Communication–Computation Trade-offs in PIR

Asra Ali
Google
asraa@google.com

Tancrède Lepoint
Google
tancrede@google.com

Sarvar Patel
Google
sarvar@google.com

Mariana Raykova
Google
marianar@google.com

Phillipp Schoppmann
Humboldt-Universität zu Berlin
schoppmann@informatik.hu-berlin.de

Karn Seth
Google
karn@google.com

Kevin Yeo
Google
kwlyeo@google.com

Abstract

We study the computation and communication costs and their possible trade-offs in various constructions for private information retrieval (PIR), including schemes based on homomorphic encryption and the Gentry–Ramzan PIR (ICALP'05).

We improve over the construction of SealPIR (S&P'18) using compression techniques and a new oblivious expansion, which reduce the communication bandwidth by 60% while preserving essentially the same computation cost. We then present MulPIR, a PIR protocol leveraging multiplicative homomorphism to implement the recursion steps in PIR. This eliminates the exponential dependence of PIR communication on the recursion depth due to the ciphertext expansion, at the cost of an increased computational cost for the server. Additionally, MulPIR outputs a regular homomorphic encryption ciphertext, which can be homomorphically post-processed. As a side result, we describe how to do conjunctive and disjunctive PIR queries.

On the other end of the communication–computation spectrum, we take a closer look at Gentry–Ramzan PIR, a scheme with asymptotically optimal communication rate. Here, the bottleneck is the server's computation, which we manage to reduce significantly. Our optimizations enable a tunable trade-off between communication and computation, which allows us to reduce server computation by as much as 85%, at the cost of an increased query size. We further show how to efficiently construct PIR for sparse databases. Our constructions support batched queries, as well as symmetric PIR.

We implement all of our PIR constructions, and compare their communication and computation overheads with respect to each other for several application scenarios.

1 Introduction

Accessing public databases often brings privacy concerns for the querier as the query may already reveal sensitive information. For example, queries of medical data can reveal sensitive health information, and access patterns of financial data may leak investment strategies. In settings where such privacy leakage has significant risk, clients may shy away from accessing the database. On the flip side, data providers often do not want access to sensitive client queries, as they could later become a liability for them.

Private information retrieval (PIR) is a cryptographic primitive that aims to address the above question by enabling clients to query a database without revealing any information about their queries to the data owner. While the feasibility of this primitive has been resolved for a long time [15], the search for concretely efficient constructions for practical applications has been an active area of research [4, 5, 20, 24, 25, 30, 36, 47, 66]. In this context, there are several parameters and efficiency measures that characterize a PIR setting and determine what solution might be most suitable for a particular scenario. However, a baseline solution that candidate PIR solutions should improve on is the trivial PIR that returns the whole database to the client.

In some scenarios private queries are only a building block in more complex functionalities that restrict the information which the client can learn about the database and allow her to obtain only the evaluation of a fixed function on the database content in the locations of her queries. For example, these could be conjunctive or disjunctive queries over the content of multiple PIR queries. PIR has also been used to compute communication efficiently joins over set with asymmetric size. This functionality has been used in the context of private set intersection protocols [13], but it can also be used as a first step in *private join and compute* protocols [59] which enable restricted computation over the join of two private databases.

In this work, we take a deep dive into the setting of PIR where data is stored on a single server. This is the relevant PIR model in practical settings where no additional party is available to assist with the data storage and query execution and one does not wish to trust secure hardware. Non-trivial single server PIR constructions are known to require computational assumptions [43], and such solutions bring significant overheads for both the communication and computation costs compared to information theoretic constructions that are possi-

ble in the multi-server setting [22]. While theoretical constructions for PIR [43] achieve poly-logarithmic communication, most efficient single server PIR implementations stop short of this goal and implement only variants of the construction with higher asymptotic communication costs [4, 5, 36, 47].

We analyze the communication–computation trade-offs that different PIR construction approaches offer and the hurdles towards achieving the optimal asymptotic communication costs in practice. We present a new PIR construction using somewhat-homomorphic encryption, which leverages multiplicative homomorphism as opposed to layers of additive HE, and improves the communication and computation costs of recursion in existing PIR schemes, enabling for the first time measurements with recursion level beyond three. This construction supports natively extended computation over the results of many PIR queries. As an alternative to HE-based PIR, we consider the Gentry-Ramzan PIR construction, which achieves optimal communication but has a high computation overhead. Here, we propose a new client-aided model of computation that allows for a tunable trade-off between communication and computation costs.

Our constructions support the asymmetric variant of PIR as well as multi-queries using probabilistic batch codes (PBC). We also provide a new construction for sparse databases, a.k.a, keyword PIR [14], where the number of database entries is much smaller than the query key domain, and the server's cost depends only on the actual database size as opposed to the key domain size. We implement our new PIR constructions and compare the communication/computation trade-offs they offer against existing constructions instantiated with different HE schemes. We evaluate the PIR schemes using different database shapes motivated by three applications.

For space constraints, we present related work in Appendix A.

1.1 Our Contributions

In this paper we present a portfolio of PIR constructions as well as extended functionality. They provide various *practical* tradeoffs between computation and communication, which we experimentally evaluate using three application scenarios.

Improving SealPIR communication. The most efficient (secure) single server PIR constructions implemented in recent years [4,5,20,24,25,30,36,47,66] are based on homomorphic encryption (HE) techniques and achieve sub-linear communication. Among those the scheme that currently provides best implementation performance is SealPIR [3, 4]. While theoretically this construction supports sublinear communication complexity $O(d \cdot n^{1/d})$ leveraging d recursion levels, in practice it has not been instantiated with the recursion depth $d > 2$. The reason for this is that this construction uses a layered additive homomorphic encryption approach, where if the encryption scheme has ciphertext expansion F, the PIR response will include F^{d-1} ciphertexts (where $F = 10$ in [3]). This yields an PIR expansion of $O(F^2)$, which becomes unacceptable when the elements become large. Our first contribution reduces the communication of SealPIR by (1) using symmetric key encryption to reduce the upload size, (2) using modulus switching to reduce the value of F down to $F \approx 4$, and (3) introducing a new oblivious expansion algorithm which can further halve the upload communication for some parameter sets. Our optimized SealPIR reduces the upload by 300% and the download by 400% for large database entries.

MulPIR: Leveraging Multiplicative Homomorphism. As SealPIR only uses the fact that FV is additively homomorphic, this has two undesired consequences. First, the download communication still depends exponentially on the recursion level (the previous contribution reduced the basis of the exponential). Additionally, when recursion is used, the PIR result is an encryption of a ciphertext parsed as plaintexts, and therefore cannot be operated on homomorphically. This prevents the server to applying any post-processing on the PIR result. We propose a different approach (MulPIR) that uses both additive and the *multiplicative* homomorphisms of HE to implement the recursive selection by doing one multiplication of encrypted values per recursion step. This reduces the size of the upload and download together from $O(\lceil d \cdot n^{1/d}/N \rceil + F^{d-1})$ from the previous approach, where F is the number of plaintexts needed to fit a single HE ciphertext, to $\lceil d \cdot n^{1/d}/N \rceil \cdot c(d)$, where $c(d)$ is the size of a ciphertext that supports d multiplications (hence, a depth of $\log d$). Our new techniques also allow us to achieve the ideal asymptotic communication complexities for PIR in practice and enable for the first time implementation experiments with recursion level beyond three. For databases with large entries, we show that MulPIR can further reduce the download by 200% compared to the optimized SealPIR construction. We also show how we to use MulPIR to achieve extended functionalities that enable post-processing computation on the results of the PIR queries on the server side before returning the results to the client. In particular, we consider conjunction and disjunctions queries over binary databases.

Gentry–Ramzan PIR: New Efficiency Trade-offs. The Gentry–Ramzan PIR construction [32] achieves optimal communication complexity for several settings but it pays with significant computational cost. Thus, our contributions focus on ways to reduce this computation overhead, which includes new efficient techniques for encoding the server's database in CRT form needed for the computation in the scheme, new techniques for fast modular exponentiation needed to answer each query, as well as techniques for client-aided PIR that trade-off between communication and computation.

In this PIR protocol, the server database $\{\mathsf{D}_i\}_{i\in[n]}$ needs to be encoded as $x = \mathsf{D}_i \bmod \pi_i$ for $i \in [n]$, where π_i are pair-

wise coprime integers. A naive application of the Chinese Remainder Theorem requires computation at least quadratic in the size of the database. We leverage a divide-and-conquer modular interpolation algorithm [8] that enables us to achieve computation complexity $\tilde{O}(n\log^2 n)$. This technique also allows for pre-computation that can be reused for computations that use the same set of moduli π_i.

The main computation cost on the server side is the modular exponentiation, where the server cannot know the prime factorization of the modulus and thus we cannot use techniques that leverage the factorization to speed-up the computation of the exponentiation. Our approach is to compute the exponentiation as a product of precomputed powers of the generator and to use Straus's algorithm [64] to do this efficiently. This enables a client-aided technique that allows to improve the server's computation at the price of additional work at the client. In particular since the precomputed powers of the generator are independent of the exponent, they can be computed at the client who knows the order of the group that it is using for the PIR query and thus can compute exponentiation in this group faster by first reducing the exponent modulo the order of the group. This gives a new way to trade-off computation and communication complexity for the protocol. In Section 6, we show evidence that providing several precomputed powers optimizes the server's work.

We also apply batching techniques leveraging probabilistic batch codes from Cuckoo hashing [49] for a multi-query setting of Gentry-Ramzan PIR, which provide better scalability for broader sets of the database parameters compared to previous batching approaches [37].

New Construction for Sparse PIR. We present a new PIR construction for sparse databases, which provides the client with an answer that either contains the corresponding data if the element is present in the database or is empty, otherwise (see Appendix E). Our construction leverages Cuckoo hashing [49] in a new way inspired by ideas from private set intersection [12, 23, 55, 57] and oblivious RAM [35, 52, 54]. In particular, we observe that we can compress the domain of the database from a large sparse domain to a small dense domain using Cuckoo hashing, which in comparison to regular hashing distributes the items in the hash tables guaranteeing that no collisions occur.

Comparison and Empirical Evaluation of PIR. We present a comprehensive comparison of the costs of PIR based on homomorphic encryption. This includes detailed concrete efficiency estimates for the ciphertext size and the computation costs for encryption, decryption and homomorphic operations of different HE schemes. We leverage these estimates to profile the efficiency costs of PIR constructions using the corresponding schemes when instantiated with and without recursion. We further present empirical evaluations of implementations of these PIRs with databases of different shapes (numbers of records and entry sizes). Our benchmarks demonstrate that for the majority of the settings constructions based on lattice based HE constructions, which could also offer multiplicative homomorphism, outperform in computation other additive HE schemes. In terms of communication, additive HE solutions have advantage when the dominant communication cost is the download, e.g., in solutions without recursion for small databases with large entries, since these encryption provides best ratio between plaintext and ciphertext.

We evaluate our new PIR construction, MulPIR, that uses somewhat-homomorphic encryption (SHE) and compare it against SealPIR. MulPIR enables a trade-off of computation for communication, which reduces the communication of SealPIR by 80% while increasing the computation roughly twice. We also provide the first empirical evaluation of PIR with recursive level beyond three (see Appendix D). Surprisingly, we observe that higher recursion level does not necessarily improve communication. This is due to the fact that lattice-based HE encryptions have a complex relationship between parameters sizes, support for homomorphic operations and number of encryption slots. While recursion improves complexity when the database size increases beyond the number of encryption slots in a ciphertext, increasing the database size requires support for more homomorphic operations, which leads to larger parameters and more slots. In our experiments, Gentry–Ramzan PIR always achieves the best communication complexity but comes with a significant computation cost that can be prohibitive in some settings. However, we show that in terms of *monetary* cost, Gentry–Ramzan can outperform all other PIR approaches considered when database elements are small.

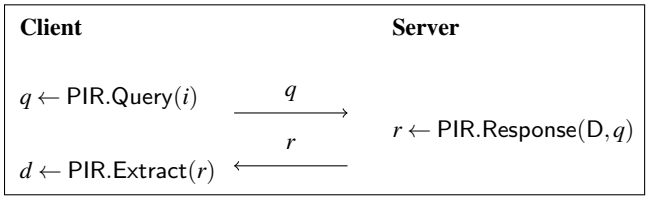
Finally, we apply our construction for keyword PIR to a *password checkup* problem, where a client aims to check if their password is contained in a dataset of leaked passwords, without revealing it to the server. Previous approaches to this problem [65] first reveal a k-anonymous identifier to the server to reduce the number of candidate passwords to compare against to k, and then apply a variant of Private Set Intersection to compare the current password against the k candidates. Our implementations of Gentry–Ramzan and MulPIR enable such lookups with communication *sublinear* in k, therefore either enabling better anonymity for the same bandwidth, or same anonymity and smaller bandwidth.

2 Preliminaries

Throughout the rest of this paper, we assume a server owns a database $\mathsf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ of n elements of size l bits.

For any $m \in \mathbb{Z}$, $m \geq 1$, we denote by $[m]$ the interval $[1, m]$. We denote by $\delta_{i,j}$ the Kronecker delta function, defined as $\delta_{i,j} = 0$ if $i \neq j$, and $\delta_{j,j} = 1$. For two party computation protocols we will use the notation $[\![a, b]\!]$ to denote either inputs or outputs for the two parties, i.e., a is either an input

Figure 1: A non-interactive PIR protocol. Correctness of the protocol will ensure that $d = \mathsf{D}_i$.



or output for the first party, and similarly b is either input or output for the second party.

2.1 Private Information Retrieval (PIR)

Definition 2.1 (Private Information Retrieval [15])**.** A *private information retrieval* protocol addresses the setting where a server holds a database $\mathsf{D} = \{\mathsf{D}_1, \ldots, \mathsf{D}_n\}$ of n elements, and a client has an input index i. The goal of the protocol is to enable the client to learn D_i while guaranteeing that the server does not learn anything about i. A PIR scheme is specified with the following two algorithms:

- $q \leftarrow \mathsf{PIR.Query}(i)$ – this is an algorithm that the client runs on its input index i to generate a corresponding query.
- $[\![\mathsf{D}_i, \perp]\!] \leftarrow \mathsf{PIR.Eval}([\![q, \mathsf{D}]\!])$ – this is a two-party computation protocol with inputs the client's encoded query and the server's database that outputs the corresponding database items to the client. Most PIR constructions are non-interactive and we can replace the evaluation protocol with the following two algorithms (cf. Fig. 1).
  - $r \leftarrow \mathsf{PIR.Response}(\mathsf{D}, q)$ – an algorithm that the server runs on the client's encoded query to compute an encoded response.
  - $\mathsf{D}_i \leftarrow \mathsf{PIR.Extract}(r)$ – an algorithm that the client runs on the server's response to extract the output for the queried item.

Definition 2.2 (Symmetric Private Information Retrieval (SPIR))**.** Symmetric PIR extends the PIR functionality with privacy requirement also for the database guaranteeing the client does not learn anything beyond the element D_i.

2.2 Homomorphic Encryption

For ease of notation and without loss of generality, recall that an homomorphic encryption (HE) scheme $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with plaintext space $\mathbb{Z}_t$ is an encryption scheme with the following properties:

1. $\mathsf{Enc}(\mathsf{sk}, m_1) + \mathsf{Enc}(\mathsf{sk}, m_2) = \mathsf{Enc}(\mathsf{sk}, (m_1 + m_2) \bmod t)$,
2. $\mathsf{Enc}(\mathsf{sk}, m_1) \times \mathsf{Enc}(\mathsf{sk}, m_2) = \mathsf{Enc}(\mathsf{sk}, (m_1 \times m_2) \bmod t)$,
3. $\mathsf{Enc}(\mathsf{sk}, m_1) \cdot \lambda = \mathsf{Enc}(\mathsf{sk}, m_1 \cdot \lambda \bmod t)$,

for every $m_1, m_2, \lambda \in \mathbb{Z}_t$, for some specific operations $+$, $\times$, and $\cdot$ over the ciphertexts. An HE scheme that does not verify item 2 is called an *additive HE scheme*.

Below, we recall the Fan–Vercauteren (FV) homomorphic encryption scheme [26]. For space constraints, El Gamal and Paillier/Damgård–Jurik are recalled in Appendix B.

Fan–Vercauteren. An FV ciphertext is a pair of polynomials over R/qR, where $R = \mathbb{Z}[x]/(x^N + 1)$, and encrypts a message $m(x) \in R/tR$ for a $t < q$. In addition to the standard operations of an encryption scheme (key generation, encryption, decryption), FV also supports homomorphic operations: addition, scalar multiplication, and multiplication.

- **Addition:** Given two ciphertexts c_1 and c_2, respectively encrypting $m_1(x)$ and $m_2(x)$, the homomorphic addition of c_1 and c_2, denoted $c_1 + c_2$, results in a ciphertext that encrypts the sum $m_1(x) + m_2(x) \in R/tR$.
- **Scalar multiplication:** Given a ciphertext $c \in (R/qR)^2$ encrypting $m(x) \in R/tR$, and given $m'(x) \in R/tR$, the scalar multiplication of c by $m'(x)$, denoted $m'(x) \cdot c$, results in a ciphertext that encrypts $m'(x) \cdot m(x) \in R/tR$.
- **Multiplication:** Given two ciphertexts c_1 and c_2, respectively encrypting $m_1(x)$ and $m_2(x)$, the homomorphic multiplication of c_1 and c_2, denoted $c_1 \cdot c_2$, results in a ciphertext that encrypts the product $m_1(x) \cdot m_2(x) \in R/tR$.

Finally, [4] introduced a specific operation called substitution, instantiated using the plaintext slot permutation of [31].

- **Substitution:** Given a ciphertext $c \in (R/qR)^2$, that encrypts $m(x) \in R/tR$, and an integer k, the substitution operation $\mathsf{Sub}_k(\cdot)$ applied on c results in a ciphertext that encrypts $m(x^k) \in R/tR$.

2.3 PIR Based on Additive HE

The majority of PIR constructions that achieve sub-linear communication rely on homomorphic encryption and enable the client to compress its query. More precisely, there are two flavors of HE-based PIR protocols with sub-linear communication that exist in the literature, those based on additive homomorphic encryption (AHE) schemes and those based on fully homomorphic encryption (FHE) schemes.

In this section, we focus on the former flavor, that captures schemes based on El Gamal, Paillier/Damgård–Jurik, and captures the SealPIR protocol proposed by Angel et al. [4] (based on lattice-based additive homomorphic encryption).

Baseline PIR. We recall the baseline solution for PIR based on homomorphic encryption [43]. Let l denote the bit-size of the elements of the database and let $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a homomorphic encryption scheme with plaintext space $\mathbb{Z}_t$ for $t \geq 2^l$. Denote by $\mathcal{C}$ the ciphertext space of $\mathcal{HE}$. Note that we will interpret each element D_i as an element of $\mathbb{Z}_t$.

Procedure 1 PIR.HE.Query

Input: $k \in [1,n]$.
$\vec{s} = (s_i)_{i=1\dots n} = (\delta_{i,k})_{i=1\dots n}$.
$\forall i \in [1,n], q_i \leftarrow \mathsf{Enc}(\mathsf{sk}, s_i)$.
Output: $\vec{q} = (q_i)_{i=1\dots n} \in \mathcal{C}^n$.

Procedure 2 PIR.HE.Response

Input: $\mathsf{D} \in \mathbb{Z}_t^n, \vec{q} \in \mathcal{C}^n$.
$r = \langle \vec{q}, \mathsf{D} \rangle = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}, \mathsf{D} \rangle\Big)$ as in Eq. (1).
Output: $r \in \mathcal{C}$.

The baseline PIR protocol works as follows (cf. Algorithms 1 to 3). To construct the query for index k, the client encrypts component by component the *selection vector* $\vec{s} = (s_i)_{i=1\dots n}$ proportional to the size of the database n, which verifies $s_i = \delta_{i,k} = 0$ for $i \neq k$ and $s_k = \delta_{k,k} = 1$. To answer the query $q = (\mathsf{Enc}(\mathsf{sk}, s_i))_{i=1\dots n}$, the server computes the inner product between the query and the database D (where $\mathsf{D}_i \in \mathbb{Z}_t$), eventually yielding

$$\begin{aligned}\langle q, \mathsf{D} \rangle &= \sum_{i=1}^{n} \mathsf{Enc}(\mathsf{sk}, s_i) \cdot \mathsf{D}_i \\ &= \mathsf{Enc}\Big(\mathsf{sk}, \sum_{i=1}^{n} \delta_{i,k} \mathsf{D}_i\Big) = \mathsf{Enc}(\mathsf{sk}, \mathsf{D}_k). \end{aligned} \tag{1}$$

In the rest of the paper, we will instantiate this protocol with the Paillier/Damgård–Jurik cryptosystem [21, 50], the El-Gamal cryptosystem [29], the FV cryptosystem [4, 26, 47]. In Appendix B and Table 7, we report on the specific communication and computation costs of these schemes.

We note that this baseline PIR has communication linear in the database size n. Two approaches have been proposed in the literature to reduce the overall communication cost: either using recursion (also called folding [30]) using additive homomorphic encryption, or using fully homomorphic encryption. We survey these two approaches below.

Recursion/Folding. Kushilevitz, Ostrovsky [43], and later Stern [63], propose the following modification of Algorithms 1 to 3. Instead of representing the database D as a vector of size n, one can represent D as a $n^{1/2} \times n^{1/2}$ matrix $M = (M_{i,j})$, where $M_{i,j} \coloneqq \mathsf{D}_{in^{1/2}+j}$. Now, instead of sending (the encryption of) one selection vector $\vec{s} = (\delta_{i,k})$ of dimension n for index k, the client writes $k = i'n^{1/2} + j'$ where $i', j' \in [n^{1/2}]$, and sends two binary selection vectors $\vec{s}_1 = (s_{1,i}) = (\delta_{i,i'})$ and $\vec{s}_2 = (s_{2,j}) = (\delta_{j,j'})$ of dimension $n^{1/2}$. In particular, it holds that $s_{1,i} \cdot s_{2,j} = 1$ if and only if $i = i'$ and $j = j'$.
The server then performs three steps:

1. For each of the $n^{1/2}$ rows $M_i = (M_{i,1} \cdots M_{i,n^{1/2}})$, the server computes the response with the (encryption of the) selection vector $\vec{s}_2$ as in Eq. (1), i.e., the server obtains the $n^{1/2}$ ciphertexts
$$c_i = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}_2, (M_{i,j})_j \rangle\Big) = \mathsf{Enc}\Big(\mathsf{sk}, \mathsf{D}_{in^{1/2}+j'}\Big).$$
2. Since the ciphertext expansion is $F > 1$, for each $i \in [n^{1/2}]$, the server represents [1] c_i as F plaintext elements $c_{i,1}, \dots, c_{i,F}$.
3. For each of the vectors $(c_{1,f} \cdots c_{n^{1/2},f})$ with $f \in [F]$, the server computes the response with the (encryption of the) selection vector $\vec{s}_1$ as in Eq. (1), i.e., the server obtains the F ciphertexts
$$c'_f = \mathsf{Enc}\Big(\mathsf{sk}, \langle \vec{s}_1, (c_{i,f})_i \rangle\Big) = \mathsf{Enc}\Big(\mathsf{sk}, c_{i',f}\Big).$$

Upon reception of the response, $r = (c'_1, \dots, c'_F) \in \mathcal{C}^F$, the client finally extracts the desired result as follows.

1. It uses the homomorphic encryption decryption key to recover $c_{i',f}$ for all $f \in [F]$.
2. It reconstructs $c_{i'}$ from the $c_{i',f}$'s elements.
3. It uses the homomorphic encryption decryption key on $c_{i'}$ to recover $\mathsf{D}_{i'n^{1/2}+j'} = \mathsf{D}_k$.

This method easily generalizes by representing the database as a d-dimensional hyperrectangle $[n_1] \times \cdots \times [n_d]$ with $n = n_1 \cdot n_2 \cdots n_d$ (the baseline PIR corresponds to $d = 1$ with $n_1 = n$, and the recursion above to $d = 2$ with $n_1 = n_2 = n^{1/2}$).

Procedure 3 PIR.HE.Extract

Input: $r \in \mathcal{C}$.
$d \coloneqq \mathsf{Dec}(\mathsf{sk}, r) \bmod 2^l$.
Output: $d \in \mathbb{Z}_{2^l}$.

Cost of recursion. When $n_i = n^{1/d}$, we accomplish the following communication complexity: $O\big(c(n) \cdot dn^{1/d}\big)$ for the user's query and $O\big(F^{d-1}c(n)\big)$ for the server's response, where $c(n)$ is the size of the ciphertext. In particular, for small values of d, we will get sub-linear communication. However, note that for full recursion, i.e., $d = \log n$, communication becomes polynomial in n.

3 Improving SealPIR

The SealPIR protocol was proposed by Angel et al. [4], and improves over the XPIR protocol proposed by Aguilar Melchor et al. [47]. Both SealPIR and XPIR instantiate the recursive PIR described above, using the FV homomorphic encryption scheme viewed as an *additive* HE scheme.

[1] We assume without loss of generality that $F \in \mathbb{Z}$. Note that we do not ask for any algebraic conditions from the map; for example we could just break down a binary representation of elements of $\mathcal{C}$ into F plaintexts. For the Paillier cryptosystem, or more precisely the generalization from Damgård and Jurik [21], we will take a different approach: we will select parameters so that the ciphertext after the first folding exactly fits in the plaintext space for the second folding; cf. Appendix B.

Procedure 4 SealPIR.Query

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Index $k \in [1, n]$
2: Generate $\vec{s}_j = (s_{j,i})_{i \in [m]}$ the d selections vectors in $\{0,1\}^m$.
3: $\ell \leftarrow \lceil m/2^c \rceil$
4: $\forall j \in [d]$, parse $\vec{s}_j$ as $\vec{s}_{j,1}, \ldots, \vec{s}_{j,\ell}$ vectors in $\{0,1\}^{2^c}$
5: $\forall j \in [d], \forall j' \in [\ell], m_{j,j'} \leftarrow \sum_{i \in [2^c]} \vec{s}_{j,j'}[i] \cdot x^i \in R/tR$
6: $\forall j \in [d], \forall j' \in [\ell], q_{j,j'} \leftarrow \mathsf{Enc}(\mathsf{sk}, m_{j,j'})$.
Output: $\vec{q} = (q_{j,j'})_{j \in [d], j' \in [\ell]} \in \mathcal{C}^{d \cdot \ell}$.

Procedure 5 SealPIR Oblivious Expansion

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Ciphertexts $(q_{j,j'} = (c_{0,j,j'}, c_{1,j,j'}))_{j \in [d], j' \in [\lceil m/2^c \rceil]}$
2: $\ell \leftarrow \lceil m/2^c \rceil$
3: $\mathsf{ciphertexts} \leftarrow []$
4: **for** $j = 1$ to d **do**
5: $\quad \mathsf{ciphertexts}_j \leftarrow []$
6: $\quad$ **for** $j' = 1$ to ℓ **do**
7: $\quad\quad \mathsf{ctxts} = [q_{j,j'} = (c_0, c_1)]$ // start the expansion of $q_{j,j'}$
8: $\quad\quad$ **for** $a = 0$ to $c - 1$ **do**
9: $\quad\quad\quad$ **for** $b = 0$ to $2^a - 1$ **do**
10: $\quad\quad\quad\quad c_0 \leftarrow \mathsf{ctxts}[b]$
11: $\quad\quad\quad\quad c_1 \leftarrow x^{-2^a} \cdot c_0$ // scalar multiplication
12: $\quad\quad\quad\quad c'_b \leftarrow c_0 + \mathsf{Sub}_{2^{c-a}+1}(c_0)$
13: $\quad\quad\quad\quad c'_{b+2^a} \leftarrow c_1 + \mathsf{Sub}_{2^{c-a}+1}(c_1)$
14: $\quad\quad\quad$ **end for**
15: $\quad\quad\quad \mathsf{ctxts} = [c'_0, \ldots, c'_{2^{a+1}-1}]$
16: $\quad\quad$ **end for**
17: $\quad\quad \mathsf{ciphertexts}_j \leftarrow \mathsf{ciphertexts}_j \| \mathsf{ctxts}$
18: $\quad$ **end for**
19: $\quad \mathsf{ciphertexts} \leftarrow \mathsf{ciphertexts} \| \mathsf{ciphertexts}_j[0..m-1]$
20: **end for**
21: **for** $j = 0$ to $m - 1$ **do**
22: $\quad o_j \leftarrow (2^{-c} \bmod t) \cdot \mathsf{ciphertexts}[j]$ // normalization
23: **end for**
Output: $\mathsf{output} = [o_0, \ldots, o_{m-1}]$

SealPIR [4] improves over XPIR by encrypting many bits in a single ciphertext (one per polynomial coefficient) and shows how the server can *obliviously expand* such a ciphertext to obtain encryptions of each of the bits separately. SealPIR's Query algorithm is given in Algorithm 4 and enables to decrease the upload cost *up to* a factor N (the polynomial ring dimension).[2]

Now, when the server receives such a compressed query, it needs to perform an oblivious expansion into the original query, to then apply Response (Algorithm 2). SealPIR's oblivious expansion is recalled in Algorithm 5. We note that [4] only described the inner loop and normalization (Lines 8–16 and 21–23), but we provide here the algorithm in full for better comparison with our new algorithm (Algorithm 7).

[2] Such a compression factor can be obtained for example when the compression $c = \log_2 N$, and $m = n^{1/2} = N$, then $\ell = 1$ and the query consists of $d = 2$ ciphertexts instead of $2m = 2N$ ciphertexts.

Procedure 6 New Query

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Index $k \in [1, n]$
2: Generate $\vec{s}_j = (s_{j,i})_{i \in [m]}$ the d selections vectors in $\{0,1\}^m$.
3: $\ell \leftarrow \lceil d \cdot m/2^c \rceil$
4: Parse $(\vec{s}_1, \ldots, \vec{s}_d)$ as $(\vec{s}'_1, \ldots, \vec{s}'_\ell)$ vectors in $\{0,1\}^{2^c}$
5: $\forall j \in [\ell], m_j \leftarrow \sum_{i \in [2^c]} (2^{-c} \bmod t) \cdot \vec{s}_j[i] \cdot x^i \in R/tR$
6: $\forall j \in [\ell], q_j \leftarrow \mathsf{Enc}(\mathsf{sk}, m_j)$.
Output: $\vec{q} = (q_j)_{j \in [\ell]} \in \mathcal{C}^\ell$.

Procedure 7 New Oblivious Expansion

Parameters: $d \in [1, \log n]$, $m = n^{1/d}$, compression $c \in [0, \log_2 N]$
Input: Ciphertexts $(q_j = (c_{0,j}, c_{1,j}))_{j \in [\lceil d \cdot m/2^c \rceil]}$
2: $\ell \leftarrow \lceil d \cdot m/2^c \rceil$
3: $\mathsf{ciphertexts} \leftarrow []$
4: **for** $j = 1$ to ℓ **do**
5: $\quad \mathsf{ctxts} = [q_j = (c_0, c_1)]$ // start the expansion of q_j
6: $\quad$ **for** $a = 0$ to $c - 1$ **do**
7: $\quad\quad$ **for** $b = 0$ to $2^a - 1$ **do**
8: $\quad\quad\quad c_0 \leftarrow \mathsf{ctxts}[b]$
9: $\quad\quad\quad c_1 \leftarrow x^{-2^a} \cdot c_0$ // scalar multiplication
10: $\quad\quad\quad c'_k \leftarrow c_0 + \mathsf{Sub}_{2^{c-a}+1}(c_0)$
11: $\quad\quad\quad c'_{k+2^a} \leftarrow c_1 + \mathsf{Sub}_{2^{c-a}+1}(c_1)$
12: $\quad\quad$ **end for**
13: $\quad\quad \mathsf{ctxts} = [c'_0, \ldots, c'_{2^{a+1}-1}]$
14: $\quad$ **end for**
15: $\quad \mathsf{ciphertexts} \leftarrow \mathsf{ciphertexts} \| \mathsf{ctxts}$
16: **end for**
Output: $\mathsf{output} = [o_0, \ldots, o_{m-1}]$

3.1 Halving SealPIR Communication

This section proposes methods to halve "for free" (with minor computational cost) the communication of SealPIR [4].

Compressing the upload. We remark that the client, who creates the query ciphertexts, knows the secret key of the homomorphic encryption scheme. Henceforth, instead of using the public key encryption algorithm as in SealPIR, the client can use the secret key encryption algorithm of FV, i.e., encrypting with the secret key. Recall that a FV ciphertext is a tuple (c_0, c_1) in R/qR. A key observation is that when using secret key encryption, the first element c_0 is sampled uniformly at random in R/qR, whereas it depends on the public key when using public key encryption. Therefore, instead of sending c_0, the client can instead send a seed $\rho \in \{0,1\}^\lambda$, and the server can reconstruct c_0 from the seed locally. *This reduces the upload by a factor 2x.*

Compressing the download. At the end of the server computation, the ciphertext will no longer be processed and will only be decrypted by the client. Henceforth, we propose to use *modulus switching* to compress its size as much as possible. This operation allows to transform a ciphertext

$(c_0, c_1) \in (R/qR)^2$ with a noise of norm $\approx E$ into a ciphertext $(c_0, c_1) \in (R/pR)^2$ with a noise of norm $\approx \min(t, (p/q) \cdot E)$ where t is the plaintext space [17]. It therefore enables us to reduce the download communication in PIR as follows. After finishing to compute the response $\vec{r} = (r_i)_{i=1\dots\ell}$ (Algorithm 2), the server will use modulus switching on each ciphertext $r_i \in (R/qR)^2$ to create a new ciphertext $r'_i \in (R/pR)^2$, where $p \geq t^2$ is chosen large enough to ensure decryption. In practice, this reduces the download size by $\approx \log_2 q/(2\log t)$; using SealPIR parameters and using modulus switching to a prime $p \approx 2^{25}$, *this techniques enables to reduce the download by a factor* $60/25 = 2.4x$.

Remark 1. We note that, when recursion is used, one can further reduce the communication requirement *at the cost of increasing the computation cost*. Recall that in Step 2 of the recursion, for each $i \in [n^{1/2}]$, the server represents c_i as F plaintext elements $c_{i,1}, \dots, c_{i,F}$, where F is the ciphertext expansion. If the server uses modulus switching on all the c_i's (i.e., perform $n^{1/2}$ modulus switching) before parsing them as $c_{i,j}$'s, their sizes will be smaller by a factor $\approx \log_2 p/\log_2 q$.

3.2 New Oblivious Expansion

We now describe optimized versions of the Query and oblivious expansion algorithms in Algorithms 6 and 7, which enable to reduce the upload computation up to a factor d (the recursion level) compared to Algorithms 6 and 7 (differences are highlighted in blue). For example, when $d = 2, N = 2048$ and $n = 2^{20}$ (a parameter setting from [4]), the upload with Algorithms 4 and 5 consists of 2 ciphertexts, and with Algorithms 6 and 7 consists of a *single* ciphertext (for the same parameters!).

The key insight behind our new algorithms is that oblivious expansion (Algorithm 5) is *linear over the plaintext space*. Indeed, all operations used in the algorithms are linear over the plaintext space: additions, substitutions, and scalar multiplications. In particular, it follows that Algorithm 5 enables to expand encryptions of *any* vectors: if $m = \sum_{i \in [N]} m_i x^i \in R/tR$, then the output of the oblivious expansion consists of N ciphertexts, respectively encrypting each of the m_i's in the constant coefficient of the plaintexts.

We propose to modify Algorithm 5 as follows. First, as the algorithm is linear, we propose to perform the normalization in the Query algorithm itself (cf. Line 5 of Algorithm 6). Indeed, in SealPIR [4], the normalization is applied on ciphertexts which in turn requires to use larger parameters to handle the noise growth.[3] This additionally comes with a minor efficiency improvement as it is not necessary to compute any modular product anymore. Second, instead of encrypting the d selection vectors independently in $d \cdot \lceil m/2^c \rceil$ ciphertexts (Lines 4-6 of Algorithm 4), we parse the concatenation of the selection vectors as one vector of length $d \cdot m$ and encrypt it in $\lceil d \cdot m/2^c \rceil$ ciphertexts (Lines 4-6 of Algorithm 6). This further simplifies the implementation of the oblivious expansion algorithm because each ciphertext in the query gets expanded individually (compare Line 15 of Algorithm 7 to Lines 5–6, 17–19 of Algorithm 5).

[3] We note that in the *implementation* of SealPIR, the normalization step happens after decryption, which avoids the need for parameter increase.

Table 1: Gain from our compression techniques (Sections 3.1 and 3.2), compared to SealPIR, for a database of size $n = 2^{20}$ with different length entries and recursion $d = 2$.

Entry size		288B	8kB	2MB
Communication (kB)	up	down	down	down
SealPIR [4]	61.4	307.2	921	200,294
Ours w/o Remark 1	15.4	128	384	83,456
Ours w/ Remark 1	15.4	64	192	41,728
MulPIR	119	119	119	13,660

For SealPIR, we use the parameters of [4, Fig. 9] with plaintext modulus is $t = 2^{12} + 1$, *and we use modulus switching to a prime of* 25 *bits. For MulPIR, we use a polynomial of dimension* 8192 *with* $50 + 2 \cdot 55$ *bit modulus, modulus switching to* 50 *bits, and plaintext modulus* $t = 2^{20} + 2^{19} + 2^{17} + 2^{16} + 2^{14} + 1$.

3.3 Communication Costs

We note that the techniques from the previous sections can be use concurrently. We report in Table 1 the gains obtained by using these techniques on SealPIR with the exact same parameters as in [4], with and without the (computation expensive) optimization Remark 1 for a database of size $n = 2^{20}$ with elements of 288B (as in [4]), but also 20kB and 2MB. In the next section, we show how to further decrease the download costs by using the multiplicative homomorphism of the FV scheme.

4 MulPIR: SealPIR with Multiplicative Homomorphism

Recursion using additive homomorphism, as described in Section 2.3, provides a way to emulate multiplicative homomorphism in one very restricted setting, which suffices for PIR construction. However it has two significant drawbacks. First, it has communication complexity $O\big(F^{d-1}c(n)\big)$ where F is the ciphertext expansion. Henceforth, the PIR download expansion (size of the PIR result compared to the size of the entry) is $O(F^d)$. In practice, for SealPIR [4] with the optimizations of Section 3, the expansion is larger than 20, as shown by Table 1. Second, the ciphertext obtained on the server is not an encryption of the result, but of another ciphertext. As such, the server cannot perform homomorphic operations on the ciphertext to post-process it.

In this section, we aim at tackling the aforementioned issues. In Section 4.1, we recall the generic technique that uses fully homomorphic encryption to construct PIR with optimal

Table 2: Communication-Computation Trade-Off of homomorphic encryption based PIR Protocols.

	Total Communication in number of ciphertexts		**Approximate computation cost** Expressed in homomorphic computation unit: A: addition; S: scalar multiplication; M: multiplication		
Recursion	$1 \leq d \leq \log n$	$d = \log(n)$	$1 \leq d \leq \frac{\log n}{\log F}$	$\frac{\log n}{\log F} < d \leq \log n$	$d = \log(n)$
Additive HE	$O\left(dn^{\frac{1}{d}} + F^{d-1}\right)$	$O\left(\log n + F^{\log n - 1}\right)$	$n(A+S)$	$n^{\frac{1}{d}} F^{d-1}(A+S)$	$F^{\log n-1}(A+S)$
Somewhat HE	$O\left(dn^{\frac{1}{d}}\right)$	$O(\log n)$	$n(A+S) + n^{\frac{d-1}{d}} M$	$n(A+S) + n^{\frac{d-1}{d}} M$	$n(A+S+M)$
Fully HE	–	$O(\log n)$	–	–	$n \log n M + n(A+S)$

This tables aims at giving an insight on the overall trend but does not reflect accurately the costs; e.g., the communication in indicated in number of ciphertexts while the actual size of the ciphertexts may depend on the database size, and similarly the costs of the homomorphic operations differ between each row.

communication complexity, and explain how to instantiate it in Section 4.2. Next, Section 4.3 presents MulPIR, a variant of SealPIR that trades off computation for better communication, especially for large entries. Finally, Section 4.4 explains how to construct conjunctive and disjunctive PIR queries that enable a *batch private set membership* functionality.

4.1 PIR with Fully Homomorphic Encryption

Assume the homomorphic encryption scheme $\mathcal{HE}$ is fully homomorphic, i.e., (w.l.o.g. for ease of presentation) there exists a Eval procedure that takes as input ciphertexts c_i for respective messages m_i and any function description $f\colon \mathbb{Z}_t^\kappa \to \mathbb{Z}_t$, and outputs a ciphertext of $f(m_1, \ldots, m_\kappa)$, which we denote

$$\mathsf{Eval}(\{\mathsf{Enc}(\mathsf{sk}, m_i)\}_{i\in[\kappa]}, f) = \mathsf{Enc}(\mathsf{sk}, f(m_1, \ldots, m_\kappa)).$$

A possible approach to computing the selection vector for the PIR query using FHE is based on the following observation: the i-th bit in the PIR query vector is the output of the equality check between the query index k and i. Hence, instead of sending the selection vector $\vec{s}$, the client can encrypt each bit k_j of the index k and send the resulting $\kappa = \log n$ ciphertexts to the server. The server then homomorphically computes the selection vector and proceeds as in the baseline PIR construction. This construction achieves communication complexity: $O(\log n)$ for the user's query and $O(1)$ for the server's response (note that the ciphertext size is independent of the database, hence included in the O notation.).

In practice, for a given database size, the circuit corresponding to the PIR evaluation has bounded multiplicative depth, and one can use *somewhat* homomorphic encryption (homomorphic encryption that can evaluate multivariate polynomials of bounded degree). In Section 4.2 and Appendices C.1 and C.2 we propose three different methods to implement the PIR homomorphic evaluation: applying successive multiplications, reconstructing the selection vector using the equality circuit, or reconstructing the selection vector using tensor products. Additionally, in Appendix D we report on the cost of evaluating full recursion using the latter method. We note that while the successive multiplication method has larger multiplicative depth than the other two methods, in practice for $d = 2$, the depth is the same and this method is the computationally most efficient. Table 2 illustrates that homomorphic multiplications enable to reduce the communication of recursion, which becomes a bottleneck for large levels of recursion.

4.2 PIR with Successive Multiplication

Using the same notation as in Section 2.3, the PIR protocol becomes as follows. The server performs two steps:

1. For each of the $n^{1/2}$ rows $M_i = (M_{i,1} \cdots M_{i,n^{1/2}})$, the server computes the response with the (encryption of the) selection vector $\vec{s}_2$ as in Eq. (1), i.e., the server obtains the $n^{1/2}$ ciphertexts
$$c_i = \mathsf{Enc}\big(\mathsf{sk}, \langle \vec{s}_2, (M_{i,j})_j \rangle\big) = \mathsf{Enc}\big(\mathsf{sk}, \mathsf{D}_{in^{1/2}+j'}\big).$$
2. The server now computes the response with the (encryption of the) selection vector $\vec{s}_1$ using homomorphic multiplication, i.e., the server obtains the ciphertext
$$c = \mathsf{Enc}\big(\mathsf{sk}, \langle \vec{s}_1, \{\mathsf{D}_{in^{1/2}+j'}\}_i \rangle\big) = \mathsf{Enc}\big(\mathsf{sk}, \mathsf{D}_{i'n^{1/2}+j'}\big).$$

Upon reception of the response, $r = c \in \mathcal{C}$, the client directly uses the HE decryption key to recover $\mathsf{D}_{i'n^{1/2}+j'} = \mathsf{D}_k$.

Here again, this method easily generalizes by representing the database as a d-dimensional hyperrectangle $[n_1] \times \cdots \times [n_d]$ with $n = n_1 \cdot n_2 \cdots n_d$. When $n_i = n^{1/d}$, we accomplish the following communication complexity: $O\big(c(n) \cdot dn^{1/d}\big)$ for the user's query and $O(c(n))$ for the server's response.

4.3 MulPIR

We note that the FV homomorphic encryption scheme used in SealPIR, that is actually a *somewhat* homomorphic encryption scheme, and the parameters can hence be chosen so as to handle an a priori bounded number of multiplications.

We therefore propose MulPIR, which combines the layered multiplication approach from Section 4.2 with the optimizations from Section 3. In particular,

- The MulPIR.Query algorithm is the same as in our optimized variant of SealPIR (Algorithm 6).
- Upon receipt of the query, the server obliviously expands the query using Algorithm 7;
- Then the server runs the layered multiplication algorithm of Section 4.2;
- Next the server compresses the response using modulus-switching as in Section 3.2;
- Finally, the client extracts the database elements by decrypting the result.

MulPIR trades off computation (higher computational costs for the server) for smaller communication (in total communication, and more particularly for the download communication). We report communication costs compared to SealPIR in Table 1, and will report on both concrete communication and computation costs in Section 6.

4.4 Conjunctive and Disjunctive PIR Queries

The PIR functionality provides the client with the database content at the location specified in the query. However, in some settings we may want to reveal strictly less information to the client, which is a function of the outputs of multiple queries. One example is the setting of conjunctive and disjunctive queries where we want to enable the client to learn only whether all of her queries are present in that database in the first case, or if any of her queries are in the database in the second case. Here, the answers of individual queries are bit values and the goal is to compute the conjunction or disjunction over the responses of multiple queries.

PIR constructions based on somewhat and fully homomorphic encryption naturally provide capability for further computation over the answers of the queries (contrary to e.g., SealPIR), if the server obtains the encrypted answers of the individual queries under HE that supports further computation over the encrypted data.

Conjunctive Queries. A conjunction over the k bit values $b_1, \ldots, b_k$ can be computed as follows $c = r \cdot (\sum_{i \in [k]} (1 - b_i))$, where r is a randomly chosen non-zero value. The value c is 0 if all input value $b_i = 1$ for $i \in [k]$, and its a random non-zero-value, otherwise. In the context of PIR queries, if the server obtains the answers of k PIR queries as $\mathsf{Enc}(b_1), \ldots, \mathsf{Enc}(b_k)$, it suffices that Enc is additively homomorphic to enable the server to compute $r \cdot (\sum_{i \in [k]} (\mathsf{Enc}(1) - \mathsf{Enc}(b_i))) = \mathsf{Enc}(c)$ and obtain the encrypted answer of the conjunctive query that it can return to the client.[4]

Bloom filters (BFs) [7] are a data structure which gives a space efficient way to store a sparse database. A Bloom filter is associated with k hash functions $H_1, \ldots, H_k$ and an array of size n, the output domain of the hash function, which is initialized to contain all zeros. To insert and entry x in a Bloom filter, we set $\mathsf{BF}[H_i(x)] = 1$ for all $i \in [k]$. To check whether x is stored in a Bloom filter, we check whether the values at all locations $\mathsf{BF}[H_i(x)]$ are equal to 1, i.e. the output if the conjunction of the values $\mathsf{BF}[H_i(x)]$.

We can implement PIR membership queries over a sparse database, by having the server create a Bloom filter that contains its database with public parameters $H_1, \ldots, H_k$. A client query for an element x will be executed as a conjunctive query over the values $H_1(x), \ldots, H_k(x)$.

Disjunctive Queries. A disjunction over the k bit values $b_1, \ldots, b_k$ can be computed as follows $c = r \cdot (\prod_{i \in [k]} (1 - b_i))$, where r is a randomly chosen non-zero value. The value c is 0 if there is at least one input value $b_i = 1$, and its a random non-zero-value, otherwise. Thus, in order to enable the server is a multi-query PIR to compute a disjunction over the answers $\mathsf{Enc}(b_1), \ldots, \mathsf{Enc}(b_k)$ of individual queries, we need a HE that supports $k - 1$ multiplications.

The disjunctive query functionality enables computing whether the intersection of two sets is non-empty. One example scenario where this functionality can be used is a privacy preserving exposure notification in contact tracing[5], where a server holds a database of identifiers of people who have reported to have tested positive and a client holds a database of the identifier of people whom she has been in proximity with. The goal for the client is to learn whether she has been in proximity to any infected person.

Composition of Conjunction and Disjunction Queries. We also note that the disjunction and conjunction constructions compose well: both of the constructions output 0 on a match and random value on a non-match. These can be used as inputs in following conjunction and disjunction queries as follows: a conjunction will be computed as a sum of all input and a disjunction can be computed as a product of all inputs.

5 Improving Gentry–Ramzan PIR

An alternative to PIR based on homomorphic encryption is the protocol of Gentry and Ramzan [32], which achieves logarithmic communication and a constant communication rate. While it has been implemented in previous work [18, 19, 51], it is usually dismissed due to its computational complexity [3, 18].

In this section, we describe several optimizations to Gentry–Ramzan PIR that allow us to get a practically efficient implementation. Since the main computation bottleneck for large databases is the server computation (cf. Algorithm 8), we focus on optimizing this part of the protocol. We will first revisit

[4] Note that when recursion is *not* used instead of sending k queries, the client can send one query that is a k-hot vector (a vector with k values 1), as PIR will naturally compute the sum. This is compatible with our oblivious expansion algorithm Algorithm 7 which is linear over the plaintext space.

[5] https://www.apple.com/covid19/contacttracing

Procedure 8 PIR.GR.Query

Parameters: security parameter λ.
Input: $k \in [n]$.
$Q_1 := 2q_1 + 1$ s.t. Q_1 and q_1 are prime and $\log_2(Q_1) \geq \lambda$.
$Q_2 := 2q_2\pi_k + 1$ s.t. Q_2 and q_2 are prime and $\log_2(Q_2) \geq \lambda$.
$m := Q_1 Q_2$.
$g \leftarrow_s \mathbb{Z}_m$ s.t. $|\langle g \rangle| = q_1 q_2 \pi_k$.
Output: $(m, g) \in \mathbb{Z} \times \mathbb{Z}_m^*$.

Procedure 9 PIR.GR.Response

Input: $\mathsf{D}, (m, g) \in \mathbb{Z} \times \mathbb{Z}_m^*$.
Encode D as an integer E as in Eq. (2).
$g' := g^E \bmod m$.
Output: $g' \in \mathbb{Z}_m^*$.

the original protocol [32] (Section 5.1). Then, in Section 5.2, we show how to apply existing techniques [8, 61] to speed up the server setup of Gentry–Ramzan PIR. While this is a one-time setup, it is non-trivial to implement with complexity sub-quadratic in the database size. Finally, in Section 5.3, we show how to speed up the response computation with a novel *client-aided* variant of Gentry–Ramzan PIR, using the fact that the client can perform modular exponentiations more efficiently since he knows the order of the multiplicative group.

5.1 Gentry–Ramzan PIR

The basic PIR protocol of Gentry and Ramzan [32] works by interpreting the server's database as a number in a Residue Number System (RNS). That is, given n coprime integers $\pi_1, \ldots, \pi_n$, with $\pi_i \geq 2^l$ for all $i \in [n]$, we encode D as an integer E, such that

$$E \leq \prod_{i=1}^{n} \pi_i, \quad \text{and} \quad E \equiv \mathsf{D}_i \bmod \pi_i \text{ for all } i \in [n]. \tag{2}$$

The existence and uniqueness of E follows from the Chinese Remainder Theorem, which can also be used to compute E given D and all π_i. Observe that (2) implies that we can retrieve the element at index i by reducing E modulo π_i. The idea of [32] is to have the server perform this reduction *in the exponent* of a multiplicative group, thus hiding i. We give the description of the PIR protocol in Algorithms 8 to 10, and refer the reader to [32] for the details.

5.2 Fast Modular Interpolation

Before being able to answer queries, the server must encode the database D according to Eq. (2). Let $M = \prod_{i=1}^{n} \pi_i$ be the product of all moduli, and $M_k = M/\pi_k = \prod_{i=1, i\neq k}^{n} \pi_i$. A naive application of the Chinese Remainder Theorem computes E as follows:

Procedure 10 PIR.GR.Extract

Input: $g' \in \mathbb{Z}_m^*$.
$h := g^{q_1 q_2}$
$h' := g'^{q_1 q_2}$
Solve $h' = h^d$ for d using Pohlig–Hellman algorithm.
Output: $d \in \mathbb{Z}_{\pi_i}$.

1. For each $k \in [n]$, use the extended Euclidean algorithm to compute integers a_k, b_k such that $a_k M_k + b_k \pi_k = 1$.
2. Compute $E = \sum_{k=1}^{n} D_k a_k M_k = \sum_{k=1}^{n} D_k a_k \left(\prod_{i=1, i\neq k}^{n} \pi_i \right)$.

It is clear that a given modulus π_k divides all summands from Step 2 except the k-th. Then, using the identity from Step 1, we have $E \equiv D_k a_k M_k \equiv D_k - D_k b_k \pi_k \equiv D_k \bmod \pi_k$ for all $k \in [n]$. The problem with that solution is that each M_k has already size $\Omega(n)$. While there are quasi-linear variants of integer multiplication [61] and the extended Euclidean algorithm [62], we have to perform each of those at least n times, and therefore end up with a total running time of $\Omega(n^2)$.

To avoid the quadratic complexity, we rely on the modular interpolation algorithm by Borodin and Moenck [8]. Their main observation is that if we divide our set of moduli π_i evenly into two parts, and call the products of those parts M_1 and M_2, then the first half of the summands in Step 2 above contains M_2 as a factor, while the other half contains M_1. Thus, M_1 and M_2 can be factored out of the sum, reducing the computation to two smaller sums and two multiplications:

$$E = M_2 \cdot \left(\sum_{k=1}^{\lfloor n/2 \rfloor} d_k a_k \left(\prod_{i=1, i\neq k}^{\lfloor n/2 \rfloor} \pi_i \right) \right) + M_1 \cdot \left(\sum_{k=\lfloor n/2 \rfloor + 1}^{n} d_k a_k \left(\prod_{i=\lfloor n/2 \rfloor + 1, i\neq k}^{n} \pi_i \right) \right).$$

Repeating the above transformation recursively leads to a divide-and-conquer algorithm for modular interpolation, which, using the Schönhage-Strassen integer multiplication [61], has a total running time of $O(n \log^2 n \log\log n)$ [8]. It relies on the fact that the *supermoduli* M_1, M_2 can be precomputed, as well as the inverses a_k. This is especially useful, as we can reuse those for multiple interpolations, as long as the set of moduli π_i remains the same. We will make use of this precomputation when applying our implementation of Gentry–Ramzan PIR to databases with large entries (Section 6.2).

5.3 Client-Aided Gentry–Ramzan

As we can see in Algorithm 9, to compute the response to a query, the server has to compute a modular exponentiation, where the exponent encodes the entire database as described in the previous section. Prior work [19] has shown that in practice this step is by far the most expensive part in Gentry–Ramzan PIR.

To speed up the response computation, we rely on the well known fact that one can use Euler's Theorem to perform modular exponentiations of the form $g^x \bmod m$ by first reducing the exponent modulo $\varphi(m) = (Q_1 - 1)(Q_2 - 1)$ and computing

$$g^x \bmod m = g^{x \bmod \varphi(m)} \bmod m. \tag{3}$$

While we cannot apply this directly to Algorithm 9 because the server does not know $\varphi(m)$, the client can use Eq. (3) to perform a part of the server's computation *without knowing* E, by precomputing powers of the generator g.

Concretely, the server rewrites the large exponent E according to some base $b \geq 2$. Without loss of generality, we know that $E = E_0 + E_1 b + E_2 b^2 + \ldots + E_l b^l$. It follows that $g^E = g^{E_0} \cdot (g^b)^{E_1} \cdot (g^{b^2})^{E_2} \cdots (g^{b^l})^{E_l}$. Observe that since b and l are public, the client can compute the $l+1$ values $g, g^b, g^{b^2}, \ldots, g^{b^l}$ without knowing the exponent E. Furthermore, these l exponentiations may be efficiently computed by the client using the prime factorization of m as shown in Eq. (3). Note that revealing the additional powers of g to the server does not leak any information, as they could be computed by the server as well, just not as fast. Given these $l+1$ values, the server's task reduces to the problem of computing the product of multiple parallel exponentiations. To do this efficiently, one can refer to the survey by Bernstein [6]. For our implementation, we choose Straus's algorithm [64], a description of which can be found in [40, Alg. 14.88]. In our experiments (Table 5 and Fig. 3), we show how sending more generators reduces significantly the server computation time.

6 Experimental Evaluation

We present experimental results that measure the efficiency of different PIR protocols and illustrate some of the possible tradeoffs that they enable. These results can inform decision making of what is the most appropriate PIR instantiation for a particular application. All our experiments are performed in a virtual machine with a Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 128GB of RAM, running Debian.

6.1 SealPIR and MulPIR

First, we report on the relative costs of SealPIR and MulPIR. For SealPIR, we use the parameters of [4]: polynomials of dimension 2048 and a modulus of 60 bits, providing 115 bits of security. The plaintext modulus set to $t = 2^{12} + 1$. One ciphertext will therefore encrypt $12 \cdot 2048/8 = 3072$ bytes. For MulPIR, we use polynomials of dimension 8192 and a modulus of 160 bits, providing 180 bits of security. The plaintext modulus set to $t = 2^{20} + 2^{19} + 2^{17} + 2^{16} + 2^{14} + 1$. One ciphertext will encrypt $20 \cdot 8192/8 = 20480$ bytes. We use the SealPIR implementation available on Microsoft's GitHub [3] based on Seal 3.2.0, and we implement MulPIR with the latest version of Seal (3.5.4) [2].

The first database is a "Pung-style" database, as used in [4]. This is a database of $n = 2^{16}, 2^{18}, 2^{20}, 2^{22}$ elements of 288B. The first step in SealPIR is to reshape the database into a database of $\lceil n/10 \rceil$ entries of 2880B, to *fully pack* each ciphertext. Similarly, MulPIR reshapes the database into a database of $\lceil n/71 \rceil$ entries of 20448B. We present the communication and computation comparison in Table 3. As expected, the communication is smaller than the implementation of SealPIR from [4] for a slightly larger computational cost. We note that reimplementing SealPIR with the optimizations from Section 3.2 and the latest version of Seal should give better communication than MulPIR (cf. Table 1). This is due to the fact that the database is *long* and *skinny*: it has many entries that are really short; hence the PIR expansion factor is not the bottleneck (cf. the latest column of Table 1).

To better visualize the communication–computation tradeoff of MulPIR, we also benchmark MulPIR (and estimate the communication and computation of the optimized version of SealPIR from Section 3.2, without the computation-expensive Remark 1 which could make it more costly than MulPIR and requires careful benchmarking) for a database with larger entries: we consider a database of size 100,000 with entries of 40kB. For SealPIR, we consider an upload of $(128 + 2048 * 60)/8 = 15376$ bytes, and the download to be

$$\left\lceil \frac{40000}{3072} \right\rceil \cdot \left\lceil \frac{2 \cdot 60}{12} \right\rceil 2 \cdot 2048 \cdot 25/8 = 1{,}792{,}000$$

bytes. We estimate the timings of SealPIR by multiplying the server response time minus the server expansion time corresponding to the column where the actual number of rows is $104858 \approx 100{,}000$ by $14 = \lceil \frac{40000}{3072} \rceil$ and ignore the cost of the optimizations from Section 3.2. We conclude from Table 4 that, for a similar computation cost, MulPIR enables to reduce the communication of SealPIR by a factor 7x in that setting.

6.2 Comparison with Other PIRs

For completeness, we want to compare the cost of SealPIR/MulPIR with other additive homomorphic encryption schemes, and in particular El Gamal and Damgård–Jurik. Since we expect those schemes to be much slower, and in particular prohibitively expensive for the client, we first run a complete benchmark on a very small database of 5000 elements of length 288B (such database was used for evaluation in [5]), without using recursion (so as to maximize speed). We report communication and computation costs when the database is packed (i.e., when possible, the database is reshaped so as to maximize the number of elements in the response; as done in SealPIR [4] and in the previous section). We also consider a "private file download" application, that uses a "short" and "fat" database with $10,000$ files of $307,200$ bytes (3GB database), and serve it with PIR without recursion.

Table 3: Communication and CPU costs (in ms) of SealPIR and MulPIR (recursion $d = 2$) for a database of n elements of 288B.

	SealPIR ($d = 2$) [3]				MulPIR ($d = 2$)			
Database size n	65536	262144	1048576	4194304	65536	262144	1048576	4194304
Actual number of rows after packing	*6554*	*26215*	*104858*	*419431*	*924*	*3693*	*14769*	*59075*
Client Query	42	42	42	42	175	206	226	247
Server Expand	93	186	357	724	219	451	896	1839
Server Respond	390	1172	3765	14050	899	2234	5950	18122
Upload (kB)	61.4	61.4	61.4	61.4	122	122	122	122
Download (kB)	307	307	307	307	119	119	119	119

Table 4: Communication and CPU costs (in ms) of SealPIR and MulPIR (recursion $d = 2$) for a 4GB database with $n =$ 100,000 elements of 40kB.

	Optimized SealPIR	MulPIR
Client Query (ms)	42	263
Server Expand (ms)	357	3560
Server Response (ms)	47712	52280
Upload (kB)	15	119
Download (kB)	1792	238

In this regime, all the PIR protocols are fully packed and need to replicate their operations over "# chunks" ciphertexts. We report communication costs and benchmarks in Table 5 and in Fig. 2.

For El Gamal, we use the NIST P-224r1 curve and the plaintext size is chosen to be 4 bytes for fast decryption. For Gentry–Ramzan, we use a 2048-bit modulus and a blocksize of 500. For Damgård–Jurik, we use $s = 1$ and 1160-bit primes, and a ciphertext encrypts about 290 bytes. For MulPIR, we use a polynomial of dimension 2048 and a modulus of 60 bits. All the implementations are standalone and rely only on OpenSSL for BigNum and elliptic curve operations. Damgård–Jurik client's setup includes precomputation to speed up the query creation. Finally, the table reports the server cost for a single execution of the experiment on Google's Cloud Platform [1], using a preemptible general-purpose VM with a single CPU core. As expected, Damgård–Jurik and ElGamal are significantly slower than MulPIR and Gentry–Ramzan, and will not be considered further in the rest of the section. We also note that the server computation in client-aided PIR reduces significantly as we send more generators.

6.3 Application: Password Checkup

Recent works study the problem of preventing credential stuffing attacks [44,65] by proposing privacy-preserving protocols where a client queries a centralized breach repository to determine whether her username and password combination has been part of breached data, without revealing the information queried. While this application seems to be a perfect fit for keyword PIR, the size of leaked credentials (4+ billion credentials [65]) remains prohibitively large for PIR. Instead, [44,65] propose protocols where the client and the server first run an oblivious PRF evaluation (both on usernames and on the tuple username/password), then use the first value to retrieve a bucket and the second value to test for membership after downloading the whole bucket. Precisely, [65] proposes to use 2^{16} buckets, which we infer to contain about $60k$ elements, and downloading a whole bucket is about 1.6MB.

In this section, we propose to replace the download of the entire bucket with a PIR query. Table 6 shows that using PIR on each bucket is practical (i.e., is comparable to the median waiting time of a few seconds for the client, reported in [65, Tab. 2]) and enables decreasing communication or the number of buckets (or both).

For Gentry–Ramzan, we propose to perform keyword PIR over a bucket using Cuckoo hashing, as introduced in Appendix E.1. The communication is extremely small for any bucket size. For buckets of size 50k, the server computation time is only slightly larger than one second. Unfortunately, the client needs to generate large safe prime numbers which has high computation cost and may impact the applicability of this protocol in practical deployments, such as the one of [65]. In Fig. 3, we illustrate the communication-computation tradeoff offered in client-aided Gentry–Ramzan PIR: the larger the messages (i.e., the more generators are sent by the client), the smaller the computation time required on the server.

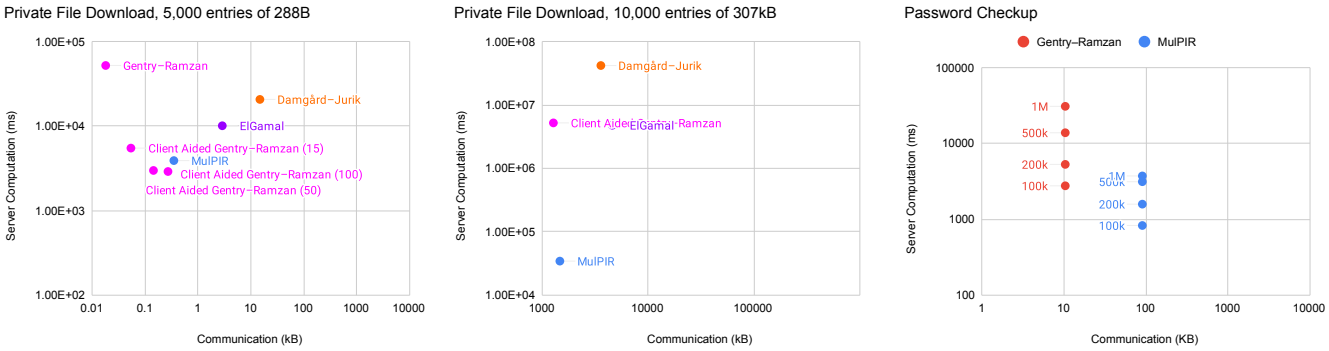
We also propose to use MulPIR, which features really low client's computation costs and low server computation costs. While we could use the Cuckoo hash-based keyword PIR as above, MulPIR would perform worse than Gentry–Ramzan for two reasons. First, the client needs to query as many locations as the number of hash functions. While Gentry–Ramzan supports CRT batching, MulPIR does not support batching natively and its server costs are multiplied by the number of hash functions. Second, a lot of space available in a MulPIR ciphertext is wasted by using Cuckoo hashing, since each bucket row contains at most one element. Therefore, we propose to use a simpler solution: the server selects a random

Table 5: Communication and computation costs for PIR protocols for two databases, without recursion.

	# chunks	Communication (kB) upload	download	Computation (ms) C.Setup	S.Setup	C.Create	S.Respond	C.Process	Server Cost (US cents)
1MB database: 5000 elements of 288B.									
MulPIR	1	14	21	0	39	154	3,910	0	0.0019
Gentry–Ramzan (1 generator)	5	0.5	1.3	0	1,532	3,294	51,803	377	0.0145
Client-Aided Gentry–Ramzan (15 generators)	5	4.1	1.3	0	1,540	2,688	5,495	381	0.0016
Client-Aided Gentry–Ramzan (50 generators)	5	13.1	1.3	0	1,594	3,966	2,988	393	**0.0011**
Client-Aided Gentry–Ramzan (100 generators)	5	25.8	1.3	0	1,796	7,980	2,904	417	0.0014
Damgård–Jurik ($s=1$)	1	1,480	0.6	40,636	2	14,334	20,710	6	0.0382
ElGamal	72	280	8	283	29	893	10,105	26,544	0.0091
Private File Download – 3GB database: 10,000 elements of 307kB.									
MulPIR	100	79.4	1,385	0	88,815	198	34,388	23	**0.0417**
Client-Aided Gentry–Ramzan (50 generators)	4,955	13.1	1,259	6	1,347,036	28,684	5,221,052	355,940	1.4782
Damgård–Jurik ($s=1$)	1,060	2,960	614	$\approx$ 80,000	$\approx$ 3,200	$\approx$ 28600	$\approx$ 42,000,000	$\approx$ 2,500	11.7451
ElGamal	76,800	280	4,300	$\approx$ 300	$\approx$ 88,800	$\approx$ 2250	$\approx$ 4,800,000	$\approx$ 30,715,200	1.4338

Median over 10 computations. The timings indicated with $\approx$ have been estimated on a smaller number of chunks to finish in a reasonable amount of time.

Figure 2: Server computation with respect to the communication for the private file download database, and the password checkup application.



hash function h of image size k, and use it to construct k bins by placing each of the m elements e in the bin of index $h(e)$. The client then performs a PIR query over a database of size k. In order to minimize k, we want to make the number of elements in each bucket as large as possible while still fitting in one MulPIR ciphertext. Denote $m = ck\ln k$ for a constant c. From [60, Th. 1], we know that with overwhelming probability, the maximum size of the bucket will be $(d_c+1)\ln k$ where d_c is the unique root of $f(x) = 1+x(\ln c - \ln x + 1) - c$ larger than c. For every bucket size, we find experimentally the smallest k such that the whole bin after hashing fits in one MulPIR ciphertext. We instantiate MulPIR with parameters polynomials of dimension 2048, modulus of 60 bits and using modulus switching to a 35-bit modulus, and plaintext modulus $t = 17$ to enable recursion $d = 2$. Finally, k is respectively equal to 403, 403, 1k, 3k, 8k, 22k, and 58k. We report on the communication and computation costs in Table 6. In particular, we conclude that for buckets of size 50k, the server computation time less than 1s for about 50kB of communication (plus the one-time keys that need to be transferred), making MulPIR a promising replacement of bucket download in the application of [65].

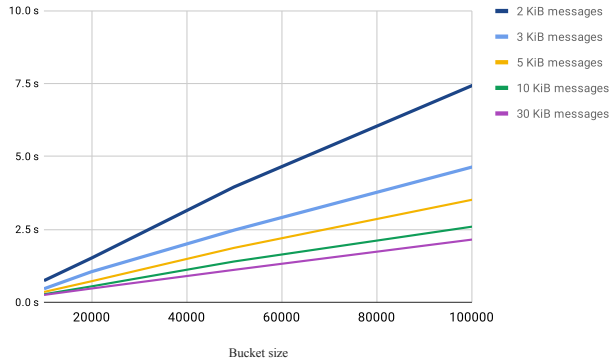
Table 6: Password Checkup application.

Bucket size	Gentry–Ramzan Com. (kB)	Client (ms)	Server (ms)	MulPIR Com. (kB)	Client (ms)	Server (ms)
10k	10.4	24,324	317	90.5	156	475
20k	10.4	19,888	573	90.5	189	515
50k	10.4	24,906	1,649	90.5	195	810
100k	10.4	30,644	2,774	90.5	195	830
200k	10.4	21,571	5,318	90.5	236	1,588
500k	10.4	53,137	13,913	90.5	285	3,143
1M	10.4	49,819	31,055	90.5	265	3,742

7 Conclusion

Similar to other advanced cryptographic primitives, PIR is on the verge of transitioning from a theoretical to a practical tool. Our paper presents significant progress in this direction including new PIR constructions and optimization techniques, which provide new ways to trade-off communication and computation. We implement several PIR constructions using

Figure 3: Computation time in the Password Checkup application when using client-aided Gentry–Ramzan with different number of generators.



different HE schemes as well as the Gentry–Ramzan PIR, and present a comprehensive evaluation of their costs in different settings. Our results demonstrate that the lattice-based FV homomorphic encryption outperforms Paillier and ElGamal in HE-based PIR constructions, while Gentry–Ramzan provides best communication overhead as well as dollar cost for some databases.

Our MulPIR construction enables post-processing of the PIR result using homomorphic encryption, and for the first time an experimental evaluation of full recursion PIR. Overall, our constructions show competitive efficiency for various applications, and we hope our results will serve as a useful reference to inform the choices of PIR construction and parameters in practice.

References

[1] All prices | google compute engine documentation, 2019. https://cloud.google.com/compute/all-pricing. Accessed 2019-11-01.

[2] Microsoft SEAL, 2020. https://github.com/microsoft/SEAL. Accessed 2020-06-17.

[3] SealPIR: A computational PIR library that achieves low communication costs and high performance, 2020. https://github.com/microsoft/SealPIR. Accessed 2020-06-16.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society, 2018.

[5] Sebastian Angel and Srinath T. V. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569. USENIX Association, 2016.

[6] Daniel J. Bernstein. Pippenger's exponentiation algorithm, 2002. http://cr.yp.to/papers/pippenger.pdf.

[7] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] Allan Borodin and R. Moenck. Fast modular transforms. *J. Comput. Syst. Sci.*, 8(3):366–386, 1974.

[9] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography Conference*, pages 662–693. Springer, 2017.

[10] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, 1999.

[11] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography Conference*, pages 694–726. Springer, 2017.

[12] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM Conference on Computer and Communications Security*, pages 1223–1237. ACM, 2018.

[13] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, 2017.

[14] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptology ePrint Archive*, 1998:3, 1998.

[15] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, 1998.

[16] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In *SCN*, volume 11035 of *Lecture Notes in Computer Science*, pages 464–482. Springer, 2018.

[17] Anamaria Costache, Kim Laine, and Rachel Player. Homomorphic noise growth in practice: comparing BGV and FV. *IACR Cryptology ePrint Archive*, 2019:493, 2019.

[18] Sergiu Costea, Dumitru Marian Barbu, Gabriel Ghinita, and Razvan Rughinis. A comparative evaluation of private information retrieval techniques in location-based services. In *INCoS*, pages 618–623. IEEE, 2012.

[19] Emiliano De Cristofaro, Yanbin Lu, and Gene Tsudik. Efficient techniques for privacy-preserving sharing of sensitive information. In *TRUST*, volume 6740 of *Lecture Notes in Computer Science*, pages 239–253. Springer, 2011.

[20] Wei Dai, Yarkin Doröz, and Berk Sunar. Accelerating SWHE based pirs using gpus. In *Financial Cryptography Workshops*, volume 8976 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 2015.

[21] Ivan Damgård and Mads Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[22] Daniel Demmler, Amir Herzberg, and Thomas Schneider. Raid-pir: Practical multi-server pir. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, 2014.

[23] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *PoPETs*, 2018(4):159–178, 2018.

[24] Changyu Dong and Liqun Chen. A fast single server private information retrieval protocol with low communication cost. In *ESORICS (1)*, volume 8712 of *Lecture Notes in Computer Science*, pages 380–399. Springer, 2014.

[25] Yarkin Doröz, Berk Sunar, and Ghaith Hammouri. Bandwidth efficient PIR from NTRU. In *Financial Cryptography Workshops*, volume 8438 of *Lecture Notes in Computer Science*, pages 195–207. Springer, 2014.

[26] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[27] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005.

[28] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Proceedings of the Second International Conference on Theory of Cryptography*, TCC'05, 2005.

[29] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.

[30] Craig Gentry and Shai Halevi. Compressible fhe with applications to pir. Cryptology ePrint Archive, Report 2019/733, 2019. https://eprint.iacr.org/2019/733.

[31] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[32] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer, 2005.

[33] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, 1998.

[34] Yael Gertner, Yuval Ishai, Eyal Kushilevitz, and Tal Malkin. Protecting data privacy in private information retrieval schemes. *J. Comput. Syst. Sci.*, 60(3), June 2000.

[35] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 157–167. Society for Industrial and Applied Mathematics, 2012.

[36] Matthew Green, Watson Ladd, and Ian Miers. A protocol for privately reporting ad impressions at scale. In *ACM Conference on Computer and Communications Security*, pages 1591–1601. ACM, 2016.

[37] Jens Groth, Aggelos Kiayias, and Helger Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Public Key Cryptography*, volume 6056 of *Lecture Notes in Computer Science*, pages 107–123. Springer, 2010.

[38] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, 2004.

[39] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, 2009.

[40] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[41] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. Optimal rate private information retrieval from homomorphic encryption. *Proceedings on Privacy Enhancing Technologies*, 2015(2):222–243, 2015.

[42] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4), December 2009.

[43] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *FOCS*, pages 364–373. IEEE Computer Society, 1997.

[44] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *ACM Conference on Computer and Communications Security*. ACM, 2019.

[45] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th International Conference on Information Security*, ISC'05, 2005.

[46] Helger Lipmaa and Kateryna Pavlyk. A simpler rate-optimal cpir protocol. In *International Conference on Financial Cryptography and Data Security*, pages 621–638. Springer, 2017.

[47] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[48] Moni Naor, Benny Pinkas, and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, STOC '99, 1999.

[49] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2), May 2004.

[50] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[51] Stavros Papadopoulos, Spiridon Bakiras, and Dimitris Papadias. pcloud: A distributed system for practical PIR. *IEEE Trans. Dependable Sec. Comput.*, 9(1):115–127, 2012.

[52] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious ram with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.

[53] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *ACM Conference on Computer and Communications Security*, pages 1002–1019. ACM, 2018.

[54] Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In *Annual Cryptology Conference*, pages 502–519. Springer, 2010.

[55] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security Symposium*, pages 515–530. USENIX Association, 2015.

[56] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT (3)*, volume 11478 of *Lecture Notes in Computer Science*, pages 122–153. Springer, 2019.

[57] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 125–157. Springer, 2018.

[58] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), January 2018.

[59] Google Blog Post. Helping organizations do more without collecting more data. Cryptology ePrint Archive, Report 2020/531, 2019. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html.

[60] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In *RANDOM*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

[61] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3-4):281–292, 1971.

[62] Damien Stehlé and Paul Zimmermann. A binary recursive gcd algorithm. In *ANTS*, volume 3076 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2004.

[63] Julien P. Stern. A new efficient all-or-nothing disclosure of secrets protocol. In *ASIACRYPT*, volume 1514 of *Lecture Notes in Computer Science*, pages 357–371. Springer, 1998.

[64] Ernst G. Straus. Addition chains of vectors (problem 5125). In *American Mathematical Monthly*, volume 70, pages 806–808, 1964.

[65] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*, pages 1556–1571. USENIX Association, 2019.

[66] Xun Yi, Md. Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Trans. Knowl. Data Eng.*, 25(5):1125–1134, 2013.

A Related Work

Efficient Constructions of Single Server PIR. The most efficient (secure) single server PIR constructions implemented in the recent years [4, 5, 20, 24, 25, 30, 36, 47, 66] are based on homomorphic encryption (HE) techniques and achieve sub-linear communication. The baseline PIR solution (with linear communication complexity) has the client send a selection vector proportional to the database size n encrypted under additive homomorphic encryption, and has the server return a single encrypted entry by performing n homomorphic multiplications with a constant and n homomorphic additions. Sub-linear complexity is achieved by using recursion [63]: the database is viewed as a d-dimensional database, and the query complexity becomes $O(d \cdot n^{1/d})$. Now, for the recursion to work with additive homomorphic encryption schemes, the ciphertext after one level of recursion is viewed as a plaintext in the next layer. In particular, if the additive homomorphic encryption scheme has ciphertext expansion F, the PIR response will include F^{d-1} ciphertexts (where, e.g., $F \geq 6.4$ in lattice-based schemes, as per [4]). This has limited the recursion depth to $d \leq 3$ in practice [4, 47].

Along this line of work, there are several papers that present implementations with various resource tradeoffs. Aguilar-Melchor et al. [47] present XPIR with small computation costs but quite large communication costs. On the other hand, another line of work [41, 46] obtain much smaller (almost optimal) communication at the cost of significantly larger computation. In a recent work, Angel, Chen, Laine, and Setty [4], present SealPIR that strikes a better balance in the communication–computation cost. SealPIR requires only slightly more computation than XPIR but uses almost 1000 times less communication than XPIR (but does not achieve the almost optimal rate of the works [41, 46]). SealPIR is instantiated with the FV (lattice-based) homomorphic encryption scheme [26]. It builds upon XPIR [47, 63] and adds a clever query compression technique that reduces the query communication complexity from $O(dn^{1/d})$ to $O(d\lceil n^{1/d}/N \rceil)$, where N is the number of elements that can be packed in one query ciphertext.

Another known PIR construction that achieves logarithmic communication complexity is the construction of Gentry–Ramzan [32], which does not rely on homomorphic encryption. This PIR construction extends the idea from the work of Cachin et al. [10] which proposes to encode the database $\{\mathsf{D}_i\}_{i\in[n]}$ using the Chinese Remainder Theorem (CRT) representation as $x \in \mathbb{Z}$ s.t. $x \equiv \mathsf{D}_i \bmod \pi_i$ for pairwise coprime moduli $\{\pi_i\}_{i\in[n]}$. The query for an element at position i consists of a group $\mathbb{G}$ and a generator g of a subgroup of $\mathbb{G}$ with order $q \cdot \pi_i$. The server evaluation of the query computes $h = g^x$ in $\mathbb{G}$, which effectively performs a modular reduction in the exponent to select the component $\mathsf{D}_i \mod \pi_i$ masked with the random value q. The client recovers the value D_i by computing the discrete logarithm of h with base g^q. The work of Cachin et al. [10] handled only binary data items, and the Gentry–Ramzan construction [32] shows how to handle larger plaintext domains for the database entries and improves the communication rate to constant. While the resulting construction achieves optimal asymptotic communication rate, it has significant computation costs in several places: the generation of prime numbers needed to instantiate different groups $\mathbb{G}$ at each query, the computation time at the server exponentiating in the query group $\mathbb{G}$, and the decoding which requires computing a discrete logarithm. Because of its computational overhead this PIR construction has been rarely considered as a candidate for implementation and practical applications [18, 19, 51].

In recent years, single server PIR has also been studied in slightly different settings. Two works [9, 11] consider *doubly-efficient PIRs* that attempt to obtain schemes with sub-linear computational costs, but require both significant server overhead and new cryptographic assumptions precluding them from practical applications. Another work [53] introduces the notion of *private stateful information retrieval* where clients store some state over multiple queries. Assuming clients perform enough queries, this scheme obtains both smaller communication and computational costs. In contrast, we build PIR schemes suitable for all settings where clients are stateless and our efficiency guarantees will hold regardless of the number of queries performed by the client.

Specialized PIR Settings. Multi-query PIR considers the setting where several PIR queries are executed at the same time. Ishai et al. [38] proposed a construction based on batch codes, which achieves asymptotic improvements in the communication and computation amortized cost multi-query PIR but remains impractical. The work on SealPIR [4] presented a construction based on probabilistic batch codes instantiated with Cuckoo hashing in a similar spirit as private set intersection constructions, which amortizes CPU cost while introducing a small probability of failure ($\approx 2^{-40}$).

PIR for sparse databases, also known as *keyword PIR* [14], considers the setting where the database size is much smaller than its index domain. Chor et al. [14] presented a solution that builds a binary search tree over the items in the database and

reduces the computation to a logarithmic number PIR queries for the tree levels. Amortized multi-query PSI techniques [12, 16, 56] could also be viewed as solutions in this setting.

Symmetric PIR (SPIR) [34] extents PIR with additional privacy requirement for the database which guarantees that the querier does not learn anything more than the requested item. SPIR is also known as 1-out-of-n oblivious transfer. Naor and Pinkas [48] provided general transformation from PIR to SPIR using oblivious polynomial evaluation, and there have also been direct constructions [43, 45].

B Application to Existing HE Schemes

In this section, we discuss instantiations of the PIR approaches from Section 3 with specific homomorphic encryption schemes. In particular, we consider additive ElGamal [29], Paillier/Damgård–Jurik [21], and FV [26], the constructions of which we overview next.

Additive ElGamal [29]. Let $\mathbb{G} = (g)$ be a cyclic group of order p. The public key is a group element $h = g^x$, where the secret key x is a random integer in $[p-1]$. To encrypt $m \in [p]$, sample randomly $r \leftarrow [p-1]$ and output $c = (c_1, c_2) = (g^r, g^m \cdot h^r)$. To decrypt, compute the discrete logarithm of c_2/c_1^x. This scheme is additively homomorphic: let $c = (c_1, c_2)$ encryption m and $c' = (c'_1, c'_2)$ encrypting m', then $(c_1 c'_1, c_2 c'_2)$ encrypts $m_1 + m_2$ mod p. Note that decrypting requires to compute the discrete logarithm in base g, i.e., we can only decrypt small messages. In particular, an ElGamal ciphertext will have expansion at least $F \geq 2$.

Paillier/Damgård–Jurik [21]. Let $N = pq$ be an RSA modulus. The Damgård–Jurik generalization of the Paillier cryptosystem [50] is an additive homomorphic encryption scheme parametrized by an integer s, such that the plaintext space is $\mathbb{Z}_{N^s}$ and the ciphertext space is $\mathbb{Z}_{N^{s+1}}$. In particular, the ciphertext expansion F can be made as small as desired and $F > 1$. This unusual property enables to simplify the recursion in PIR (cf. Section 3). Using the notation of Section 3, after Step (1), the server obtained $n^{1/2}$ ciphertexts $c_i \in \mathbb{Z}_{N^{s+1}}$. It can then parse this ciphertext as a plaintext element for a Damgård–Jurik scheme with parameter $s+1$; assuming the selection vector $\vec{s}_1$ is encrypted under such a scheme, it can then compute $c' = \mathsf{Enc}_{s+1}(\mathsf{sk}, \langle \vec{s}_1, \{c_i\}_i \rangle) \in \mathbb{Z}_{N^{s+2}}$. In particular, assume a database with elements in N^k. The communication after d levels of recursion, where $1 \leq d \leq \log n$, is:

- $n^{1/d}(dk + d(d+1)/2)\log N$ bits from the client to the server, since each selection vector is encrypted with a modulus $\log N$ bits larger than the previous one,
- $(d+k)\log N$ bits from the server to the client to send the response.

Table 7: Bounds on plaintext size, expansion, and decryption cost.

Scheme	Plaintext size	Expansion	Decryption cost
ElGamal	pt small	≥ 2	2^{pt} mults of 2^{λ}_{EG}-bit nums
Damgård –Jurik	$\mathsf{pt} \leq s \cdot \lambda_{\mathsf{DJ}}$ bits	$\geq 1 + 1/s$	1 exponentiation with λ_{DJ}-bit exp
FV	$\mathsf{pt} < \log(q) \cdot \lambda_{\mathsf{FV}}$ bits	≥ 2	add and mult in $\mathbb{Z}_q[x]/(x^{\lambda_{\mathsf{FV}}}+1)$
Gentry–Ramzan	$\mathsf{pt} < \lambda_{\mathsf{GR}}/4$	> 4	$4\mathsf{pt}\sqrt{n}$

Here, s is an integer parameter, and λ_{DJ}, λ_{EG}, λ_{FV}, and λ_{GR} are the security parameters for the different encryption schemes, the size of which is determined by the underlying hardness assumptions. Although not exactly an encryption scheme, we include Gentry–Ramzan here. In this case, Decryption corresponds to solving a discrete logarithm, for which the running time depends on the database size n [32, p. 808].

FV [26]. The description of FV is given in Section 2.2; we use the notation of that section. Since FV is additively homomorphic, we can apply the baseline PIR and the recursive PIR protocol of Section 3. The size of a ciphertext is given by $|ct| = 2N \log q$. In particular, the communication after d levels of recursion, for $1 \leq d \leq \log n$ is

- $(d \cdot n^{1/d} + \lceil 2\log q / \log t \rceil^d) \cdot (2N \log q)$ bits, from the client to the server where the expansion $F = 2\log q / \log t > 2$,
- $\lceil 2\log q / \log t \rceil^{d-1} \cdot (2N \log q)$ bits from the server to the client to send the response.

However, since FV is also somewhat (and fully) homomorphic, we can apply the PIR protocols of Section 4.1. This enables to reduce the communication to

- $(d \cdot n^{1/d}) \cdot (2N \log q)$ bits, from the client to the server,
- $2N \log q$ bits from the server to the client to send the response.

C Two other SHE-Based PIR Protocols

C.1 Equality Circuit

A first approach consists in implementing the protocol described for fully homomorphic encryption schemes that leverage the observation that, since the values k and i have at most $\kappa = \log n$ bits, the arithmetic circuit for computing equality comparison has multiplicative depth $\log \kappa = \log \log n$. Indeed, computing the equality comparison bit for two bit values b_1 and b_2 is equivalent to computing $1 - (b_1 + b_2 - 2b_1 b_2)$ over the integers. Note that in our case only one of the bits coming from the query will be encrypted. Thus, bit equality computation will not require any multiplicative homomorphism. The dominant cost is therefore the multiplication of $\log n$ encrypted bits, which requires $\log \log n$ multiplicative degree.

Table 8: Baseline PIR communication and computation complexities for with different recursion levels and different homomorphic encryption instantiations on a database of size n.

PIR protocol	**PIR** Baseline	**PIR** Recursion $d = 2$	**PIR** Recursion $d = \log n$
Additive ElGamal	**Comm:** $(n+1)\cdot\lambda_{\mathsf{EG}}$ bits **Comp:** n mults of λ_{EG}-bit nums	**Comm:** $(2n^{1/2}+\lceil F\rceil)\cdot\lambda_{\mathsf{EG}}$ bits **Comp:** $n+n^{1/2}\cdot F$ mults of λ_{EG}-bit nums	**Comm:** $(\log n+\lceil F\rceil^{\log n-1})\cdot\lambda_{\mathsf{EG}}$ bits **Comp:** $F^{\log n-1}$ mults of λ_{EG}-bit nums
Damgård–Jurik ($\mathsf{pt} = N^k$ with $N = 2^{\lambda_{DJ}}$)	**Comm:** $(n+1)\cdot(k+1)\log N$ bits **Comp:** n mults of $(k+1)\lambda_{DJ}$-bit nums	**Comm:** $n^{1/2}(2k+3)\log N+(2+k)\log N$ bits **Comp:** n mults of $(k+1)\lambda_{DJ}$-bit nums + $n^{1/2}$ mults of $(k+2)\lambda_{DJ}$-bit nums	**Comm:** $\approx(k+\log n(1+k\log n+\log n^2))\log N$ bits **Comp:** $n^{i/\log n}$ mults of $(k+1+i)\lambda_{DJ}$-bit nums for all $i\in[\log n]$.
Gentry–Ramzan	**Comm:** $3\lambda_{\mathsf{GR}}$ bits **Comp:** $2\cdot n\cdot\mathsf{pt}$ multiplications of λ_{GR}-bit numbers.	N/A	N/A
FV	**Comm:** $2(n+1)\log(q)\cdot\lambda_{\mathsf{FV}}$ bits **Comp:** n scalar mults + additions in $\mathbb{Z}_q[x]/(x^{\lambda_{\mathsf{FV}}}+1)$	**Comm:** $2(2n^{1/2}+\lceil F\rceil)\log(q)\cdot\lambda_{\mathsf{FV}}$ bits **Comp:** $n+n^{1/2}\lceil F\rceil$ scalar mults + additions in $\mathbb{Z}_q[x]/(x^{\lambda_{\mathsf{FV}}}+1)$	**Comm:** $2\log n+\lceil F\rceil^{\log n-1})\log(q)\cdot\lambda_{\mathsf{FV}}$ bits **Comp:** $F^{\log n-1}$ scalar mults + additions in $\mathbb{Z}_q[x]/(x^{\lambda_{\mathsf{FV}}}+1)$

Hence, it suffices to use a somewhat homomorphic encryption that supports $\log\kappa$ nested multiplications. Then the ciphertext size depends on the size of the database and the communication complexity becomes $O(c(n)\log n)$ for the user's query and $O(c(n))$ for the server's response.

C.2 Selection Vector Reconstruction

Note that the approach of Section 4.2 keeps the layered approach of recursion. In particular, performs *sequentially* d homomorphic multiplications, effectively requiring the somewhat homomorphic encryption scheme to support circuits of multiplicative depth d. In particular, for full recursion, this means that the SHE scheme needs to support circuits of depth $\kappa = \log n$, which increases the size of the ciphertexts compared to the first approach[6], where the SHE only required to handle depth $\log\kappa = \log\log n$.

We propose below a method that trades communication for computation as follows. First, note that

$$\mathsf{D}_{i'n^{1/2}+j'} = \langle \vec{s}_1 \otimes \vec{s}_2, \{\mathsf{D}_i\}_{i\in[n]}\rangle\,,$$

where $\vec{s}_1\otimes\vec{s}_2$ is the tensor product of $\vec{s}_1$ and $\vec{s}_2$. More generally, if $\vec{s}_1,\ldots,\vec{s}_d$ denote the selection vectors of dimension $n^{1/d}$, such that the indices of the 1 element in $\vec{s}_i$ is j_i, then

$$\mathsf{D}_{\sum_{j=0}^{d-1} j_i\cdot n^{j/d}} = \langle \vec{s}_1\otimes\cdots\otimes\vec{s}_d, \{\mathsf{D}_i\}_{i\in[n]}\rangle\,.$$

Hence, this hints to a new protocol, where the client sends the $d\cdot n^{1/d}$ encryptions of the bits s_{j,i_j} for $j\in[d], i_j\in[n^{1/d}]$, the server computes homomorphically

$$\mathsf{Enc}(\mathsf{sk}, s_{1,i_1}\times\cdots\times s_{d,i_d}),\quad \forall i_1,\ldots,i_d\in[n^{1/d}]\,,$$

and then computes the inner product with the original database, as in the baseline PIR (cf. Eq. (1)). Now, note that the latter product can be computed using a binary tree of depth $\log d$. For full recursion, i.e., $d=\log n$, the dominant cost in this algorithm is the multiplication of $d=\log n$ encrypted bits, hence requires $\log d = \log\log n$ multiplicative degree.

D Full Recursion with Leveled homomorphic Encryption

In this section, we report on an implementation of *full* recursion $d=\log n$, using the technique from Appendix C.2. We use the SEAL library [2] with polynomials of degree 8192 and a modulus q of 147 bits (product of three 49-bit moduli), and plaintext space $t=2$. We implemented full recursion for a Pung-style databases of n elements of 288B (in particular, we will have one element per ciphertext) [5] and provide benchmarks for databases of size 2^{14} to 2^{17} in Table 9.[7] While this approach does not bring any benefit in practice compared to recursion $d=2$ using one homomorphic multiplication, we

[6] Indeed, the parameters of somewhat homomorphic encryption schemes scales at least linearly in the multiplicative depth (using techniques called modulus switching or relinearization); hence reducing the multiplicative depth exponentially with also reduce the ciphertext size exponentially.

[7] We ran out of RAM for $n=2^{18}$ with our tree-based implementation of the tensor product. Careful optimizations of the tensor product computation and regular folding would enable to reduce the memory usage of the program at the cost of increasing computation.

report for the first time benchmarks for PIR with full recursion.

Table 9: Full Recursion using Seal-MulPIR.

n	Communication (kB)	Server computation (s)
2^{14}	$14 \cdot 150 + 150$	167
2^{15}	$15 \cdot 150 + 150$	324
2^{16}	$16 \cdot 150 + 150$	658
2^{17}	$17 \cdot 150 + 150$	2109

E Beyond PIR: Sparsity and Database Privacy

In this section we consider functionalities beyond the traditional setup for PIR that bring extended computation capability, efficiency and security properties, which can be advantageous in different application scenarios.

E.1 Keyword PIR using Cuckoo Hashing

The traditional setup for PIR constructions assumes that the database entries have public indices which are known to the client submitting queries. In particular, these indices coincide with the domain of all possible queries for the client. Under this assumption the size of the database is equal to the query domain size, which directly affects the computation and communication costs of the constructions which depend on the database size. In cases when the server database is sparse and only a small fraction of the domain indices correspond to actual entries, using a PIR solution directly will incur a large overhead forcing dependence on the whole domain size. This sparse database setting has been considered as keyword PIR by Chor et al. [14]. The idea of this work is to build an efficiently searchable structure, instantiated with a search tree, over the sparse indices of the database entries and then use PIR to execute the search queries. This approach requires logarithmic number of PIR queries on a database of proportional to the number of sparse items. We propose a new construction which leverages Cuckoo hashing and reduces the overhead to a constant number of PIR queries on a database proportional to the number of data entries.

The idea of our approach is to use Cuckoo hashing to compress the index on the server side. Cuckoo hashing [27, 49] is a dictionary with worst case constant look-up time, which has size linear in the number of inserted items. A Cuckoo hash table is defined by κ hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_\kappa$ and each item with label i is placed in one of the κ locations $\mathsf{H}_1(i), \ldots, \mathsf{H}_\kappa(i)$. The Cuckoo hash table is initialized by inserting all items in order, resolving collisions using a recursive eviction procedure: whenever an element is hashed to a location that is occupied, the occupying element is evicted and recursively reinserted using a different hash function. For each sequence of items, there is a small set of hash function sets that are incompatible with the sequence and cannot be used to distribute the items, but this can be handled by choosing new hash functions. Overall, inserting n elements into a cuckoo hash table can be performed in expected $O(n)$ time [49]. Note that with this procedure the hash functions are dependent on the items placed in the Cuckoo hash table but—unlike in PSI protocols based on Cuckoo hashing [12, 23, 55, 57]—this is not an issue for our use of Cuckoo hashing in the context of PIR where the data is considered public and we do not need to provide any privacy guarantees for it.

Our construction works as follows. The server builds a Cuckoo hash table for its sparse database, which will be of size proportional to the number of non-empty entries (with a constant multiplicative overhead), and provides the Cuckoo hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_\kappa$ for a $\kappa \geq 2$. In order to query an item i, the client executes κ PIR queries for items $\mathsf{H}_j(i), j \in [\kappa]$ for the database that contains the Cuckoo hash table.

We note that our approach to compress the server index using Cuckoo hashing is orthogonal to the use of Cuckoo hashing to batch multiple PIR queries described in Appendix E.3. Combining these two techniques we optimize on two different axis of the PIR construction. Next we present the formal construction for PIR on sparse data.

Construction 1. Let

$$(\mathsf{Cuckoo.KeyGen}, \mathsf{Cuckoo.Query}, \mathsf{Cuckoo.Insert})$$

be a Cuckoo hash scheme and $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ be a PIR protocol. We construct a new PIR protocol $(\mathsf{PIR}'.\mathsf{Query}, \mathsf{PIR}'.\mathsf{Eval})$ where the indices of the server's database are sparse over the whole domain:

- Pre-processing: The server generates parameters for the Cuckoo hash that will fit its input

  $$(\mathsf{H}_1, \mathsf{H}_2, \ldots, \mathsf{H}_\kappa, m) \leftarrow \mathsf{Cuckoo.KeyGen}(|\mathsf{D}|).$$

  It initializes the Cuckoo hash table using its input, invoking $\mathsf{Cuckoo.Insert}(i,d)$ for all $(i,d) \in \mathsf{D}$. It sends to the client $\{\mathsf{H}_j\}_{j \in [\kappa]}$.
- $q_i = (q_i^1, \ldots, q_i^\kappa) \leftarrow \mathsf{PIR}'.\mathsf{Query}(i)$: The client computes $q_i^j \leftarrow \mathsf{PIR.Query}(\mathsf{H}_j(i))$ for $j \in [\kappa]$.
- $[\mathsf{D}[i], \perp] \leftarrow \mathsf{PIR}'.\mathsf{Eval}([q_i, \mathsf{D}])$: The client and the server run $[\mathsf{T}_j[\mathsf{H}_j(i)], \perp] \leftarrow \mathsf{SPIR.Eval}([q_i^j, \mathsf{T}_j])$ for $j \in [\kappa]$. The client checks if any of the $\mathsf{T}_j[\mathsf{H}_j(i)], j \in [\kappa]$ contains item i. If the items is present, the client outputs it and otherwise, the client outputs $\perp$.

E.2 Symmetric PIR from OPRFs

The security requirements of a PIR protocol pertain only to the privacy of the query. Symmetric private information retrieval (SPIR) [33] considers also database privacy in addition

to query privacy. While some PIR solutions based on homomorphic encryption do effectively provide SPIR guarantees in the case when the server returns a single ciphertext that encrypts only the retrieved database entry, other approaches do provide more information about the database to the client. We provide a simple transformation that enables SPIR given any PIR scheme. Our idea is to encrypt each database entry using a symmetric encryption under a key that is derived in a pseudorandom manner from the index of the data item. In particular, the server derives the encryption keys using pseudorandom function that also offers oblivious evaluation mechanism (OPRF) [28, 39]. To execute a SPIR query the client and the server execute the corresponding PIR query on the database of encrypted entries and in addition to this they run an oblivious PRF evaluation that enables the client to get a single decryption key corresponding to the query entry. We present our protocol next.

Construction 2. Let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a semantically secure encryption scheme, $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ be a PIR scheme and $(\mathsf{PRF.KeyGen}, \mathsf{PRF.Eval}, \mathsf{PIR.OblivEvaluate})$ be an oblivious PRF function. We construct an SPIR protocol as follows:

- Pre-processing: The server encrypts its database D of size n as follows. It samples a PRF key $\mathsf{K} \leftarrow \mathsf{PRF.KeyGen}(1^\lambda)$ and for each $i \in [n]$, it computes $\mathsf{K}_i \leftarrow \mathsf{PRF.Eval}(\mathsf{K}, i)$ and sets $\tilde{\mathsf{D}}[i] = \mathsf{Enc}(\mathsf{K}_i, \mathsf{D}[i])$.
- $q_i \leftarrow \mathsf{SPIR.Query}(i)$: Output $\mathsf{PIR.Query}(i)$.
- $[\mathsf{D}[i], \perp] \leftarrow \mathsf{SPIR.Eval}([q_i, \mathsf{D}])$:
  1. The client and the server run $[\tilde{\mathsf{D}}[i], \perp] \leftarrow \mathsf{PIR.Eval}([q_i, \tilde{\mathsf{D}}])$.
  2. The client and the server evaluate
  $$[\mathsf{K}_i, \perp] \leftarrow \mathsf{PRF.OblivEvaluate}([i, \mathsf{K}]).$$
  3. The client retrieves its output $\mathsf{D}[i] = \mathsf{Dec}(\mathsf{K}_i, \tilde{\mathsf{D}}[i])$.

We note that handling sparse data in the setting of SPIR, requires to use oblivious Cuckoo hashing where the hash function parameters are independent of the data inserted in the hash table. Achieving oblivious Cuckoo hashing requires addition a of stash of size $O(\log n)$ that stores items which could not be allocated in the hash table due to collisions [42]. The SPIR construction for sparse data proceeds as follows: the server generates a PRF key K and hash functions for oblivious Cuckoo hashing, it encrypts each item i in its database with key $\mathsf{PRF.Eval}(\mathsf{K}, i)$, the server initializes the oblivious Cuckoo hash with the encrypted data. The server sends the Cuckoo hash functions and the encrypted stash to the client. The client executes a query for item i by running two SPIR queries for $\mathsf{H}_1(i)$ and $\mathsf{H}_2(i)$ using the SPIR construction above. It uses the decryption key $\mathsf{PRF.Eval}(\mathsf{K}, i)$ it has obtained to try to decrypt both the answers in the SPIR queries as well as the encrypted items in the stash. The communication related to the stash can be amortized across different queries.

E.3 Multi-Query PIR

The traditional definition of PIR considers a setting where queries are executed independently one by one. However, there are scenarios where several queries may be available to be executed at the same time. Multi-query PIR solutions aim to leverage the capability for parallel execution of such queries in order to amortize the complexity. We leverage two main types of techniques for batching: probabilistic batch codes based on Cuckoo hashing [49], which have been used in the context of PIR [4] and private set intersection [23, 58], as well as a CRT batching technique introduced by Groth et al. [37] for Gentry–Ramzan.

E.4 Private Set + Functionalities

In this section we discuss functionalities which can be solved using specific PIR instantiations. Two such functionalities are private set membership (PSM) and private set intersection (PSI). Private set membership considers the question how to check whether an element held by one party is in the set held by another party. This problem can be viewed as sparse PIR where the database content is the indices themselves. Private set intersection aims to compute the intersection of two private sets. This problem is a generalization of PSM from a single query to multiple queries. Thus, the PSI problem can be phrased as a multi-query symmetric PIR on a sparse database. In setting where the two intersection sets have asymmetric sizes, i.e., one of the sets is much smaller, solving PSI using multi-query PIR using the smaller set as queries could provide better asymptotic communication complexity than PSI solutions that require communication linear in the size of the sets.