Hintless Single-Server Private Information Retrieval

Baiyu Li[1][0000−0003−1088−9328], Daniele Micciancio[2][0000−0003−3323−9985], Mariana Raykova[1], and Mark Schultz-Wu[2][0000−0001−5761−9662]⋆

[1] Google {baiyuli,marianar}@google.com
[2] UCSD {daniele@cs,mdschultz@eng}.ucsd.edu

Abstract. We present two new constructions for private information retrieval (PIR) in the classical setting where the clients do not need to do any preprocessing or store any database dependent information, and the server does not need to store any client-dependent information.

Our first construction (HintlessPIR) eliminates the client preprocessing step from the recent LWE-based SimplePIR (Henzinger et. al., USENIX Security 2023) by outsourcing the "hint" related computation to the server, leveraging a new concept of *homomorphic encryption with composable preprocessing*. We realize this concept with RLWE encryption schemes, and by leveraging the composibility of this technique we are able to preprocess almost all the expensive parts of the homomorphic computation and reuse them across multiple protocol executions. As a concrete application, we propose highly efficient matrix vector multiplication that allows us to build HintlessPIR. For a database of size 8GB, HintlessPIR achieves throughput about 6.37GB/s without requiring transmission of any client or server state. We additionally formalize the matrix vector multiplication protocol as a novel primitive that we call LinPIR, which may be of independent interest.

In our second construction (TensorPIR) we reduce the communication of HintlessPIR from square root to cubic root in the database size. We show how to use RLWE encryption with preprocessing to outsource LWE decryption for ciphertexts generated by homomorphic multiplications. This allows the server to do more complex processing using a more compact query under LWE.

We implement and benchmark HintlessPIR which achieves better concrete costs than TensorPIR for a large set of databases of interest. We show that it improves the communication of recent preprocessing constructions when clients do not have large numbers of queries or the database updates frequently. The computation cost for removing the hint is small and decreases as the database becomes larger, and it is always more efficient than other constructions with client hints such as Spiral PIR (Menon and Wu, S&P 2022). In the setting of anonymous queries we also improve on Spiral's communication.

⋆ Work performed at Google

1 Introduction

How to enable client access to a public database without revealing any information about the query to the server hosting the database? This question comes up in numerous applications such as anonymous messaging [1, 5, 43], contact discovery [11], password breach checkup [50], safe browsing [35], privacy enhanced advertising [8, 30] and many others, and has been the topic of study for the area of private information retrieval (PIR) [16, 36]. Since downloading the database is a trivial solution when the database is public, PIR constructions aim to achieve sublinear communication. Early theoretical results [9, 14, 22, 26] have shown feasibility of constructions with communication polylogarithmic in the size of the database, but have also given a linear computation lower bound in the basic model of PIR[3]. The pursuit of a practical PIR construction has lead to a long line of works [1, 3, 4, 17, 18, 19, 24, 34, 35, 37, 38, 39, 40, 44, 45, 54, 55] that have drawn a much more complex picture of the possible trade-offs between communication and computation for constructions.

One approach to obtain better efficiency has been to consider multi-server settings where the database is held by two or more servers that are assumed to be non-colluding. Two/multi-server PIR constructions [12, 16, 32, 37] offer better efficiency (and even information-theoretic security guarantees [16, 21, 32]), but they require the significant assumption that the servers are non-colluding. While there are settings which match this security model, our focus in this paper is on scenarios where we can rely only on a single server.

In the single-server PIR setting, an emerging paradigm that has enabled concrete reductions in the online computational cost to the server (even achieving sublinear computation) has been the model of preprocessing. In this model, the server and the client preprocess the database to obtain auxiliary information, which is stored and then used when processing the PIR queries, with the goal of reducing the online (query-time) communication and computation. In many constructions the (server side) preprocessing is client specific, i.e. depends on private input from the client, and therefore needs to be computed by the client in an interaction with the server [18, 37, 45, 55]. On the client side, the result of the preprocessing depends on the database, and it is often called a "hint", as it provides partial information about the database content.

Two recent works that use preprocessing and hints to design practically efficient PIR protocols are [34, 40]. However, the two works optimize PIR protocols along different dimensions. The SimplePIR protocol proposed in [34] minimizes the server's query processing time, but at the cost of high communication. The Spiral PIR protocol proposed in [40] minimizes communication at the cost of substantially higher computation times.

In this paper we propose new PIR protocols that are practically efficient both in terms of query processing times and communication complexity, combining the best features of [34, 40]. Moreover, we do so while avoiding the use of (client

[3] without preprocessing

side) hints and (client dependent) preprocessing, yielding usability benefits to our scheme that we summarize later.

In the rest of the introduction we first discuss the benefits of not using hints in PIR protocols. Then we provide a more detailed description of previous protocols, including Spiral PIR and SimplePIR which is the starting point of our work. Finally, we describe the new techniques developed in our work to design improved, practically efficient PIR protocols that do not require client side hints.

The problem with preprocessing

The cost of preprocessing can be substantial, both in terms of computation (typically on the server side), and communication (in some parameter regimes yielding worse bandwidth than trivially transmitting the entire database to the client.) But, since preprocessing is query-independent, it can be performed off-line, in advance. Moreover, the result of preprocessing can be re-used across several queries, reducing its amortized (per query) cost. Still, client-specific preprocessing for the server, and database-specific hints for the client, have a number of drawbacks:

- Database-dependent storage on the client can be prohibitive in some settings, especially if a client wishes to query multiple databases. For storage-constrained clients, this may require clients to evict "old" hints before enough queries have been issued to a database to amortize the high cost of the hint to something concretely reasonable.
- When the database is updated, all hints need to be recomputed to maintain correctness across all clients. For highly dynamic databases, this may add significant communication overhead[4] to keep the hints in sync. These required communications are not associated with any client query, and therefore work *against* the amortization argument for the communication required to transfer the hint. Such dynamically-updating databases are common in some areas, such as real-time data (e.g. a database of real-time stock prices).
- Another example where hints may create significant additional overhead is the "streaming setting" defined in [40] where the same query is executed across several different databases. This setting is relevant to a common technique for fitting a database with large B-bit entries into a PIR scheme that leverages homomorphic computation on b-bit numbers. One can *shard* the database into B/b databases of b-bit numbers, and issue queries to all B/b databases simultaneously. If each database requires a hint, this increases the storage requirement on the client by a factor B/b, which may quickly become prohibitive in practice.

[4] While in principle one can dynamically update the hint, this has many practical difficulties. We direct readers to the application of SimplePIR to Certificate Transparency Auditing [34, Section 7.2], where the SimplePIR authors preferred to have clients cache "stale" queries and periodically redownload the entire hint, rather than leverage their dynamic hint update mechanisms [34, Appendices C.3 and E.3].

– Client-dependent state stored on the server is also undesirable, if not just for the fact that increases storage requirements on the server. But that's not the only issue. Another downside is that even though PIR completely hides the *content* of the clients' queries, it does not provide anonymity regarding *which client* is querying the database at *which time*. In fact, if the server requires client-dependent state, the server must have knowledge of this information for correct protocol execution. This linkability can be lessened by rotating the client-dependent state regularly, but yet again this cuts against the amortization argument. Note that there have been several efforts to provide anonymity for user traffic, such as Apple's iCloud Private Relay [6], Google One VPN [28], Privacy Sandbox IP Protection [29], Tor [52]. The classical PIR model (without client-dependent state on the server) allows seamless composition with such solutions.

In this work we propose practically efficient protocols that address all these issues by avoiding the use of hints, and the presence of client-dependent state on the server.

Recent work

SimplePIR [34] proposed a preprocessing solution, where the hint does not depend on any client private information. It can therefore be computed independently by the server, and can also be reused across clients. SimplePIR demonstrated how a database-dependent (but client and query-independent) hint can enable significant speedups in the server's query processing time. Moreover, the initial high cost to transmit the database-dependent hint may be amortized over the total number of queries each client makes, so provided each client makes sufficiently many queries, the overall cost may be minimized.

The same paper presents a second construction, DoublePIR, which theoretically reduces the hint size to be independent of the *number* of database records. This hint is still *strongly* dependent on the *size* of each record though. For example, for records of size $\geq$ 256B, practically DoublePIR seems to often have hints that are many times the size of the entire database [7, Figure 5.1].

A different technique to minimize communication has the client send a compressed query, which must be expanded (using homomorphic encryption) at the server before processing the query. Many papers [3, 4, 24, 39, 40, 44] have explored variations of this general approach, and Spiral PIR [40] is the most recent and best performing paper in this category. This construction uses a "Packed Regev" RLWE-based ciphertext [48], which is homomorphically expanded to a GSW-type ciphertext [27], before proceeding with the rest of the protocol. This homomorphic expansion is concretely expensive, and the cost of Spiral PIR exceeds that of preprocessing-based constructions like SimplePIR by at least an order of magnitude in our measurements (Table 2). Moreover, while each Spiral PIR query is small, this is only true if one ignores the transmission of the encrypted key material Spiral PIR requires for homomorphic computations. Of course, similarly to SimplePIR, Spiral PIR can amortize the high cost of the transmission of this client-dependent data to the server, provided each client makes sufficiently many PIR queries.

It has been long known that preprocessing can bypass the linear computation bound for PIR [9], but until recently this was mostly a theoretical result. Recently, the Piano construction [55] showed that constructions with sublinear server computation can be concretely competitive. Piano improves the online communication and computation costs for processing a PIR query, and also reduces the client storage requirement, but requires increased preprocessing communication and computation. In fact, the client needs to stream the entire database, which may come at a large cost (though may be amortized over a large-enough number of queries).

We also mention that a concurrent, independent work has proposed [33, Appendix A.2] a PIR protocol that, similarly to ours, removes the hint from SimplePIR using RLWE. So, for completeness, we will compare our work to the PIR protocol of [33], which we call Tiptoe PIR after the name of that paper. As a reference, Tiptoe PIR increases the size of client queries by an additive factor ≈ 22MB (roughly two orders of magnitude larger than $\mathsf{HintlessPIR}$ for databases we consider in Section 7) larger than those of SimplePIR.

Technical Contributions.

Our goal is to construct a PIR scheme that requires neither database-dependent state at the clients, nor client-dependent state at the server, while at the same time being practically efficient both in terms of query computation time and communication. More specifically, we wish to stay as close as possible to the throughput of recent LWE-based constructions (namely SimplePIR), which are faster than recent RLWE-based constructions (namely Spiral PIR) by roughly an order of magnitude. Moreover, we aim to reduce the communication cost per-query even in settings where amortization arguments are unavailable, e.g. a client making a single PIR query.

We present two PIR constructions of differing asymptotic efficiency, which we name $\mathsf{HintlessPIR}$ and $\mathsf{TensorPIR}$. Both of them require neither client side preprocessing (or database-dependent state), nor client-dependent state on the server.

$\mathsf{HintlessPIR}$ The starting point for our first construction is the SimplePIR construction. This arranges the database (of size m) as a square matrix (of dimension $\sqrt{m} \times \sqrt{m}$). In this format, one can execute a PIR query by homomorphically computing a matrix-vector multiplication between this database and a $\sqrt{m}$-dimensional selection vector $\mathbf{u_i}$. This recovers the column $\mathsf{DB} \cdot \mathbf{u_i} = \mathsf{DB}_i$ that contains the desired record. One can therefore *encrypt* the selection vector $\mathbf{u_i}$, and *homomorphically* multiply by the database to obtain a PIR scheme, which is equivalent to a (heavily unoptimized) version of SimplePIR.

This simple idea has two significant issues, both related to the fact that LWE ciphertexts $[A, \mathbf{b}]$ contain a

- *pseudorandom* component $\mathbf{b}$, which contains an encoding of $\mathbf{u_i}$, and
- a *public random* A, which is independent of $\mathbf{u_i}$.

Both of these components are required to decrypt (using the secret key $\mathbf{s}$) as follows

$$\mathbf{b} - A \cdot \mathbf{s} \approx \mathbf{u_i}.$$

The source of both issues is that the matrix A is *large* — a factor $N \approx 2^{10}$ (the LWE secret dimension) larger than $\mathbf{b}$, and $N \log_2 q \approx 2^{15}$ larger than the index $i \in [\sqrt{m}]$ that is the client's input. The largeness of A implies significant overhead for both

- bandwidth, in the obvious way, and
- server compute, as homomorphically multiplying by DB requires computation of $\mathsf{DB} \cdot A$, at a cost of $N \approx 2^{10}$ times larger than a linear database scan.

SimplePIR solves both of these issues with the following optimization. It is well-known that one can shrink A to a short seed, which is expanded back to a uniformly random matrix using a random oracle. This shrinks the size of the initial encryption of $\mathbf{u_i}$, but does not help the server compute. It also does not help the server send a small reply back to the client, as one cannot find a short seed' that expands to a specific target $\mathsf{DB} \cdot A$. To fix both of these issues the server requires all client encryptions are done relative to a short seed, and then transmits $A' = \mathsf{DB} \cdot A$ as a database-dependent hint to clients. Then, when a client receives a value $\mathbf{b}' = \mathsf{DB} \cdot \mathbf{b}$ from the server, they can compute

$$\mathbf{b}' - A' \cdot \mathbf{s} = \mathsf{DB} \cdot (\mathbf{b} - A \cdot \mathbf{s}) \approx \mathsf{DB} \cdot \mathbf{u_i}.$$

We modify the SimplePIR construction by replacing the local (client side) computation of $A' \cdot \mathbf{s}$ that required the hint A' with a secure protocol to compute $A' \cdot \mathbf{s}$ (with the help of the server). We view this as a mild extension of standard PIR, that we call linear PIR.

Linear PIR The secure computation of $(A', \mathbf{s}) \mapsto A' \cdot \mathbf{s}$ resembles the original matrix-vector multiplication $\mathsf{DB} \cdot \mathbf{u_i}$ used to compute the PIR query response. The only difference is that the vector is no longer a selection vector $\mathbf{u_i} \in \{0,1\}^{\sqrt{m}}$, but an LWE secret $\mathbf{s} \in \mathbb{Z}_q^N$, and the databases A', DB are of different sizes. We call this more general functionality *linear PIR*, and note that it gives a way to securely compute a multiplication $A' \cdot \mathbf{s}$, where A' is a public matrix, and $\mathbf{s}$ is a secret vector. Similarly to standard PIR, our goal is to securely compute this product in lower bandwidth than the trivial solution of transmitting A' to the client. This matrix-vector multiplication functionality appears to be independently useful — it was recently used in the private web search construction of Henzinger et al. [33].

In these terms, the SimplePIR protocol can be viewed as reducing a PIR query (to the database DB) to a linear PIR query (to the hint $A' := \mathsf{DB} \cdot A$), which is solved via the trivial protocol of transmitting A' to the client. As A' is smaller than DB, this gives some bandwidth savings, and (after computing $A' := \mathsf{DB} \cdot A$, which is expensive for large databases) yields a practically fast protocol.

We define a novel linear PIR protocol that we call NTTlessPIR (Section 4), which suffices to replace the linear PIR query implicit to SimplePIR, and yield a hintless variant of SimplePIR (HintlessPIR, Section 5). Note that NTTlessPIR may additionally be used independently of SimplePIR as a full-fledged PIR protocol, though we find performance benefits[5] when using it solely to remove the hint A' from SimplePIR, so we focus on this in our work.

NTTlessPIR proceeds by using RLWE-based homomorphic encryption to securely compute the aforementioned matrix-vector multiplication. This is done using a preexisting homomorphic matrix-vector multiplication algorithm [31]. This algorithm homomorphically computes $(A', \mathsf{Enc}(\mathbf{s})) \mapsto \mathsf{Enc}(A' \cdot \mathbf{s} \bmod p)$ for so-called Number-Theoretic Transform (NTT) friendly moduli p.

We show that in the setting of linear PIR, where one does not need to perform further computation on $\mathsf{Enc}(A' \cdot \mathbf{s})$, one can extend this algorithm to general moduli Q by computing $A' \cdot \mathbf{s} \bmod p_i$ for many NTT-friendly primes p_i. Then, one may recover $A' \cdot \mathbf{s}$ over the integers using the Chinese Remainder Theorem. We also show that one may instantiate this algorithm using an atypically small amount of encrypted key material, namely a single rotation key. These, combined with several other non-asymptotic optimizations (summarized in Appendix E.1), suffice to instantiate NTTlessPIR. Despite these optimizations, the practical efficiency of the scheme is still lacking. We fix this via an asymptotic speedup of the underlying homomorphic algorithm (and many others) using a technique we call homomorphic encryption with composable preprocessing.

Homomorphic Encryption with Composable Preprocessing The high-level idea behind homomorphic encryption with composable preprocessing (Section 3) is similar to SimplePIR[6], albeit in the setting of RLWE-based encryption, and for a wider class of computations. For SimplePIR, the server leveraged that it knew A before protocol execution to precompute the hint $A' = \mathsf{DB} \cdot A$, and remove the computation and transmission of this from the online portion of the protocol. We develop analogous optimizations for fundamental RLWE-based homomorphic operations, namely gadget products and associated operations, like gadget-based key-switching. More importantly (and differently than SimplePIR),

[5] In particular, we are able to get considerable (though non-asymptotic) speedups in server processing time, which was our primary goal in this work. NTTlessPIR used in isolation *significantly* improves the server preprocessing of SimplePIR (and schemes that build on it) from $O(mN)$ to $O(m \log n)$, and is likely of independent interest in applications where minimizing this quantity is of primary importance.

[6] Our optimization is even compatible with a SimplePIR-type "hint" to reduce our per-query bandwidth. As it has a smaller impact (2× reduction) in our setting than that of SimplePIR ($2^{10}\times$ reduction), we instead omit it to achieve our goal of no database-dependent state on our clients.

we show that our preprocessing is *composable*, e.g. we can preprocess not only ciphertexts corresponding to the initial input to the protocol, but also intermediate ciphertexts that arise during the homomorphic evaluation of complex circuits.

This is done by identifying a certain invariant that many RLWE-based homomorphic operations preserve. Namely, if one has an input ciphertext $(a, b) = \mathsf{Enc}_v(m)$, and a collection of encrypted key material $\{(a_i, b_i)\}_i = \{\mathsf{Enc}(f_i(v))\}_i$, many homomorphic operations produce an output ciphertext (a'', b'') where a'' depends only on (and can be computed deterministically from) a and $\{a_i\}_i$. A trivial example is that the sum of two ciphertexts (a, b) and (a'_0, b'_0) has $a'' = a + a'_0$. More interestingly, in Section 3 we show that this occurs also in gadget-based key-switching. Moreover, this property is preserved when combining several operations together, allowing us to apply the preprocessing optimization to entire circuits. This class of circuits includes algorithms of practical interest such as the matrix-vector multiplication algorithm of [31], and the RLWE expansion algorithm of [15]. This list is non-exhaustive, and more applications are certainly possible, but in this work we focus on the operations that are needed for our application to concretely efficient PIR protocols.

We next describe how we use the aforementioned invariant to speedup homomorphic computation. For gadget-based key-switching, one is input a ciphertext $\mathsf{ct} = (a, b)$, and collection of encrypted key material $\mathsf{ksk} = \{(a'_i, b'_i)\}_{i \in [\ell]}$, and must compute a certain function $F(a)$ of a. After this function $F(a)$ is computed, the rest of the homomorphic computation amounts to computing a certain linear combination of the input ciphertexts, e.g. highly efficient operations. We have the server precompute $F(a)$, and then replace the superlinear-time computation of $F(a)$ with a memory access, yielding a linear-time algorithm for the homomorphic computation.

In more details, the so-called "gadget product" computes $F(a)$ in time $O(\ell n \log n)$, via computing $O(\ell)$ NTTs, where n is the RLWE ring degree. This is a common sub-routine in lattice-based cryptography, and often heavily contributes to the cost of protocols (to the point that some papers summarize their protocol's complexity by counting the number of NTTs they require). In our protocols, we avoid having the server compute any (online) NTTs, via precomputing them offline.

This does impose some overhead. If a client sends a new ciphertext $(a_{\mathsf{new}}, b_{\mathsf{new}}) = \mathsf{Enc}_v(m_{\mathsf{new}})$, the server is unable to reuse the preprocessing for the old ciphertext (a, b) on this query. In our application to PIR, each new PIR query would require new server preprocessing, e.g. we have not accomplished much yet. This is addressed by having the server publish a single short seed that may be expanded via a random oracle to a specific value a^*, and have all clients encrypt their queries relative to this value, e.g. produce ciphertexts (a^*, b). We recall that in (R)LWE encryption, different a's can be used to encrypt multiple (in fact, arbitrarily many) messages under the same key s. However, each ciphertext must use its own (independently chosen) value of a for encryption to be secure. Still, reusing the same a is allowed[7] when encrypting multiple messages under *different* keys $\mathbf{s}_1, \mathbf{s}_2, \ldots$. This is a very good match for our PIR application where clients can sample fresh RLWE secret keys[8] for each query. This allows our server to perform the aforementioned preprocessing in a secure and efficient way (reusing the same "a" values), independently of the number of clients or queries it handles, and the number of databases it maintains.

This technique results in a $O(\log n)$ speedup in our homomorphic computations. Moreover, the majority of the server's computation is extremely efficient operations, namely coordinate-wise sums and products of vectors (along with occasionally permuting a vector). These facts combine to yield a practically efficient RLWE-based protocol, with performance characteristics much closer to SimplePIR than Spiral PIR.

We finally describe our scheme HintlessPIR, which uses SimplePIR to reduce a PIR query to a Linear PIR query on the SimplePIR "hint" $\mathsf{DB} \cdot A$, which we respond to with our Linear PIR scheme NTTlessPIR. We use homomorphic encryption with composable preprocessing (and several other optimizations) within HintlessPIR, leading to a practical scheme.

We summarize the theoretical efficiency of HintlessPIR in Figure 1, where we compare it with the state of the art (fastest in practice[9]) lattice-based single server protocols predating our work (SimplePIR and Spiral PIR,) as well as the hintless variant of SimplePIR (Tiptoe PIR) described in concurrent, independent work [33, Appendix A.2]. HintlessPIR improves on SimplePIR by requiring that clients download database-dependent state, without impacting performance too much. We find that HintlessPIR's performance matches that of SimplePIR, up to lower-order terms in the size of the database, without requiring clients download any database-dependent state. This is with the exception of the size of the server response, which is (asymptotically) a constant factor larger[10] than SimplePIR's server response. This suggests that as $m \to \infty$, the overhead of HintlessPIR (compared to SimplePIR) should approach zero on all metrics except for the server response, all while removing the database-dependent hint from SimplePIR. Note that while Tiptoe PIR is similarly "hintless", its per-query communication is asymptotically worse ($O(\sqrt{m}+n^2)$ compared to $O(\sqrt{m}+n)$), yielding concretely worse per-query bandwidth (by two orders of magnitude)

[7] This is the idea at the basis of the "amortized LWE" cryptosystem of [46], as well as the property SimplePIR leverages to securely allow all clients to use the same hint.

[8] Of larger impact is that clients have to resample any encrypted key material they use. We minimize the use of such key material in our protocol to reduce this cost.

[9] There has been another recent practically-fast single-server PIR scheme, namely Piano [55]. This scheme requires the entire database be streamed to a memory-constrained client in a preprocessing step. As we are not modelling memory-constraints on clients, in our setting this protocol is essentially equivalent to the trivial PIR scheme that transmits the whole database to the client in a preprocessing step, so we will not formally compare our work to theirs.

[10] In our current implementation, this constant factor is somewhat large — ≈ 33× larger. We discuss several optimizations that would reduce this to ≈ 9× larger in Appendix E.1, though they are not currently implemented.

than HintlessPIR, while also being slower by constant factors. We discuss this in more detail at the end of the introduction, and in Section 7.

We have implemented the scheme (Section 7), and summarize our concrete findings regarding the scheme later in the introduction.

Scheme	Off. Comm.	Off. Comp.	On. Comm.	On. Comp.	C. State	S. State
SimplePIR [34]	$n\sqrt{m}$	nm	$\sqrt{m}$	m	Yes	No
Spiral PIR* [40]	$O(n)$	$O(n(\log m)^2)$	$O(\log m)$	$\tilde{O}(m)$	No	Yes
Tiptoe PIR [33]	$O(1)$	nm	$O(\sqrt{m}+n^2)$	$m+O(n\sqrt{m})$	No	No
HintlessPIR	$O(1)$	$nm+\widetilde{O}(\sqrt{m}n)$	$O(\sqrt{m}+n)$	$m+O(n\sqrt{m})$	No	No
TensorPIR	$O(1)$	$nm+O(n^2)$	$O(m^{1/3}+n)$	$m+O(nm^{2/3})$	No	No

Fig. 1. Comparison of the Asymptotic (Offline and Online) Communication and (Server) Computation of practically-efficient single-server PIR schemes, as well as whether the schemes require client-side (database-dependent) state, or server-side (client-dependent) state. Throughout, m is the number of records in the database, and n the (R)LWE secret dimension, typically $\in [2^{10}, 2^{12}]$. Computational costs are measured in $\mathbb{Z}_{2^{32}}$ operations and elements of $\mathbb{Z}_{2^{32}}$. Tiptoe PIR is our name for the "hintless" variant of SimplePIR described in Appendix A.2 of [33]. Costs for SpiralPIR are imprecise estimates, as the dependence of the many parameters of SpiralPIR on m and n is not discussed in [40].

TensorPIR So far, the PIR schemes we have constructed have bandwidth $\Theta(\sqrt{m})$. We next discuss a PIR scheme which enables us to get $\Theta(\sqrt[3]{m})$ bandwidth, via a more complex construction. We call this scheme TensorPIR, for reasons that will become apparent soon.

The high-level idea is to note that if we transmit *two* selection vectors $\mathbf{u}_{\mathbf{i_0}}, \mathbf{v}_{\mathbf{i_1}}$ of dimensions $d_{\mathbf{u}}, d_{\mathbf{v}}$, then we can take their homomorphic tensor product to obtain a selection vector $\mathbf{u}_{\mathbf{i_0}} \otimes \mathbf{v}_{\mathbf{i_1}}$ of dimension $d_{\mathbf{u}} d_{\mathbf{v}}$. For $d_{\mathbf{u}} = d_{\mathbf{v}} = d_{\mathbf{w}} = \Theta(\sqrt[3]{m})$, this high-level sketch would suffice to achieve our claim.

The issue with this high-level idea is that the LWE-based homomorphic tensor product yields massive ciphertexts. In particular, rather than containing a component $\mathsf{DB} \cdot A$ of dimension $\sqrt{m} \times N$ for LWE secret dimension $N \approx 2^{10}$ (which was already problematic), the homomorphic tensor product contains matrices of size $\sqrt[3]{m}(N^2+2N)$. Typically one would include encrypted key material called a relinearization key, to convert these ciphertexts back to standard LWE ciphertexts, but the bandwidth to transmit these vectors of $\Omega(N^3)$ dimension is much too large for our application. Instead, we show how the client can upload certain RLWE encryptions, which are vectors of $O(n)$ dimension, of their LWE secret key to have the server homomorphically compute the values the client requires for decryption.

This is conceptually the same as our hint removal for SimplePIR, though practically it is more complex. The client now computes the decryption equation

$$\mathsf{DB} \cdot (\mathbf{b}_0 - A_0 \cdot \mathbf{s}) \otimes (\mathbf{b}_1 - A_1 \cdot \mathbf{s}).$$

After distributing terms, there is now one term that depends on $\mathbf{s} \otimes \mathbf{s}$, and two terms that depends on $\mathbf{s}$. We show in Section 6 that homomorphic computation of this decryption equation reduces to homomorphic computation of the quadratic form (as well as two simpler versions of this expression)

$$\sum_{i \in [d_{\mathbf{u}}]} \langle \mathbf{a}_i, \mathbf{s} \rangle * (\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s}),$$

where $\mathbf{a}_i$ is the ith row of A_0 and DB_i are certain $\sqrt[3]{m} \times \sqrt[3]{m}$-dimensional submatrices of DB. The server can perform this computation by

- having the client pack $\langle \mathbf{a}_i, \mathbf{s} \rangle$ for all i into a single ciphertext, and use the RLWE expansion algorithm of [15] to expand it to encryptions of the constants $\langle \mathbf{a}_i, \mathbf{s} \rangle$, and
- using NTTlessPIR to homomorphically compute $\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s}$ for each $i \in [d_{\mathbf{u}}]$.

Note that these computations are still amenable to our preprocessing optimization. We performed microbenchmarks on the various components of TensorPIR, and found that for large enough databases (e.g. $\geq$ 1TB) TensorPIR requires much smaller bandwidth while having comparable throughput.

Implementation. We implemented the HintlessPIR construction, which we believe offers practically more efficient parameters for the majority of databases. Our benchmarks demonstrate that for a single initial query, HintlessPIR achieves better communication than both SimplePIR and Spiral PIR, where we count the hint and all parameters that need to be transmitted in order to make the first query. We find that our protocol has lower bandwidth until one is able to reuse a hint for ≈ 50 to 100 SimplePIR queries to the same database, and our bandwidth advantage over Spiral holds for the first 3 to 5 queries. Our computation cost is always better than Spiral PIR, and the overhead that we incur over SimplePIR for moving the hint dependent computation to the server is moderate: the time spent on downloading the SimplePIR hint over a 85Mbps Internet connection to a mobile device is comparable to making 5 to 20 queries in HintlessPIR protocol for typical databases. Moreover, we find that our additional server computation does become small as $m \to \infty$. For example, for a database of 2^{30} records and total size ≈ 8.5GB, our HintlessPIR protocol is only $\approx 25\%$ slower than SimplePIR, and has server preprocessing that is only $\approx 1\%$ slower than that of SimplePIR.

In comparison to Tiptoe PIR [33, Appendix A.2], which is similarly "hintless", our bandwidth is much better. Our hint-removal overhead is 2 orders of magnitude smaller than that of Tiptoe PIR, with running time between 1× to 3× faster. Tiptoe PIR has mildly more performant server preprocessing[11]. With

[11] Its server preprocessing is identical to SimplePIR, so the gap between its cost and HintlessPIR's costs similarly vanishes as $m \to \infty$.

the exception of the smallest database size[12] we consider (where Tiptoe PIR is 3× faster), $\mathsf{HintlessPIR}$'s server preprocessing is at most 12% slower than Tiptoe PIR's. Tiptoe PIR's server responses are $\approx 3.5\times$ smaller than ours, though due to their massive client queries their total bandwidth is still over an order of magnitude larger.

Asymptotically, $\mathsf{TensorPIR}$ should outperform $\mathsf{HintlessPIR}$ for extremely large databases that take advantage of its smaller $\Theta(\sqrt[3]{m})$ query and response sizes. Concretely (Section 7.3), we find benefits starting with extremely large databases of 2^{40} 1B records (1100GB total), where we microbenchmark $\mathsf{TensorPIR}$ to have 38MB bandwidth, compared to $\mathsf{HintlessPIR}$'s 103MB bandwidth, and Tiptoe PIR's 115MB bandwidth. All three schemes have estimated throughput an order of magnitude better than Spiral PIR (17MB bandwidth) for this database, and do not require transmitting the 69GB SimplePIR hint to clients, which is especially onerous in this setting.

2 Preliminaries

2.1 Mathematical Background

For $n \in \mathbb{N}$ we write $[n] := \{0, 1, \ldots, n-1\}$.

Notation for Different Vector Spaces Throughout, we use bold-face $\mathbf{a}$ to write a vector and upper-case A for a matrix. We write $[\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_n]$ for the matrix obtained by horizontal concatenation of the vectors $\mathbf{a}_i$, and $(\mathbf{a}_0, \mathbf{a}_1, \ldots, \mathbf{a}_n) := [\mathbf{a}_0^t, \mathbf{a}_1^t, \ldots, \mathbf{a}_n^t]^t$ for vertical concatenation. We write $\mathrm{diag}(A)$ for the main diagonal of the (square) matrix A. We write $\mathsf{rot}^{\circ i}(\mathbf{a})$ to denote the cyclically rotated vector, and for a matrix A we write $\mathsf{rot}^{\circ i}(A)$ to denote applying $\mathsf{rot}^{\circ i}$ to each column of A independently. We define $\mathrm{diag}_i(A) = \mathrm{diag}(\mathsf{rot}^{\circ i}(A^t)^t)$ for the ith generalized diagonal of A.

We will require computations involving basis vectors of different dimensions. For clarity, we will use the notation $\mathbf{u_i}, \mathbf{v_i}, \mathbf{w_i}$ to refer to the ith standard basis vector in dimensions $d_{\mathbf{u}}$, $d_{\mathbf{v}}$, $d_{\mathbf{w}}$, respectively. We write $\|\mathbf{x}\|_\infty = \max_i |\mathbf{x}_i|$.

As our work will use four different products on linear-algebraic objects, for clarity we will avoid leaving products implicit. We will write matrix-vector and matrix-matrix multiplication with $\cdot$, e.g. $A \cdot B$ and $A \cdot \mathbf{b}$. We will write polynomial multiplication with $*$, e.g. $a * b$. We will write Hadamard (element-wise) multiplication with $\circ$, e.g. $(\mathbf{a} \circ \mathbf{b})_i = \mathbf{a}_i \mathbf{b}_i$. Hadamard multiplication of vectors of polynomials corresponds to element-wise (polynomial) multiplication of each component. We will write Kronecker multiplication (or the "tensor product") of

[12] For this database size, Tiptoe PIR's bandwidth is 3× larger than the trivial PIR scheme of transmitting the entire database. Our bandwidth is $\approx 1/16$ of the cost of this trivial protocol.

matrices/vectors as $\otimes$. This is defined as the block-matrix

$$A \otimes B = \begin{pmatrix} A_{1,1} \cdot B & A_{2,1} \cdot B & \dots \\ A_{2,1} \cdot B & A_{2,2} \cdot B & \dots \\ \vdots & \vdots & \ddots \end{pmatrix}.$$

Tensor product satisfies the "mixed-product property" $(A \cdot B) \otimes (C \cdot D) = (A \otimes C) \cdot (B \otimes D)$, whenever A, B, C, D are such that all of the above matrix products are well-defined.

Polynomial Rings We consider only power-of-two cyclotomic rings $R_n := \mathbb{Z}[X]/(X^n + 1)$, where n is a power of 2. We write $R_{n,q} := R_n/qR_n$, and we say $R_{n,q}$ is NTT-friendly if q is a product of distinct primes q_i such that $q_i \equiv 1 \bmod 2n$. We will often abuse notation and refer to this as solely a property of q when the choice of n is unambiguous. For NTT-friendly prime modulus q, there is a ring isomorphism between $R_{n,q} \cong (\mathbb{Z}_q^n, +, \circ)$, which amounts to evaluating the polynomial on certain roots of unity in $\mathbb{Z}_q$. The forward direction of this isomorphism is denoted as NTT, and the inverse direction as iNTT. We write a polynomial a in the coefficient domain, and $\widehat{a}$ in the evaluation (or NTT) domain.

We will additionally need the Chinese Remainder Theorem, or the isomorphism between the rings $\mathbb{Z}_P \cong \prod_i \mathbb{Z}_{p_i}$ when $P = \prod_i p_i$ is a product of coprime integers. We refer to both parts of these isomorphisms as $\mathsf{CRT} : \mathbb{Z}_P \mapsto \prod_i \mathbb{Z}_{p_i}$, and iCRT for the inverse isomorphism. Note that this isomorphism additionally implies an isomorphism $R_{n,P} \cong \prod_i R_{n,p_i}$. We will abuse notation and use CRT and iCRT to refer to these isomorphisms as well.

2.2 Probability Background

For a distribution $\mathcal{D}$, we will write $x \leftarrow \mathcal{D}$ to denote a random sample from $\mathcal{D}$. For a set S, we will write $x \leftarrow\!\!\$\ S$ to denote a random sample from the uniform distribution on S. We write χ_σ for a centered binomial $\sum_{i \in [\sigma^2]} X_i - X_i'$, where X_i, X_i' are i.i.d. uniform on $\{0, 1\}$. We write χ_σ^n for the corresponding distribution on $\mathbb{Z}^n$ with independent components. We will also require standard notions of sub-Gaussian and sub-Exponential random variables. See [53] for an introduction to this theory, or Appendix A for a collection of facts that we will use.

2.3 Lattice-Based Hardness Assumptions

We include in Appendix B some background materials regarding the hardness of the LWE and RLWE problems, including in the setting (important to our work) where one reuses the public randomness (e.g. random matrix A or polynomial a) of the (R)LWE samples.

2.4 LWE and RLWE-based Encryptions

We will exclusively use LWE (and RLWE)-based encryption where the ciphertext $[A, \mathbf{b}]$ has A expanded from some short seed via a random oracle, so we adapt our notation to this setting.

Definition 1 (Private-key LWE-based Encryption). *Let $N, Q, \Delta, d \in \mathbb{N}$. Let $\sigma > 0$. Let* RO *be a random oracle. Private-key* LWE *Encryption is defined to be the tuple of algorithms*

- $\mathsf{KGen}(1^\lambda)$: *Samples $\mathbf{s} \leftarrow \chi_\sigma^N$, and returns this value.*
- $\mathsf{Enc}_\mathbf{s}(\mathbf{m}; \mathsf{seed})$: *Samples $A := \mathsf{RO}(\mathsf{seed}) \in \mathbb{Z}_Q^{d \times N}$ and $\mathbf{e} \leftarrow \chi_\sigma^d$, and outputs* $\mathbf{b} := A \cdot \mathbf{s} + \mathbf{e} + \Delta \mathbf{m}$.
- $\mathsf{Dec}_\mathbf{s}(C)$*: Parses $[A, \mathbf{b}] = C$, and returns* $\left\lfloor \frac{\mathbf{b} - A \cdot \mathbf{s}}{\Delta} \right\rceil$.

When seed is omitted we mean that we are not applying this optimization, e.g. A is freshly sampled. A LWE ciphertext $C = [A, \mathbf{b}]$ encrypting $\mathbf{m}$ under secret key $\mathbf{s}$ satisfies $C \cdot (-\mathbf{s}, 1) = \Delta \mathbf{m} + \mathbf{e}$. We call $\mathbf{e}$ the *error* of the ciphertext C.

Definition 2 (Private-key RLWE-based Encryption). *Let $k, q, \Delta \in \mathbb{N}$. Let $n = 2^k$, and $\sigma > 0$. Private-key* RLWE *Encryption is defined to be the tuple of algorithms*

1. $\mathsf{KGen}(1^\lambda)$: *Samples $v \leftarrow \chi_\sigma^n$, and returns this value.*
2. $\mathsf{Enc}_v(m)$*: Samples $a \leftarrow\!\$\ R_{n,q}$ and $e \leftarrow \chi_\sigma^n$, and outputs* $\mathsf{ct} = [a, a * v + e + \Delta m] \in R_{n,q}^2$.
3. $\mathsf{Dec}_v(\mathsf{ct})$*: Parses $[a, b] = \mathsf{ct}$, and returns* $\left\lfloor \frac{b - a * v}{\Delta} \right\rceil$.

We define the analogous notion of *error* for RLWE-based ciphertexts.

Plaintext Slots The native plaintext space in RLWE-based encryption is the ring $R_{n,p}$. When p is a NTT-friendly prime[13], the plaintext ring $R_{n,p}$ is isomorphic to a $\mathbb{Z}_p$-algebra $(\mathbb{Z}_p^n, +, \circ)$, usually called the "slot algebra", where addition and multiplication between polynomials in $R_{n,p}$ correspond to component-wise addition and multiplication over $\mathbb{Z}_p^n$. We denote using encode_p the inverse isomorphism from $\mathbb{Z}_p^n$ to $R_{n,p}$, and decode_p its inverse isomorphism from $R_{n,p}$ back to $\mathbb{Z}_p^n$. Such isomorphism and its inverse can be computed using iNTT and NTT over $R_{n,p}$. As such, we can also view $\mathbb{Z}_p^n$ as the plaintext space of RLWE-based encryption for suitable p, and we call $\widehat{\mathbf{a}} \in \mathbb{Z}_p^n$ the slots of a plaintext polynomial $a = \mathsf{encode}_p(\widehat{\mathbf{a}})$. We usually drop the subscript p when there is no ambiguity on plaintext modulus.

We will need the *Galois automorphism* θ_j for $j \in \mathbb{Z}_{2n}^*$, which is the ring automorphism $a(X) \mapsto a(X^j)$. For any plaintext vector $\widehat{\mathbf{a}} \in \mathbb{Z}_p^n$, we write

$$\mathsf{rot}^{\circ j}(\widehat{\mathbf{a}}) = \begin{pmatrix} \widehat{a}_{j \bmod n/2}, \ldots, \widehat{a}_{(n/2-1+j) \bmod n/2}, \\ \widehat{a}_{n/2+(j \bmod n/2)}, \ldots, \widehat{a}_{n/2+(n/2-1+j \bmod n/2)} \end{pmatrix},$$

[13] In general p can be a prime power, but in our application we only use the case where p is a prime number.

for cyclic rotation by j of the first and second half of $\widehat{\mathbf{a}}$. For power-of-two cyclotomics we have $\mathsf{encode}(\mathsf{rot}^{\circ 1}(\mathsf{decode}(a))) = a(X^5)$, e.g. rotation by one and the Galois automorphism $X \mapsto X^5$ are equivalent operations. More generally, rotation by j is equivalent to the Galois automorphism $X \mapsto X^{5^j}$ in $R_{n,p}$.

RLWE Key-Switching We will use what is known as "gadget-based" key-switching. See [23] for a more complete reference on gadgets in lattice-based cryptography.

Definition 3 (Gadgets). *Let G be an additive group. A G-*gadget $\mathbf{g}$ *of size ℓ and quality γ is a pair of a vector $\mathbf{g} \in G^\ell$ and (non-linear) mapping $\mathbf{g}^{-1} : \mathbb{Z}^\ell \to G$ such that for all $x \in \mathbb{Z}_q$, $\langle \mathbf{g}^{-1}(x), \mathbf{g} \rangle = x$, and $\left\|\mathbf{g}^{-1}(x)\right\|_\infty \leq \gamma$.*

Note that the equality $\langle \mathbf{g}^{-1}(x), \mathbf{g} \rangle = x$ must hold in the group G, e.g. in $G = \mathbb{Z}_q$ it hold $\mod q$. Typically one first builds a gadget for $\mathbb{Z}_q$, and then applies it coordinate-wise to $R_{n,q} \cong \mathbb{Z}_q^n$. One may use gadgets to key-switch.

Definition 4 (Key-switching Key). *Let $\mathbf{g}$ be a $\mathbb{Z}_q$-gadget of size ℓ and quality γ. A $\mathbf{g}$-based key-switching key (from key v_0 to v_1) is collection of ℓ RLWE encryptions ksk where $\mathsf{ksk}_i = \mathsf{Enc}_{v_1}(\mathbf{g}_i v_0)$. For a polynomial a, the $\diamond$-product is*

$$a \diamond \mathsf{ksk} = \sum_{i \in [\ell]} \mathbf{g}^{-1}(a)_i * \mathsf{ksk}_i.$$

Lemma 1. *For $\mathbf{g}$ a gadget of size ℓ and quality γ, let ksk be a $\mathbf{g}$-based key-switching key from v_0 to v_1. Let e_i be the error within ksk_i. If $\mathsf{ct} = [a, b]$ is an RLWE encryption of Δm under v_0 with error e, then $[0, b] - a \diamond \mathsf{ksk}$ is an RLWE encryption of Δm under v_1 with error $e + \sum_{i \in [\ell]} \mathbf{g}^{-1}(a)_i * e_i$.*

With certain Galois automorphisms θ_j we can homomorphically rotate the plaintext slots encrypted in a ciphertext ct, but the resulting ciphertext ct' is encrypted under the substituted secret $\theta_j(v)$ and cannot be used for subsequent homomorphic computation. We can apply a key-switching key from $\theta_j(v)$ to v on ct' to convert it back to a ciphertext that can be decrypted to the rotated slots using the secret v. Such special key-switching key is usually called a *rotation key*. We will use the notation $\mathsf{Rotate}_{\{\mathsf{ksk}\}}(\mathsf{ct}, j)$ for the procedure that starts with $\mathsf{ct} := [a, b]$, maps this to $[a(X^{5^j}), b(X^{5^j})]$, and then key-switches this from an encryption under $v(X^{5^j})$ back to v. Such operation homomorphically rotates the slots in ct by j, and the resulting ciphertext is encrypted under the same RLWE secret key. As discussed above, this requires a single $\diamond$-product.

We will require several standard homomorphic algorithms. We summarize these, and their noise growth, in Appendix D.

2.5 Linear Single-Server Private Information Retrieval

We introduce a definition that we call linear PIR (LinPIR), where rather than querying single record DB_i, a client may query an arbitrary linear combination of records $\sum_i a_i \mathsf{DB}_i$.

Definition 5 (Single-Server LinPIR with Preprocessing). *A Single-Server* LinPIR *Scheme with Preprocessing is a tuple of four algorithms that all implicitly take as input the security parameter* 1^λ*.*

1. $\mathsf{S.setup}(\mathsf{DB}) \to (\mathsf{hint}_\mathsf{C}, \mathsf{hint}_\mathsf{S})$*: Given a database* $\mathsf{DB} \in \mathbb{Z}_p^m$*, output a client hint* hint_C*, and server hint* hint_S*.*
2. $\mathsf{C.query}(\mathbf{a}, \mathsf{C}_\mathsf{hint}) \to (\mathsf{qu}, \mathsf{C}_\mathsf{state})$*: Give a linear query* $\mathbf{a} \in \mathbb{Z}_p^m$*, and the client hint* C_hint*, output a query* qu*, and client state* $\mathsf{C}_\mathsf{state}$
3. $\mathsf{S.response}(\mathsf{qu}, \mathsf{S}_\mathsf{hint})$ *: Given a query* qu*, and the server hint* S_hint*, output a server response* $\mathsf{rsp} \in \mathcal{R}$
4. $\mathsf{C.recover}(\mathsf{C}_\mathsf{state}, \mathsf{rsp}) \to \mathbb{Z}_Q^m$ *: Given the client hint* $\mathsf{C}_\mathsf{state}$*, and a server response* $\mathsf{rsp} \in \mathcal{R}$*, recover a linear combination of elements* $\sum_i a_i \mathsf{DB}_i \bmod p$*.*

Note that standard PIR is simply linear PIR where one queries a basis vector. In general, our goal is to minimize bandwidth costs in the above protocol, measured by minimizing the sizes of $\mathsf{C}_\mathsf{hint}, \mathsf{qu}$, and rsp, while still producing a concretely efficient protocol.

Definition 6 (Correctness). *Let* $\delta \in [0, 1]$*. A Single-Server* LinPIR *scheme with Preprocessing Scheme is said to be* $(1-\delta)$*-correct if for any database* $\mathsf{DB} \in \mathbb{Z}_Q^m$*, for any index* $i \in [m]$*, we have that*

$$\Pr\left[\mathsf{C.recover}(\mathsf{C}_\mathsf{state}, \mathsf{rsp}) \neq \sum_i a_i \mathsf{DB}_i : \begin{array}{r} (\mathsf{hint}_\mathsf{C}, \mathsf{hint}_\mathsf{S}) \leftarrow \mathsf{S.setup}(\mathsf{DB}) \\ (\mathsf{C}_\mathsf{state}, \mathsf{qu}) \leftarrow \mathsf{C.query}(\mathbf{a}, \mathsf{C}_\mathsf{hint}) \\ \mathsf{rsp} \leftarrow \mathsf{S.response}(\mathsf{qu}, \mathsf{S}_\mathsf{hint}) \end{array}\right] \leq \delta.$$

Definition 7 (Security). *A Single-Server* LinPIR *scheme with Preprocessing is said to be secure if for all* $(i, j) \in [m]^2$*, the distributions of* $\mathsf{query}(i)$ *and* $\mathsf{query}(j)$ *are computationally indistinguishable.*

LWEPIR: Unifying $\mathsf{SimplePIR}$ and $\mathsf{FrodoPIR}$ We will refer to the scheme of Figure 2 as LWEPIR. As mentioned in [20, Section 7.2] and [34, Section 2], this essentially recovers $\mathsf{SimplePIR}$ [34] and $\mathsf{FrodoPIR}$ [19, 20], depending on whether the database is formatted as a square matrix $\mathsf{DB} \in \mathbb{Z}_Q^{\sqrt{m} \times \sqrt{m}}$, or a row vector $\mathsf{DB} \in \mathbb{Z}_Q^{1 \times m}$.

Note that there are additional optimizations presented in [19, 34]. In particular, [19] notices that if one institutes a per-client bound on the number of queries made, one may precompute the products $H\mathbf{s}$ used in $\mathsf{C.recover}$, allowing one to only store H (client-side) during a preprocessing step. [34] notes one may use a second invocation of PIR to reduce the client-side hint to a database-independent quantity (though one that is still concretely large, see [7]). As our generalization of LWEPIR will not feature a hint H, we do not bother with either of these optimizations. While it was previously shown in [19, 34] that LWEPIR is solely a PIR scheme, it is straightforward to see that it additionally is a LinPIR scheme.

```
Server Algorithms in LWEPIR
S.setup(DB) :
    seed $\leftarrow\!\!\$\ \{0,1\}^\lambda$
    $A \leftarrow \mathsf{RO}(\mathsf{seed})$ // $A \in \mathbb{Z}_Q^{d_{\mathbf{u}} \times N}$
    $H := \mathsf{DB} \cdot A$ // $H \in \mathbb{Z}_Q^{d_{\mathbf{v}} \times N}$
    return $((H, \mathsf{seed}), \mathsf{seed})$
S.response(DB, query)
    return DB · query
```

```
Client Algorithms in LWEPIR
C.query($i$, $\mathsf{C_{hint}}$) :
    $i_0 = i \bmod d_{\mathbf{u}}, i_1 = (i - i_0)/d_{\mathbf{u}}$
    $(\mathbf{s}, \mathbf{e}) \leftarrow \chi_\sigma^N \times \chi_\sigma^{d_{\mathbf{u}}}$
    $(H, \mathsf{seed}) \leftarrow \mathsf{C_{hint}}$
    query := $\mathsf{LWE.Enc}_{\mathbf{s}}(\mathbf{u}_{\mathbf{i_0}}; \mathsf{seed})$
    $\mathbf{c}_0 := H \cdot \mathbf{s}$
    return $((\mathbf{c}_0, i_1), \mathsf{query})$
C.recover($(\mathbf{c}_0, i_1)$, rsp) :
    return $\left\lfloor \frac{\langle \mathsf{rsp}, \mathbf{v}_{\mathbf{i_1}} \rangle - \mathbf{c}_0}{\Delta} \right\rceil$
```

Fig. 2. The client and server's algorithms in LWEPIR.

Tiptoe PIR: Hintless PIR from Linearly Homomorphic Encryption We next describe a (concurrently developed) hintless variant of LWEPIR, namely the scheme described in [33, Appendix A.2], which we call Tiptoe PIR. Similarly to HintlessPIR, this protocol builds on top of LWEPIR and it eliminates the need to transmit the LWEPIR hint $H = \mathsf{DB} \cdot A$ by homomorphically computing the matrix-vector multiplication $(H, \mathbf{s}) \mapsto H \cdot \mathbf{s}$ on the server. As we will not technically require any details of Tiptoe PIR, we keep our description high-level. The main difference between this LinPIR scheme and the scheme we develop in Section 4 (NTTlessPIR) is that Tiptoe PIR utilizes the column-major (rather than diagonally-dominant) matrix-vector multiplication algorithm of [31]: the product $H \cdot \mathbf{s}$ is computed as

$$H \cdot \mathbf{s} = \sum_{i \in [N]} H_i \mathbf{s}_i, \tag{1}$$

where H_i is the ith columns of H, and $\mathbf{s}_i \in \mathbb{Z}_Q$ are scalars. Concretely, in addition to employing LWEPIR with LWE secret $\mathbf{s}$, in Tiptoe PIR the client encrypts scalars $\mathbf{s}_i$, for all $i \in [N]$, into RLWE ciphertexts $\{\mathsf{Enc}_v(\mathbf{s}_i)\}_{i \in [N]}$, i.e. each ciphertext encrypts a scalar $\mathbf{s}_i$ as a constant polynomial. The server handles the LWEPIR query as usual, and then computes the RLWE response

$$\sum_{i \in [N]} \mathsf{Enc}_v(\mathbf{s}_i) * H_i.$$

where H_i is an encoding of the ith column of H into a polynomial[14]. This LinPIR scheme inflates client queries (compared to packing $\mathbf{s}$ in a single polynomial) by a multiplicative factor $O(N)$. But the server online computation involves just

[14] For simplicity, we assume the number of rows of H is equal to the RLWE ring degree n.

ciphertext-plaintext multiplication, which is implementable without using NTTs, and does not require any encrypted key material. Further, this LinPIR scheme can be based on any *linearly homomorphic* encryption scheme, not necessarily the RLWE-based one. The resulting LinPIR scheme is relatively simple, and has good server compute, though large client queries. As we show later in Section 7, one may simultaneously obtain better server compute and smaller client queries using the techniques of the next two sections.

3 Linearly Homomorphic Encryption with Preprocessing

In this section, we detail the main technical tool used in our work to improve the performance of our PIR protocols. The main idea is that RLWE-based ciphertexts (including simple RLWE encryptions, gadget-encoded encryptions, switching keys, etc.) consists of two parts (a, b) where a is some public randomness that does not depend on the encrypted message, and is often available in advance. Moreover, the public randomness of the ciphertexts output by homomorphic operations often depends only on the public randomness of the input. This results in a cascading effect, where the public randomness of all intermediate ciphertexts occurring during the execution of a protocol can be pre-computed and pre-processed to speed up the rest of the computation.

As a matter of notation, we write $\alpha(\mathsf{ct})$ for the public randomness part of a ciphertext ct, and $\beta(\mathsf{ct})$ for the pseudorandom part of the ciphertext. For example, if $\mathsf{ct} = (a(X), b(X) = a(X) \cdot v(X) + e(X) + \Delta \cdot m(X))$ is an encryption of message $m(X)$ under RLWE key $v(X)$, then $\alpha(\mathsf{ct}) = a(X)$, and $\beta(\mathsf{ct}) = b(X)$. If $C = (\mathsf{ct}_{i,j})_{i,j}$ is a vector or matrix of RLWE ciphertexts (e.g., the encryption of a gadget-encoded message), then $\alpha(C) = (\alpha(\mathsf{ct}_{i,j}))_{i,j}$ is the public randomness of the individual components, and similarly for $\beta(C)$.

Now, let $F(m)$ be an operation that we want to evaluate homomorphically on a ciphertext $\mathsf{ct} = \mathsf{Enc}_v(m)$. In other words, there is an evaluation algorithm $\mathsf{Eval}_F(\mathsf{ek}, \cdot)$ that, with the help of an evaluation key ek, given an encryption of m outputs an encryption of $F(m)$:

$$\mathsf{Eval}_F(\mathsf{ek}, \mathsf{Enc}_v(m)) = \mathsf{Enc}_v(F(m)).$$

For several common operations F (e.g., as used in our PIR protocols) we show that there is a preprocessing function $\mathsf{Preproc}_F$ and optimized evaluation function Apply_F such that

$$\mathsf{Eval}_F(\mathsf{ek}, \mathsf{ct}) = \mathsf{Apply}_F(\mathsf{Preproc}_F(\alpha(\mathsf{ek}), \alpha(\mathsf{ct})), \mathsf{ek}, \mathsf{ct}) \tag{2}$$

where Apply_F has a much smaller evaluation cost than Eval_F. Moreover, the public randomness component of the output

$$\alpha(\mathsf{Apply}_F(g, \mathsf{ek}, \mathsf{ct})) = \mathsf{Apply}_F^{\alpha}(g) \tag{3}$$

depends only on the result of the preprocessing $g = \mathsf{Preproc}_F(\alpha(\mathsf{ek}), \alpha(\mathsf{ct}))$, and in particular, it can be computed in advance.

This allows to combine several homomorphic evaluation together. E.g., if we want to evaluate homomorphically the function composition $F \circ G(m) = F(G(m))$, we can do so using the preprocessing function

$$\begin{aligned} \mathsf{Preproc}_{F \circ G}(\mathsf{ek}_\alpha, \mathsf{ct}_\alpha) = (g_G, g_F) \quad &\text{where} \\ g_G &= \mathsf{Preproc}_G(\mathsf{ek}_\alpha, \mathsf{ct}_\alpha) \\ g_F &= \mathsf{Preproc}_F(\mathsf{ek}_\alpha, \mathsf{Apply}^\alpha_G(g_G)). \end{aligned}$$

The optimized evaluation functions $\mathsf{Apply}_{F \circ G}$ and its public randomness component $\mathsf{Apply}^\alpha_{F \circ G}$ are defined in the obvious way. Here we have assumed that Eval_F and Eval_G use the same evaluation key ek and produce only one ciphertext as output, but the composition operation is easily extended to more general setting of functions taking multiple ciphertexts as input and using different evaluation keys.

In summary, an evaluation algorithm with preprocessing for operation F is given by algorithms $\mathsf{Preproc}_F, \mathsf{Apply}_F, \mathsf{Apply}^\alpha_F$ satisfying properties (2) and (3). The algorithms are used in the obvious way, precomputing $g = \mathsf{Preproc}_F(\alpha(\mathsf{ek}), \alpha(\mathsf{ct}))$ in advance, and then evaluating $\mathsf{Apply}_F(g, \mathsf{ek}, \mathsf{ct})$ at protocol execution time after the (encrypted) inputs become available. The performance of an evaluation algorithm with preprocessing is described by the following parameters:

- The pre-computation time, i.e., the running time of evaluating $g = \mathsf{Preproc}_F(\alpha(\mathsf{ek}), \alpha(\mathsf{ct}))$. This value is computed off-line, so it is not as critical for the practical performance of an algorithm. Still, we want this cost to be reasonable.
- The size of the preprocessed information g. This value will need to be stored, and often kept in-between executions of the algorithm. So, we want to take a reasonable amount of space.
- The online computation time, i.e., the running time of $\mathsf{Apply}(g, \mathsf{ek}, \mathsf{ct})$, given the result of the preprocessing g. This will be the most critical parameter affecting performance, and should be minimized. As a side note, we remark that part of the protocol input ek, ct may already be contained in the preprocessing information g. So, in practice, there is no need to pass the whole ek, ct to Apply_F, and it is enough to provide input-dependent portion of ek, ct, namely $\beta(\mathsf{ek}), \beta(\mathsf{ct})$.

In the next subsection we give efficient evaluation algorithms with preprocessing for the homomorphic operations used by our PIR protocol. Throughout this section we will not analyze the noise growth of our algorithms, as the algorithms themselves have identical noise growth to their variants that do not take advantage of precomputation, which are all standard algorithms.

3.1 Preprocessing $\diamond$-Products and Key-Switching

We first detail our preprocessing optimization for the $\diamond$-product of Definition 4. Typically, the $\diamond$-product takes as input a gadget-based RLWE ciphertext $\mathsf{ct} =$

$[\mathsf{ct}_0, \ldots, \mathsf{ct}_{\ell-1}]$, where $\mathsf{ct}_i = \mathsf{Enc}(\mathbf{g}_i * m)$, and scalar a, and outputs

$$\sum_{i \in [\ell]} \mathbf{g}^{-1}(a)_i * \mathsf{ct}_i.$$

The operation $*$ is naively computable in $O(n^2)$ $\mathbb{Z}_q$ operations, so instead this is typically computed in the NTT domain, e.g. one takes as input a gadget-based RLWE ciphertext in the NTT domain $\widehat{\mathsf{ct}} = [\widehat{\mathsf{ct}}_0, \ldots, \widehat{\mathsf{ct}}_{\ell-1}]$, and a NTT-domain scalar $\widehat{a}$, and outputs

$$\sum_{i \in [\ell]} \mathsf{NTT}(\mathbf{g}^{-1}(\mathsf{iNTT}(\widehat{a}))_i) \circ \widehat{\mathsf{ct}}_i. \tag{4}$$

The quadratic time operation $*$ is now computable in $O(n)$ $\mathbb{Z}_q$ operations, but gadget-decomposition requires one call to iNTT and ℓ calls to NTT, e.g. $O(\ell)$ calls to an algorithm that takes $O(n \log n)$ $\mathbb{Z}_q$ operations.

We next show how this may be preprocessed to be computable in $(\ell+1)n$ $\mathbb{Z}_q$ operations, at the cost of $(\ell+1)n$ elements of $\mathbb{Z}_q$ of storage.

Lemma 2 (Preprocessing $\diamond$-products). *There exist functions* $(\mathsf{Preproc}_\diamond, \mathsf{Apply}_\diamond, \mathsf{Apply}_\diamond^\alpha)$ *such that for any* NTT*-domain polynomial* $\widehat{a}$*, and any* NTT*-domain gadget-encoded ciphertext* $\widehat{\mathsf{ct}} = [\widehat{\mathsf{ct}}_0, \ldots, \widehat{\mathsf{ct}}_{\ell-1}]$*, we have that* $\widehat{a} \diamond \widehat{\mathsf{ct}} = \mathsf{Apply}_\diamond(\mathsf{Preproc}_\diamond(\alpha(\widehat{\mathsf{ct}}), \widehat{a}), \widehat{\mathsf{ct}}, \widehat{a})$.

Moreover, $\mathsf{Preproc}_\diamond$ *runs in* $O(\ell n \log n)$ $\mathbb{Z}_q$ *operations, and produces an output of size* $(\ell+1)n$ *elements of* $\mathbb{Z}_q$*,* $\mathsf{Apply}_\diamond$ *runs in* $(\ell+1)n$ $\mathbb{Z}_q$ *operations.*

Proof. Let $g_\diamond = \mathsf{Preproc}_\diamond(\alpha(\widehat{\mathbf{c}}), \widehat{a})$ be such that

$$(g_\diamond)_i = \begin{cases} \mathsf{NTT}(\mathbf{g}^{-1}(\mathsf{iNTT}(\widehat{a}))_i) & 0 \le i < \ell, \\ \sum_{j \in [\ell]} \mathsf{NTT}(\mathbf{g}^{-1}(\mathsf{iNTT}(\widehat{a}))_j) \circ \alpha(\widehat{\mathbf{c}})_j & i = \ell. \end{cases}$$

Note that $g_\diamond \in (\mathbb{Z}_q^n)^{\ell+1}$, is such that

$$\widehat{a} \diamond \widehat{\mathsf{ct}} = \mathsf{Apply}_\diamond(\mathsf{Preproc}_\diamond(\alpha(\widehat{\mathsf{ct}}), \widehat{a}), \widehat{\mathsf{ct}}, \widehat{a}),$$

for $\mathsf{Apply}_\diamond(g_\diamond, \widehat{\mathsf{ct}}, \widehat{a}) = ((g_\diamond)_\ell, \sum_{i \in [\ell]} (g_\diamond)_i \circ \beta(\widehat{\mathsf{ct}})_i)$. Note that this function satisfies

$$\alpha(\mathsf{Apply}_\diamond(g_\diamond, \widehat{\mathsf{ct}}, \widehat{a})) = (g_\diamond)_\ell,$$

e.g. $\mathsf{Apply}_\diamond^\alpha$ is solely a function of the pre-processed data. □

We next show how to pre-process (gadget-based) key-switching.

Lemma 3 (Preprocessing Key-Switching). *There exists functions* $(\mathsf{Preproc}_{\mathsf{ks}}, \mathsf{Apply}_{\mathsf{ks}}, \mathsf{Apply}_{\mathsf{ks}}^\alpha)$ *such that, for any* RLWE *encryption* $\mathsf{ct} = \mathsf{Enc}_{v'}(m)$*, and any gadget-based key-switching key* ksk *from* v' *to* v*, we have that*

$$\mathsf{Enc}_v(m) = \mathsf{Apply}_{\mathsf{ks}}(\mathsf{Preproc}_{\mathsf{ks}}(\alpha(\mathsf{ksk}), \alpha(\mathsf{ct})), \mathsf{ksk}, \mathsf{ct}).$$

Moreover, $\mathsf{Preproc}_{\mathsf{ks}}$ *runs in* $O(\ell n \log n)$ $\mathbb{Z}_q$ *operations, and produces an output of size* $(\ell+1)n$ *elements of* $\mathbb{Z}_q$*, and* $\mathsf{Apply}_{\mathsf{ks}}$ *runs in* $(\ell+2)n$ $\mathbb{Z}_q$ *operations.*

Proof. Recall that for gadget-based key-switching it suffices to compute

$$[0, \beta(\widehat{\mathsf{ct}})] - \alpha(\widehat{\mathsf{ct}}) \diamond \widehat{\mathsf{ksk}}.$$

So, we set $g_{\mathsf{ks}} = \mathsf{Preproc}_{\mathsf{ks}}(\alpha(\widehat{\mathsf{ct}}), \alpha(\widehat{\mathsf{ct}})) = \mathsf{Preproc}_{\diamond}(\alpha(\widehat{\mathsf{ct}}), \alpha(\widehat{\mathsf{ct}}))$, and thus

$$\mathsf{Apply}_{\mathsf{ks}}(g_{\mathsf{ks}}, \widehat{\mathsf{ksk}}, \widehat{\mathsf{ct}}) = [0, \beta(\widehat{c})] - \mathsf{Apply}_{\diamond}(g_{\mathsf{ks}}, \widehat{\mathsf{ksk}}, \widehat{\mathsf{ct}}) = [0, \beta(\widehat{c})] - \alpha(\widehat{\mathsf{ct}}) \diamond \widehat{\mathsf{ksk}},$$

as desired. Note that

$$\mathsf{Apply}_{\mathsf{ks}}^{\alpha}(g_{\mathsf{ks}}, \widehat{\mathsf{ksk}}, \widehat{\mathsf{ct}}) = -(g_{\mathsf{ks}})_{\ell},$$

is solely a function of the precomputed information g_{ks}, as claimed. □

We next show that this suffices for the batch generation of the rotations

$$\{\mathsf{Enc}_v(\mathsf{encode}(\mathsf{rot}^{\circ i}(\mathbf{m})))\}_{i \in [R]},$$

from a single ciphertext $\mathsf{Enc}_v(\mathsf{encode}(\mathbf{m}))$, as well as a rotation key, e.g. a key-switching key from $v(X^5) \mapsto v$.

Lemma 4 (Pre-processing Rotations). *Let $R \in \mathbb{N}$. There exists functions* $(\mathsf{Preproc}_{\mathsf{rot}^{\circ R}}, \mathsf{Apply}_{\mathsf{rot}^{\circ R}}, \mathsf{Apply}_{\mathsf{rot}^{\circ R}}^{\alpha})$ *such that, for any* RLWE *encryption* $\mathsf{ct} = \mathsf{Enc}_v(\mathsf{encode}(\mathbf{m}))$ *of* $\mathbf{m}$ *encoded in plaintext slots, and any rotation key* ek*, we have that* $\mathsf{Apply}_{\mathsf{rot}^{\circ R}}(\mathsf{Preproc}_{\mathsf{rot}^{\circ R}}(\alpha(\widehat{\mathsf{ek}}), \alpha(\widehat{\mathsf{ct}}), \widehat{\mathsf{ek}}, \widehat{\mathsf{ct}}))$ *generates (for $i \in [R]$) the rotations* $\mathsf{Enc}_{\mathsf{ct}}(\mathsf{encode}(\mathsf{rot}^{\circ i}(\mathbf{m})))$.

Moreover, $\mathsf{Preproc}_{\mathsf{rot}^{\circ R}}$ *runs in $O(\ell n R \log n)$ $\mathbb{Z}_q$ operations, and produces an output of size $(R-1)(\ell+1)n$ $\mathbb{Z}_q$ elements, and* $\mathsf{Apply}_{\mathsf{rot}^{\circ R}}$ *runs in $(R-1)(\ell+2)n$ $\mathbb{Z}_q$ operations.*

Proof. Recall that, in NTT form, a single rotation may be computed via first mapping $\widehat{\mathsf{ct}} \mapsto \mathsf{rot}(\widehat{\mathsf{ct}})$, and then key-switching from the rotated key back to the initial key. This is to say that one may preprocess a rotation by applying n $\mathbb{Z}_q$ operations (e.g. the rotation itself), followed by an application of Lemma 3. To generate encryptions of $\mathsf{rot}^{\circ i}(\mathbf{m})$ for $i \in [R]$ rotations, iterate this process $R-1$ times. The complexity estimates reduce to $(R-1)$-times the complexity of Lemma 3, though computing the rotation of $\beta(\widehat{\mathsf{ct}})$ in each iteration increases the cost of our protocol by an additive factor n $\mathbb{Z}_q$ operations more than the cost of Lemma 3. □

3.2 Precomputing RLWE-based Matrix-Vector Multiplication

We next show that one may combine our previous tools to precompute the homomorphic evaluation of

$$(\{A_i\}_{i \in [M]}, \mathsf{Enc}(\mathsf{encode}(\mathbf{m}))) \mapsto \{\mathsf{Enc}(\mathsf{encode}(A_i \cdot \mathbf{m}))\}_{i \in [M]},$$

for some (public) set of matrices $\{A_i\}_{i \in [M]}$. We homomorphically compute this by first homomorphically computing $\mathsf{rot}^{\circ i}(\mathbf{m})$ for sufficiently many i, and then computing a simple linear combination of the encryptions of $\{\mathsf{rot}^{\circ i}(\mathbf{m})\}_i$ with constants that depend on A_j. Note that this first step is independent of the matrices A_j we will multiply by, e.g. we will only compute it once, independently of the value of M.

We preprocess the (homomorphic) diagonally-dominant matrix-vector multiplication algorithm of [31]. This relies on the following linear-algebraic fact.

Lemma 5. *Let $n, n_{\mathsf{cols}} \in \mathbb{N}$. Let $A \in \mathbb{Z}_q^{n \times n_{\mathsf{cols}}}$, and $\mathbf{m} \in \mathbb{Z}_q^{n_{\mathsf{cols}}}$. Then*

$$A \cdot \mathbf{m} \bmod q = \sum_{i \in [n_{\mathsf{cols}}]} \mathrm{diag}_i(A) \circ \mathsf{rot}^{\circ i}(\mathbf{m}) \bmod q, \tag{5}$$

We can extend this to arbitrary matrices $A \in \mathbb{Z}_q^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$ by vertically partitioning A into $M = \lceil n_{\mathsf{rows}}/n \rceil$ sub-matrices A_i of size $n \times n_{\mathsf{cols}}$, e.g. reducing the single matrix-vector product into M matrix-vector products of the form of Lemma 5.

Theorem 1 (Pre-processing Eq. (5)). *Let $M, n, n_{\mathsf{cols}} \in \mathbb{N}$, where $n_{\mathsf{cols}} \leq n$. Let $A_i \in \mathbb{Z}_p^{n \times n_{\mathsf{cols}}}$ be a collection of M square matrices. There exists functions $(\mathsf{Preproc}_{\{A_i\}_i}, \mathsf{Apply}_{\{A_i\}_i}, \mathsf{Apply}^{\alpha}_{\{A_i\}_i})$ such that if $\mathsf{ct} = \mathsf{Enc}_v(\mathsf{encode}(\mathbf{m}))$, and ek is a gadget-based rotation key associated with the RLWE secret v then*

$$\mathsf{Apply}_{\{A_i\}_i}(\mathsf{Preproc}_{\{A_i\}_i}(\alpha(\widehat{\mathsf{ek}}), \alpha(\widehat{\mathsf{ct}})), \widehat{\mathsf{ek}}, \widehat{\mathsf{ct}})$$

outputs a collection of ciphertexts $\{\mathsf{Enc}_v(\mathsf{encode}(A_i \cdot \mathbf{m}))\}_{i \in [M]}$.

Moreover, $\mathsf{Preproc}_{\{A_i\}_i}$ runs in $O(\ell n n_{\mathsf{cols}} \log n)$ $\mathbb{Z}_q$ operations, and produces an output of size $n n_{\mathsf{cols}}(\ell + 1)$ $\mathbb{Z}_q$ elements, and $\mathsf{Apply}_{\{A_i\}_i}$ runs in $n n_{\mathsf{cols}}(M + \ell + 2)$ $\mathbb{Z}_q$ operations.

Proof. It suffices to set $\mathsf{Preproc}_{\{A_i\}_i} = \mathsf{Preproc}_{\mathsf{rot}^{\circ n_{\mathsf{cols}}}}$, so we focus on the description of $\mathsf{Apply}_{\{A_i\}_i}$. This uses $\mathsf{Apply}_{\mathsf{rot}^{\circ n_{\mathsf{cols}}}}$ to compute ciphertexts $\mathsf{ct}_i = \mathsf{Enc}_v(\mathsf{encode}(\mathsf{rot}^{\circ i}(\mathbf{m})))$, then returns (for each $j \in [M]$) the values $\widehat{c}_j' = \sum_{i \in [n_{\mathsf{cols}}]} \mathrm{diag}_i(A_j) \circ \widehat{c}_i$. As a consequence of Lemma 5, one can check that $\widehat{c}_j'$ are of the form $\mathsf{Enc}_v(\mathsf{encode}(A_j \cdot \mathbf{m}))$, as desired.

As $\mathsf{Preproc}_{\{A_i\}_i} = \mathsf{Preproc}_{\mathsf{rot}^{\circ i}}$, it suffices to examine the complexity of $\mathsf{Apply}_{\{A_i\}_i}$. This calls $\mathsf{Apply}_{\mathsf{rot}^{\circ i}}$, and post-processes the result of this with $M n n_{\mathsf{cols}}$ operations in $\mathbb{Z}_q$, leading to the quoted complexity. □

We note that the complexity of our scheme ($n n_{\mathsf{cols}}(M + \ell + 2)$ $\mathbb{Z}_q$ operations) is quite close to the complexity of the underlying plaintext computation we are performing (naively, $n n_{\mathsf{cols}} M$ $\mathbb{Z}_p$ operations), e.g. concretely our scheme has low overhead, and should be performant. We validate this in Section 7.

4 NTTlessPIR: A LinPIR Scheme from RLWE

In this section we specify NTTlessPIR: a performant LinPIR scheme using RLWE-based (linearly homomorphic) encryption. After applying our optimizations of

Section 3, we find that it inherits the benefits of LWEPIR-type schemes (implementable using simple, coordinate-wise operations on modular integers) as well as RLWE-based encryption (compact ciphertexts).

We investigate this LinPIR scheme in isolation in this section, before showing in next section it may be used to remove the database-dependent hint from LWEPIR, by replacing the local computation $(\mathbf{s}, H := \mathsf{DB} \cdot A) \mapsto H \cdot \mathbf{s} \bmod Q$ in LWEPIR (that requires the hint H) with a LinPIR query $\mathbf{s}$ to H, viewed as a database. Throughout, we assume the database $\mathsf{DB} \in \mathbb{Z}_Q^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$, where[15] $n_{\mathsf{cols}} \leq n$ (and n_{rows} is arbitrary).

Handling Arbitrary Modulus The bulk of our scheme immediately follows from Theorem 1. We briefly describe the one step that doesn't, namely the extension of Theorem 1 (where the plaintext modulus is NTT-friendly) to arbitrary plaintext modulus.

Note that if the client can recover the LinPIR query $\mathsf{DB} \cdot \mathbf{m}$ over the integers, they can then manually reduce this $\bmod Q$, and compute the correct value. To have the client recover the LinPIR query over the integers, we have the client execute a LinPIR query $\bmod p_j$ for sufficiently many (coprime) NTT-friendly moduli p_j. The client may then CRT interpolate their results to recover $\mathsf{DB} \cdot \mathbf{m} \bmod \prod_j p_j$. Provided $\prod_j p_j$ is large enough such that no modular reduction occurs, we are done, e.g. we may compute $\mathsf{DB}{\cdot}\mathbf{m} \bmod Q$ for an arbitrary modulus Q by computing $\mathsf{DB}{\cdot}\mathbf{m} \bmod p_j$ for sufficiently many NTT-friendly moduli, which we do efficiently via Theorem 1.

4.1 NTTlessPIR Protocol Specification

As the security and correctness of $\mathsf{NTTlessPIR}$ essentially follows from the security of the underlying homomorphic encryption and standard correctness analysis, we focus on explicitly describing $\mathsf{NTTlessPIR}$ here and summarizing its efficiency. and defer formally establishing correctness and security for Appendix E.

Lemma 6. *Let* $\mathsf{DB} \in \mathbb{Z}_Q^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$ *where* $n_{\mathsf{cols}} \leq n$. *Let* $\mathbf{m}$ *satisfy* $\|\mathbf{m}\|_\infty \leq B$. *Then the* LinPIR *scheme described in Figure 3 requires*

- *Server Preprocessing:* $O(k\ell n n_{\mathsf{cols}} \log n)$ *operations in* $\mathbb{Z}_q$,
- *Server Long-term Storage:* $knn_{\mathsf{cols}}(\ell + 1)$ *elements of* $\mathbb{Z}_q$,
- *Server Response Time:* $kn_{\mathsf{cols}}(n_{\mathsf{rows}} + n + (\ell + 2)n)$ $\mathbb{Z}_q$ *operations,*
- *Client Upload:* $kn + \ell n$ *elements of* $\mathbb{Z}_q$,
- *Client Download:* $2k\lceil n_{\mathsf{rows}}/n \rceil n$ *elements of* $\mathbb{Z}_q$.

Proof. Server preprocessing and server long-term storage amounts to running the preprocessing algorithm of Theorem 1 k times, as well as sampling a λ-bit seed (which we ignore, as its cost is dwarfed by the cost of other preprocessing). The cost to compute the server response is similarly k-times the online cost of Theorem 1.

[15] We reassure the reader that this restriction on n_{cols} will be unimportant to our main application of this scheme, namely removing the hint from LWEPIR in Section 5.

```
Server Algorithms in NTTlessPIR
$\mathsf{S.setup}(\{\mathsf{DB}_i\}_{i\in[M]})$
  $\mathsf{seed} \leftarrow \{0,1\}^\lambda$
  for $i \in [\ell]$
    $\widehat{a}'_i \leftarrow \mathsf{RO}(\mathsf{seed}||0||i)$
  for $j \in [k]$
    $\widehat{a}_j \leftarrow \mathsf{RO}(\mathsf{seed}||1||j)$
    $g_j = \mathsf{Preproc}_{\{\mathsf{DB}_i \bmod p_j\}_{i\in[M]}}(\widehat{\mathbf{a}}', \widehat{a}_j)$
  return $(\mathsf{seed}, \{g_j\}_{j\in[k]})$
$\mathsf{S.response}(\{\widehat{b}_j\}_{j\in[k]}, \{\widehat{b}'_i\}_{i\in[\ell]}, \{g_j\}_{j\in[k]})$
  for $j \in [k]$
    $\widehat{\mathsf{ct}}_j = \mathsf{Apply}_{\{\mathsf{DB}_i \bmod p_j\}_{i\in[M]}}(g_j, \widehat{\mathbf{b}}', \widehat{b}_j)$
  return $\{\widehat{\mathsf{ct}}_j\}_{j\in[k]}$
```

```
Client Algorithms in NTTlessPIR.
$\mathsf{C.query}(\mathbf{m}, \mathsf{seed})$
  $v \leftarrow \chi_\sigma^n$
  for $i \in [\ell]$
    $\widehat{b}'_i = \mathsf{Enc}_v(\mathbf{g}_i\mathsf{rot}(v); \mathsf{seed}||0||i)$
  for $j \in [k]$
    $\widehat{b}_j = \mathsf{Enc}_v(\mathsf{encode}_{p_j}(\mathbf{m}); \mathsf{seed}||1||j)$
  return $(v, \{\widehat{b}_j\}_{j\in[k]}, \{\widehat{b}'_i\}_{i\in[\ell]})$
$\mathsf{C.recover}(\{\widehat{\mathsf{ct}}_j\}_{j\in[k]}, v)$
  for $j \in [k]$
    for $i \in [M]$
      $\mathbf{m}_{i,j} \leftarrow \mathsf{decode}_{p_j}(\mathsf{Dec}_v((\widehat{\mathsf{ct}}_j)_i))$
  for $i \in [M]$
    $\mathbf{m}_{i,P} = \mathsf{iCRT}_{\mathbf{p}}(\mathbf{m}_{i,0}, \ldots, \mathbf{m}_{i,k-1})$
    $\mathbf{m}_{i,Q} = \mathbf{m}_{i,P} \bmod Q$
  return $(\mathbf{m}_{0,Q}, \ldots, \mathbf{m}_{M-1,Q})$
```

Fig. 3. The Algorithms of NTTlessPIR. Each ciphertext $\widehat{\mathsf{ct}}_j$ in the S.response and C.recover is an M-tuple of ciphertexts encrypting $\mathsf{DB}_i \cdot \mathbf{m} \bmod p_j$ for $i \in [M], j \in [k]$.

5 Removing the Hint from LWEPIR

We next show how to leverage NTTlessPIR to remove the hint form LWEPIR with low overhead. The scheme itself is straightforward extension of LWEPIR, where the client replaces the local computation of $(H := \mathsf{DB} \cdot A, \mathbf{s}) \mapsto H \cdot \mathbf{s}$ with a LinPIR query $\mathbf{s}$ to the database H. We call the resulting scheme HintlessPIR.

Similarly to NTTlessPIR, we solely describe the protocol itself (and summarize its efficiency) in this section, and defer the standard analysis of correctness and security to Appendix F. As discussed in that section (Lemma 11), the server must periodically reseed every κ queries for security. This comes at no asymptotic running-time cost provided $\kappa = \omega(\log n)$.

Lemma 7. *Let* $\mathsf{DB} \in \mathbb{Z}_p^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$ *Then* HintlessPIR *requires*

- *Server Preprocessing:* $O(k\ell nN \log n)$ *operations in* $\mathbb{Z}_q$ *and* $2mN$ *operations in* $\mathbb{Z}_Q$*,*
- *Server Long-term Storage:* $knN(\ell + 1)$ *elements of* $\mathbb{Z}_q$ *and* $\sqrt{m}N$ *elements of* $\mathbb{Z}_Q$*,*
- *Server Response Time:* $kN(\sqrt{m} + n + (\ell + 2)n)$ $\mathbb{Z}_q$ *operations, and* m $\mathbb{Z}_Q$ *operations,*
- *Client Upload:* $(k + \ell)n$ *elements of* $\mathbb{Z}_q$ *and* $\sqrt{m}$ *elements of* $\mathbb{Z}_Q$*,*
- *Client Download:* $2k(\sqrt{m} + n)$ *elements of* $\mathbb{Z}_q$ *and* $\sqrt{m}$ *elements of* $\mathbb{Z}_Q$

Proof. HintlessPIR reduces to

- LWEPIR, invoked on a database $\mathsf{DB} \in \mathbb{Z}_p^{\sqrt{m} \times \sqrt{m}}$, and
- NTTlessPIR, invoked on a database $H := \mathsf{DB} \cdot A \in \mathbb{Z}_Q^{\sqrt{m} \times N}$. Note that $N := n_{\mathsf{cols}} \leq n$ for security, so the condition required for NTTlessPIR is satisfied.

Server Algorithms in HintlessPIR

```
$\mathsf{S.setup}(\mathsf{DB})$ :
    $(\Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}}, \Pi_{\mathsf{LWE}}.\mathsf{S}_{\mathsf{hint}}) \leftarrow \Pi_{\mathsf{LWE}}.\mathsf{setup}(\mathsf{DB})$
    $(H, \mathsf{seed}) \leftarrow \Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}}$
    $\Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}} = (0, \mathsf{seed})$
    $(\Pi_{\mathsf{NTT}}.\mathsf{C}_{\mathsf{hint}}, \Pi_{\mathsf{NTT}}.\mathsf{S}_{\mathsf{hint}}) \leftarrow \Pi_{\mathsf{NTT}}.\mathsf{setup}(H)$
    return $((\Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}}, \Pi_{\mathsf{NTT}}.\mathsf{C}_{\mathsf{hint}}),$
            $(\Pi_{\mathsf{LWE}}.\mathsf{S}_{\mathsf{hint}}, \Pi_{\mathsf{NTT}}.\mathsf{S}_{\mathsf{hint}}))$
$\mathsf{S.response}(\mathsf{query}_{\mathsf{LWE}}, \mathsf{query}_{\mathsf{NTT}})$
    return $(\Pi_{\mathsf{LWE}}.\mathsf{response}(\mathsf{query}_{\mathsf{NTT}}),$
            $\Pi_{\mathsf{NTT}}.\mathsf{response}(\mathsf{query}_{\mathsf{NTT}}))$
```

Client Algorithms in HintlessPIR

```
$\mathsf{C.query}(i, (\Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}}, \Pi_{\mathsf{NTT}}.\mathsf{C}_{\mathsf{hint}}))$ :
    $(c_0, i_1), \mathsf{qu}_{\mathsf{LWE}} = \Pi_{\mathsf{LWE}}.\mathsf{query}(i, \Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}})$
    $v, \mathsf{qu}_{\mathsf{NTT}} = \Pi_{\mathsf{NTT}}.\mathsf{query}(\mathbf{s}, \Pi_{\mathsf{NTT}}.\mathsf{C}_{\mathsf{hint}})$
    return $((i_1, v), (\mathsf{qu}_{\mathsf{LWE}}, \mathsf{qu}_{\mathsf{NTT}}))$
$\mathsf{C.recover}((i_1, v), (\mathsf{rsp}_{\mathsf{LWE}}, \mathsf{rsp}_{\mathsf{NTT}}))$ :
    $c_0 = \Pi_{\mathsf{NTT}}.\mathsf{recover}(v, \mathsf{rsp}_{\mathsf{NTT}})$
    return $\Pi_{\mathsf{LWE}}((c_0, i_1), \mathsf{rsp}_{\mathsf{LWE}})$
```

Fig. 4. The algorithms in HintlessPIR. Throughout, we write $\Pi_{\mathsf{NTT}} = \mathsf{NTTlessPIR}$ and $\Pi_{\mathsf{LWE}} = \mathsf{LWEPIR}$ for brevity, e.g. $\Pi_{\mathsf{NTT}}.\mathsf{query}$ is Client's query algorithm from NTTlessPIR. In the algorithms, $\mathbf{s}$ is the LWE encryption key sampled during the LWEPIR query algorithm. Note that the $\Pi_{\mathsf{LWE}}.\mathsf{C}_{\mathsf{hint}}$ hint contains an LWEPIR hint $H = 0$ that is incorrect, and therefore the value $c_0 = H \cdot \mathbf{s}$ is incorrect as well. We use NTTlessPIR to retrieve the correct value of c_0 via a LinPIR query to the database H.

6 TensorPIR: Recursing a Single Time

We now describe our second PIR scheme, which we call TensorPIR. Assume the database is a three-dimensional object $\mathsf{DB} \in \mathbb{Z}_Q^{d_{\mathbf{u}}} \times \mathbb{Z}_Q^{d_{\mathbf{v}}} \times \mathbb{Z}_Q^{d_{\mathbf{w}}}$, for $m = d_{\mathbf{u}} d_{\mathbf{v}} d_{\mathbf{w}}$. Our goal is to retrieve the row in the $d_{\mathbf{w}}$ dimension using two $O(\sqrt[3]{m})$ size selection vectors $\mathbf{u}$ and $\mathbf{v}$ in the $d_{\mathbf{u}}$ and $d_{\mathbf{v}}$ dimensions. Let $C_0 = [A_0, \mathbf{b}_0]$ and $C_1 = [A_1, \mathbf{b}_1]$ be LWE encryptions of $\mathbf{u} \in \mathbb{Z}^{d_{\mathbf{u}}}$ and $\mathbf{v} \in \mathbb{Z}^{d_{\mathbf{v}}}$, respectively, under the client's LWE secret $\mathbf{s}$. These LWE ciphertexts satisfy the raw decryption relations $\mathbf{u} \approx \mathbf{b}_0 - A_0 \cdot \mathbf{s} \bmod Q$ and $\mathbf{v} \approx \mathbf{b}_1 - A_1 \cdot \mathbf{s} \bmod Q$. Thus $\mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v})$ can be approximately computed by

$$\begin{aligned}\mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v}) \approx \mathsf{DB} \cdot (\mathbf{b}_0 \otimes \mathbf{b}_1) - \mathsf{DB} \cdot (A_0 \cdot \mathbf{s} \otimes \mathbf{b}_1) - \mathsf{DB} \cdot (\mathbf{b}_0 \otimes A_1 \cdot \mathbf{s}) \\ + \mathsf{DB} \cdot (A_0 \cdot \mathbf{s} \otimes A_1 \cdot \mathbf{s}) \bmod Q. \qquad (6)\end{aligned}$$

The right hand side is a noisy version of $\mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v})$. So if the client can obtain these terms, then it can round and remove the error to get the desired records.

In TensorPIR, the client encrypts $\mathbf{u}$ and $\mathbf{v}$ under its LWE secret $\mathbf{s}$ as above, and it additionally uses a RLWE-based scheme Enc for the terms involving the LWE secret vector $\mathbf{s}$. Specifically, the client samples a fresh RLWE secret key v and sends the following ciphertexts to the server:

- $\mathsf{ct}_{A_0\mathbf{s}} \leftarrow \mathsf{Enc}_v(\sum_{i\in[d_\mathbf{u}]} \langle \mathbf{a}_i, \mathbf{s}\rangle \cdot X^i)$, and
- $\mathsf{ct}_\mathbf{s} \leftarrow \mathsf{Enc}_v(\mathsf{encode}(\mathbf{s}))$,

where $\mathbf{a}_i = \mathbf{u_i}^t \cdot A_0$ is the ith row of A_0. The client also includes a Galois key for $\theta : X \mapsto X^5$ in its query. We adopt the CRT decomposition technique used in Section 4 to handle homomorphic computation over arbitrary modulus Q that does not match the plaintext modulus of RLWE encryptions. Since the homomorphic computation over each plaintext modulus is exactly the same, we describe TensorPIR without explicitly mentioning the plaintext modulus.

Note that we can write the database as $\mathsf{DB} = [\mathsf{DB}_1, \ldots, \mathsf{DB}_{d_\mathbf{u}}] = \sum_{i\in[d_\mathbf{u}]} \mathbf{u_i}^t \otimes \mathsf{DB}_i$. Given the ciphertexts in a client query, the server can then homomorphically compute, for all $i \in [d_\mathbf{u}]$,

- the RLWE encryptions $\mathsf{Enc}_v(\langle \mathbf{a}_i, \mathbf{s}\rangle)$ of scalars $\langle \mathbf{a}_i, \mathbf{s}\rangle$, and
- the RLWE encryptions $\mathsf{Enc}_v(\mathsf{encode}(\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s}))$.

The first set of ciphertexts can be efficiently generated from $\mathsf{ct}_{A_0\mathbf{s}}$ via RLWE expansion [15] (see Lemma 15). The second set of ciphertexts are exactly the homomorphic matrix-vector products between $\mathsf{DB}_i \cdot A_1$ and $\mathsf{Enc}(\mathsf{encode}(\mathbf{s}))$, and we invoke NTTlessPIR to compute them. We note that the above homomorphic computations are all compatible with our preprocessing optimization of Section 3. Furthermore, we let the server generate all $\log n$ Galois keys required by the RLWE expansion algorithm (via Lemma 14), which is also compatible with the NTT preprocessing optimization.

Afterwards, the server can just encode the plaintext terms $\mathsf{DB}_i \cdot A_1$, $\mathbf{u_i}^t \mathbf{b}_0$, and $\mathsf{DB}_i \cdot \mathbf{b}_1$ accordingly for all $i \in [d_\mathbf{u}]$, and then homomorphically compute the sum of products in Eq. (6). We refer to Appendix G for detailed protocol specification and analysis.

7 Implementation and Evaluation

7.1 HintlessPIR Implementation

We implemented[16] NTTlessPIR with preprocessing optimization and applied it to SimplePIR. For SimplePIR, the main constraint is to choose the modulus $Q \in \{2^{32}, 2^{64}\}$ such that one can directly compute arithmetic in hardware; so we set secret key dimension[17] $N = 1408$, ciphertext modulus $Q = 2^{32}$, error standard

[16] https://github.com/google/hintless_pir

[17] We set N higher than the one proposed in [34] according to latest lattice attack estimates.

deviation $\sigma = 6.4$, and sample the LWE secret key from the uniform ternary distribution. We set our RLWE parameters as $n = 2^{12}$, ciphertext modulus $q \approx 2^{90}$, and error standard deviation 3.2 for both the RLWE secret key and the error terms. Both LWE and RLWE parameters are at the 128-bit security level with up to 2^{30} samples [2]. The ℓ_∞ norm of the LWE decryption vector $H \cdot \mathbf{s}$ can be bounded by 2^{42}, where $H = \mathsf{DB} \cdot A \bmod Q$ is the hint matrix. So, we choose two NTT-friendly plaintext moduli p_0, p_1 of 22 bits each for CRT decomposing H and $\mathbf{s}$. Note that our RLWE parameters can also handle the alternative LWE parameters suggested in [33] with secret dimension $N = 2048$, which may have different efficiency tradeoffs for very large database dimensions. Due to space constraint we report only on the smaller LWE parameters.

Our NTTlessPIR scheme implementation is based on the Residue Number System (RNS) variant of Brakerski/Fan-Vercauteren (BFV) FHE scheme that supports linear homomorphic operations. In particular, we implemented all the preprocessing optimizations of Section 3. Our RLWE parameters provide 4096 slots in each plaintext polynomial, so we pack two copies of $\mathbf{s}$ in the query ciphertexts. Correspondingly, we pad H to $H' = [H \mid \mathbf{0}]$ of 2048 columns, and we pack two diagonals of each 1024×2048 block of H in each plaintext polynomial. For each p_j, this packing strategy reduces the number of rotations to 511, and reduces the number of ciphertext-plaintext multiplications by half.

7.2 HintlessPIR Evaluation

We benchmarked several typical database dimensions, with sizes up to 8.59GB: 1) one million small records (8 bytes and 256 bytes each) as common baselines for PIR [3, 40]; 2) up to 1 billion small records; and 3) smaller number of moderate-size records (32KB each and 8.59GB total). We ran our server program on an AWS `r7iz.4xlarge` instance with Intel Sapphire Rapids CPUs running at 3.00GHz and with 128GB RAM. We took advantage of the SIMD instruction sets such as AVX-512, and compiled our test program using `clang` 16 and executed it using a single thread. We also benchmarked the public implementations of SimplePIR and DoublePIR [49], Spiral [10], as well as Tiptoe PIR [51], using the same testing environment.

The LWE plaintext space is about 8 to 10 bits for databases up to 2^{38} records. For large database records, we follow the suggestion in [34] to encode each record using $d > 1$ LWE plaintext elements and vertically stack them in a column of the database matrix DB. In [34], this minimizes the hint size; and for NTTlessPIR, this minimizes both the response size and the server online time.

For communication cost, a NTTlessPIR query is 323KB (which includes two compressed ciphertexts and a compressed rotation key), and its response size scales roughly with $\sqrt{dN}$, which is asymptotically the same as in SimplePIR. Comparing with SimplePIR, a pair of NTTlessPIR query and response is only about 1% of SimplePIR hint for all databases we measured except the first (very small) database of dimension $2^{20} \times 8$ bytes. Comparing with [33], a NTTlessPIR query is roughly 1.5% of the RLWE part of a Tiptoe PIR query.

Database (total size)		$2^{20} \times 8$B	$2^{20} \times 256$B	$2^{26} \times 8$B	$2^{30} \times 1$B	$2^{18} \times 32$KB
		(8MB)	(268MB)	(537MB)	(1.07GB)	(8.59GB)
HintlessPIR	Query Size	334KB	388KB	415KB	453KB	1502KB
	Response Size	288KB	1540KB	2212KB	3080KB	3080KB
Tiptoe PIR	Query Size	23MB	23MB	23MB	23MB	24MB
	Response Size	333KB	1336KB	2002KB	2671KB	2925KB
SimplePIR	Hint Size	16MB	92MB	131MB	185MB	185MB
	Query Size	12KB	66KB	93KB	131KB	1180KB
	Response Size	12KB	66KB	93KB	131KB	117KB
DoublePIR	Hint Size	242MB	6897MB	242MB	31MB	874GB
	Query Size	352KB	352KB	354KB	398KB	352KB
	Response Size	352KB	10035KB	352KB	44KB	1273MB
Spiral	Parameter Size	8MB	8MB	9MB	9MB	10MB
	Query Size	16KB	16KB	16KB	16KB	16KB
	Response Size	21KB	21KB	21KB	21KB	61KB

Table 1. Communication costs of HintlessPIR, Tiptoe PIR, SimplePIR, DoublePIR, and Spiral, for several typical database dimensions. For HintlessPIR, a query includes two RLWE ciphertexts and a rotation key as well as a LWEPIR query vector, and a response includes RLWE ciphertexts encrypting $\mathsf{DB} \cdot A \cdot \mathbf{s}$ and the LWEPIR response vector. For Tiptoe PIR, a query contains N RLWE ciphertexts as well as a LWEPIR query vector. For Spiral, the parameter includes key materials for expanding RLWE encryptions into RGSW ciphertexts.

In terms of computation overhead, it takes 102ms to homomorphically generate all rotations of $\mathbf{s} \pmod{p_j}$ for each p_j, which is a one-time cost independent of the database dimension. The homomorphic matrix-vector multiplication proceeds over each $1024 \times N$ block of H. The time for the client to recover the PIR answer remains inexpensive in our benchmarks, taking no more than 50ms for databases up to 1GB and 145ms for 8GB databases. Comparing with Tiptoe PIR, we see that HintlessPIR runs faster for all databases we benchmarked, despite that the Tiptoe PIR implementation uses a smaller RLWE parameter. This is probably because half of the ciphertext polynomials in the NTTlessPIR response are precomputed offline and a faster RLWE implementation in NTTlessPIR. We summarize the communication and computation overhead of HintlessPIR in Tables 1 and 2.

Bandwidth and latency of making a few queries. One of the advantages of HintlessPIR is the absence of offline interaction between the client and the server, which makes it very appealing for situations where the client only makes a few queries before the database is updated. We show in Fig. 5 the bandwidth and latency of a new client making its initial query to databases using HintlessPIR, SimplePIR, and Spiral. To measure latency we model the connection using median upload and download speeds for mobile devices in the US, which are 85.32Mbps and 8.34Mbps respectively as of August 2023[18]. For all the database dimensions

[18] `https://www.speedtest.net/global-index/united-states`

Database (total size)		$2^{20} \times$ 8B (8MB)	$2^{20} \times$ 256B (268MB)	$2^{26} \times$ 8B (537MB)	$2^{30} \times$ 1B (1.07GB)	$2^{18} \times$ 32KB (8.59GB)
HintlessPIR	Server Online Time	233ms	385ms	478ms	613ms	1347ms
	Server Throughput	35MB/s	698MB/s	1122MB/s	1750MB/s	6376MB/s
	Server Preproc. Time	3.11s	51.57s	93.58s	199.15s	2128s
	Client Recovery time	4.78ms	25.50ms	36.66ms	51.00ms	52.32ms
Tiptoe PIR	Server Online Time	295ms	963ms	1393ms	1890ms	2675ms
	Server Throughput	28MB/s	278MB/s	385MB/s	568MB/s	3211MB/s
	Server Preproc. Time	0.96s	46.39s	85.24s	188.03s	2116s
	Client Recovery time	1.69ms	7.69ms	10.81ms	14.67ms	15.28ms
SimplePIR	Server Online Time	0.75ms	28ms	53ms	105ms	841ms
	Server Throughput	11201MB/s	9675MB/s	10046MB/s	10268MB/s	10213MB/s
	Server Preproc. Time	0.96s	45.39s	85.24s	188.03s	2116s
DoublePIR	Server Online Time	1.19ms	29ms	58ms	113ms	1120ms
	Server Throughput	7046MB/s	9340MB/s	9288MB/s	9506MB/s	7672MB/s
	Server Preproc. Time	4.03s	110.57s	241.46s	485.53s	-
Spiral	Server Online Time	691ms	694ms	1229ms	2319ms	12769ms
	Server Throughput	12MB/s	387MB/s	436MB/s	463MB/s	673MB/s
	Client Parameter Generation Time	139ms	193ms	326ms	326ms	326ms

Table 2. Computational costs of HintlessPIR, Tiptoe PIR, SimplePIR, DoublePIR, and Spiral, on several typical database dimensions. The server throughput is computed as database size / server online time. For SimplePIR, the server preprocessing time includes generating the LWEPIR hint matrix. For HintlessPIR, the server preprocessing time additionally includes the preprocessing time for NTTlessPIR, and the client recovery time includes decrypting the server response and deriving the PIR answer. For DoublePIR, the server preprocessing time includes generating its two hint matrices, which ran out of memory for the largest database. For Tiptoe PIR, server preprocessing is identical to SimplePIR, and we reused the measured numbers to highlight this. For Spiral, we also measured the time for the client to generate the offline parameters (i.e. key materials).

we consider, HintlessPIR has lower cost when making the first query than the other three protocols. Comparing with Spiral, HintlessPIR maintains this bandwidth advantage for the first 3 to 5 queries, and always has smaller latency. Comparing with SimplePIR, HintlessPIR has lower latency for making up to 10 queries except for the smallest database while the communication cost is always lower for up to 75 queries. We note that, in terms of latency, the trivial PIR protocol is better than SimplePIR and Spiral for the smallest database while worse than HintlessPIR.

Parallelization. In previous protocols based on RLWE homomorphic encryption, it is sometimes hard to fully take advantage of parallelism (e.g. speedup a factor k using k more processors) due to memory I/O bottlenecks from converting between evaluation and coefficient forms. With our NTT precomputation optimization, the server's online algorithm consists of point-wise additions

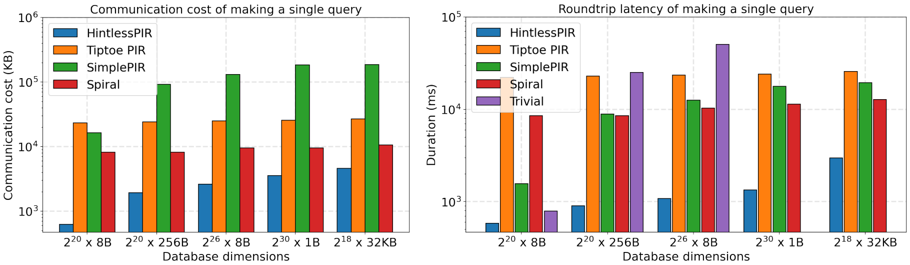

Fig. 5. The communication cost and roundtrip latency of using HintlessPIR, Tiptoe PIR, SimplePIR, and Spiral to make the initial query on several databases. We assume the download speed is 85.32Mbps and the upload speed is 8.34Mbps, which are the median mobile device speeds in the US in August 2023. For SimplePIR, we account for the cost of downloading the hint from the server and making a single query. For Spiral, we account for the cost of uploading the parameters to the server and making a single query. For the first three databases, we also include the latency of the trivial solution that downloads the entire database. All graphs are in the log scale.

and multiplications of vectors, which are much less memory-intensive than the previously-described protocols, and have simple (and predictable) memory access patterns. So, we tuned our implementation to use all four CPU cores: the first step of NTTlessPIR's online algorithm is distributed to two threads, one per plaintext modulus, and the second step to four threads. In addition, we also parallelized our SimplePIR implementation in four threads. See Table 3 for latency and throughput with multi-threading. Our benchmark results show that we were able to take advantage of all CPU cores available in the test environment, especially for large databases. One can expect to further parallelize our protocol on more powerful CPUs. In particular, it seems easier to accelerate our protocol using GPUs with powerful SIMD capacity.

Database (total size)		$2^{20} \times$ 8B (8MB)	$2^{20} \times$ 256B (268MB)	$2^{26} \times$ 8B (537MB)	$2^{30} \times$ 1B (1.07GB)	$2^{18} \times$ 32KB (8.59GB)
HintlessPIR	Online Time (Four Thread)	115ms	162ms	193ms	232ms	418ms
	Throughput	0.07GB/s	1.66GB/s	2.78GB/s	4.62GB/s	20.53GB/s
SimplePIR	Online Time (Four Threads)	0.30ms	7ms	14ms	27ms	213ms
	Throughput	28.15GB/s	38.78GB/s	39.22GB/s	40.36GB/s	40.42GB/s

Table 3. Multi-threading computational costs of HintlessPIR and SimplePIR, on several typical database dimensions. The server throughput is computed as database size / server online time.

7.3 TensorPIR

While asymptotically TensorPIR provides better communication than HintlessPIR and better computation than Spiral, finding concrete parameters where this asymptotic behavior becomes apparent is much more challenging. Next we try to give intuition why this is and provide estimates for the settings when the concrete efficiency of TensorPIR becomes competitive.

For the databases in Tables 1 and 2, TensorPIR does not have advantage over HintlessPIR for either computation and communication. This is because we need to use larger RLWE parameters and more CRT plaintext moduli. In more detail, according to our analysis in Appendix G, if we want to use the same LWE parameters as the ones for HintlessPIR in Section 7.1, TensorPIR requires an RLWE modulus for supporting depth-2 homomorphic computation over 20-bit plaintext space. To reduce communication cost, we let the client send two Galois keys for the server to generate all $\log d_{\mathbf{u}}$ many Galois keys needed for ciphertext expansion, where the error size of the composed Galois keys is at least $2^{15}\ell B$ (for composing ℓ times with gadget quality B). As a result, we estimate the RLWE modulus size is at least 150 bits when using a gadget of quality 8 bits, and so the ring degree is at least 2^{13}.

To find a parameter regime where TensorPIR is concretely competitive, we need to consider much bigger databases. We estimate that for databases of size 2^{40} with 1 byte records that TensorPIR has server running time close to HintlessPIR and with smaller communication cost. We performed experiments on the core components of TensorPIR with the help of the preprocessing optimization, where we set the ring degree to $n = 2^{13}$, ciphertext modulus $q \approx 2^{144}$, and sub-dimensions $d_{\mathbf{u}} = 2^7, d_{\mathbf{v}} = 2^{21}, d_{\mathbf{w}} = 2^{12}$. Based on these micro benchmarks, we expect TensorPIR to have server throughput roughly 8.3GB/s and total communication 37.6MB for each query. On the same database, we expect HintlessPIR to have server throughput about 9.1GB/s and total communication 103MB, Tiptoe PIR to have throughput 6.57GB/s and total communication 115MB, and Spiral's server throughput will be roughly 667MB/s with total online communication 17MB [19].

8 Conclusion

We presented two new PIR schemes with neither client-dependent preprocessed state on the server nor database-dependent preprocessed state on the client. With the composable preprocessing optimization, we were able to achieve concretely fast server processing time in our first construction, HintlessPIR, namely up to 60% of the throughput of Simple PIR, and up to 9.4× higher throughput than Spiral PIR. Our communication cost is consistently small, improving on total

[19] For Spiral, we could not find parameters that natively support this database on the public Spiral implementation; so we shard the database into smaller ones and estimated Spiral's performance when running sequentially on the sub-databases.

communication costs (compared to Simple PIR and Spiral PIR) in settings where many clients are making few queries each.

In terms of preprocessing in homomorphic encryption, so far we have been able to apply it to basic homomorphic operations and gadget-based key switching. It seems very interesting to extend such technique to additional homomorphic operations and constructions. For example, it seems nontrivial to apply this technique to the GHS [25] variant of key-switching. In addition, it may be useful to apply composable preprocessing to other protocols to improve both online computation and communication.

Bibliography

[1] Ahmad, I., Yang, Y., Agrawal, D., Abbadi, A.E., Gupta, T.: Addra: Metadata-private voice communication over fully untrusted infrastructure. In: Brown, A.D., Lorch, J.R. (eds.) 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021 (2021)

[2] Albrecht, M.R., Player, R., Scott, S.: On the concrete hardness of learning with errors. J. Math. Cryptol. **9**(3), 169–203 (2015), `http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml`

[3] Ali, A., Lepoint, T., Patel, S., Raykova, M., Schoppmann, P., Seth, K., Yeo, K.: Communication-computation trade-offs in PIR. In: Bailey, M., Greenstadt, R. (eds.) USENIX Security 2021: 30th USENIX Security Symposium. pp. 1811–1828. USENIX Association (Aug 11–13, 2021)

[4] Angel, S., Chen, H., Laine, K., Setty, S.: Pir with compressed queries and amortized query processing. In: 2018 IEEE Symposium on Security and Privacy (SP) (2018)

[5] Angel, S., Setty, S.: Unobservable communication over fully untrusted infrastructure. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16) (2016)

[6] Apple: icloud private relay overview (2021), `https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf`, `https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf`

[7] Artioli, S.: How practical is single-server private information retrieval? (2023), `https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/How_practical_is_single_server_private_information_retrieval_corrected.pdf`

[8] Backes, M., Kate, A., Maffei, M., Pecina, K.: Obliviad: Provably secure and practical online behavioral advertising. In: 2012 IEEE Symposium on Security and Privacy (2012)

[9] Beimel, A., Ishai, Y., Malkin, T.: Reducing the servers computation in private information retrieval: PIR with preprocessing. In: Bellare, M. (ed.) Advances in Cryptology – CRYPTO 2000. Lecture Notes in Computer Science, vol. 1880, pp. 55–73. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 20–24, 2000). `https://doi.org/10.1007/3-540-44598-6_4`

[10] blyss SDK for accessing data privately using homomorphic encryption. `https://github.com/blyssprivacy/sdk` (2023)

[11] Borisov, N., Danezis, G., Goldberg, I.: DP5: A private presence service. Proc. Priv. Enhancing Technol. (2015)

[12] Boyle, E., Gilboa, N., Ishai, Y.: Function secret sharing: Improvements and extensions. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16 (2016)

[13] Brakerski, Z., Döttling, N., Garg, S., Malavolta, G.: Leveraging linear decryption: Rate-1 fully-homomorphic encryption and time-lock puzzles. In: Hofheinz, D., Rosen, A. (eds.) TCC 2019: 17th Theory of Cryptography Conference, Part II. Lecture Notes in Computer Science, vol. 11892, pp. 407–437. Springer, Heidelberg, Germany, Nuremberg, Germany (Dec 1–5, 2019). https://doi.org/10.1007/978-3-030-36033-7_16

[14] Cachin, C., Micali, S., Stadler, M.: Computationally private information retrieval with polylogarithmic communication. In: Stern, J. (ed.) Advances in Cryptology – EUROCRYPT'99. Lecture Notes in Computer Science, vol. 1592, pp. 402–414. Springer, Heidelberg, Germany, Prague, Czech Republic (May 2–6, 1999). https://doi.org/10.1007/3-540-48910-X_28

[15] Chen, H., Chillotti, I., Ren, L.: Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019: 26th Conference on Computer and Communications Security. pp. 345–360. ACM Press, London, UK (Nov 11–15, 2019). https://doi.org/10.1145/3319535.3354226

[16] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: Private information retrieval. In: 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995 (1995)

[17] Corrigan-Gibbs, H., Henzinger, A., Kogan, D.: Single-server private information retrieval with sublinear amortized time. In: Dunkelman, O., Dziembowski, S. (eds.) Advances in Cryptology – EUROCRYPT 2022, Part II. Lecture Notes in Computer Science, vol. 13276, pp. 3–33. Springer, Heidelberg, Germany, Trondheim, Norway (May 30 – Jun 3, 2022). https://doi.org/10.1007/978-3-031-07085-3_1

[18] Corrigan-Gibbs, H., Kogan, D.: Private information retrieval with sublinear online time. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology – EUROCRYPT 2020, Part I. Lecture Notes in Computer Science, vol. 12105, pp. 44–75. Springer, Heidelberg, Germany, Zagreb, Croatia (May 10–14, 2020). https://doi.org/10.1007/978-3-030-45721-1_3

[19] Davidson, A., Pestana, G., Celi, S.: FrodoPIR: Simple, scalable, single-server private information retrieval. Proceedings on Privacy Enhancing Technologies **2023**(1), 365–383 (Jan 2023). https://doi.org/10.56553/popets-2023-0022

[20] Davidson, A., Pestana, G., Celi, S.: FrodoPIR: Simple, scalable, single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/981 (2022), https://eprint.iacr.org/2022/981, https://eprint.iacr.org/2022/981

[21] Devet, C., Goldberg, I., Heninger, N.: Optimally robust private information retrieval. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012. USENIX Association (2012)

[22] Dvir, Z., Gopi, S.: 2-server PIR with subpolynomial communication. Journal of the ACM (JACM) **63**(4), 1–15 (2016)

[23] Genise, N., Micciancio, D., Polyakov, Y.: Building an efficient lattice gadget toolkit: Subgaussian sampling and more. In: Ishai, Y., Rijmen, V.

(eds.) Advances in Cryptology – EUROCRYPT 2019, Part II. Lecture Notes in Computer Science, vol. 11477, pp. 655–684. Springer, Heidelberg, Germany, Darmstadt, Germany (May 19–23, 2019). https://doi.org/10.1007/978-3-030-17656-3_23

[24] Gentry, C., Halevi, S.: Compressible FHE with applications to PIR. In: Theory of Cryptography: 17th International Conference, TCC 2019 (2019)

[25] Gentry, C., Halevi, S., Smart, N.P.: Homomorphic evaluation of the AES circuit. In: Safavi-Naini, R., Canetti, R. (eds.) Advances in Cryptology – CRYPTO 2012. Lecture Notes in Computer Science, vol. 7417, pp. 850–867. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 19–23, 2012). https://doi.org/10.1007/978-3-642-32009-5_49

[26] Gentry, C., Ramzan, Z.: Single-database private information retrieval with constant communication rate. In: Proceedings of the 32nd International Conference on Automata, Languages and Programming. p. 803–815. ICALP'05 (2005)

[27] Gentry, C., Sahai, A., Waters, B.: Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013, Part I. Lecture Notes in Computer Science, vol. 8042, pp. 75–92. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 18–22, 2013). https://doi.org/10.1007/978-3-642-40041-4_5

[28] Google: VPN by Google One, https://one.google.com/about/vpn, https://one.google.com/about/vpn

[29] Google: Privacy sandbox IP protection proposal (2023), https://developer.chrome.com/en/docs/privacy-sandbox/ip-protection/, https://developer.chrome.com/en/docs/privacy-sandbox/ip-protection/

[30] Green, M., Ladd, W., Miers, I.: A protocol for privately reporting ad impressions at scale. CCS '16 (2016)

[31] Halevi, S., Shoup, V.: Algorithms in HElib. In: Garay, J.A., Gennaro, R. (eds.) Advances in Cryptology – CRYPTO 2014, Part I. Lecture Notes in Computer Science, vol. 8616, pp. 554–571. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2014). https://doi.org/10.1007/978-3-662-44371-2_31

[32] Henry, R.: Polynomial batch codes for efficient IT-PIR. Proc. Priv. Enhancing Technol. **2016**(4), 202–218 (2016)

[33] Henzinger, A., Dauterman, E., Corrigan-Gibbs, H., Zeldovich, N.: Private web search with tiptoe. In: Proceedings of the The 29th ACM Symposium on Operating Systems Principles (2023)

[34] Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One server for the price of two: Simple and fast single-server private information retrieval. In: Calandrino, J.A., Troncoso, C. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023 (2023)

[35] Kogan, D., Corrigan-Gibbs, H.: Private blocklist lookups with checklist. In: 30th USENIX Security Symposium (USENIX Security 21) (2021)

[36] Kushilevitz, E., Ostrovsky, R.: Replication is NOT needed: SINGLE database, computationally-private information retrieval. In: 38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997 (1997)

[37] Lazzaretti, A., Papamanthou, C.: TreePIR: Sublinear-time and polylog-bandwidth private information retrieval from DDH. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II (2023)

[38] Lin, W., Mook, E., Wichs, D.: Doubly efficient private information retrieval and fully homomorphic RAM computation from ring LWE. In: Saha, B., Servedio, R.A. (eds.) Proceedings of the 55th Annual ACM Symposium on Theory of Computing, STOC 2023, Orlando, FL, USA, June 20-23, 2023

[39] Melchor, C.A., Barrier, J., Fousse, L., Killijian, M.: XPIR : Private information retrieval for everyone. Proc. Priv. Enhancing Technol. (2016)

[40] Menon, S.J., Wu, D.J.: SPIRAL: Fast, high-rate single-server PIR via FHE composition. In: 2022 IEEE Symposium on Security and Privacy. pp. 930–947. IEEE Computer Society Press, San Francisco, CA, USA (May 22–26, 2022). https://doi.org/10.1109/SP46214.2022.9833700

[41] Menon, S.J., Wu, D.J.: Spiral: Fast, high-rate single-server PIR via FHE composition. IACR Cryptol. ePrint Arch. p. 368 (2022), https://eprint.iacr.org/2022/368

[42] Micciancio, D., Schultz, M.: Error correction and ciphertext quantization in lattice cryptography. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology – CRYPTO 2023. pp. 648–681. Springer Nature Switzerland, Cham (2023)

[43] Mittal, P., Olumofin, F., Troncoso, C., Borisov, N., Goldberg, I.: PIR-Tor: Scalable anonymous communication using private information retrieval. In: 20th USENIX Security Symposium (USENIX Security 11) (2011)

[44] Mughees, M.H., Chen, H., Ren, L.: Onionpir: Response efficient single-server pir. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. CCS '21 (2021)

[45] Patel, S., Persiano, G., Yeo, K.: Private stateful information retrieval. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. CCS '18 (2018)

[46] Peikert, C., Vaikuntanathan, V., Waters, B.: A framework for efficient and composable oblivious transfer. In: Wagner, D. (ed.) Advances in Cryptology – CRYPTO 2008. Lecture Notes in Computer Science, vol. 5157, pp. 554–571. Springer, Heidelberg, Germany, Santa Barbara, CA, USA (Aug 17–21, 2008). https://doi.org/10.1007/978-3-540-85174-5_31

[47] Peikert, C., Waters, B.: Lossy trapdoor functions and their applications. In: Ladner, R.E., Dwork, C. (eds.) 40th Annual ACM Symposium on Theory of Computing. pp. 187–196. ACM Press, Victoria, BC, Canada (May 17–20, 2008). https://doi.org/10.1145/1374376.1374406

[48] Regev, O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N., Fagin, R. (eds.) 37th Annual ACM Symposium

on Theory of Computing. pp. 84–93. ACM Press, Baltimore, MA, USA (May 22–24, 2005). https://doi.org/10.1145/1060590.1060603

[49] The reference implementation of SimplePIR and DoublePIR. https://github.com/ahenzinger/simplepir (2023)

[50] Thomas, K., Pullman, J., Yeo, K., Raghunathan, A., Kelley, P.G., Invernizzi, L., Benko, B., Pietraszek, T., Patel, S., Boneh, D., Bursztein, E.: Protecting accounts from credential stuffing with password breach alerting. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019 (2019)

[51] Tiptoe's linearly homomorphic encryption scheme. https://github.com/ahenzinger/underhood (2023)

[52] Tor: The tor project, https://www.torproject.org/, https://www.torproject.org/

[53] Vershynin, R.: High-dimensional probability: An introduction with applications in data science, vol. 47. Cambridge university press (2018)

[54] Yeo, K.: Lower bounds for (batch) PIR with private preprocessing. In: Hazay, C., Stam, M. (eds.) Advances in Cryptology – EUROCRYPT 2023, Part I. Lecture Notes in Computer Science, vol. 14004, pp. 518–550. Springer, Heidelberg, Germany, Lyon, France (Apr 23–27, 2023). https://doi.org/10.1007/978-3-031-30545-0_18

[55] Zhou, M., Park, A., Zheng, W., Shi, E.: Piano: Extremely simple, single-server pir with sublinear server computation. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 55–55. IEEE Computer Society, Los Alamitos, CA, USA (may 2024). https://doi.org/10.1109/SP54263.2024.00055, https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00055

A Sub-Gaussian Analysis

Definition 8 (Sub-Gaussian Random Variable). *A random variable X is said to be sub-Gaussian of parameter σ if for every $t > 0$*

$$\Pr[|X| > t] \leq \exp(-t^2/2\sigma^2). \tag{7}$$

We define the minimum sub-Gaussian parameter of a random variable X to be $\|X\|_{\psi_2}$. For a random vector $\mathbf{x}$, we define $\|\mathbf{x}\|_{\psi_2} = \max_i \|\mathbf{x}_i\|_{\psi_2}$.

Definition 9 (Sub-Exponential Random Variable). *A random variable X is said to be sub-Exponential of parameter σ if for every $t > 0$*

$$\Pr[|X| > t] \leq \exp(-t/2\sigma).$$

We define the minimal sub-Exponential parameter of a random variable X to be $\|X\|_{\psi_1}$. For a random vector $\mathbf{x}$, we define $\|\mathbf{x}\|_{\psi_1} = \max_i \|\mathbf{x}_i\|_{\psi_1}$.

We finally define the norm $\|\mathbf{x}\|_{\psi_\infty}$ as the maximal (in the ℓ_∞ norm) value that the random vector $\mathbf{x}$ takes on, e.g. the minimal parameter B such that $\Pr[\|\mathbf{x}\|_\infty > B] = 0$.

The quantities $\|\cdot\|_{\psi_\alpha}$ are norms on the space of random variables, e.g. are closed under scalar multiplication and addition of (possibly dependent) random variables. When the random variables are independent, one may obtain tighter bounds.

Lemma 8 (Pythagorean Additivity). *Let $X_1, \ldots, X_n$ be independent random variables. Then*

$$\left\|\sum_i X_i\right\|_{\psi_2} \leq \sqrt{\sum_i \|X_i\|_{\psi_2}^2}.$$

We will additionally need that $\|xy\|_{\psi_\alpha} \leq \|x\|_{\psi_\alpha}\|y\|_{\psi_\infty}$, and $\|xy\|_{\psi_1} \leq \|x\|_{\psi_2}\|y\|_{\psi_2}$. Both are well-known, see for example [53].

We will need the following results regarding bounding $f * g$ assuming the coefficients of f, g are of bounded ψ_α-norm. The following mildly extends [41, Lemma 2.6].

Lemma 9 (ψ_α bounds on $*$). *Let f, g be random variables such that $\deg f \leq D_f$ and $\deg g \leq D_g$. Let $D = \min(D_f, D_g)$. Then*

- *If $\|f\|_{\psi_2}, \|g\|_{\psi_\infty} < \infty$, then*

$$\|f * g\|_{\psi_2} \leq D\|f\|_{\psi_2}\|g\|_{\psi_\infty},$$

and if all coordinates of f and g are (mutually) independent, then

$$\|f * g\|_{\psi_2} \leq \sqrt{D}\|f\|_{\psi_2}\|g\|_{\psi_\infty}.$$

- *If $\|f\|_{\psi_2}, \|g\|_{\psi_2} < \infty$, then*

$$\|f * g\|_{\psi_1} \leq D\|f\|_{\psi_2}\|g\|_{\psi_2}.$$

Proof. All of the bounds reduce to analyzing any coordinate of the product $f * g$, which is a sum of at most D products of coordinates $\mathbf{f}_i \mathbf{g}_{i'}$. Our claimed bounds then follow immediately from the previously described properties of $\|\cdot\|_{\psi_\alpha}$.

We will use the independence heuristic, or the heuristic assumption that intermediate values within homomorphic computations are independent, and therefore one may apply pythagorean addivity in all situations. This means the noise bounds we derive will be heuristic — we will validate them against our implementation in Section 7.

We will need the following standard tail-bound on (possibly dependent) ψ_α-random variables.

Lemma 10. *Let $\mathbf{u}$ be an n-dimensional random variable of bounded ψ_α-norm for $\alpha \neq \infty$. Then for any $\delta > 0$*

$$\Pr[\|\mathbf{u}\|_\infty > \sqrt[\alpha]{\ln(1 + n\delta^{-1})}\|\mathbf{u}\|_{\psi_\alpha}] \leq \delta.$$

B The LWE and RLWE Problems

Definition 10 (LWE **Distribution).** *Let $N, m, Q \in \mathbb{N}$, and let $\sigma > 0$. Let $\mathbf{s} \in \mathbb{Z}_Q^N$. Then we call the distribution*

$$(A, A \cdot \mathbf{s} + \mathbf{e}),$$

where $A \leftarrow \mathbb{Z}_Q^{m \times N}, \mathbf{e} \leftarrow \chi_\sigma^m$ the LWE distribution*.*

For appropriate $\mathbf{s}$ (sampled either uniformly, or from χ_σ^N), the computational problem of distinguishing the LWE distribution from the uniform distribution is known as the decisional LWE problem.

Definition 11 (LWE **Problem).** *Let $N, m, Q \in \mathbb{N}$, and let $\sigma > 0$. The $\mathsf{LWE}_{Q,\sigma}^{N,m}$ problem is to distinguish samples from a distribution that is either*

1. *the LWE distribution relative to $\mathbf{s} \leftarrow \chi_\sigma^N$, or*
2. *the uniform distribution over $\mathbb{Z}_Q^{m \times (N+1)}$.*

We will also use the RLWE problem (restricted to power-of-two cyclotomic rings) in this work. In the ring setting, we abuse notation and use χ_σ^n to mean the distribution over degree n polynomials whose coefficients are independently sampled from χ_σ.

Definition 12 (RLWE **Distribution).** *Let $k, q \in \mathbb{N}$, and let $n = 2^k$. Let $s \in R_{n,q}$. Then we call the distribution*

$$(a, a * s + e),$$

where $a \leftarrow R_{n,q}, e \leftarrow \chi_\sigma^n$, the RLWE distribution*.*

Definition 13 (RLWE **Problem).** *Let $k, q \in \mathbb{N}$, let $\sigma > 0$, and let $n = 2^k$. The $\mathsf{RLWE}_{q,\sigma}^n$ problem is to distinguish samples from a distribution that either*

1. *samples $s \leftarrow \chi_\sigma^n$, then outputs samples from the RLWE distribution, or*
2. *outputs samples from the uniform distribution over $R_{n,q}^2$.*

It is well-known ([47, Section 6]) that one may fix a and resample s_i, rather than fix s and resample a, with only a mild reduction in security.

Lemma 11. *For any integer $\kappa \geq 1$, let ϵ_κ denote the maximal advantage of an adversary in distinguishing between*

- *the distribution $(a, a * s_1 + e_1, a * s_2 + e_2, \ldots, a * s_\kappa + e_\kappa)$ for $s_i, e_i \leftarrow \chi_\sigma^n$ i.i.d, and $a \leftarrow R_{n,q}$, and*
- *the uniform distribution on $R_{n,q}^{1+\kappa}$.*

Then $\epsilon_\kappa \leq \kappa \epsilon_1$.

A similar result holds for LWE. While one can choose $\kappa = \mathsf{poly}(n)$ without degrading (asymptotic) security, there is a (multiplicative) concrete loss of security which necessitates larger parameters n, q. We will therefore choose (for $T_{\mathsf{preprocess}}$ and T_{query} the server preprocessing and query response times) $\kappa = \omega(T_{\mathsf{preprocess}}/T_{\mathsf{response}})$, so that the amortized running time

$$T_{\mathsf{amortize}} = \frac{T_{\mathsf{preprocess}} + \kappa T_{\mathsf{response}}}{\kappa} = (1 + o(1))T_{\mathsf{response}},$$

is asymptotically the same as the server response time.

C Homomorphic Multiplications

Below, we include material regarding (unrelinearized) homomorphic LWE $\otimes$'s and RLWE $*$'s, which are used in our scheme $\mathsf{TensorPIR}$.

Lemma 12 **(LWE Homomorphic $\otimes$-product).** *Given two LWE-based ciphertexts $C_i \in \mathbb{Z}_Q^{d_i \times n} \times \mathbb{Z}_Q^{d_i \times 1}$ encrypting $\mathbf{m}_i \in \mathbb{Z}_Q^{d_i}$ with scaling factor Δ and error $\mathbf{e}_i$, one may homomorphically compute a LWE-based ciphertext*

$$C_{\otimes} \in \mathbb{Z}_{Q^2}^{d_0 d_1 \times n^2} \times \mathbb{Z}_{Q^2}^{d_0 d_1 \times 2n} + \mathbb{Z}_{Q^2}^{d_0 d_1 \times 1}$$

such that

$$C_{\otimes}(\mathbf{s} \otimes \mathbf{s}, -\mathbf{s}, 1) = \Delta^2 \mathbf{m_0} \otimes \mathbf{m_1} + \mathbf{e}_{\otimes}$$

where

$$\mathbf{e}_{\otimes} = \Delta\left(\mathbf{m_0} \otimes \mathbf{e}_1 + \mathbf{e}_0 \otimes \mathbf{m}_1\right) + \mathbf{e}_0 \otimes \mathbf{e}_1.$$

Lemma 13 **(RLWE Homomorphic $*$-product).** *Given two RLWE-based ciphertexts $\mathsf{ct}_i \in R_{n,q}^2$ encrypting $m_i \in R_{n,q}$ with scaling factor Δ and error e_i, one may homomorphically compute a RLWE-based ciphertext $\mathsf{ct}_* \in R_{n,q^2}^3$ that may be linearly-decrypted to*

$$\Delta^2 m_0 * m_1 + e_*$$

where

$$e_* = \Delta(m_0 * e_1 + e_0 * m_1) + e_0 * e_1.$$

D The Noise Growth of our Homomorphic Computations

In some applications we will require all $n/2$ rotation keys. One can generate these from a single rotation key at the cost of mild noise growth.

Lemma 14. *Let $\mathbf{g}$ be a gadget of size ℓ and quality γ. Let v be a RLWE secret-key. Let ksk be a $\mathbf{g}$-based key-switching key from $v(X^5)$ to v with error that has ψ_2-norm at most σ and ψ_∞-norm at most B. Then one can compute all $n/2$*

rotation keys from ksk *at the cost of* $\ell(n-1)$ $\diamond$*-products. Moreover, the error* e' *of any of these rotation keys satisfies*

$$\|e'\|_{\psi_2} \leq \sqrt{\ell}\gamma\sigma^2 n.$$

and

$$\|e'\|_{\psi_\infty} \leq \ell\gamma B n^2.$$

Proof. We compute each rotation key iteratively from the last. Note that a single application increases the error from $e \mapsto e + \sum_{i\in[\ell]} \mathbf{g}^{-1}(a)_i * e_i$, where e_i are the errors in the initial rotation key. Iterating, we see that the error in the jth component of the ith rotation key is of the form

$$e_{ij} = e_{(i-1)j} + \sum_{k\in[\ell]} \mathbf{g}^{-1}(a_{(i-1)j})_k * e_k. \tag{8}$$

We can solve the recurrence to see that we can instead write

$$\begin{aligned} e_{ij} &= \sum_{h\in(2,i-1]} \sum_{k\in[\ell]} \mathbf{g}^{-1}(a_{hj})_k * e_k \\ &= \sum_{k\in[\ell]} e_k * \left(\sum_{h\in(2,i-1]} \mathbf{g}^{-1}(a_{hj})_k \right) \end{aligned}$$

We next note that $\sum_h \mathbf{g}^{-1}(a_{hj})_k$ is of ψ_2-norm at most $\sqrt{n/2}\gamma$. Recalling $\|e_k\|_{\psi_\infty} \leq B$ and applying Lemma 9, it easily follows that the overall error is of ψ_2-norm at most $\sqrt{\ell(n/2)n}B\gamma$, e.g. the claimed bound holds.

The bounds in the ψ_∞ case proceed analogously, but are much weaker. In particular, we can get the bound $\|e'\|_{\psi_\infty} \leq \ell\gamma B n^2$. □

We will require a frequently used technique [3, 4, 15, 24, 31] to homomorphically expand $\mathsf{Enc}(\sum_i a_i X^i) \mapsto \{\mathsf{Enc}(na_i)\}_i$.

Lemma 15 **(**RLWE **Expansion [15]).** *Let* ct *be an* RLWE *encryption of* $\sum_{i\in[n]} a_i X^i \in R_{n,p}$ *under secret key* v*. Then one may homomorphically compute* n *ciphertexts* $\mathsf{ct}_i = \mathsf{Enc}_v(na_i)$ *with a total of* $n-1$ $\diamond$*-products. If the error* e *in the input ciphertext satisfies* $\|e\|_{\psi_2} \leq \sigma$*, and the error* e_i *of each rotation key satisfies* $\|e_i\|_{\psi_2} \leq \sigma_{\mathsf{ksk}}$*, then the error* e' *of each output ciphertext* ct_i *satisfies*

$$\|e'\|_{\psi_2}^2 \leq n^2\sigma^2 + \frac{n^2-1}{3}\ell\gamma^2\sigma_{\mathsf{ksk}}^2$$

Note that this computes encryptions of constant polynomials $m_i(X) := n \cdot a_i$ rather than a_i. One can fix this by instead initially encrypting $\sum_{i\in[n]} n^{-1} \cdot a_i X^i \bmod p$.

We will sometimes call $\mathsf{Expand}_{\mathsf{ksk}}$ on a vector of RLWE ciphertexts $\overrightarrow{\mathsf{ct}}$ of dimension d, e.g. $\mathsf{Expand}_{\mathsf{ksk}}(\overrightarrow{\mathsf{ct}})$. By this, we mean calling $\mathsf{Expand}_{\mathsf{ksk}}(\mathsf{ct}_i)$ for each $i \in [d]$, and concatenating the results.

We will need the following result regarding the correctness of LWEPIR.

```
$\mathsf{Expand}_{\mathsf{ksk}}(\mathsf{ct})$
$\mathsf{ct}_0 := \mathsf{ct}$
for $i \in [\log n]$
    $k = n/2^i + 1$
    for $b \in [2^{i+1}]$
        $\mathsf{ct}_{2b} = \mathsf{ct}_b + \mathsf{Rotate}_{\mathsf{ksk}}(\mathsf{ct}_b, k)$
        $\mathsf{ct}_{2b+1} = (\mathsf{ct}_b - \mathsf{Rotate}_{\mathsf{ksk}}(\mathsf{ct}_b, k)) * X^{-k}$
return $\{\mathsf{ct}_i\}_{i \in [n]}$
```

Fig. 6. The RLWE Expansion Algorithm of [15], where $\mathsf{ct} = \mathsf{Enc}(\sum_i a_i X^i)$ is an RLWE encryption of many scalars a_i, and ct_i is an RLWE encryption of the single scalar na_i. Note that this algorithm solely computes homomorphic additions and rotations, and is compatible with our preprocessing techniques of Section 3.

Lemma 16. *Let* $\mathsf{DB} \in \mathbb{Z}_p^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$*. The error in a* LWEPIR *server response is sub-Gaussian of parameter at most* $\sqrt{n_{\mathsf{cols}}} p\sigma$*.*

Proof. It is straightforward to verify that the error in the server response is $\mathsf{DB} \cdot \mathbf{e}$ for fresh LWE error $\mathbf{e} \leftarrow \chi_\sigma^{n_{\mathsf{rows}}}$. Standard sub-Gaussian analysis then gives that $\|\mathbf{e}\|_{\psi_2}^2 \leq n_{\mathsf{cols}} \sigma^2 p^2$.

E Analysis of NTTlessPIR

Theorem 2. *Let* $\mathsf{DB} \in \mathbb{Z}_Q^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$*. Let* $\mathbf{m} \in \mathbb{Z}^{n_{\mathsf{cols}}}$ *be such that* $\|\mathbf{m}\|_\infty \leq B$*. Let* σ, q*, and* n *be such that the* RLWE *assumption is hard. Then, assuming the circular security of* RLWE*, the* LinPIR *scheme specified in Figure 3 is a secure* LinPIR *scheme in the random oracle model.*

Proof. If it were not for our seed reuse optimization, the security of our LinPIR scheme would immediately reduce to the security of the underlying FHE scheme, e.g. to the security of RLWE with our parameter set (as well as a circular security assumption to handle our inclusion of the rotation key $\mathsf{Enc}_v(\mathsf{rot}^{\circ 1}(v))$). This seed reuse optimization degrades security by at most a multiplicative factor κ (Lemma 11). As we assume the server is efficient, its running time (which upper-bounds κ) is at most polynomial in λ, and the security of our scheme degrades by a factor at most polynomial in the security parameter.

As suggested after Lemma 11, we set κ minimal such that the amortized cost of preprocessing disappears (asymptotically). For NTTlessPIR, one can check that this is $\kappa = \omega(\log n)$. For this reason, we will ignore this overhead in our asymptotic analysis of the running time of our LinPIR scheme (though we take this into account in our experiments in Section 7).

Lemma 17. *Let* $\mathsf{DB} \in \mathbb{Z}_Q^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$. *Let* $\mathbf{m}$ *satisfy* $\|\mathbf{m}\|_\infty \leq B$. *Provided the underlying FHE scheme can correct errors of size up to*

$$\sqrt{\ln(1 + \frac{k(n_{\mathsf{rows}} + n)}{\delta})} \max_j \sqrt{\ell n_{\mathsf{cols}}} n \sigma \gamma p_j,$$

and $\prod_i p_i > n_{\mathsf{cols}} p B$, *the* LinPIR *scheme specified in Figure 3 is a correct* LinPIR *scheme with probability at least* $1 - \delta$.

Proof. We first compute the errors in the server response. Note that the server homomorphically

- computes (at most) n_{cols} rotations of the client's fresh ciphertexts $\widehat{b}_j$, and then
- takes a linear combination with coefficients of size at most p_j of these rotated ciphertexts.

We analyze the noise growth of each part separately. Note that each $\diamond$-product computes a sum of ℓ products of polynomials f, g. These polynomials are

- f: the gadget-decompositions of other polynomials, e.g. of ψ_∞-norm at most γ, and
- g: the error contained within the key-switching key, e.g. of ψ_2-norm at most σ.

By Lemma 9, each product has ψ_2-norm at most $\sqrt{n}\gamma\sigma$, and therefore the $\diamond$-product increases the ψ_2-norm of the error by an additive factor at most $\sqrt{\ell n}\sigma\gamma$. It follows that each of the n_{cols} rotations of ciphertexts has ψ_2-norm at most $\sqrt{\ell n_{\mathsf{cols}}} n\sigma\gamma$.

The server then uses these rotations to homomorphically compute Eq. (5), e.g. to compute a sum of n_{cols} polynomial products, where one of the polynomials has ψ_2-norm at most $\sqrt{\ell n_{\mathsf{cols}}} n\sigma\gamma$, and the other has ψ_∞-norm at most $\max_j p_j$. Under the independence heuristic, this leads to errors of size at most $\max_j \sqrt{\ell n_{\mathsf{cols}}} n\sigma\gamma p_j$. Note that if we concatenate all of our error vectors, we obtain one (large) error vector $\mathbf{e}$ of dimension $k\lceil n_{\mathsf{rows}}/n \rceil n \leq k(n_{\mathsf{rows}} + n)$. By Lemma 10, we get that

$$\Pr[\|\mathbf{e}\|_\infty > \sqrt{\ln(1 + \frac{k(n_{\mathsf{rows}} + n)}{\delta})} \max_j \sqrt{\ell n_{\mathsf{cols}}} n\sigma\gamma p_j] \leq \delta,$$

e.g. there will be a ciphertext that decrypts incorrectly with probability at most δ.

We next estimate how many NTT-friendly primes p_j we will require for the client's CRT interpolation to succeed. We require that $\prod_j p_j \geq \|\mathsf{DB} \cdot \mathbf{m}\|_\infty$, where the matrix-vector multiplication is computed over $\mathbb{Z}$. In the worst-case, this requires that $\prod_j p_j > n_{\mathsf{cols}} \|\mathsf{DB}\|_\infty \|\mathbf{m}\|_\infty$, where $\|\mathsf{DB}\|_\infty$ is the ℓ_∞-norm of DB viewed as a vector. Provided this condition holds, it is straightforward to see that $\mathsf{DB} \cdot \mathbf{m} \bmod \prod_j p_j$ is equal to $\mathsf{DB} \cdot \mathbf{m}$ over $\mathbb{Z}$, so may be successfully reduced $\bmod\ Q$ to recover $\mathsf{DB} \cdot \mathbf{m} \bmod Q$.

Note that if we may assume that the database DB has uniformly random entries, an average-case analysis can weaken the condition $\prod_j p_j > n_{\mathsf{cols}}\|\mathsf{DB}\|_\infty\|\mathbf{m}\|_\infty$ to $\Omega(\sqrt{n_{\mathsf{cols}}}\|\mathsf{DB}\|_\infty\|\mathbf{m}\|_\infty)$ via a standard sub-Gaussian analysis.

E.1 Further Optimizations for NTTlessPIR

The following optimizations give non-asymptotic improvements to NTTlessPIR without impacting security. Our implementation in Section 7 currently only uses the first optimization. The other two could practically decrease the size of our server response by a factor ≈ 4×, at the cost of introducing database-dependent state (analogous to that of Simple PIR) to our clients. The state is much smaller in our setting than that of Simple PIR (on the order of hundreds of kilobytes, rather than megabytes), so this may be acceptable. We instead have chosen to completely remove database-dependent state from our clients.

Packing Theorem 1 In Section 3, we presented homomorphic encryption algorithms, e.g. where one ends up with an ciphertext (for example) $\mathsf{ct} = \mathsf{Enc}(A\cdot\mathbf{m})$ that decrypts to $A\cdot\mathbf{m}$ For matrix-vector multiplication in particular (where the matrix A has a number of columns n_{cols} that is a proper divisor $n_{\mathsf{cols}} \mid n$ of the RLWE dimension), we may obtain a constant-factor speedup, by weakening our requirement that $\mathsf{Dec}(\mathsf{ct}) = A\cdot\mathbf{m}$ exactly, to that it can be efficiently post-processed to $A\cdot\mathbf{m}$.

Recall that our matrix-vector multiplication homomorphically evaluates the formula

$$A\cdot\mathbf{m} = \sum_{i\in[n_{\mathsf{cols}}]} \mathrm{diag}_i(A)\circ \mathsf{rot}^{\circ i}(\mathbf{m}).$$

This only uses n_{cols} of our RLWE slots. For A where $n_{\mathsf{cols}} \neq n$, we can use the extra slots we have available to pack this single computation (of n_{cols} summands) into n/n_{cols} parallel computations (of $\approx n_{\mathsf{cols}}/(n/n_{\mathsf{cols}}) \approx n_{\mathsf{cols}}^2/n$ summands). As the complexity of our preprocessing algorithm in Lemma 4 scales linearly with the number of summands, this gives us a constant-factor improvement to the most expensive part of our protocol effectively for free.

We discuss concretely for $n/n_{\mathsf{cols}} = 2$ — the general case easily follows. Assume as well that $2 \mid n_{\mathsf{cols}}$ for simplicity.

Note that both $\mathrm{diag}_i(A)$ and $\mathsf{rot}^{\circ i}(\mathbf{m})$ are of length n_{cols}. To homomorphically compute Lemma 5 in $\mathbb{Z}_p^n$, we must replace $\mathbf{m}$ with $\mathbf{m}' = (\mathbf{m},\mathbf{m})$. This is so that $\mathsf{rot}^{\circ i}(\mathbf{m}') = (\mathsf{rot}^{\circ i}(\mathbf{m}), \mathsf{rot}^{\circ i}(\mathbf{m}))$ has the first n_{cols} as the expected value $\mathsf{rot}^{\circ i}(\mathbf{m})$. Then, the first n_{cols} coordinates of the result will contain $A\cdot\mathbf{s}$, as desired.

We can additionally make use of the second n_{cols} coordinates as follows. Set $\mathbf{d}_i' = (\mathrm{diag}_i(A), \mathrm{diag}_{(n_{\mathsf{cols}}/2)+i}(A))$. One can then check that

$$\sum_{i\in[n_{\mathsf{cols}}/2]} \mathbf{d}_i'\circ\mathsf{rot}^{\circ i}(\mathbf{m}') = \begin{pmatrix} \sum_{0\leq i<n_{\mathsf{cols}}/2}\mathrm{diag}_i(A)\circ\mathsf{rot}^{\circ i}(\mathbf{m}) \\ \sum_{n_{\mathsf{cols}}/2\leq i<n_{\mathsf{cols}}}\mathrm{diag}_i(A)\circ\mathsf{rot}^{\circ i-n_{\mathsf{cols}}/2}(\mathbf{m}). \end{pmatrix} \quad (9)$$

Summing the two halves of the vector then recovers Eq. (5), e.g. one can compute that equation using a sum of half as many terms $n_{\mathsf{cols}}/2$. More generally, one can reduce the number of summands by a factor $\lceil n/n_{\mathsf{cols}}\rceil$. This directly reduces the number of rotations one must generate using Lemma 4, saving a factor in $\lceil n/n_{\mathsf{cols}}\rceil$ in running time in the most expensive part of our protocol, e.g. speeding things up by a factor 2 or 4 in practice.

Reducing our Server Response's Size by Half Our server response transmits many NTT-domain ciphertexts $[\widehat{a}_{i,j}, \widehat{b}_{i,j}]$ to the client. These ciphertexts are the result of homomorphically evaluating $\mathsf{Apply}_{\{A \bmod p_j\}_j}$. Their public randomness $\mathsf{Apply}^{\alpha}_{\{A \bmod p_j\}_j}$ is therefore a function of solely the precomputed value $\mathsf{Preproc}_{\{A \bmod p_j\}_j}$, and therefore may be computed by the server before the protocol occurs.

It follows that the server can augment the protocol's public parameters (as specified, only a λ-bit seed) to additionally contain $\mathsf{Apply}^{\alpha}_{\{A \bmod p_j\}_j}$. This is an RLWE variant of the database-dependent hint of LWEPIR, though it is much smaller size (in particular, at most half of the size of a server response), compared to the LWEPIR hint, which is $N \approx 2^{10}$ times larger than a server response. This is to say that removing the transmission of this quantity is not nearly as impactful of an optimization as in LWEPIR (and consequentially, we may ignore this optimization, if we do not wish for our public parameters to contain a database-dependent hint).

Note that regardless of whether one sends these database-dependent parameters to the client ahead of time, the server can store them long-term, removing the need to recompute them during each query. This speeds up running time of the server response by a factor ≈ 2.

Lossily Compressing the $\widehat{b}$ Components of the Server Response The server responds to the client with several NTT-domain RLWE ciphertexts, e.g. elements of $\mathbb{Z}_q^n$ (or $(\mathbb{Z}_q^n)^2$ if one does not apply the previous optimization). As we no longer need to homomorphically compute on these ciphertexts, one can use standard techniques (modulus switching, or perhaps the compression technique of [13], which was shown to be quasi-optimal for compressing LWE ciphertexts in [42]) to reduce these elements of $\mathbb{Z}_q^n$ to nearly $\mathbb{Z}_{p_i}^n$, where p_i is the plaintext modulus. This ends up saving an additional factor $\approx k$ over solely the previous optimization. Note that both of the mentioned compression techniques must be computed on coefficient-domain representations of the ciphertexts, so this technique does introduce some mild number of online iNTTs (k, at a cost of $\Theta(kn \log n)$ $\mathbb{Z}_q$ operations) to the server response, but in practice this is negligible compared to the $\Omega(kn_{\mathsf{cols}}n_{\mathsf{rows}})$ complexity of the rest of the protocol.

F Correctness and Security Analysis of HintlessPIR

Lemma 18 (Security of HintlessPIR). *Let (N, Q, σ) be such that* LWE *is hard. Let (n, q, σ) be such that* RLWE *is hard, and moreover assume that* RLWE *is circular secure. Then* HintlessPIR *is a secure* PIR *with preprocessing scheme.*

Proof. When querying HintlessPIR on an index i, the adversary observes

- an LWEPIR query on i, and
- a RLWE encryption of the LWE secret key, and
- a RLWE rotation key.

It is straightforward to see that under the decisional LWE and RLWE assumptions (and a circular security assumption) that this is not only indistinguishable from a HintlessPIR query on index j, but is indistinguishable from uniformly random strings on the same domain, and therefore HintlessPIR is secure.

Again, when handling multiple queries, we must handle the degredation of the security of LWE and RLWE-based encryption with reuse of the pads $\mathbf{A}$ (via a variant of Lemma 11 for LWE). Here, given Simple PIR's high cost to regenerate its hint, we want to instead reseed after every $\omega(N)$ queries so that the amortized cost of the protocol does not increase. This requires setting larger LWE parameters than is typically required, which we do in Section 7

Lemma 19 (Correctness of HintlessPIR). *Let $\mathsf{DB} \in \mathbb{Z}_p^{n_{\mathsf{rows}} \times n_{\mathsf{cols}}}$. Let (N, Q, σ) be such that* LWE *is hard. Let (n, q, σ) be such that* RLWE *is hard. Then, for any $\delta > 0$, provided*

$$Q > \sqrt{n_{\mathsf{cols}}} p^2 \sigma \sqrt{\ln(1 + \frac{n_{\mathsf{rows}}}{\delta/3})}$$

$$q > \sqrt{\ln(1 + \frac{k(n_{\mathsf{rows}} + n)}{\delta/3})} \max_j \sqrt{\ell} N n \sigma \gamma p_j^2$$

$$\prod_j p_j > Q\sigma\sqrt{N} \sqrt{\ln(1 + \frac{kn\lceil n_{\mathsf{rows}}/n \rceil}{\delta/3})},$$

then HintlessPIR *is correct with probability at least $1 - \delta$.*

Proof. We parameterize each part of the protocol that may fail (the LWEPIR query, the NTTlessPIR query's decryption, and NTTlessPIR's post-decryption CRT interpolation) such that they fail with probability at most $\delta/3$, and then get that HintlessPIR fails with probability at most δ, as desired.

G TensorPIR: Detailed Protocol and Analysis

We adopt the same CRT decomposition technique used in Section 4 to handle homomorphic computation over arbitrary modulus Q. Namely, we homomorphically compute the above terms over k NTT-friendly moduli p_j such that the plaintext computation never wrap around $\mathrm{mod}_{j\in[k]} \prod_j p_j$. Since the homomorphic computation over each p_j is exactly the same, we describe TensorPIR without explicitly mentioning these plaintext moduli.

The high level idea about TensorPIR is that, we can rearrange database as $\mathsf{DB} \in \mathbb{Z}^{d_\mathbf{w} \times d_\mathbf{u} \times d_\mathbf{v}}$, and it holds that

$$\mathsf{DB} = [\mathsf{DB}_1, \ldots, \mathsf{DB}_{d_\mathbf{u}}] = \sum_{i\in[d_\mathbf{u}]} {\mathbf{u_i}}^t \otimes \mathsf{DB}_i. \tag{10}$$

If the client encrypts two selection vectors $\mathbf{u} \in \{0,1\}^{d_\mathbf{u}}$ and $\mathbf{v} \in \{0,1\}^{d_\mathbf{v}}$ into LWE ciphertexts $C_0 = [A_0, \mathbf{b}_0]$ and $C_1 = [A_1, \mathbf{b}_1]$, then one may compute $\mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v})$ as

$$\begin{aligned} \mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v}) \approx \mathsf{DB} \cdot (\mathbf{b}_0 \otimes \mathbf{b}_1) - \mathsf{DB} \cdot (A_0 \cdot \mathbf{s} \otimes \mathbf{b}_1) - \mathsf{DB} \cdot (\mathbf{b}_0 \otimes A_1 \cdot \mathbf{s}) \\ + \mathsf{DB} \cdot (A_0 \cdot \mathbf{s} \otimes A_1 \cdot \mathbf{s}) \bmod Q. \end{aligned} \tag{11}$$

The right hand side is a noisy version of $\mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v})$; so if the client can obtain these terms, then it can round and remove the error to get the desired records.

We now describe TensorPIR in more details. Recall that the client encrypts $\mathbf{u}$ and $\mathbf{v}$ under its LWE secret $\mathbf{s}$ to obtain ciphertexts

$$C_0 = [A_0, \mathbf{b}_0 = A_0\mathbf{s} + \mathbf{e} + \Delta\mathbf{u}], C_1 = [A_1, \mathbf{b}_1 = A_1\mathbf{s} + \mathbf{e} + \Delta\mathbf{v}].$$

The client additionally uses a RLWE-based scheme Enc for the terms involving the LWE secret vector $\mathbf{s}$. Specifically, the client samples a fresh RLWE secret key v and sends the following ciphertexts to the server:

- $\mathsf{ct}_{A_0\mathbf{s}} \leftarrow \mathsf{Enc}_v(\sum_{i\in[d_\mathbf{u}]} \langle \mathbf{a}_i, \mathbf{s} \rangle \cdot X^i)$, and
- $\mathsf{ct}_\mathbf{s} \leftarrow \mathsf{Enc}_v(\mathsf{encode}(\mathbf{s}))$,

where $\mathbf{a}_i = {\mathbf{u_i}}^t \cdot A_0$ is the ith row of A_0. The client also includes a Galois key for $\theta : X \mapsto X^5$ in its query.

The server's task is to perform homomorphic computation to obtain

1. $\mathsf{DB} \cdot (\mathsf{Enc}_v(A_0 \cdot \mathbf{s}) \otimes \mathbf{b}_1)$,
2. $\mathsf{DB} \cdot (\mathbf{b}_0 \otimes A_1 \cdot \mathsf{Enc}_v(\mathbf{s}))$,
3. $\mathsf{DB} \cdot (\mathsf{Enc}_v(A_0 \cdot \mathbf{s}) \otimes A_1 \cdot \mathsf{Enc}_v(\mathbf{s}))$, and
4. $\mathsf{DB} \cdot (\mathbf{b}_0 \otimes \mathbf{b}_1)$.

Let us start with the term $\mathsf{DB} \cdot (\mathsf{Enc}_v(A_0 \cdot \mathbf{s}) \otimes A_1 \cdot \mathsf{Enc}_v(\mathbf{s}))$. According to Eq. (10), we can expand the underlying plaintext computation as $\mathsf{DB} \cdot (A_0 \cdot \mathbf{s} \otimes A_1 \cdot \mathbf{s}) = \sum_{i\in[d_\mathbf{u}]} ({\mathbf{u_i}}^t \cdot A_0 \cdot \mathbf{s}) \otimes (\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s})$. Since the ${\mathbf{u_i}}^t \cdot A_0 \cdot \mathbf{s}$ is one-dimensional, we can rewrite our homomorphic computation as

$$\mathsf{Enc}(A_0 \cdot \mathbf{s}), \mathsf{Enc}(\mathbf{s}) \mapsto \sum_{i\in[d_\mathbf{u}]} \mathsf{Enc}(\langle \mathbf{a}_i, \mathbf{s} \rangle) * \mathsf{Enc}(\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s}), \tag{12}$$

where $\mathbf{a}_i = {\mathbf{u_i}}^t \cdot A_0$ is the ith row of A_0.

Given the ciphertexts in a client query, the server can then homomorphically compute, for all $i \in [d_\mathbf{u}]$,

– the RLWE encryptions $\mathsf{Enc}_v(\langle \mathbf{a}_i, \mathbf{s}\rangle)$ of scalars $\langle \mathbf{a}_i, \mathbf{s}\rangle$, and
– the RLWE encryptions $\mathsf{Enc}_v(\mathsf{encode}(\mathsf{DB}_i \cdot A_1 \cdot \mathbf{s}))$.

The first set of ciphertexts can be efficiently generated from a compact encryption of $A_0 \cdot \mathbf{s}$ via RLWE expansion of Lemma 15. In practice, since the expansion algorithm requires $\log n$ rotation keys which are expensive to send in each query, we let the client send just a single rotation key corresponding to rotation by 1, and let the server generate all $\log n$ rotation keys via Lemma 14. The second set of ciphertexts are exactly the homomorphic matrix-vector products between $\mathsf{DB}_i \cdot A_1$ and a ciphertext encrypting $\mathbf{s}$ in the slots. So we invoke $\mathsf{NTTlessPIR}$ to compute them.

The above ciphertexts are also useful to compute the other two terms:

– For $\mathsf{DB} \cdot (\mathsf{Enc}(A_0 \cdot \mathbf{s}) \otimes \mathbf{b}_1)$, the server can multiply ct_i by the plaintext $\mathsf{DB}_i \cdot \mathbf{b}_1 \bmod p_j$ and homomorphically sum up these ciphertexts.
– For $\mathsf{DB} \cdot (\mathbf{b}_0 \otimes (A_1 \cdot \mathsf{Enc}(\mathsf{s})))$, the server can simply multiply ct'_i with a scalar $\langle \mathbf{b}_0, \mathbf{u_i}\rangle$, and sum up the resulting ciphertexts across all i's.

The full server and client algorithms of $\mathsf{TensorPIR}$ are shown in Fig. 7.

G.1 Security and Efficiency of TensorPIR

We first briefly discuss the security of $\mathsf{TensorPIR}$, which follows security properties of component schemes of the protocol and is standard.

Lemma 20 (Security of $\mathsf{TensorPIR}$). *Let $N, Q, n, q, m \in \mathbb{N}$ and let $\sigma > 0$. Assume $\mathsf{LWE}_{Q,\sigma}^{N,m}$ is hard, and assume $\mathsf{RLWE}_{q,\sigma}^{n}$ is hard. Moreover, assume that $\mathsf{RLWE}_{q,\sigma}^{n}$ is circular secure. Then $\mathsf{TensorPIR}$ is a secure PIR with preprocessing scheme for m-dimensional databases.*

We next estimate the size of the plaintext space we need for our homomorphic computation in $\mathsf{TensorPIR}$ to be correct.

Lemma 21. *Let $N, Q, n, q, m \in \mathbb{N}$, and let $\sigma > 0$. Then provided one computes the $\mathsf{TensorPIR}$ protocol with respect to NTT-friendly primes p_i such that*

$$\prod_i p_i > d_{\mathbf{u}} N Q^3,$$

CRT interpolation at the end of $\mathsf{TensorPIR}$ will succeed.

Proof. We estimate the size of result of the plaintext computation that we are homomorphically computing. For simplicity, we solely give a worst-case analysis[20]. Note that we are computing

$$\sum_i c_i (\mathsf{DB}_i \cdot A_1)\mathbf{c}',$$

[20] It seems unlikely an average-case analysis would help much, as the most problematic term, the Q^3, would be unaffected by this.

```
Server Algorithms in TensorPIR
S.setup(DB) :
  for $i \in [d_{\mathbf{u}}], j \in [k]$
    $(\mathsf{seed}_{i,j}, \mathsf{S}^{\mathsf{hint}}_{i,j}) \leftarrow \mathsf{NTTlessPIR.setup}(\mathsf{DB}_i A_1 \bmod p_j)$
  return $[\mathsf{seed}_{i,j}, \mathsf{S}^{\mathsf{hint}}_{i,j} : i \in [d_{\mathbf{u}}], j \in [k]]$
S.response($\overrightarrow{\mathsf{ct}_{A_0\mathbf{s}}}, \mathsf{ct}_{\mathbf{s}}, \mathsf{ksk}, \mathbf{b}_0, \mathbf{b}_1$)
  $\{\mathsf{ksk}_i\}_{i \in [n/2]} \leftarrow \mathsf{GenAllRotationKeys}(\mathsf{ksk})$
  for $j \in [k]$
    $\mathsf{ct}_{A_0\mathbf{s}\otimes\mathbf{b}_1,j} = \mathbf{0}$
    $\mathsf{ct}_{\mathbf{b}_0\otimes A_0\mathbf{s},j} = \mathbf{0}$
    $\mathsf{ct}_{A_0\mathbf{s}\otimes A_1\mathbf{s},j} = \mathbf{0}$
    for $i \in [d_{\mathbf{u}}], j \in [k]$
    $\{\mathsf{ct}_{i,j}\}_i \leftarrow \mathsf{Expand}_{\mathsf{ksk}}(\overrightarrow{\mathsf{ct}_{A_0\mathbf{s},j}})$
    $\{\mathsf{ct}'_{i,j}\} \leftarrow \mathsf{NTTlessPIR.response}(\mathsf{ct}_{\mathbf{s},j}, \mathsf{DB}_i \cdot A_1)$
  for $i \in [d_{\mathbf{u}}], j \in [k]$
    $\mathsf{ct}_{A_0\mathbf{s}\otimes\mathbf{b}_1,j} \mathrel{+}= \mathsf{ct}_{i,j} * \mathsf{encode}_{p_j}(\mathsf{DB}_i \cdot \mathbf{b}_1)$
    $\mathsf{ct}_{\mathbf{b}_0\otimes A_1\mathbf{s},j} \mathrel{+}= (\mathbf{b}_0)_i * \mathsf{ct}'_{i,j}$
    $\mathsf{ct}_{A_0\mathbf{s}\otimes A_1\mathbf{s},j} \mathrel{+}= \mathsf{ct}_{i,j} * \mathsf{ct}'_{i,j}$
  $\mathbf{d} = \mathsf{DB} \cdot (\mathbf{b}_0 \otimes \mathbf{b}_1)$
  return $(\{\mathsf{ct}_{A_0\mathbf{s}\otimes\mathbf{b}_1,j}, \mathsf{ct}_{\mathbf{b}_0\otimes A_1\mathbf{s},j}, \mathsf{ct}_{A_0\mathbf{s}\otimes A_1\mathbf{s},j}\}_{j\in[k]}, \mathbf{d})$
```

```
Client Algorithms in TensorPIR
C.query($\mathbf{u}, \mathbf{v}, \{\mathsf{seed}_{i,j}\}_{i\in[d_{\mathbf{u}}],j\in[k]}$)
  $\mathbf{s} \leftarrow \mathsf{LWE.KGen}(1^\lambda)$
  $v \leftarrow \mathsf{RLWE.KGen}(1^\lambda)$
  $\mathsf{ksk} \leftarrow \mathsf{RLWE.Enc}_v(\mathbf{g} * v(X^5); \mathsf{seed}_{0,0}||0)$
  $\overrightarrow{\mathsf{ct}_{A_0\mathbf{s}}} \leftarrow [\mathsf{RLWE.Enc}_v(A_0\mathbf{s} \bmod p_j; \mathsf{seed}_{i,j}||10) : j \in [k]]$
  $\mathsf{ct}_{\mathbf{s}} \leftarrow [\mathsf{RLWE.Enc}_v(\mathsf{encode}_{p_j}(\mathbf{s}); \mathsf{seed}_{i,j}||11) : j \in [k]]$
  $[A_0, \mathbf{b}_0] \leftarrow \mathsf{LWE.Enc}_{\mathbf{s}}(\mathbf{u})$
  $[A_1, \mathbf{b}_1] \leftarrow \mathsf{LWE.Enc}_{\mathbf{s}}(\mathbf{v})$
  return $(\overrightarrow{\mathsf{ct}_{A_0\mathbf{s}}}, \mathsf{ct}_{\mathbf{s}}, \mathsf{ksk}, [A_0, \mathbf{b}_0], [A_1, \mathbf{b}_1])$
C.recover($\{\overrightarrow{\mathsf{ct}_{\mathsf{rsp},j}}\}_{j\in[k]}, \mathbf{d}$)
  for $j \in [k]$
    $\{\mathsf{ct}_{A_0\mathbf{s}\otimes\mathbf{b}_1}, \mathsf{ct}_{\mathbf{b}_0\otimes A_1\mathbf{s}}, \mathsf{ct}_{A_0\mathbf{s}\otimes A_1\mathbf{s}}\} = \overrightarrow{\mathsf{ct}_{\mathsf{rsp},j}}$
    $\mathbf{m}_j \leftarrow \mathsf{decode}_{p_j}(\mathsf{Dec}(\mathsf{ct}_{A_0\mathbf{s}\otimes\mathbf{b}_1}))$
    $\mathbf{m}'_j \leftarrow \mathsf{decode}_{p_j}(\mathsf{Dec}(\mathsf{ct}_{\mathbf{b}_0\otimes A_1\mathbf{s}}))$
    $\mathbf{m}''_j \leftarrow \mathsf{decode}_{p_j}(\mathsf{Dec}(\mathsf{ct}_{A_0\mathbf{s}\otimes A_1\mathbf{s}}))$
  $\mathbf{m} = \mathsf{iCRT}_P(\mathbf{m}_0, \ldots, \mathbf{m}_{k-1})$
  $\mathbf{m}' = \mathsf{iCRT}_P(\mathbf{m}'_0, \ldots, \mathbf{m}'_{k-1})$
  $\mathbf{m}'' = \mathsf{iCRT}_P(\mathbf{m}''_0, \ldots, \mathbf{m}''_{k-1})$
  $\mathbf{z} = \left\lfloor \frac{\mathbf{d} - \mathbf{m} - \mathbf{m}' + \mathbf{m}''}{\Delta^2} \right\rceil$
  return $\mathbf{z}$
```

Fig. 7. Algorithms of TensorPIR. Note that we slightly modify NTTlessPIR.setup to use the same seed for the rotation key across all invocations.

where c_i is either the ith coordinate of $\mathbf{b}_0$, or $\langle \mathbf{a}_i, \mathbf{s}\rangle$, and $\mathbf{c}'$ is either $\mathbf{b}_1$ or $\mathbf{s}$. The jth coordinate of this is of the form

$$\sum_i c_i \langle \mathbf{a}'_{ij}, \mathbf{c}'\rangle, \tag{13}$$

where $\mathbf{a}'_{ij}$ is the jth row of $\mathsf{DB}_i \cdot A_1$. Note that the server can manually reduce this vector $\bmod Q$ before homomorphically computing, e.g. we may assume a'_{ij} is a vector with entries at most $Q/2$. Similarly, c_i may be assumed to be bounded by $Q/2$ as well. It follows that worst-case, each coordinate of our output plaintext has size at most $d_{\mathbf{u}} N Q^3$, and therefore provided $\prod_i p_i$ is greater than this quantity, CRT interpolation will succeed. □

Practically, this means that we need to support plaintext computations of size up to $\approx m^{1/3} 2^{106}$. Assuming we use $\approx$ 20-bit NTT-friendly prime plaintexts, and that $m \leq 2^{40}$, it suffices to use 6 NTT friendly primes, e.g. 3× as many as we use for $\mathsf{NTTlessPIR}$. This does represent a slight increase in the hidden constant for the server response size, but one that is practically small compared to the asymptotic $\Theta(\sqrt[3]{m})$ query size achievable by Tensor PIR.

We next discuss correctness of $\mathsf{TensorPIR}$ and its efficiency properties. Note that, by combing the recovered decryption terms in Algorithm 7, the result is a vector $\mathbf{d} - \mathbf{m} - \mathbf{m}' + \mathbf{m}'' = \Delta^2 \cdot \mathsf{DB} \cdot (\mathbf{u} \otimes \mathbf{v}) + \mathbf{e}_{\otimes}$, where $\mathbf{e}_{\otimes}$ is the error in the LWE ciphertext $\mathsf{DB} \cdot (C_0 \otimes C_1)$.

Lemma 22 (Correctness of $\mathsf{TensorPIR}$). *Let $N, m, Q \in \mathbb{N}$, and let $\sigma > 0$. Assume $\mathsf{DB} \in \mathbb{Z}_p^m$, where $m = d_{\mathbf{u}} d_{\mathbf{v}} d_{\mathbf{w}}$. Let $C_0 \in \mathbb{Z}_Q^{d_{\mathbf{u}} \times (N+1)}$ and $C_1 \in \mathbb{Z}_Q^{d_{\mathbf{v}} \times (N+1)}$ be LWE encryptions of selection vectors $\mathbf{u} \in \{0,1\}^{d_{\mathbf{u}}}$ and $\mathbf{v} \in \{0,1\}^{d_{\mathbf{v}}}$ with error sub-Gaussian parameter σ. Then, the error in the ciphertext $\mathsf{DB} \cdot (C_0 \otimes C_1)$ is sub-exponential of parameter $(p/2)^2 d_{\mathbf{v}} \sigma^2 + (p/2)^2 d_{\mathbf{u}} \sigma^2 + \sigma^4 \frac{p^4}{4Q^2} d_{\mathbf{u}} d_{\mathbf{v}}$. Furthermore, provided*

$$\frac{Q}{2p} > \ln(1/\delta) \frac{p}{2} \sigma \sqrt{d_{\mathbf{u}} + d_{\mathbf{v}} + \sigma^4 d_{\mathbf{u}} d_{\mathbf{v}} \frac{p^2}{Q^2}},$$

$\mathsf{TensorPIR}$ *is $(1-\delta)$-correct for a single query.*

Lemma 23 (Efficiency of $\mathsf{TensorPIR}$). *Let $\mathsf{DB} \in \mathbb{Z}_p^{d_{\mathbf{u}} \times d_{\mathbf{v}} \times d_{\mathbf{w}}}$, where $m = d_{\mathbf{u}} d_{\mathbf{v}} d_{\mathbf{w}}$. Then* $\mathsf{TensorPIR}$ *requires*

- *Server Preprocessing: $O(k \ell n N \log n)$ operations in $\mathbb{Z}_q$ and $2mN$ operations in $\mathbb{Z}_Q$,*
- *Server Long-term Storage: $knN(\ell+1)$ elements of $\mathbb{Z}_q$ and $d_{\mathbf{u}} d_{\mathbf{w}} N$ elements of $\mathbb{Z}_Q$,*
- *Server Response Time: $kN(d_{\mathbf{w}} d_{\mathbf{v}} + n + (\ell+2)n)$ $\mathbb{Z}_q$ operations, and m $\mathbb{Z}_Q$ operations,*
- *Client Upload: $(k+\ell)n$ elements of $\mathbb{Z}_q$ and $d_{\mathbf{u}} + d_{\mathbf{v}}$ elements of $\mathbb{Z}_Q$,*
- *Client Download: $2k(d_{\mathbf{w}} + n)$ elements of $\mathbb{Z}_q$ and $d_{\mathbf{w}}$ elements of $\mathbb{Z}_Q$*

H More Details on Evaluation

We discuss a bit more about our experiments on HintlessPIR and comparing with Tiptoe PIR, SimplePIR and Spiral.

Comparing with Tiptoe PIR. Both schemes are "hintless", so we compare solely their performance. Here, we find that HintlessPIR is more performant on most metrics of interest, with throughout between 1× and $\approx 3\times$ higher than Tiptoe PIR, and bandwidth between one to two orders of magnitude smaller. Tiptoe PIR's has a small advantage when it comes to server preprocessing time, where it is between 1.005× to 3.2× faster, though this shrinks to 1.005× to 1.11× faster when restricting to databases where Tiptoe PIR has lower bandwidth (per-query) than transmitting the entire database.

Comparing with SimplePIR. When executing in a single thread, the online throughput of our SimplePIR implementation is very close to 11GB/s/core, which is also close to the memory I/O throughput. Despite the extremely fast online processing speed, SimplePIR requires the client to download a database-dependent hint. For the database dimensions we benchmarked (which are typical for PIR applications) the hint size is at least 1/6 of the entire database (for the first three dimensions) or larger than 180MB, and it may require several seconds to even download the hint. So, in the anonymous PIR setting, or when the client makes only a small number of PIR queries in between database updates, our HintlessPIR protocol requires much less communication at the cost of slightly increased server computation. We also note that, for large databases, the cost of homomorphically generating rotations of s is less significant, and the total server computation cost of HintlessPIR becomes close to that of SimplePIR. For example, for a database of dimension $2^{18} \times 32$KB and total size 8.59GB, the latency of HintlessPIR is about 1.3s while the latency of SimplePIR is 0.8s. For the offline phase, the overhead due to NTTlessPIR preprocessing becomes cheaper than SimplePIR preprocessing for databases larger than 60MB. For example, for database dimension $2^{20} \times 256$ bytes, preprocessing in NTTlessPIR takes 6.18s while computing the hint matrix in SimplePIR takes 45s.

Comparing with Spiral. The advantage of Spiral over SimplePIR is usually the smaller offline communication cost, which almost completely vanishes when comparing to HintlessPIR. In terms of the online communication cost, Spiral is more efficient than HintlessPIR, as its query can be close to $O(\log m)$ and its response size can be close to the size of a single record for typical database dimensions. However, our protocol requires less total communication bandwidth in the anonymous PIR setting. More importantly, the online latency of our protocol is significantly smaller than Spiral. For example, for small databases such as 2^{20} records of 256 bytes each, our protocol runs in 385ms while Spiral runs in 694ms. For a database of 1GB large, with 2^{30} records, the throughput of our protocol is about 1.75GB/s while Spiral only achieves 463MB/s. Note that our implementation does not currently leverage specially-structured NTT friendly primes (e.g. of the form $q = 2^i - 2^j + 1$) that admit concretely faster arithmetic operations, as done in Spiral. We did not benchmark variants of Spiral that optimize for large records, as they require larger offline or online communication.