Communication–Computation Trade-offs in PIR

Asra Ali*, Tancrède Lepoint*, Sarvar Patel*, Mariana Raykova*, Phillipp Schoppmann†, Karn Seth*, Kevin Yeo*
* Google, {asraa,tancrede,sarvar,marianar,karn,kwlyeo}@google.com
† Humboldt-Universität zu Berlin, schoppmann@informatik.hu-berlin.de

December 24, 2019

Abstract

In this work, we study the computation and communication costs and their possible trade-offs in existing constructions for private information retrieval (PIR), including schemes based on homomorphic encryption and the Gentry–Ramzan PIR (ICALP'05). We present MulPIR, a PIR construction which uses somewhat homomorphic encryption in a new way that provides a better trade-off between communication and computation, and for the first time enables the implementation of PIR with full recursion. Our construction leverages optimizations from SealPIR (S&P'18) and extends them with new packing and compression techniques. We also present improvements for the Gentry–Ramzan PIR that reduce significantly the computation cost, the main overhead for this scheme, which achieves optimal communication in several settings. We further show how to efficiently construct PIR for sparse databases. Our constructions support batching for multi-queries as well as symmetric PIR. We finally implement several PIR constructions and present a comprehensive comparison of their communication and computation overheads, as well as a cost analysis assuming a standard price setting for CPU power and bandwidth.

I. Introduction

Access to public data often brings privacy concerns for the querier as it may reveal sensitive information about this party. For example, queries of medical data can reveal sensitive health information and patterns of accessing 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 information (such as their queries), as it could later become liability for them.

Private information retrieval (PIR) is a cryptographic primitive that aims to address the above question by enabling clients to query a database without revealing any information about their queries to the data owner. While the feasibility of this primitive has been resolved for a long time [CKGS98], the search for concretely efficient constructions for practical applications has been an active area of research [YKPB13], [DC14], [DSH14], [DDS15], [MBFK16], [AS16], [GLM16], [ACLS18], [GH19]. In this context, there are several parameters and efficiency measures that characterize a PIR setting and determine what solution might be most suitable for a particular scenario. However, a baseline solution that candidate PIR solutions should improve on is the trivial PIR that returns the whole database to the client.

In 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 [KO97b], 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 [DHS14]. While theoretical constructions for PIR [KO97b] achieve polylogarithmic communication, most efficient single server PIR implementations stop short of this goal and implement only variants of the construction with higher asymptotic communication costs [MBFK16], [AS16], [GLM16], [ACLS18].

In this paper, we analyze the communication–computation trade-offs that different PIR construction approaches offer and the hurdles towards achieving the optimal asymptotic communication costs in practice. We present a new PIR construction using somewhat-homomorphic encryption that improves the communication and computation costs of recursion in existing PIR schemes, enabling for the first time implementation measurements with recursion level beyond three. While the classic PIR setting is our starting point, we extend our solutions to the symmetric variant of PIR, and the batching setting of executing multiple PIR queries in parallel. We also propose a new construction for sparse databases, a.k.a, keyword PIR [CGN97], where the number of database entries is much smaller than the query key domain. Our solution reduces the server cost to be proportional only to the actual database size and be independent of the key domain size. In each of the above settings we analyze the communication/computation trade-offs depending on the shape of the underlying database determined by the number of database entires, and the size of each entry. We consider three example applications, which involve two opposite database shapes, and evaluate the performance of our schemes in these settings. The first one is PIR over a small set of large text files, the second one is PIR over a large set of small entries, and finally the third one is private set membership. Our batched multi-query techniques for PIR further provide solutions for the problems of private set intersection (PSI) and PSI cardinality, which bring communication improvements in the case of sets of asymmetric sizes.

A. *Background*

a) Efficient Constructions of Single Server PIR: The most efficient (secure) single server PIR constructions implemented in the recent years [YKPB13], [DC14], [DSH14], [DDS15], [MBFK16], [AS16], [GLM16], [ACLS18], [GH19] are based on homomorphic encryption (HE) techniques and achieve sublinear 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. Sublinear complexity is achieved by using recursion [Ste98]: 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 [ACLS18]). This has limited the number of recursion layers in practice to $d \leq 3$ [MBFK16], [ACLS18].

Along this line of work, there are several papers that present implementations with various resource tradeoffs. Aguilar-Melchor et al. [MBFK16] present XPIR with small computation costs but quite large communication costs. On the other hand, another line of work [KLL$^+$15], [LP17] obtain much smaller (almost optimal) communication at the cost of significantly larger computation. In a recent work, Angel, Chen, Laine, and Setty [ACLS18], present SealPIR that strikes a better balance in the communication–computation cost. SealPIR requires only slightly more computation than XPIR but uses almost 1000 times less communication than XPIR (but does not achieve the almost optimal rate of the works [KLL$^+$15], [LP17]). SealPIR is instantiated from the FV (lattice-based) homomorphic encryption scheme [FV12] and builds upon XPIR [Ste98], [MBFK16] 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. However, the recursion technique used in SealPIR still suffers from the exponential number of ciphertexts in the query response.

Another known PIR construction that achieves logarithmic communication complexity is the construction of Gentry–Ramzan [GR05], which does not rely on homomorphic encryption. This PIR construction extends the idea from the work of Cachin et al. [CMS99] which proposes to encode the database $\{\mathsf{D}_i\}_{i\in[n]}$ using the Chinese Remainder Theorem (CRT) representation as $x \in [n]$ s.t. $x \equiv \mathsf{D}_i \mod \pi_i$ 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 $q\pi_i$. 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. [CMS99] handled only binary data items, and the Gentry–Ramzan construction [GR05] shows how to handle larger plaintext domain for the database entries and improves the communication rate to constant. While the resulting construction achieves optimal 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 overhear this PIR construction has been rarely considered as a candidate for implementation and practical applications [PBP12].

In recent years, single server PIR has also been studied in slightly different settings. Two works [CHR17], [BIPW17] 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 [PPY18] 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.

b) Handling Multiple Queries: There are PIR settings where multiple queries need to (or can) be executed at the same time, and batching techniques enable parallel execute of such queries with smaller amortized cost. One application of batched-PIR queries are constructions for private set intersection (PSI) in the asymmetric setting [CHLR18], [CLR17], [DRRT18a], [KRS$^+$19].

The observation that executing multiple PIR queries in parallel can be achieved at a lower communication and computation cost was made in the work of Ishai et al. [IKOS04] who proposed a construction based on batch codes, which gives asymptotic guarantees but has remained impractical for real implementations. Inspired by the use of Cuckoo hashing in the context of private set intersection, the work on SealPIR [ACLS18] proposed a batched PIR construction leveraging a probabilistic batch code, which amortizes CPU cost while introducing a small probability of failure ($\approx 2^{-40}$). Groth et al. [GKL10] proposed a batched multi-query version of the Gentry–Ramzan PIR. This construction leverages the fact that the group order of the client query can encode multiple query indices by being divisible by the corresponding primes. This technique saves communication and server's work, but imposes further restrictions on the database entries' size.

c) PIR for Sparse Databases: Standard PIR settings assume that the server database has an entry for each possible index. The complexity of PIR protocols in this setting is proportional to the size of the index space. However, there are scenarios where the underlying data is sparse, i.e., the database size is much smaller than its index domain. This setting is also known as *keyword PIR* [CGN97]. The technique proposed by Chor et al. [CGN97] is to build a binary search tree over the items in the database and to use a separate PIR instance for each tree level. A PIR query on the database consists of a logarithmic number PIR queries for the tree levels. The amortized multi-query PSI techniques [CO18], [CHLR18], [PSTY19] could be viewed as solutions.

d) Symmetric PIR (SPIR): SPIR [GIKM00] 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 [NPP99] provided general transformation from PIR to SPIR using oblivious polynomial evaluation, and there have also been direct constructions [KO97a], [Lip05].

B. Our Contributions

In this paper we analyze the exact trade-offs between communication and computation in the context of PIR, and we study the best communication complexity that we can achieve in practice. We present a new approach for using HE in PIR constructions that enables new communication–computation trade-offs. We also implement and optimize the PIR protocol of Gentry and Ramzan [GR05], providing the first implementation of this protocol that scales to databases with millions of elements. Together, our protocols provide various *practical* tradeoffs between computation and communication, which we experimentally evaluate using three application scenarios.

a) Communication–Computation Trade-offs in HE-based PIR: As we discussed above, the work of Angel at al. [ACLS18] proposed compression techniques that enable them to pack selection vectors in the slots of a homomorphic ciphertext and thus they achieve upload communication of $O(d\lceil n^{1/d}/N\rceil)$ for a database of size n using HE with modulus size N and PIR recursion level d. While this allows to decrease the upload cost by increasing the recursion level d, the download communication depends exponentially on the recursion level. The reason for this is the PIR selection algorithm used in this work, which uses layers of HE encryption to implement the partial database selection in different recursion layers. This leads to an explosion of the parameter sizes needed for the outermost encryption.

We propose a different approach (MulPIR) that leverages both the additive and the *multiplicative* homomorphism of HE to implement the recursive selection by doing one multiplication of encrypted values per recursion step. This reduces the size of the upload and download together from $O(d\lceil n^{1/d}/N\rceil + F^{d-1})$ in existing approaches, where F is the number of plaintexts needed to fit a single HE ciphertext, to $d \cdot \lceil n^{1/d}/N \rceil \cdot c(d)$, where $c(d)$ is the size of a ciphertext that supports d multiplications (hence, a depth of $\log d$). In addition to the technique leveraging homomorphic multiplication, we propose a new compression technique that allows us to pack multiple selection vectors in the same ciphertext and this reduces further the communication cost. Inspired by optimization techniques in recent proposals of post-quantum secure encryption schemes [ADPS16], [BDK$^+$18], we apply "bit dropping" techniques, which guarantee that a ciphertext can decrypt correctly even when omitting some of the least significant bits that account for the noise, to further optimize the concrete communication of our scheme.

Our new techniques also allow us to achieve the ideal asymptotic communication complexities for PIR in practice and enable for the first time implementation experiments with recursion level beyond three.

b) Gentry–Ramzan PIR: The Gentry–Ramzan PIR construction [GR05] achieves optimal communication complexity for several settings but it pays with significant computational cost. Thus, our contributions focus on ways to reduce this computation overhead, which includes new efficient techniques for encoding the server 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 query response that trade-off between communication and computation overhead.

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 coprime integers. A naive application of the Chinese Remainder Theorem requires computation at least quadratic in the size of the database. We leverage a divide-and-conquer modular interpolation algorithm [BM74] that enables us to achieve computation complexity $\tilde{O}(n \log^2 n)$. This technique also allows for pre-computation that can be reused for computations that use the same set of moduli π_i.

The main computation cost on the server side is the modular exponentiation, where the server cannot know the prime factorization of the modulus and thus we cannot use techniques that leverage the factorization to speed-up the computation of the exponentiation. Our approach will be to compute the exponentiation as a product of precomputed powers of the generator and to use Straus's algorithm [Str64] to do this efficiently. Further we notice that since the precomputed powers of the generator are independent of the exponent, they can be computed at the client who knows the order of the group that it is using for the PIR query and thus can compute exponentiation in this group faster by first reducing the exponent modulo the order of the group. This gives a way to trade-off computation and communication complexity for the protocol. In Section VI, we show evidence that providing several precomputed powers optimizes the server's work.

To implement private set membership, we also apply batching techniques leveraging probabilistic batch codes from Cuckoo hashing [PR04], which provide better scalability for broader sets of the database parameters compared to previous batching approaches for Gentry–Ramzan PIR [GKL10].

c) New Construction for Sparse PIR: We present a new PIR construction for sparse databases, which provides the client with an answer that either contains the corresponding data if the element is present in the database or is empty, otherwise (see Appendix C). Our construction leverages Cuckoo hashing [PR04] in a new way inspired by ideas from private set intersection [CHLR18], [PSSZ15], [PSWW18], [DRRT18b] and oblivious RAM [PR10], [GMOT12], [PPRY18]. In particular, we observe that we can compress the domain of the database from a large sparse domain to a small dense domain using Cuckoo hashing, which in comparison to regular hashing distributes the items in the hash tables guaranteeing that no collisions occur. To implement a PIR construction that does not require any privacy for the database, we can just have the client run as many PIR queries on the new dense database as the number of hash functions used by the Cuckoo hashing. We achieve a symmetric variant of the sparse PIR construction that reveals only the sparsity of the server's database (without revealing the empty locations). For this we leverage Cuckoo hashing with a stash [KMW09] where the parameters for the scheme are guaranteed to be independent of the data that is hashed and insertion failures occur with negligible probability. In this case the client executes SPIR queries for each location, where the communication of any part of the database that may need to be checked at each query, e.g. a Cuckoo stash, can be amortized across queries.

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

We evaluate our new PIR construction, MulPIR, that uses somewhat-homomorphic encryption (SHE) and compare it against SealPIR. MulPIR enables a trade-off of computation for communication, which allows us reduce the communication of SealPIR by 80% while increasing the computation roughly twice. We also provide the first empirical evaluation of PIR with recursive level beyond three. Surprisingly, we observe that higher recursion level does not necessarily improve communication. This is due to the fact that lattice-based HE encryptions have a complex relationship between parameters sizes, support for homomorphic operations and number of encryption slots. In this context recursion improves complexity when the database size increases beyond the number of encryption slots in a ciphertext, however, at the same time increasing the database size requires support for more homomorphic operations, which leads to larger parameters.

In our experiments, Gentry–Ramzan construction 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, which for example is the case in the SealPIR application we consider [AS16].

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 [TPY$^+$19] 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 Genry–Ramzan and MulPIR enable such lookups to be *sub-linear* in k, therefore either enabling better anonymity for the same bandwidth, or same anonymity and smaller bandwidth.

II. Background

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, each at most l bits long.

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.

A. Private Information Retrieval (PIR)

Definition 1 (Private Information Retrieval [CKGS98]). *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 client has an input index i. The goal of the protocol is to enable the client to learn D_i while guaranteeing that the server does not learn anything about i. A PIR scheme is specified with the following two algorithms:*

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

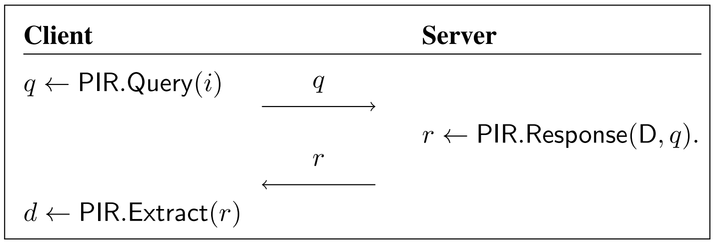

Client
Server
$q \leftarrow \mathsf{PIR.Query}(i)$
q
$r \leftarrow \mathsf{PIR.Response}(\mathsf{D}, q)$.
r
$d \leftarrow \mathsf{PIR.Extract}(r)$

Fig. 1. A non-interactive PIR protocol. At the end of the protocol, the correctness of the PIR protocol will ensure that $d = \mathsf{D}_i$.

Definition 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*.*

B. Homomorphic-Encryption-based PIR

In this section, we recall the baseline PIR solution based on additive homomorphic encryption [KO97b]. For ease of presentation and w.l.o.g., recall that an additive homomorphic encryption scheme $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ with plaintext space $\mathbb{Z}_t$ is an encryption scheme with the following properties:

- $\mathsf{Enc}(\mathsf{sk}, m_1) + \mathsf{Enc}(\mathsf{sk}, m_2) = \mathsf{Enc}(\mathsf{sk}, (m_1 + m_2) \bmod t)$,
- $\mathsf{Enc}(\mathsf{sk}, m_1) \cdot \lambda = \mathsf{Enc}(\mathsf{sk}, m_1 \cdot \lambda \bmod t)$,

for every $m_1, m_2, \lambda \in \mathbb{Z}_t$, for some specific operations $+$ and $\cdot$ over the ciphertexts.

Let $t \geq 2^l$ and $\mathcal{HE} = (\mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ be a homomorphic encryption scheme with plaintext space $\mathbb{Z}_t$ and ciphertext space $\mathcal{C}$. The PIR protocol works by interpreting each element D_i as an element of $\mathbb{Z}_t$. To construct the query for index k, the client encrypts component by component the *selection vector* $\vec{s} = (s_i)_{i=1\ldots n}$ proportional to the size of the database n, which verifies $s_i = \delta_{i,k} = 0$ for $i \neq k$ and $s_k = \delta_{k,k} = 1$. To answer the query $q = (\mathsf{Enc}(\mathsf{sk}, s_i))_{i=1\ldots n}$, the server computes the inner product between the query and the database D (where $\mathsf{D}_i \in \mathbb{Z}_t$), eventually yielding

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

We give the description of the HE-based PIR protocol in Procedures 1 to 3. In the rest of the paper, we will instantiate this protocol with the Paillier/Damgård–Jurik cryptosystem [Pai99], [DJ01], the El-Gamal cryptosystem [Gam85], and an RLWE-based homomorphic cryptosystem [FV12], [MBFK16], [ACLS18].

C. Gentry–Ramzan PIR

In this section, we recall the baseline PIR protocol by Gentry and Ramzan [GR05]. It works by interpreting the server's database as a number in a Residue Number System (RNS). That is, given n coprime integers $\pi_1, \ldots, \pi_n$,

---

Procedure 1 $\mathsf{PIR.HE.Query}$

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

Procedure 2 $\mathsf{PIR.HE.Response}$

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

---

with $\pi_i \geq 2^l$ for all $i \in [n]$, we encode D as an integer E, such that

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

The existence and uniqueness of E follows from the Chinese Remainder Theorem, which can also be used to compute E given D and all π_i. Observe that (2) implies that we can retrieve the element at index i by reducing E modulo π_i. The idea of [GR05] 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 4 to 6.

Remark 1. *An advantage of the Gentry-Ramzan PIR protocol is its constant communication rate ($c < 1/4$ in practice), which comes at the cost of increased server computation time. In Section V-B we introduce a extension to Gentry-Ramzan that we dub* Client-Aided PIR*, which allows users to trade off between computation and communication costs.*

III. Homomorphic Encryption-Based PIR

The majority of private information retrieval constructions that achieve sublinear communication rely on homomorphic encryption in order to enable the client to compress its query.

We start from the baseline PIR and survey the two flavors of homomorphic-encryption-based PIR protocols with sublinear communication that exist in the literature, respectively based on additive homomorphic encryption (AHE) schemes and fully homomorphic encryption (FHE) schemes. Then, we propose a third flavor that exploits the homomorphic multiplication in a somewhat homomorphic encryption (SHE) scheme to trade-off computation and communication. We summarize the communication and computation costs for different PIR constructions in Table I.

Finally, in Appendix A, we discuss the instantiations of the above approaches with existing HE schemes and report on the specific HE costs for different schemes in Table VII.

A. Baseline PIR

Let us start from the baseline PIR protocol. For ease of exposition, assume the plaintext space of the homomorphic encryption scheme is $\mathbb{Z}_t$ with $t \geq 2^l$, where l is the bitsize of the elements of the database. In this protocol, the client sends a selection vector proportional to the database size n encrypted under additive homomorphic encryption (cf. Section II-B). The server returns a single encrypted entry by performing n homomorphic multiplications with a scalar (the database element) and n homomorphic additions as in Eq. (1).

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. A direct way to trade-off communication of upload and download is by reducing the length k of the selection vector and returning n/k database items (assuming $t > 2^{kl}$). This PIR construction could reduce the

```
**Procedure 3** PIR.HE.Extract
Input: $r \in \mathcal{C}$.
    $d := \mathsf{Dec}(\mathsf{sk}, r) \bmod 2^l$.
Output: $d \in \mathbb{Z}_{2^l}$.
```

```
**Procedure 4** PIR.GR.Query
Input: $i \in [n]$.
    $Q_1 := 2q_1 + 1 \leftarrow_{\$} \mathbb{P}$.
    $Q_2 := 2q_2\pi_i + 1 \leftarrow_{\$} \mathbb{P}$.
    $m := Q_1 Q_2$.
    $g \leftarrow_{\$} \mathbb{Z}_m$ s.t. $|\langle g \rangle| = q_1 q_2 \pi_i$.
Output: $(m, g) \in \mathbb{Z} \times \mathbb{Z}_m^*$.
```

TABLE I
COMMUNICATION-COMPUTATION TRADE-OFF OF HOMOMORPHIC ENCRYPTION BASED PIR PROTOCOLS, FROM SECTIONS III-B TO III-D.

	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 (Section III-B)	$\mathcal{O}\left(dn^{\frac{1}{d}} + F^{d-1}\right)$	$\mathcal{O}\left(\log n + F^{\log n - 1}\right)$	$n(A+S)$	$n^{\frac{1}{d}} F^{d-1}(A+S)$	$F^{\log n - 1}(A+S)$
Somewhat HE (Section III-D)	$\mathcal{O}\left(dn^{\frac{1}{d}}\right)$	$\mathcal{O}(\log n)$	$n(A+S) + n^{\frac{d-1}{d}}M$	$n(A+S) + n^{\frac{d-1}{d}}M$	$n(A+S+M)$
Fully HE (Section III-C)	–	$\mathcal{O}(\log n)$	–	–	$n \log nM + n(A+S)$

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

communication to $O(n^{1/2})$ ciphertexts, by sending a selection vector of size $n^{1/2}$ and returning $O(n^{1/2})$ encrypted database entries.

Two approaches have been proposed in the literature to reduce the overall communication cost: either using recursion (also called folding [GH19]) using additive homomorphic encryption, or a trivial solution using fully homomorphic encryption. We survey these two approaches below, before presenting our protocols based on somewhat homomorphic encryption.

B. Recursion with Additive HE

Kushilevitz, Ostrovsky [KO97b], and later Stern [Ste98], 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 (say) $M_{i,j} := \mathsf{D}_{in^{1/2}+j}$. 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 $\vec{s}_2 = (s_{2,i}) = (\delta_{j,j'})$ of dimension $n^{1/2}$. In particular, it holds that $s_{1,i} \cdot s_{2,j} = \delta_{i,i'} \cdot \delta_{j,j'} = \delta_{in^{1/2}+j,k}$, for all i, j.
The server then performs three steps:

```
Procedure 5 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: $g' \in \mathbb{Z}_m^*$.
```

```
Procedure 6 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 algorihm.
Output: $d \in \mathbb{Z}_{\pi_i}$.
```

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 c_i as F plaintext elements $c_{i,1}, \ldots, c_{i,F}$.
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, $r = (c'_1, \ldots, c'_F) \in \mathcal{C}^F$, 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 $n_1 = n_2 = n^{1/2}$). When $n_i = n^{1/d}$, we accomplish the following communication complexity: $\mathcal{O}\left(c(n) \cdot dn^{1/d}\right)$ for the user's query and $\mathcal{O}\left(F^{d-1}c(n)\right)$ for the server's response. In particular, for small values of d, we will get sublinear communication. However, note that for full recursion, i.e., $d = \log n$, communication becomes polynomial in n.

The "layering technique" explained above provides a way to emulate multiplicative homomorphism in one very restricted setting, which, however, suffices for PIR construction. The computation that is enabled by the layering approach for multiplication is inner product with a selection vector that has exactly one non-zero entry which is equal to one.

Remark 2. *We assumed w.l.o.g. 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 [DJ01], 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 A.*

C. Polylogarithmic Communication with FHE

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

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

A possible approach to computing the selection vector for the PIR query using FHE is based on the following observation: the i-th bit in the PIR query vector is the output of the equality check between the query index k and i. Hence, instead of sending the selection vector $\vec{s}$, the client can encrypt each bit k_j of the index k and send the resulting $\kappa = \log n$ ciphertexts to the server. The server then homomorphically computes the selection vector and proceeds as in the baseline PIR construction.

This construction achieves communication complexity: $\mathcal{O}(\log n)$ for the user's query and $\mathcal{O}(1)$ for the server's response (note that the ciphertext size is independent of the database, hence included in the $\mathcal{O}$ notation.).

D. Computation-Communication Trade-Offs using SHE

Somewhat homomorphic encryption (SHE) provides bounded level of multiplicative homomorphism. Most existing FHE solutions can be instantiated also as SHE [BGV14], [FV12]. In this section, we present a PIR construction that leverages SHE to achieve new computation-communication trade-offs.

a) First Approach: Equality Circuit: A first approach is essentially implementing the protocol from Section III-C leveraging the observation that since the values k and i have at most $\kappa = \log n$ bits and the arithmetic circuit for computing equality comparison has multiplicative depth $\log \kappa = \log \log n$. In that case, the ciphertext size depends on the size of the database and the communication complexity becomes $\mathcal{O}\left(c(n) \log n\right)$ for the user's query and $\mathcal{O}\left(c(n)\right)$ for the server's response, it suffices to use somewhat homomorphic encryption that supports such level of multiplicative homomorphism. In that case, the ciphertext size depends on the size of the database and the communication complexity becomes $\mathcal{O}\left(c(n) \log n\right)$ for the user's query and $\mathcal{O}\left(c(n)\right)$ for the server's response.

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 the multiplication of $\log n$ encrypted bits, which requires $\log \log n$ multiplicative degree.

b) Second Approach: Layered Multiplications: A second approach would be to start from the recursive protocol of Section III-B, but using SHE homomorphic multiplication as opposed to "emulating" with additions Using the same notation as in Section III-B, 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: $\mathcal{O}\left(c(n) \cdot dn^{1/d}\right)$ for the user's query and $\mathcal{O}\left(c(n)\right)$ for the server's response.

c) Third Approach: Reconstruct Selection Vector: Note that the approach above keeps the layered approach of the recursive PIR of Section III-B, and 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, where the SHE only required to handle depth $\log \kappa = \log \log n$. 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.

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.

In Appendix A we overview which of the techniques above are instantiatable with specific HE schemes.

IV. Optimizations and Trade-Offs for SealPIR

The SealPIR protocol proposed by Angel et al. [ACLS18] improves over the XPIR protocol proposed by Aguilar Melchor et al. [MBFK16]. Both SealPIR and XPIR instantiate the recursive PIR using the lattice-based FV homomorphic encryption scheme [FV12] (viewed as an additive homomorphic encryption scheme). SealPIR further proposes several optimizations; most notably a compression and oblivious expansion procedures that reduce significantly the communication cost.

In this section, we first recall the SealPIR protocol (Section IV-B), and then present several optimizations that further reduce the communication bandwidth (Sections IV-C and IV-D). Then, we note in Section IV-E that since the FV scheme is a *somewhat* homomorphic encryption scheme, it is actually possible to adopt the PIR protocol we introduced in Section III-D. We thus introduce MulPIR which offer a computation-communication trade-off compared to SealPIR.

A. The Fan–Vercauteren Cryptosystem

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, [ACLS18] introduced a specific operation called substitution, instantiated using the plaintext slot permutation of [GHS12].

- **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 $m(x^k) \in R/tR$.

B. SealPIR

SealPIR instantiates the protocol of Section III-B on top of the FV homomorphic encryption scheme [FV12] with a beautiful optimization called compression and oblivious expansion. This optimization aims at reducing the communication from the client to the server by "compressing" the query on the client side, and expanding it obliviously on the server side.

In the PIR protocols described in Section III, the query consists of encryptions of the bits of the selection vector(s). We start with the natural observation that many bits can be encrypted in a single ciphertext (and, in particular, at least one per polynomial coefficient). The work of [ACLS18] shows that an encryption with one bit per coefficient can be *obliviously expanded* by the server to obtain encryptions of each of the bits in the constant coefficient of the plaintext polynomial.

Denote $\vec{s} = (s_i) \in \{0,1\}^m$ the selection vector the client wants to encrypt. (For example, in the baseline PIR, $m = n$, while in the recursive PIR, the client will encrypt d of these selection vectors of size $m = n^{1/d}$). Without loss of generality, assume $m \leq N$, otherwise the selection vector can be additionally split into $\lceil m/N \rceil$ different selection vectors. SealPIR's Query algorithm is given in Algorithm 7.

In particular, this technique enables to decrease the upload cost in SealPIR by a factor $\approx N$ (the polynomial ring dimension); practical results are presented in Table III for selection vectors of dimension 5000 and 500. Note that, as soon as recursion is used, the selection vectors will be of size $\leq n^{1/2}$, which will be below $N = 2048$ when the database size is $n \leq 2^{24}$.

TABLE II

GAIN FROM OUR COMPRESSION TECHNIQUES (SECTIONS IV-C TO IV-E), AND OF MULPIR (SECTION IV-E), COMPARED TO SEALPIR.

Database size	$n = 2^{18}$			$n = 2^{20}$		
Recursion	$d = 1$	$d = 2$	$d = 3$	$d = 1$	$d = 2$	$d = 3$
SealPIR upload (kB)	416	64	96	1664	64	96
SealPIR download (kB)	32	256	2048	32	256	2048
SealPIR and optimizations (Sections IV-C and IV-D) Upload (kB)	**183**	14	14	**733**	14	14
SealPIR and optimizations (Sections IV-C and IV-D) Download (kB)	**10**	82	655	**10**	82	655
Total communication wrt SealPIR	**0.43**$\times$	0.30$\times$	0.31$\times$	**0.44**$\times$	0.30$\times$	0.31$\times$
MulPIR upload (kB)	**183**	**19**	**59**	**733**	**19**	**59**
MulPIR download (kB)	**10**	**21**	**43**	**10**	**21**	**43**
Total communication wrt SealPIR	**0.43**$\times$	**0.13**$\times$	**0.04**$\times$	**0.44**$\times$	**0.13**$\times$	**0.04**$\times$

For SealPIR, we use the same parameters as in [ACLS18, Fig. 9]. The plaintext modulus is fixed to $t = 2^{12} + 1$. *For the optimizations, we use modulus switching to a prime of* 25 *bits for SealPIR, drop respectively* 5 *and* 8 *bits for upload and download ciphertexts. For MulPIR, the parameters depend on the recursion: for* $d = 2$, *we use* $N = 2048$ *and* $\log_2(q) = 80, \log_2(p) = 48$; *for* $d = 3$, *we use* $N = 4096$ *and* $\log_2(q) = 120, \log_2(p) = 50$.

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 given in Algorithm 8.

C. Compression and Expansion Beyond Selection Vector

Our first contribution comes from the following observation: SealPIR oblivious expansion (Algorithm 8) is *linear over the plaintext space*. Indeed, all operations used in the algorithms are linear over the plaintext space: additions, substitutions, and scalar multiplications. Hence, it follows that the SealPIR oblivious expansion algorithm enables to expand encryptions of vectors beyond binary selection vectors. Indeed, 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 use the above observation to further compress the size of the query in SealPIR and remark that it could also be applicable in other contexts as well. In SealPIR with recursion d, the upload consists of $d \cdot \lceil n^{1/d}/N \rceil$ ciphertexts, where the factor d comes from the fact that we have d selection vectors, and $\lceil n^{1/d}/N \rceil$ comes from the fact that one selection vector of size $n^{1/d}$ can be embedded in $\lceil n^{1/d}/N \rceil$ plaintext polynomials in R/tR. Now we can consider the concatenation of the d selections vectors of size $n^{1/d}$ as one vector of size $d \cdot n^{1/d}$ and use the compression technique over that vector. It follows that the upload size becomes $\lceil d \cdot n^{1/d}/N \rceil$ ciphertexts. In practice, for $d \geq 2$, we usually have $d \cdot n^{1/d} < N$, which enables to reduce the upload to a unique ciphertext in SealPIR. We report in Table II the practical improvements in communication when using this technique on the recommended parameters of SealPIR.

D. Compressing the ciphertexts

The previous technique is helpful to reduce the upload communication, but does not affect the download. In this section, we propose to use three compression techniques for homomorphic encryption ciphertexts that will enable to reduce further the communication: using secret key encryption, modulus switching, and bit dropping.

Procedure 7 SealPIR.Query

```
Input: $k \in [1, n]$.
  Generate $\vec{s}_j = (s_{j,i})_{i \in [m]}$ the $d$ selections vectors in $\{0,1\}^m$.
  $\forall j \in [d], m_j \leftarrow \sum_{i \in [m]} s_{j,i} x^i \in R/tR$.
  $\forall j \in [d], q_j \leftarrow \mathsf{Enc}(\mathsf{sk}, m_j)$.
Output: $\vec{q} = (q_j)_{j \in [d]} \in \mathcal{C}^d$.
```

a) Using symmetric encryption: The first optimization comes from the fact that the client, who creates the query ciphertexts, knows the secret key of the homomoprhic encryption scheme. In particular, instead of using the public key encryption algorithm, it can use the secret key encryption algorithm of FV. Recall that a FV ciphertext is a tuple (c_0, c_1) in R/qR. We briefly describe below the public key and secret key encryption algorithms of FV:

- *Secret Key Encryption.* The secret key is a small polynomial $s \in R/qR$. To encrypt $m \in R/tR$, sample c_0 uniformly at random in R/qR and $e \in R/qR$ a small polynomial, and define $c_1 = c_0 \cdot s + e + \lceil q/t \rceil m$.
- *Public Key Encryption.* The secret key is a small polynomial $s \in R/qR$ and the public key $(a, b = as+e)$ is an encryption of 0 using the algorithm above. To encrypt $m \in R/tR$, sample $r, e_1, e_2 \in R/qR$ small polynomials, and define $c_0 = a \cdot r + e_1$, $c_2 = b \cdot r + e_2 + \lceil q/t \rceil m$.

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 saves roughly a factor two in size for the upload ciphertexts.

b) Using Modulus Switching: In FV, there exists a technique called *modulus* switching, that 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 [CLP19]. This technique 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 $60/25 = 2.4$.

c) Using Bit Dropping: Finally, we propose to use a technique used in most post-quantum lattice-based encryption schemes proposed for standardization to NIST, such as NewHope [ADPS16] and CRYSTAL-Kyber [BDK⁺18], that we call bit-dropping. Essentially, this technique enables to drop the least significant bits of the ciphertext as they carry no information about the message. Indeed, at the end of the PIR computation, each of the ciphertext r_i in the response if a tuple $(c_0, c_1) \in (R/qR)^2$ such that

$$c_1 - c_0 \cdot s \bmod q = \lceil q/t \rceil \cdot m + e \in \mathbb{Z}\,,$$

where $\|e\|$ is small (and in particular, $\|e\|_\infty \leq \lceil q/t \rceil$) and m is the plaintext. Now, assume that instead of c_0 and c_1, the server sends the $\log_2 q - b$ most significant bits from c''_0, c''_1 only. This essentially corresponds to defining

Procedure 8 SealPIR Oblivious Expansion

```
Input: Query $q = \mathsf{Enc}(\sum_{i=0}^{k-1} s_i x^i)$, $k \in [N]$
  Find smaller $m = 2^\ell \geq k$
  ciphertexts = $[q]$
  for $j = 0$ to $\ell - 1$ do
    for $k = 0$ to $2^j - 1$ do
      $c_0 \leftarrow$ ciphertexts$[k]$
      $c_1 \leftarrow x^{-2^j} \cdot c_0$                                // scalar multiplication
      $c'_k \leftarrow c_0 + \mathsf{Sub}_{N/2^j+1}(c_0)$
      $c'_{k+2^j} \leftarrow c_1 + \mathsf{Sub}_{N/2^j+1}(c_1)$
    end for
    ciphertexts = $[c'_0, \dots, c'_{2^{j+1}-1}]$
  end for
  inverse $\leftarrow m^{-1} \bmod t$                                  // normalization
  for $j = 0$ to $k - 1$ do
    $o_j \leftarrow$ inverse $\cdot$ ciphertexts$[j]$
  end for
Output: output = $[o_0, \dots, o_{k-1}]$
```

$c_i' = c_i + e_i$ where e_i is a small noise such that the b least significant bits of c_i' are 0, and send $c_i'' = c_i'/2^b$ to the client. Then, the client can reconstruct the c_i' and compute

$$c_1' - c_0' \cdot s \bmod q = \lceil q/t \rceil \cdot m + (e + e_1 - e_0) \bmod q\,.$$

Now, if $e + e_1 - e_0$ is small enough, the last equality will hold over $\mathbb{Z}$ and the client will be able to decrypt the ciphertext and recover m. This compression technique can be used both for upload and download, and enable saving a few bits per polynomial coefficient.

d) Compatibility and Cost: Note that all the techniques described above can be use concurrently, and are also compatible with the technique from Section IV-C. We report in Table II the gain obtained by using these techniques on SealPIR.

Additionally, none of these techniques depend on the size of the database, and need only to be performed on the input and output ciphertexts, effectively adding a negligible computational cost to SealPIR.

E. MulPIR: Leveraging Homomorphic Multiplication

While the optimizations from the previous sections enable to critically improve the communication over previous schemes using the FV homomorphic encryption scheme without optimization [MBFK16], the total communication cost per PIR query may be prohibitive when $d = 1$ (baseline PIR) or $d \geq 3$. For the former case, this is because the upload cost is linear in the size of the database, while in the latter case, this comes from the fact that the result consists of F^{d-1} ciphertexts where $F \geq 4$ is the expansion factor of FV ciphertexts.

Now, for the latter case, we have seen in Section III-B that it is possible to use a somewhat homomorphic encryption scheme to further reduce the communication bandwidth when recursion is used. Since FV is somewhat homomorphic, we propose to directly use the homomorphic multiplication of the FV homomorphic encryption scheme to instantiate the PIR protocol in Section III-D-b. We call the resulting scheme, together with all the optimizations from Sections IV-C and IV-D, MulPIR, and report the communication costs in Table II.

Since we need to enable one homomorphic multiplication, we need to select larger parameters in MulPIR than in SealPIR, which explains why the *upload* cost is higher than for SealPIR with the optimizations from Sections IV-C and IV-D.

In Appendix B we discuss the one time communication cost for SealPIR and MulPIR associated with sending the Galois Keys required to perform the Substitute algorithm.

V. Efficiently Implementing Gentry–Ramzan PIR

We now describe our concrete implementation of the Gentry–Ramzan PIR protocol [GR05]. Since the main computation bottleneck for large databases is the server computation (cf. Procedure 4), we focus on optimizing this part of the protocol. First, we show how to efficiently encode the database such that the encoding satisfies Eq. (2). While this is a one-time setup, it is non-trivial to implement with complexity sub-quadratic in the database size. Second, we describe an algorithm for fast exponentiation that exploits the fact that we have a fixed exponent that is independent of the query. We further speed up the exponentiation with a *client-aided* variant. Here, we leverage the fact that the client can exponentiate more quickly by using the prime factorization of the modulus M.

A. Fast Modular Interpolation

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

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 $E = \sum\limits_{k=1}^{n} d_k a_k M_k = \sum\limits_{k=1}^{n} d_k a_k \left(\prod\limits_{i=1, i\neq k}^{n} \pi_i \right)$.

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 $E \equiv d_k a_k M_k \equiv d_k - d_k b_k \pi_k \equiv d_k \mod \pi_k$ 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 [SS71] and the extended Euclidean algorithm [SZ04], 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 [BM74]. 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 [SS71], has a total running time of $O(n \log^2 n \log\log n)$ [BM74]. It relies on the fact that the *supermoduli* M_1, M_2 can be pre-computed, 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 VI-C).

B. Fast Modular Exponentiation

An important subroutine in the Gentry–Ramzan PIR protocol is performing a modular exponentiation with a large exponent that encodes the entire database. This occurs in a client-server setting where the client holds both the base g and the modulus M while the server holds the large exponent E. For privacy, it is critical the server never learns the prime factorization of the modulus M.

Now, it is a well known fact that one can use the prime factorization of the modulus M to reduce the cost of modular exponentiation (e.g., using Fermat's Little Theorem). The main idea of our faster algorithm is thus to leverage the client's knowledge of the prime factorization of M to improve the computational costs of the server. To materialize this idea, the client will compute exponentiations of the base g to small exponents using the prime factorization of M. These small exponents will be independent of the large exponent E. Nonetheless, the server will be able to use these exponentiations as hints to compute the large exponentiation.

Concretely, the server rewrites the large exponent E according to some base $b \geq 2$ (setting $b = 2$ is binary and $b = 10$ is decimal). Without loss of generality, we know that $E = E_0 + E_1 b + E_2 b^2 + \ldots + E_m b^m$. Then, we know that $g^E = g^{E_0} \cdot (g^b)^{E_1} \cdot (g^{b^2})^{E_2} \cdots (g^{b^m})^{E_m}$. So, the client can compute the $m+1$ values: $g, g^b, g^{b^2}, \ldots, g^{b^m}$ without knowing the exponent E. Furthermore, these m exponentiations may be efficiently computed by the client using the prime factorization of M. Note that revealing these 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 $m+1$ values, the server's task reduces to the problem of given $m+1$ bases and $m+1$ exponents, compute the multiplication of the bases after they have been exponentiated with their corresponding exponents. To do this efficiently, one can refer to the survey by Bernstein [Ber]. For our implementation, we choose Straus's algorithm [Str64], which description can be found in [KMVOV96, Alg. 14.88].

VI. Experimental Evaluation

In this section, we present experimental results that measure the efficiency of different PIR protocols and illustrate the possible tradeoffs that they enable. These results can inform decision making of what is the most appropriate PIR instantiation for a particular application.

A. Experimental Setup

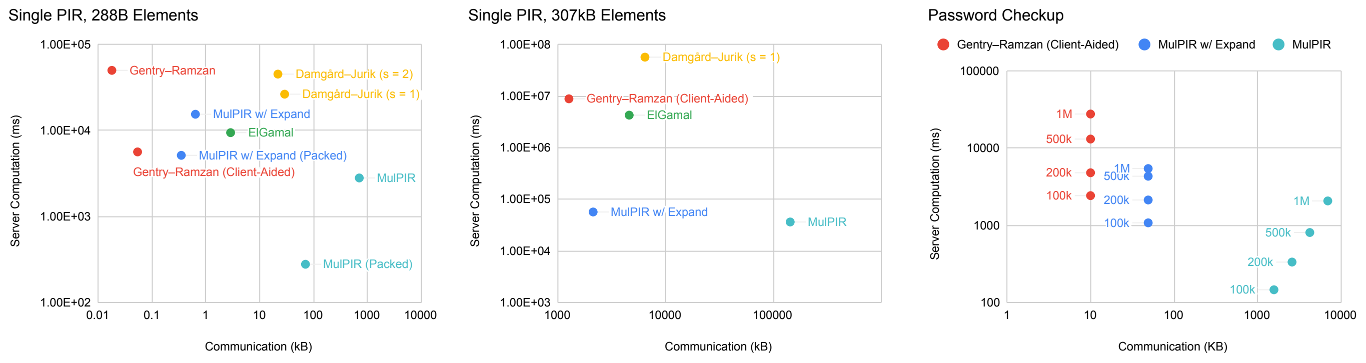
All our experiments are performed on a desktop computer with a Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz, 64GB of RAM, running Ubuntu. Unless specified otherwise, the parameters of the PIR protocols are as follows:

TABLE III
SMALL ELEMENTS DATABASE: 5000 ELEMENTS OF 288B, FOR A TOTAL SIZE OF $\approx$ 1MB.

		Communication (kB)		Computation (ms)					Total Server Cost (US cents)
	packed	upload	download	C.Setup	S.Setup	C.Create	S.Respond	C.Process	
SealPIR/MulPIR	✗	70480	21	0	522	5474	2803	0.3	0.54
SealPIR/MulPIR with Expand	✗	43	21	0	512	247	15437	0.3	0.0048
SealPIR/MulPIR	✓	7048	21	0	52	550	278	0.3	0.054
SealPIR/MulPIR with Expand	✓	14	21	0	53	242	5136	0.3	0.0017
Damgård–Jurik ($s = 1$)	✓	2900	0.6	62806	1	29148	26418	6	0.030
Damgård–Jurik ($s = 2$)	✓	2175	0.9	168273	1	32681	45210	14	0.030
Gentry–Ramzan	✓	0.5	1.3	0	1278	12037	49991	361	0.014
Gentry–Ramzan (Client-Aided)	✓	4.1	1.3	0	1280	11327	5631	367	0.0016
ElGamal	✓	280	8	560	22	736	9428	11586	0.0048

Average over 10 computations. "Packed" indicates that the database was reduced to store as many elements as possible per ciphertext. Since Gentry–Ramzan and Damgård–Jurik plaintext block sizes are smaller than the size of the entries, respectively 5 and 72 ciphertexts are needed to store a database element. Damgård–Jurik client's setup includes precomputation to speed up the query creation. Total server costs were computed using Google Cloud Platform prices [gcp], which were at the time of writing at one cent per CPU-hour and 8 cents per GB of network traffic.

Fig. 2. Communication–computation tradeoff of PIR constructions, based on data from Table III (left), Table IV (middle), and Table VI (right).



- **El Gamal PIR**: NIST P-224r1 curve and a plaintext block size of 32 bits.
- **Damgård–Jurik PIR**: 1160-bit primes.
- **MulPIR**: Polynomials of dimension 2048 and a modulus of 60 bits. Plaintext modulus set to $t = 2^{12} + 1$.
- **Gentry–Ramzan**: 2048-bit modulus and plaintext block size of 500 bits. When specified as "client-aided", the client sends 15 generators to the server (cf. Section V-B).

All the implementations are standalone and rely only on OpenSSL for BigNum and elliptic curve operations.

B. Baseline Computation Costs

We start with an evaluation of the baseline computation cost of the PIR protocols from Sections II-C and III. Note that in this setting, since we do not use recursion (hence no homomorphic multiplication is performed), SealPIR and MulPIR offer essentially the same performance.

In Table III, we consider a database of 5000 elements of length 288B (such database was used for evaluation in [AS16]) and evaluate the client and server costs to setup, create a request, respond to this request, and extract the response. We report communication and computation costs when the database is packed (i.e., the database is

TABLE IV
PRIVATE FILE DOWNLOAD: 10,000 ELEMENTS OF 307KB, FOR A TOTAL DATABASE SIZE OF $\approx$ 3GB.

		Communication (kB)		Computation (ms)					Total Server Cost (US cents)
	# chunks	upload	download	C.Setup	S.Setup	C.Create	S.Respond	C.Process	
SealPIR/MulPIR	100	140960	2048	0	105670	11063	36270	25	1.1
SealPIR/MulPIR with Expand	100	71	2048	0	105594	248	56656	25	0.032
Gentry–Ramzan (Client-Aided)	4955	4.1	1259	0	1262133	9676	8848360	344518	2.5
Damgård–Jurik ($s = 1$)	1060	5800	614	$\approx$ 39000000	$\approx$ 2500	$\approx$ 250000	$\approx$ 57000000	$\approx$ 7000	16
ElGamal	76800	280	4300	128	$\approx$ 41000	$\approx$ 1500	$\approx$ 4400000	$\approx$ 12500000	1.2

Average over 10 computations. The number of chunks indicates how many ciphertexts are needed to store a database element. The timings indicated with $\approx$ have been estimated on a smaller number of chunks to finish in a reasonable amount of time. Total server costs were computed as in Table III.

TABLE V
CPU COSTS (IN MS) OF SEALPIR AND SEAL-MULPIR (RECURSION $d = 2$) FOR A DATABASE OF n ELEMENTS OF 288B.

	SealPIR ($d = 2$) [sea19b]				Seal-MulPIR ($d = 2$)			
database size n	65536	262144	1048576	4194304	65536	262144	1048576	4194304
actual number of rows after packing	*6554*	*26215*	*104858*	*419431*	*6554*	*26215*	*104858*	*419431*
Client Setup	40	40	40	40	10	11	11	12
Client Query / Client Extract	1	1	1	1	1	1	1	1
Server Setup	324	1245	4792	19063	768	2982	11772	47819
Server Expand	70	140	279	553	165	330	653	1325
Server Respond	**300**	**907**	**3087**	**11513**	**626**	**1873**	**6016**	**21705**

For this comparison only, we reimplemented MulPIR using the Seal library [sea19a] (Seal-MulPIR) and used SealPIR's implementation from Github without modification [sea19b]. Here, because of the noise growth in the Seal library, Seal-MulPIR uses a polynomial dimension of 4096, *a* 120*-bit modulus (product of* 2 60*-bit moduli), and a plaintext modulus $t = 2^6 + 1$.*

reshaped so as to maximize the number of elements in the response; as done in SealPIR [ACLS18]). For comparison we also report the costs without packing.

The table also reports on the gain of sending several generators for Gentry–Ramzan. Recall that in Section V-B, we proposed to use Straus's algorithm to compute the exponentiation at the core of the Gentry–Ramzan PIR protocol. Note that the cost of Straus's algorithm (expressed in number of multiplications in [Ber] for example), for which one could derive an optimal number of generators to send, does not account for the precomputation cost. However, in Gentry–Ramzan, the server does not know the modulus m before receiving the client's request, hence this cost is factored in the server response. In practice, we have determined that about 15 generators was the best communication-computation trade-off one could obtain in Gentry–Ramzan.

Finally, the table shows the price one would have to pay for a single execution of the experiment on Google's Cloud Platform [gcp], using a preemptible general-purpose VM with a single CPU core.

C. Application: Private File Download

Our first application is that of a private file download service. We consider a "fat" database $10,000$ files of $307,200$ bytes. The total size of the database is therefore 3GB. 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 IV.

As expected, Damgård–Jurik and ElGamal are significantly slower than the (packed) variant of SealPIR/MulPIR and Gentry–Ramzan, and will not be considered further in the rest of the section. Furthermore, we can see that Expand enables to reduce the communication requirements of SealPIR/MulPIR significantly. If it remains far from the efficient communication cost of Gentry–Ramzan, it offers much better performance.

D. Application: Comparison With SealPIR Example Database

This section revisits the application of SealPIR [ACLS18], i.e., serving a database of 288B messages. We use this section to further compare the open-source implementation of SealPIR (without modification) available on

GitHub [sea19b] against an implementation of MulPIR that uses one homomorphic multiplication (Section IV-E). For this experiment only, and to facilitate comparison, *both* implementations rely on the SEAL homomorphic encryption library [sea19a]—we refer to this MulPIR implementation by "Seal-MulPIR". We use the same database sizes as in [ACLS18] and report the costs in Table V. The experiment results reflects the use of the more costly homomorphic multiplication in Seal-MulPIR. Note that our custom made implementation of MulPIR (as used in the other sections) will feature a smaller noise growth and hence will enable to select smaller parameters in the Password checkup experiment (Section VI-E).

E. Application: Password Checkup

Recent works study the problem of preventing credential stuffing attacks [TPY$^+$19], [LPA$^+$19] by proposing privacy-preserving protocols where a client queries a centralized breach repository to determine whether her username and password combination has been part of breached data, without revealing the information queried. While this application seems to be a perfect fit for keyword PIR, the size of leaked credentials (4+ billion credentials [TPY$^+$19]) remains prohibitively large for PIR. Instead, [TPY$^+$19], [LPA$^+$19] propose protocols where the client and the server first run an oblivious PRF evaluation (both on usernames and on the tuple username/password), then use the first value to retrieve a bucket and the second value to test for membership after downloading the whole bucket. Precisely, [TPY$^+$19] proposes to use 2^{16} buckets, which we infer to contain about $60k$ elements, and downloading a whole bucket is about 1.6MB of communication.

In this section, we propose to replace the download of the entire bucket with a PIR query. Table VI 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 [TPY$^+$19, Tab. 2]) and enables decrease of the communication bandwidth or the number of buckets (or even both).

For Gentry–Ramzan, we propose to perform keyword PIR over a bucket using Cuckoo hashing, as introduced in Appendix C1. 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 [TPY$^+$19].

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

Therefore, we propose to use a simpler solution: the server selects a random hash function h of image size k, and use it to construct k bins by placing each of the m elements e in the bin of index $h(e)$. The client then performs a PIR query over a database of size k. In order to minimize k, we want to make the number of elements in each bucket as large as possible while still fitting in one MulPIR ciphertext. Denote $m = ck \ln k$ for a constant c. From [RS98, 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 report on the communication and computation costs in Table VI. In particular, we conclude that for buckets of size 50k, the server computation time is less than 100ms for about 1MB of communication, and about 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 [TPY$^+$19].

In Appendix E we present experimental results from the first implementation of full-recursion PIR.

VII. Conclusion

Similarly 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

TABLE VI
PASSWORD CHECKUP: SERVER COMPUTATION AND COMMUNICATION.

Bucket size	Gentry–Ramzan Comm. (kB)	Gentry–Ramzan Time (ms)	MulPIR Comm. (kB)	MulPIR Time (ms)	MulPIR wo/ Expand Comm. (kB)	MulPIR wo/ Expand Time (ms)
10k	**10**	254	49	540	612	**34**
20k	**10**	508	49	540	612	**34**
50k	**10**	1308	49	1020	979	**69**
100k	**10**	2428	49	1078	1571	**146**
200k	**10**	4807	49	2133	2586	**334**
500k	**10**	13161	49	4335	4221	**807**
1M	**10**	27788	49	5450	6928	**2074**

The plaintext modulus of MulPIR is $t = 17$ to enable recursion $d = 2$, and k is respectively equal to 403, 403, 1k, 3k, 8k, 22k, and 58k.

several PIR constructions using different HE schemes as well as the Gentry–Ramzan PIR, and present a comprehensive evaluation of their costs in different settings. Our results demonstrate that the lattice-based FV homomorphic encryption outperforms Paillier and ElGamal in HE-based PIR constructions, while Gentry-Ramzan provides best communication overhead as well as dollar cost for some databases. Our new SHE-based PIR enables for the first time experimental evaluation of full recursion PIR. Overall our results show competitive efficiency for PIR applications (e.g., file download, password checkup), and we hope they will serve as a useful reference to inform the choices of PIR construction and parameters for different applications.

REFERENCES

[ACLS18] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2018, pp. 962–979.

[ADPS16] E. Alkim, L. Ducas, T. Pöppelmann, and P. Schwabe, "Post-quantum key exchange - A new hope," in *USENIX Security Symposium*. USENIX Association, 2016, pp. 327–343.

[AS16] S. Angel and S. T. V. Setty, "Unobservable communication over fully untrusted infrastructure," in *OSDI*. USENIX Association, 2016, pp. 551–569.

[BDK$^+$18] J. W. Bos, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, J. M. Schanck, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS - kyber: A cca-secure module-lattice-based KEM," in *EuroS&P*. IEEE, 2018, pp. 353–367.

[Ber] D. J. Bernstein, "Pippenger's exponentiation algorithm."

[BGV14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," *TOCT*, vol. 6, no. 3, pp. 13:1–13:36, 2014.

[BIPW17] E. Boyle, Y. Ishai, R. Pass, and M. Wootters, "Can we access a database both locally and privately?" in *Theory of Cryptography Conference*. Springer, 2017, pp. 662–693.

[BM74] A. Borodin and R. Moenck, "Fast modular transforms," *J. Comput. Syst. Sci.*, vol. 8, no. 3, pp. 366–386, 1974.

[CGN97] B. Chor, N. Gilboa, and M. Naor, "Private information retrieval by keywords," 1997.

[CGN98] ——, "Private information retrieval by keywords," *IACR Cryptology ePrint Archive*, vol. 1998, p. 3, 1998.

[CHLR18] H. Chen, Z. Huang, K. Laine, and P. Rindal, "Labeled PSI from fully homomorphic encryption with malicious security," in *ACM Conference on Computer and Communications Security*. ACM, 2018, pp. 1223–1237.

[CHR17] R. Canetti, J. Holmgren, and S. Richelson, "Towards doubly efficient private information retrieval," in *Theory of Cryptography Conference*. Springer, 2017, pp. 694–726.

[CKGS98] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan, "Private information retrieval," *J. ACM*, vol. 45, no. 6, pp. 965–981, 1998.

[CLP19] A. Costache, K. Laine, and R. Player, "Homomorphic noise growth in practice: comparing BGV and FV," *IACR Cryptology ePrint Archive*, vol. 2019, p. 493, 2019.

[CLR17] H. Chen, K. Laine, and P. Rindal, "Fast private set intersection from homomorphic encryption," in *ACM Conference on Computer and Communications Security*, ser. CCS '17, 2017.

[CMS99] C. Cachin, S. Micali, and M. Stadler, "Computationally private information retrieval with polylogarithmic communication," in *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, ser. EUROCRYPT'99, 1999.

[CO18] M. Ciampi and C. Orlandi, "Combining private set-intersection with secure two-party computation," in *SCN*, ser. Lecture Notes in Computer Science, vol. 11035. Springer, 2018, pp. 464–482.

[DC14] C. Dong and L. Chen, "A fast single server private information retrieval protocol with low communication cost," in *ESORICS (1)*, ser. Lecture Notes in Computer Science, vol. 8712. Springer, 2014, pp. 380–399.

[DDS15] W. Dai, Y. Doröz, and B. Sunar, "Accelerating SWHE based pirs using gpus," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 8976. Springer, 2015, pp. 160–171.

[DHS14] D. Demmler, A. Herzberg, and T. Schneider, "Raid-pir: Practical multi-server pir," in *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, ser. CCSW '14, 2014.

[DJ01] I. Damgård and M. Jurik, "A generalisation, a simplification and some applications of paillier's probabilistic public-key system," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, vol. 1992. Springer, 2001, pp. 119–136.

[DRRT18a] D. Demmler, P. Rindal, M. Rosulek, and N. Trieu, "PIR-PSI: scaling private contact discovery," *PoPETs*, vol. 2018, no. 4, pp. 159–178, 2018.

[DRRT18b] ——, "Pir-psi: Scaling private contact discovery," *Proceedings on Privacy Enhancing Technologies*, vol. 2018, no. 4, pp. 159–178, 2018.

[DSH14] Y. Doröz, B. Sunar, and G. Hammouri, "Bandwidth efficient PIR from NTRU," in *Financial Cryptography Workshops*, ser. Lecture Notes in Computer Science, vol. 8438. Springer, 2014, pp. 195–207.

[FIPR05] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *Proceedings of the Second International Conference on Theory of Cryptography*, ser. TCC'05, 2005.

[FPSS05] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis, "Space efficient hash tables with worst case constant access time," *Theory Comput. Syst.*, vol. 38, no. 2, pp. 229–248, 2005.

[FV12] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," *IACR Cryptology ePrint Archive*, vol. 2012, p. 144, 2012.

[Gam85] T. E. Gamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Trans. Information Theory*, vol. 31, no. 4, pp. 469–472, 1985.

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

[GH19] C. Gentry and S. Halevi, "Compressible fhe with applications to pir," Cryptology ePrint Archive, Report 2019/733, 2019, https://eprint.iacr.org/2019/733.

[GHS12] C. Gentry, S. Halevi, and N. P. Smart, "Fully homomorphic encryption with polylog overhead," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 7237. Springer, 2012, pp. 465–482.

[GIKM98] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin, "Protecting data privacy in private information retrieval schemes," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, ser. STOC '98, 1998.

[GIKM00] ——, "Protecting data privacy in private information retrieval schemes," *J. Comput. Syst. Sci.*, vol. 60, no. 3, Jun. 2000.

[GKL10] J. Groth, A. Kiayias, and H. Lipmaa, "Multi-query computationally-private information retrieval with constant communication rate," in *Public Key Cryptography*, ser. Lecture Notes in Computer Science, vol. 6056. Springer, 2010, pp. 107–123.

[GLM16] M. Green, W. Ladd, and I. Miers, "A protocol for privately reporting ad impressions at scale," in *ACM Conference on Computer and Communications Security*. ACM, 2016, pp. 1591–1601.

[GMOT12] M. T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, "Privacy-preserving group data access via stateless oblivious ram simulation," in *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2012, pp. 157–167.

[GR05] C. Gentry and Z. Ramzan, "Single-database private information retrieval with constant communication rate," in *ICALP*, ser. Lecture Notes in Computer Science, vol. 3580. Springer, 2005, pp. 803–815.

[IKOS04] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, ser. STOC '04, 2004.

[JL09] S. Jarecki and X. 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*, ser. TCC '09, 2009.

[KLL$^+$15] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang, "Optimal rate private information retrieval from homomorphic encryption," *Proceedings on Privacy Enhancing Technologies*, vol. 2015, no. 2, pp. 222–243, 2015.

[KMVOV96] J. Katz, A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC press, 1996.

[KMW09] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM J. Comput.*, vol. 39, no. 4, Dec. 2009.

[KO97a] E. Kushilevitz and R. Ostrovsky, "Replication is not needed: Single database, computationally-private information retrieval," in *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, ser. FOCS '97, 1997.

[KO97b] ——, "Replication is NOT needed: SINGLE database, computationally-private information retrieval," in *FOCS*. IEEE Computer Society, 1997, pp. 364–373.

[KRS$^+$19] D. Kales, C. Rechberger, T. Schneider, M. Senker, and C. Weinert, "Mobile private contact discovery at scale," in *USENIX Security Symposium*. USENIX Association, 2019, pp. 1447–1464.

[Lip05] H. Lipmaa, "An oblivious transfer protocol with log-squared communication," in *Proceedings of the 8th International Conference on Information Security*, ser. ISC'05, 2005.

[LP17] H. Lipmaa and K. Pavlyk, "A simpler rate-optimal cpir protocol," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 621–638.

[LPA$^+$19] L. Li, B. Pal, J. Ali, N. Sullivan, R. Chatterjee, and T. Ristenpart, "Protocols for checking compromised credentials," in *ACM Conference on Computer and Communications Security*. ACM, 2019.

[MBFK16] C. A. Melchor, J. Barrier, L. Fousse, and M. Killijian, "XPIR : Private information retrieval for everyone," *PoPETs*, vol. 2016, no. 2, pp. 155–174, 2016.

[NPP99] M. Naor, B. Pinkas, and B. Pinkas, "Oblivious transfer and polynomial evaluation," in *Proceedings of the Thirty-first Annual ACM Symposium on Theory of Computing*, ser. STOC '99, 1999.

[Pai99] P. Paillier, "Public-key cryptosystems based on composite degree residuosity classes," in *EUROCRYPT*, ser. Lecture Notes in Computer Science, vol. 1592. Springer, 1999, pp. 223–238.

[PBP12] S. Papadopoulos, S. Bakiras, and D. Papadias, "pcloud: A distributed system for practical PIR," *IEEE Trans. Dependable Sec. Comput.*, vol. 9, no. 1, pp. 115–127, 2012.

[PPRY18] S. Patel, G. Persiano, M. Raykova, and K. Yeo, "Panorama: Oblivious ram with logarithmic overhead," in *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2018, pp. 871–882.

[PPY18] S. Patel, G. Persiano, and K. Yeo, "Private stateful information retrieval," in *ACM Conference on Computer and Communications Security*. ACM, 2018, pp. 1002–1019.

[PR04] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, May 2004.

[PR10] B. Pinkas and T. Reinman, "Oblivious ram revisited," in *Annual Cryptology Conference*. Springer, 2010, pp. 502–519.

[PSSZ15] B. Pinkas, T. Schneider, G. Segev, and M. Zohner, "Phasing: Private set intersection using permutation-based hashing," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 515–530.

[PSTY19] B. Pinkas, T. Schneider, O. Tkachenko, and A. Yanai, "Efficient circuit-based PSI with linear communication," in *EUROCRYPT (3)*, ser. Lecture Notes in Computer Science, vol. 11478. Springer, 2019, pp. 122–153.

[PSWW18] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder, "Efficient circuit-based psi via cuckoo hashing," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 125–157.

[PSZ18] B. Pinkas, T. Schneider, and M. Zohner, "Scalable private set intersection based on ot extension," *ACM Trans. Priv. Secur.*, vol. 21, no. 2, Jan. 2018.

[RS98] M. Raab and A. Steger, ""balls into bins" - A simple and tight analysis," in *RANDOM*, ser. Lecture Notes in Computer Science, vol. 1518. Springer, 1998, pp. 159–170.

[sea19a] "Microsoft SEAL," 2019, https://github.com/microsoft/SEAL. Accessed 2019-10-30.

[sea19b] "SealPIR: A computational PIR library that achieves low communication costs and high performance," 2019, https://github.com/microsoft/SealPIR. Accessed 2019-10-30.

[SS71] A. Schönhage and V. Strassen, "Schnelle multiplikation großer zahlen," *Computing*, vol. 7, no. 3-4, pp. 281–292, 1971.

[Ste98] J. P. Stern, "A new efficient all-or-nothing disclosure of secrets protocol," in *ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 1514. Springer, 1998, pp. 357–371.

[Str64] E. G. Straus, "Addition chains of vectors (problem 5125)," in *American Mathematical Monthly*, vol. 70, 1964, pp. 806–808.

[SZ04] D. Stehlé and P. Zimmermann, "A binary recursive gcd algorithm," in *ANTS*, ser. Lecture Notes in Computer Science, vol. 3076. Springer, 2004, pp. 411–425.

[TPY+19] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, "Protecting accounts from credential stuffing with password breach alerting," in *USENIX Security Symposium*. USENIX Association, 2019, pp. 1556–1571.

[YKPB13] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino, "Single-database private information retrieval from fully homomorphic encryption," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 5, pp. 1125–1134, 2013.

Appendix

A. Application to Existing HE Schemes

In this section, we discuss instantiations of the PIR approaches from Section III with specific homomorphic encryption schemes. In particular, we consider additive ElGamal [Gam85], Paillier/Damgård–Jurik [DJ01], and FV [FV12], the constructions of which we overview next.

a) Additive ElGamal [Gam85]: 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 $(c_1c'_1, c_2c'_2)$ encrypts $m_1 + m_2 \bmod p$. 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 $F \geq 2$.

b) Paillier/Damgård–Jurik [DJ01]: Let $N = pq$ be an RSA modulus. The Damgård–Jurik generalization of the Paillier cryptosystem [Pai99] 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 III-B). Using the notation of Section III-B, after Step (1), the server obtained $n^{1/2}$ ciphertexts $c_i \in \mathbb{Z}_{N^{s+1}}$. It can then parse this ciphertext as a plaintext element for a Damgård–Jurik scheme with parameter $s+1$; assuming the selection vector $\vec{s}_1$ is encrypted under such a scheme, it can then compute $c' = \mathsf{Enc}_{s+1}(\mathsf{sk}, \langle \vec{s}_1, \{c_i\}_i \rangle) \in \mathbb{Z}_{N^{s+2}}$. In particular, assume a database with elements in N^k. The communication after d levels of recursion, where $1 \leq d \leq \log n$, is:

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

TABLE VII

BOUNDS ON PLAINTEXT SIZE, EXPANSION, AND DECRYPTION COST.

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

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 [GR05, p. 808].

c) FV [FV12]: The description of FV is given in Section IV-A; 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 III-B. The size of a ciphertext is given by $|ct| = 2N \log q$. In particular, the communication after d levels of recursion, for $1 \leq d \leq \log n$ is

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

However, since FV is also somewhat (and fully) homomorphic, we can apply the PIR protocols of Sections III-C and III-D. 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.

B. (One-Time) Key Information Size

The communication costs both in Table II as well as the SealPIR paper are the communication costs *per query*, assuming the server knows the Galois Keys that will be required to perform the Substitute algorithm that is used in the oblivious expansion algorithm. Similarly for MulPIR, we also require the client to send one additional key-switching key to perform the homomorphic multiplication. Note that all this key information does not depend on the index that is queried, and can be generated beforehand/offline by the client, and reused for multiple query. The communication cost of such key information are provided in Table IX.

Note that SealPIR requires to send $\log N$ Galois keys, where each Galois key is consists of $\log_2 q/3$ ciphertexts; hence it is possible to use two of the optimizations from Section IV-D: sending a seed rather than a random polynomial, and bit-dropping (in practice $b = 5$ bits are dropped). Note that the size of the Galois keys in MulPIR may be higher than in SealPIR. Indeed, as explained in Table II, the number of coefficients and size of moduli depends on the recursion depth.

We propose an optimization that trades computation for communication, as follows. Instead of sending $\log N$ Galois keys, it is possible to send one Galois keys only (a generator of the Galois group) and apply it repeatedly. For example, for any substitution $m(x) \mapsto m(x^{2^j+1}), j \leq \log N$ that we need to perform during oblivious expansion, the substitution $m(x) \mapsto m(x^5)$ can be applied repeatedly to get all possible substitution powers. This enables to reduce the number of keys to send from $\log N$ to 1 Galois key.

C. 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.

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

TABLE IX
SIZE OF ONE-TIME KEYS REQUIRED FOR SEALPIR AND MULPIR.

Keys	Recursion	Size (kB)
SealPIR Galois keys	2, 3	6758
MulPIR Galois keys	2	5707
MulPIR Galois keys	3	28270
MulPIR Galois key generator	2	518
MulPIR Galois key generator	3	2356
MulPIR Switching key	2	19
MulPIR Switching key	3	59

1) Keyword PIR using Cuckoo Hashing: The traditional setup for PIR constructions assumes that the database entries have public indices which are known to the client submitting queries. In particular, these indices coincide with the domain of all possible queries for the client. Under this assumption the size of the database is equal to the query domain size, which directly affects the computation and communication costs of the constructions which depend on the database size. In cases when the server database is sparse and only a small fraction of the domain indices correspond to actual entries, using a PIR solution directly will incur a large overhead forcing dependence on the whole domain size. This sparse database setting has been considered as keyword PIR by Chor et al. [CGN98]. The idea of this work is to build an efficiently searchable structure, instantiated with a search tree, over the sparse indices of the database entries and then use PIR to execute the search queries. This approach requires logarithmic number of PIR queries on a database of proportional to the number of sparse items. We propose a new construction which leverages Cuckoo hashing and reduces the overhead to a constant number of PIR queries on a database proportional to the number of data entries.

The idea of our approach is to use Cuckoo hashing to compress the index on the server side. Cuckoo hashing [PR04], [FPSS05] is a dictionary with worst case constant look-up time, which has size linear in the number of inserted items. A Cuckoo hash table is defined by κ hash functions $\mathsf{H}_1, \ldots, \mathsf{H}_\kappa$ and each item with label i is placed in one of the κ locations $\mathsf{H}_1(i), \ldots, \mathsf{H}_\kappa(i)$. The Cuckoo hash table is initialized by inserting all items in order, resolving collisions using a recursive eviction procedure: whenever an element is hashed to a location that is occupied, the occupying element is evicted and recursively reinserted using a different hash function. For each sequence of items, there is a small set of hash function sets that are incompatible with the sequence and cannot be used to distribute the items, but this can be handled by choosing new hash functions. Overall, inserting n elements into a cuckoo hash table can be performed in expected $O(n)$ time [PR04]. Note that with this procedure the hash functions are dependent on the items placed in the Cuckoo hash table but—unlike in PSI protocols based on Cuckoo hashing [CHLR18], [PSSZ15], [PSWW18], [DRRT18b]—this is not an issue for our use of Cuckoo hashing in the context of PIR where the data is considered public and we do not need to provide any privacy guarantees for it.

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

We note that our approach to compress the server index using Cuckoo hashing is orthogonal to the use of Cuckoo hashing to batch multiple PIR queries described in Appendix C3. Combining these two techniques we optimize on two different axis of the PIR construction. Next we present the formal construction for PIR on sparse data.

Construction 1. *Let* $(\mathsf{Cuckoo.KeyGen}, \mathsf{Cuckoo.Query}, \mathsf{Cuckoo.Insert})$ *be a Cuckoo hash scheme and* $(\mathsf{PIR.Query}, \mathsf{PIR.Eval})$ *be a PIR scheme. We construct a new PIR scheme* $(\mathsf{PIR'.Query}, \mathsf{PIR'.Eval})$ *where the indices of the server's database are sparse over the whole domain:*

- *Preprocessing: 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* $[\mathsf{T}_j[\mathsf{H}_j(i)], \perp] \leftarrow \mathsf{SPIR.Eval}([q_i^j, \mathsf{T}_j])$ *for* $j \in [\kappa]$. *The client checks if any of the* $\mathsf{T}_j[\mathsf{H}_j(i)]$, $j \in [\kappa]$ *contains item* i. *If the items is present, the client outputs it and otherwise, the client outputs* $\perp$.

2) Symmetric PIR from OPRFs: The security requirements of a PIR protocol pertain only to the privacy of the query. Symmetric private information retrieval (SPIR) [GIKM98] 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) [FIPR05], [JL09]. To execute a SPIR query the client and the server execute the corresponding PIR query on the database of encrypted entries and in addition to this they run an oblivious PRF evaluation that enables the client to get a single decryption key corresponding to the query entry. We present our protocol next.

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

- *Preprocessing: 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* $[\mathsf{K}_i, \bot] \leftarrow \mathsf{PRF.OblivEvaluate}([i, \mathsf{K}])$.
3) *The client retrieves its output* $\mathsf{D}[i] = \mathsf{Dec}(\mathsf{K}_i, \tilde{\mathsf{D}}[i])$.

We note that handling sparse data in the setting of SPIR, requires to use oblivious Cuckoo hashing where the hash function parameters are independent of the data inserted in the hash table. Achieving oblivious Cuckoo hashing requires addition a of stash of size $O(\log n)$ that stores items which could not be allocated in the hash table due to collisions [KMW09]. The SPIR construction for sparse data proceeds as follows: the server generates a PRF key K and hash functions for oblivious Cuckoo hashing, it encrypts each item i in its database with key $\mathsf{PRF.Eval}(\mathsf{K}, i)$, the server initializes the oblivious Cuckoo hash with the encrypted data. The server sends the Cuckoo hash functions and the encrypted stash to the client. The client executes a query for item i by running two SPIR queries for $\mathsf{H}_1(i)$ and $\mathsf{H}_2(i)$ using the SPIR construction above. It uses the decryption key $\mathsf{PRF.Eval}(\mathsf{K}, i)$ it has obtained to try to decrypt both the answers in the SPIR queries as well as the encrypted items in the stash. The communication related to the stash can be amortized across different queries.

3) Multi-Query PIR: The traditional definition of PIR considers a setting where queries are executed independently one by one. However, there are scenarios where several queries may be available to be executed at the same time. Multi-query PIR solutions aim to leverage the capability for parallel execution of such queries in order to amortize the complexity. We leverage two main types of techniques for batching: probabilistic batch codes based on Cuckoo hashing [PR04], which have been used in the context of PIR [ACLS18] and private set intersection [PSZ18], [DRRT18a], as well as a CRT batching technique introduced by Groth et al. [GKL10] for Gentry–Ramzan.

D. 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 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.

E. Implementation of Full Recursion

In this section, we report on an implementation of *full* recursion $d = \log n$, using the technique from Section III-D-c. We use the Seal library [sea19a] with polynomials of degree 8192 and a modulus q of 147 bits (product of three 49-bit moduli), and plaintext space $t = 2$. We implemented full recursion for a Pung-style databases of n elements of 288B (in particular, we will have one element per ciphertext) [AS16] and provide benchmarks for databases of size 2^{14} to 2^{17} in Table X.[1] While this approach does not bring any benefit in practice compared to recursion $d = 2$ using one homomorphic multiplication, we report for the first time benchmarks for PIR with full recursion.

TABLE X
FULL RECURSION USING SEAL-MULPIR.

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

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