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

Samir Jordan Menon*, David J. Wu†

*Unaffiliated
menon.samir@gmail.com

†UT Austin
dwu4@cs.utexas.edu

2022 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-1316-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/SP46214.2022.9833700

***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 $4.5\times$ reduction in query size, $1.5\times$ reduction in response size, and $2\times$ 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 $1.9\times$ greater than that of the no-privacy baseline where the client directly downloads the desired record.**

I. Introduction

A private information retrieval (PIR) [1] 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 [2, 3, 4, 5], contact discovery [6, 7], private contact tracing [8], private navigation [9, 10], and safe browsing [11].

Private information retrieval protocols fall under two main categories: (1) multi-server protocols where the database is replicated across multiple servers [1]; and (2) single-server protocols where the database lives on a single server [12]. We refer to [13, 14] 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 [15]. While this assumption applied to earlier number-theoretic PIR schemes [12, 16, 17, 18], recent lattice-based constructions [19, 5, 20, 21, 22, 23] 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 [5, 23] for databases with millions of records and tens of gigabytes of data).

The current state-of-the-art single-server PIR, OnionPIR [23], 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 [24]. 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 $4.5\times$ reduction in query size, a $1.5\times$ increase in the rate, and a $2\times$ increase in the server throughput compared to OnionPIR [23] (see Table I).

Like previous PIR protocols [5, 20, 21, 25, 23, 22], 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 ($2\times$ higher than OnionPIR) and an effective server throughput of up to 1.5 GB/s (roughly $10\times$ higher than OnionPIR). We provide more detailed benchmarks in Section V-C and Table III.
- 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 V-C and Table II, 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 $2.1\times$ increase in rate and $5.5\times$ increase in throughput compared to the base version of SPIRAL. However, the size of the public parameters is $4.2\times$ higher and the query size is over $2000\times$ 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 ($\geq$ 30 KB), the server throughput of our streaming constructions is 2–$4\times$ 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 $2.9\times$ slower than the best two-server PIR using *hardware-accelerated* AES [24], hardware acceleration for the lattice-based building blocks underlying our construction could help bridge this gap (e.g., [26]).

A limitation of SPIRAL is that it generally requires larger *public parameters* compared with previous schemes. To compare, the public parameters in SealPIR [5], FastPIR [22], and OnionPIR [23] 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 I-B and III) 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).

A. Background on Lattice-Based PIR

The most efficient single-server PIR protocols [4, 19, 5, 20, 21, 25, 23] use lattice-based fully homomorphic encryption (FHE) schemes [27, 28, 29, 30, 31, 32].[1] These protocols follow the general paradigm of constructing PIR from homomorphic encryption [12]. 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 [4, 19, 5] relied on recursive composition and additive homomorphism, more recent protocols [20, 21, 25, 23] 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 [19, 4, 5, 20, 25, 23] are based on either the Regev encryption scheme [33], which is additively homomorphic, or its generalization, the Brakerski/Fan-Vercauteren (BFV) scheme [29, 30], 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 [19, 4, 5].

A solution: FHE composition. Recently, Chillotti et al. [35, 36] 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 [33] with a ciphertext encrypted under the encryption scheme of Gentry, Sahai, and Waters (GSW) [32]. 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 [37] can encrypt $n \times n$ plaintext elements with a ciphertext of size $n \times (n+1)$.

[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 [27] 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) [34].

The external product operation from [35, 36] 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 [20] and the recently-implemented OnionPIR protocol [23]. 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.

B. 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 V). Here, we provide an overview of our techniques.

High rate via ciphertext amortization. To achieve higher rate, we take the Gentry-Halevi [20] approach of using the amortized version of Regev encryption [37] (over rings [38]) 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 $1.5\times$–$6\times$ better than OnionPIR (Table I) 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 [5, 39], Gentry and Halevi estimate that the size of the queries in their construction to be 30 MB, which is more than $450\times$ worse compared to existing schemes. The reason for this blowup is that the Angel et al. query compression technique [5] 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 [5, 39].

In this work, we describe a general technique for translating between matrix Regev ciphertexts (of *any* dimension) and GSW ciphertexts (Section III). 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 [5, 39], 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 III.1), 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 I-A). 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. [5] 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 [5] 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 $\mathcal{C}_{\mathsf{Reg}}$ and $\mathcal{C}_{\mathsf{GSW}}$.
  - **First dimension expansion:** Next, the server uses $\mathcal{C}_{\mathsf{Reg}}$ 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 $\mathsf{ScalToMat}$ 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 III-A.
  - **GSW ciphertext expansion:** The server then uses $\mathcal{C}_{\mathsf{GSW}}$ 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 $\mathsf{RegevToGSW}$ that we describe in Section III-B.
- **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 [20] 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

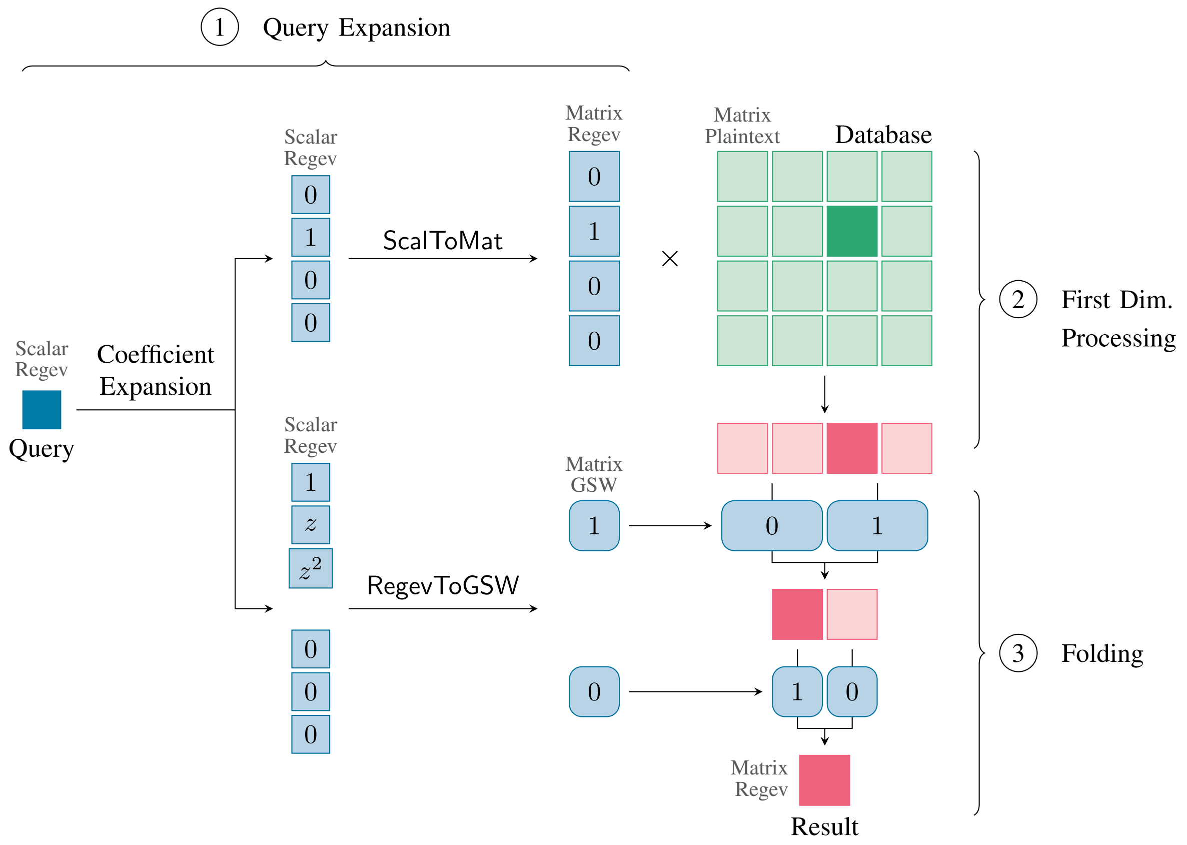

Fig. 1: Server processing for a single SPIRAL query. The parameter z here is a decomposition base and is used for the translation between Regev ciphertexts and GSW ciphertexts. We refer to Section III-B for more details.

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 IV 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 IV-A.

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 [19]. We describe our parameter selection methodology in Section V-A. Our system allows choosing parameters that either minimize 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 V-C. In Appendix D, 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 $1.9\times$ higher than the no-privacy baseline (where the client directly downloads the movie of interest). This is a $9\times$ reduction in cost compared to the previous state of the art, OnionPIR [23].

II. 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 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 recall the formal definition of a private information retrieval protocol in Appendix A.

A. Lattice-Based Homomorphic Encodings

Like previous lattice-based PIR protocols [19, 5, 20, 25, 23, 22], 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 $R_q = R/qR$.

Ring learning with errors. The security of our PIR protocol relies on the ring learning with errors (RLWE) problem [38]. Very briefly, the RLWE assumption (in normal form [40]) states that the following two distributions are computationally indistinguishable: $(\mathbf{a}, s\mathbf{a} + \mathbf{e})$ and $(\mathbf{a}, \mathbf{u})$, where $\mathbf{a} \xleftarrow{\text{R}} R_q^m$, $s \leftarrow \chi$, $\mathbf{e} \leftarrow \chi^m$, $\mathbf{u} \xleftarrow{\text{R}} R_q^m$, $m = \mathsf{poly}(\lambda)$ is the number of samples, and χ is an error distribution (typically a discrete Gaussian).

Gadget matrices. In this work, we use gadget matrices with *different* bases [41]. 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 $[-z/2, z/2]$. 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 [33, 37, 38]. 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. Here, we provide an informal description of the encoding scheme and defer the formal description to the full version of this paper [42].

Let n be the plaintext dimension and q be the encoding modulus. The encoding scheme consists of a pair of efficient algorithms $(\mathsf{KeyGen}, \mathsf{Encode})$. The KeyGen algorithm samples a secret key $\mathbf{S} \in R_q^{(n+1) \times n}$. The Encode algorithm takes the secret key $\mathbf{S}$ and a matrix $\mathbf{M} \in R_q^{n \times n}$, and outputs an encoding $\mathbf{C} \in R_q^{(n+1) \times n}$ where $\mathbf{S}^\mathsf{T}\mathbf{C} = \mathbf{M} + \mathbf{E}$ and the entries of $\mathbf{E}$ are small (i.e., sampled from the error distribution χ). The Regev encoding scheme supports homomorphic addition (Add) and scalar multiplication (ScalarMul). In particular, if $\mathbf{C}_1$ and $\mathbf{C}_2$ are encodings of $\mathbf{M}_1$ and $\mathbf{M}_2$, then $\mathbf{C}_1 + \mathbf{C}_2$ is an encoding of $\mathbf{M}_1 + \mathbf{M}_2$ with a larger error. Similarly, if $\mathbf{T} \in R_q^{(n+1) \times n}$, then $\mathbf{C}_1\mathbf{T}$ is an encoding of $\mathbf{M}_1\mathbf{T}$.

Finally, if $p \ll q$ is a plaintext modulus, we say that $\mathbf{C}$ is a *redundant* encoding of a plaintext matrix $\hat{\mathbf{M}} \in R_p^{n \times n}$ if $\mathbf{S}^\mathsf{T}\mathbf{C} = \lfloor q/p \rfloor \hat{\mathbf{M}} + \mathbf{E}$, for some $\|\mathbf{E}\| \ll q/(2p)$. In this case, we can recover the encoded message $\mathbf{M}$ by computing $p/q \cdot \mathbf{S}^\mathsf{T}\mathbf{C}$ and rounding to the nearest integer. We denote this operation by $\mathsf{Decode}(\mathbf{S}^\mathsf{T}\mathbf{C})$.

GSW encodings. Next, we describe the key properties of the Gentry-Sahai-Waters encryption scheme [32]. Similar to the case for Regev encodings, we describe the scheme as an encoding scheme (without an explicit decryption functionality). We provide the high-level details here and defer the formal description to the full version of this paper [42].

Let n be the dimension and $z \in \mathbb{N}$ be a decomposition base. The GSW encoding scheme also consists of a pair of algorithms $(\mathsf{KeyGen}, \mathsf{Encode})$. The KeyGen algorithm is identical to that for the Regev encoding scheme while Encode takes the secret key $\mathbf{S}$ and a *scalar* $\mu \in R_q$ and outputs an encoding $\mathbf{C} \in R_q^{(n+1) \times n}$ where $\mathbf{S}^\mathsf{T}\mathbf{C} = \mu\mathbf{S}^\mathsf{T}\mathbf{G}_{n+1,z} + \mathbf{E}$ and the entries of $\mathbf{E}$ are small. By construction, if $\mathbf{C}$ is a GSW encoding of a bit $\mu \in \{0, 1\}$, then $\mathsf{Complement}(\mathbf{C}) := \mathbf{G}_{n+1,z} - \mathbf{C}$ is a GSW encoding of the complement $1 - \mu$.

Regev-GSW multiplication. The main homomorphism that we rely on in this work is a way to multiply a Regev encoding with a GSW encoding to obtain a Regev encoding of the product [36, 20, 39, 23]. In particular, if $\mathbf{C}_{\mathsf{Regev}}$ is an encoding of a matrix $\mathbf{M} \in R_q^{n \times n}$ and $\mathbf{C}_{\mathsf{GSW}}$ is an encoding of a scalar $\mu \in R_q$ under the *same* secret key $\mathbf{S} \in R_q^{(n+1) \times n}$, the multiplication algorithm $\mathsf{Multiply}(\mathbf{C}_{\mathsf{GSW}}, \mathbf{C}_{\mathsf{Regev}})$ outputs the Regev encoding $\mathbf{C}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}})$. By definition,

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

where we write $\mathbf{G} = \mathbf{G}_{n+1,z}$ and $\tilde{\mathbf{E}} = \mu\mathbf{E}_{\mathsf{Regev}} + \mathbf{E}_{\mathsf{GSW}}\mathbf{G}^{-1}(\mathbf{C}_{\mathsf{Regev}})$. Thus, $\mathbf{C}_{\mathsf{GSW}}\mathbf{G}_{n+1,z}^{-1}(\mathbf{C}_{\mathsf{Regev}})$ is a Regev encoding of $\mu\mathbf{M}$ with error $\tilde{\mathbf{E}}$, which is small as long as μ, $\mathbf{E}_{\mathsf{Regev}}$, and $\mathbf{E}_{\mathsf{GSW}}$ are small.

III. 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 I-A). A naïve implementation would require at least one encoding for each dimension of the hypercube in the query. Previously, Angel et al. [5] and Chen et al. [39] introduced a query compression algorithm to pack the ciphertexts for the different dimensions into a *single* RLWE ciphertext (specifically, a BFV ciphertext [29, 30]).

To achieve higher rate and reduce noise growth, Spiral follows the Gentry-Halevi approach [20] of encoding the index along the first dimension using a matrix Regev encoding and the subsequent dimensions using GSW encodings (see Section IV). In this section, we introduce new building blocks to enable an analogous query compression approach as [5, 39] 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 [5] or OnionPIR [23]. However, due to better control of noise growth, Spiral can be instantiated with smaller lattice parameters, thus resulting in *smaller* queries (see Section V-C). Our approach relies on four main ingredients which we describe in this section:

- In Section III-A, 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 III-B, 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 $n \geq 1$ (Section III-B). 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 [5, 39] to pack multiple Regev encodings of *scalars* into a *single* Regev encoding (of a polynomial).
- Finally, after server processing, we apply modulus switching [28, 31] 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 III-D). 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 [28, 31]. 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 [5, 39] also require publishing key-switching matrices (to support automorphisms), which also necessitate making a circular security assumption.

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 $\mathbf{a} \xleftarrow{\text{R}} R_q^m$ 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 $\mathbf{W}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) + [\ \mathbf{0}^n \mid c_1\mathbf{I}_n\ ]^{\mathsf{T}}$.

We refer to Appendix B for a sketch of the correctness proof and to the full version of this paper [42] for the full details.

B. 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. [39] 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. This will enable more flexibility in parameter selection (see Section V-A) 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 $\mathbf{a} \xleftarrow{R} R_q^{2t_{\mathsf{conv}}}$ and $\mathbf{E} \leftarrow \chi^{n \times 2t_{\mathsf{conv}}}$ and construct the matrix

$$\mathbf{V} = \begin{bmatrix} \mathbf{a}^\mathsf{T} \\ \tilde{\mathbf{s}}_{\mathsf{GSW}} \mathbf{a}^\mathsf{T} + \mathbf{E} \end{bmatrix} + \begin{bmatrix} \mathbf{0}^{1 \times 2t_{\mathsf{conv}}} \\ -\tilde{\mathbf{s}}_{\mathsf{GSW}} \cdot (\mathbf{s}_{\mathsf{Regev}}^\mathsf{T} \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\mathsf{T}) \end{bmatrix}. \tag{III.1}$$

– Define the permutation matrix $\mathbf{\Pi} \in \{0,1\}^{m_{\mathsf{GSW}} \times m_{\mathsf{GSW}}}$ such that

$$\begin{bmatrix} \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} & \mathbf{0}^{1 \times n t_{\mathsf{GSW}}} \\ \mathbf{0}^{n \times t_{\mathsf{GSW}}} & \mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} \otimes \mathbf{I}_n \end{bmatrix} \mathbf{\Pi} = \mathbf{G}_{n+1, z_{\mathsf{GSW}}}.$$

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 $\hat{\mathbf{C}} = [\ \mathbf{c}_1 \mid \cdots \mid \mathbf{c}_{t_{\mathsf{GSW}}}\ ] \in R_q^{2 \times t_{\mathsf{GSW}}}$.

We refer to Appendix B for a sketch of the correctness proof and to the full version of this paper [42] for the full details.

Remark III.1 (Compressing GSW Encodings)**.** The $\mathsf{RegevToGSW}$ algorithm takes t_{GSW} Regev encodings (consisting of $2 \cdot t_{\mathsf{GSW}}$ elements of R_q) and outputs a single GSW encoding with $(n+1)m_{\mathsf{GSW}} = (n+1)^2 t_{\mathsf{GSW}}$ 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.

C. Coefficient Extraction on Regev Encodings

The next ingredient we require is the coefficient expansion algorithm by Angel et al. [5] and extended by Chen et al. [39]. 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 [5, 39], so we defer the description to Appendix C.

D. Modulus Switching

Modulus switching [28, 31] 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 [28, 31, 43, 20] 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. 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. We refer to the the full version of this paper [42] for additional discussion. We now describe our variant of the modulus switching procedure $\mathsf{ModulusSwitch}$ along with an encoding-recovery procedure $\mathsf{Recover}$ that takes a rescaled encoding $(\hat{\mathbf{c}}_1, \hat{\mathbf{C}}_2)$ (as output by $\mathsf{ModulusSwitch}$) 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 Decode procedure from Section II-A. Both the $\mathsf{ModulusSwitch}$ and $\mathsf{Recover}$ algorithms are parameterized by a pair of moduli $q_1, q_2 \in \mathbb{N}$. We provide the correctness analysis in the full version of this paper [42].

- $\mathsf{ModulusSwitch}_{q_1,q_2}(\mathbf{C})$: 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 $\mathbf{Z} = \lfloor (q_1/q_2)(-\tilde{\mathbf{s}}\hat{\mathbf{c}}_1^\mathsf{T}) \rceil + \hat{\mathbf{C}}_2$, and output $\mathbf{Z} \bmod q_1$.

IV. The Spiral Protocol

The structure of the basic Spiral protocol follows recent constructions of PIR based on composing Regev-based encryption with GSW encryption [20, 23]. The primary difference is that it uses the techniques from Section III 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 I-B and Fig. 1 for a general overview.

Construction IV.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 protocol 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 III-C and Algorithm 1);
  - z_{conv} is the decomposition base used to translate scalar Regev encodings into matrix Regev encodings (Section III-A); 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 V-A 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 give the SPIRAL protocol in Fig. 2.

Remark IV.2 (Query Size Trade-off and the SPIRALSTREAM Protocol)**.** To reduce noise growth in Construction IV.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 V, 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 III-B and Remark III.1) 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. Correctness of Construction IV.1 holds as long as the encoding modulus q is large enough to accommodate the noise accumulation from the homomorphic operations. We give a detailed description of our parameter selection methodology in Section V-A. We provide a formal statement in the full version of this paper [42].

Security. Security of our construction follows from the RLWE assumption and a circular security assumption (for the public parameters). Specifically, the query in SPIRAL is a Regev encryption of the query, and the public parameters consist of key-switching matrices, which are encryptions of key-dependent messages. Security can thus be based on RLWE and similar circular-security assumptions as those underlying previous lattice-based PIR schemes [5, 23]. We provide the formal statement and analysis in the full version of this paper [42].

A. 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 V-A, 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 describe our packing approach in the full version of this paper [42].

V. 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.

A. Automatic Parameter Selection

Parameter selection trade-offs. We now describe our general methodology for selecting parameters to support a database

- $\mathsf{Setup}(1^\lambda, 1^N)$: 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
  $$\mu(x) := 2^{-r_1} \mu_{i^*}(x^2) + 2^{-r_2} x \mu_{j^*}(x^2) \in R_q, \tag{IV.1}$$
  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 $\mathbf{C}_j^{(\mathsf{GSW})} \leftarrow \mathsf{RegevToGSW}\big(\mathsf{ck}, \mathbf{c}_{(j-1)t_{\mathsf{GSW}}+1}^{(\mathsf{GSW})}, \ldots, \mathbf{c}_{j t_{\mathsf{GSW}}}^{(\mathsf{GSW})}\big)$.

     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
  $$\mathbf{C}_j^{(r)} = \mathsf{Add}\left(\mathsf{Multiply}\left(\mathsf{Complement}(\mathbf{C}_r^{(\mathsf{GSW})}), \mathbf{C}_j^{(r-1)}\right), \mathsf{Multiply}\left(\mathbf{C}_r^{(\mathsf{GSW})}, \mathbf{C}_{2^{\nu_2-r}+j}^{(r-1)}\right)\right). \tag{IV.2}$$
  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 $\mathbf{C} \leftarrow \mathsf{Decode}(\mathbf{Z}) \in R_p^{n \times n}$.

Fig. 2: The SPIRAL PIR protocol.

$\mathcal{D}$ with up to N records, where each record is at most S bits. The parameters of interest in Construction IV.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 $z_{\mathsf{coeff,Reg}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}}$[3], and a 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 V-B). We choose parameters to tolerate a correctness error of at most 2^{-40} and a security level of 128 bits of (classical) security. We refer to the full version of this paper [42] for a more detailed

[3]For finer control over the noise introduced by the ciphertext expansion algorithm [5, 39], we use different decomposition bases to expand the Regev and the GSW ciphertexts (denoted $z_{\mathsf{coeff,Reg}}$ and $z_{\mathsf{coeff,GSW}}$, respectively). We refer to the full version of this paper [42] for additional details.

discussion of the different scheme parameters.

Automatic parameter selection. Balancing the different scheme parameters is important for obtaining a good trade-off between computational costs and communication. Similar to XPIR [19], 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). 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 $N \leq 2^{22}$. 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 $t_{\mathsf{coeff,Reg}}$, $t_{\mathsf{coeff,GSW}}$, t_{conv}, t_{GSW} (which are functions of $z_{\mathsf{coeff,Reg}}, z_{\mathsf{coeff,GSW}}, z_{\mathsf{conv}}, z_{\mathsf{GSW}}$), 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 $t_{\mathsf{coeff,Reg}}, t_{\mathsf{conv}} \in \{2, 4, 8, 16, 32, 56\}$ and $t_{\mathsf{GSW}} \in [2, 56]$.[4] We fix the decomposition base $z_{\mathsf{coeff,GSW}} = 2$, which fixes the dimension $t_{\mathsf{coeff,GSW}} = 56$. Finally, we consider all database configurations $\nu_1, \nu_2 \in [2, 11]$.[5] 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 fitting a series of linear models based on empirically-measured running times for the different steps of the protocol. We provide the full description in the full version of this paper [42]. Applying the AWS monetary cost model (see Section V-B) 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 use an almost identical procedure to select parameters for SPIRALPACK.

Remark V.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 V-C.

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

[5]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.

B. 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 IV.2) can achieve even better performance. In our experimental evaluation (Section V-C), 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 [44]); and (2) metadata-hiding messaging systems where a user is repeatedly reading from a "mailbox" (e.g. Pung [4] or Addra [22]). 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 [22] are designed specifically for the streaming setting and as such, achieve higher server throughput compared to SealPIR [5], but require larger queries. We can easily adapt SPIRAL to the setting using the approach from Remark IV.2. Namely, in SPIRALSTREAM, the client uploads all of the Regev encodings directly without using the query packing approach from [5]. As we show in Section V-C, 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 V-A) 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 [45, 46, 47].

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)) [48, 49]. 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 [23], 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 IV.1.

Code. Our implementation consists of roughly 4,000 lines of C++.[6] We adapt the procedure from the SEAL homomorphic encryption library [50] to implement the FFTs for homomorphic evaluation. We use the Intel HEXL library [51] to implement FFTs in the response decoding procedures.

Experimental setup. We compare our PIR protocol against the public implementations of SealPIR [5], FastPIR [22], and OnionPIR [23]. 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 [5, 23, 22], 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 [52]. 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. 4).

C. *Evaluation Results for* SPIRAL

We start by comparing the performance of SPIRAL and SPIRALSTREAM to existing systems on three different database configurations in Table I:

- A database with many small records (2^{20} records of size 256 B). This is a common baseline for PIR [4, 25, 22].
- A database with moderate-size records (2^{18} records of size 30 KB). This is the optimal configuration for OnionPIR [23].
- 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 $2.2\times$ 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 $2400\times$ 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 III. 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 $5.6\times$ higher than the previous state-of-the-art; SPIRALSTREAM simultaneously achieves a $2\times$ increase in rate as well. Measured in terms of monetary cost, SPIRALSTREAM is $5.4\times$ less expensive compared to OnionPIR for this database configuration. The trade-off is SPIRALSTREAM requires larger

[6]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 [25].

TABLE I: Comparison of Spiral and SpiralStream with recent PIR protocols (SealPIR [5], FastPIR [22], MulPIR [25], OnionPIR [23]) 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 V-B).

queries, though this is a less significant factor in streaming settings where the same query is reused across multiple requests.

Packing. In Table II (Appendix D), we compare the packed versions of Spiral and SpiralStream with the vanilla versions on each of the main benchmarks. As shown in Table II, packing enables higher rates and throughput, but requires larger public parameter for the packing keys (Section IV-A). 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. 3 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 V-B, 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. 5 and Table III (Appendix D). 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 [22] which have queries of the same size. FastPIR is a PIR protocol tailored for the 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

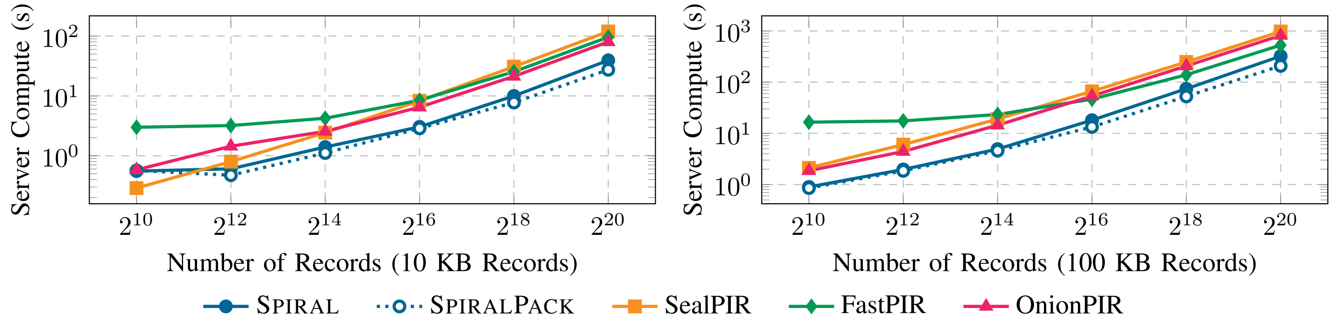

Fig. 3: Server computation time as a function of database size for different PIR protocols.

improvements to the server throughput and the rate, and we explore these trade-offs in more detail in the full version of this paper [42].

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).

Microbenchmarks and other measurements. Due to space limitations, we include additional microbenchmarks and system analysis in Appendix D and the full version of this paper [42]. We also estimate the concrete costs of SPIRAL to support several application settings.

VI. Related Work

Number-theoretic constructions. Many early constructions of single-server PIR [17, 53] follow the Kushilevitz-Ostrovsky paradigm [12] based on homomorphic encryption. These were typically instantiated using number-theoretic assumptions such as Paillier [54] or the Damgård-Jurik [55] encryption schemes. Another line of works [16, 18] gave constructions with polylogarithmic communication from the ϕ-hiding assumption. Döttling et al. [56] 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 [19], a number of systems have progressively reduced the computational cost of single-server PIR [4, 5, 20, 21, 25, 22, 23]. While early constructions only relied on additive homomorphism, more recent constructions also incorporate multiplicative homomorphism for better concrete efficiency [20, 21, 25, 23]. 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* [57, 58, 59, 5] where the server's computational cost is amortized over a *batch* of queries. In particular, Angel et al. [5] 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* [60, 23, 61, 62] 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 [23] 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 [61] 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. [62] 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* [57] or *doubly-efficient PIR* [63, 64] 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. [63] and Canetti et al. [64] 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 [65] (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 [1, 66, 67, 68, 69, 70, 24] consider the multi-server setting where the database is replicated across several *non-colluding* servers (see also the survey by Gasarch [13] 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, a Microsoft Research Faculty Fellowship, and a Google Research Scholar award.

References

[1] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *FOCS*, 1995.

[2] P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "Pir-tor: Scalable anonymous communication using private information retrieval," in *USENIX Security*, 2011.

[3] A. Kwon, D. Lazar, S. Devadas, and B. Ford, "Riffle: An efficient communication system with strong anonymity," *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 2, 2016.

[4] S. Angel and S. T. V. Setty, "Unobservable communication over fully untrusted infrastructure," in *OSDI*, 2016.

[5] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *IEEE S&P*, 2018.

[6] N. Borisov, G. Danezis, and I. Goldberg, "DP5: A private presence service," *Proc. Priv. Enhancing Technol.*, vol. 2015, no. 2, 2015.

[7] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "PIR-PSI: scaling private contact discovery," *Proc. Priv. Enhancing Technol.*, vol. 2018, no. 4, 2018.

[8] N. Trieu, K. Shehata, P. Saxena, R. Shokri, and D. Song, "Epione: Lightweight contact tracing with strong privacy," *IEEE Data Eng. Bull.*, vol. 43, no. 2, 2020.

[9] E. Fung, G. Kellaris, and D. Papadias, "Combining differential privacy and PIR for efficient strong location privacy," in *SSTD*, 2015.

[10] D. J. Wu, J. Zimmerman, J. Planul, and J. C. Mitchell, "Privacy-preserving shortest path computation," in *NDSS*, 2016.

[11] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with checklist," in *USENIX Security*, 2021.

[12] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *FOCS*, 1997.

[13] W. I. Gasarch, "A survey on private information retrieval," *Bull. EATCS*, vol. 82, 2004.

[14] R. Ostrovsky and W. E. Skeith III, "A survey of single database PIR: techniques and applications," *IACR Cryptol. ePrint Arch.*, 2007.

[15] R. Sion and B. Carbunar, "On the practicality of private information retrieval," in *NDSS*, 2007.

[16] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *EUROCRYPT*, 1999.

[17] Y. Chang, "Single database private information retrieval with logarithmic communication," in *ACISP*, 2004.

[18] C. Gentry and Z. Ramzan, "Single-database private information retrieval with constant communication rate," in *ICALP*, 2005.

[19] C. A. Melchor, J. Barrier, L. Fousse, and M. Killijian, "XPIR : Private information retrieval for everyone," *Proc. Priv. Enhancing Technol.*, vol. 2016, no. 2, 2016.

[20] C. Gentry and S. Halevi, "Compressible FHE with applications to PIR," in *TCC*, 2019.

[21] J. Park and M. Tibouchi, "SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval," in *ESORICS*, 2020.

[22] I. Ahmad, Y. Yang, D. Agrawal, A. E. Abbadi, and T. Gupta, "Addra: Metadata-private voice communication over fully untrusted infrastructure," in *OSDI*, 2021.

[23] M. H. Mughees, H. Chen, and L. Ren, "OnionPIR: Response efficient single-server PIR," in *ACM CCS*, 2021.

[24] S. M. Hafiz and R. Henry, "A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server PIR," *Proc. Priv. Enhancing Technol.*, vol. 2019, no. 4, 2019.

[25] A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo, "Communication-computation trade-offs in PIR," in *USENIX Security*, 2021.

[26] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. G. Dreslinski, C. Peikert, and D. Sánchez, "F1: A fast and programmable accelerator for fully homomorphic encryption," in *MICRO*, pp. 238–252, 2021.

[27] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *STOC*, 2009.

[28] Z. Brakerski and V. Vaikuntanathan, "Efficient fully homomorphic encryption from (standard) LWE," in *FOCS*, 2011.

[29] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *CRYPTO*, 2012.

[30] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, 2012.

[31] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) fully homomorphic encryption without bootstrapping," in *ITCS*, 2012.

[32] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *CRYPTO*, 2013.

[33] O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," in *STOC*, 2005.

[34] A. Viand, P. Jattke, and A. Hithnawi, "SoK: Fully homomorphic encryption compilers," in *IEEE S&P*, 2021.

[35] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *IACR Cryptol. ePrint Arch.*, 2018.

[36] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: fast fully homomorphic encryption over the torus," *J. Cryptol.*, vol. 33, no. 1, 2020.

[37] C. Peikert, V. Vaikuntanathan, and B. Waters, "A framework for efficient and composable oblivious transfer," in *CRYPTO*, 2008.

[38] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *EUROCRYPT*, 2010.

[39] H. Chen, I. Chillotti, and L. Ren, "Onion ring ORAM: efficient constant bandwidth oblivious RAM from (leveled) TFHE," in *ACM CCS*, 2019.

[40] B. Applebaum, D. Cash, C. Peikert, and A. Sahai, "Fast cryptographic primitives and circular-secure encryption based on hard learning problems," in *CRYPTO*, 2009.

[41] D. Micciancio and C. Peikert, "Trapdoors for lattices: Simpler, tighter, faster, smaller," in *EUROCRYPT*, 2012.

[42] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server PIR via FHE composition," *IACR Cryptol. ePrint Arch.*, 2022. Available at https://eprint.iacr.org/2022/368.pdf.

[43] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO*, 2012.

[44] T. Gupta, N. Crooks, W. Mulhern, S. T. V. Setty, L. Alvisi, and M. Walfish, "Scalable and private media consumption with popcorn," in *NSDI*, 2016.

[45] S. D. Galbraith, "Space-efficient variants of cryptosystems based on learning with errors," 2013.

[46] J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila, "Frodo: Take off the ring! practical, quantum-secure key exchange from LWE," in *ACM CCS*, 2016.

[47] Y. Ishai, H. Su, and D. J. Wu, "Shorter and faster post-quantum designated-verifier zkSNARKs from lattices," in *ACM CCS*, 2021.

[48] V. Lyubashevsky, D. Micciancio, C. Peikert, and A. Rosen, "SWIFFT: A modest proposal for FFT hashing," in *FSE*, 2008.

[49] P. Longa and M. Naehrig, "Speeding up the number theoretic transform for faster ideal lattice-based cryptography," in *CANS*, 2016.

[50] "Microsoft SEAL (release 3.2)." https://github.com/Microsoft/SEAL, Feb. 2019. Microsoft Research, Redmond, WA.

[51] F. Boemer, S. Kim, G. Seifu, F. D. de Souza, V. Gopal, *et al.*, "Intel HEXL (release 1.2)." https://github.com/intel/hexl, Sept. 2021.

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

[53] H. Lipmaa, "An oblivious transfer protocol with log-squared communication," in *ISC*, 2005.

[54] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, 1999.

[55] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *PKC*, 2001.

[56] N. Döttling, S. Garg, Y. Ishai, G. Malavolta, T. Mour, and R. Ostrovsky, "Trapdoor hash functions and their applications," in *CRYPTO*, 2019.

[57] A. Beimel, Y. Ishai, and T. Malkin, "Reducing the servers computation in private information retrieval: PIR with preprocessing," in *CRYPTO*, 2000.

[58] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *STOC*, 2004.

[59] J. Groth, A. Kiayias, and H. Lipmaa, "Multi-query computationally-private information retrieval with constant communication rate," in *PKC*, 2010.

[60] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *ACM CCS*, 2018.
[61] H. Corrigan-Gibbs and D. Kogan, "Private information retrieval with sublinear online time," in *EUROCRYPT*, 2020.
[62] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan, "Single-server private information retrieval with sublinear amortized time," in *EUROCRYPT*, 2022.
[63] E. Boyle, Y. Ishai, R. Pass, and M. Wootters, "Can we access a database both locally and privately?," in *TCC*, 2017.
[64] R. Canetti, J. Holmgren, and S. Richelson, "Towards doubly efficient private information retrieval," in *TCC*, 2017.
[65] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," in *CRYPTO*, 2001.
[66] S. Yekhanin, "Towards 3-query locally decodable codes of subexponential length," in *STOC*, 2007.
[67] K. Efremenko, "3-query locally decodable codes of subexponential length," in *STOC*, 2009.
[68] A. Beimel, Y. Ishai, E. Kushilevitz, and I. Orlov, "Share conversion and private information retrieval," in *CCC*, 2012.
[69] N. Gilboa and Y. Ishai, "Distributed point functions and their applications," in *EUROCRYPT*, 2014.
[70] E. Boyle, N. Gilboa, and Y. Ishai, "Function secret sharing: Improvements and extensions," in *ACM CCS*, 2016.
[71] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT*, 2012.
[72] Speedtest, "Speedtest global index," 2022. https://www.speedtest.net/global-index. Last accessed: March 18, 2022.

Appendix A
PIR Definition

We now recall the standard definition of a two-message single-server PIR protocol [12]. Like most lattice-based PIR schemes [5, 20, 21, 22, 25, 23], 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 A.1 (Two-Message Single-Server PIR [12, 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 $\mathcal{D} = \{d_1, \dots, d_N\}$, 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 $\mathcal{D} = \{d_1, \dots, d_N\}$ 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.

Appendix B
Ciphertext Translation Algorithms

In this section, we give a sketch of the correctness proofs for the main ciphertext translation algorithms from Section II-A. We refer to the full version of this paper [42] for the formal analysis.

ScalToMat. To see correctness of ScalToMat, let $\mathbf{c}_0 = (c_0, c_1)$ be a Regev encoding of a scalar $\mu \in R_q$ with respect to the secret key $\mathbf{s}_0$ and error e. Let $\mathbf{W} \leftarrow \mathsf{ScalToMatSetup}(\mathbf{s}_0, \mathbf{S}_1, z)$ and $\mathbf{C}_1 \leftarrow \mathsf{ScalToMat}(\mathbf{W}, \mathbf{c}_0)$. Observe that $\mathbf{S}_1^\mathsf{T}\mathbf{W} = \mathbf{E} - \tilde{s}_0\mathbf{G}_{n,z}$. Then,
$$\begin{aligned}\mathbf{S}_1^\mathsf{T}\mathbf{C}_1 &= \mathbf{S}_1^\mathsf{T}\mathbf{W}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) + \mathbf{S}_1^\mathsf{T}\begin{bmatrix} \mathbf{0}^{1\times n} \\ c_1\mathbf{I}_n \end{bmatrix} \\ &= \mathbf{E}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) - \tilde{s}_0 c_0\mathbf{I}_n + c_1\mathbf{I}_n \\ &= \mathbf{E}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n) + \mathbf{I}_n(\mathbf{s}_0^\mathsf{T}\mathbf{c}) \\ &= \mu\mathbf{I}_n + e\mathbf{I}_n + \mathbf{E}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n),\end{aligned}$$
and we see that $\mathbf{C}_1$ is an encoding of $\mu\mathbf{I}_n$ with new error $e\mathbf{I}_n + \mathbf{E}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n)$. This transformation introduces a fixed additive error of $\mathbf{E}\mathbf{G}_{n,z}^{-1}(c_0\mathbf{I}_n)$, where $\mathbf{E}$ is freshly sampled from the error distribution.

RegevToGSW. To see correctness of RegevToGSW, suppose $\mathbf{c}_1, \dots, \mathbf{c}_{t_{\mathsf{GSW}}}$ are Regev encodings of $\mu, \mu z_{\mathsf{GSW}}, \dots, \mu z_{\mathsf{GSW}}^{t_{\mathsf{GSW}}-1} \in R_q$ under $\mathbf{s}_{\mathsf{Regev}}$ with errors $e_1, \dots, e_{t_{\mathsf{GSW}}} \in R_q$. Let $\mathbf{S}_{\mathsf{GSW}}$ be the GSW secret key, $\mathsf{ck} \leftarrow \mathsf{RegevToGSW}(\mathbf{s}_{\mathsf{Regev}}, \mathbf{S}_{\mathsf{GSW}}, z_{\mathsf{GSW}}, z_{\mathsf{conv}})$, and $\mathbf{C} \leftarrow \mathsf{RegevToGSW}(\mathsf{ck}, \mathbf{c}_1, \dots, \mathbf{c}_{t_{\mathsf{GSW}}})$. Let $\hat{\mathbf{e}} = [\ e_1 \mid \cdots \mid e_{t_{\mathsf{GSW}}}\ ]^\mathsf{T}$. Consider now the components of $\mathbf{S}_{\mathsf{GSW}}^\mathsf{T}\mathbf{C}$:

- By construction of $\mathbf{V}$ from Eq. (III.1), we have that $\mathbf{S}_{\mathsf{GSW}}^\mathsf{T}\mathbf{V} = \mathbf{E} - \tilde{\mathbf{s}}_{\mathsf{GSW}} \cdot (\mathbf{s}_{\mathsf{Regev}}^\mathsf{T} \otimes \mathbf{g}_{z_{\mathsf{conv}}}^\mathsf{T})$.
$$\begin{aligned}\mathbf{S}_{\mathsf{GSW}}^\mathsf{T}\mathbf{V}\mathbf{g}_{z_{\mathsf{conv}}}^{-1}(\hat{\mathbf{C}}) &= \mathbf{E}\mathbf{g}_{z_{\mathsf{conv}}}^{-1}(\hat{\mathbf{C}}) - \tilde{\mathbf{s}}_{\mathsf{GSW}}\mathbf{s}_{\mathsf{Regev}}^\mathsf{T}\hat{\mathbf{C}} \\ &= \mathbf{E}\mathbf{g}_{z_{\mathsf{conv}}}^{-1}(\hat{\mathbf{C}}) - \tilde{\mathbf{s}}_{\mathsf{GSW}}(\mu\mathbf{g}_{z_{\mathsf{GSW}}}^\mathsf{T} + \hat{\mathbf{e}}^\mathsf{T}).\end{aligned}$$

- By correctness of ScalToMat (Section III-A), for $i \in [t_{\mathsf{GSW}}]$, we have $\mathbf{S}_{\mathsf{GSW}}^{\mathsf{T}}\mathbf{C}_i = z_{\mathsf{GSW}}^{i-1} \cdot \mu \mathbf{I}_n + \mathbf{E}_i$. Letting $\tilde{\mathbf{E}} = [\ \mathbf{E}_1 \mid \cdots \mid \mathbf{E}_{t_{\mathsf{GSW}}}\ ]$, we can then write

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

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

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

Thus, $\mathbf{C}$ is a GSW encoding of μ with error $\tilde{\mathbf{E}}'$. This transformation scales the initial error $\hat{\mathbf{e}}$ in the Regev encodings by $\tilde{\mathbf{s}}_{\mathsf{GSW}}$. Since the components of the secret key are drawn from the error distribution, this only increases the noise magnitude by a few bits. Otherwise, the noise increases by a small additive factor, much like the case with the scalar-to-matrix transformation from Section III-A.

Appendix C
Coefficient Extraction on Regev Encodings

In this section, we recall the coefficient expansion algorithm by Angel et al. [5] and extended by Chen et al. [39]. 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 $\tau_\ell \colon R \to R$ 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 III-A and III-B), 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 $\tau \colon R_q \to R_q$, and a decomposition base $z \in \mathbb{N}$, let $t = \lfloor \log_z q \rfloor + 1$. Sample $\mathbf{a} \xleftarrow{\text{R}} R_q^t$, $\mathbf{e} \leftarrow \chi^t$, 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 $\tau \colon R_q \to R_q$, and encoding $\mathbf{c} = (c_0, c_1) \in R_q^2$, output $\mathbf{W}_\tau \mathbf{g}_z^{-1}(\tau(c_0)) + [\ 0 \mid \tau(c_1)\ ]^{\mathsf{T}}$.

We refer to previous works [31, 71] for the correctness and noise analysis for the automorphisms.

Coefficient expansion algorithm. We recall the coefficient expansion procedure by Angel et al. [5] and extended by Chen et al. [39]. 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 $\tau_\ell \colon R_q \to R_q$ and linear operations, and can be implemented *homomorphically* on encodings.

Algorithm 1: Coefficient expansion [5, 39].

```
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 $2^r \le d$
Output: the scaled coefficients
       $2^r \cdot (f_0, \ldots, f_{2^r-1}) \in \mathbb{Z}_q^{2^r}$
1 $f_0 \leftarrow f$
2 for $i = 0$ to $r - 1$ do
3     $\ell \leftarrow 2^{r-i} + 1$
4     for $j = 0$ to $2^i - 1$ do
5         $f'_j \leftarrow f_j \cdot x^{-2^j}$        ▷ $x^{-2^j} = -x^{d-2^j} \in R$
6         $f_j \leftarrow f_j + \tau_\ell(f_j)$   $f_{j+2^i} \leftarrow f'_j + \tau_\ell(f'_j)$
7     end
8 end
9 return $f_0, f_1, \ldots, f_{2^r-1}$
```

Remark C.1 (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 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).

Appendix D
Additional Experiments

In this section, we provide some additional benchmarks and evaluation of our system.

Optimizing for rate or throughput. In the the full version of this paper [42], we also explore how we can trade-off the server throughput or the rate of the protocol for query size or the public parameter size.

Microbenchmarks. Finally, we provide a more fine-grained breakdown of the different components of the client's and server's computation in Fig. 4. 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 (≈ 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.

Database	Metric	Best Previous	SPIRAL	SPIRALSTREAM	SPIRALPACK	SPIRALSTREAMPACK
$2^{20} \times$ 256B (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	1.69 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**
$2^{18} \times$ 30KB (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**
$2^{14} \times$ 100KB (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 II: Comparison for all four Spiral variants with the best alternative system: FastPIR [22] for the database with small records ($2^{20} \times$ 256B) and OnionPIR otherwise [23].

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 III: Performance of FastPIR [22], OnionPIR [23], 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.

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 IV.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 V-B, 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 $2\times$ 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 $2.1\times$ 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–$1.4\times$ 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 $2\times$ slowdown.

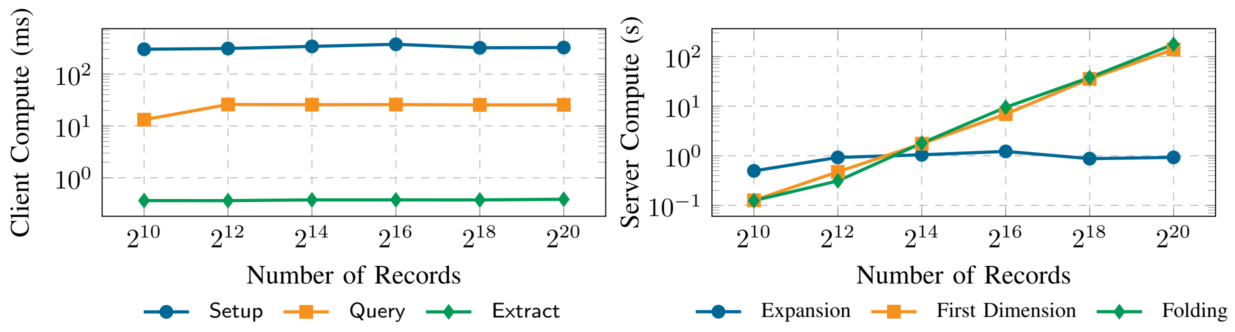

Fig. 4: Microbenchmarks for client and server computation in SPIRAL for processing databases with 100 KB records. The client computation consists of the Setup, Query, and Extract algorithms while the server computation consists of the Answer algorithm from Construction IV.1. We separately measure the costs of the query expansion (Step 1), first dimension processing (Step 2), and ciphertext folding (Step 3) in the Answer algorithm.

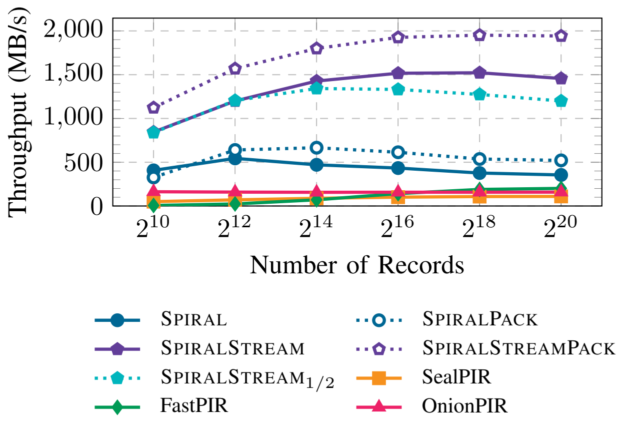

Fig. 5: Server throughput in the streaming setting as a function of the number of database records. In the streaming setting, we ignore the query expansion costs (if present) and use the optimal record size for each system. The query sizes for SealPIR, FastPIR, SPIRAL/SPIRALPACK, SPIRALSTREAM/SPIRALSTREAMPACK and $\text{SPIRALSTREAM}_{1/2}$, 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).

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 [22]. 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 [72]. 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.