Communication–Computation Trade-offs in PIR

Asra Ali
Google
asraa@google.com

Tancrède Lepoint
crypto@tancre.de

Sarvar Patel
Google
sarvar@google.com

Mariana Raykova
Google
marianar@google.com

Phillipp Schoppmann
Google
schoppmann@google.com

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 80% while preserving essentially the same computation cost. We then present MulPIR, a PIR protocol additionally leveraging multiplicative homomorphism to implement the recursion steps in PIR. While using the multiplicative homomorphism has been considered in prior work, we observe that in combination with our other techniques, it introduces a meaningful tradeoff by significantly reducing communication, at the cost of an increased computational cost for the server, when the databases have large entries. For some applications, we show that this could reduce the total monetary server cost by up to 35%.

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 tradeoff between communication and computation, which allows us to reduce server computation by as much as 85%, at the cost of an increased query size.

Finally, we introduce new ways to handle PIR over sparse databases (keyword PIR), based on different hashing techniques. We implement all of our 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 problem 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 [13], the search for concretely efficient constructions for practical applications has been an active area of research [4, 5, 17, 22, 23, 29, 34, 45, 63]. 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.

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 [41], and such solutions bring significant overheads for both the communication and computation costs compared to information theoretic constructions that are possible in the multi-server setting [19]. While theoretical constructions for PIR [41] 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, 34, 45].

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. This includes the two main types of PIR constructions that rely on conceptually different techniques: PIR leveraging homomorphic encryption, and the PIR approach of Gentry and Ramzan [30] leveraging groups with hidden smooth subgroups. The first type of techniques are used in the majority of existing PIR constructions. While fully homomorphic encryption has been proposed as a tool for building PIR [28], existing PIR implementations [4, 5, 34, 45] leverage constructions approaches that rely only on additive homomorphic encryption. Such constructions emulate a restricted form of multiplicative homomorphism with layers of additive encryption. While this approach allows state-of-the-art protocols such as SealPIR [3, 5] to perform well in terms of computation, it incurs a ciphertext expansion that is exponential in the multiplicative depth of the computation, and has a large communication overhead in practice even for small numbers of layered multiplications.

We present a new PIR construction, MulPIR, that improves on this state of the art in multiple ways. First, we show that the communication overhead of SealPIR can be significantly reduced at next to no cost in terms of computation. We further show that by using the multiplicative homomorphism of the underlying HE scheme, we can further reduce the communication overhead for databases with large entries, this time at an increased computation cost. While using the multiplicative homomorphism has been considered before, we are the first to show that, in combination with our other improvements, it enables a meaningful trade-off between communication and computation. In our experiments on Google Cloud Platform, we observed that this can reduce the total monetary server cost by up to 35%.

We also revisit the Gentry–Ramzan PIR scheme [30], which achieves optimal communication but has a high computation overhead. We show how to efficiently implement Gentry–Ramzan PIR even for large databases, and propose a new client-aided variant that allows for a tunable trade-off between communication and computation costs. We experimentally show that depending on the database shape, either MulPIR or client-aided Gentry–Ramzan PIR minimize the total server cost.

Finally, we turn to *keyword PIR* [12], a variant where the database size is much smaller than the query key domain. While regular PIR constructions assume dense databases and so their complexity depends on the index domain size, keyword PIR aims to achieve server computation cost that depends only on the actual database size as opposed to the key domain size. We present two constructions, based on two different hashing schemes, and show that they enable another way to trade off communication and computation.

We implement all of our novel PIR schemes, as well as alternative approaches, and compare them experimentally on a wide range of applications, including anonymous messaging (as used in previous work [5]), private file download, and password checkup [62]. Due to space constraints, we present related work in Appendix A.

1.1 Our Contributions

Improving SealPIR communication. The most efficient (secure) single server PIR constructions implemented in recent years [4, 5, 17, 22, 23, 29, 34, 45, 50, 63] 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, 5]. While theoretically this construction supports sub-linear communication complexity $O(d \cdot n^{1/d})$ leveraging d recursion levels, it comes with a large communication overhead in practice. This is due to the layered additive homomorphic encryption approach: if the encryption scheme has ciphertext expansion F, the PIR response will include F^{d-1} ciphertexts (where $F = 10$ in [3]). This yields communication expansion of $O(F^2)$, which becomes unacceptable for databases with large entries.

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. Therefore, our optimized SealPIR reduces by up to 75% the upload communication, and up to 80% the download communication.

Leveraging Multiplicative Homomorphism. When recursion is used in SealPIR, the download communication depends exponentially on the recursion level (the previous contribution reduced the basis of the exponential). Instead, we propose to use both the additive and *multiplicative* homomorphisms of the underlying HE scheme 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 successive multiplications. Together with our improvements to SealPIR mentioned above, the multiplicative homomorphism enables a highly communication-efficient PIR scheme, which we call MulPIR. For databases with large entries, its advantage over (optimized) SealPIR is already visible with low recursion level (download communication reduced by 60%), and in fact we observe that $d = 2$ remains optimal for the database sizes we are interested in.

Gentry–Ramzan PIR: New Efficiency Trade-offs. The Gentry–Ramzan PIR construction [30] achieves optimal communication complexity for several settings but it pays with significant computational cost. Thus, our contributions here 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 pairwise co-prime 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 [7] 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 for each query on the server side is the modular exponentiation, where the exponent is the encoded database, and the base and the modulus are chosen by the client. Our approach is to compute the exponentiation as a product of precomputed powers of the generator and to use Straus's algorithm [61] to do this efficiently. This enables a client-aided technique that allows us to improve the server's computation at the price of (small) additional work at the client. In particular, we observe that powers of the generator can be precomputed more efficiently by the client, by using the prime factorization of the modulus to reduce the exponent modulo the order of the group prior to exponentiating. 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.

Keyword PIR for Sparse Databases. We consider the setting of sparse databases where the server's database is sparse in the index domain, and hence a client query corresponds to a keyword lookup. We present two constructions, both of which are based on hashing schemes. The first is based on simple hashing, where database elements are assigned to buckets using a public hash function. The client then retrieves the bucket corresponding to their query. While the size of the buckets can generally get quite large, this is no concern in schemes that have a large plaintext size anyway, such as MulPIR. Our second constructions leverages cuckoo hashing [47] in a novel way. Unlike previous work [5], where cuckoo hashing was used to batch multiple client queries into one, we use it to compress the sparse server database into a dense domain. Cuckoo hashing guarantees that at most one element gets hashed to any bucket, at the cost of an increased (but constant) number of client queries. This variant is especially useful for Gentry–Ramzan PIR, where we have small plaintexts, and additionally can use CRT batching [35] to compress multiple client queries into one.

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 enables a trade-off of computation for communication, and compare it against SealPIR.

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 [62] 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

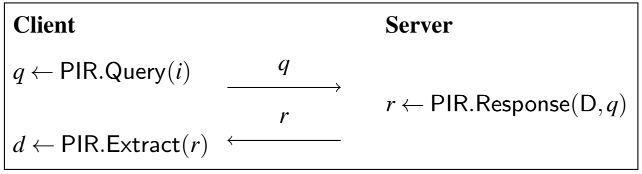
Throughout the rest of this paper, we assume a server owns a database $\mathsf{D} = \{\mathsf{D}_1,\ldots,\mathsf{D}_n\}$ of n elements of size l bits.

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 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 [13])**.** 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

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


Client
Server
$q \leftarrow \mathsf{PIR.Query}(i)$
q
$r \leftarrow \mathsf{PIR.Response}(\mathsf{D}, q)$
r
$d \leftarrow \mathsf{PIR.Extract}(r)$

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 a 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 [24]. For space constraints, ElGamal and Paillier/Damgård–Jurik are recalled in Appendix D.

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, [5] 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 ElGamal, Paillier/Damgård–Jurik, and captures the SealPIR protocol proposed by Angel et al. [5] (based on lattice-based additive homomorphic encryption).

Baseline PIR. We recall the baseline solution for PIR based on homomorphic encryption [41]. 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 $\mathbb{Z}_t$.

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 number of elements in 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

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: $r \in \mathcal{C}$.
```

Procedure 3 PIR.HE.Extract

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

and the database D (where $\mathsf{D}_i \in \mathbb{Z}_t$), eventually yielding

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

In the rest of the paper, we will instantiate this protocol with the Paillier/Damgård–Jurik cryptosystem [18, 48], the El-Gamal cryptosystem [26], the FV cryptosystem [5, 24, 45]. In Appendix D and Table 7, we report on the specific communication and computation costs of these schemes.

Cost of the baseline PIR Denote by $c(n)$ the size of a ciphertext element that enables n homomorphic scalar multiplications followed by n homomorphic additions. The overall communication cost is $n \cdot c(n) + 1 \cdot c(n)$, hence, is at least linear in the database size.

Two approaches have been proposed in the literature to reduce the overall communication cost: either using recursion (also called folding [29]) using additive homomorphic encryption, or using fully homomorphic encryption. We survey these two approaches below.

Recursion/Folding. Kushilevitz, Ostrovsky [41], and later Stern [60], 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 $M_{i,j} := \mathsf{D}_{in^{1/2}+j}$. 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 $n_1 = n_2 = n^{1/2}$).

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 constant values of d, we will get sub-linear communication. However, note that for full recursion, i.e., $d = \log n$, communication becomes super-linear in n.

3 SealPIR: Optimizations and Multiplicative Homomorphism

SealPIR was proposed by Angel et al. [5], and improves over the XPIR protocol proposed by Aguilar Melchor et al. [45]. Both SealPIR and XPIR instantiate the recursive PIR described above, using the FV homomorphic encryption scheme viewed as an additive HE scheme.

This section presents three optimizations to SealPIR: compressing the upload using the secret key for encryption (Section 3.1), compressing the download with modulus switching

[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 [18], 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 D.

(Section 3.1), and a new oblivious expansion technique (Section 3.2). Next, Section 3.4 investigates the impact of using both the additive and multiplicative homomorphism of the FV homomorphic encryption (this variant is called MulPIR), and show that for some database shapes, the download and total communication can be reduced compared to (an optimized version of) SealPIR.

3.1 Halving SealPIR Communication

This section proposes methods to halve "for free" (with minor computational cost) the communication of SealPIR [5].

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 [14]. 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 $\approx \log_2 p / \log_2 q$.

Procedure 4 SealPIR.Query

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

Procedure 5 SealPIR Oblivious Expansion

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

3.2 New Oblivious Expansion

SealPIR 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 [5] only described the inner loop and normalization (Lines 8–16

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

Procedure 7 New Oblivious Expansion

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

and 21–23), but we provide here the algorithm in full for better comparison with our new algorithm (Algorithm 7).

We now describe optimized versions of the Query and oblivious expansion algorithms in Algorithms 6 and 7, which enable to reduce the upload communication 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 [5]), 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 [5], 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).

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 [5]	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 [5, Fig. 9] with plaintext modulus $t = 2^{12} + 1$, 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 [5], with and without the (computation expensive) optimization Remark 1 for a database of size $n = 2^{20}$ with elements of 288B (as in [5]), but also 20kB and 2MB.

3.4 Using Multiplicative Homomorphism – Introducing MulPIR

Recursion using additive homomorphism only, as described in Section 2.3, provides a way to *emulate* multiplicative homomorphism in one very restricted setting, which suffices for PIR construction. It was proposed at a time where no candidate for somewhat/fully homomorphic encryption was known. Since [27], it is well-known that PIR can be instantiated using homomorphic additions and multiplications; we overview several approaches in Appendix B.

In practice however, SealPIR (and XPIR) only use the additive homomorphism of the underlying scheme. This is explained in SealPIR by the significantly higher computational

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

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.

cost of homomorphic multiplications compared to homomorphic additions [5, Sec. 3.1]. Indeed, for the databases considered in [5], the communication complexity $O\big(F^{d-1}c(n)\big)$, where F is the ciphertext expansion, remains suitable for many applications and offers excellent performance. For databases with large elements, however, Table 1 shows that the large PIR expansion yields unacceptable download communication. This is the setting we focus on in this section.

We introduce MulPIR, a variant of SealPIR with the optimizations above, which further replaces the emulated multiplications with homomorphic multiplications during recursion (recursion is described in Section 2.3; and we provide a full description when using homomorphic multiplications in Appendix B.2). Therefore, MulPIR *trades* off computation (higher computational costs for the server) with smaller communication *for databases with large entries* (in total communication, and more particularly for the download communication). In particular,

- The MulPIR.Query algorithm is given in 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 Appendix B.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.

On the communication front only, we report the communication costs compared to SealPIR in Table 1. Our experiment sections (Section 6) will quantify the impact of using MulPIR in practice, by reporting both its concrete communication and computation costs.

4 Improving Gentry–Ramzan PIR

An alternative to PIR based on homomorphic encryption is the protocol of Gentry and Ramzan [30], which achieves logarithmic communication and a constant communication rate. While it has been implemented in previous work [15, 16, 49], it is usually dismissed due to its computational complexity [3, 15].

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 the original protocol [30] (Sec. 4.1). Then, in Sec. 4.2, we show how to apply existing techniques [7, 58] 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 Sec. 4.3, we show how to speed up the response computation with a novel *client-aided* variant of Gentry–Ramzan, using the fact that the client can perform modular exponentiations more efficiently since she knows the order of the multiplicative group.

4.1 Gentry–Ramzan PIR

The basic PIR protocol of Gentry and Ramzan [30] works by interpreting the server's database as a number in a Residue Number System (RNS). That is, given n co-prime 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

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

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 [30] 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 [30] for the details.

4.2 Fast Modular Interpolation

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

Procedure 8 PIR.GR.Query

```
Parameters: security parameter $\lambda$.
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 $\log_2(Q_2) \geq \lambda$.
    $m := Q_1 Q_2$.
    $g \leftarrow\$ \mathbb{Z}_m$ 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 $\mathsf{D}$ as an integer $E$ as in Eq. (2).
    $g' := g^E \bmod m$.
Output: $g' \in \mathbb{Z}_m^*$.
```

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:

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 [58] and the extended Euclidean algorithm [59], 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 [7]. 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 [58], has a total running time of $O(n \log^2 n \log\log n)$ [7]. 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).

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: $d \in \mathbb{Z}_{\pi_i}$.
```

4.3 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 [16] 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 [61], a description of which can be found in [38, Alg. 14.88]. In our experiments (Table 5 and Fig. 3), we show how sending more generators significantly reduces the server computation time.

5 Sparse Databases

The traditional setting for PIR over a database of size n assumes that each database element has a unique index in $[n]$

known to the client, which is used to create a query. However, in some scenarios, such dense indices are not immediately available, and database elements are instead indexed by *keywords* from a much larger domain. This sparse database setting has been considered as *keyword PIR* by Chor et al. [12]. The latter work builds an efficient search data 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 proportional to the number of sparse items. We propose a new construction based on hashing that reduces the overhead to a constant number of PIR queries.

A straight-forward way to use hashing for sparse PIR is to let the server map its $n = |\mathsf{D}|$ elements to a set of m bins using *simple hashing*. That is, for each pair $(i,d) \in \mathsf{D}$, the server inserts d into the bin number $\mathsf{H}(i)$, where H is a public hash function. For a query index i', the client then retrieves bucket $\mathsf{H}(i')$ using PIR. Despite its simplicity, this scheme has the drawback that the size of the buckets grows asymptotically with the number of items n. While this works well with PIR schemes that have a large plaintext size (such as MulPIR), for other schemes (such as Gentry–Ramzan) we ideally want the bucket size to be a small constant. To achieve this, we will instead use *cuckoo hashing* on the server side, which ensures that each bucket only contains a single database element.

Cuckoo hashing [47] has been used for private set intersection [11, 20, 53, 54, 56], in particular asymmetric PSI [11] and as a multi-query batching technique for PIR [5]. In these works, the client (i.e., the party holding the smaller set of elements or queries) uses cuckoo hashing to map its inputs to a set of buckets held by the server, such that each client input only needs to be compared against server elements inside the corresponding bucket. Our approach leverages cuckoo hashing in a different way that is similar to some of its uses as a building block for ORAM constructions [40, 52]: we apply cuckoo hashing on the server side, to compress the domain of its indices. Unlike PSI and ORAM, we don't need to provide privacy for the server's database in PIR. This allows us to guarantee correctness, since the server can just choose different hash functions in case cuckoo hashing fails. In the following, we therefore assume that hash functions are chosen dependent on the database D.

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. We formalize this dependence of the hash functions using a data-dependent key generation procedure $\mathsf{Cuckoo.KeyGen}(\mathsf{D})$.

We present a PIR construction which works as follows. The server generates cuckoo hash functions using the data dependent key generation $\mathsf{Cuckoo.KeyGen}(D)$ and builds a cuckoo hash table for its sparse database using the insertion algorithm $\mathsf{Cuckoo.Insert}$, which will be of size proportional to the number of non-empty entries (with a constant multiplicative overhead). The server provides the cuckoo hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_\kappa$ for a $\kappa \geq 2$. 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 stress again 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.2 and Angel et al. [5]. We now present the formal construction for PIR on sparse data.

Construction 1. Let $(\mathsf{Cuckoo.KeyGen}, \mathsf{Cuckoo.Insert})$ be a cuckoo hashing scheme and $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ be a PIR protocol. We construct a new PIR protocol $(\mathsf{PIR'.Query}, \mathsf{PIR'.Eval})$ 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 $\{\mathsf{H}_j\}_{j\in[\kappa]}$.
- $q_i = (q_i^1, \ldots, q_i^\kappa) \leftarrow \mathsf{PIR'.Query}(i)$: The client computes $q_i^j \leftarrow \mathsf{PIR.Query}(\mathsf{H}_j(i))$ for $j \in [\kappa]$.
- $[\mathsf{D}[i], \perp] \leftarrow \mathsf{PIR'.Eval}([q_i, \mathsf{D}])$: The client and the server run $[\mathsf{T}_j[\mathsf{H}_j(i)], \perp] \leftarrow \mathsf{PIR.Eval}([q_i^j, \mathsf{T}_j])$ 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 $\perp$.

Theorem 1. *Let* $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ *be a PIR protocol that provides correctness and query privacy. Then Construction 1 provides correctness and query privacy.*

Proof. The correctness of the above scheme is guaranteed by the correctness of the cuckoo hash, which guarantees that an item with an index i will be located in one of the positions determined by the hash functions, and the correctness of $\mathsf{PIR.Query}$, which returns the respective items. The privacy of the query is guaranteed by the privacy of $\mathsf{PIR.Query}$ and the fact that we only make a constant number κ of queries. □

The query efficiency of the above construction depends only on the size of the sparse database and the number of cuckoo hash functions. The latter dependency can be removed applying multi-query PIR techniques. In Section 6.3, we will apply our sparse PIR constructions to a secure password checkup problem, using cuckoo hashing for Gentry–Ramzan PIR, and simple hashing for MulPIR. We will see that both approaches yield different tradeoffs in terms of communication and computation.

6 Experimental Evaluation

We present experimental results that measure the efficiency of different PIR protocols and illustrate some of the possible trade-offs 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. Monetary costs were computed using Google Cloud Platform prices [1], which at the time of writing were at one cent per CPU-hour and 8 cents per GB of internet traffic.

6.1 SealPIR and MulPIR

First, we report on the relative costs of SealPIR and MulPIR. For SealPIR, we use the parameters of [5]: polynomials of dimension 2048 and a modulus of 60 bits, providing 115 bits of security. The plaintext modulus has a size of 12 bits ($d = 2$) / 16 bits ($d = 3$). For MulPIR, we use polynomials of dimension 8192 and a modulus of 160 bits, providing 180 bits of security. The plaintext modulus size is set to 21 bits ($d = 2$) / 17 bits ($d = 3$). We use the SealPIR implementation available on Microsoft's GitHub [3] based on Seal 3.2.0, and we implement MulPIR with Seal 3.5.4 [2].

The first database is a "Pung-style" database, as used in [5]. This is a database of $n = 2^{18}$, 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 ($d = 2$) or $\lceil n/14 \rceil$ entries of 4032B ($d = 3$), to *fully pack* each ciphertext. Similarly, MulPIR reshapes the database into a database of $\lceil n/71 \rceil$ entries of 20448B (for $d = 2$), or $\lceil n/56 \rceil$ entries of 16128B (for $d = 3$). We present the communication and computation comparison in Table 3. As expected, the communication is smaller than the implementation of SealPIR from [5] for a slightly larger computational cost. For most database sizes, this also results in the lowest monetary server cost. Finally, we observe that $d = 3$ doesn't improve either communication or computation of MulPIR or SealPIR, due to the fact that the upload for $d = 2$ already consists of only a single ciphertext.

We note that re-implementing 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. To better visualize the communication–computation trade-off, we also benchmark MulPIR, and estimate the communication and computation of the optimized version of SealPIR from Section 3.2[4], 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, which also results in a reduction of the monetary server costs by 35%.

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 ElGamal 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 [4]), 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 [5] 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. 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 ElGamal, 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 block size 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]. As expected, Damgård–Jurik and ElGamal are significantly

[4] For a fair comparison, we omit the computation-expensive Remark 1 here, since it could make SealPIR more costly than MulPIR and requires careful benchmarking.

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

	SealPIR [3] ($d = 2$)			SealPIR [3] ($d = 3$)			MulPIR ($d = 2$)			MulPIR ($d = 3$)		
Database size n	262144	1048576	4194304	262144	1048576	4194304	262144	1048576	4194304	262144	1048576	4194304
Actual number of rows after packing	*26215*	*104858*	*419431*	18725	74899	299594	*3693*	*14769*	*59075*	*4682*	*18725*	*74899*
Client Query	19	19	19	19	19	19	172	192	213	126	128	161
Server Expand	145	294	590	33	55	90	391	783	1610	396	395	841
Server Respond	1020	3520	12891	1136	3519	11554	1919	5213	16307	3268	11677	30501
Upload (kB)	61.4	61.4	61.4	92.2	92.2	92.2	122	122	122	130	130	130
Download (kB)	307	307	307	1966	1966	1966	119	119	119	130	130	130
Server Cost (US cents)	0.0033	0.0040	**0.0067**	0.017	0.017	0.020	**0.0026**	**0.0036**	0.0069	0.0031	0.0054	0.011

Table 4: Communication and CPU costs of SealPIR and MulPIR for a 4GB database with 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
Server Cost (US cents)	0.028	**0.018**

slower than MulPIR and Gentry–Ramzan. 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 [42, 62] 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 [62]) remains prohibitively large for PIR. Instead, [42, 62] 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, [62] 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 [62, 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 Section 5. We use the parameters from [20, Appendix B] with 3 hash functions. Note that the three client queries can be batched into a single Gentry–Ramzan query using CRT batching (see [35] and Appendix E.2). 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 [62]. In Fig. 3, we illustrate the communication-computation trade-off 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 low client and server computation costs. However, with 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. 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 use the approach based on simple hashing (cf. Section 5): the server selects a random 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 $\mathsf{H}(e)$. 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 [57, 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

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

		Communication (kB)		Computation (ms)					Server Cost (US cents)
	# chunks	upload	download	C.Setup	S.Setup	C.Create	S.Respond	C.Process	
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	≈ 80,000	≈ 3,200	≈ 28600	≈ 42,000,000	≈ 2,500	11.7451
ElGamal	76,800	280	4,300	≈ 300	≈ 88,800	≈ 2250	≈ 4,800,000	≈ 30,715,200	1.4338

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

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

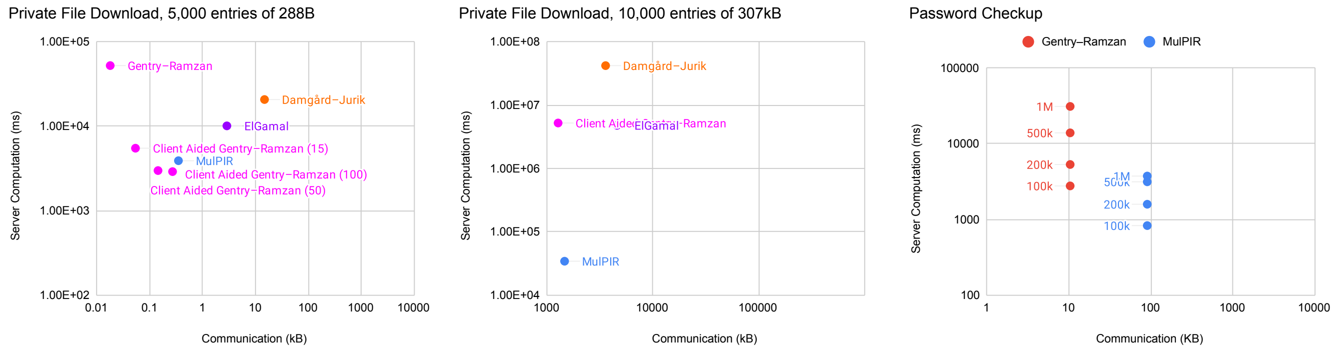
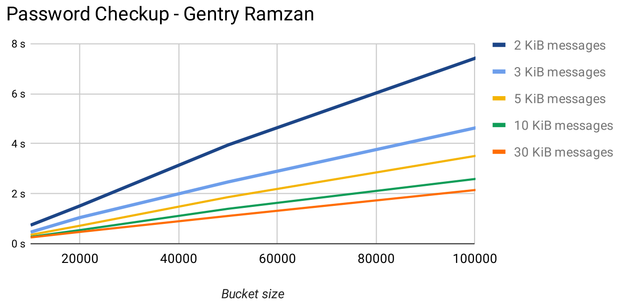


Figure 3: Computation time in the Password Checkup application when using client-aided Gentry–Ramzan.



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 [62]. In terms of total server cost, we observe that MulPIR outperforms Gentry–Ramzan in our experiments as soon as the bucket size is 200k or more.

Table 6: Password Checkup application.

	Gentry–Ramzan				MulPIR			
Bucket size	Com. (kB)	Client (ms)	Server (ms)	Server (US ¢)	Com. (kB)	Client (ms)	Server (ms)	Server (US ¢)
10k	10.4	24,324	317	**0.00017**	90.5	156	475	0.00086
20k	10.4	19,888	573	**0.00024**	90.5	189	515	0.00087
50k	10.4	24,906	1,649	**0.00054**	90.5	195	810	0.00095
100k	10.4	30,644	2,774	**0.00085**	90.5	195	830	0.00095
200k	10.4	21,571	5,318	0.0016	90.5	236	1,588	**0.0012**
500k	10.4	53,137	13,913	0.0039	90.5	285	3,143	**0.0016**
1M	10.4	49,819	31,055	0.0087	90.5	265	3,742	**0.0018**

Overall, our protocols respectively can check a single password with 10.4 KB or 90.5 KB communication. This is in contrast with prior work [11], which is optimized for the batched setting. For the smallest batch size of 256, Chen et al. [11] report communication of 17.6 MB. See also Appendix A.

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 different HE schemes as well as the Gentry–Ramzan PIR and a new approach to handle database sparsity. We evaluate our protocols on various applications ranging from private messaging, file downloads, and password checkup. Our evaluation shows that our improved MulPIR and client-aided GR implementations significantly improve the state of the art, resulting in the lowest dollar cost in most settings.

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

[5] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Casey Devet and Ian Goldberg. The best of both worlds: Combining information-theoretic and computational pir for communication efficiency. In *Privacy Enhancing Technologies*, 2014.

[22] 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.

[23] 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.

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

[25] 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.

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

[27] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.

[28] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

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

[30] 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.

[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] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

[39] 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.

[40] E. Kushilevitz, Steve Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In *SODA*, 2012.

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

[42] 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.

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

[44] 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.

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

[46] 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.

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

[48] 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.

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

[50] Jeongeun Park and Mehdi Tibouchi. SHECS-PIR: somewhat homomorphic encryption-based compact and scalable private information retrieval. In *ESORICS (2)*, volume 12309 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.

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

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

[53] 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.

[54] 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.

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

[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] 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.

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

[59] 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.

[60] 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.

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

[62] 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.

[63] 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, 17, 22, 23, 29, 34, 45, 63] 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 [60]: 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 [5]). This has limited the recursion depth to $d \leq 3$ in practice [5, 45].

Along this line of work, there are several papers that present implementations with various resource trade-offs. Aguilar-Melchor et al. [45] present XPIR with small computation costs but quite large communication costs. On the other hand, another line of work [39, 44] obtain much smaller (almost optimal) communication at the cost of significantly larger computation. In a recent work, Angel, Chen, Laine, and Setty [5], 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 [39, 44]). SealPIR is instantiated with the FV (lattice-based) homomorphic encryption scheme [24]. It builds upon XPIR [45, 60] 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. In a work concurrent to ours, Park and Tibouchi [50] present a construction that uses GSW-style homomorphic encryption that support logarithmic multiplicative degree and achieves $O(\log n)$ communication. Compared to SealPIR, their approach offers a similar trade-off as ours, i.e., a reduction of the communication by 80% at the cost of increased computation time [50, Table 5]. The work of Devet et al. [21] introduces a hybrid model between computational and information theoretic PIR, which allows graceful degradation of query privacy when the database servers are colluding, and leverages CPIR recursion techniques to improve communication efficiency.

Another known PIR construction that achieves logarithmic communication complexity is the construction of Gentry–Ramzan [30], which does not rely on homomorphic encryption. This PIR construction extends the idea from the work of Cachin et al. [9] 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. [9] handled only binary data items, and the Gentry–Ramzan construction [30] 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 [15, 16, 49].

In recent years, single server PIR has also been studied in slightly different settings. Two works [8, 10] 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 [51] 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. [36] 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 [5] 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* [12], considers the setting where the database size is much smaller than its index domain. Chor et al. [12] 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. In contrast, our approach based on cuckoo hashing only incurs a constant overhead. Another approach to PIR on sparse databases is given by PSI protocols. In particular, Chen et al. [11] present a *labeled PSI* protocol that has communication sublinear in the server's dataset. However, they optimize for the multi-query setting, and therefore their protocol is not directly suited to the applications we consider (Section 6.3). For example, the smallest number of batched queries reported in [11] is 256, where the protocol uses 17.6 MiB communication. In contrast, our protocol can handle single queries for as little as 10 KiB (see Table 6).

Symmetric PIR (SPIR) [33] 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 [46] provided general transformation from PIR to SPIR using oblivious polynomial evaluation, and there have also been direct constructions [41, 43].

B PIR using Additive and Multiplicative Homomorphisms

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 [27]: 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). Appendices B.1 to B.3 presents 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. 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.

B.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 $1-(b_1+b_2-2b_1b_2)$ 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.

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.

B.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.

B.3 Selection Vector Reconstruction

Note that the approach of Section B.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[5], 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.

C Correctness of Query Expansion

Below we prove that the combination of the new query and oblivious expansion algorithm (Algorithms 6 and 7) correctly expands the ciphertexts into a vector of n ciphertexts encrypting the selection vectors.

Theorem 2. *Let n be an integer, N be a power of 2, $d \in [1, \log n]$, $c \in [0, \log_2 N]$. Let k be an index in $[1, n]$, and $\vec{q} = (q_j)_{j\in\ell}$ the output of the* Query *algorithm (Algorithm 6). Denote $\vec{s} = (s_i)_{i\in[n]} \in \{0,1\}^n$ the concatenation of the d selection vectors for index k. The n output ciphertexts $o_0, \ldots, o_{n-1}$ of the expansion algorithm (Algorithm 7) on input $\vec{q}$ satisfy, for all $0 \le i \le n-1$:*
$$o_i = \begin{cases} Enc(1) & \text{if } s_i = 1 \\ Enc(0) & \text{otherwise} \end{cases}.$$

Proof. It suffices to prove the claim for the first element of the query. By construction q_0 encrypts $m_0 = \sum_{i\in[2^c]} (2^{-c} \bmod t) s_i x^i$. Now, simplifying the notation for ease of exposition, since the encryption scheme is homomorphic, we have that
$$q_0 = \mathsf{Enc}(\mathsf{sk}, m_0) = (2^{-c} \bmod t) \sum_{i\in[2^c]} \mathsf{Enc}(\mathsf{sk}, s_i x^i)\,. \tag{4}$$

Now, we remark that the addition, subsection, and scalar multiplications are linear over the plaintext space. Henceforth, the output of Algorithm 7 is the sum of the outputs of Algorithm 7 over the $\mathsf{Enc}(\mathsf{sk}, s_i x^i)$'s. Now, consider such a plaintext $m' = x^{i'}$: this is exactly the form of a SealPIR query. Now, observing that the core loop of the expansion is the same as in Algorithm 5, we can use the exact same arguments as the proof of correctness of the oblivious expansion in SealPIR (cf. [5, App. A.2]). It follows that the output of Algorithm 7) on $\mathsf{Enc}(\mathsf{sk}, m')$ is a vector $\vec{o}' = (o'_0, \ldots, o'_{2^c-1})$ of size 2^c such that
$$o_i = \begin{cases} Enc(2^c) & \text{if } i = i' \\ Enc(0) & \text{otherwise} \end{cases}.$$

The result then follows directly from Eq. (4). □

[5] 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.

D 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 [26], Paillier/Damgård–Jurik [18], and FV [24], the constructions of which we overview next.

Additive ElGamal [26]. 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 [18]. Let $N = pq$ be an RSA modulus. The Damgård–Jurik generalization of the Paillier cryptosystem [48] 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.

FV [24]. 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.

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 [30, p. 808].

However, since FV is also somewhat (and fully) homomorphic, we can apply the PIR protocols of Appendix B. 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.

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 Symmetric PIR from OPRFs

The security requirements of a PIR protocol pertain only to the privacy of the query. Symmetric private information retrieval (SPIR) [32] 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) [25, 37]. 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

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 $\mathbb{Z}_q[x]/(x^{\lambda_{\mathsf{FV}}} + 1)$

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 this approach also works together with our techniques for handling sparsity from Section 5, even if the cuckoo hashing parameters depend on the server's database. The reason is that the client can only evaluate the OPRF on values used as a query. Thus, any leakage of OPRF values of other database elements can be simulated due to the fact that OPRF values look pseudorandom to the client. This is in contrast to uses of cuckoo hashing on the client side [5], where data-dependent hash functions leak information to the server, and therefore the failure probability of cuckoo hashing must be considered in the security analysis.

E.2 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. Our constructions are compatible with the multi-query techniques available in the literature. In our implementation, we leverage probabilistic batch codes based on cuckoo hashing [47], which have been used in the context of PIR [5] and private set intersection [20, 55], as well as a CRT batching technique introduced by Groth et al. [35] for Gentry–Ramzan.

E.3 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.