YPIR: High-Throughput Single-Server PIR with Silent Preprocessing

Samir Jordan Menon
Blyss
samir@blyss.dev

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

Abstract

We introduce YPIR, a single-server private information retrieval (PIR) protocol that achieves high throughput (up to 75% of the memory bandwidth of the machine) *without* any offline communication. For retrieving a 1-bit (or 1-byte) record from a 32-GB database, YPIR achieves 10.9 GB/s/core server throughput and requires 2.5 MB of total communication. On the same setup, the state-of-the-art SimplePIR protocol achieves a 12.6 GB/s/core server throughput, requires 1.5 MB total communication, but *additionally* requires downloading a 724 MB hint in an offline phase. YPIR leverages a new lightweight technique to remove the hint from high-throughput single-server PIR schemes with small overhead. We also show how to reduce the server preprocessing time in the SimplePIR family of protocols by a factor of 10–15×.

By removing the need for offline communication, YPIR significantly reduces the server-side costs for private auditing of Certificate Transparency logs. Compared to the best previous PIR-based approach, YPIR reduces the server-side costs by a factor of 5.6×. Note that to reduce communication costs, the previous approach assumed that updates to the Certificate Transparency log servers occurred in *weekly* batches. Since there is no offline communication in YPIR, our approach allows clients to always audit the most recent Certificate Transparency logs (e.g., updating once a day). Supporting daily updates using the prior scheme would cost 30× more than YPIR (based on current AWS compute costs).

1 Introduction

A private information retrieval (PIR) [CGKS95, KO97] protocol allows a client to privately retrieve a record from a database without revealing to the database which record was requested. PIR is a useful building block in systems for metadata-hiding messaging [MOT+11, KLDF16, AS16, ACLS18], private database queries and web search [WYG+17, HDCZ23], password breach alerting [LPA+19, TPY+19, ALP+21], Certificate Transparency auditing [HHC+23a], private media consumption [GCM+16], private ad delivery [Jue01, BKMP12, GLM16], and more.

Recently, there has been a flurry of works pushing the limits on the concrete efficiency of *single-server* PIR. Most concretely-efficient PIR constructions rely on an initial *offline phase* where the client either uploads or downloads some information to or from the server:

- **Downloading a hint:** The fastest single-server PIR schemes [DPC22, HHC+23a, ZPSZ24, MSR23] rely on the client first downloading a query-independent "hint" in an offline phase. With a $\sqrt{N}$-size hint (where N is the size of the database) the SimplePIR scheme [HHC+23a] achieves a throughput (i.e., the ratio of the database size and the time needed to answer a query) that is comparable to the memory bandwidth of the system (i.e., the speed at which the PIR server can read the database from memory; this is 14.6 GB/s on our machine). Moreover, if the client can *stream* the entire database in the offline step (and cache $O(\sqrt{N})$ bits), then schemes like [ZPSZ24, MSR23, GZS24] even allow the server to answer queries with *sublinear* online computation; this enables protocols that can easily handle databases with hundreds of GB of data (e.g., in an application to private DNS lookups). Of course, this assumes that clients can perform a streaming download of this size.
- **Uploading client-specific state:** In an alternative model [ACLS18, AYA+21, MCR21, MW22], clients instead upload a "public key" to the server in the offline phase. The public key is typically used to "compress" the query and response. For retrieving large records (e.g., tens of KB long), these protocols currently achieve the best communication. However, the highest server throughput [MW22] achieved by these approaches is much smaller (over 10×) than the throughput of their hint-based counterparts.

Challenges of offline communication. While moving some of the communication to the offline phase has been critical to the concrete efficiency of PIR, it also imposes challenges for practical deployments. In the hint-based approach, the offline download is large: for an 8 GB database such as the one used in the application to signed certificate timestamp (SCT) auditing in Certificate Transparency from [HHC+23a], the size of the hint is over 200 MB using the SimplePIR scheme and 16 MB using the DoublePIR scheme. This is problematic for dynamic databases, since each time the database updates, every client must re-download portions of the hint. When using DoublePIR for private SCT auditing, the protocol of [HHC+23a] compromised by having clients update their hints on a *weekly* basis, even though Certificate Transparency log servers typically update their databases *daily*. Such a scheme sacrifices real-time monitoring for efficiency. Indeed, if the log database updates daily and the client always audits against the most recent version of the database, the hint downloads and updates are more than 90% of the total cost (see Section 4.4). Dynamic databases are common to many other applications of PIR, such as metadata-hiding messaging or private DNS lookups. Moreover, if a client uses PIR to access multiple databases, it would need to cache hints from each database, which imposes storage burdens for the client.

Client-specific state introduces its own share of challenges. The server here needs to store the (large) public key of each client, so these solutions impose high storage requirements for the server and also require additional infrastructure to support efficient client state lookups and retrieval. Reuse of client-specific public keys can also enable active attacks on the application [HHC+23a].

Silent preprocessing. High offline communication costs and large client-side or server-side storage requirements are major bottlenecks in the most concretely-efficient single-server PIR protocols. A natural question is whether we can achieve good concrete efficiency with preprocessing, but *without* offline communication (i.e., a protocol with *silent* preprocessing). In fact, two recent works have already made great strides in this direction: Tiptoe [HDCZ23][1] and HintlessPIR [LMRS23]. Both of these schemes essentially leverage a form of "bootstrapping" [Gen09] to remove the hint from the SimplePIR protocol [HHC+23a], where the server *homomorphically* selects the portion of the SimplePIR hint relevant to the client. While these protocols eliminate the client's need to download the hint, they incur a computation and communication penalty. For example, the throughput of the Tiptoe system on a 32 GB database is over 7× slower than SimplePIR, and the communication cost (for retrieving a 1-bit record) is over 35× greater than the online communication of SimplePIR. HintlessPIR is more lightweight, but has a throughput that is at most 60% of the SimplePIR throughput.

1.1 Our Contributions

In this work, we introduce YPIR, a new single-server PIR protocol with silent preprocessing. Like Tiptoe [HDCZ23] and HintlessPIR [LMRS23], we build on SimplePIR and its recursive variant, DoublePIR. However, instead of using bootstrapping, we take a *packing* approach (which has a conceptually-similar flavor to the response packing techniques from [MW22]) and "pack" the DoublePIR response into a more compact representation using polynomial rings.[2] We provide a technical overview of YPIR in Section 1.2 and the full construction in Section 3 (see also Fig. 1 in Section 3 for a visualization).

High throughput with silent preprocessing. The YPIR protocol can be viewed as appending a lightweight post-processing step to DoublePIR to "compress" the DoublePIR response. When retrieving a single bit from a 32 GB database, YPIR achieves a throughput of 10.9 GB/s, which is 85% of the throughput of SimplePIR (and 75% of the memory bandwidth of the machine).[3] In contrast, the HintlessPIR work only demonstrates a maximum throughput that is 60% of SimplePIR (and concretely, 3.7 GB/s on their test machine) [LMRS23]. The YPIR response size is the same as that in DoublePIR, 9–37× shorter than the response size in SimplePIR, and over 100× shorter than that of HintlessPIR and Tiptoe. On the flip side, YPIR queries are about 2.8–3.8× larger than those in DoublePIR, 2.5–6.6× larger than SimplePIR,

[1]Tiptoe is a system for performing private web queries, but as part of their design, they introduce a hintless variant of SimplePIR. In this work, when we refer to Tiptoe, we refer specifically to their hintless PIR scheme.

[2]The Y in YPIR is to reflect the protocol design combining the high-throughput capabilities of PIR based on integer lattices (i.e., the LWE assumption [Reg05]) with the response compression techniques from PIR based on ideal lattices (i.e., the RLWE assumption [LPR10]).

[3]Here, we compare against our implementation of SimplePIR which is slightly faster than the reference implementation of SimplePIR [HHC+23b]. The reference implementation of SimplePIR achieves a throughput of 10.4 GB/s on our test system, which is actually *slower* than YPIR.

and 2–3× larger than those in HintlessPIR. We refer to Section 4 for a more detailed breakdown and comparison. In short, for retrieving small records from a large database, YPIR achieves 85% of the throughput of one of the fastest single-server PIR schemes while fully eliminating *all* offline communication and only incurring a modest increase in query size.

Faster server preprocessing. While YPIR requires no offline communication, it still relies on an offline server preprocessing step (the same as that in SimplePIR). In Section 4.1, we describe a simple approach to improve the server preprocessing throughput by a factor of 10–15×. For instance, while preprocessing a 32 GB database in SimplePIR requires two hours, it just requires 11 minutes with the YPIR approach. Our technique can be used to reduce the preprocessing costs of any of the protocols in the SimplePIR family.

Cross-client batching. The throughput of protocols like SimplePIR are bounded by the memory bandwidth of the system. Since the server throughput is memory-bounded rather than CPU-bounded, we can achieve higher *effective* throughput by increasing the number of CPU operations per byte of memory read. In Section 4.2, we propose a simple *cross-client batching* approach where the PIR server uses a single scan over the database to answer multiple queries from *non-coordinating* clients.[4] In this work, we show that it is straightforward to tweak SimplePIR (and generalizations like DoublePIR and YPIR) to allow the server to answer a small batch of k queries using a single linear scan through memory. While cross-client batching does *not* reduce the raw number of instructions performed by the CPU, it achieves better utilization of the CPU. With just 4 clients, cross-client batching improves the effective server throughput for a protocol like SimplePIR by a factor of 1.4× to 17 GB/s; applied to YPIR, we achieve an effective throughput of 14 GB/s. In typical applications where servers routinely process queries from multiple clients simultaneously, cross-client batching provides a way to increase the effective throughput for the server and make better use of the available computing resources on the server.

Application to Certificate Transparency. In Section 4.4, we compare the server-side costs of using YPIR to realize an application to private SCT auditing in Certificate Transparency [Lau14, LLK13]. In this setting, a log server holds a set of SCTs and a client (e.g., a web browser) periodically checks that the SCTs it received from web servers are contained in the log. In private SCT auditing, the goal is to perform these audits without requiring clients to reveal their browsing history to the log server. Henzinger et al. [HHC+23a] designed an elegant solution for private SCT auditing by combining PIR with Bloom filters. In their protocol, an SCT audit translates to a single PIR query to the log server. A major challenge in this setting is that Certificate Transparency logs update on a daily basis (with millions of certificates added daily). When built from protocols like DoublePIR, clients will frequently need to download hint updates when performing an audit. To mitigate these communication costs, the work of [HHC+23a] compromise by updating the database on a *weekly* basis. Thus, their approach does not support real-time auditing.

Based on current AWS computation and communication costs, YPIR has 5.7× lower server costs compared to the DoublePIR system that could only support *weekly* updates to the log server (i.e., the cost drops from $1822 per million clients for DoublePIR to $320 per million clients for YPIR). The cost of YPIR further drops to $267 per million clients if we leverage cross-client batching with a batch size of 4 (i.e., assume that the server always has saturated queue of at least 4 queries). Moreover, with YPIR, the client always audits the latest version of the log server. In fact, the *total* communication incurred by YPIR each week is smaller than the total communication of the DoublePIR approach. In other words, YPIR reduces the total communication even after accounting for the fact that the cost of downloading the DoublePIR hint can be amortized over the course of a week. Conversely, if we were to use DoublePIR to support *daily* log updates, the weekly server cost balloons to $10,000 per million clients, which is over 34× higher than using YPIR. Compared to other hintless PIR schemes such as Tiptoe and HintlessPIR, we estimate YPIR achieves a cost savings of 9–60× for private SCT auditing (see Table 6).

Limitations. The main limitation of YPIR is the larger query sizes compared to SimplePIR and DoublePIR. Specifically, a YPIR query is 2–3× larger than a DoublePIR query (for an 8 GB database, YPIR queries are 1.5 MB while DoublePIR queries are 724 KB) and 3–8× larger than a SimplePIR query. This is because the post-processing step in

[4]We contrast this with single-client batching [BIM00, IKOS04, GKL10, ACLS18], which seeks to amortize the cost over multiple queries from a single client). Our cross-client batching applies even if each client makes a *single* query and is entirely transparent to the client (i.e., requires no client-side changes).

YPIR requires communicating a "packing key" as part of the query. Moreover, YPIR is tailored for retrieving databases with small records (e.g., 1-bit to 8-bits), which is suitable for applications like private SCT auditing, but not all PIR applications.

1.2 Overview of YPIR

The starting point for this work is the SimplePIR/DoublePIR schemes from [HHC+23a] based on the learning with errors (LWE) problem [Reg05]. First, an LWE encryption of $\mu \in \mathbb{Z}_p$ is a pair $\mathsf{ct} = (\mathbf{a}, b) \in \mathbb{Z}_q^{n+1}$ where $b = \mathbf{s}^\mathsf{T}\mathbf{a} + e + \Delta \cdot \mu$. Here, n is the lattice dimension, $\mathbf{s} \in \mathbb{Z}_q^n$ is the secret key, $e \in \mathbb{Z}$ is a (small) error term, and Δ is a scaling factor (typically, $\lfloor q/p \rfloor$). Given ct and the secret key $\mathbf{s}$, the user can compute $b - \mathbf{s}^\mathsf{T}\mathbf{a} = \Delta \cdot \mu + e \bmod q$. If e is small relative to the scaling factor Δ (i.e., $|e| < \Delta/2$), the user can recover $\mu \in \mathbb{Z}_p$ from ct by rounding.

In SimplePIR and DoublePIR, the database is represented by a matrix $\mathbf{D} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$ and records are indexed by a row-column pair (i, j). The query consists of LWE encryptions of the components of the indicator vectors $\mathbf{u}_i$ and $\mathbf{u}_j$ (i.e., $\mathbf{u}_i$ is the vector that is 0 everywhere and 1 in index i). In SimplePIR, the response consists of ℓ_2 ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_{\ell_2} \in \mathbb{Z}_q^{n+1}$ which encrypt the ℓ_2 entries of row i of the database. In DoublePIR, the response is an LWE encryption of ct_j, which is itself an encryption of the element in row i, column j of $\mathbf{D}$.

An LWE encryption of an element of $\mathbb{Z}_p$ consists of $(n+1)$ $\mathbb{Z}_q$ elements. Since $\mathsf{ct}_j \in \mathbb{Z}_q^{n+1}$ is a vector over $\mathbb{Z}_q$, an encryption of ct_j (i.e., the DoublePIR response) contains $\kappa(n+1)^2$ elements over $\mathbb{Z}_q$, where $\kappa = \log q / \log p$. The extra factor of κ comes from the fact that the plaintext space for the LWE encryption scheme is $\mathbb{Z}_p$, so to encrypt the components of ct_j over $\mathbb{Z}_q$, DoublePIR first decomposes each $\mathbb{Z}_q$ element into its base-p representation (consisting of κ digits in $\mathbb{Z}_p$). For security, the lattice dimension n is around $2^{10} = 1024$, so the response is very large. The insight in [HHC+23a] is that most of the components in the response only depend on the database and *not* the query. Thus, these can be prefeteched as a hint in the offline phase. For example, for an 8 GB database with 2^{36} 1-bit records, the query-independent portion of the response is 16 MB while the query-dependent portion is just 32 KB.

Packing the DoublePIR responses. The YPIR protocol eliminates the offline hint from DoublePIR by *compressing* the *full* DoublePIR response using ring LWE [LPR10]. Specifically, we work over the polynomial ring $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power-of-two. RLWE ciphertexts have the advantage of having a much smaller ciphertext expansion factor. With vanilla LWE, encoding a single value $\mu \in \mathbb{Z}_p$ requires a vector of $(n+1)$ elements over $\mathbb{Z}_q$ whereas encoding a *ring* element $\mu \in R_p$ only requires two elements in R_q (where $R_q := R/qR$). If we consider the ciphertext expansion factor (i.e., the ratio of the ciphertext size to the plaintext size), RLWE decreases the expansion factor from $(n+1)\log q/\log p$ to $2\log q/\log p$. For concrete values of $n \approx 2^{10}$, this is a 1000× reduction in ciphertext expansion factor.

In YPIR, we use the LWE-to-RLWE packing technique from [CDKS21]. This transformation takes a collection of d LWE ciphertexts $\mathsf{ct}_1, \ldots, \mathsf{ct}_d \in \mathbb{Z}_q^{d+1}$ that encode messages $\mu_1, \ldots, \mu_d \in \mathbb{Z}_p$ (under a secret key $\mathbf{s}$) and packs them into an RLWE ciphertext that encrypts the polynomial $f(x) := \sum_{i \in [d]} \mu_i x^{i-1} \in R_p$ (under a key $s \in R_q$ derived from $\mathbf{s}$). Note that we assume the lattice dimension n in LWE coincides with the ring dimension d in RLWE. Critically, the transformation takes $d(d+1)$ elements over $\mathbb{Z}_q$ and compresses them into just $2d$ elements over $\mathbb{Z}_q$. This yields a factor $(d+1)/2$ reduction in ciphertext size.[5] For an 8 GB database, this packing approach compresses the full 16 MB DoublePIR response into a 12 KB response (see Table 2). The cost is the query now needs to include a "packing key" for the transformation from [CDKS21] (which essentially consists of RLWE key-switching matrices). This increases the query size from 724 KB in DoublePIR by a factor of 2× to 1.5 MB.

Faster preprocessing. YPIR relies on the same preprocessing as SimplePIR (and DoublePIR). The main cost of this preprocessing is computing a product of the form $\mathbf{AD}$ where $\mathbf{A} \in \mathbb{Z}_q^{n \times \ell_1}$ is a (random) matrix and $\mathbf{D} \in \mathbb{Z}_p^{\ell_1 \times \ell_2}$ is the database. While this process only needs to be performed once, it is a very expensive process for large databases: on a single core, this precomputation has a throughput of under 4 MB/s; for a 32 GB database, the SimplePIR preprocessing

[5]It is also possible to pack LWE encodings (e.g., using the Spiral approach for response compression [MW22]) into a packed LWE ciphertext [PVW08], but this requires $O(d)$ key-switching matrices. Since these key-switching matrices must now be communicated with the query, this does not help reduce communication. The LWE-to-RLWE transformation only requires $O(\log d)$ key-switching matrices, which can be included as part of the query with only modest communication overhead.

takes over two hours. In this work, we observe that we can replace $\mathbf{A}$ with a *structured* matrix and use number-theoretic transforms (NTTs) to compute the matrix-vector product. Asymptotically, this yields a $n/\log n$ improvement to preprocessing, and concretely, we observe a 10–15× increase in the throughput. The only cost of this is that security of the scheme now rests on the ring LWE assumption rather than the LWE assumption. Note that this optimization *only* changes the preprocessing and *not* the online server computation. In particular, the online server computation is still over $\mathbb{Z}_q$ (and *not* over a polynomial ring). We describe our approach in more detail in Section 4.1. We also stress that our approach is *not* just lifting SimplePIR to work over polynomial rings. While this works in theory, the performance bottleneck in practice is the *memory bandwidth* of the system. As we discuss in Remark 4.1, a ring-based SimplePIR has higher memory requirements, which is enough to reduce throughput from 11.5 GB/s to just 3.2 GB/s.

2 Preliminaries

We write λ for the security parameter. For a positive integer $n \in \mathbb{N}$, we write $[n]$ for the set $\{1, \ldots, n\}$. For integers $a, b \in \mathbb{Z}$, we write $[a, b]$ for the set $\{a, a+1, \ldots, b\}$. For a positive integer $q \in \mathbb{N}$, we write $\mathbb{Z}_q$ to denote the integers modulo q. We use bold uppercase letters to denote matrices (e.g., $\mathbf{A}, \mathbf{B}$) and bold lowercase letters to denote vectors (e.g., $\mathbf{u}$, $\mathbf{v}$). For a matrix $\mathbf{A}$, we write $\mathbf{A}^{\mathsf{T}}$ to denote its transpose. When $\mathbf{A} \in \mathbb{Z}_q^{n \times m}$ and $\mathbf{B} \in \mathbb{Z}^{m \times k}$, we write $\mathbf{AB} \in \mathbb{Z}_q^{n \times k}$ to denote the matrix product over $\mathbb{Z}_q$ where $\mathbf{B}$ is first lifted into $\mathbb{Z}_q^{m \times k}$ by associating each of its entries $B_{i,j} \in \mathbb{Z}$ with its unique mod-q representative in the range $(-q/2, q/2]$. We define $\mathbf{AB}$ where $\mathbf{A} \in \mathbb{Z}^{n \times m}$ and $\mathbf{B} \in \mathbb{Z}_q^{m \times k}$ analogously.

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 $c \in \mathbb{N}$. We say 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 except with non-negligible probability.

Rounding. For an input $x \in \mathbb{R}$, we write $\lfloor x \rceil$ to denote the rounding function; that is $\lfloor x \rceil$ outputs the nearest integer to x (rounding up in case of ties). For integers $q > p$, we write $\lfloor \cdot \rceil_{q,p} \colon \mathbb{Z}_q \to \mathbb{Z}_p$ to denote the rounding function that first takes the input $x \in \mathbb{Z}_q$, lifts it to an integer in the interval $x' \in (-q/2, q/2]$, and outputs $\lfloor p/q \cdot x' \rceil$ as an element of $\mathbb{Z}_p$. Here, the division and the rounding are performed over the *rationals*. We extend $\lfloor \cdot \rceil_{q,p}$ to operate component-wise on vector-valued and matrix-valued inputs.

Discrete Gaussians and tail bounds. We recall some basic facts about the discrete Gaussian distribution, and refer to [Pei16] for more details and references. The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ over $\mathbb{Z}$ with mean 0 and width σ is the distribution with probability mass function

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

where $\rho_\sigma(x) := \exp(-\pi x^2/\sigma^2)$ is the Gaussian function with width σ. We say a real-valued random variable X is subgaussian with parameter σ if for every $t \geq 0$, $\Pr[|X| > t] \leq 2\exp(-\pi t^2/\sigma^2)$. The discrete Gaussian distribution $D_{\mathbb{Z},\sigma}$ is subgaussian with parameter σ. Moreover, the following properties hold for subgaussian random variables:

- If X is subgaussian with parameter σ, then for all $c \in \mathbb{R}$, cX is subgaussian with parameter $|c|\,\sigma$.
- If $X_1, \ldots, X_k$ are *independent* subgaussian random variables with parameters $\sigma_1, \ldots, \sigma_k$, respectively, then their sum $\sum_{i \in [k]} X_i$ is also subgaussian with parameter $\left(\sum_{i \in [k]} \sigma_i^2\right)^{1/2}$.

Polynomial rings. Our construction will use the cyclotomic ring $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. For a positive integer $q \in \mathbb{N}$, we write $R_q := R/qR$. We now define the Coeffs and $\mathsf{NCyclicMat}$ functions over R (and by extension, R_q). Let $g = \sum_{i=0}^{d-1} \alpha_i x^i \in R$ be a ring element.

- Let $\mathsf{Coeffs} \colon R \to \mathbb{Z}^d$ be the mapping $g \mapsto [\alpha_0, \ldots, \alpha_{d-1}]^{\mathsf{T}}$ that outputs the vector of coefficients of g.

- Let $\mathsf{NCyclicMat}\colon R \to \mathbb{Z}^{d\times d}$ be the linear transformation over $\mathbb{Z}^d$ associated with multiplication by $g \in R$. Namely, for all $f \in R$, it holds that $\mathsf{Coeffs}(f)^\intercal \cdot \mathsf{NCyclicMat}(g) = \mathsf{Coeffs}(fg)^\intercal$. Specifically,

$$\mathsf{NCyclicMat}(g) := \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_{d-1} \\ -\alpha_{d-1} & \alpha_0 & \alpha_1 & \cdots & \alpha_{d-2} \\ -\alpha_{d-2} & -\alpha_{d-1} & \alpha_0 & \cdots & \alpha_{d-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\alpha_1 & -\alpha_2 & -\alpha_3 & \cdots & \alpha_0 \end{bmatrix} \in \mathbb{Z}^{d\times d}$$

We extend $\mathsf{NCyclicMat}$ to operate on vectors in a component-wise manner. In particular, this means that for all $f \in R$ and $\mathbf{g} = (g_1, \ldots, g_m) \in R^m$,

$$\mathsf{Coeffs}(f \cdot \mathbf{g})^\intercal = [\mathsf{Coeffs}(fg_1)^\intercal \mid \cdots \mid \mathsf{Coeffs}(fg_m)^\intercal] = \mathsf{Coeffs}(f)^\intercal \cdot \mathsf{NCyclicMat}(\mathbf{g}^\intercal). \tag{2.2}$$

We define both operators over R_q in the identical manner. When $q = 1 \bmod 2d$, we say that q is "NTT-friendly;" in this case, polynomial multiplication in R_q can be implemented using a negacyclic convolution [LMPR08, LN16], which can in turn be computed using fast radix-2 number-theoretic transforms (NTTs). For $f \in R$, we write $\|f\|_\infty$ to denote the ℓ_∞ norm of the vector of coefficients $\mathsf{Coeffs}(f)$. For all polynomials $f, g \in R$, it holds that $\|fg\|_\infty \le d\|f\|_\infty \|g\|_\infty$. Finally, we say that $f \in R$ is sampled from a subgaussian distribution with parameter σ if each coefficient of f is independently distributed according to a subgaussian distribution σ. Our analysis will only rely on the following basic lemma from [MW22] on bounding the norm of the coefficients of a product of two polynomials over R:

Lemma 2.1 (Polynomials with Subgaussian Coefficients [MW22, Lemma 2.6, adapted])**.** *Let $R = \mathbb{Z}[x]/(x^d + 1)$ and $g = \sum_{i\in[0,d-1]} g_i x^i \in R$ be any polynomial where $\|g\|_\infty \le B$. Let $f = \sum_{i\in[0,d-1]} f_i x^i$ where each f_i is independently sampled from a subgaussian distribution with parameter σ and let $h = fg = \sum_{i\in[0,d-1]} h_i x^i \in R$. Then, the distribution of each h_i is subgaussian with parameter $\sqrt{d}B\sigma$.*

Gadget matrices. Next, we recall the notion of the gadget matrix from [MP12]. For a modulus $q \in \mathbb{N}$ and a decomposition base $z \in \mathbb{N}$, we write $\mathbf{g}_z = [1, z, z^2, \ldots, z^{t-1}] \in \mathbb{Z}_q^t$ where $t = \lceil \log q / \log t \rceil$. For a dimension $n \in \mathbb{N}$, we define $\mathbf{G}_{n,z} := \mathbf{I}_n \otimes \mathbf{g}_z^\intercal \in \mathbb{Z}_q^{n\times nt}$ to be the gadget matrix. We write $\mathbf{G}_{n,z}^{-1}\colon \mathbb{Z}_q^n \to \mathbb{Z}^{nt}$ to denote the base-z digit decomposition operator that expands each component of the input vector into its base-z representation (where each output component is an integer between $-z/2$ and $z/2$). We write $\mathbf{g}_z^{-1}\colon \mathbb{Z}_q \to \mathbb{Z}^t$ for the 1-dimensional operator $\mathbf{G}_{1,z}^{-1}$. We extend $\mathbf{G}_{n,z}^{-1}$ to operate on matrices $\mathbf{M} \in \mathbb{Z}_q^{n\times k}$ by independently applying $\mathbf{G}_{n,z}^{-1}$ to each column of $\mathbf{M}$. Both the gadget matrix $\mathbf{G}_{n,z}$ and its digit decomposition $\mathbf{G}_{n,z}^{-1}$ are defined identically over the ring R_q.

Ring learning with errors. Like many lattice-based PIR schemes [MBFK16, ACLS18, GH19, MCR21, MW22, LMRS23], the security of our protocol relies on the ring learning with errors (RLWE) problem [Reg05, LPR10]. We state the "normal form" of the assumption where the RLWE secret is sampled from the error distribution; this version reduces to the one where the secret key is uniform [ACPS09].

Definition 2.2 (Ring Learning with Errors [LPR10])**.** Let λ be a security parameter, $d = d(\lambda)$ be a power-of-two, and $R = \mathbb{Z}[x]/(x^d + 1)$. Let $m = m(\lambda)$ be the number of samples, $q = q(\lambda)$ be a modulus, and $\chi = \chi(\lambda)$ be an error distribution over R. The ring learning with errors (RLWE) assumption $\mathsf{RLWE}_{d,m,q,\chi}$ in Hermite normal form states that for $\mathbf{a} \xleftarrow{\mathrm{R}} R_q^m$, $s \leftarrow \chi$, $\mathbf{e} \leftarrow \chi^m$, and $\mathbf{v} \xleftarrow{\mathrm{R}} \mathbb{R}_q^m$, the following two distributions are computationally indistinguishable:

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

LWE and RLWE encodings. We say that a vector $\mathbf{c} \in \mathbb{Z}_q^{n+1}$ is an "LWE encoding" of a value $\mu \in \mathbb{Z}_q$ with respect to a secret key $\mathbf{s} \in \mathbb{Z}_q^n$ and error $e \in \mathbb{Z}$ if $[-\mathbf{s}^\intercal \mid 1] \cdot \mathbf{c} = \mu + e$. For a ring $R = \mathbb{Z}[x]/(x^d + 1)$, we say that $\mathbf{c} \in R_q^2$ is an "RLWE encoding" of a value $\mu \in R_q$ with respect to a secret key $s \in R_q$ and error $e \in R$ if $[-s \mid 1] \cdot \mathbf{c} = \mu + e$. In our setting, it will typically be the case that $\mu = \lfloor q/p \rfloor v$ for some $v \in \mathbb{Z}_p$ (or $v \in R_p$). Given $\mu + e$ for sufficiently small e, it is then possible to recover the value of v by rounding. We state this in the following lemma from [MW22]:

Lemma 2.3 (Message Decoding [MW22, Theorem 2.11])**.** *Let $R = \mathbb{Z}[x]/(x^d + 1)$. Suppose $z = \lfloor q/p \rfloor v + e \in R$ where $\|v\|_\infty < p$ and $\|e\|_\infty < \frac{q}{2p} - (q \bmod p)$. Then, $\lfloor z \rceil_{q,p} = v$.*

Independence heuristic. Like many lattice-based PIR constructions [ACLS18, GH19, MCR21, MW22, LMRS23] and other systems based on homomorphic encryption [GHS12b, CGGI18, CCR19], we use the independence heuristic that models the error terms arising in intermediate homomorphic computations to be independent. Specifically, instead of bounding the worst-case magnitude on the noise, we bound the *variance* of the noise vector (i.e., σ^2 where σ is the subgaussian width parameter associated with the noise distribution). Since the variance is additive for *independent* subgaussian variables, bounding the variance often yields a square-root improvement in the noise bound compared to a worst-case bound.

Packing LWE encodings. Observe that RLWE encodings have better *rate:* namely, an encoding of $\mu \in R_q$ consists of just two elements of R_q, whereas an LWE encoding of $\mu \in \mathbb{Z}_q$ requires a vector of $(n+1)$ elements, where n is the lattice dimension (i.e., the security parameter). Chen, Dai, Kim, and Song [CDKS21] described a general transformation to "pack" multiple LWE encodings into a single RLWE encoding. This is the main technique we use to remove the hint from DoublePIR. We state the transformation here, and for completeness, include a description of their algorithm in Appendix A (Construction A.3). In Appendix A.1, we also provide a proof of the additional pseudorandomness property. The security proof relies on a standard "circular security" assumption on RLWE encodings.

Theorem 2.4 (LWE-to-RLWE Packing [CDKS21])**.** *Let λ be a security parameter, $R = \mathbb{Z}[x]/(x^d+1)$ where $d = d(\lambda)$ is a power of two, $q = q(\lambda)$ be an encoding modulus, and $\chi = \chi(\lambda)$ be an error distribution. There exist a pair of efficient algorithms* (CDKS.Setup, CDKS.Pack) *with the following properties:*

- $\mathsf{CDKS.Setup}(1^\lambda, s, z) \to \mathsf{pk}$*: On input the security parameter λ, a secret key $s \in R_q$, and a decomposition base $z \in \mathbb{N}$, the setup algorithm outputs a packing key* pk.
- $\mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C}) \to \mathbf{c}'$*: On input the packing key* pk *and a collection of LWE encodings $\mathbf{C} \in \mathbb{Z}_q^{(d+1)\times d}$, the packing algorithm outputs an RLWE encoding $\mathbf{c}' \in R_q^2$.*

Moreover, these algorithms satisfy the following properties:

- **Correctness:** *Take any security parameter $\lambda \in \mathbb{N}$, any secret key $s \in R_q$, any matrix $\mathbf{C} = [\mathbf{c}_1 \mid \cdots \mid \mathbf{c}_d] \in \mathbb{Z}_q^{(d+1)\times d}$ where $\mathbf{c}_i \in \mathbb{Z}_q^{d+1}$, and any decomposition base $z \le q$. Let $\mathbf{s} = \mathsf{Coeffs}(s)$. Suppose $[-\mathbf{s}^\intercal \mid 1] \cdot \mathbf{c}_i = v_i \in \mathbb{Z}_q$. Suppose $\mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s, z)$ and $\mathbf{c}' \leftarrow \mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C})$. Under the independence heuristic,*

$$[-s \mid 1] \cdot \mathbf{c}' = d \sum_{i \in [d]} v_i x^{i-1} + e' \in R_q,$$

  where e' is subgaussian with parameter σ' and $(\sigma')^2 \le \frac{1}{3}(d^2-1)\big(tdz^2\sigma_\chi^2/4\big)$, where $t = \lfloor \log_z q \rfloor + 1$.

- **Pseudorandomness given the public key:** *Let $m = m(\lambda)$ be the number of samples. For a bit $b \in \{0,1\}$, a decomposition base $z \in \mathbb{N}$, and an adversary $\mathcal{A}$, let*

$$W_b := \Pr\left[\mathcal{A}(1^\lambda, \mathsf{pk}, \mathbf{a}, \mathbf{t}_b) = 1 : \begin{array}{c} s \leftarrow \chi, \mathbf{a} \xleftarrow{R} R_q^m, \mathbf{e} \leftarrow \chi^m \\ \mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s) \\ \mathbf{t}_0 = s\mathbf{a} + \mathbf{e}, \mathbf{t}_1 \xleftarrow{R} R_q^m \end{array}\right].$$

  We say the construction satisfies pseudorandomness (for m samples) given the public key if for all $z \le q$ and all efficient adversaries $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ *such that for all $\lambda \in \mathbb{N}$, $|W_0 - W_1| = \mathsf{negl}(\lambda)$.*

Modulus switching. A standard technique to reduce the size of lattice-based encodings after performing homomorphic operations on them is to use *modulus switching* [BV11, BGV12]. Modulus switching takes an (R)LWE encoding mod q and rescales it to an encoding mod q_1 where $q_1 < q$. This reduces the size of the encoding. Here, we describe a more fine-grained variant from [MW22] where two different moduli are used. We describe the approach for encodings over any ring $R = \mathbb{Z}[x]/(x^d+1)$; the case where $d = 1$ corresponds to the case of the integers.

- $\mathsf{ModReduce}_{q_1,q_2}(\mathbf{c})$: For integers $q > q_1 \ge q_2$ and on input an encoding $\mathbf{c} \in R_q^{n+1}$ where $\mathbf{c} = \left[\begin{smallmatrix} \mathbf{c}_1 \\ c_2 \end{smallmatrix}\right]$ for $\mathbf{c}_1 \in R_q^n$ and $c_1 \in R_q$, output $(\lfloor \mathbf{c}_1 \rceil_{q,q_1}, \lfloor c_2 \rceil_{q,q_2}) \in R_{q_1}^n \times R_{q_2}$.

When there is a single modulus q_1, we write $\mathsf{ModReduce}_{q_1}(\mathbf{c})$ to denote $\mathsf{ModReduce}_{q_1,q_1}(\mathbf{c})$. We extend $\mathsf{ModReduce}_{q_1,q_2}$ to matrices by column-wise evaluation. We now state the main correctness guarantee from [MW22].

Lemma 2.5 (Modulus Switching [MW22, Theorem 3.4, adapted])**.** *Let $q > q_1 \geq q_2 > p$ and let $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. Suppose $[-\mathbf{s}^{\mathsf{T}} \mid 1]\mathbf{c} = \lfloor q/p \rfloor \mu + e \bmod q$ for some $\mathbf{s} \in R_q^n$, $\mathbf{c} \in R_q^{n+1}$, $\|\mu\|_\infty \leq p/2$ and $e \in R$. Suppose the components of $\mathbf{s}$ are independent subgaussian random variables with parameter σ_s and e is subgaussian with parameter σ_e. Let $(\mathbf{c}_1', c_2') = \mathsf{ModReduce}_{q_1,q_2}(\mathbf{c})$. Then $\lfloor -\mathbf{s}^{\mathsf{T}}\mathbf{c}_1' \rceil_{q_1,q_2} + c_2' = \lfloor q_2/p \rfloor \mu + e' \bmod q_2$ where $e' = e_1' + e_2'$,*

$$\|e_1'\|_\infty \leq \frac{1}{2}\left(2 + (q_2 \bmod p) + \frac{q_2}{q}(q \bmod p)\right),$$

and the components of e_2' are subgaussian with parameter $\sigma' = \sqrt{(q_2/q_1)^2 n d \sigma_s^2/4 + (q_2/q)^2 \sigma_e^2}$.

Private information retrieval. We now recall the formal definition of a (two-message) single-server PIR protocol [KO97]. We work in the model where there is an initial database-dependent preprocessing algorithm that outputs a set of public parameters (assumed to be known to the client and to the server) and an internal server state.

Definition 2.6 (Private Information Retrieval [KO97, adapted])**.** Let $N \in \mathbb{N}$ be an integer. A (two-message) single-server private information retrieval (PIR) scheme $\Pi_{\mathsf{PIR}} = (\mathsf{DBSetup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$ with message space $\mathbb{Z}_N$ is a tuple of efficient algorithms with the following properties:

- $\mathsf{DBSetup}(1^\lambda, \mathbf{D}) \to (\mathsf{pp}, \mathsf{dbp})$: On input the security parameter λ and a database $\mathbf{D}$, the setup algorithm outputs a set of public parameters pp and database parameters dbp.
- $\mathsf{Query}(\mathsf{pp}, \mathsf{idx}) \to (\mathsf{q}, \mathsf{qk})$: On input the public parameters pp and an index idx, the query algorithm outputs a query q and a query key qk.
- $\mathsf{Answer}(\mathsf{dbp}, \mathsf{q}) \to \mathsf{resp}$: On input the database parameters dbp, a query q, the answer algorithm outputs a response resp.
- $\mathsf{Extract}(\mathsf{qk}, \mathsf{resp}) \to D_i$: On input the client state qk and a response resp, the extract algorithm outputs a database record $D_i \in \mathbb{Z}_N$.

The algorithms should satisfy the following properties:

- **Correctness:** For all $\lambda \in \mathbb{N}$, all databases $\mathbf{D}$, and all indices idx, if we sample $(\mathsf{pp}, \mathsf{dbp}) \leftarrow \mathsf{DBSetup}(1^\lambda, \mathbf{D})$, $(\mathsf{q}, \mathsf{qk}) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{idx})$, and $\mathsf{resp} \leftarrow \mathsf{Answer}(\mathsf{dbp}, \mathsf{q})$, then

  $$\Pr\left[\mathsf{Extract}(\mathsf{qk}, \mathsf{resp}) = \mathbf{D}[\mathsf{idx}]\right] \geq 1 - \delta,$$

  where $\mathbf{D}[\mathsf{idx}]$ denotes the element of $\mathbf{D}$ indexed by idx, and where the probability is taken over the randomness of $\mathsf{DBSetup}$, Query, Answer, and $\mathsf{Extract}$. We refer to δ as the correctness error. When $\delta = 0$, we say the scheme satisfies perfect correctness.

- **Query privacy:** For a bit $b \in \{0,1\}$, we define the query privacy game between an adversary $\mathcal{A}$ and a challenger as follows:

  - On input the security parameter 1^λ, the adversary outputs a database $\mathbf{D}$.
  - The challenger computes $(\mathsf{pp}, \mathsf{dbp}) \leftarrow \mathsf{DBSetup}(1^\lambda, \mathbf{D})$ and gives pp to $\mathcal{A}$.
  - Algorithm $\mathcal{A}$ now outputs a pair of indices $\mathsf{idx}_0, \mathsf{idx}_1$. The challenger computes $(\mathsf{q}, \mathsf{qk}) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{idx}_b)$ and replies with q.
  - Algorithm $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.

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

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

3 The YPIR Protocol

In this section, we describe the YPIR protocol (we refer to Fig. 1 for a visual description). As described in Section 1.2, the YPIR protocol first invokes DoublePIR [HHC+23a] over the database, and then packs the DoublePIR response (a collection of LWE encodings) into a small number of RLWE encodings.

Construction 3.1 (YPIR Protocol)**.** Let λ be a security parameter. We model the database as a matrix $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$. In the scheme, we associate the records in $\mathbf{D}$ with its integer representative in the interval $(-N/2, N/2]$. We index records by a row-column pair $(i_1, i_2) \in [\ell_1] \times [\ell_2]$. YPIR uses different sets of lattice parameters for the initial pass (i.e., the linear scan over the database—"SimplePIR") and for the second pass (i.e., recursing on the output of the first step—"DoublePIR"). This is because the parameters for the second pass must be compatible with the LWE-to-RLWE packing transformation. We define the parameters below:

- Let $d_1 = d_1(\lambda), d_2 = d_2(\lambda)$ be ring dimensions, where each is a power of two. We write $R_{d_1} := \mathbb{Z}[x]/(x^{d_1} + 1)$ and $R_{d_2} := \mathbb{Z}[x]/(x^{d_2} + 1)$. For $j \in \{1, 2\}$ and a modulus q, we write $R_{d_j,q} := R_{d_j}/qR_{d_j}$.
- Let $q_1 = q_1(\lambda), q_2 = q_2(\lambda)$ be the encoding modulus and $\tilde{q}_1 = \tilde{q}_1(\lambda)$, $\tilde{q}_{2,1} = \tilde{q}_{2,1}(\lambda)$, and $\tilde{q}_{2,2} = \tilde{q}_{2,2}(\lambda)$ be a set of reduced modulus (for modulus switching). We require that $\gcd(d_2, q_2) = 1$.
- Let $\chi_1 = \chi_1(\lambda), \chi_2 = \chi_2(\lambda)$ be error distributions over R_{d_1} and R_{d_2}, respectively.
- Let $z = z(\lambda)$ be a decomposition parameter (for the LWE-to-RLWE packing).
- Let $p = p(\lambda)$ be an intermediate modulus and let $\kappa = \lceil \log \tilde{q}_1 / \log p \rceil$.
- Let $(\mathsf{CDKS.Setup}, \mathsf{CDKS.Pack})$ be the LWE-to-RLWE packing algorithms from Theorem 2.4 defined over R_{d_2,q_2} with error distribution χ_2.

The YPIR = $(\mathsf{DBSetup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$ scheme is defined as follows:

- $\mathsf{DBSetup}(1^\lambda, \mathbf{D})$: On input the security parameter λ and a database $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$, where $\ell_1 = m_1 d_1$ and $\ell_2 = m_2 d_2$ for integers $m_1, m_2 \in \mathbb{N}$,[6] the setup algorithm samples $\mathbf{a}_j \xleftarrow{\mathrm{R}} R_{d_j,q_j}^{m_j}$ and sets $\mathbf{A}_j = \mathsf{NCyclicMat}(\mathbf{a}_j^{\mathsf{T}}) \in \mathbb{Z}_{q_j}^{d_j \times \ell_j}$ where $j \in \{1, 2\}$. Finally, the setup algorithm computes

$$\mathbf{H}_1 = \mathbf{G}_{d_1,p}^{-1}(\lfloor \mathbf{A}_1 \mathbf{D} \rceil_{q_1,\tilde{q}_1}) \in \mathbb{Z}^{\kappa d_1 \times \ell_2} \quad \text{and} \quad \mathbf{H}_2 = \mathbf{A}_2 \cdot \mathbf{H}_1^{\mathsf{T}} \in \mathbb{Z}_{q_2}^{d_2 \times \kappa d_1}. \tag{3.1}$$

  The setup algorithm then outputs the public parameters $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$ together with the server state $\mathsf{dbp} = (1^\lambda, \mathbf{D}, \mathbf{H}_1, \mathbf{H}_2)$.

- $\mathsf{Query}(\mathsf{pp}, \mathsf{idx})$: On input the public parameters $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$ and an index $\mathsf{idx} = (i_1, i_2) \in [\ell_1] \times [\ell_2]$, the query algorithm proceeds as follows:

  1. **Key generation:** Sample two secret keys $s_1 \leftarrow \chi_1$ and $s_2 \leftarrow \chi_2$. Compute the packing key $\mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s_2, z)$.
  2. **Query encoding:** Define the scaling factors $\Delta_1 = \lfloor q_1/N \rfloor$ and $\Delta_2 = \lfloor q_2/p \rfloor$. The query encodings are then constructed as follows:
     (a) For $j \in \{1, 2\}$, let $m_j = \ell_j/d_j$ and $i_j = \alpha_j d_j + \beta_j$ where $\alpha_j \in [m_j]$ and $\beta_j \in [d_j]$. Let $\boldsymbol{\mu}_j = x^{\beta_j} \mathbf{u}_{\alpha_j} \in R_{d_j,q_j}^{m_j}$, where $\mathbf{u}_j$ denotes the j^{th} elementary basis vector (of the appropriate dimension).
     (b) For $j \in \{1, 2\}$, sample $\mathbf{e}_j \leftarrow \chi_j^{m_j}$ and construct the encoding $\mathbf{c}_j = \mathsf{Coeffs}(s_j \mathbf{a}_j + \mathbf{e}_j + \Delta_j \boldsymbol{\mu}_j) \in \mathbb{Z}_{q_j}^{\ell_j}$.

  Output the query $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$ and the query key $\mathsf{qk} = (s_1, s_2)$.

[6]For ease of exposition, we describe our construction for the setting where the database dimensions are a multiple of the ring dimensions d_1 and d_2. This can be ensured by padding the database with dummy rows and columns. It is straightforward to extend the scheme to support arbitrary dimensions *without* padding, but this introduces additional notational burden. We defer the description of the modified scheme to Remark 3.3.

- Answer(dbp, q): On input the database parameters dbp $= (1^\lambda, \mathbf{D}, \mathbf{H}_1, \mathbf{H}_2)$ and the query q $= (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$, where $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$, $\mathbf{H}_1 \in \mathbb{Z}^{\kappa d_1 \times \ell_2}$, $\mathbf{H}_2 \in \mathbb{Z}_{q_2}^{d_2 \times \kappa d_1}$, $\mathbf{c}_1 \in \mathbb{Z}_{q_1}^{\ell_1}$, and $\mathbf{c}_2 \in \mathbb{Z}_{q_2}^{\ell_2}$, the answer algorithm proceeds as follows:
  1. **Compute the SimplePIR response:** Let $\mathbf{T} = \mathbf{g}_p^{-1}(\lfloor \mathbf{c}_1^\mathsf{T} \mathbf{D} \rceil_{q_1, \tilde{q}_1}) \in \mathbb{Z}^{\kappa \times \ell_2}$.
  2. **Compute the DoublePIR response:** Compute
  $$\mathbf{C} = (d_2^{-1} \bmod q_2) \cdot \begin{bmatrix} \mathbf{H}_2 & \mathbf{A}_2 \mathbf{T}^\mathsf{T} \\ \mathbf{c}_2^\mathsf{T} \mathbf{H}_1^\mathsf{T} & \mathbf{c}_2^\mathsf{T} \mathbf{T}^\mathsf{T} \end{bmatrix} \in \mathbb{Z}_{q_2}^{(d_2+1) \times \kappa(d_1+1)}. \tag{3.2}$$
  3. **Pack encodings:** Let $\rho = \lceil \kappa(d_1+1)/d_2 \rceil$ and parse $[\mathbf{C} \mid \mathbf{0}^{(d_2+1) \times (d_2 \rho - \kappa(d_1+1))}] = [\mathbf{C}_1 \mid \cdots \mid \mathbf{C}_\rho]$ where each $\mathbf{C}_i \in \mathbb{Z}_{q_2}^{(d_2+1) \times d_2}$. Namely, $\mathbf{C}_1, \ldots, \mathbf{C}_\rho$ are the blocks of $\mathbf{C}$ and $\mathbf{C}_\rho$ is padded to the required dimension with columns of all-zeroes. Then, for each $i \in [\rho]$, compute $\tilde{\mathbf{c}}_i \leftarrow \mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C}_i) \in R_{d_2, q_2}^2$.
  4. **Apply (split) modulus switching:** For each $i \in [\rho]$, let $(c_{i,1}, c_{i,2}) = \mathsf{ModReduce}_{\tilde{q}_{2,1}, \tilde{q}_{2,2}}(\tilde{\mathbf{c}}_i)$.

  Output resp $= ((c_{1,1}, c_{1,2}), \ldots, (c_{\rho,1}, c_{\rho,2}))$.

- Extract(qk, resp): On input the client state qk $= (s_1, s_2)$ and the response resp $= ((c_{1,1}, c_{1,2}), \ldots, (c_{\rho,1}, c_{\rho,2}))$ where $c_{i,1} \in R_{d_2, \tilde{q}_{2,1}}$ and $c_{i,2} \in R_{d_2, \tilde{q}_{2,2}}$, the extract algorithm computes $v_i' = \lfloor -s_2 c_{i,1} \rceil_{\tilde{q}_{2,1}, \tilde{q}_{2,2}} + c_{i,2} \in R_{d_2, \tilde{q}_{2,2}}$ and $v_i = \lfloor v_i' \rceil_{\tilde{q}_{2,2}, p} \in R_{d_2, p}$ for each $i \in [\rho]$. Let
  $$\bar{\mathbf{w}} = \begin{bmatrix} \mathsf{Coeffs}(v_1) \\ \vdots \\ \mathsf{Coeffs}(v_\rho) \end{bmatrix} \in \mathbb{Z}_p^{d_2 \rho}.$$
  Parse $\bar{\mathbf{w}} = \left[\begin{smallmatrix} \mathbf{w} \\ \mathbf{w}' \end{smallmatrix}\right]$ where $\mathbf{w} \in \mathbb{Z}_p^{\kappa(d_1+1)}$ and $\mathbf{w}' \in \mathbb{Z}_p^{d_2 \rho - \kappa(d_1+1)}$. Compute
  $$\mathbf{c}' = \begin{bmatrix} \mathbf{G}_{d_1, p} & \mathbf{0}^{d_1 \times \kappa} \\ \mathbf{0}^{1 \times \kappa d_1} & \mathbf{g}_p^\mathsf{T} \end{bmatrix} \mathbf{w} = \mathbf{G}_{d_1+1, p} \mathbf{w} \in \mathbb{Z}_{\tilde{q}_1}^{d_1+1}$$
  and the scaled message $\mu' = [-\mathsf{Coeffs}(s_1) \mid 1] \cdot \mathbf{c}' \in \mathbb{Z}_{\tilde{q}_1}$. Compute $\mu = \lfloor \mu' \rceil_{\tilde{q}_1, N} \in \mathbb{Z}_N$ and output the representative of μ in $\mathbb{Z}_N$.

Remark 3.2 (Silent Preprocessing)**.** As described, the public parameters pp in Construction 3.1 are very long (specifically, the vectors and $\mathbf{a}_1$ and $\mathbf{a}_2$). However, the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$ are *uniformly random*, and could be derived from a random oracle. This is a standard technique used in lattice-based PIR [MCR21, MW22, HHC+23a, DPC22]. With this modification, the public parameters in Construction 3.1 only consist of the *meta-parameters* for the database itself (i.e., the database dimensions and the record size). Thus, we say YPIR supports *silent* preprocessing.

Remark 3.3 (Supporting Arbitrary Dimension)**.** To simplify the description, Construction 3.1 assumes that the database dimensions are multiples of the ring dimensions d_1 and d_2. While this can be guaranteed by padding the database with dummy records, it is straightforward to extend the scheme to support databases with arbitrary dimensions without padding. We describe the modifications to DBSetup and Query:

- DBSetup($1^\lambda, \mathbf{D}$): Suppose $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$. The DBSetup algorithm now sets $m_1 = \lceil \ell_1/d_1 \rceil$ and $m_2 = \lceil \ell_2/d_2 \rceil$. For $j \in \{1,2\}$, it samples $\mathbf{a}_j \xleftarrow{\mathrm{R}} R_{d_j, q_j}^{m_j}$ and lets $\bar{\mathbf{A}}_j = \mathsf{NCyclicMat}(\mathbf{a}_j^\mathsf{T}) \in \mathbb{Z}_{q_j}^{d_j \times m_j d_j}$. It parses $\bar{\mathbf{A}}_j = [\mathbf{A}_j \mid \mathbf{A}_j']$ where $\mathbf{A}_j \in \mathbb{Z}_{q_j}^{d_j \times \ell_j}$ and $\mathbf{A}_j' \in \mathbb{Z}_{q_j}^{d_j \times (m_j d_j - \ell_j)}$. The computation of $\mathbf{H}_1$ and $\mathbf{H}_2$ then proceeds as in Eq. (3.1).
- Query(pp, idx): The query algorithm proceeds exactly as in Construction 3.1 with $m_1 = \lceil \ell_1/d_1 \rceil$ and $m_2 = \lceil \ell_2/d_2 \rceil$, except it now computes $\bar{\mathbf{c}}_j = \mathsf{Coeffs}(s_j \mathbf{a}_j + \mathbf{e}_j + \Delta_j \boldsymbol{\mu}_j) \in \mathbb{Z}_{q_j}^{m_j d_j}$. It then parses $\bar{\mathbf{c}}_j = \left[\begin{smallmatrix} \mathbf{c}_j \\ \mathbf{c}_j' \end{smallmatrix}\right]$ where $\mathbf{c}_j \in \mathbb{Z}_{q_j}^{\ell_j}$ and $\mathbf{c}_j' \in \mathbb{Z}_{q_j}^{m_j d_j - \ell_j}$. The query is still q $= (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$.

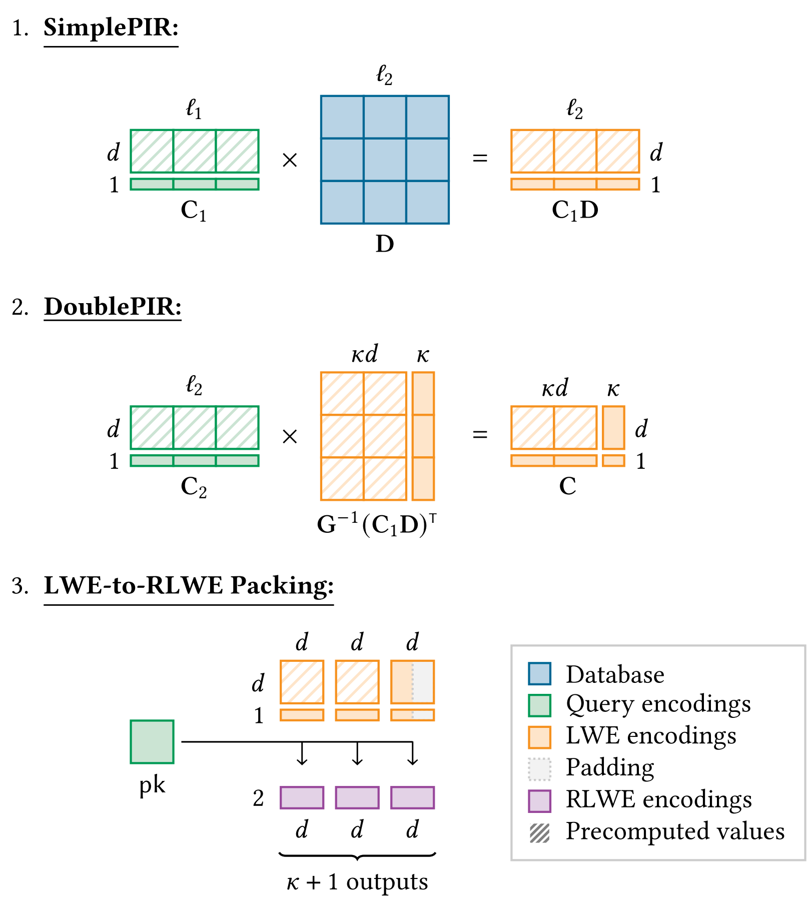

Figure 1: Illustration of the server computation in YPIR (i.e., the Answer algorithm in Construction 3.1). The operator $\times$ denotes matrix multiplication, the parameter d is the lattice (and ring) dimension, and κ is the ratio of the encoding modulus (i.e., q) to the size of the (intermediate) plaintext modulus (i.e., p). Each square cell represents a $d \times d$ matrix of $\mathbb{Z}_q$ values. The database is represented as an ℓ_1-by-ℓ_2 matrix, and each database values is an element of $\mathbb{Z}_N$. We let $\mathbf{C}_1$ and $\mathbf{C}_2$ be the stacked matrices $\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{c}_1^\top \end{bmatrix}$ and $\begin{bmatrix} \mathbf{A}_2 \\ \mathbf{c}_2^\top \end{bmatrix}$, respectively. The striped cells represent values that are precomputed by the server (i.e., these are independent of the query). We omit the modulus switching and the use of different lattice dimensions for simplicity.

Correctness and security. We state our correctness and security theorems below, and defer their formal proofs to Appendix B.

Theorem 3.4 (Correctness). *Let $N \in \mathbb{N}$ be the record size and $\ell_1, \ell_2 \in \mathbb{N}$ be the database dimensions. Let d_1, d_2, q_1, q_2, $\tilde{q}_1$, $\tilde{q}_{2,1}$, $\tilde{q}_{2,2}$, χ_1, χ_2, z, p be the scheme parameters from Construction 3.1. Suppose χ_1 and χ_2 are subgaussian with parameters σ_1 and σ_2, respectively. Let $\kappa = \lceil \log \tilde{q}_1 / \log p \rceil$, $\rho = \lceil \kappa(d_1 + 1)/d_2 \rceil$, and $t = \lfloor \log_z q_2 \rfloor + 1$. Then, under the independence heuristic, Construction 3.1 has correctness error*

$$\delta \le 2 d_2 \rho \exp(-\pi \tau_{\text{double}}^2 / \sigma_{\text{double}}^2) + 2 \exp(-\pi \tau_{\text{simple}}^2 / \sigma_{\text{simple}}^2),$$

SimplePIR	d_1	σ_1	N	q_1	$\tilde{q}_1$		
	2^{10}	$11\sqrt{2\pi}$	2^8	2^{32}	2^{28}		
DoublePIR	d_2	σ_2	p	q_2	$\tilde{q}_{2,1}$	$\tilde{q}_{2,2}$	z
	2^{11}	$6.4\sqrt{2\pi}$	2^{15}	$\approx 2^{56}$	2^{28}	2^{20}	2^{19}

Table 1: YPIR parameters chosen to provide correctness error $\delta \le 2^{-40}$ and 128-bits of security. These parameters support databases up to 64 GB (and dimensions $\ell_1, \ell_2 \le 2^{18}$). The parameters are partitioned into those used in the "SimplePIR" step (Step 1) and those used in the "DoublePIR" step (Step 2).

where

$$\begin{aligned}
\tau_{\text{double}} &= \frac{\tilde{q}_{2,2}}{2p} - (\tilde{q}_{2,2} \bmod p) - \frac{1}{2}\left(2 + (\tilde{q}_{2,2} \bmod p) + (\tilde{q}_{2,2}/q_2)(q_2 \bmod p)\right) \\
\sigma^2_{\text{double}} &\le (\tilde{q}_{2,2}/\tilde{q}_{2,1})^2 d_2 \sigma_2^2/4 + (\tilde{q}_{2,2}/q_2)^2(\sigma_2^2/4)(\ell_2 p^2 + (d_2^2 - 1)(t d_2 z^2)/3) \\
\tau_{\text{simple}} &= \frac{\tilde{q}_1}{2N} - (\tilde{q}_1 \bmod N) - \frac{1}{2}\left(2 + \tilde{q}_1 \bmod N + (\tilde{q}_1/q_1)(q_1 \bmod N)\right)/2 \\
\sigma^2_{\text{simple}} &\le d_1\sigma_1^2/4 + (\tilde{q}_1/q_1)^2 \ell_1 N^2 \sigma_1^2/4.
\end{aligned}$$

Theorem 3.5 (Security)**.** *Under the* $\mathsf{RLWE}_{d_1,m_1,q_1,\chi_1}$ *assumption and assuming the LWE-to-RLWE packing scheme* (CDKS.Setup, CDKS.Pack) *satisfies pseudorandomness given the public key (Theorem 2.4), then Construction 3.1 satisfies query privacy.*

Security assumptions. As we show in Appendix A.1, the security of the LWE-to-RLWE packing scheme relies on hardness of $\mathsf{RLWE}_{d_2,m_2',q_2,\chi_2}$ where $m_2' = m_2 + (\lfloor \log_z q_2 \rfloor + 1) \cdot \log d_2$ along with a "circular security" assumption on RLWE encodings. The latter assumption is a standard assumption when working with lattice-based homomorphic encryption schemes [Gen09, BV11, BGV12, Bra12] and used in many previous RLWE-based PIR schemes [ACLS18, AYA+21, MCR21, MW22, LMRS23].

4 Implementation and Evaluation

In this section, we describe our implementation and experimental evaluation of the YPIR protocol (Construction 3.1).

Parameter selection. Theorem 3.4 bounds the correctness error δ of YPIR as function of the scheme parameters. We now describe how we instantiate the different parameters to achieve a correctness error $\delta \le 2^{-40}$ and 128-bits of security (as estimated by the Lattice Estimator [APS15][7]). We select a single parameter set for YPIR using the following procedure:

- Like SimplePIR [HHC+23a], we set $d_1 = 2^{10} = 1024$ and $q_1 = 2^{32}$. We set χ_1 to be a discrete Gaussian distribution with parameter $s_1 = 11\sqrt{2\pi}$ (to achieve 128-bits of security for this choice of ring dimension and modulus).
- For the DoublePIR and LWE-to-RLWE packing steps, we work over a larger ring, to allow for the extra noise added by the LWE-to-RLWE transformation. Here, we choose $d_2 = 2^{11} = 2048$ and q_2 to be a 56-bit modulus that splits into a product of two (28-bit) NTT-friendly modulus (specifically, $q_2 = (2^{28} - 2^{16} + 1) \cdot (2^{28} - 2^{24} - 2^{21} + 1)$). Using two 28-bit NTT-friendly modulus allows us to use native 64-bit integer arithmetic to implement arithmetic operations modulo each of the prime factors of q_2.[8] We choose χ_2 to be a discrete Gaussian distribution with parameter $s_2 = 6.4\sqrt{2\pi}$ (to achieve 128-bits of security for this choice of ring dimension and modulus).

[7]We use commit `4195c66` (2024/02/06) from `https://github.com/malb/lattice-estimator` for our security estimates.

[8]Since we use 64-bit integer arithmetic to implement arithmetic operations with respect to a 28-bit modulus, we do *not* need to perform a modulus reduction after *every* arithmetic operation. For instance, in our experiments, the computation of Eq. (3.2) is 60× slower if we perform a modulus reduction after every arithmetic operation. In our implementation, we reduce only when the computation might "overflow" the 64-bit integer.

- We choose parameters to support any choice of $\ell_1, \ell_2 \le 2^{18}$ (recall that the database in YPIR is represented as an ℓ_1-by-ℓ_2 matrix). This is sufficient to support databases with up to 2^{36} records (and for our choice of N, up to 64 GB in size).
- We choose the largest value for the gadget decomposition base $z \in \mathbb{N}$ that achieves correctness error at most $\delta \le 2^{-40}$. This allows for faster computation (smaller gadget decompositions).
- We choose the largest value of N and the largest intermediate decomposition base p that achieves correctness error at most δ when $\tilde{q}_1 = q_1$ and $\tilde{q}_{2,1} = \tilde{q}_{2,2} = q_2$. This minimizes the communication overhead of the scheme. We constrain N and p to be powers-of-two so elements can be represented by a single machine word.
- After fixing N and p, we choose the smallest modulus switching parameters $\tilde{q}_1, \tilde{q}_{2,1}, \tilde{q}_{2,2}$ that achieve correctness error at most δ. This minimizes the size of the responses.

We summarize the lattice parameters we select in Table 1. When the database consists of ℓ one-bit records, we let $\ell' = \lceil \ell / \log N \rceil$, and set $\ell_1 = 2^{\lceil \log \ell' / 2 \rceil}$ and $\ell_2 = 2^{\lfloor \log \ell' / 2 \rfloor}$.

4.1 NTT-Based Hint Computation

We now describe our approach to efficiently compute the hints $\mathbf{H}_1$ and $\mathbf{H}_2$ in Eq. (3.1) of YPIR. By using structured matrices, the YPIR approach is asymptotically faster (by a factor $d/\log d$, where d is the lattice dimension) and concretely faster (10–15×) compared to the preprocessing approaches of protocols like SimplePIR [HHC+23a] or FrodoPIR [DPC22]. Our approach directly applies to reduce the preprocessing cost in any system that builds on SimplePIR/FrodoPIR (e.g., [CNC+23, HDCZ23, LMRS23, dCL24]) with *zero* impact to the online costs of the protocol (the online server processing is unchanged). The only difference is security relies on RLWE rather than LWE.

SimplePIR and FrodoPIR preprocessing. SimplePIR [HHC+23a] and FrodoPIR [DPC22] achieve high throughput by moving the majority of the server processing cost to a query-independent offline phase. There, the offline precomputation consists of computing a matrix-vector product $\mathbf{A} \cdot \mathbf{D} \in \mathbb{Z}_q^{d \times \ell_2}$, where $\mathbf{A} \in \mathbb{Z}_q^{d \times \ell_1}$ is a random matrix and $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$ is the database. Here, d is the lattice dimension. The query in their protocols consist of LWE encodings of the index with respect to the matrix $\mathbf{A}$. With a naïve matrix-matrix multiplication algorithm, computing the product $\mathbf{A} \cdot \mathbf{D}$ requires $O(\ell_1 \ell_2 d)$ arithmetic operations. Concretely, in these schemes, $d \ge 2^{10}$, so the offline precomputation is very expensive.

Our approach. In YPIR, we achieve faster preprocessing by using a *structured* negacyclic matrix $\mathbf{A} = \mathsf{NCyclicMat}(\mathbf{a}^\mathsf{T})$, where $\mathbf{a} \in R_q^m$. When $\mathbf{A}$ is negacyclic and the modulus q is NTT-friendly, we can use the NTT to compute the product $\mathbf{A} \cdot \mathbf{D}$ using $O(\ell_1 \ell_2 \log d)$ arithmetic operations. This yields a $d/\log d$ speed-up over previous approaches. On the flip side, security of the scheme now relies on the *ring* LWE assumption (as opposed to plain LWE), since the queries are now encodings of the index with respect to the structured matrix $\mathbf{A}$. We provide more details in Appendix C.

Modulus selection. SimplePIR/FrodoPIR use a power-of-two modulus q so arithmetic operations can make use of native machine-word arithmetic and avoid expensive modular reductions. Unfortunately, such q is not NTT-friendly (since $q \ne 1 \bmod 2d$). In YPIR, we retain the same power-of-two modulus q as in prior work. However, to compute the hint $\mathbf{A} \cdot \mathbf{D}$ using NTTs, we work over $\mathbb{Z}_M$ where $M > dNq$ is an NTT-friendly modulus. Noting that the entries in $\mathbf{A}$ are bounded by q and those in $\mathbf{D}$ are bounded by N, computing $\mathbf{A} \cdot \mathbf{D} \bmod M$ is equivalent to computing $\mathbf{A} \cdot \mathbf{D}$ over the integers. This is sufficient for computing $\mathbf{A} \cdot \mathbf{D} \in \mathbb{Z}_q^{d \times \ell_2}$. Even though this approach requires working over a larger modulus M, the ability to use NTTs to speed up the matrix-matrix multiplication outweighs the costs (see Section 4.3).

Remark 4.1 (SimplePIR with RLWE)**.** Given that the use of structured matrices (and ring LWE) allows faster preprocessing, a natural question is why we do not just apply SimplePIR over polynomial rings. In this setting, we represent the database as $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$ by packing each block of d entries into the coefficients of a polynomial; the resulting database is $\hat{\mathbf{D}} \in R_N^{\ell_1/d \times \ell_2}$. The public matrix $\mathbf{A} \in \mathbb{Z}_q^{d \times \ell_2}$ is replaced by a vector $\hat{\mathbf{a}} \in R_q^{\ell_2}$ and the query vector $\mathbf{q} \in \mathbb{Z}_q^{\ell_1}$ becomes $\hat{\mathbf{q}} \in R_q^{\ell_1/d}$. Like in SimplePIR, the online computation is a matrix-vector product of ring elements $\hat{\mathbf{q}}^\mathsf{T}\hat{\mathbf{D}}$. If

$\hat{\mathbf{q}}$ and $\hat{\mathbf{D}}$ are in NTT representation, then computing $\hat{\mathbf{q}}^{\mathsf{T}}\hat{\mathbf{D}}$ requires the *same* number of arithmetic operations modulo q as computing $\mathbf{q}^{\mathsf{T}}\mathbf{D}$ in SimplePIR. While this seems like a reasonable approach, there are two significant limitations:

- **Choice of modulus q:** For fast processing, the query vector $\hat{\mathbf{q}}$ and the database $\hat{\mathbf{D}}$ need to be in NTT representation. This means we either need to choose an NTT-friendly modulus q or we need to run FFT over the complex numbers. Both are more costly compared to using a power-of-two modulus (where mod q operations can be implemented with native integer arithmetic).

- **Size of in-memory representation:** A bigger limitation in practice is the fact that to leverage NTTs, the server needs to store the database $\mathbf{D}$ in NTT representation (over the *encoding ring* $\mathbb{Z}_q$). This increases the *size* of the in-memory representation of the database. For instance, if the database elements are $\mathbb{Z}_N$-elements and the query encodings are $\mathbb{Z}_q$-elements, there is a $\log q / \log N$ overhead in storage. As we discuss in Sections 4.2 and 4.3, the throughput of the SimplePIR-family of approaches is bottlenecked by *memory bandwidth*. This means a $\log q / \log N$ increase in representation *size* translates to an equal reduction in throughput. Experimentally, we compared a basic implementation of SimplePIR with RLWE encodings to standard SimplePIR and observed that the use of RLWE incurs a 3.6× reduction in throughput (from 11.5 GB/s to 3.2 GB/s). For this parameter setting, $\log q / \log N = 56/16 = 3.5$.

The approach we take in YPIR allows us to get the best of both worlds. We keep the SimplePIR structure of working over the integers, but replace $\mathbf{A}$ with a negacyclic matrix. This allows us to use NTTs for fast preprocessing, but does not introduce additional storage overhead for the server processing (since the online computation is still performed over the integers $\mathbb{Z}_q$ rather than the polynomial ring R_q).

4.2 Cross-Client Batching and the Memory Bandwidth Barrier

SimplePIR [HHC⁺23a] achieves the highest concrete server throughput among all single-server protocols (that do not require streaming the full database in an offline phase). The bottleneck in SimplePIR is the *memory bandwidth* of the machine and *not* the cost of performing the underlying arithmetic operations during query processing. One way to increase the *effective* throughput of the protocol then is to perform additional computation for each byte of memory accessed. A natural approach would be to process *multiple* queries with a single scan through memory. The notion of batch PIR is well-studied in the case where a *single* client seeks to make multiple queries [BIM00, IKOS04, GKL10, ACLS18, MR23], and indeed batch PIR enables significant improvements to server throughput both concretely and asymptotically.

Here, we show that *concrete* reductions in computation time are possible even if multiple *independent* clients are making queries. We refer to our approach as *cross-client batching*. As we show in Section 4.3, cross-client batching can increase the effective server throughput of many PIR schemes by 1.5–1.7×. The benefit of our approach is that it is entirely transparent to the client (i.e., requires no changes client-side) and applies even if the clients are each making a single query.

Cross-client batching. The idea in cross-client batching is to use a *single* scan through the database to simultaneously answer multiple queries from *non-coordinating* clients. The structure of many recent PIR protocols is directly amenable to cross-client batching. As a concrete example, in SimplePIR, the query computation corresponds to computing a matrix-vector product $\mathbf{q}^{\mathsf{T}}\mathbf{D}$ where $\mathbf{q}$ is the query vector and $\mathbf{D}$ is the database. Suppose k different clients each issue a SimplePIR query $\mathbf{q}_1^{\mathsf{T}}, \dots, \mathbf{q}_k^{\mathsf{T}}$. Let $\mathbf{Q}$ be the matrix formed by stacking the vectors $\mathbf{q}_1^{\mathsf{T}}, \dots, \mathbf{q}_k^{\mathsf{T}}$. To answer the k queries, the server now computes the matrix-matrix product $\mathbf{Q} \cdot \mathbf{D}$. While this would require the *same* number of elementary multiplications as computing $\mathbf{q}_1^{\mathsf{T}}\mathbf{D}, \dots, \mathbf{q}_k^{\mathsf{T}}\mathbf{D}$ individually, the server can now compute $\mathbf{Q} \cdot \mathbf{D}$ with only a *single* pass over the database. Put another way, for every entry of $\mathbf{D}$ that the CPU accesses, it now performs k arithmetic operations rather than 1. Since memory bandwidth is the bottleneck of the basic protocol, computing $\mathbf{Q} \cdot \mathbf{D}$ is overall *faster* than separately computing $\mathbf{q}_1^{\mathsf{T}}\mathbf{D}, \dots, \mathbf{q}_k^{\mathsf{T}}\mathbf{D}$. If it takes time T to process k queries on a database of size ℓ, we define the effective throughput of the protocol to be $k\ell/T$. Similar batching techniques have been used to improve the throughput of database systems in settings where the storage bandwidth is saturated [CMS16, CGB⁺14].

Practical considerations. Cross-client batching only makes sense in a setting where the server often has a queue that is at least k deep. In our experiments, a small batch size of $k = 4$ is sufficient to get most of the advantages of cross-client batching. In many of the applications of PIR (e.g., private DNS or private Certificate Transparency auditing), assuming a queue of size $k = 4$ is a mild assumption. The non-private versions of each of these services employ large server fleets that regularly process requests from more than 4 clients simultaneously. Moreover, there is no requirement that the server must wait until the queue is full before processing the query.

Memory bandwidth in RLWE-based PIR. PIR schemes based on RLWE [ACLS18, MCR21, MW22] require less communication than SimplePIR and DoublePIR, but have much smaller throughput. Even though the overall throughput of these schemes is much smaller than the system's memory bandwidth, we observe that memory bandwidth is also a constraint in these schemes (for reasons similar to those outlined in Remark 4.1).

In a typical RLWE-based PIR scheme, the database records are represented by polynomials in R_N, where N is the plaintext modulus. To process the query, the server performs multiplications over the ring R_q (i.e., the encoding space), where the encoding modulus q is much greater than N. To efficiently implement the polynomial arithmetic over R_q, the plaintext polynomials need to be stored in their NTT representation (over R_q). This incurs a $\log q / \log N$ blowup in the size of the in-memory representation of the database (or the protocol incurs a significant degradation in throughput). Because RLWE-based schemes must pay for this $\log q / \log N$ factor in representation size, even when they saturate memory bandwidth, they can only achieve a throughput of $M \log N / \log q$, where M is the memory bandwidth. For example, the memory blowup factor is 8 for the Spiral system [MW22] ($\log N = 8$ and $\log q \approx 64$). A standard per-core memory bandwidth is 14.6 GB/s, which leads to a throughput upper bound of roughly 1.8 GB/s (roughly the throughput reported for the fastest version of Spiral [MW22]). We refer to Remark 4.1 for a similar analysis in the setting of SimplePIR instantiated with RLWE.

Our cross-client batching approach can be applied to RLWE schemes like Spiral to improve the effective throughput. However, the benefit there is smaller since cross-client batching only helps improve the initial linear scan over the database (and does not help with the subsequent folding steps). The Spiral scheme sets the parameters to balances the cost of the initial linear scan with the folding steps; as such, cross-client batching would only be beneficial to the first half of the protocol processing as opposed to the full processing.

4.3 Experimental Evaluation

In this section, we describe our experimental evaluation of the YPIR protocol and compare it against other PIR protocols. For our comparisons, we focus on the state-of-the-art high-throughput single-server PIR schemes: SimplePIR/DoublePIR [HHC⁺23a], and the hintless variant proposed in Tiptoe [HDCZ23]. Where applicable, we also report communication numbers for HintlessPIR [LMRS23].[9] We measure the communication (query size and response size), and the server throughput (the ratio of the database size to the server computation time). We also measure the throughput of the server precomputation, and for SimplePIR and DoublePIR, the size of the hints (i.e., the offline communication). We do not benchmark schemes in alternative models such as the sublinear schemes that require streaming the database in the offline phase [ZPSZ24, MSR23, GZS24] or the RLWE-based schemes that require maintaining client-specific keys [ACLS18, AYA⁺21, ALP⁺21, MCR21, MW22].

Experimental setup. We implement YPIR in 3000 lines of Rust, with a 1000 line C++ kernel for fast 32-bit matrix-multiplication adapted from the public SimplePIR implementation [HHC⁺23a].[10] We use the approach from Section 4.1 to implement the server precomputation (and for computing $\mathbf{A}_2\mathbf{T}^\intercal$ in Eq. (3.2)), as well as leverage an additional optimization of the LWE-to-RLWE packing transformation described in Remark A.8. As discussed in Remarks 3.2 and A.7, we compress the vectors $\mathbf{a}_1$ and $\mathbf{a}_2$ in the public parameters pp as well as the pseudorandom components of the packing key pk using the output of a stream cipher (ChaCha20 in counter mode). We benchmark YPIR against the public implementations of SimplePIR, DoublePIR [HHC⁺23a] and the PIR scheme from Tiptoe [HDCZ23]. We compile Tiptoe with support for the Intel HEXL [BKS⁺21] acceleration library. We use an Amazon EC2 `r6i.16xlarge` instance running Ubuntu 22.04, with 64 vCPUs (Intel Xeon Platinum 8375C CPU @ 2.9 GHz) and 512 GB of RAM. We

[9]At the time of writing, we could not find a public implementation of HintlessPIR.
[10]Our code is available at `https://github.com/menonsamir/ypir`.

Database	Metric	SimplePIR	DoublePIR	HintlessPIR	Tiptoe	YPIR
1 GB	**Prep. Throughput**	3.7 MB/s	3.4 MB/s	$*$	1.6 MB/s	42 MB/s
	Off. Download	121 MB	16 MB	–	–	–
	Upload	120 KB	312 KB	506 KB	33 MB	846 KB
	Download	120 KB	32 KB	1.5 MB	2.1 MB	12 KB
	Server Time	74 ms	94 ms	$*$	2.47 s	428 ms
	Throughput	13.6 GB/s	10.6 GB/s	$*$	415 MB/s	2.3 GB/s
8 GB	**Prep. Throughput**	3.1 MB/s	2.9 MB/s	$*$	1.6 MB/s	49 MB/s
	Off. Download	362 MB	16 MB	–	–	–
	Upload	362 KB	724 KB	506 KB	33 MB	1.5 MB
	Download	362 KB	32 KB	1.5 MB	8.6 MB	12 KB
	Server Time	708 ms	845 ms	$*$	9.75 s	992 ms
	Throughput	11.3 GB/s	9.5 GB/s	$*$	840 MB/s	8.1 GB/s
32 GB	**Prep. Throughput**	3.3 MB/s	3.3 MB/s	$*$	1.4 MB/s	49 MB/s
	Off. Download	724 MB	16 MB	–	–	–
	Upload	724 KB	1.4 MB	$*$	34 MB	2.5 MB
	Download	724 KB	32 KB	$*$	17 MB	12 KB
	Server Time	3.08 s	3.22 s	$*$	21.00 s	2.93 s
	Throughput	10.4 GB/s	9.9 GB/s	$*$	1.5 GB/s	10.9 GB/s

Table 2: Communication and computation needed to retrieve a single bit for databases of varying sizes. For each scheme, we also measure the speed of the preprocessing algorithm ("Prep. Throughput") that the server must run upon each database update, and if applicable, the size of the hint that the client must download in the offline phase ("Off. Download"). The measurements for SimplePIR, DoublePIR, [HHC+23a] and Tiptoe [HDCZ23] are obtained by running their official implementation on our test system. At the time of writing, we could not find a public implementation of HintlessPIR, so we only report the communication costs based on the provided measurements [LMRS23, Table 2] on a similarly-sized database.

use the same (single-threaded) benchmarking environment for all experiments, and compile all of the implementations using `GCC 11`. The processor supports the AVX2 and AVX-512 instruction sets, and we enable SIMD instruction set support for all schemes. All of our runtime measurements are averaged from a minimum of 5 sample runs and have a standard deviation of at most 5%.

Server throughput. In Table 2, we report the different computational and communication costs for retrieving a 1-bit record from databases of varying sizes. We focus on single-bit retrieval since this is the setting of interest in private SCT auditing and provides a common baseline for comparing different schemes. Each YPIR response actually encodes an element of $\mathbb{Z}_N$ (for our parameters, each record is 8-bits long).

For small databases (e.g., 1 GB), the throughput of YPIR is 6× slower than SimplePIR and 5× slower than DoublePIR. This is because most of the query-processing time is spent on the LWE-to-RLWE transformation (80%; see Table 3). However, since the cost of this transformation is essentially *independent* of the size of the database, the throughput of YPIR quickly approaches that of DoublePIR as the size of database increases. With an 8 GB database, the throughput is just 1.4× slower than SimplePIR and DoublePIR. At 32 GB, the throughput of YPIR is 10.9 GB/s, which is 5–10% *faster* than the reference implementations of SimplePIR and DoublePIR and 75% of the memory bandwidth of the system. The efficiency gain over SimplePIR and DoublePIR is due both to a different choice of parameters in YPIR compared to the reference implementation [HHC+23b] and to a more optimized implementation. To compare the schemes on an even footing, we include measurements against *our* implementation of these protocols (denoted SimplePIR* and DoublePIR*) with our parameters from Table 1 in Appendix D (Table 7). Compared to our SimplePIR*

and DoublePIR* implementations, the throughput of YPIR on a 32 GB database is about 1.15× slower. Thus, for moderate-size databases, YPIR achieves comparable throughput to SimplePIR and DoublePIR *without* any offline hints. We also show the throughput of the different schemes as a function of the database size in Fig. 2.

Compared to the Tiptoe approach [HDCZ23], YPIR achieves 5–10× higher throughput. This is because over 85% of the server processing time in Tiptoe is spent on the LWE-to-RLWE conversion algorithm (based on homomorphic decryption). In YPIR, for large databases, the LWE-to-RLWE packing is only 10–20% of the total server processing time (see Table 3). For the private web search application that Tiptoe was designed for, this packing step does not impact client query latency, but the server incurs the full computational costs of the packing process. Tiptoe's throughput increases with database size, because larger databases help amortize the cost of LWE-to-RLWE packing (which scales with the square root of the database size).

The HintlessPIR paper [LMRS23] reports a maximum throughput that is 60% of the throughput of SimplePIR (for an 8 GB database). With YPIR, we are able to achieve server throughput that is 85% of the throughput of (our faster version of) SimplePIR. In absolute terms, the server throughput reported in [LMRS23] is only 3.8 GB/s; in our test environment, SimplePIR and YPIR on a similar-size database achieve server throughputs of 11.3 GB/s and 8.1 GB/s, respectively, which are considerably higher. Like Tiptoe and YPIR, HintlessPIR also relies on an LWE-to-RLWE transformation. Their approach requires working over a larger ring ($d = 2^{12}$) and a 108-bit modulus q; moreover, they apply their transformation to the full SimplePIR response whereas YPIR only applies the LWE-to-RLWE packing to the smaller DoublePIR response. The number of (expensive) ring operations HintlessPIR performs scales with the square root of the database size, while the number of such operations needed in YPIR only depend on the ring dimension. For this reason, we conjecture that the YPIR packing approach has substantially smaller concrete costs in a direct comparison.

Communication. Comparing the communication requirements of YPIR to hint-based schemes, the queries in YPIR are about 1.8–2.7× larger than DoublePIR and 3.5–7× larger than SimplePIR (with smaller overheads for larger databases). The larger queries are due to the key-switching matrices needed for the LWE-to-RLWE packing. On the flip side, the response size for YPIR is 2.7× *smaller* than DoublePIR and 10–60× smaller than SimplePIR. This is due to the better *rate* of RLWE encodings compared to LWE encodings, as well as the use of modulus switching in our implementation. The response size of YPIR and DoublePIR depend only on the lattice parameters and *not* the database size; in SimplePIR, the response scales with the square root of the database size. If we look at total *online* communication (both upload and download), the cost of YPIR is only 1.8–3.6× larger compared to SimplePIR and 1.8–2.5× larger compared to DoublePIR. The key advantage, of course, is that YPIR does not require the client to download a hint. In the case of a 32 GB database, the size of the hint is 724 MB for SimplePIR and 16 MB for DoublePIR. A client would have to make 681 queries to SimplePIR or 15 queries to DoublePIR before the *total* communication of YPIR is worse. Thus, for dynamic settings where the database is frequently changing and the client makes a small number of queries at a time, YPIR gives a net reduction in communication with only a small hit to throughput.

Compared to HintlessPIR, YPIR queries are 1.7–3× larger and responses are 125× smaller. The YPIR response size is significantly smaller because the HintlessPIR response size scales with the square root of the database size (like SimplePIR). On the other hand, YPIR queries are larger than in HintlessPIR due to needing more key-switching matrices for the LWE-to-RLWE packing. Compared to Tiptoe, YPIR has 13–39× smaller queries and 175–1417× smaller responses. because its response does not scale with the database size. The *total* communication cost for a issuing a single query for an 8 GB database is 1.5 MB for YPIR, 2 MB for HintlessPIR, and 42 MB for Tiptoe.

Preprocessing cost. Our NTT-based precomputation (Section 4.1) is about 10–15× faster than that of SimplePIR or DoublePIR. For a 32 GB database, the offline precomputation of YPIR would take about 11 CPU-minutes, whereas for SimplePIR/DoublePIR, it would take roughly 144 CPU-minutes. As described in Section 4.1, our precomputation method can be used directly in SimplePIR/DoublePIR without affecting the online performance of the protocol.

Microbenchmarks. Table 3 provides a fine-grained breakdown of the server computation costs of YPIR (i.e., the Answer algorithm in Construction 3.1). We separately measure the costs of the SimplePIR step (Step 1), the DoublePIR step (Step 2), and the LWE-to-RLWE packing step (Step 3). The modulus switching cost is insignificant

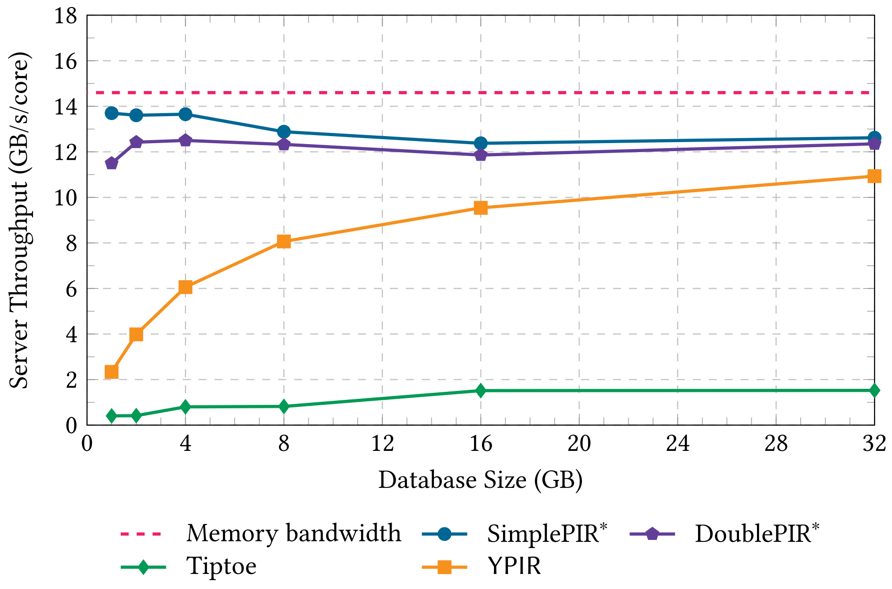

Figure 2: Server throughput for retrieving a single bit from different databases. For SimplePIR and DoublePIR, we report throughput using our reference implementation and parameter choices (denoted SimplePIR* and DoublePIR*), since these were faster than those of the reference implementation [HHC+23b] in our test setup. We measure the memory bandwidth of the system using STREAM [McC95].

Size	**SimplePIR**	**DoublePIR**	**Packing**	**Total**
1 GB	0.07 s (17%)	0.01 s (3%)	0.34 s (80%)	0.43 s
4 GB	0.29 s (45%)	0.03 s (4%)	0.34 s (51%)	0.66 s
16 GB	1.29 s (77%)	0.06 s (3%)	0.34 s (20%)	1.69 s
32 GB	2.54 s (86%)	0.06 s (2%)	0.34 s (12%)	2.94 s

Table 3: Breakdown of YPIR server computation time for retrieving a single bit from databases of varying sizes. For each database size, we report the time spent in the SimplePIR step (Step 1), the DoublePIR step (Step 2), and the LWE-to-RLWE packing step (Step 3) for the Answer algorithm in Construction 3.1. In parentheses, we report the percentage of the total time spent on the associated step.

compared to the other three components so we do not include it in the breakdown. First, we observe that the packing transformation essentially incurs a *fixed* cost to the server processing time. This is because the LWE-to-RLWE packing transformation in YPIR is applied to the DoublePIR responses, which does *not* scale with the size of the database. For small databases (e.g., 1 GB), the packing transformation represents 80% of the server processing time, but as the size of the database grows, the cost of the linear scan over the database (i.e., the SimplePIR step) dominates. With a 32 GB database, the packing transformation is only 12% of the overall cost of the server processing. In this case, the throughput of YPIR quickly approaches that of DoublePIR.

In Table 4, we provide a breakdown of the different components of the YPIR query. From Construction 3.1, the YPIR query consists of two sets of LWE encodings $\mathbf{c}_1, \mathbf{c}_2$ (that encode indicator vectors of the row and column of the desired database record) as well as the packing parameters pk (i.e., the key-switching matrices) for the LWE-to-RLWE packing transformation (Theorem 2.4). The size of the packing parameters matrices are *fixed* (concretely, these are 462 KB), while the encodings of the indicator vectors for the row and the column scale with the number of rows and columns, respectively. In our experiments, the database is arranged as a square with an equal number of rows and columns. As such, the number of LWE encodings needed to encode the indicator vectors for the row ($\mathbf{c}_1$) and for the column ($\mathbf{c}_2$) are the same. However, we use larger parameters for the second set of encodings $\mathbf{c}_2$ (to support the LWE-to-RLWE

Database Size	$\lvert \mathbf{c}_1 \rvert$	$\lvert \mathbf{c}_2 \rvert$	$\lvert \mathsf{pk} \rvert$	**Total Size**
1 GB	128 KB (15%)	256 KB (30%)	462 KB (55%)	846 KB
4 GB	256 KB (21%)	512 KB (42%)	462 KB (38%)	1.2 MB
16 GB	512 KB (26%)	1.0 MB (51%)	462 KB (23%)	2.0 MB

Table 4: Breakdown of YPIR query size for retrieving a single bit from databases of varying sizes. Recall from Construction 3.1 that the query consists of three components: (1) the LWE encoding $\mathbf{c}_1$ of the row of interest (processed in the initial SimplePIR step), (2) the LWE encoding $\mathbf{c}_2$ of the column of interest (processed in the DoublePIR step), and (3) the key-switching parameters pk for the LWE-to-RLWE packing. We report the size of each of these components. In parenthesis, we report the percentage of the total query size associated with each component.

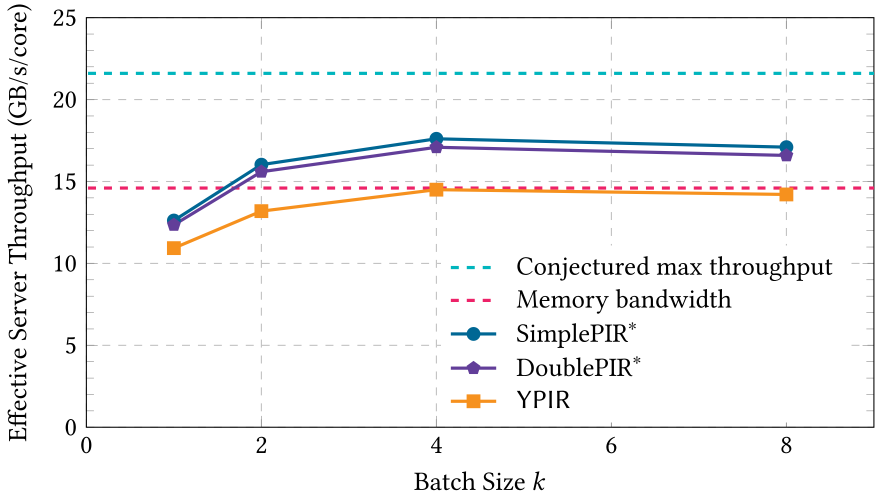

Figure 3: Effective per-query server throughput for retrieving a single bit from a 32 GB database with cross-client batching. For a batch of k client queries and a database of size ℓ, the effective per-query server throughput is $k\ell/T$, where T is the time to process all k queries. As in Fig. 2, we use *our* implementation of SimplePIR and DoublePIR (i.e., SimplePIR* and DoublePIR*) for the comparisons. We measure the memory bandwidth of the system using STREAM [McC95]. We compute the conjectured maximum possible throughput for these schemes based on the assumption that processing each database byte requires a *minimum* of two 32-bit arithmetic operations and using the clocks-per-instruction values provided by the CPU vendor [Int23].

packing transformation). As such, the encoding $\mathbf{c}_2$ is roughly $(\log q_2 / \log q_1) \approx 2$ larger than the encoding $\mathbf{c}_1$.

Cross-client batching. We modify SimplePIR, DoublePIR, and YPIR to support cross-client batching as described in Section 4.2. For a database of size ℓ, we define the effective (per-query) server throughput to process a batch of k queries to be $k\ell/T$, where T is the time it takes to answer all k queries. We consider batch sizes ranging from $k = 1$ to $k = 8$ and measure the effective throughput of the scheme for retrieving a single bit from a 32 GB database in Fig. 3. In all cases, using cross-client batching increases the effective throughput by a factor of up to 1.4×. In the case of SimplePIR and DoublePIR, processing a batch of 4 queries yields a 1.4× improvement (an effective throughput of over 17 GB/s). This is higher than the *memory throughput* of the machine. With YPIR, the effective throughput for a batch size of 8 is over 14 GB/s, which is also 1.4× larger than the single-query throughput. The gap in effective throughput between YPIR and SimplePIR widens as we increase k, since the fixed cost of the LWE-to-RLWE packing (see Table 3) does not benefit from cross-client batching. These results show that for setting where a server needs to process concurrent queries from different clients, it is advantageous to process them in a batch rather than sequentially, even

	Tiptoe [HDCZ23]	**Chen et al. [CDKS21]**
Parameter Size	32 MB	528 KB
Computation	306 ms	340 ms

Table 5: Concrete costs of translating $d = 2^{11}$ LWE encodings into an RLWE encoding using the Tiptoe bootstrapping-based approach and the Chen et al. [CDKS21] packing approach used in YPIR. We assume the LWE encodings are over $\mathbb{Z}_q^{n+1}$ where $n = 2^{11}$ is the lattice dimension and $\log q \approx 64$. We report the server computation time as well as the size of the public parameters the client must upload to the server.

though there is no reduction in the total number of arithmetic operations the server performs.

LWE-to-RLWE translation. Tiptoe [HDCZ23], HintlessPIR [LMRS23], and YPIR all apply some form of LWE-to-RLWE translation to *compress* the SimplePIR/DoublePIR responses and eliminate the need for an offline hint download. Both Tiptoe and HintlessPIR rely on a bootstrapping-based approach where the client provides an RLWE encoding of the secret key in its query. After the server computes its SimplePIR response, it uses the RLWE encoding of the secret key to homomorphically compute the inner product between the encoded secret key and the SimplePIR response. This essentially yields an RLWE encoding of the database record (this relies on the fact that LWE decryption is a linear operation so the inner product between an LWE encoding and the secret key yields an encoding of the plaintext).

On the other hand, YPIR applies the Chen-Dai-Kim-Song packing transformation [CDKS21] to the *DoublePIR* response. While this can also be viewed as a type of "bootstrapping" (since the transformation relies on key-switching which is in some sense a homomorphic decryption operation), a key difference is that YPIR only applies it to the DoublePIR output. The SimplePIR output scales with the square root of the database size whereas the DoublePIR output scales only with the ring dimension. Concretely, the number of (relatively more) expensive ring operations that YPIR has to perform is smaller than the corresponding number in Tiptoe or HintlessPIR. To illustrate the difference in cost between the SimplePIR step and the subsequent processing steps, we note that in HintlessPIR, they use a 1408-dimension lattice and a power-of-two modulus $q = 2^{32}$ for SimplePIR, but a considerably-larger 4096-dimension ring and 108-bit non-power-of-two modulus for the subsequent step. In YPIR, the initial SimplePIR step uses a 1024-dimension lattice and $q = 2^{32}$ while the DoublePIR and LWE-to-RLWE packing steps operates over a 2048-dimension lattice/ring and a 56-bit modulus.

In Table 5, we provide microbenchmarks to compare the effectiveness of the Tiptoe bootstrapping-based approach for packing LWE encodings into more compact RLWE encodings with the Chen et al. [CDKS21] packing approach we use in YPIR. Both approaches require the client to upload a set of translation parameters to the server; in Tiptoe, this is the encoding of the secret key whereas in YPIR, these are key-switching matrices. Since Tiptoe encodes each component of the secret key individually, the translation parameters have size $O(d^2)$, where d is the lattice/ring dimension. The key-switching matrices in the Chen et al. [CDKS21] approach are $O(d \log d)$ in size, which are significantly smaller. This is significant in practice: the concrete key size for the Chen et al. approach is just 528 KB, over 60× shorter than the Tiptoe parameters (32 MB). On the flip side, the Tiptoe approach only requires fast integer arithmetic, and thus, is 1.1× faster than the Chen et al. approach (306 ms vs. 340 ms to process 2^{11} encodings). However, in the context of the overall PIR protocol, Tiptoe applies their transformation to the SimplePIR response, which is much larger than the DoublePIR response (e.g., 180× larger for a 32 GB database). So even though the raw packing rate of the Tiptoe approach is modestly faster than the Chen et al. [CDKS21] approach, the end-to-end server throughput for the overall PIR protocol is still much slower because the packing has to be performed on a much larger input. Moreover, the significance reduction in the size of the translation key is critical to maintaining small queries in YPIR.

4.4 Application to Private SCT Auditing

Certificate Transparency (CT) [Lau14, LLK13] is a standard for monitoring and auditing the issuance of digital certificates by maintaining a public append-only log of every certificate issued by every certificate authority. In this model, whenever a certificate authority (CA) issues a certificate, it also deposits the certificate into one or more CT

Update Frequency	**DoublePIR** *Weekly*	**DoublePIR** *Daily*	**Tiptoe** –	**HintlessPIR** –	**YPIR** –
Offline Download	16 MB	112 MB	–	–	–
Upload	14 MB	14 MB	659 MB	9.9 MB	29 MB
Download	640 KB	640 KB	172 MB	31 MB	240 KB
Computation	16.90 s	16.90 s	194.96 s	$*$	19.84 s
Communication Cost	\$0.001569	\$0.010610	\$0.016215	\$0.002898	\$0.000022
Computation Cost	\$0.000253	\$0.000253	\$0.002924	$*$	\$0.000298
Total Cost	\$0.001822	\$0.010863	\$0.019139	\$0.002898	\$0.000320

Table 6: Weekly server costs per client needed to support private SCT auditing using the PIR-based approach of Henzinger et al. [HHC⁺23a]. Following [HHC⁺23a, DeB24], we assume the client performs 20 SCT audits each week, where each audit corresponds to retrieving a single bit using a PIR query over an 8 GB database. For DoublePIR [HHC⁺23a], the client needs to download a hint associated with the current state of the SCT database. We consider the setting where the client downloads the hint once each week and the case where the client downloads the hint (or a hint update) each day. The other schemes (Tiptoe [HDCZ23], HintlessPIR [LMRS23] and YPIR) do not require hints. We disregard the cost of the server preprocessing in these measurements (since this is a one-time cost that can be amortized across all clients). We measure the cost of running such a service based on current AWS costs: (\$0.09 per outbound GB and $\$1.5 \cdot 10^{-5}$/core-second; inbound communication is free) [HHC⁺23a]. For HintlessPIR, we do not have measurements for the server computation time, so we optimistically assume these are free for sake of comparison.

logs. The log operator responds with a *signed certificate timestamp* (SCT). The SCT is embedded within the certificate and represents a commitment from the log operator to include the certificate in its log within a certain timeframe (e.g., typically 24 hours). Whenever a client receives a certificate with an embedded SCT, the client can verify the SCT with the log server to confirm that the server has indeed received the associated certificate. The client may choose to reject certificates that do not contain a valid SCT. In turn, domain owners can check with log servers to obtain the certificates that have been issued for their domain, and identify any fraudulent certificates.

To defend against log operators falsifying SCTs (i.e., issuing an SCT but *not* depositing the certificate into the log), clients must regularly verify that (a subset of) the SCTs they receive from web servers are actually contained in the CT log. A naïve implementation of this would have the client simply reveal the SCTs they are auditing to the log operator, which in turn, reveals the client's browsing habits to the log operator. Several methods for privacy-preserving SCT auditing are based on matching hash prefixes [DeB24] or accessing the log server via an anonymizing proxy [DPRS21], but these approaches do not provide formal cryptographic guarantees to privacy.

Private SCT auditing. In this work, we consider the recent approach of Henzinger et al. [HHC⁺23a] on using PIR to construct a *private* SCT auditing protocol. In their approach, each log operator prepares a data structure (based on Bloom filters) representing the set of active SCTs in the log. To test whether a particular SCT is contained in the log, the client privately reads a single bit from this data structure using PIR.

To represent the set of 5 billion currently-active SCTs, the Henzinger et al. approach encodes the SCTs as a database of size 2^{36} bits (8 GB). Each SCT audit in turn corresponds to a single PIR query to this database. In [HHC⁺23a], the underlying PIR protocol is instantiated using DoublePIR.

Cost of private SCT auditing. A limitation of using DoublePIR for private SCT auditing is the need to download (and store) the large hint. SCT databases are constantly updated, with roughly 10 million certificates issued each day [Mer24]. To audit against the latest version of the log, the clients must first download the hint for the current log state.[11] To mitigate this, the approach taken in [HHC⁺23a] is to have clients download the hints on a *weekly*

[11] Instead of downloading the full hint each time, the client could download an update instead. The size of the update scales roughly with the number of rows in the database that has changed. Since the bits of the database correspond to the bits of a Bloom filter, updates will typically

basis and wait to test an SCT if its validity falls outside the time window associated with the current hint. While this reduces the communication costs of the protocol, it introduces delays in detecting malicious log behavior. In addition, the log server also needs to maintain multiple copies of the SCT database (in order to support PIR queries for hints issued at different points in time).

A PIR scheme with silent preprocessing avoids these deployment issues in private SCT auditing. Following [HHC+23a], we assume a client makes 10^4 TLS connections each week and performs two audits for a 1/1000-fraction of connections (this is also the setting Chrome uses [DeB24]). This corresponds to a client making 20 audits (i.e., PIR queries) over the course of a week. In Table 6, we report the monetary costs of the outbound communication[12] and the server computation based on current AWS pricing when instantiating the [HHC+23a] approach with DoublePIR, Tiptoe, HintlessPIR, and YPIR.

When the client downloads weekly hints, the DoublePIR approach [HHC+23a] has a weekly server cost of \$1822 per 1 million clients. Over 80% of this cost is from clients downloading the 16 MB hint. If we consider daily updates,[13] and have clients audit the most recent version of the log, the weekly cost of DoublePIR balloons to \$10,863 per 1 million clients.

A system based on YPIR would have a weekly cost of \$320 per 1 million clients. This is 5.7× cheaper than using DoublePIR with *weekly* updates and over 34× cheaper than using DoublePIR with daily updates. Most of YPIR's costs are from server computation, not communication. If we apply cross-client batching with a queue of size 4, YPIR's estimated weekly server cost drops to just \$267 per 1 million clients.

Scheme likes HintlessPIR, Tiptoe, and YPIR that do not require hints are unaffected by update frequency and can be better-suited for SCT auditing. The upload in YPIR is 3× larger than for HintlessPIR, and 23× smaller than for Tiptoe. Since AWS only charges for *outgoing* communication, and Tiptoe and HintlessPIR have larger responses than YPIR, they have substantially higher AWS costs (60× and 9×, respectively). Overall, the total communication required by YPIR is 1.4× lower than HintlessPIR, 28× lower than Tiptoe, and 4.3× lower than DoublePIR with daily updates. In fact, YPIR's total communication with daily updates is also *smaller* when compared to DoublePIR's total communication with *weekly* updates. In other words, even if the client amortizes the DoublePIR hint across multiple queries over the course of the week, YPIR still achieves smaller end-to-end communication costs. If we compare against the communication requirements of Chrome's k-anonymity-based approach for private SCT auditing [DeB24], which does not provide cryptographic privacy, the communication requirements of the YPIR-based approach are only 12.6× higher (concretely, the *weekly* communication costs are 2.3 MB for the k-anonymity approach, and 29 MB for YPIR).

5 Related Work

Private information retrieval was first introduced in [CGKS95]. The original construction considered the multi-server model where the database is replicated across multiple (non-colluding) servers. The reliance on non-colluding servers enables lightweight constructions (based on on symmetric cryptography or even no cryptography at all) [Yek07, Efr09, BIKO12, GI14, BGI16, HH19].

Kushilevitz and Ostrovsky [KO97] gave the first single-server PIR scheme based on additively homomorphic encryption. Subsequently, single-server PIR has been constructed from many number-theoretic assumptions [CMS99, GR05, DGI+19, CGH+21, BCM22]. Of particular note are the lattice-based constructions, which yield the most (concretely)-efficient constructions of single-server PIR [MBFK16, AS16, ACLS18, GH19, PT20, ALP+21, AYA+21, MCR21, MR23, MW22, DPC22, HHC+23a, HDCZ23, LMRS23].

Reducing the computational cost of PIR. There are many techniques to reduce the computational burden of PIR. Batch PIR [BIM00, IKOS04, GKL10, ACLS18, MR23] allows a single client to retrieve many elements from the database with low overhead relative to the cost of a single query. In *stateful PIR* [PPY18, KC21, MCR21, MZRA22], the client retrieves *private* state from the server in an offline phase in order to reduce the cost of the online phase. Recently, several works have also shown how to construct single-server PIR protocols with amortized sublinear

occur in random positions. Since the number of insertions each day is significantly larger than the number of rows, the size of a daily hint update is comparable to the size of the entire hint.

[12]AWS only charges for *outbound* communication.

[13]Since SCTs are promises to include certificates in logs within a 24-hour period, the maximum useful frequency of database updates is daily.

online computation [LP23, ZPSZ24, MSR23, WZLY23]. Notably, several of these constructions only rely on symmetric cryptography [ZPSZ24, MSR23, WZLY23] and can plausibly handle queries to extremely large databases (on the order of hundreds of GB). However, these systems have the limitation that the client has to stream the *entire* database in the offline phase, which may be infeasible for large databases.

In *doubly-efficient PIR* [CHR17, BIPW17], the server first locally preprocesses the database in a way that allows it to answer queries in sublinear time. Notably, no communication is needed in the offline phase. A recent breakthrough [LMW23] gives a construction of doubly-efficient PIR from the RLWE assumption; however, the concrete costs of this protocol still seem too high to be practically viable for realistic database sizes [OPPW23].

PIR variations. A number of recent works have also sought to strengthen PIR to provide security in the presence of *malicious* servers [WZ18, BKP22, CNC$^+$23, DT23, dCL24]. Other extensions of PIR include retrieving records by keyword instead of index [CGN98]; this case can be reduced to standard PIR [CGN98, ALP$^+$21, MK22, PSY23].

Acknowledgments

We thank Alexandra Henzinger for helpful pointers on the Tiptoe implementation. David J. Wu is supported in part by NSF CNS-2140975, CNS-2318701, a Microsoft Research Faculty Fellowship, a Google Research Scholar award, and a grant from Protocol Labs.

References

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

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

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

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

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

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

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

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

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

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

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

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

[BKMP12] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *IEEE S&P*, 2012.

[BKP22] Shany Ben-David, Yael Tauman Kalai, and Omer Paneth. Verifiable private information retrieval. In *TCC (3)*, volume 13749, 2022.

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

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

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

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

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

[CGB$^+$14] Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, Robert DeLine, John C. Platt, James F. Terwilliger, and John Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4), 2014.

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

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

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

[CGN98] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, 1998.

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

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

[CMS16] Alvin Cheung, Samuel Madden, and Armando Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). *ACM Trans. Database Syst.*, 41(2), 2016.

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

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

[DeB24] Joe DeBlasio. Opt-out SCT auditing in chrome. https://docs.google.com/document/d/16G-Q7iN3kB46GSW5b-sfH5MO3nKSYyEb77YsM7TMZGE/edit, January 2024.

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

[DPC22] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *Cryptology ePrint Archive*, 2022.

[DPRS21] Rasmus Dahlberg, Tobias Pulls, Tom Ritter, and Paul Syverson. Privacy-preserving & incrementally-deployable support for certificate transparency in tor. *Proc. Priv. Enhancing Technol.*, 2021(2), 2021.

[DT23] Marian Dietz and Stefano Tessaro. Fully malicious authenticated PIR. *IACR Cryptol. ePrint Arch.*, 2023.

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

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

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

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

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

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

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

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

[GLM16] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *ACM CCS*, 2016.

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

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

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

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

[HHC+23a] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *USENIX Security Symposium*, 2023.

[HHC+23b] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval, 2023. https://github.com/ahenzinger/simplepir.

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

[Int23] Intel® intrinsics guide v3.6.7. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html, 2023. © Intel Corporation.

[Jue01] Ari Juels. Targeted advertising ... and privacy too. In *CT-RSA*, 2001.

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

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

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

[Lau14] Ben Laurie. Certificate transparency. *Commun. ACM*, 57(10), 2014.

[LLK13] Ben Laurie, Adam Langley, and Emilia Käsper. Certificate transparency. *RFC*, 6962, 2013.

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

[LMRS23] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz. Hintless single-server private information retrieval. *IACR Cryptol. ePrint Arch.*, 2023.

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

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

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In *CRYPTO*, 2023.

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

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

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

[McC95] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, 1995.

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

[Mer24] Merkle town. `https://ct.cloudflare.com`, January 2024. Cloudflare, Inc.

[MK22] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: single-round keyword PIR via constant-weight equality operators. In *USENIX Security Symposium*, 2022.

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

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

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

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

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

[MZRA22] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. Incremental offline/online PIR (extended version). In *USENIX Security*, 2022.

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

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

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

[PSY23] Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don't be dense: Efficient keyword PIR for sparse databases. In *USENIX Security Symposium*, 2023.

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

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

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

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

[WYG+17] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.

[WZ18] Xingfeng Wang and Liang Zhao. Verifiable single-server private information retrieval. In *ICICS*, volume 11149, 2018.

[WZLY23] Yinghao Wang, Jiawen Zhang, Jian Liu, and Xiaohu Yang. Crust: Verifiable and efficient private information retrieval with sublinear online time. *IACR Cryptol. ePrint Arch.*, 2023.

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

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

A Lattice Algorithms

In this section, we recall some additional lattice preliminaries and algorithms that we use in our construction and analysis. Throughout, we write $R = \mathbb{Z}[x]/(x^d + 1)$.

Automorphisms over R_q. Let $R = \mathbb{Z}[x]/(x^d + 1)$ where d is a power of two. For a positive integer $\ell \in \mathbb{N}$, we let $\tau_\ell \colon R \to R$ denote the ring automorphism that maps $p(x) \mapsto p(x^\ell)$. We define $\tau_\ell \colon R_q \to R_q$ analogously, and for ease of notation, denote both automorphisms by the mapping $\tau_\ell(\cdot)$. We extend τ_ℓ to vector-valued and matrix-valued inputs via component-wise application of τ_ℓ to the entries of the vector or matrix.

Automorphisms on RLWE encodings. A number of works have shown how to *homomorphically* evaluate automorphisms on RLWE encodings [GHS12a, BGV12]. We recall the main algorithms here. Some of our presentation is adapted from [MW22]:

Construction A.1 (Automorphisms on RLWE Encodings [GHS12a, BGV12, adapted])**.** Let λ be a security parameter and $d = d(\lambda)$, $q = q(\lambda)$ be lattice parameters where $d = 2^\ell$ is a power of two. Let $R = \mathbb{Z}[x]/(x^d + 1)$ and $\chi = \chi(\lambda)$ be an error distribution over R.

- $\mathsf{AutomorphSetup}(1^\lambda, s, \tau, z)$: On input a secret key $s \in R_q$, 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 $\mathbf{a} \xleftarrow{\mathrm{R}} R_q^t$ and $e \leftarrow \chi^t$. Output

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

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

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

Theorem A.2 (Homomorphic Evaluation of Automorphisms [GHS12a, BGV12, MW22, adapted])**.** *For a positive integer $\ell \in \mathbb{N}$, let $\tau_\ell \colon R_q \to R_q$ be the automorphism $r(x) \mapsto r(x^\ell)$ and $z \in \mathbb{N}$ be a decomposition base. Suppose $[-s \mid 1] \cdot \mathbf{c} = \mu + e$ for some $s \in R_q$, $\mathbf{c} \in R_q^2$, $\mu \in R_q$ and $e \in R$. Let $\mathbf{W}_\tau \leftarrow \mathsf{AutomorphSetup}(1^\lambda, s, \tau_\ell, z)$ and $\mathbf{c}' \leftarrow \mathsf{Automorph}(\mathbf{W}_\tau, \mathbf{c}, \tau_\ell, z)$. Suppose e is subgaussian with parameter σ and the error distribution χ in Construction A.1 is subgaussian with parameter σ_χ. Then, under the independent heuristic, $[-s \mid 1] \cdot \mathbf{c}' = \tau_\ell(\mu) + e'$, e' is subgaussian with parameter σ', $(\sigma')^2 \le \sigma^2 + tdz^2\sigma_\chi^2/4$, and $t = \lfloor \log_z q \rfloor + 1$.*

The CDKS transformation. We now review the LWE-to-ring-LWE packing transformation introduced by Chen, Dai, Kim, and Song [CDKS21]. Specifically, let $d = 2^\ell$ be the lattice dimension for the LWE encoding as well as the degree of the ring for ring LWE encodings. The [CDKS21] algorithm takes as input d independent LWE encodings of scalars $\mu_1, \ldots, \mu_d$ and outputs an RLWE encoding of the polynomial $\sum_{i \in [d]} \mu_i x^{i-1}$. The key insight underlying the [CDKS21] transformation is the fact that over $R = \mathbb{Z}[x]/(x^d + 1)$, it holds that

$$\sum_{j \in [\ell]} \tau_{2^{\ell-j+1}+1}(x^i) = \begin{cases} d & i = 0 \\ 0 & 0 < i < d, \end{cases}$$

where $\tau_\ell \colon R \to R$ is the ring automorphism $f(x) \mapsto f(x^\ell)$. We describe the [CDKS21] packing transformation below. In addition to the main $\mathsf{CDKS.Setup}$ and $\mathsf{CDKS.Pack}$ algorithms, the $\mathsf{CDKS.Pack}$ algorithm additionally relies on an internal helper algorithm $\mathsf{CDKS.PackHelper}$. We describe the algorithm below and then follow with the proof the security property from Theorem 2.4.

Construction A.3 (LWE-to-RLWE Packing [CDKS21])**.** Let λ be a security parameter, $d = d(\lambda)$ be a power-of-two (i.e., $d = 2^\ell$), and $q = q(\lambda)$ be a modulus. Let $R = \mathbb{Z}[x]/(x^d + 1)$.

- $\mathsf{CDKS.Setup}(1^\lambda, s, z)$: On input the security parameter λ, a secret key $s \in R_q$, and a decomposition base $z \in \mathbb{N}$, the setup algorithm computes $\mathbf{W}_i \leftarrow \mathsf{AutomorphSetup}(1^\lambda, s, \tau_{2^i+1}, z)$ for each $i \in [\ell]$. It outputs the packing key $\mathsf{pk} = (\mathbf{W}_1, \ldots, \mathbf{W}_\ell)$.

- $\mathsf{CDKS.PackHelper}(\mathsf{pk}, (\mathbf{c}_0, \ldots, \mathbf{c}_{2^t-1}))$: On input the packing key $\mathsf{pk} = (\mathbf{W}_1, \ldots, \mathbf{W}_\ell)$ and a collection of 2^t encodings $\mathbf{c}_0, \ldots, \mathbf{c}_{2^t-1} \in R_q^2$ where $t \le \ell$, the recursive helper algorithm proceeds as follows:
  - If $t = 1$, return $\mathbf{c}_0$.

– If $t > 1$, then compute

$$\begin{aligned} \mathbf{c}_{\text{even}} &\leftarrow \mathsf{CDKS.PackHelper}\big(\mathsf{pk}, (\mathbf{c}_0, \mathbf{c}_2, \dots, \mathbf{c}_{2^t-2})\big) \\ \mathbf{c}_{\text{odd}} &\leftarrow \mathsf{CDKS.PackHelper}\big(\mathsf{pk}, (\mathbf{c}_1, \mathbf{c}_3, \dots, \mathbf{c}_{2^t-1})\big). \end{aligned}$$

Let $\mathbf{c}_1 \leftarrow \mathbf{c}_{\text{even}} + x^{d/2^t} \cdot \mathbf{c}_{\text{odd}}$ and $\mathbf{c}_2 \leftarrow \mathbf{c}_{\text{even}} - x^{d/2^t} \cdot \mathbf{c}_{\text{odd}}$. Output $\mathbf{c}_1 + \mathsf{Automorph}(\mathbf{W}_t, \mathbf{c}_2, \tau_{2^t+1}, z) \in R_q^2$.

- $\mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C})$: On input the packing key $\mathsf{pk} = (\mathbf{W}_1, \dots, \mathbf{W}_\ell)$ and a matrix $\mathbf{C} \in \mathbb{Z}_q^{(d+1) \times d}$ of d LWE encodings, the packing algorithm first parses the encodings as

$$\mathbf{C} = \begin{bmatrix} \mathbf{a}_0 & \cdots & \mathbf{a}_{d-1} \\ b_0 & \cdots & b_{d-1} \end{bmatrix},$$

where $\mathbf{a}_i \in \mathbb{Z}_q^d$ and $b_i \in \mathbb{Z}_q$ for all $i \in [0, d-1]$. Then, for each $i \in [0, d-1]$, let $\tilde{a}_i = \sum_{j \in [0, d-1]} a_{i,j} x^{-j} \in R_q$, where $\mathbf{a}_i = [a_{i,0}, \dots, a_{i,d-1}]^\intercal$. Let $\mathbf{c}_i = [\tilde{a}_i \mid b_i]^\intercal \in R_q^2$. Finally, output $\mathsf{CDKS.PackHelper}(\mathsf{pk}, (\mathbf{c}_0, \dots, \mathbf{c}_{d-1}))$.

Correctness of Construction A.3. We refer to [CDKS21, Appendix A.3] for a proof that Construction A.3 satisfies the correctness property stated in Theorem 2.4. In Appendix A.1, we show that Construction A.3 also satisfies the required security property assuming a key-dependent pseudorandomness property of RLWE encodings.

A.1 Security of Construction A.3

Here, we provide a short proof that the pseudorandomness security property in Theorem 2.4 holds for Construction A.3. This follows assuming a key-dependent pseudorandomness property on RLWE encodings (namely, that $(a, sa + e)$ even given fresh RLWE encodings of functions of the secret key s). This can also be stated as a "circular security" property on RLWE encodings. We state the specific assumption we use below:

Definition A.4 (Key-Dependent Pseudorandomness for RLWE Encodings)**.** Let λ be a security parameter, $d = d(\lambda)$ be a power-of-two, $m = m(\lambda)$ be the number of samples, $q = q(\lambda)$ be an encoding modulus, and $\chi = \chi(\lambda)$ be an error distribution over the ring $R = \mathbb{Z}[x]/(x^d + 1)$. Let $\mathcal{F}$ be an efficiently-computable set of functions from R_q to R_q. For a bit $b \in \{0, 1\}$ and an adversary $\mathcal{A}$, let

$$W_b := \Pr\left[\mathcal{A}^{O(\cdot)}(1^\lambda, \mathbf{a}, \mathbf{t}_b) : \begin{array}{c} s \leftarrow \chi, \mathbf{a} \xleftarrow{\text{R}} \mathbb{R}_q^m, \mathbf{e} \leftarrow \chi^m \\ \mathbf{t}_0 = s\mathbf{a} + \mathbf{e}, \mathbf{t}_1 \xleftarrow{\text{R}} R_q^m \end{array}\right],$$

where the oracle O takes as input a function $f \in \mathcal{F}$, and outputs $(a, sa + e + f(s))$, where $a \xleftarrow{\text{R}} R_q$ and $e \leftarrow \chi$.

To show that Construction A.3 satisfies the security property in Theorem 2.4, we require key-dependent pseudorandomness with respect to the set of (scaled) automorphisms $\mathcal{F}_{\text{auto}}$:

$$\mathcal{F}_{\text{auto}} = \{r \mapsto k \cdot \tau_\ell(r) : k \in \mathbb{Z}_q, \ell \in \mathbb{N}\}, \tag{A.1}$$

where $\tau_\ell \colon R \to R$ is the automorphism $r(x) \mapsto r(x^\ell)$. We give the formal statement and analysis below:

Lemma A.5 (Security of Construction A.3)**.** *Let λ be a security parameter, $d = d(\lambda)$ be a power-of-two, $q = q(\lambda)$ be an encoding modulus, and $\chi = \chi(\lambda)$ be an error distribution over the ring $R = \mathbb{Z}[x]/(x^d + 1)$. Let $\mathcal{F}_{\text{auto}}$ be the family of scaled automorphisms on R_q defined in Eq. (A.1). If RLWE encodings satisfy key-dependent pseudorandomness (for m samples) with respect to $\mathcal{F}_{\text{auto}}$, then Construction A.3 is pseudorandom given the public key (also for m samples).*

Proof. Take any decomposition base $z \le q$ and write $d = 2^\ell$. Suppose there exists an efficient adversary $\mathcal{A}$ that can break pseudorandomness of Construction A.3. We use $\mathcal{A}$ to construct an efficient adversary $\mathcal{B}$ that breaks key-dependent pseudorandomness of RLWE encodings with respect to $\mathcal{F}_{\text{auto}}$. Algorithm $\mathcal{B}$ operates as follows:

1. At the beginning of the game, algorithm $\mathcal{B}$ receives the challenge $\mathbf{a}, \mathbf{t} \in R_q^m$.

2. Let $t = \lfloor \log_z q \rfloor + 1$. Algorithm $\mathcal{B}$ constructs the public key pk as follows. For each $i \in [\ell]$ and each $j \in [t]$, algorithm $\mathcal{B}$ queries the oracle on the function $f_{i,j}(r) := -z^{j-1} \cdot \tau_{2^i+1}(r)$. It obtains an encoding $(a_{i,j}, b_{i,j}) \in R_q^2$. Algorithm $\mathcal{B}$ now sets
$$\mathbf{W}_i = \begin{bmatrix} a_{i,1} & a_{i,2} & \cdots & a_{i,t} \\ b_{i,1} & b_{i,2} & \cdots & b_{i,t} \end{bmatrix} \in R_q^{2 \times t}.$$
Finally, it sets $\mathsf{pk} = (\mathbf{W}_1, \ldots, \mathbf{W}_\ell)$ and gives pk to algorithm $\mathcal{A}$.

3. Algorithm $\mathcal{B}$ gives $(1^\lambda, \mathsf{pk}, \mathbf{a}, \mathbf{t})$ to $\mathcal{A}$. Algorithm $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$, which algorithm $\mathcal{B}$ also outputs.

Let $s \in R_q$ be the secret key the challenger samples in the key-dependent pseudorandomness game. By construction, the challenger samples $\mathbf{a} \xleftarrow{\text{R}} R_q^m$ and either sets $\mathbf{t} = s\mathbf{a} + \mathbf{e}$ where $s \leftarrow \chi$ and $\mathbf{e} \leftarrow \chi^m$ or samples $\mathbf{t} \xleftarrow{\text{R}} R_q^m$. The former corresponds to the pseudorandom distribution for $\mathcal{A}$ while the latter corresponds to the truly random distribution for $\mathcal{A}$. Thus, it suffices to argue that algorithm $\mathcal{B}$ correctly constructs the public key pk. By definition, for all $i \in [\ell]$ and $j \in [t]$, we have that $a_{i,j} \xleftarrow{\text{R}} R_q$ and $b_{i,j} = sa_{i,j} + e_{i,j} - z^{j-1} \cdot \tau_{2^i+1}(s)$, where $a_{i,j} \xleftarrow{\text{R}} R_q$ and $e_{i,j} \leftarrow \chi$. If we define $\mathbf{a}_i^{\mathsf{T}} = [a_{i,1} \mid \cdots \mid a_{i,t}]$, $\mathbf{e}_i^{\mathsf{T}} = [e_{i,1} \mid \cdots \mid e_{i,t}]$, then
$$\mathbf{W}_i = \begin{bmatrix} \mathbf{a}_i^{\mathsf{T}} \\ s\mathbf{a}_i^{\mathsf{T}} + \mathbf{e}_i^{\mathsf{T}} - \tau_{2^i+1}(s) \cdot \mathbf{g}_z^{\mathsf{T}} \end{bmatrix},$$
which exactly coincides with the output distribution of $\mathsf{CDKS.Pack}(1^\lambda, s, z)$. Hence, algorithm $\mathcal{B}$ perfectly simulates the public key for $\mathcal{A}$, and we can conclude that the advantage of $\mathcal{B}$ in the key-dependent pseudorandomness security game is precisely the advantage of $\mathcal{A}$ in the pseudorandom game. The claim follows. □

Remark A.6 (Number of Samples)**.** In the proof of Lemma A.5, algorithm $\mathcal{B}$ makes ℓt queries to its oracle and learns ℓt *additional* RLWE encodings (on top of the m encodings in the challenge). Thus, if we require (CDKS.Setup, CDKS.Pack) satisfy pseudorandomness for m samples, then the reduction algorithm in $\mathcal{B}$ obtains $m + \ell t$ RLWE encodings. For this reason, we only consider parameter instantiations for Construction A.3 where the $\mathsf{RLWE}_{d,m+\ell t,q,\chi}$ plausibly holds.

Remark A.7 (Shorter Query Keys)**.** The public parameters in Construction A.3 consist of $\ell = \log d$ key-switching matrices $\mathbf{W}_1, \ldots, \mathbf{W}_\ell$. Each key-switching matrix (Construction A.1) is a 2-by-t matrix of ring elements (where $t = \lfloor \log_z q \rfloor + 1$ and z is the decomposition base). Moreover, the first row of each $\mathbf{W}_i$ is a uniform random element over R_q^t. Thus, similar to Remark 3.2, we can "compress" these random elements using a short seed for a pseudorandom generator (and appeal to the random oracle heuristic). This reduces the size of the packing parameters in Construction A.3 by a factor of 2.

Remark A.8 (Reducing Online Packing Cost)**.** In Construction A.3, the CDKS.Pack procedure needs to evaluate $\mathsf{Automorph}(\mathbf{W}, \mathbf{c}_2, \tau_{2^t+1}, z)$. Evaluating these automorphisms are the most expensive part of the packing procedure since they involve gadget decomposition and evaluating multiple NTTs. In YPIR (Construction 3.1), we can reduce the online costs of packing by moving part of the computation to the offline phase. Specifically, we observe that the first κd_1 input LWE encodings to CDKS.Pack in the Answer algorithm of Construction 3.1 have a pseudorandom component that only depends on the database contents and *not* the query (i.e., this is the precomputed $\mathbf{H}_2$ component in Eq. (3.2)). Therefore, the server can precompute the gadget decompositions and NTTs on the fixed query-independent encoding components during setup (i.e., in DBSetup) rather than during the online Answer algorithm. By moving some of the packing transformation to the preprocessing step, we observe that we can reduce the online cost of CDKS.Pack by about 25%.

B Correctness and Security of YPIR

In this section, we provide the correctness and security analysis for the YPIR protocol (Construction 3.1).

B.1 Correctness (Proof of Theorem 3.4)

Take any security parameter λ, any database $\mathbf{D} \in \mathbb{Z}_N^{\ell_1 \times \ell_2}$ (with records $D_{i_1,i_2} \in \mathbb{Z}_N$ where $i_1 \in [\ell_1]$ and $i_2 \in [\ell_2]$), and any index $\mathsf{idx} = (i_1, i_2) \in [\ell_1] \times [\ell_2]$. Let $(\mathsf{pp}, \mathsf{dbp}) \leftarrow \mathsf{DBSetup}(1^\lambda, \mathbf{D})$, $(\mathsf{q}, \mathsf{qk}) \leftarrow \mathsf{Query}(\mathsf{pp}, \mathsf{idx})$, and $\mathsf{resp} \leftarrow \mathsf{Answer}(\mathsf{dbp}, \mathsf{q})$. These components are constructed as follows:

- By construction of DBSetup, this means $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$, where $\mathbf{a}_1 \in R_{d_1,q_1}^{m_1}$ and $\mathbf{a}_2 \in R_{d_2,q_2}^{m_2}$, and $\mathsf{dbp} = (1^\lambda, \mathbf{D}, \mathbf{H}_1, \mathbf{H}_2)$, where $\mathbf{H}_1 = \mathbf{G}_{d_1,p}^{-1}(\mathbf{A}_1\mathbf{D})$, $\mathbf{H}_2 = \mathbf{A}_2 \cdot \mathbf{H}_1^{\mathsf{T}}$, $\mathbf{A}_1 = \mathsf{NCyclicMat}(\mathbf{a}_1^{\mathsf{T}})$, and $\mathbf{A}_2 = \mathsf{NCyclicMat}(\mathbf{a}_2^{\mathsf{T}})$.
- The query-generation algorithm samples $s_j \leftarrow \chi_j$, $\mathbf{e}_j \leftarrow \chi_j^{m_j}$, and sets $\mathbf{c}_j = \mathsf{Coeffs}(s_j\mathbf{a}_j + \mathbf{e}_j + \Delta_j \boldsymbol{\mu}_j) \in \mathbb{Z}_{q_j}^{\ell_j}$, where $\boldsymbol{\mu}_j = x^{\beta_j}\mathbf{u}_{\alpha_j}$, $i_j = \alpha_j d_j + \beta_j$, $\alpha_j \in [m_j]$ and $\beta_j \in [d_j]$ for $j \in \{1,2\}$. It also samples $\mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s_2, z)$ and sets $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$ and $\mathsf{qk} = (s_1, s_2)$.
- The answer algorithm first computes the encoding $\mathbf{C} \in \mathbb{Z}_{q_2}^{(d_2+1)\times\kappa(d_1+1)}$ according to Eq. (3.2). It then packs the encodings together $\tilde{\mathbf{c}}_i \leftarrow \mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C}_i) \in R_{d_2,q_2}^2$ where $\mathbf{C}_1, \ldots, \mathbf{C}_\rho$ are the (padded) blocks of the matrix $\mathbf{C}$. Finally, the response $\mathsf{resp} = ((c_{1,1}, c_{1,2}), \ldots, (c_{\rho,1}, c_{\rho,2}))$ is obtained by applying modulus switching to $\mathbf{c}_1, \ldots, \mathbf{c}_\rho$.

We now consider the output of $\mathsf{Extract}(\mathsf{qk}, \mathsf{resp})$. Using Eq. (2.2), we write the query encodings $\mathbf{c}_j$ for $j \in \{1,2\}$ as

$$\begin{aligned}\mathbf{c}_j^{\mathsf{T}} = \mathsf{Coeffs}(s_j\mathbf{a}_j + \mathbf{e}_j + \Delta_j\boldsymbol{\mu}_j)^{\mathsf{T}} &= \mathsf{Coeffs}(s_j\mathbf{a}_j)^{\mathsf{T}} + \mathsf{Coeffs}(\mathbf{e}_j)^{\mathsf{T}} + \mathsf{Coeffs}(\Delta_j\boldsymbol{\mu}_j)^{\mathsf{T}} \\ &= \tilde{\mathbf{s}}_j^{\mathsf{T}} \cdot \mathsf{NCyclicMat}(\mathbf{a}_j^{\mathsf{T}}) + \tilde{\mathbf{e}}_j^{\mathsf{T}} + \Delta_j \mathbf{u}_{\alpha_j d_j + \beta_j}^{\mathsf{T}} \\ &= \tilde{\mathbf{s}}_j^{\mathsf{T}}\mathbf{A}_j + \tilde{\mathbf{e}}_j^{\mathsf{T}} + \Delta_j\mathbf{u}_{i_j}^{\mathsf{T}} \in \mathbb{Z}_{q_j}^{\ell_j}\end{aligned}$$

where $\tilde{\mathbf{s}}_j = \mathsf{Coeffs}(s_j) \in \mathbb{Z}_{q_j}^{d_j}$ and $\tilde{\mathbf{e}}_j = \mathsf{Coeffs}(\mathbf{e}_j) \in \mathbb{Z}_{q_j}^{\ell_j}$. Let $\delta = d_2^{-1} \bmod q_2$. Since $\mathbf{H}_2 = \mathbf{A}_2 \cdot \mathbf{H}_1^{\mathsf{T}}$, we can write Eq. (3.2) as

$$\mathbf{C} = \delta \begin{bmatrix} \mathbf{H}_2 & \mathbf{A}_2\mathbf{T}^{\mathsf{T}} \\ \mathbf{c}_2^{\mathsf{T}}\mathbf{H}_1^{\mathsf{T}} & \mathbf{c}_2^{\mathsf{T}}\mathbf{T}^{\mathsf{T}} \end{bmatrix} = \delta \begin{bmatrix} \mathbf{A}_2 \\ \mathbf{c}_2^{\mathsf{T}} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{H}_1^{\mathsf{T}} \mid \mathbf{T}^{\mathsf{T}} \end{bmatrix} = \delta \begin{bmatrix} \mathbf{A}_2 \\ \tilde{\mathbf{s}}_2^{\mathsf{T}}\mathbf{A}_2 + \tilde{\mathbf{e}}_2^{\mathsf{T}} + \Delta_2\mathbf{u}_{i_2}^{\mathsf{T}} \end{bmatrix} \cdot \begin{bmatrix} \mathbf{H}_1^{\mathsf{T}} \mid \mathbf{T}^{\mathsf{T}} \end{bmatrix}.$$

This means that

$$[-\tilde{\mathbf{s}}_2^{\mathsf{T}} \mid 1] \cdot \mathbf{C} = \delta(\tilde{\mathbf{e}}_2^{\mathsf{T}} + \Delta_2\mathbf{u}_{i_2}^{\mathsf{T}})[\mathbf{H}_1^{\mathsf{T}} \mid \mathbf{T}^{\mathsf{T}}] \in \mathbb{Z}_{q_2}^{\kappa(d_1+1)}.$$

Define $\boldsymbol{\alpha}_1, \ldots, \boldsymbol{\alpha}_\rho \in \mathbb{Z}_{q_2}^{d_2}$ where

$$[\boldsymbol{\alpha}_1^{\mathsf{T}} \mid \cdots \mid \boldsymbol{\alpha}_\rho^{\mathsf{T}}] = [-\tilde{\mathbf{s}}_2^{\mathsf{T}} \mid 1] \cdot [\mathbf{C}_1 \mid \cdots \mid \mathbf{C}_\rho] \in \mathbb{Z}_q^{d_2\rho}. \tag{B.1}$$

Since $\Delta_2 = \lfloor q_2/p \rfloor$, this in particular means that

$$\begin{bmatrix} \boldsymbol{\alpha}_1 \\ \vdots \\ \boldsymbol{\alpha}_\rho \end{bmatrix} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \\ \mathbf{0}^{d_2\rho - \kappa(d_1+1)} \end{bmatrix} \cdot \delta(\tilde{\mathbf{e}}_2 + \lfloor q_2/p \rfloor \mathbf{u}_{i_2}) = \delta \lfloor q_2/p \rfloor \begin{bmatrix} \boldsymbol{\alpha}_{1,1} \\ \vdots \\ \boldsymbol{\alpha}_{\rho,1} \end{bmatrix} + \delta \begin{bmatrix} \boldsymbol{\alpha}_{1,2} \\ \vdots \\ \boldsymbol{\alpha}_{\rho,2} \end{bmatrix},$$

where $\boldsymbol{\alpha}_{i,1}, \boldsymbol{\alpha}_{i,2} \in \mathbb{Z}_{q_2}^{d_2}$ and

$$\begin{bmatrix} \boldsymbol{\alpha}_{1,1} \\ \vdots \\ \boldsymbol{\alpha}_{\rho,1} \end{bmatrix} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \\ \mathbf{0}^{d_2\rho - \kappa(d_1+1)} \end{bmatrix} \mathbf{u}_{i_2} \quad \text{and} \quad \begin{bmatrix} \boldsymbol{\alpha}_{1,2} \\ \vdots \\ \boldsymbol{\alpha}_{\rho,2} \end{bmatrix} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \\ \mathbf{0}^{d_2\rho - \kappa(d_1+1)} \end{bmatrix} \tilde{\mathbf{e}}_2.$$

Since $\|\mathbf{H}_1\|_\infty, \|\mathbf{T}\|_\infty \le p/2$ and $\mathbf{u}_{i_2}$ is a unit vector, it holds that $\|\boldsymbol{\alpha}_{i,1}\|_\infty \le p/2$ for all $i \in [\rho]$. Since $\tilde{\mathbf{e}}_2$ is subgaussian with parameter σ_2, it follows that each $f_{i,2}$ is subgaussian with parameter $\sigma_{\mathsf{scan}} \le \sqrt{\ell_2}(p/2)\sigma_2$. Now, for each $i \in [\rho]$, let $f_{i,1}, f_{i,2} \in R_{d_2,q_2}$ where $\mathsf{Coeffs}(f_{i,1}) = \boldsymbol{\alpha}_{i,1}$ and $\mathsf{Coeffs}(f_{i,2}) = \boldsymbol{\alpha}_{i,2}$. Let $f_i = \delta(\lfloor q_2/p \rfloor f_{i,1} + f_{i,2}) \in R_{d_2,q_2}$. By construction, observe that $\mathsf{Coeffs}(f_i) = \boldsymbol{\alpha}_i$. Since $\tilde{\mathbf{c}}_i = \mathsf{CDKS.Pack}(\mathsf{pk}, \mathbf{C}_i)$, $\tilde{\mathbf{s}}_2 = \mathsf{Coeffs}(s_2)$ and Eq. (B.1) holds, we appeal to Theorem 2.4 and the fact that $\delta d_2 = 1 \bmod q_2$ to conclude that

$$[-s_2 \mid 1] \cdot \tilde{\mathbf{c}}_i = d_2 f_i + e_{\mathsf{pack},i} = d_2\delta(\lfloor q_2/p \rfloor f_{i,1} + f_{i,2}) + e_{\mathsf{pack},i} = \lfloor q_2/p \rfloor f_{i,1} + f_{i,2} + e_{\mathsf{pack},i} \in R_{d_2,q_2},$$

where each $e_{\mathsf{pack},i} \in R_{d_2}$ is subgaussian with parameter σ_{pack} and $\sigma^2_{\mathsf{pack}} \le \frac{1}{3}(d_2^2 - 1)(t d_2 z^2 \sigma_2^2/4)$ and $t = \lfloor \log_z q_2 \rfloor + 1 \in R_{d_2,q_2}$. Since $(c_{i,1}, c_{i,2}) = \mathsf{ModReduce}_{\tilde{q}_{2,1},\tilde{q}_{2,2}}(\tilde{\mathbf{c}}_i)$, we appeal to Lemma 2.5 to conclude that

$$v_i' = \lfloor -s_2 c_{i,1} \rceil_{\tilde{q}_{2,1},\tilde{q}_{2,2}} + c_{i,2} = \lfloor \tilde{q}_{2,2}/p \rfloor f_{i,1} + e_{\mathsf{double},i} \in R_{d_2,\tilde{q}_{2,2}},$$

where $e_{\mathsf{double},i} = e_{\mathsf{double},i,1} + e_{\mathsf{double},i,2}$ and $\|e_{\mathsf{double},1}\|_\infty \le \frac{1}{2}\big(2 + (\tilde{q}_{2,2} \bmod p) + (\tilde{q}_{2,2}/q_2)(q_2 \bmod p)\big)$ and the components of $e_{\mathsf{double},i,2}$ are subgaussian with parameter σ_{double} where (assuming the independence heuristic),

$$\begin{aligned}\sigma^2_{\mathsf{double}} &\le (\tilde{q}_{2,2}/\tilde{q}_{2,1})^2 d_2 \sigma_2^2/4 + (\tilde{q}_{2,2}/q_2)^2(\sigma^2_{\mathsf{scan}} + \sigma^2_{\mathsf{pack}}) \\ &\le (\tilde{q}_{2,2}/\tilde{q}_{2,1})^2 d_2 \sigma_2^2/4 + (\tilde{q}_{2,2}/q_2)^2(\sigma_2^2/4)(\ell_2 p^2 + (d_2^2-1)(t d_2 z^2)/3).\end{aligned}$$

By Lemma 2.3, if $|e_{\mathsf{double},i}| \le \frac{\tilde{q}_{2,2}}{2p} - (\tilde{q}_{2,2} \bmod p)$, then $v_i = \lfloor v_i' \rceil_{\tilde{q}_{2,2},p} = f_{i,1}$. Let

$$\tau_{\mathsf{double}} = \frac{\tilde{q}_{2,2}}{2p} - (\tilde{q}_{2,2} \bmod p) - \frac{1}{2}\left(2 + (\tilde{q}_{2,2} \bmod p) + (\tilde{q}_{2,2}/q_2)(q_2 \bmod p)\right).$$

Then, by a subgaussian tail bound and a union bound,

$$\Pr[\forall i \in [\rho] : v_i = f_{i,1}] \ge 1 - 2 d_2 \rho \exp(-\pi \tau^2_{\mathsf{double}}/\sigma^2_{\mathsf{double}}). \tag{B.2}$$

Suppose for all $i \in [\rho]$, $v_i = f_{i,1}$. Then,

$$\bar{\mathbf{w}} = \begin{bmatrix} \mathsf{Coeffs}(v_1) \\ \vdots \\ \mathsf{Coeffs}(v_\rho) \end{bmatrix} = \begin{bmatrix} \boldsymbol{\alpha}_{1,1} \\ \vdots \\ \boldsymbol{\alpha}_{1,\rho} \end{bmatrix} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \\ \mathbf{0}^{d_2\rho - \kappa(d_1+1)} \end{bmatrix} \mathbf{u}_{i_2} \in \mathbb{Z}_p^{d_2\rho}.$$

In particular, this means that

$$\mathbf{w} = \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \end{bmatrix} \mathbf{u}_{i_2} \in \mathbb{Z}_p^{\kappa(d_1+1)}. \tag{B.3}$$

Suppose Eq. (B.3) holds. Since $\mathbf{H}_1 = \mathbf{G}^{-1}_{d_1,p}(\lfloor \mathbf{A}_1\mathbf{D} \rceil_{q_1,\tilde{q}_1})$, $\mathbf{T} = \mathbf{G}^{-1}_{1,p}(\lfloor \mathbf{c}_1^{\mathsf{T}}\mathbf{D} \rceil_{q_1,\tilde{q}_1})$, and $\mathbf{c}_1^{\mathsf{T}} = \tilde{\mathbf{s}}_1^{\mathsf{T}}\mathbf{A}_1 + \tilde{\mathbf{e}}_1^{\mathsf{T}} + \Delta_1 \mathbf{u}_{i_1}^{\mathsf{T}}$, this means

$$\begin{aligned}\mathbf{G}_{d_1+1,p}\mathbf{w} = \begin{bmatrix} \mathbf{G}_{d_1,p} & \mathbf{0}^{d_1\times\kappa} \\ \mathbf{0}^{1\times\kappa d_1} & \mathbf{g}_p^{\mathsf{T}} \end{bmatrix} \begin{bmatrix} \mathbf{H}_1 \\ \mathbf{T} \end{bmatrix} \mathbf{u}_{i_2} &= \begin{bmatrix} \lfloor \mathbf{A}_1\mathbf{D} \rceil_{q_1,\tilde{q}_1} \\ \lfloor \mathbf{c}_1^{\mathsf{T}}\mathbf{D} \rceil_{q_1,\tilde{q}_1} \end{bmatrix} \mathbf{u}_{i_2} \\ &= \mathsf{ModReduce}_{\tilde{q}_1}\left(\begin{bmatrix} \mathbf{A}_1\mathbf{D} \\ \tilde{\mathbf{s}}_1^{\mathsf{T}}\mathbf{A}_1\mathbf{D} + \tilde{\mathbf{e}}_1^{\mathsf{T}}\mathbf{D} + \Delta_1\mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D} \end{bmatrix}\right) \cdot \mathbf{u}_{i_2} \in \mathbb{Z}_{\tilde{q}_1}^{d_1+1}.\end{aligned}$$

Since $\Delta_1 = \lfloor q_1/N \rfloor$, we have

$$[-\tilde{\mathbf{s}}_1^{\mathsf{T}} \mid 1] \begin{bmatrix} \mathbf{A}_1\mathbf{D} \\ \tilde{\mathbf{s}}_1^{\mathsf{T}}\mathbf{A}_1\mathbf{D} + \tilde{\mathbf{e}}_1^{\mathsf{T}}\mathbf{D} + \Delta_1\mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D} \end{bmatrix} = \lfloor q_1/N \rfloor\, \mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D} + \tilde{\mathbf{e}}_1^{\mathsf{T}}\mathbf{D} \in \mathbb{Z}_{q_1}^{\ell_2}.$$

Since the components of $\tilde{\mathbf{e}}_1$ are subgaussian with parameter σ_1 and $\|\mathbf{D}\|_\infty \le N/2$, the components of $\tilde{\mathbf{e}}_1^{\mathsf{T}}\mathbf{D}$ are subgaussian with parameter $\sqrt{\ell_1}(N/2)\sigma_1$. By Lemma 2.5, this means

$$[-\tilde{\mathbf{s}}_1^{\mathsf{T}} \mid 1] \cdot \mathsf{ModReduce}_{\tilde{q}_1}\left(\begin{bmatrix} \mathbf{A}_1\mathbf{D} \\ \tilde{\mathbf{s}}_1^{\mathsf{T}}\mathbf{A}_1\mathbf{D} + \tilde{\mathbf{e}}_1^{\mathsf{T}}\mathbf{D} + \Delta_1\mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D} \end{bmatrix}\right) = \lfloor \tilde{q}_1/N \rceil \cdot \mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D} + \mathbf{e}^{\mathsf{T}}_{\mathsf{simple}}.$$

where $\mathbf{e}^{\mathsf{T}}_{\mathsf{simple}} = \mathbf{e}^{\mathsf{T}}_{\mathsf{simple},1} + \mathbf{e}^{\mathsf{T}}_{\mathsf{simple},2}$ and $\|\mathbf{e}_{\mathsf{simple},1}\|_\infty \le \frac{1}{2}\big(2 + \tilde{q}_1 \bmod N + (\tilde{q}_1/q_1)(q_1 \bmod N)\big)$ and the components of $\mathbf{e}_{\mathsf{simple},2}$ are subgaussian with parameter σ_{simple} and

$$\sigma^2_{\mathsf{simple}} \le d_1\sigma_1^2/4 + (\tilde{q}_1/q_1)^2 \ell_1 N^2 \sigma_1^2/4.$$

Putting the above pieces together,

$$\mu' = [-\tilde{\mathbf{s}}_1^{\mathsf{T}} \mid 1] \cdot \mathbf{c}' = [-\tilde{\mathbf{s}}_1^{\mathsf{T}} \mid 1] \cdot \mathbf{G}_{d_1+1,p}\mathbf{w} = \lfloor \tilde{q}_1/N \rfloor \cdot \mathbf{u}_{i_1}^{\mathsf{T}}\mathbf{D}\mathbf{u}_{i_2} + \mathbf{e}^{\mathsf{T}}_{\mathsf{simple}}\mathbf{u}_{i_2}.$$

Let $e_{\mathsf{simple}} = \mathbf{e}_{\mathsf{simple}}^{\mathsf{T}} \mathbf{u}_{i_2}$. Since $\mathbf{u}_{i_2}$ is a unit vector, $|e_{\mathsf{simple}}| \le \|\mathbf{e}_{\mathsf{simple}}\|_\infty$. By Lemma 2.3, $\mu = \lfloor \mu' \rceil_{\tilde{q}_1, N} = \mathbf{u}_{i_1}^{\mathsf{T}} \mathbf{D} \mathbf{u}_{i_2} = D_{i_1, i_2}$ as long as $|e_{\mathsf{simple}}| \le \frac{\tilde{q}_1}{2N} - (\tilde{q}_1 \bmod N)$. Let

$$\tau_{\mathsf{simple}} = \frac{\tilde{q}_1}{2N} - (\tilde{q}_1 \bmod N) - \frac{1}{2}\left(2 + \tilde{q}_1 \bmod N + \frac{\tilde{q}_1}{q_1}(q_1 \bmod N)\right).$$

Then, by a subgaussian tail bound

$$\Pr[\mu = D_{i_1, i_2}] \ge 1 - 2\exp\left(-\pi \tau_{\mathsf{simple}}^2 / \sigma_{\mathsf{simple}}^2\right).$$

The claim now follows by combining Eq. (B.2) and applying the union bound. □

B.2 Security (Proof of Theorem 3.5)

We proceed with a hybrid argument:

- $\mathsf{Hyb}_0^{(b)}$: This is the real query privacy experiment with bit $b \in \{0,1\}$. Specifically, the game proceeds as follows:
  1. At the beginning of the game, the adversary chooses a database $\mathbf{D} \in \mathbb{Z}_N^{\ell}$. The challenger computes the parameters $(\mathsf{pp}, \mathsf{dbp}) \leftarrow \mathsf{DBSetup}(1^\lambda, \mathbf{D})$ and gives pp to $\mathcal{A}$. In this case, $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$ where $\mathbf{a}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$ and $\mathbf{a}_2 \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{m_2}$.
  2. Algorithm $\mathcal{A}$ outputs a pair of indices $\mathsf{idx}_0 = \left(i_1^{(0)}, i_2^{(0)}\right)$ and $\mathsf{idx}_1 = \left(i_1^{(1)}, i_2^{(1)}\right)$. The challenger computes $(\mathsf{q}, \mathsf{qk}) \leftarrow \mathsf{Query}\left(\mathsf{pp}, \left(i_0^{(b)}, i_1^{(b)}\right)\right)$ and gives q to $\mathcal{A}$. Concretely, let $\boldsymbol{\mu}_1^{(b)} \in R_{d_1,q_1}^{m_1}$ and $\boldsymbol{\mu}_2^{(b)} \in R_{d_2,q_2}^{m_2}$ be the messages derived from idx_b according to the specification of the Query algorithm. Then, for each $j \in \{1,2\}$, the challenger samples $s_j \leftarrow \chi_j$, $\mathbf{e}_j \leftarrow \chi_j^{m_j}$ and computes $\mathbf{c}_j = \mathsf{Coeffs}\left(s_j \mathbf{a}_j + \mathbf{e}_j + \Delta_j \boldsymbol{\mu}_j^{(b)}\right)$, where Δ_j is the scaling factor defined in Construction 3.1. The challenger also computes $\mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s_2, z)$ and sets $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_0, \mathbf{c}_1)$.
  3. At the end of the game, algorithm $\mathcal{A}$ outputs a bit $b' \in \{0,1\}$, which is the output of the experiment.
- $\mathsf{Hyb}_1^{(b)}$: Same as $\mathsf{Hyb}_0^{(b)}$ except when responding to the query, the challenger now samples $\mathbf{r}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$ and sets $\mathbf{c}_1 = \mathsf{Coeffs}(\mathbf{r}_1)$.
- $\mathsf{Hyb}_2^{(b)}$: Same as $\mathsf{Hyb}_1^{(b)}$ except when responding to the query, the challenger now samples $\mathbf{r}_2 \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{m_2}$ and sets $\mathbf{c}_2 = \mathsf{Coeffs}(\mathbf{r}_2)$.

We now argue that each adjacent pair of hybrids are computationally indistinguishable.

- First, $\mathsf{Hyb}_0^{(b)}$ and $\mathsf{Hyb}_1^{(b)}$ are computationally indistinguishable under the $\mathsf{RLWE}_{d_1,m_1,q_1,\chi_1}$ assumption. Namely, given an RLWE challenge $(\mathbf{u}_1, \mathbf{v}_1)$ where $\mathbf{u}_1, \mathbf{v}_1 \in R_{d_1,q_1}^{m_1}$, the reduction algorithm sets $\mathbf{a}_1 = \mathbf{u}_1$ and samples $\mathbf{a}_2 \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{m_2}$. It gives $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$ to the adversary. After the adversary outputs indices $\mathsf{idx}_0, \mathsf{idx}_1$, the reduction algorithm computes $\boldsymbol{\mu}_1^{(b)}$ and $\boldsymbol{\mu}_2^{(b)}$ from idx_0 and idx_1. It then sets $\mathbf{c}_1 = \mathsf{Coeffs}\left(\mathbf{v}_1 + \Delta_1 \boldsymbol{\mu}_1^{(b)}\right)$. It samples $s_2 \leftarrow \chi_2$, $\mathbf{e}_2 \leftarrow \chi^{m_2}$, and sets $\mathbf{c}_2 = \mathsf{Coeffs}\left(s_2 \mathbf{a}_2 + \mathbf{e}_2 + \Delta_2 \boldsymbol{\mu}_2^{(b)}\right)$. Finally, it computes $\mathsf{pk} \leftarrow \mathsf{CDKS.Setup}(1^\lambda, s_2, z)$ and gives $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$ to the adversary. The reduction algorithm outputs whatever the adversary outputs. If the RLWE challenger sampled $\mathbf{u}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$ and set $\mathbf{v}_1 = s_1 \mathbf{u}_1 + \mathbf{e}_1$, where $s_1 \leftarrow \chi_1$ and $\mathbf{e}_1 \leftarrow \chi_1^{m_1}$, then the reduction perfectly simulates $\mathsf{Hyb}_0^{(b)}$. Conversely, if the reduction algorithm sampled $\mathbf{u}_1, \mathbf{v}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$, then it perfectly simulates $\mathsf{Hyb}_1^{(b)}$.
- Next, $\mathsf{Hyb}_1^{(b)}$ and $\mathsf{Hyb}_2^{(b)}$ are computationally indistinguishable if $(\mathsf{CDKS.Setup}, \mathsf{CDKS.Pack})$ satisfies pseudorandomness given the public key (with m_2 samples). Namely, given the challenge $(1^\lambda, \mathsf{pk}, \mathbf{u}_2, \mathbf{v}_2)$, the reduction algorithm sets $\mathbf{a}_2 = \mathbf{u}_2$. It samples $\mathbf{a}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$ and gives $\mathsf{pp} = (1^\lambda, \ell_1, \ell_2, N, \mathbf{a}_1, \mathbf{a}_2)$ to the adversary. After the

adversary outputs indices $\mathsf{idx}_0, \mathsf{idx}_1$, the reduction algorithm computes $\boldsymbol{\mu}_1^{(b)}$ and $\boldsymbol{\mu}_2^{(b)}$ from idx_0 and idx_1. It then samples $\mathbf{r}_1 \xleftarrow{\mathrm{R}} R_{d_1,q_1}^{m_1}$ and sets $\mathbf{c}_1 = \mathsf{Coeffs}(\mathbf{r}_1)$. It sets $\mathbf{c}_2 = \mathsf{Coeffs}\big(\mathbf{v}_2 + \Delta_2 \boldsymbol{\mu}_2^{(b)}\big)$. Finally, it gives $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$ to the adversary. The reduction algorithm outputs whatever the adversary outputs. If the challenger sampled $\mathbf{u}_2 \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{m_2}$ and set $\mathbf{v}_2 = s_2 \mathbf{u}_2 + \mathbf{e}_2$, where $s_2 \leftarrow \chi_2$ and $\mathbf{e}_2 \leftarrow \chi_2^{m_2}$, then the reduction perfectly simulates $\mathsf{Hyb}_1^{(b)}$. Conversely, if the reduction algorithm sampled $\mathbf{u}_2, \mathbf{v}_2 \xleftarrow{\mathrm{R}} R_{d_2,q_2}^{m_2}$, then it perfectly simulates $\mathsf{Hyb}_2^{(b)}$.

- Finally, $\mathsf{Hyb}_2^{(0)}$ and $\mathsf{Hyb}_2^{(1)}$ are identical experiments (the distribution of q in the two experiments is independent of $\mathsf{idx}_0, \mathsf{idx}_1$).

Since each pair of adjacent hybrid experiments are computationally indistinguishable, query privacy now follows by a hybrid argument. □

C Hint Computation in YPIR

For a polynomial $g = \sum_{i=0}^{d-1} \alpha_i x^i$, we define $\mathsf{NCoeffs}(g) = [\alpha_{d-1}, \ldots, \alpha_0]^\intercal$ to denote the coefficients of g in *reverse* order. By inspection, over $R = \mathbb{Z}[x]/(x^d + 1)$, we have for all $f \in R$,

$$\mathsf{NCyclicMat}(g) \cdot \mathsf{NCoeffs}(f) = \mathsf{NCoeffs}(fg). \tag{C.1}$$

Let $\mathbf{a}_1 = [a_{1,1}, \ldots, a_{1,m_1}]^\intercal$ be the vector in Construction 3.1. By definition, Construction 3.1 sets $\mathbf{A}_1 = \mathsf{NCyclicMat}(\mathbf{a}_1^\intercal)$. For a vector $\mathbf{d}$, this means

$$\mathbf{A}_1 \mathbf{d} = [\mathsf{NCyclicMat}(a_{1,1}) \mid \cdots \mid \mathsf{NCyclicMat}(a_{1,m_1})] \cdot \mathbf{d}.$$

If we parse

$$\mathbf{d} = \begin{bmatrix} \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{m_1} \end{bmatrix} \in \mathbb{Z}_{q_1}^{m_1 d_1} \quad \text{where} \quad \mathbf{d}_i \in \mathbb{Z}_{q_1}^{d_1},$$

then

$$\mathbf{A}_1 \mathbf{d} = \sum_{i \in [m_1]} \mathsf{NCyclicMat}(a_{1,i}) \cdot \mathbf{d}_i. \tag{C.2}$$

By appealing to Eq. (C.1), we can compute the term $\mathsf{NCyclicMat}(a_{1,i}) \cdot \mathbf{d}_i$ in $O(d_1 \log d_1)$ time using the NTT (and inverse NTT). Specifically, let $d_i \in R$ be the polynomial where $\mathsf{NCoeffs}(d_i) = \mathbf{d}_i$. Then,

$$\mathsf{NCyclicMat}(a_{1,i}) \cdot \mathbf{d}_i = \mathsf{NCyclicMat}(a_{1,i}) \cdot \mathsf{NCoeffs}(d_i) = \mathsf{NCoeffs}(a_{1,i} d_i).$$

Thus, each term in Eq. (C.2) can be computed as follows:

1. Apply the NTT to the polynomials $a_{1,i}$ and d_i and compute the product $a_{1,i} d_i$.
2. Apply the inverse NTT to the product $a_{1,i} d_i \in R_q$ to obtain $\mathsf{NCoeffs}(a_{1,i} d_i)$.

Overall, computing $\mathbf{A}_1 \mathbf{d}$ requires $m_1 \cdot O(d_1 \log d_1)$ time. In contrast, when $\mathbf{A}_1 \xleftarrow{\mathrm{R}} \mathbb{Z}_{q_1}^{d_1 \times \ell_1}$, computing $\mathbf{A}_1 \mathbf{d}$ would naïvely require $O(\ell_1 d_1) = m_1 \cdot O(d_1^2)$ time. Thus, by taking $\mathbf{A}_1$ to be a structured (negacyclic) matrix, we can reduce the preprocessing cost by a factor of $d_1 / \log d_1$. For typical parameters, $d_1 \approx 2^{10}$, so using a structured $\mathbf{A}_1$ translates to a substantial reduction in the concrete preprocessing costs.

D Additional Evaluation

Table 7 provides the full breakdown of the computation and communication costs of YPIR compared to SimplePIR, DoublePIR, HintlessPIR, and Tiptoe. In particular, we include comparisons with the reference implementation of SimplePIR/DoublePIR [HHC+23b] as well as an implementation using our parameters from Table 1 (labeled SimplePIR* and DoublePIR*). The latter comparisons provide a more direct illustration of the extra overhead incurred by YPIR over SimplePIR and DoublePIR. We refer to Section 4 for further discussion.

Database	Metric	SimplePIR	SimplePIR*	DoublePIR	DoublePIR*	HintlessPIR	Tiptoe	YPIR
1 GB	**Prep. Throughput**	3.7 MB/s	51 MB/s	3.4 MB/s	42 MB/s	*	1.6 MB/s	42 MB/s
	Off. Download	121 MB	112 MB	16 MB	14 MB	–	–	–
	Upload	120 KB	128 KB	312 KB	224 KB	506 KB	33 MB	846 KB
	Download	120 KB	112 KB	32 KB	12 KB	1.5 MB	2.1 MB	12 KB
	Server Time	74 ms	73 ms	94 ms	87 ms	*	2.47 s	428 ms
	Throughput	13.6 GB/s	13.7 GB/s	10.6 GB/s	11.5 GB/s	*	415 MB/s	2.3 GB/s
8 GB	**Prep. Throughput**	3.1 MB/s	51 MB/s	2.9 MB/s	49 MB/s	*	1.6 MB/s	49 MB/s
	Off. Download	362 MB	224 MB	16 MB	14 MB	–	–	–
	Upload	362 KB	512 KB	724 KB	448 KB	506 KB	33 MB	1.5 MB
	Download	362 KB	224 KB	32 KB	12 KB	1.5 MB	8.6 MB	12 KB
	Server Time	708 ms	621 ms	845 ms	649 ms	*	9.75 s	992 ms
	Throughput	11.3 GB/s	12.9 GB/s	9.5 GB/s	12.3 GB/s	*	840 MB/s	8.1 GB/s
32 GB	**Prep. Throughput**	3.3 MB/s	51 MB/s	3.3 MB/s	49 MB/s	*	1.4 MB/s	49 MB/s
	Off. Download	724 MB	448 MB	16 MB	14 MB	–	–	–
	Upload	724 KB	1.0 MB	1.4 MB	896 KB	*	34 MB	2.5 MB
	Download	724 KB	448 KB	32 KB	12 KB	*	17 MB	12 KB
	Server Time	3.08 s	2.54 s	3.22 s	2.59 s	*	21.00 s	2.93 s
	Throughput	10.4 GB/s	12.6 GB/s	9.9 GB/s	12.4 GB/s	*	1.5 GB/s	10.9 GB/s

Table 7: Costs of retrieving a single bit using YPIR compared to other PIR schemes. This is the same breakdown as in Table 2, except we include columns for our implementation of SimplePIR (SimplePIR*) and DoublePIR (DoublePIR*) using the parameters from Table 1 and with modulus switching.