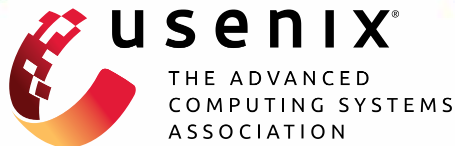

YPIR: High-Throughput Single-Server PIR with Silent Preprocessing

Samir Jordan Menon, *Blyss;* David J. Wu, *UT Austin*

https://www.usenix.org/conference/usenixsecurity24/presentation/menon

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

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

Samir Jordan Menon
Blyss

David J. Wu
UT Austin

Abstract

We introduce YPIR, a single-server private information retrieval (PIR) protocol that achieves high throughput (up to 83% 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 12.1 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 12.5 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 8×. 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 48× more than YPIR (based on current AWS compute costs).

1 Introduction

A private information retrieval (PIR) [23, 50] 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 [71, 51, 5, 4], private database queries and web search [82, 41], password breach alerting [57, 81, 3], Certificate Transparency auditing [43], private media consumption [39], private ad delivery [47, 7, 37], 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 [26, 43, 86, 74] 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 [43] 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 [86, 74, 35] 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 [4, 1, 72, 67], 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 [67] 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 [43], 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 [43] 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.

The client-specific state used by OnionPIR [72], Spiral [67], and similar schemes [4, 1] introduces its own share of challenges. In these schemes, the server must store a (large) public key for each client, imposing high storage requirements for the server and also requiring additional infrastructure to support efficient client state lookups and retrieval. Reuse of client-specific public keys can also enable active attacks on the application [43].

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 [41][1] and HintlessPIR [55]. Both schemes essentially leverage a form of "bootstrapping" [32] to remove the hint from the SimplePIR protocol [43], where the server *homomorphically* compresses the SimplePIR hint using an encoding of the client secret key. We refer to the full version of this paper [68] for a more detailed summary of these two schemes. 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 still requires 4× more online communication than SimplePIR and has a throughput that is at most 62% of the SimplePIR throughput. We provide a more detailed comparison of the bootstrapping-based approach from Tiptoe and HintlessPIR with our "key-switching-based" approach in Section 4.2, and an overview of the design of Tiptoe and HintlessPIR in the full version of this paper [68].

[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.

1.1 Our Contributions

In this work, we introduce YPIR, a new single-server PIR protocol with silent preprocessing. Like Tiptoe [41] and HintlessPIR [55], 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 [67]) 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.

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 12.1 GB/s, which is 97% of the throughput of SimplePIR (and 83% of the memory bandwidth of the machine). In contrast, HintlessPIR achieves a maximum throughput that is only 62% of SimplePIR (and concretely, 6.4 GB/s on our machine) [55]. For database sizes ranging from 1 GB to 32 GB, 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 1.8–3× larger than those in DoublePIR, 3–7× larger than SimplePIR, and similar to 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 97% 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. Asymptotically, our approach reduces the offline preprocessing cost by a factor of $n/\log n$, where n is the lattice dimension (in SimplePIR-based systems, $n \geq 1024$). 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 is 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

[2]The Y in YPIR is to reflect the fact that the protocol design combines the high-throughput capabilities of PIR based on integer lattices (i.e., the LWE assumption [80]) with the response compression techniques from PIR based on ideal lattices (i.e., the RLWE assumption [62]).

increasing the number of CPU operations per byte of memory read. In Section 4.1, we describe 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.[3] 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 $1.4\times$ to 17 GB/s; applied to YPIR, we achieve an effective throughput of 16 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 [52, 53]. 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. [43] 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 [43] compromises 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 $8\times$ 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 \$228 per million clients for YPIR). The cost of YPIR further drops to \$183 per million clients if we leverage cross-client batching with a batch size of 4 (i.e., assume that the server always has a 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 over \$10,000 per million clients, which is $48\times$ higher than using YPIR. Compared to other hintless PIR schemes such as Tiptoe and HintlessPIR, we estimate YPIR achieves a cost savings of 16–$84\times$ 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 1.8–$3\times$ larger than a DoublePIR query (for an 8 GB database, YPIR queries are 1.5 MB while DoublePIR queries are 724 KB) and 3–$7\times$ larger than a SimplePIR query. If the application setting has a small, fixed communication budget, YPIR may not be appropriate; for example, for a 32 GB database, the minimum YPIR query size is 1.1 MB. We refer to Section 4.2 for more details on the communication-computation trade-offs in YPIR, HintlessPIR, SimplePIR, and DoublePIR.

1.2 Overview of YPIR

The starting point for this work is the SimplePIR/DoublePIR schemes from [43] based on the learning with errors (LWE) problem [80]. 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 $\mathsf{ct}_1, \dots, \mathsf{ct}_{\ell_2} \in \mathbb{Z}_q^{n+1}$ 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 [43] 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

[3] We contrast this with single-client batching [9, 46, 38, 4], 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). Cross-client batching was also used in [60] to improve the effective throughput of PIR in the multi-server setting.

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 [62]. 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 $1000\times$ reduction in ciphertext expansion factor.

In YPIR, we use the LWE-to-RLWE packing technique from [21]. 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.[4] For an 8 GB database, this packing approach compresses the full 16 MB DoublePIR response into a 12 KB response (see Table 1). The cost is that the query must now include a "packing key" for the transformation from [21] (which essentially consists of RLWE key-switching matrices). This increases the query size from 724 KB in DoublePIR by a factor of $2\times$ to 1.5 MB. We additionally note that most of the computational costs of the [21] transformation can actually be moved to an offline preprocessing phase (because it is applied to *query-independent* components). In our experiments, we observed a $9\times$ reduction in the online computational cost by having the server perform a modest amount of additional work in the offline phase. We describe this approach in Section 4.1.

Supporting large records. A limitation of DoublePIR is that it only supports retrieving small records (i.e., a single element of the plaintext space $\mathbb{Z}_p$). This is sufficient for some applications like private SCT auditing (see Section 4.4), but other PIR applications may require support for large records. In Section 4.3, we show that we can also apply the same packing approach to SimplePIR to obtain a PIR protocol (YPIR+SP) that supports queries to databases with large records. This is a similar setting considered in HintlessPIR (i.e., composing SimplePIR with a LWE-to-RLWE transformation) [55]. As we describe in Section 4.3, our YPIR+SP protocol achieves a $2.2\times$ reduction in total communication with only a 5% reduction in throughput compared to HintlessPIR when considering databases with 32–64 KB records.

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 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$\times$ 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 Section 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., $\mathbf{u}$, $\mathbf{v}$).

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.

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

[4] It is also possible to pack LWE encodings (e.g., using the SPIRAL approach for response compression [67]) into a packed LWE ciphertext [79], 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.

rationals. We extend $\lfloor\cdot\rceil_{q,p}$ to operate component-wise on vector-valued and matrix-valued inputs.

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 $\mathsf{Coeffs}(f)^\mathsf{T}\cdot\mathsf{NCyclicMat}(g)=\mathsf{Coeffs}(fg)^\mathsf{T}$. Specifically,

$$\mathsf{NCyclicMat}(g) := \begin{bmatrix} \alpha_0 & \alpha_1 & \alpha_2 & \cdots & \alpha_{d-1} \\ -\alpha_{d-1} & \alpha_0 & \alpha_1 & \cdots & \alpha_{d-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\alpha_1 & -\alpha_2 & -\alpha_3 & \cdots & \alpha_0 \end{bmatrix}.$$

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

$$\begin{aligned}\mathsf{Coeffs}(f\cdot\mathbf{g})^\mathsf{T} &= [\mathsf{Coeffs}(fg_1)^\mathsf{T}\mid\cdots\mid\mathsf{Coeffs}(fg_m)^\mathsf{T}] \\ &= \mathsf{Coeffs}(f)^\mathsf{T}\cdot\mathsf{NCyclicMat}(\mathbf{g}^\mathsf{T}).\end{aligned} \tag{2.1}$$

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 [61, 59], 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 $\mathsf{Coeffs}(f)$.

Gadget matrices. Next, we recall the notion of the gadget matrix from [70]. 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 $t=\lceil\log q/\log z\rceil$. For a dimension $n\in\mathbb{N}$, we define $\mathbf{G}_{n,z}:=\mathbf{I}_n\otimes\mathbf{g}_z^\mathsf{T}\in\mathbb{Z}_q^{n\times nt}$ 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 $\mathbf{M}$.

Ring learning with errors. Like many lattice-based PIR schemes [66, 4, 33, 72, 67, 55], the security of our protocol relies on the ring learning with errors (RLWE) problem [80, 62]. 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 [6].

Definition 2.1 (Ring Learning with Errors [62])**.** 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 $\mathbf{v}\xleftarrow{\mathrm{R}}R_q^m$, the following two distributions are computationally indistinguishable: $(\mathbf{a},s\mathbf{a}+\mathbf{e})$ 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 $[-\mathbf{s}^\mathsf{T}\mid 1]\cdot\mathbf{c}=\mu+e$. 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.

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 [21] 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 give an informal overview of the transformation here and defer the formal description to the full version of this paper [68]. Informally, there are two algorithms $(\mathsf{CDKS.Setup},\mathsf{CDKS.Pack})$ 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 LWE encodings $\mathbf{C}\in\mathbb{Z}_q^{(d+1)\times d}$, the packing algorithm outputs an RLWE encoding $\mathbf{c}'\in R_q^2$.

Correctness says that if the columns $\mathbf{c}_1,\ldots,\mathbf{c}_d\in\mathbb{Z}_q^{d+1}$ of $\mathbf{C}$ are LWE encodings of $\mu_1,\ldots,\mu_d\in\mathbb{Z}_p$ with respect to the secret key $\mathbf{s}=\mathsf{Coeffs}(s)\in\mathbb{Z}_q^d$, then the $\mathbf{c}'\in R_q^2$ output by $\mathsf{CDKS.Pack}$ is an RLWE encoding of $\sum_{i\in[d]}\mu_i x^{i-1}\in R_p$ (with respect to the secret key $s\in R_q$). Namely, the transformation takes $d(d+1)$ elements over $\mathbb{Z}_q$ and packs them into just $2d$ elements over $\mathbb{Z}_q$ (which encode a polynomial with $\mu_1,\ldots,\mu_d$ as its coefficients). We do assume that the lattice dimension of the LWE encodings coincide with the ring dimension.

Modulus switching. A standard technique to reduce the size of lattice-based encodings after performing homomorphic operations on them is to use *modulus switching* [17, 16]. Modulus switching takes an (R)LWE encoding $\bmod q$ and rescales it to an encoding $\bmod q_1$ where $q_1<q$. This reduces the size of the encoding. Here, we describe a more fine-grained variant from [67] 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 $q > q_1 \geq q_2$ 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 give the formal correctness property in the full version of this paper [68].

Private information retrieval. We now recall the formal definition of a (two-message) single-server PIR protocol [50]. 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 2.2 (Private Information Retrieval [50, 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 PIR scheme is correct if 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

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

where $\mathbf{D}[\mathsf{idx}]$ denotes the element of $\mathbf{D}$ indexed by idx. Here, δ denotes a correctness error. The scheme satisfies query privacy if no efficient adversary can distinguish a query to an index idx_0 from a query to an index idx_1. We provide the formal correctness and security definitions for PIR in the full version of this paper [68].

3 The YPIR Protocol

In this section, we describe the YPIR protocol. As described in Section 1.2, the YPIR protocol first invokes DoublePIR [43] 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 (see the full version of this paper [68] for the precise parameter instantiation).

The $\mathsf{YPIR} = (\mathsf{DBSetup}, \mathsf{Query}, \mathsf{Answer}, \mathsf{Extract})$ 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 $m_1, m_2 \in \mathbb{N}$,[5] 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

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

  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:

[5] 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 the full version of this paper [68].

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 jth 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 $\mathsf{qk} = (s_1, s_2)$.

- $\mathsf{Answer}(\mathsf{dbp}, \mathsf{q})$: On input the database parameters $\mathsf{dbp} = (1^\lambda, \mathbf{D}, \mathbf{H}_1, \mathbf{H}_2)$ and the query $\mathsf{q} = (\mathsf{pk}, \mathbf{c}_1, \mathbf{c}_2)$, 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:** Let

$$\mathbf{C} = (d_2^{-1} \bmod q_2) \cdot \begin{bmatrix} \mathbf{H}_2 & \mathbf{A}_2\mathbf{T}^\mathsf{T} \\ \mathbf{c}_2^\mathsf{T}\mathbf{H}_1^\mathsf{T} & \mathbf{c}_2^\mathsf{T}\mathbf{T}^\mathsf{T} \end{bmatrix} \in \mathbb{Z}_{q_2}^{(d_2+1)\times\kappa(d_1+1)}. \tag{3.2}$$

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

- $\mathsf{Extract}(\mathsf{qk}, \mathsf{resp})$: On input the client state $\mathsf{qk} = (s_1, s_2)$ and the response $\mathsf{resp} = ((c_{1,1}, c_{1,2}), \ldots, (c_{\rho,1}, c_{\rho,2}))$ 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 [72, 67, 43, 26]. 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 (in the random oracle model).

Supporting arbitrary dimension. Construction 3.1 assumes that the database dimensions are multiples of the ring dimensions d_1 and d_2. We describe in the full version of this paper [68] a straightforward way to adapt the scheme to support databases with arbitrary dimensions (and without padding).

Correctness and security. We now give the formal correctness and security theorems for the YPIR protocol, but defer their proofs to the full version of this paper [68]. To summarize, the security of YPIR relies on the hardness of the RLWE assumption as well as a standard circular security assumption (for the use of key-switching matrices in the LWE-to-RLWE packing). The circular-security assumption is common when working with lattice-based homomorphic encryption schemes [32, 17, 16, 15] and also used in many previous RLWE-based PIR schemes [4, 1, 72, 67, 55].

Theorem 3.3 (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*

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

where the parameters τ_{double}, σ_{double}, τ_{simple}, and σ_{simple} are given in Fig. 1.

Theorem 3.4 (Security)**.** *Under the $\mathsf{RLWE}_{d_1,m_1,q_1,\chi_1}$ assumption and assuming the LWE-to-RLWE packing scheme* $(\mathsf{CDKS.Setup}, \mathsf{CDKS.Pack})$ *satisfies pseudorandomness given the packing key (see the full version of this paper [68]), then Construction 3.1 satisfies query privacy.*

4 Implementation and Evaluation

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

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

Figure 1: Parameters for Theorem 3.3 (Correctness of YPIR).

Parameter selection. Theorem 3.3 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 $\delta \leq 2^{-40}$ and 128-bits of security (as estimated by the Lattice Estimator [2][6]). We select a single parameter set for YPIR using the following procedure:

- Like SimplePIR [43], we set $d_1 = 2^{10} = 1024$ and $q_1 = 2^{32}$. We set χ_1 to be a discrete Gaussian distribution[7] 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).
- We choose parameters to support any choice of $\ell_1, \ell_2 \leq 2^{18}$ (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 $\delta \leq 2^{-40}$. This allows for faster computation.
- 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 the full version of this paper [68]. 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 Additional Optimizations

In the full version of this paper [68], we describe some additional techniques to improve the concrete efficiency of YPIR. We provide a brief overview of these here.

NTT-based hint computation. First, we show that using structured matrices (specifically, negacyclic matrices; see Section 2), we can significantly reduce the computational cost of computing the hints $\mathbf{H}_1$ and $\mathbf{H}_2$ in Eq. (3.1) of YPIR. Specifically, in YPIR, the matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ are defined to be $\mathbf{A}_i := \mathsf{NCyclicMat}(\mathbf{a}_i^{\mathsf{T}})$. In SimplePIR/FrodoPIR, the corresponding matrices $\mathbf{A}_1$ and $\mathbf{A}_2$ were uniformly random. By using a structured matrix, we can leverage NTTs to more efficiently compute the hints $\mathbf{H}_1$ and $\mathbf{H}_2$ in Eq. (3.1). In particular, 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 [43] or FrodoPIR [26]. Our approach directly applies to reduce the preprocessing cost in any system that builds on SimplePIR/FrodoPIR (e.g., [24, 41, 55, 27]) with *zero* impact to the online costs of the protocol (the online server processing is unchanged). The only difference is security now relies on RLWE (due to the use of structured $\mathbf{A}_i$) rather than LWE.

Preprocessing the CDKS transformation. We show how to speed up the [21] LWE-to-RLWE packing transformation by moving a large portion of the online packing computation to the *offline* preprocessing stage (i.e., from Answer to Setup). Specifically, our approach reduces the number of NTTs that must be performed in Answer from $O(\kappa d_1 + \log d_2)$ to $O(\kappa + \log d_2)$. Concretely, this reduces the online cost of the packing transformation by 9× (see Table 4).

Cross-client batching. Protocols like SimplePIR and YPIR are ultimately constrained by the memory bandwidth of the

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

[7]We refer to the full version of this paper [68] for a definition of the discrete Gaussian distribution.

[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. In our implementation, we reduce only when the computation might "overflow" the 64-bit integer.

system (see Section 4) and not the cost of evaluating the underlying arithmetic operations during query processing. One approach to increase the *effective* throughput in these protocols is to perform additional computation for each byte of memory accessed. A natural approach to overcome the memory bandwidth barrier is to support "cross-client batching" [60] where the server uses a single scan through memory to answer requests from multiple *independent* clients.

The structure of SimplePIR makes it amenable for cross-client batching as the core computation is a matrix-vector product $\mathbf{q}^{\mathsf{T}}\mathbf{D}$ between the query $\mathbf{q}$ and the database $\mathbf{D}$. If a batch of k queries $\mathbf{q}_1, \ldots, \mathbf{q}_k$ arrive simultaneously, then the computation becomes a matrix-vector product $\mathbf{QD}$, where the rows of $\mathbf{Q}$ are the queries $\mathbf{q}_1, \ldots, \mathbf{q}_k$. In this way, the server performs more arithmetic operations per byte of memory fetched, which in turn increases the effective throughput of the protocol. We provide the full details in Section 4.1. As we show in Section 4.2, cross-client batching can increase the effective server throughput of many PIR schemes by 1.5–1.7×. A 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.

Our technique is similar to the cross-client batching technique of Lueks and Goldberg [60] who used faster matrix-vector multiplication algorithms to obtains asymptotic and concrete server speed-ups. In our setting, we leverage batching as a means to improve CPU utilization and as such, our approach only provides a concrete (but not asymptotic) improvement to server throughput.

4.2 Experimental Evaluation

In this section, we describe our experimental evaluation of the YPIR protocol and compare it against other PIR protocols. We compare against the state-of-the-art high-throughput single-server PIR schemes: SimplePIR/DoublePIR [43] as well as the hintless schemes proposed in Tiptoe [41] and HintlessPIR [55]. We refer to the full version of this paper [68] for a more detailed summary of the design of the PIR scheme from Tiptoe as well as the HintlessPIR scheme. We do not benchmark schemes in alternative models such as the sublinear schemes that require streaming the database in the offline phase [86, 74, 35] or the RLWE-based schemes that require maintaining client-specific keys [4, 1, 3, 72, 67].

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 [43].[9] We use the approach from Section 4.1 to implement the server hint precomputation, and use the approach from Section 4.1 to speed up the LWE-to-RLWE packing transformation. As discussed in Remark 3.2, 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 [43] (commit `e9020b0`), the PIR scheme from Tiptoe [41] (commit `f053a81`), and HintlessPIR [55] (commit `4be2ae8`). When relevant, we compile each scheme with support for the Intel HEXL [11] 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 use the same (single-threaded)[10] 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. We write KB, MB, and GB to denote 2^{10}, 2^{20}, and 2^{30} bytes, respectively. 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 1, 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 43% slower than SimplePIR and 26% slower than DoublePIR. This is because a significant portion of the query-processing time is spent on the LWE-to-RLWE transformation (30%; see Table 2). 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 3–18% *faster* than the reference implementations of SimplePIR and DoublePIR and 79% 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 [44] 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 lattice parameters. Compared to our SimplePIR* and DoublePIR* implementations, the throughput of YPIR on a 8 GB database is only 10% slower, and for a 32 GB database, only 1% slower. Thus, for moderate-size databases, YPIR achieves similar throughput to SimplePIR/-DoublePIR *without* any offline hints. We provide more details in Fig. 2.

Compared to the Tiptoe approach [41], YPIR achieves 8–

[9] Our code is available at `https://github.com/menonsamir/ypir`.

[10] The primary computational cost in the SimplePIR-family of protocols (including YPIR and HintlessPIR) is computing a matrix-vector product. This is a highly parallelizable operation. However, for ease of comparison, we focus on a single-threaded execution in our evaluation.

Database	Metric	SimplePIR	DoublePIR	Tiptoe	HintlessPIR	YPIR
1 GB	**Prep. Throughput**	3.7 MB/s	3.4 MB/s	1.6 MB/s	4.8 MB/s	39 MB/s
	Off. Download	121 MB	16 MB	—	—	—
	Upload	120 KB	312 KB	33 MB	488 KB	846 KB
	Download	120 KB	32 KB	2.1 MB	1.7 MB	12 KB
	Server Time	74 ms	94 ms	2.47 s	743 ms	129 ms
	Throughput	13.6 GB/s	10.6 GB/s	415 MB/s	1.3 GB/s	7.8 GB/s
8 GB	**Prep. Throughput**	3.1 MB/s	2.9 MB/s	1.6 MB/s	5.2 MB/s	46 MB/s
	Off. Download	362 MB	16 MB	—	—	—
	Upload	362 KB	724 KB	33 MB	1.4 MB	1.5 MB
	Download	362 KB	32 KB	8.6 MB	1.7 MB	12 KB
	Server Time	708 ms	845 ms	9.75 s	1.62 s	687 ms
	Throughput	11.3 GB/s	9.5 GB/s	840 MB/s	4.9 GB/s	11.6 GB/s
32 GB	**Prep. Throughput**	3.3 MB/s	3.3 MB/s	1.4 MB/s	5.7 MB/s	48 MB/s
	Off. Download	724 MB	16 MB	—	—	—
	Upload	724 KB	1.4 MB	34 MB	2.4 MB	2.5 MB
	Download	724 KB	32 KB	17 MB	3.2 MB	12 KB
	Server Time	3.08 s	3.22 s	21.00 s	5.00 s	2.64 s
	Throughput	10.4 GB/s	9.9 GB/s	1.5 GB/s	6.4 GB/s	12.1 GB/s

Table 1: 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, [43], Tiptoe [41], and HintlessPIR [55] are all obtained by running their *official* reference implementations on our test system [44, 42, 56]. We refer to the full version of this paper [68] for a direct comparison with *our* implementations of SimplePIR and DoublePIR (derived from the subprotocols of YPIR), which achieve higher throughput than the provided reference implementation.

19× 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 1–10% of the total server processing time (see Table 2).

Compared to HintlessPIR, YPIR achieves 2–6× higher server throughput. Notably, the HintlessPIR reference implementation peaks at 6.4 GB/s while YPIR peaks at 12.1 GB/s. One reason underlying this performance gap is because HintlessPIR applies the LWE-to-RLWE transformation to pack $O(\sqrt{N})$ encodings, where N is the number of records in the database. In contrast, YPIR only needs to pack a fixed number of LWE encodings (independent of the number of records). For a 32 GB database, HintlessPIR spends roughly 50% of its time performing packing (because it packs $O(\sqrt{N})$ encodings), whereas YPIR spends only 1% of its time packing.

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. 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.

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. 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 and 8× faster than HintlessPIR. For a 32 GB

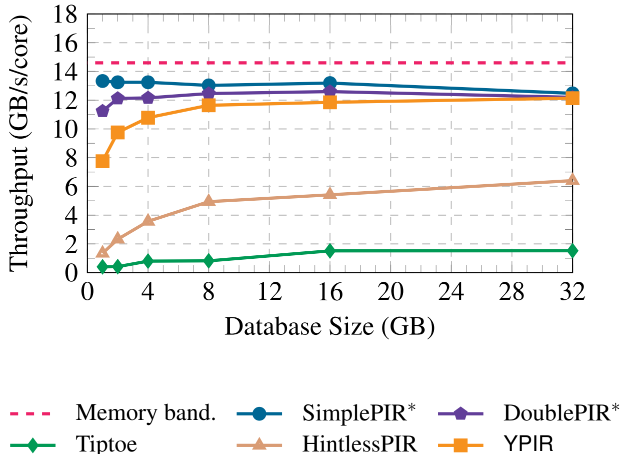

Figure 2: Server throughput for retrieving a single bit from different databases. For SimplePIR and DoublePIR, we report throughput using our reference implementation and parameter choices (denoted SimplePIR* and DoublePIR*), since these were faster than those of the reference implementation [44] in our test setup. For HintlessPIR [55], we report the bandwidth measured on our system with the reference implementation [56]. We measure the memory bandwidth of the system using STREAM [65].

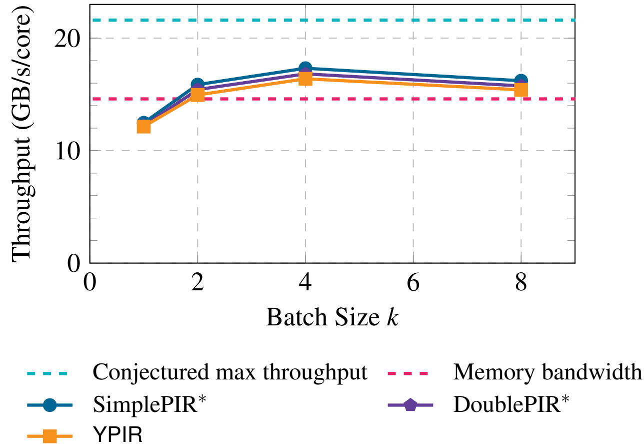

Figure 3: Effective per-query server throughput for retrieving a single bit from a 32 GB database with cross-client batching. As in Fig. 2, we use *our* implementation of SimplePIR and DoublePIR (i.e., SimplePIR* and DoublePIR*) for the comparisons. We measure the memory bandwidth of the system using STREAM [65]. 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 [45].

database, the offline precomputation of YPIR would take about 11 CPU-minutes, whereas for SimplePIR/DoublePIR, it would take roughly 144 CPU-minutes, and for HintlessPIR, it would take 95 CPU-minutes.

Server microbenchmarks. Table 2 provides a fine-grained breakdown of the server computation costs of YPIR (i.e., the Answer algorithm in Construction 3.1). 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 30% 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 1% of the overall cost of the server processing. In this case, the throughput of YPIR quickly approaches that of DoublePIR.

Size	SimplePIR	DoublePIR	Packing	Total
1 GB	0.07 s (59%)	14 ms (11%)	39 ms (30%)	0.13 s
4 GB	0.30 s (82%)	27 ms (7%)	38 ms (10%)	0.37 s
16 GB	1.21 s (93%)	57 ms (4%)	39 ms (3%)	1.31 s
32 GB	2.56 s (96%)	58 ms (2%)	39 ms (1%)	2.66 s

Table 2: 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.

Query size breakdown. In Table 3, 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. 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 packing transformation). As such, the encoding $\mathbf{c}_2$ is roughly $(\log q_2 / \log q_1) \approx 2\times$ 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 4.1. For a database of size ℓ, we define the effective

Database Size	$\|\mathbf{c}_1\|$	$\|\mathbf{c}_2\|$	$\|\mathsf{pk}\|$	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 3: 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.

(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 $1.4\times$. In the case of SimplePIR and DoublePIR, processing a batch of 4 queries yields a $1.4\times$ 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 4 is over 16 GB/s, which is $1.3\times$ 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 2) 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 though there is no reduction in the total number of arithmetic operations the server performs.

LWE-to-RLWE translation. Tiptoe [41], HintlessPIR [55], and YPIR all apply some form of LWE-to-RLWE translation to *compress* the SimplePIR/DoublePIR hints and eliminate the need for an offline hint download. Here, we provide a more detailed discussion of the different approaches.

Tiptoe and HintlessPIR rely on a bootstrapping-like approach where the client provides an RLWE encoding of the secret key in its query. The server then treats the LWE encodings in the SimplePIR hint as a vector of *plaintexts*. Then, using the RLWE encoding of the LWE secret key, it homomorphically evaluates the inner product between the encodings in the SimplePIR hint and the secret key. This yields an RLWE encoding of the desired database record. Since both of these approaches essentially implement homomorphic decryption, they set the *plaintext* modulus of the RLWE encoding scheme to be at least as large as the LWE encoding modulus. This results in needing to use a much larger RLWE encoding modulus to achieve correctness. For example, HintlessPIR uses a 90-bit RLWE modulus to implement this step (whereas the LWE encoding modulus in the SimplePIR hint is just 32 bits).

In contrast to the previous approaches, YPIR applies the Chen-Dai-Kim-Song packing transformation [21]. While this could also be viewed as a type of "bootstrapping" (since the transformation relies on key-switching, which is in some sense a homomorphic decryption operation), it does *not* require us to "re-encode" the LWE encodings under RLWE. Like most key-switching transformations, the Chen et al. transformation allows us to use the same modulus for the LWE encoding and for the RLWE encodings. Moreover, the noise introduced by key-switching is additive and is *not* scaled up by the magnitude of the LWE encoding modulus. A downside of this approach is that the LWE and RLWE encodings share a common modulus, so we cannot use a power-of-two modulus, as such moduli are not NTT-friendly.

	Tiptoe	HintlessPIR	Chen et al.
$\log(n, q, p)$	$(10, 32, 8)$	$(10, 32, 8)$	$(11, 56, 15)$
Param. Size	32 MB	360 KB	528 KB
Output Size	514 KB	180 KB	24 KB
Output Rate	0.01	0.02	0.31
Offline Comp.	—	2012 ms	1029 ms
Online Comp.	594 ms	141 ms	52 ms

Table 4: Concrete costs of packing 2^{12} input LWE encodings into RLWE encodings using the Tiptoe [41], HintlessPIR [55], and the Chen et al. [21] approach used in YPIR. We report the lattice parameters for the input LWE encodings considered in each construction: n is the lattice dimension, q is the encoding modulus, and p is the plaintext modulus. We also report the size of the parameters the client must upload to the server, and the size of the output RLWE encodings. To normalize for the differences in the lattice parameters, we also report the rate (the ratio of the plaintext size in the packed encoding to the size of the encoding). Finally, we measure the offline and online server computation times. We report the Chen et al. packing approach with the preprocessing technique described in Section 4.1.

In Table 4, we provide microbenchmarks for packing 4096 LWE encodings (of dimension n) into RLWE encodings (of dimension $d \geq n$) using the different approaches. HintlessPIR has the smallest public parameters because it only requires a single key-switching matrix. The Chen et al. approach uses $\log d$ key-switching matrices. Tiptoe uses a separate RLWE

encoding for each component of the LWE secret, so its parameters have size $O(nd)$ and are concretely larger than both approaches. The size of the packed encodings is $7.5\times$ smaller using our approach than HintlessPIR (and $21\times$ smaller than Tiptoe). The reduction in size is because the Chen et al. approach can use a smaller RLWE modulus and ring dimension (concretely, a 56-bit modulus and $d = 2048$, compared to a 90-bit modulus and $d = 4096$ in HintlessPIR). We can also apply modulus reduction to further reduce the size of the encodings. If we factor in the different lattice parameters considered in each construction and focus on the rate (i.e., the ratio of the size of the plaintext in the packed encoding to the size of the packed encoding), the Chen et al. approach is over $15\times$ higher than the approach from HintlessPIR. In terms of computation, the Chen et al. procedure with preprocessing is roughly $2.7\times$ faster than the approach taken in HintlessPIR and $11\times$ faster than the approach in Tiptoe.

4.3 Supporting Larger Records

The basic YPIR protocol is tailored for retrieving a single bit (or byte) from a database. To support larger database records, we consider a variant of YPIR where we apply the LWE-to-RLWE packing procedure to the SimplePIR output rather than the DoublePIR output. Recall that the SimplePIR output encodes an entire column of the database (as opposed to just a single record). Thus, the SimplePIR output is already naturally encoding a "large record." Note that this version of YPIR is similar to the approach taken in HintlessPIR, where they apply bootstrapping to pack the SimplePIR hint into a small number of RLWE encodings.

In Table 5, we compare the performance of our YPIR with SimplePIR (denoted YPIR+SP) approach with SimplePIR and HintlessPIR for retrieving large records from various databases. For sake of comparison, we consider the database configurations from [55]. Overall, YPIR with SimplePIR has a similar query size to HintlessPIR, but 7–14$\times$ smaller responses. As discussed in Section 4.2, HintlessPIR has larger responses because the bootstrapping approach requires it to embed the SimplePIR encoding modulus (32 bits) within the plaintext space of the output RLWE encodings. This leads to a much larger RLWE encoding modulus (and thus, response size). In contrast, the approach used by YPIR applies packing *directly* to the input LWE encodings, rather than treating them as plaintexts; this allows the RLWE ciphertext modulus to be the same as the LWE ciphertext modulus.

The throughput of YPIR+SP is similar to that of HintlessPIR, ranging from $1.8\times$ faster for small databases, to 5% slower for large databases. YPIR is faster for small databases because it uses a lighterweight LWE-to-RLWE packing procedure (see Section 4.2 and Table 4). However, the conversion step is only applied to an input of size $O(\sqrt{N})$ whereas the SimplePIR step is applied to an input of size $O(N)$, where N is the size of the database. This makes the difference in overall throughput less substantial when N is large but noticeable when N is small. Because YPIR+SP performs packing directly on the result of the SimplePIR step, it uses an NTT-friendly modulus that is not a power-of-two in the SimplePIR step. This makes the SimplePIR step of YPIR+SP about $1.4\times$ slower than the SimplePIR reference implementation. Since HintlessPIR can be directly applied to SimplePIR, it is able to achieve higher throughput than YPIR+SP when the database is large (e.g., HintlessPIR is 5% faster for a 32 GB database). Thus, for large databases with big records, YPIR+SP has substantially smaller total communication, but comes at a small reduction in throughput relative to HintlessPIR.

4.4 Application to Private SCT Auditing

Certificate Transparency (CT) [52, 53] 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 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. 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 [28] or accessing the log server via an anonymizing proxy [25], but these approaches do not provide formal cryptographic guarantees to privacy.

Private SCT auditing. Several works have proposed to use PIR for private SCT auditing [60, 48, 43]. In this work, we focus on the recent approach of Henzinger et al. [43] that leverages single-server 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

Database	Metric	SimplePIR	HintlessPIR	YPIR+SP
$2^{15} \times 32$ **KB** **(1 GB)**	**Prep. Throughput**	3.7 MB/s	4.8 MB/s	63 MB/s
	Off. Download	121 MB	—	—
	Upload	120 KB	488 KB	686 KB
	Download	120 KB	1.7 MB	120 KB
	Server Time	74 ms	743 ms	415 ms
	Throughput	13.6 GB/s	1.3 GB/s	2.4 GB/s
$2^{18} \times 32$ **KB** **(8 GB)**	**Prep. Throughput**	3.1 MB/s	5.2 MB/s	101 MB/s
	Off. Download	362 MB	—	—
	Upload	362 KB	1.4 MB	1.3 MB
	Download	362 KB	1.7 MB	228 KB
	Server Time	708 ms	1.62 s	1.56 s
	Throughput	11.3 GB/s	4.9 GB/s	5.1 GB/s
$2^{19} \times 64$ **KB** **(32 GB)**	**Prep. Throughput**	3.3 MB/s	5.7 MB/s	115 MB/s
	Off. Download	724 MB	—	—
	Upload	724 KB	2.4 MB	2.2 MB
	Download	724 KB	3.2 MB	444 KB
	Server Time	3.08 s	5.00 s	5.24 s
	Throughput	10.4 GB/s	6.4 GB/s	6.1 GB/s

Table 5: Communication and computation needed to retrieve larger records from databases of varying configurations.

size 2^{36} bits (8 GB). Each SCT audit in turn corresponds to a single PIR query to this database. In [43], 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 [69]. 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 in [43] is to have clients download the hints on a *weekly* basis and wait to test an SCT if its validity falls outside the time window associated with the current hint. While this reduces the protocol's communication costs, it also introduces delays in detecting malicious log behavior. The log server must also maintain multiple copies of the SCT database to support PIR queries for hints issued at different times.

A PIR scheme with silent preprocessing avoids these deployment issues. Following [43], 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 [28]). 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 [43] approach with DoublePIR, Tiptoe, HintlessPIR, and YPIR.

When the client downloads weekly hints, the DoublePIR approach [43] 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 \$228 per 1 million clients. This is 8× cheaper than using DoublePIR with *weekly* updates and over 48× 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 \$183 per 1 million clients.

Scheme likes HintlessPIR, Tiptoe, and YPIR that do not require hints are unaffected by update frequency. The upload in YPIR is just 1.07× larger than HintlessPIR, and 23× smaller than Tiptoe. Since AWS only charges for *outgoing* communication, and Tiptoe and HintlessPIR have larger responses than YPIR, they have substantially higher AWS costs (84× and 16×, respectively). Overall, the total communication required by YPIR is 2× 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 *smaller* even compared to DoublePIR's total communication with *weekly* updates. So even if the client amortizes the DoublePIR hint across multiple queries over the course of the week, YPIR still achieves smaller end-to-end commu-

[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 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.

Update Frequency	DoublePIR *Weekly*	DoublePIR *Daily*	Tiptoe —	HintlessPIR —	YPIR —
Offline Download	16 MB	112 MB	—	—	—
Upload	14 MB	14 MB	659 MB	27 MB	29 MB
Download	640 KB	640 KB	172 MB	34 MB	240 KB
Computation	16.90 s	16.90 s	194.96 s	32.40 s	13.74 s
Communication Cost	\$0.001569	\$0.010610	\$0.016215	\$0.003222	\$0.000022
Computation Cost	\$0.000253	\$0.000253	\$0.002924	\$0.000486	\$0.000206
Total Cost	\$0.001822	\$0.010863	\$0.019139	\$0.003708	\$0.000228

Table 6: Weekly server costs per client needed to support private SCT auditing using the PIR-based approach of Henzinger et al. [43]. Following [43, 28], 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 [43], 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 [41], HintlessPIR [55] and YPIR) do not require hints. 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) [43].

nication costs. Compared to the communication needed by Chrome's k-anonymity-based approach for private SCT auditing [28], (which does not provide cryptographic privacy), the communication requirement using YPIR is 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 [23]. 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 symmetric cryptography or even no cryptography at all) [85, 31, 8, 36, 13, 40].

Kushilevitz and Ostrovsky [50] gave the first single-server PIR scheme based on additively homomorphic encryption. Subsequently, single-server PIR has been constructed from many number-theoretic assumptions [18, 34, 30, 20, 12]. Of particular note are the lattice-based constructions, which yield the most (concretely)-efficient constructions of single-server PIR [66, 5, 4, 33, 76, 3, 1, 72, 73, 67, 26, 43, 41, 55].

Reducing the computational cost of PIR. There are many techniques to reduce the computational burden of PIR. Batch PIR [9, 46, 38, 4, 73] allows a client to retrieve many elements from the database with low overhead relative to the cost of a single query. In *stateful PIR* [77, 49, 72, 63], 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 online computation [54, 74, 84, 86]. Notably, several of these constructions only rely on symmetric cryptography [74, 84, 86] 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* [19, 14], the server 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 [58] 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 [75].

PIR variations. A number of recent works have also sought to strengthen PIR to provide security in the presence of *malicious* servers [83, 10, 24, 29, 27]. Other extensions of PIR include retrieving records by keyword instead of index [22]; this case can be reduced to standard PIR [22, 3, 64, 78].

Acknowledgments. We thank Alexandra Henzinger for helpful pointers on the Tiptoe implementation, Baiyu Li for advice on parameter selection and benchmarking for HintlessPIR, and Kevin Yeo for suggestions related to keyword PIR. We thank the Usenix Security reviewers for their helpful comments. 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

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

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

[3] 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.

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

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

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

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

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

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

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

[11] 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.

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

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

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

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

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

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

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

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

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

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

[22] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, 1998. https://eprint.iacr.org/1998/003.

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

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

[25] 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.

[26] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, scalable, single-server private information retrieval. *IACR Cryptol. ePrint Arch.*, 2022. https://eprint.iacr.org/2022/981.

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

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

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

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

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

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

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

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

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

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

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

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

[39] 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.

[40] 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.

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

[42] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private web search with Tiptoe. In *SOSP*, 2023. https://github.com/ahenzinger/underhood/commit/f053a81.

[43] 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.

[44] 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/commit/e9020b0.

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

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

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

[48] Daniel Kales, Olamide Omolola, and Sebastian Ramacher. Revisiting user privacy for certificate transparency. In *Euro S&P*, 2019.

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

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

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

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

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

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

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

[56] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz. Hintless single-server private information retrieval. In *CRYPTO*, 2024. https://github.com/google/hintless_pir/commit/4be2ae8.

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

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

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

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

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

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

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

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

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

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

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

[68] Samir Jordan Menon and David J. Wu. YPIR: High-throughput single-server PIR with silent preprocessing. *IACR Cryptol. ePrint Arch.*, 2024. https://eprint.iacr.org/2024/270.

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

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

[71] 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.

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

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

[74] Muhammad Haris Mughees, I Sun, and Ling Ren. Simple and practical amortized sublinear private information retrieval. *IACR Cryptol. ePrint Arch.*, 2023. https://eprint.iacr.org/2023/1072.

[75] Hiroki Okada, Rachel Player, Simon Pohmann, and Christian Weinert. Towards practical doubly-efficient private information retrieval. *IACR Cryptol. ePrint Arch.*, 2023. https://eprint.iacr.org/2023/1510.

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

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

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

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

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

[81] 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.

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

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

[84] 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. https://eprint.iacr.org/2023/1607.

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

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