2^{14} movies. Using SPIRALSTREAMPACK, this would require a 30 MB upload, a 2.5 GB download, and 5.6 CPU-hours of computation. The overall server cost using SPIRALSTREAMPACK is \$0.33. This is just 1.9× higher than the *no-privacy* baseline where the client just downloads the movie directly (\$0.18). Using OnionPIR for the same task would require a 63 KB upload, an 8.3 GB download, and 59.3 CPU-hours of compute. This is 17× more expensive than the non-private solution, and 9× more expensive than SPIRALSTREAMPACK.
- **Private voice calls.** Next, we consider the Addra application for private voice communication [AYA+21]. In Addra, a 5-minute voice call can be implemented with 625 rounds, and in each round, the user downloads 96 bytes. If we use SPIRALSTREAM to support a system with up to 2^{20} users, a private 5-minute voice call would require a 29 MB upload, 11 MB of download, and 112 seconds of CPU time. The per-user server cost is \$0.0016, which is a 3.9× improvement compared to FastPIR (used for the Addra system). On an absolute scale, running a system like Addra using SPIRALSTREAM remains costly at over \$300/minute to support a million users.
- **Private Wikipedia.** We can also consider a non-streaming setting where we use PIR to privately access a Wikipedia article. We consider the end-to-end latency needed to retrieve an entry from a 31 GB database (which would contain all of the text in English Wikipedia and a subset of article images) with a maximum article size of 30 KB. We split the database into 16 independent partitions and process the query in parallel on a 16-core machine with 42 GB of memory. Running this setup would require \$229 USD monthly on AWS. We model network conditions based on a median mobile upload speed of 8 Mbps and download speed of 29 Mbps [Spe22]. Under these conditions, SPIRALPACK could deliver an article in just 4.3 seconds. This is a 2.1× reduction in the end-to-end time compared to OnionPIR. Unlike the movie streaming setting above, the non-streaming setting remains one where the private solution remains much slower than non-private retrieval.

6 Related Work

Number-theoretic constructions. Many early constructions of single-server PIR [Cha04, Lip05] follow the Kushilevitz-Ostrovsky paradigm [KO97] based on homomorphic encryption. These were typically instantiated using number-theoretic assumptions such as Paillier [Pai99] or the Damgård-Jurik [DJ01] encryption schemes. Another line of works [CMS99, GR05] gave constructions with polylogarithmic communication from the ϕ-hiding assumption. Döttling et al. [DGI+19] showed how to construct rate-1 PIR (on sufficiently-large) records based on trapdoor hash functions, which can in turn be based on a broad range of classic number-theoretic assumptions.

Lattice-based PIR. The more concretely efficient single-server PIR protocols are based on lattice-based assumptions. Starting with XPIR [MBFK16], a number of systems have progressively reduced the computational cost of single-server PIR [AS16, ACLS18, GH19, PT20, ALP+21, AYA+21, MCR21]. While early constructions only relied on additive homomorphism, more recent constructions also incorporate multiplicative homomorphism for better concrete efficiency [GH19, PT20, ALP+21, MCR21]. The design of Spiral follows the recent approach of composing Regev encryption with GSW encryption to achieve a higher rate and slower noise growth.

PIR variants. Many works have introduced techniques to reduce or amortize the computation cost of single-server PIR protocols. One approach is *batch PIR* [BIM00, IKOS04, GKL10, ACLS18] where the server's computational cost is amortized over a *batch* of queries. In particular, Angel et al. [ACLS18] introduced a *generic* approach of composing a PIR protocol with a probabilistic batch code to amortize the server's computational cost.

Another line of works has focused on *stateful PIR* [PPY18, MCR21, CK20, CHK22] where the client retrieves some query-independent advice string from the database in an offline phase and uses the advice string to reduce the cost of the online phase. The recent OnionPIR system [MCR21] introduces a general approach based on private batch sum retrieval that reduces the online cost of performing PIR over a database with N records to that of a PIR over a database with $O(\sqrt{N})$ records (the overall online cost is still $O(N)$, but the bottleneck is the PIR on the $O(\sqrt{N})$ record database). Corrigan-Gibbs and Kogan [CK20] show how to obtain a single-server stateful PIR with *sublinear* online time; however, the advice string is not reusable so the (linear) offline preprocessing has to be repeated for each query. More recently, Corrigan-Gibbs et al. [CHK22] introduce a stateful PIR protocol with a *reusable* advice string which yields a single-server PIR with sublinear amortized cost.

Another variant is *PIR with preprocessing* [BIM00] or *doubly-efficient PIR* [BIPW17, CHR17] where the server first performs a linear preprocessing step to obtain an encoding of the database. Using the encoding, the server can then answer online queries in strictly sublinear time. Boyle et al. [BIPW17] and Canetti et al. [CHR17] recently showed how to construct *doubly-efficient PIR* schemes from virtual black-box obfuscation, a very strong cryptographic assumption that is possible only in idealized models [BGI+01] (and also currently far from being concretely efficient).

Multi-server PIR. While our focus in this work in the single-server setting, many PIR protocols [CGKS95, Yek07, Efr09, BIKO12, GI14, BGI16, HH19] consider the multi-server setting where the database is replicated across several *non-colluding* servers (see also the survey by Gasarch [Gas04] and the references therein). Multi-server constructions are highly efficient as the server computation can be based purely on symmetric operations rather than more expensive public-key operations. However, the non-colluding requirements imposes logistic hurdles to deployment.

Acknowledgments

We thank Henry Corrigan-Gibbs and Craig Gentry for helpful insights and pointers on this work. D. J. Wu is supported by NSF CNS-1917414, CNS-2045180, and a Microsoft Research Faculty Fellowship.

References

[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.

[ACPS09] Benny Applebaum, David Cash, Chris Peikert, and Amit Sahai. Fast cryptographic primitives and circular-secure encryption based on hard learning problems. In *CRYPTO*, 2009.

[AG11] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In *ICALP*, 2011.

[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*, 2021.

[APS15] Martin R Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3), 2015.

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

[AWS21] Amazon EC2 reserved instances pricing. https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/, 2021. Last accessed: November 28, 2021.

[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.

[BCD+16] Joppe W. Bos, Craig Costello, Léo Ducas, Ilya Mironov, Michael Naehrig, Valeria Nikolaenko, Ananth Raghunathan, and Douglas Stebila. Frodo: Take off the ring! practical, quantum-secure key exchange from LWE. In *ACM CCS*, 2016.

[BDG15] Nikita Borisov, George Danezis, and Ian Goldberg. DP5: A private presence service. *Proc. Priv. Enhancing Technol.*, 2015(2), 2015.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New directions in nearest neighbor searching with applications to lattice sieving. In *SODA*, 2016.

[BGI+01] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, 2001.

[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.

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

[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.

[BKS+21] Fabian Boemer, Sejun Kim, Gelila Seifu, Fillipe DM de Souza, Vinodh Gopal, et al. Intel HEXL (release 1.2). https://github.com/intel/hexl, 2021.

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

[BRS02] John Black, Phillip Rogaway, and Thomas Shrimpton. Encryption-scheme security in the presence of key-dependent messages. In *SAC*, 2002.

[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.

[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.

[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.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *EUROCRYPT*, 2022.

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

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

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

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0: Better lattice security estimates. In *ASIACRYPT*, 2011.

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

[DJ01] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC*, 2001.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4), 2018.

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

[FKP15] Eric Fung, Georgios Kellaris, and Dimitris Papadias. Combining differential privacy and PIR for efficient strong location privacy. In *SSTD*, 2015.

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

[Gal13] Steven D Galbraith. Space-efficient variants of cryptosystems based on learning with errors. 2013.

[Gas04] William I. Gasarch. A survey on private information retrieval. *Bull. EATCS*, 82, 2004.

[GCM+16] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath T. V. Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *NSDI*, 2016.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, 2009.

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

[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.

[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.

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

[ISW21] Yuval Ishai, Hang Su, and David J. Wu. Shorter and faster post-quantum designated-verifier zkSNARKs from lattices. In *ACM CCS*, 2021.

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *USENIX Security*, 2021.

[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.

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC*, 2005.

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

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

[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*, 2011.

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

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[OS07] Rafail Ostrovsky and William E. Skeith III. A survey of single database PIR: techniques and applications. *IACR Cryptol. ePrint Arch.*, 2007.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, 1999.

[Pei16] Chris Peikert. A decade of lattice cryptography. *Found. Trends Theor. Comput. Sci.*, 10(4), 2016.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *ACM CCS*, 2018.

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

[PVW08] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *CRYPTO*, 2008.

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

[SC07] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *NDSS*, 2007.

[SEA19] Microsoft SEAL (release 3.2). https://github.com/Microsoft/SEAL, February 2019. Microsoft Research, Redmond, WA.

[SFK+21] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald G. Dreslinski, Christopher Peikert, and Daniel Sánchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO*, pages 238–252, 2021.

[Spe22] Speedtest. Speedtest global index, 2022. https://www.speedtest.net/global-index. Last accessed: March 18, 2022.

[TSS+20] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2), 2020.

[VJH21] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully homomorphic encryption compilers. In *IEEE S&P*, 2021.

[WZPM16] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-preserving shortest path computation. In *NDSS*, 2016.

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

A Coefficient Extraction on Regev Encodings

In this section, we recall the coefficient expansion algorithm by Angel et al. [ACLS18] and extended by Chen et al. [CCR19]. This approach relies on the ability to homomorphically compute automorphisms on Regev-encoded polynomials. We review this below.

Automorphisms. As usual, let $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. For a positive integer ℓ, we write $\tau_\ell : R \to R$ to denote the ring automorphism $r(x) \mapsto r(x^\ell)$. We can define a corresponding set of automorphisms over R_q. For notational convenience, we use τ_ℓ to denote both sets of automorphisms. We extend τ_ℓ to operate on vectors and matrices of ring elements (in both R and R_q) in a component-wise manner.

Automorphisms on Regev encodings. Similar to the other translation protocols (Sections 3.1 and 3.2), supporting automorphisms requires knowledge of additional key-switching matrices. We give the parameter-generation and automorphism algorithms below:

- $\mathsf{AutomorphSetup}(\mathbf{s}, \tau, z)$: On input the secret key $\mathbf{s} = [-\tilde{s} \mid 1]^\mathsf{T}$, an automorphism $\tau : R_q \to R_q$, and a decomposition base $z \in \mathbb{N}$, let $t = \lfloor \log_z q \rfloor + 1$. Sample $\mathbf{a} \xleftarrow{R} R_q^t$, $\mathbf{e} \leftarrow \chi^t$, and output the key

$$\mathbf{W}_\tau = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{s}\mathbf{a}^\mathsf{T} + \mathbf{e}^\mathsf{T} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times t} \\ -\tau(\tilde{s}) \cdot \mathbf{g}_z \end{bmatrix} \in R_q^{2\times t}$$

- $\mathsf{Automorph}(\mathbf{W}_\tau, \mathbf{c})$: On input the automorphism key $\mathbf{W}_\tau \in R_q^{2\times t}$ associated with an automorphism $\tau : R_q \to R_q$, and encoding $\mathbf{c} = (c_0, c_1) \in R_q^2$, output $\mathbf{W}_\tau \mathbf{g}_z^{-1}(\tau(c_0)) + [\ 0 \mid \tau(c_1)\ ]^\mathsf{T}$.

Theorem A.1 (Automorphisms [GHS12a, BGV12, adapted]). *For a positive integer $\ell \in \mathbb{N}$, let $\tau_\ell : R_q \to R_q$ be the automorphism $r(x) \mapsto r(x^\ell)$ and $z \in \mathbb{N}$ be a decomposition base. Let $\mathbf{s} = [-\tilde{s} \mid 1]^\mathsf{T} \in R_q^2$ be a Regev secret key and suppose that $\mathbf{c} \in R_q^2$ encodes $\mu \in R_q$ with error $e \in R$. Let $\mathbf{W}_\tau \leftarrow \mathsf{AutomorphSetup}(\mathbf{s}, \tau_\ell, z)$ and $\mathbf{c}' \leftarrow \mathsf{Automorph}(\mathbf{W}_\tau, \mathbf{c})$. Suppose that the error distribution χ in $\mathsf{AutomorphSetup}$ is B-bounded, and let $t = \lfloor \log_z q \rfloor + 1$. Then, $\mathbf{c}'$ is an encoding of $\tau_\ell(\mu) \in R_q$ with error e' where $\|e'\|_\infty \le \|e\|_\infty + tdBz/2$. If e is subgaussian with parameter σ, χ is subgaussian with parameter σ_χ, then under the independence heuristic (Remark 2.18), e' is subgaussian with parameter σ' where $(\sigma')^2 = \sigma^2 + tdz^2\sigma_\chi^2/4$.*

Proof. Let $\mathbf{c} = [c_0 \mid c_1]^\mathsf{T}$. By definition, we have that

$$\begin{aligned}
\mathbf{s}^\mathsf{T}\mathbf{c}' = \mathbf{s}^\mathsf{T}\mathbf{W}_\tau \mathbf{g}_z^{-1}(\tau_\ell(c_0)) + \tau_\ell(c_1) &= [-\tilde{s} \mid 1] \left(\begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{s}\mathbf{a}^\mathsf{T} + \mathbf{e}^\mathsf{T} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1\times t} \\ -\tau_\ell(\tilde{s}) \cdot \mathbf{g}_z \end{bmatrix} \right) \mathbf{g}_z^{-1}(\tau_\ell(c_0)) + \tau_\ell(c_1) \\
&= \mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0)) - \tau_\ell(\tilde{s})\tau_\ell(c_0) + \tau_\ell(c_1) \\
&= \mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0)) + \tau_\ell(\mathbf{s}^\mathsf{T}\mathbf{c}) \\
&= \tau_\ell(\mu) + \tau_\ell(e) + \mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0)).
\end{aligned}$$

Thus, $\mathbf{c}'$ is an encoding of $\tau_\ell(\mu)$ with error $\tau_\ell(e) + \mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0))$. Over $R = \mathbb{Z}[x]/(x^d + 1)$, the automorphism τ_ℓ simply permutes the coefficients of the input, so $\|\tau_\ell(e)\|_\infty = \|e\|_\infty$. Letting $e' = \tau_\ell(e) + \mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0))$, we have that

$$\|e'\|_\infty \le \|\tau_\ell(e)\|_\infty + \left\|\mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0))\right\|_\infty \le \|e\|_\infty + t\gamma_R Bz/2 = \|e\|_\infty + tdBz/2.$$

Next, since $\mathbf{e}$ in $\mathbf{W}_\tau$ is sampled independently of $\mathbf{c}$, by Lemma 2.6, $\mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0))$ is subgaussian with parameter $\sqrt{td}\sigma_\chi z/2$. If we apply the independence heuristic to e and $\mathbf{e}^\mathsf{T}\mathbf{g}_z^{-1}(\tau_\ell(c_0))$, then e' is subgaussian with parameter σ' where $(\sigma')^2 = \sigma^2 + tdz^2\sigma_\chi^2/4$. □

Coefficient expansion algorithm. We recall the coefficient expansion procedure by Angel et al. [ACLS18] and extended by Chen et al. [CCR19]. The algorithm takes a polynomial $f = \sum_{i\in[0,2^r-1]} f_i x^i \in R_q$ as input and outputs a (scaled) vector of coefficients $2^r \cdot (f_0, \ldots, f_{2^r-1}) \in \mathbb{Z}_q^{2^r}$. The algorithm only relies on ring automorphisms $\tau_\ell : R_q \to R_q$ and linear operations, and can be implemented *homomorphically* on encodings.

Theorem A.2 (Correctness of Coefficient Expansion [ACLS18, CCR19]). *Let $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. Let q be an odd integer. Then, on input a polynomial $f = \sum_{i\in[0,2^r-1]} f_i x^i \in R_q$ where $2^r \le d$, Algorithm 1 outputs the scaled coefficients $2^r \cdot (f_0, \ldots, f_{2^r-1}) \in \mathbb{Z}_q^{2^r}$. More generally, for any input polynomial $\hat{f} = \sum_{i\in[0,d-1]} \hat{f}_i x^i$ to Algorithm 1, after round $i \in [r]$, the value f_j satisfies $f_j = \sum_{k:k=j-1 \bmod 2^i} \hat{f}_k x^{k-j+1}$.*

Remark A.3 (Homomorphic Expansion). By construction, Algorithm 1 only requires scalar multiplication, addition, and automorphisms over R_q. Thus, we can homomorphically evaluate Algorithm 1 on a Regev encoding of a

Algorithm 1: Coefficient expansion [ACLS18, CCR19].

Input: a polynomial $f = \sum_{i \in [0,2^r-1]} f_i x^i \in R_q$ where $R = \mathbb{Z}[x]/(x^d + 1)$ and $2^r \leq d$

Output: the scaled coefficients $2^r \cdot (f_0, \ldots, f_{2^r-1}) \in \mathbb{Z}_q^{2^r}$

$f_0 \leftarrow f$
for $i = 0$ **to** $r - 1$ **do**
  $\ell \leftarrow 2^{r-i} + 1$
  for $j = 0$ **to** $2^i - 1$ **do**
    $f'_j \leftarrow f_j \cdot x^{-2^j}$ ▷ $x^{-2^j} = -x^{d-2^j} \in R$
    $f_j \leftarrow f_j + \tau_\ell(f_j)$ ▷ $\tau_\ell \colon R_q \to R_q$ is the automorphism $f(x) \mapsto f(x^\ell)$
    $f_{j+2^i} \leftarrow f'_j + \tau_\ell(f'_j)$
  end
end
return $f_0, f_1, \ldots, f_{2^r-1}$

polynomial $f \in R_q$ to obtain (scaled) Regev encodings of the coefficients of f. To homomorphically compute r rounds of Algorithm 1 on an encoding $\mathbf{c}$, the evaluator will need access to key-switching matrices $\mathbf{W}_0, \ldots, \mathbf{W}_{r-1}$ where $\mathbf{W}_i \leftarrow \mathsf{AutomorphSetup}(\mathbf{s}, \tau_{2^{r-i}+1}, z)$, $\mathbf{s}$ is the secret key associated with $\mathbf{c}$, and $z \in \mathbb{N}$ is the desired decomposition base (chosen to control noise growth).

Theorem A.4 (Homomorphic Coefficient Expansion)**.** *Let $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two and let $\mathbf{s} \in R_q^2$ be a Regev secret key. Take any polynomial $f = \sum_{i \in [0,2^r-1]} f_i x^i \in R_q$ for some $r \in \mathbb{N}$ where $2^r \leq d$. Suppose that $\mathbf{c}$ is a Regev encoding of f with error e. Suppose automorphism keys are sampled using* AutomorphSetup *with decomposition base $z \in \mathbb{N}$ and a B-bounded error distribution χ. Then, homomorphically applying Algorithm 1 to $\mathbf{c}$ yields encodings $(\mathbf{c}'_0, \ldots, \mathbf{c}'_{2^r-1})$ where $\mathbf{c}'_i$ encodes f_i with error e'_i and $\|e'_i\|_\infty \leq 2^r(\|e\|_\infty + tdBz)$ for $t = \lfloor \log_z q \rfloor + 1$. In addition, if e is subgaussian with parameter σ and χ is subgaussian with parameter σ_χ, then under the independence heuristic (Remark 2.18), each e'_i is subgaussian with parameter σ'_i where $(\sigma'_i)^2 = 4^r(\sigma^2 + tdz^2\sigma_\chi^2/3)$.*

Proof. Correctness of the homomorphic operations (Theorems 2.12 and A.1) along with correctness of the coefficient expansion algorithm (Theorem A.2) implies that $\mathbf{c}'_i$ is a valid encoding of f_i.

To bound the error in the output encodings, consider the error accumulation in each round $i = 0, \ldots, r-1$ of Algorithm 1. Let $\hat{\mathbf{c}}$ be an encoding at the beginning of the i^{th} round and let $\hat{e}$ be its associated error (e.g., $\hat{e} = e$ when $i = 0$). By Theorem 2.12, multiplication by the monomial x^{-2^j} does not change the norm of the error. By Theorems 2.12 and A.1, the error in the encoding after the homomorphic addition and automorphism is bounded by $\|\hat{e}\|_\infty + (\|\hat{e}\|_\infty + tdBz/2) = 2\|\hat{e}\|_\infty + tdBz/2$. Thus, after r iterations, the final error e'_i in each encoding satisfies

$$\|e'_i\|_\infty \leq 2^r \|e\|_\infty + (2^r - 1)tdBz/2 \leq 2^r \|e\|_\infty + 2^r tdBz.$$

For the subgaussian case, suppose that the errors in the encodings at the beginning of the i^{th} round are subgaussian with parameter σ_i. By the same analysis as in the proof of Theorem A.1, the error associated with encodings at the end of the i^{th} round will be subgaussian with parameter σ_{i+1} where $\sigma_{i+1}^2 = 4\sigma_i^2 + tdz^2\sigma_\chi^2/4$. After r iterations,

$$\sigma_r^2 = 4^r\sigma^2 + tdz^2\sigma_\chi^2/4 \sum_{i \in [0,r]} 4^i < 4^r\sigma^2 + 4^r tdz^2\sigma_\chi^2/3. \qquad \square$$

Remark A.5 (Multiple Decomposition Bases)**.** When homomorphically evaluating Algorithm 1, we only require the ability to homomorphically evaluate automorphisms on the encrypted polynomial. For more fine-grained control of the noise introduced by the expansion procedure, we can use different decomposition bases in different rounds of the expansion. This will be useful in our protocol (Construction 4.1) since we pack two *different* polynomials into the even and odd powers of f; the two polynomials have very different degrees, so it is advantageous to use different decomposition bases to expand them.

B Correctness and Security Analysis

In this section, we provide the formal proofs of Theorem 4.5 and Theorem 4.6 for Construction 4.1.

Proof of Theorem 4.5 (Correctness). Let $\mathcal{D} = \{d_1, \ldots, d_N\}$ be the database where $d_i \in R_p^{n \times n}$ and take any index $\mathsf{idx} = (i^*, j_1^*, \ldots, j_{\nu_2}^*)$. Sample $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^N)$ and let $\mathsf{q} \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx})$, $\mathsf{r} \leftarrow \mathsf{Answer}(\mathsf{pp}, \mathcal{D}, \mathsf{q})$. In particular, $\mathsf{qk} = (\mathbf{s}, \mathbf{S})$, $\mathsf{q} = \mathbf{c} \in R_q^2$, and $\mathsf{r} = (\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2) \in R_{q_2}^n \times R_{q_1}^{n \times n}$. We first argue that $\mathbf{C}$ is a Regev encoding of d_{idx} and then proceed with the noise analysis. We consider each step of the Answer algorithm:

- **Query expansion:** By construction, $\mathbf{c}$ is an encoding of the packed polynomial $\mu(x)$ from Eq. (4.1). By Theorems A.2 and A.4, after a single iteration, $\mathbf{c}_{\mathsf{Reg}}$ encodes the message $2^{r_1 - 1}\mu_{i^*}(x^2)$, and $\mathbf{c}_{\mathsf{GSW}}$ encodes the message $2^{r_2 - 1}\mu_{j^*}(x^2)$.
  - Since $\mu_{i^*} = \lfloor q/p \rfloor x^{i^*}$, after $r_1 - 1$ rounds of expansion on $\mathbf{c}_{\mathsf{Reg}}$, for all $i \neq i^*$, $\mathbf{c}_i^{(\mathsf{Reg})}$ is an encoding of 0 while $\mathbf{c}_{i^*}^{(\mathsf{Reg})}$ is an encoding of $\lfloor q/p \rfloor$. By Theorem 3.1, $\mathbf{C}_i^{(\mathsf{Reg})}$ are encodings of $\mathbf{0}^{n \times n}$ for all $i \neq i^*$ while $\mathbf{C}_{i^*}^{(\mathsf{Reg})}$ is an encoding of $\lfloor q/p \rfloor \mathbf{I}_n$.
  - By construction of μ_{j^*}, after $r_2 - 1$ rounds of expansion on $\mathbf{c}_{\mathsf{GSW}}$, the encodings $\mathbf{c}_{(\ell-1)t_{\mathsf{GSW}}+1}^{(\mathsf{GSW})}, \ldots, \mathbf{c}_{\ell t_{\mathsf{GSW}}}^{(\mathsf{GSW})}$ encode messages $j_\ell^*, z_{\mathsf{GSW}} j_\ell^*, \ldots, (z_{\mathsf{GSW}})^{t_{\mathsf{GSW}}-1} j_\ell^*$ for each $\ell \in [\nu_2]$. Then, by Theorem 3.2, $\mathbf{C}_\ell^{(\mathsf{GSW})}$ is a GSW encoding of the bit j_ℓ^*.
- **Processing the first dimension:** By Theorem 2.12 and the fact that $\mathbf{C}_i^{(\mathsf{Reg})}$ are encodings of $\mathbf{0}^{n \times n}$ for all $i \neq i^*$, it follows that $\mathbf{C}_j^{(0)}$ is an encoding of $\lfloor q/p \rfloor \cdot d_{i^*,j}$ for each $j \in [0, 2^{\nu_2} - 1]$.
- **Folding in the subsequent dimensions:** We show that for all $r \in [0, \nu_2]$ and $j \in [0, 2^{\nu_2 - r} - 1]$, $\mathbf{C}_j^{(r)}$ encodes $\lfloor q/p \rfloor d_{i^*, \rho_r + j}$, where $\rho_r = \sum_{i \in [r]} 2^{\nu_2 - i} j_i^*$. As shown above, this is true for $r = 0$. Consider the ciphertexts output at the end of the r^{th} round. First, by definition, $\rho_r = \rho_{r-1} + 2^{\nu_2 - r} j_r^*$. Now, by correctness of the homomorphic operations (Theorems 2.12 and 2.19 and Remark 2.16), $\mathbf{C}_j^{(r)}$ is an encoding of

$$\lfloor q/p \rfloor \left((1 - j_r^*) d_{i^*, \rho_{r-1} + j} + j_r^* d_{i^*, \rho_{r-1} + 2^{\nu_2 - r} + j} \right) = \lfloor q/p \rfloor \cdot d_{i^*, \rho_r + j},$$

  as required. Finally, $\rho_{\nu_2} = \sum_{i \in [\nu_2]} 2^{\nu_2 - i} j_i^*$. Thus, the final encoding $\mathbf{C}_0^{(\nu_2)}$ has value

$$\lfloor q/p \rfloor d_{i^*, \rho_{\nu_2}} = \lfloor q/p \rfloor d_{i^*, j_1^*, \ldots, j_{\nu_2}^*} = \lfloor q/p \rfloor d_{\mathsf{idx}}.$$

As long as the noise in $\mathbf{C}_0^{(\nu_2)}$ is small enough (as required by Theorems 2.11 and 3.4), Decode will output d_{idx}, and correctness holds. We now analyze the noise in $\mathbf{C}_0^{(\nu_2)}$. We can assume this maximum is taken over all decomposition bases used for coefficient expansion (see Remark A.5).

- **Query:** The client's query $\mathbf{c} \in R_q^2$ is a fresh Regev encryption with error at most $\mathbf{e} \leftarrow \chi$. Since χ is B-bounded, $\|\mathbf{e}\|_\infty \leq B$. Similarly, if χ is subgaussian with parameter σ, the same holds for $\mathbf{e}$.
- **Query expansion:** We consider the Regev and GSW ciphertexts separately:
  - By Theorem A.4, the noise $\mathbf{e}_i^{(\mathsf{Reg})}$ associated with each $\mathbf{c}_i^{(\mathsf{Reg})}$ is bounded by $2^{\nu_1 + 1} B + 2^{\nu_1 + 1} t_{\mathsf{coeff}} d B z_{\mathsf{coeff}}$, where $t_{\mathsf{coeff}} = \lfloor \log_{z_{\mathsf{coeff}}} q \rfloor + 1 = O(\log q)$. Thus, $\|\mathbf{e}_i^{(\mathsf{Reg})}\|_\infty = O(2^{\nu_1} d B z \log q)$. By Theorem 3.1, the noise $\mathbf{E}_i^{(\mathsf{Reg})}$ in each $\mathbf{C}_i^{(\mathsf{Reg})}$ is bounded by $\|\mathbf{e}_i^{(\mathsf{Reg})}\|_\infty + (d t_{\mathsf{conv}} B z_{\mathsf{conv}})/2 = O(2^{\nu_1} d B z \log q)$, since $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1 = O(\log q)$.
  - Similarly, by Theorem A.4, and using the fact that $t_{\mathsf{GSW}} = \lfloor \log_{z_{\mathsf{GSW}}} q \rfloor + 1 = O(\log q)$, the noise $\mathbf{e}_i^{(\mathsf{GSW})}$ associated with each $\mathbf{c}_i^{(\mathsf{GSW})}$ is bounded by $O(\nu_2 B d z \log^2 q)$. By Theorem 3.2, the noise $\mathbf{E}_i^{(\mathsf{GSW})}$ associated with $\mathbf{C}_i^{(\mathsf{GSW})}$ is bounded by $O(\nu_2 B^2 d^2 z \log^2 q) + O(d B z \log q) = O(\nu_2 B^2 d^2 z \log^2 q)$.

- **Processing the first dimension:** By Theorem 2.12, the noise $\mathbf{E}_j^{(0)}$ in each ciphertext $\mathbf{C}_j^{(0)}$ is bounded by $2^{\nu_1} dnp/2 \cdot O(2^{\nu_1} dBz \log q) = O(2^{2\nu_1} d^2 npBz \log q)$.
- **Folding in the subsequent dimensions:** To bound the noise introduced in the folding step, we first observe that $\mathbf{C}_r^{(\mathsf{GSW})}$ is encrypting a bit $b \in \{0,1\}$, and moreover, either $\mathbf{C}_r^{(\mathsf{GSW})}$ or $\mathsf{Complement}(\mathbf{C}_r^{(\mathsf{GSW})})$ is a GSW encryption of 0. Appealing now to Theorem 2.19, the noise $\mathbf{E}_j^{(r)}$ in each ciphertext $\mathbf{C}_j^{(r)}$ is bounded by

$$\max\left(\|\mathbf{E}_j^{(r-1)}\|_\infty, \|\mathbf{E}_{2^{\nu_2-r}+j}^{(r-1)}\|_\infty\right) + O\left(dnz \|\mathbf{E}_r^{(\mathsf{GSW})}\|_\infty \log q\right).$$

  The final noise $\mathbf{E}_0^{(\nu_2)}$ is then bounded by $O(2^{2\nu_1} d^2 npBz \log q + \nu_2^2 B^2 d^3 nz^2 \log^3 q)$.

Recall that $\mathsf{r} = (\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$. By Theorem 3.4, the noise $\mathbf{E}$ in the final encoding $\mathbf{Z} \leftarrow \mathsf{Recover}(\mathbf{S}, \hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$ is bounded by

$$\|\mathbf{E}\|_\infty = O\left(\frac{q_1}{q}\left(2^{2\nu_1} d^2 npBz \log q + \nu_2^2 B^2 d^3 nz^2 \log^3 q + p\right) + \frac{q_1}{q_2} dB + p\right).$$

By Theorem 2.11, $\mathsf{Decode}(\mathbf{Z}) = d_{\mathsf{idx}}$ as long as $\|\mathbf{E}\|_\infty + (q_1 \bmod p) \le q_1/2p$. This condition is satisfied by taking

$$q = \Omega\left(d^2 npBz \log q (2^{2\nu_1} p + \nu_2^2 Bdz \log^2 q)\right) \quad \text{and} \quad q_2 = \Omega(dBp) \quad \text{and} \quad q_1 = \Omega(p^2),$$

and the theorem follows. □

Proof of Theorem 4.6 (Security). We proceed with a hybrid argument:

- Hyb_0: This is the real query privacy game. The challenger starts by computing $(\mathsf{pp}, \mathsf{qk}) \leftarrow \mathsf{Setup}(1^\lambda, 1^N)$ and sends pp to the adversary. The challenger samples a random bit $b \xleftarrow{R} \{0,1\}$ and replies to the adversary's queries $(\mathsf{idx}_0, \mathsf{idx}_1)$ with $\mathsf{q} \leftarrow \mathsf{Query}(\mathsf{qk}, \mathsf{idx}_b)$. In particular, the challenger samples $\mathbf{S} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^n)$, $\mathbf{s} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^1)$, computes the conversion keys $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi}) \leftarrow \mathsf{RegevToGSWSetup}(\mathbf{s}, \mathbf{S}, z_{\mathsf{conv}})$ and $\mathbf{W}_i \leftarrow \mathsf{AutomorphSetup}(\mathbf{s}, \tau_{2^{\rho-i}+1}, z_{\mathsf{coeff}})$ for all $i \in [0, \rho-1]$, and sets $\mathsf{pp} = (\mathsf{ck}, \mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1})$. In addition, the query consists of an encoding $\mathsf{Regev.Encode}(\mathbf{s}, \mu)$, where $\mu \in R_q$ is as defined in Eq. (4.1). At the end of the experiment, the adversary outputs a bit $b' \in \{0,1\}$. The experiment outputs 1 if $b = b'$ and 0 otherwise.
- Hyb_1: Same as Hyb_0 except the challenger replaces the encoding $\mathbf{V}$ in ck with an encoding of a random matrix $\mathbf{R}_1 \xleftarrow{R} R_q^{n \times m_{\mathsf{conv}}}$ under $\mathbf{S}$, where $m_{\mathsf{conv}} = (n+1) \cdot t_{\mathsf{conv}}$ and $t_{\mathsf{conv}} = \lfloor \log_{z_{\mathsf{conv}}} q \rfloor + 1$. It also replaces $\mathbf{W}$ in ck with an encoding of a random matrix $\mathbf{R}_2 \xleftarrow{R} R_q^{n \times 2t_{\mathsf{conv}}}$ under $\mathbf{S}$.
- Hyb_2: Same as Hyb_1 except the challenger replaces $\mathbf{W}_i$ with an encoding of a random element $\mathbf{r}_i \xleftarrow{R} R_q^{t_{\mathsf{coeff}}}$ under $\mathbf{s}$ for all $i \in [0, \rho-1]$ and where $t_{\mathsf{coeff}} = \lfloor \log_{z_{\mathsf{coeff}}} q \rfloor + 1$. It also replies to each query with an encoding of a random $r \xleftarrow{R} R_q$ under $\mathbf{s}$.

For an adversary $\mathcal{A}$, we write $\mathsf{Hyb}_i(\mathcal{A})$ to denote the output of experiment Hyb_i with adversary $\mathcal{A}$. By design, in Hyb_2, the adversary's view is entirely independent of b, so for all adversaries $\mathcal{A}$, $\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] = 1/2$. To complete the proof, we show that the outputs of each adjacent pair of experiments are computationally indistinguishable.

Lemma B.1. *Suppose the Regev encoding scheme with message space R_q^n is $\mathcal{F}_{\mathsf{aff}}$-KDM-secure. Then, for all efficient adversaries $\mathcal{A}$, $\mathsf{Hyb}_0(\mathcal{A}) \overset{c}{\approx} \mathsf{Hyb}_1(\mathcal{A})$,*

Proof. Suppose there is an efficient adversary $\mathcal{A}$ such that $\left|\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]\right| \ge \varepsilon$ for some non-negligible ε. We use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ that breaks KDM security of the encoding scheme:

1. The KDM security challenger starts by sampling a secret key $\mathbf{S} = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\mathsf{T} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^n)$.
2. Algorithm $\mathcal{B}$ samples for itself a key $\mathbf{s} = [-\tilde{s}_0 \mid 1]^\mathsf{T} \leftarrow \mathsf{KeyGen}(1^\lambda, 1^1)$.

3. Algorithm $\mathcal{B}$ queries the encryption oracle on input $-\tilde{s}_0 \cdot \mathbf{G}_{n,z_{\mathsf{conv}}}$ to obtain $\mathbf{V}$ and on input $-\tilde{\mathbf{s}}_1 \cdot (\mathbf{s}^\intercal \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\intercal)$ to obtain $\mathbf{W}$.[13] Note that $-\tilde{s}_0 \cdot \mathbf{G}_{n,z_{\mathsf{conv}}}$ is independent of the secret key $\mathbf{S}$ and $-\tilde{\mathbf{s}}_1 \cdot (\mathbf{s}^\intercal \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\intercal)$ is a linear function of the components of the secret key $\mathbf{S}$ (specifically, algorithm $\mathcal{B}$ can compute $\mathbf{s}^\intercal \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\intercal$ itself). Algorithm $\mathcal{B}$ computes $\mathbf{\Pi}$ exactly as in the real scheme and sets $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$.

4. Algorithm $\mathcal{B}$ computes $\mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1}$ as in the real scheme (since it only depends on $\mathbf{s}$) and gives $\mathsf{pp} = (\mathsf{ck}, \mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1})$ to $\mathcal{A}$.

5. Algorithm $\mathcal{B}$ samples a bit $\beta \xleftarrow{R} \{0,1\}$. When $\mathcal{A}$ makes a query $(\mathsf{idx}_0, \mathsf{idx}_1)$, algorithm $\mathcal{B}$ responds with $\mathsf{Query}(\mathsf{qk}, \mathsf{idx}_\beta)$. In particular, the Query algorithm depends only on $\mathbf{s}$ (which $\mathcal{B}$ knows) and *not* on $\mathbf{S}$.

6. Finally, algorithm $\mathcal{A}$ outputs a bit $\beta' \in \{0,1\}$ and algorithm $\mathcal{B}$ outputs 1 if $\beta = \beta'$ and 0 otherwise.

Let $b \in \{0,1\}$ be the bit the KDM challenger samples.

- If $b = 0$, then the challenger replies to $\mathcal{B}$'s queries with the encryption of the queried message. In this case, $\mathcal{B}$ perfectly simulates Hyb_0 and outputs 1 with $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]$. Correspondingly, $\mathcal{B}$ outputs $b = 0$ with probability $1 - \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]$.
- If $b = 1$, then the challenger replies to $\mathcal{B}$'s queries with an encryption of a random message. In this case, $\mathcal{B}$ perfectly simulates Hyb_1 and outputs 1 with probability $\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]$.

Thus, algorithm $\mathcal{B}$ outputs b with probability

$$\left|\Pr[\mathcal{B}(1^\lambda) = b] - \frac{1}{2}\right| = \left|\frac{1}{2}(1 - \Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1]) + \frac{1}{2}\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \frac{1}{2}\right| = \frac{\varepsilon}{2}. \qquad \square$$

Lemma B.2. *Suppose the Regev encoding scheme with message space R_q is $\mathcal{F}_{\mathsf{auto}}$-KDM-secure. Then, for all efficient adversaries $\mathcal{A}$, $\mathsf{Hyb}_1(\mathcal{A}) \overset{c}{\approx} \mathsf{Hyb}_2(\mathcal{A})$.*

Proof. Suppose there exists an efficient adversary $\mathcal{A}$ such that $\left|\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1] - \Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1]\right| \geq \varepsilon$ for some non-negligible ε. We use $\mathcal{A}$ to construct an algorithm $\mathcal{B}$ that breaks KDM security of the encoding scheme:

1. The KDM security challenger starts by sampling a secret key $\mathbf{s} = [-\tilde{s}_0 \mid 1]^\intercal \leftarrow \mathsf{KeyGen}(1^\lambda, 1^1)$.
2. Algorithm $\mathcal{B}$ samples for itself a key $\mathbf{S} = [-\tilde{\mathbf{s}}_1 \mid \mathbf{I}_n]^\intercal \leftarrow \mathsf{KeyGen}(1^\lambda, 1^n)$.
3. Algorithm $\mathcal{B}$ samples $\mathbf{R}_1 \xleftarrow{R} R_q^{n \times m_{\mathsf{conv}}}$ and $\mathbf{R}_2 \xleftarrow{R} R_q^{n \times 2t_{\mathsf{conv}}}$. It computes $\mathbf{V} \leftarrow \mathsf{Encode}(\mathbf{S}, \mathbf{R}_1)$ and $\mathbf{W} \leftarrow \mathsf{Encode}(\mathbf{S}, \mathbf{R}_2)$. It sets $\mathsf{ck} = (\mathbf{V}, \mathbf{W}, \mathbf{\Pi})$ where $\mathbf{\Pi}$ is defined as in the real scheme.
4. For each $i \in [0, \rho-1]$, it queries its encryption oracle on each component of $-\tau_{2^{\rho-i}+1}(\tilde{s}_0) \cdot \mathbf{g}_{z_{\mathsf{coeff}}}$ and concatenates the encodings to obtain $\mathbf{W}_i$. It gives $\mathsf{pp} = (\mathsf{ck}, \mathbf{W}_0, \ldots, \mathbf{W}_{\rho-1})$ to $\mathcal{A}$.
5. Algorithm $\mathcal{B}$ samples a bit $\beta \xleftarrow{R} \{0,1\}$. When $\mathcal{A}$ makes a query $(\mathsf{idx}_0, \mathsf{idx}_1)$, algorithm $\mathcal{B}$ computes μ according to the real scheme (following Eq. (4.1)) and queries the encryption oracle on μ to obtain the encoded query $\mathbf{c}$. It replies to $\mathcal{A}$ with $\mathbf{c}$.
6. Finally, algorithm $\mathcal{A}$ outputs a bit $\beta' \in \{0,1\}$ and algorithm $\mathcal{B}$ outputs 1 if $\beta = \beta'$ and 0 otherwise.

Let $b \in \{0,1\}$ be the bit the KDM challenger samples.

- If $b = 0$, then the challenger replies to $\mathcal{B}$'s queries with the encryption of the queried message under $\mathbf{s}$. In this case, algorithm $\mathcal{B}$ perfectly simulates the distribution in Hyb_1 and outputs 1 with probability $\Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]$. Correspondingly, $\mathcal{B}$ outputs $b = 0$ with probability $1 - \Pr[\mathsf{Hyb}_1(\mathcal{A}) = 1]$.

[13] Strictly speaking, algorithm $\mathcal{B}$ constructs $\mathbf{V}$ and $\mathbf{W}$ by concatenating encodings of the columns of $-\tilde{s}_0 \cdot \mathbf{G}_{n,z_{\mathsf{conv}}}$ and $-\tilde{\mathbf{s}}_1 \cdot (\mathbf{s}^\intercal \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\intercal)$, respectively (see Construction 2.8). This way, we only rely on KDM-security for the message space R_q^n.

- If $b = 1$, then the challenger replies to $\mathcal{B}$ with encryption of random messages. This corresponds to the distribution in Hyb_2, and $\mathcal{B}$ outputs 1 with probability $\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1]$.

By an analogous calculation as in the proof of Lemma B.1, the advantage of algorithm $\mathcal{B}$ is $\varepsilon/2$. □

As noted above, for all adversaries $\mathcal{A}$, $\Pr[\mathsf{Hyb}_2(\mathcal{A}) = 1] = 1/2$. The claim now follows from Lemmas B.1 and B.2. Correspondingly, for all efficient adversaries $\mathcal{A}$, $\Pr[\mathsf{Hyb}_0(\mathcal{A}) = 1] \leq 1/2 + \mathsf{negl}(\lambda)$. □