Private Stateful Information Retrieval

Sarvar Patel*[1], Giuseppe Persiano†[1,2], and Kevin Yeo‡[1]

[1]Google LLC
[2]Università di Salerno

Abstract

Private information retrieval (PIR) is a fundamental tool for preserving query privacy when accessing outsourced data. All previous PIR constructions have significant costs preventing widespread use. In this work, we present private *stateful* information retrieval (PSIR), an extension of PIR, allowing clients to be stateful and maintain information between multiple queries. Our design of the PSIR primitive maintains three important properties of PIR: multiple clients may simultaneously query without complex concurrency primitives, query privacy should be maintained if the server colludes with other clients, and new clients should be able to enroll into the system by exclusively interacting with the server.

We present a PSIR framework that reduces an online query to performing one single-server PIR on a sub-linear number of database records. All other operations beyond the single-server PIR consist of cryptographic hashes or plaintext operations. In practice, the dominating costs of resources occur due to the public-key operations involved with PIR. By reducing the input database to PIR, we are able to limit expensive computation and avoid transmitting large ciphertexts. We show that various instantiations of PSIR reduce server CPU by up to 10x and online network costs by up to 10x over the previous best PIR construction.

*`sarvar@google.com`
†`giuper@gmail.com`
‡`kwlyeo@google.com`

Contents

1 Introduction

Private Information Retrieval (PIR) [18, 35] is a very important privacy-preserving primitive for accessing data held by untrusted servers. PIR enables clients to retrieve a record stored in a database hosted by an untrusted server without revealing the record's identity to the server and it has been used as a critical component in several systems such as friend discovery [11], publish-subscribe [17], advertisement [25] and media consumption [27]. In this work, we are interested in single-server PIR with computational security as first considered by Kushilevitz and Ostrovsky [35].

Previous works on single-server PIRs considered stateless clients as well as a stateless server that do not maintain information between queries and this implies the following three important properties.

- **Parallel Access**: The same PIR server can be simultaneously used by several independent clients.
- **Dynamic User Sets**: The set of clients does not need to be chosen ahead of time. New clients can be added to the system without the intervention of existing clients and clients that crash can easily recover and continue to use the system.
- **Privacy Against Other Clients**: Privacy for record retrieval is guaranteed when the adversarial honest but curious server is colluding with (possibly all) other clients.

Despite its wide applicability, PIR has not been used in applications because of its high costs. In a PIR protocol, the server is required to perform at least one operation for each database record. If a record is not involved in the computation of the reply, then it is certainly not the one sought by the client. In practice, things are even worse as, in most implementations of single-server PIR, the server performs a linear number of expensive public-key operations.

In this paper, we introduce the concept of a *Private Stateful Information Retrieval* (PSIR), that extends the concept of a PIR without losing any of the three desirable properties of PIR described above. We give implementations of PSIR that drastically reduce communication and the server's computational overhead compared to PIR. Our main modification of PIR and PSIR lies in the fact that clients are stateful. That is, a client may store information between queries. The server of PSIR is stateless except for the database like in PIR. Despite having a state, clients of PSIR execute retrieval operations independently. Each retrieval only affects the state of the client performing the retrieval and does not impact other clients in any way. Together with the fact that the server is stateless, PSIR supports parallel access for multiple client without the need to deal with concurrency issues. Furthermore, the initial state of a client is obtained by the client by interacting solely with the server. Therefore, new clients (as well as crashed clients that lose their state) can enter a PSIR scheme at any time without affecting existing clients. For security, we wish to provide privacy for the client against both the server as well as other possibly adversarial clients. Formally, PSIR ensures that the identity of any records retrieved by any honest client remains private from the adversary even if the server is colluding with all other clients. Overall, PSIR maintains the three desirable properties of PIR while the availability of a client state in PSIR will significantly improve the efficiency of private retrievals as our constructions will show.

1.1 Our Contributions

We show that PSIR is more efficient than PIR constructions by giving a general construction for computationally secure PSIR with practical instantiations. More precisely, let n be the number of records in the database, let k be an adjustable parameter, $0 < k < n$, and suppose the client has memory to store c records. Our main technical contribution consists in a reduction of a query in PSIR to an online phase consisting of one blackbox PIR query on a database of n/k records and the transfer of k seeds and k integers from client to server. Both the client and server perform an additional $O(n)$ hash evaluations and other simple operations. The reduction assumes that the client has already performed an offline initialization phase with $O(n)$ communication that can be amortized over c queries thus giving $O(n/c)$ amortized communication.

As we have already pointed out, the dominating costs in practical PIR constructions is constituted by public-key operations executed by the server, usually over homomorphic ciphertexts. Previous PIR constructions all required the server to execute at least n public-key operations per query. Our PSIR scheme replaces most of these expensive public-key operations with simple, efficient operations. We stress that our reduction does not add any other public-key operation on top of the ones needed to execute PIR on a sub-linear sized database. We note that our server still performs $O(n)$ computation as otherwise our construction would not guarantee security. As an example, we can concretely instantiate our construction by setting $k = \sqrt{n}$ which means only $O(\sqrt{n})$ public-key operations are required. We construct PSIR using XPIR [5] as well as a PIR construction built from the Paillier cryptosystem [42]. In addition, we estimate costs for PSIR with SealPIR [7] using their reported results.

Our concrete results are two-fold. We present a construction of PSIR that outperforms the previous, best PIR construction in important resource costs for databases with 100K to 1M 288 byte records (a standard database size experiment used in previous works [7]). Specifically, we show that PSIR with Paillier outperforms SealPIR by giving up to 4x speedup in server CPU, a 5-10x reduction in online bandwidth and a 1.3-4x decrease in amortized total bandwidth. Furthermore, more than 50% of the total bandwidth can be performed offline during cheaper, non-busy hours. We remark though that PSIR with Paillier increases client CPU usage compared to SealPIR. In our opinion, these trade-offs are beneficial in concrete terms. For 1M items, total client CPU increases to only 660 milliseconds while server CPU decreases by more than 4.5 seconds. To achieve these benefits, the client uses up to 525 KB of state. We note that 525 KB is less than 1/12 the network costs of generic XPIR and is less than twice the network costs of generic SealPIR. On the other hand, SealPIR requires the server to store 2.9 MB of auxiliary cryptographic material for each client whereas PSIR with Paillier does not require any extra storage beyond database.

In addition, we show that PSIR variants using XPIR and SealPIR reduce resource costs compared to generic XPIR and SealPIR. In terms of concrete savings, PSIR with XPIR results in a 12-28x speedup in server CPU, up to 20x less online network costs and 17x less amortized network costs compared to generic XPIR. Our estimates for PSIR with SealPIR result in an up to 10x speedup in server CPU, but a slight increase in online network costs and an up to 50% increase in amortized network costs compared to generic SealPIR. Additionally, PSIR introduces an increase in client CPU of at most 660 milliseconds in both cases. We still view the trade-offs as excellent since, in concrete numbers, server CPU is decreased by at least 4 seconds in both cases for 1M items. For network costs, the benefits are clear for PSIR with XPIR. While network costs for PSIR with SealPIR increase, the majority of the extra network costs can be performed offline during off-peak periods. PSIR requires up to 525 KB of state in all cases.

1.2 Relation to Other Privacy-Preserving Storage Primitives

In the previous section, we presented PSIR as an extension of PIR. Many other privacy-preserving storage and retrieval primitives have been considered in previous works, We compare PSIR with ORAM as well as extensions of PIR and ORAM and present a summary in Table 1.

ORAM *Oblivious RAM* (ORAM) [23, 24] guarantees also privacy of the blocks in addition to privacy of the access pattern. This implies that the client state contains the secret key used to decrypt the block and, if any client were corrupted by the server then full privacy would be lost. The secret key enables ORAM to use sublinear computation and bandwidth (the current, best ORAM construction uses $O(\log n \cdot \log\log n)$ bandwidth and computation [43]). However, since the server does not have the secret key, enrollment of new clients would involve existing clients. In addition, in most ORAM construction the client also has a *location map* mapping logical block identity to physical server locations which dynamically changes as physical blocks are accessed by the client. The presence of a location map would make parallel access very inefficient as performing queries in parallel requires complex algorithms using expensive concurrency primitives to enable access to shared resources.

Several previous works have considered extending the ORAM scheme to enable access to a large group of clients [34, 37, 38, 49]. One way to extend ORAMs to multiple clients is to store a separate ORAM scheme for each client. Each client is given the secret key corresponding to their own ORAM scheme. As a result, a client may only access their own ORAM and an adversarial server colluding with other clients will not

Primitive	Multiple Client Access	Linear Server Storage	Add New Clients/ Recover from Crash	Stateful Server	Stateful Client(s)
PIR [5, 7, 35]	✓	✓	✓		
PSIR (This Paper)	✓	✓	✓		✓
ORAM [23, 24, 43]		✓		✓	✓
Multiple Client ORAM [34, 37, 38, 49]	✓		✓	✓	✓
Symmetric-Key DEPIR [12, 15]					✓
Public-Key DEPIR [12]	✓		✓		
PANDA [28]	✓			✓	✓

Table 1: This table compares single-server privacy-preserving storage systems. Multiple client access refers to systems that enable accessibility to large groups of clients while providing privacy against subsets of colluding clients. Add new clients/recover from crash refers to servers with the capability to enroll stateless clients into the system.

compromise privacy for honest clients. New clients may also be added by constructing new ORAM schemes. However, this scheme requires large storage since the database is replicated once for each client unlike PSIR that stores the database exactly once.

Another way to enable access to multiple clients is splitting the ORAM (as well as the secret key) into shares and distributing the shares to several non-colluding servers. To retrieve/overwrite a record, the client performs a protocol with all non-colluding servers to generate an access to the shared ORAM scheme. While these schemes are efficient, they require the strong assumption of multiple, non-colluding servers whereas PSIR focuses on the single-server setting. ORAM and its multiple client extensions do not provide practical solutions for enabling access to large groups of clients efficiently and privately compared to PSIR.

Doubly Efficient PIR *Doubly Efficient PIR* (DEPIR) refers to an extension of PIR that allows a setup phase which preprocesses the database before any retrieval requests. Beyond the preprocessed database, the server will be stateless. DEPIR schemes are categorized into symmetric-key and public-key variants. In the symmetric-key variant, each client must use a secret key to access the encoded database. On the other hand, the public-key variant is accessible by all clients. The first public-key DEPIR construction was presented by Beimel *et al.* [9] in the multiple, non-colluding server setting that is information-theoretically secure. More recently, Boyle *et al.* [12] and Canetti *et al.* [15] present symmetric-key DEPIR schemes in the single-server setting using locally decodable codes built from Reed-Muller codes [41, 45]. In addition, Boyle *et al.* [12] present a public-key DEPIR scheme using a variant of obfuscation. All above DEPIR schemes ensure that both the client and server perform a sub-linear number of operations compared to the database per retrieval.

Both symmetric-key DEPIR and ORAM share the same problems involving the secret key that must be shared by all clients that require access to the database. Any extensions to multiple clients will suffer the same problems as multiple client ORAM. On the other hand, public-key DEPIR provides many useful properties that are offered by PSIR. However, all public-key constructions are built using very strong assumptions. The scheme of Beimel *et al.* [9] requires multiple non-colluding servers while the scheme of Boyle *et al.* [12] relies on non-standard assumptions involving obfuscation while, also, requiring server storage super-linear in the database size. Therefore, DEPIR schemes are not a practical solution for privacy-preserving data access due to their reliance on strong assumptions.

Private Anonymous Data Access Hamlin *et al.* [28] have recently introduced the *Private Anonymous Data Access* (PANDA) primitive. PANDA is an extension of symmetric-key DEPIR where the server is stateful and maintains information between multiple retrievals. In addition, PANDA requires anonymity. That is, the adversary cannot learn the identity of the querying clients which can be enforced by requiring clients to use anonymous mix networks [16]. Anonymity is an extra privacy feature of PANDA that is not provided by PSIR. Using leveled fully homomorphic encryption [13] and Reed-Muller codes [41, 45], Hamlin *et al.* [28] present both a read-only and public-write PANDA scheme. Public-writes are an operation that are beyond the scope of PSIR, so we focus on the read-only variant. Their PANDA scheme must be built using a collusion parameter that determines the maximum size of the subset of colluding clients before privacy becomes compromised. Both client and server operations as well as server storage grow linearly in the collusion parameter. On the other hand, the client and server operations for a private retrieval scale poly-logarithmically in the database size.

In relation to PSIR that simply stores the database in plaintext, their PANDA scheme requires server storage super-linear in the size of the database even when providing no protection against colluding clients. If we wish the PANDA scheme to be secure in the setting that the server colludes with all clients except one, the server would require storage larger than simply replicating the database for each client. In comparison, our construction of PSIR outlined in Section 1 guarantees privacy even when the server colludes with all clients except one without sacrificing computational or storage efficiency. Also, PANDA does not allow enrolling stateless clients into the system by only interacting with the server unlike PSIR. As a consequence, PANDA does not provide a practical solution.

2 Technical Overview

Our construction is inspired by the construction of PIR with side information of Kadhe *et al.* [33] that considers the following hypothetical scenario. Suppose the client has obliviously obtained some *side information* consisting of $k-1$ records (thus the server is unaware of which records are known to the client). Can the client leverage on this side information to privately access other records more efficiently? Kadhe *et al.* [33] show that the client can randomly partition the database of n records into n/k parts of size k such that the desired query record and all $k-1$ records of side information are in the same parts. The server will add all records (that are represented as elements in a field) within the same parts and return the n/k record sums to the client. The client discards all sums except the sum of the desired record and the side information. By subtracting the side information, the client successfully retrieves the desired record. However, Kadhe *et al.* do not provide concrete algorithms to privately obtain side information. Furthermore, the network costs of communicating the partition and downloading all n/k sums are significant.

In this paper, we present a construction of PSIR which addresses these issues. Our PSIR relies on two main building blocks that are of independent interest outside of PSIR: *oblivious constrained partitions* and *private batched sum retrieval*. In addition, PSIR also uses a single-server PIR scheme.

- *Oblivious Constrained Partitions (*OCP*).* A OCP consists of two algorithms: the *construction* algorithm and the *expansion* algorithm. Given a *constraint* subset $S \subseteq [n]$ of size k, the *construction* algorithm OCP.GenerateSeed of an OCP returns a succinct representation of a partition of $[n]$, $(P_1, \ldots, P_m)$, into $m = n/k$ parts, each of size k, such that one of the m parts is equal to the constraint subset S. The obliviousness property guarantees that an adversary cannot correctly guess whether the input constraint subset S contains a given $q \in [n]$ better than guessing at random. The *expansion* algorithm OCP.ExtractPartition, given a succinct description of a partition, produces the actual partition. We give a construction algorithm that outputs a description of a partition consisting of k seeds of length equal to the security parameter λ and k integers in $[n]$. Both the constructing and expanding the description require $O(n)$ PRF evaluations.

- *Private Batched Sum Retrieval (*PBSR*).* Given c subsets $S_1, \ldots, S_c \subseteq [n]$, the client wishes to privately retrieve c sums corresponding to the sum of all records in each subset. That is, for a database $D = (\mathsf{B}_1, \ldots, \mathsf{B}_n)$, the client wishes to retrieve the c sums: $O_1 = \sum_{i \in S_1} \mathsf{B}_i, \ldots, O_c = \sum_{i \in S_c} \mathsf{B}_i$. This is an extension of private batched retrieval [30] where only records, instead of sums of records, are retrieved. For privacy, the adversary should not learn the c input subsets.

PSIR composes these primitives as follows. Assume the records are elements of some finite fields and that the client has storage sufficient for containing c records. At the initialization phase, the client and the server execute the private batched sum retrieval protocol so that the client privately retrieves c sums of records,

	K_1,v_1	K_2,v_2			
P_1	96	55		...	
P_2	21	41		...	
P_3	35	2	17	...	28
...	...	...	...	...	...
$P_{n/k}$	70	62		...	

Figure 1: The constraint subset is embedded into the third row. The pairs $(\mathcal{K}_1, v_1)$ and $(\mathcal{K}_2, v_2)$ succinctly represent the first two ordered column subsets. The remaining columns have not been generated yet.

$s_1, \ldots, s_c$, where sum s_i is over the set S_i of $k-1$ randomly chosen records from the database. Now, suppose the client wishes to retrieve record q and let S_i be the next unused set that does not contain q. Using the construction algorithm of an OCP with $S = S_i \cup \{q\}$ as constraint set, the client constructs an ordered partition $\mathcal{P} = (P_1, \ldots, P_m)$ and let j be such that $P_j = S$. The client sends the succinct description of $\mathcal{P}$ to the server. Upon receiving the description of the partition, the server expands it and constructs a database where, for $l = 1, \ldots, m$, the l-th record is the sum p_l of all the records with index in part P_l. The client in addition constructs a query, using the underlying single-server PIR, to retrieve p_j and sends the query to the server. Then, the server executes the PIR query using $(p_1, \ldots, p_m)$ as the database and returns the PIR response to the client. Finally, the client decrypts the PIR response to retrieve p_j and obtains the sought record by computing $p_j - s_i$. Once the client runs out of S_i, the client will execute the private batched sum retrieval protocol to gain new side information. Additionally, a client without state (such as a new client or a crashed client that lost state) may execute a private batched sum retrieval protocol with the server to start performing retrievals.

Let us now give informal descriptions of our constructions for OCP and PBSR.

Oblivious Constrained Partitions. Sampling a random partition of $[n]$ into m parts each of size k is equivalent to creating a random assignment of the integers of $[n]$ to the n cells of an $m \times k$ matrix M, with each row corresponding to a part. The partition can be succinctly described by the random seed to a PRF that is to be used to generate the randomness of an agreed upon procedure for sampling a random permutation, like, for example, Fisher-Yates [21]. The randomly sampled permutation is then used to assign integers to the cells of the $m \times k$ matrix. However, we have two additional goals. The first is of having a particular row be constrained to given k values from the *constraint set* S. This could be easily achieved by using Fisher-Yates to generate a permutation of $n-k$ values and then explicitly add the k constrains values to the seed of the PRF used for Fisher-Yates. However, this method does not satisfy our privacy requirement that the description of the partition be oblivious to the constraints.

Let us consider the following warm-up construction based on a family P of pseudorandom permutations. The construction algorithm randomly selects the *constraint row* $1 \le r \le m$ that will contain the k elements $\{s_1, \ldots, s_k\}$ of the constraint set S and generates a pseudorandom permutation key for each column. For the first column, a random key k_1 is selected and the algorithm computes $\rho_1 := \mathsf{P}^{-1}(k_1, s_1)$. The construction algorithm then evaluates $\mathsf{P}(k_1, \cdot)$ at values $\rho_1 - 1, \rho_1 - 2, \ldots,$ so to obtain values $M[r-1, 1], \ldots, M[1, 1]$. We let σ_1 be the value such that $M[1,1] = \mathsf{P}(k_1, \sigma_1)$; clearly, for column 1, we have $\sigma_1 = \rho_1 - (r-1)$. In addition, the algorithm evaluates $\mathsf{P}(k_1, \cdot)$ at values $\rho_1 + 1, \rho_1 + 2, \ldots$ to obtain $M[r+1, 1], \ldots, M[m, 1]$. Once the first column $T_1 = M[\star, 1]$ has been computed, the construction algorithm checks that it does not contain values from the constrained row other than s_1 and, if it does, a new key k_1 is randomly selected and the process is repeated until successful. The first column T_1 is succinctly specified by the key k_1 and by the integer

σ_1. The expansion algorithm reconstructs the column by evaluating $\mathsf{P}(k_1, x)$ for $x = \sigma_1, \ldots, \sigma_1 + (m-1)$. For the second column, the construction algorithm randomly selects key k_2 and does the same: that is, it sets $\rho_2 := \mathsf{P}^{-1}(k_2, s_2)$ and then evaluates the PRP from ρ_2 going back until the value $M[1,2]$ in row 1 is computed and forward until the value $M[m,2]$ is computed. As before, the algorithm makes sure none of the remaining constrained row values are used. If a value from the constrained row appears, then a new key k_2 is randomly selected and the selection of column 2 starts again. Furthermore, it may be the case that a generated value for column 2 appears in column 1. In this case, another key k_2 is generated randomly and the algorithm repeats. The expansion algorithm, once column 1 has been reconstructed, obtains column 2 by repeatedly applying $\mathsf{F}(k_2, \cdot)$ starting from σ_2; if during the reconstruction a value from column 1 appears, it is ignored and the process continues until m values distinct from those appearing in column 1 are obtained. The process continues until all k columns have been specified.

There are two problems with this scheme: firstly, in the later columns, we will have several collisions with values from previous columns and this will require extra invocations of the PRP. We can avoid this by using an indirection. Rather than treating the outputs of the PRP as permuted values themselves, we will treat them as pointers in a table T of unused values. Initially, T has all n unused values and is ordered from 1 to n. For column 1, all PRP evaluations are pointers to values in T. For example, if the evaluation at row 5 results in 11 then the actual value for row 5 is read from table as $T[11]$ which, for column 1, happens to be 11 also since all values are unused. For the second column, we first remove all the values from T that were used for column 1 and now have a modified T with $n - m$ ordered values. The domain of the PRP for the second column will be $1, \ldots, n-m$. This way we can avoid generating new permutation keys by colliding with values from previous columns and we only need to check that values from the constrained row values are not used. The client continues this process for all k columns.

The second problem with PRP is that instantiating them securely is quite expensive: in practice one would use PRP based on block ciphers on small domains that can be constructed from secure block ciphers at the cost of multiple evaluations (see, for example, [40, 46]). Hence, we provide our main construction below that only uses PRFs and is very efficient. Briefly, for each column, the PRF F will generate m different values that are treated as a pointer to get unused values from table T. This can be done by repeatedly evaluating the F in increasing index until m different values appear. For convenience, we will assume the first m invocations of F are unique. As a result, $\mathsf{F}(\mathcal{K}, i)$ will actually be the i-th unique value appearing when evaluating F in increasing index. As the table size or domain size changes column to column, the PRF output can be simply mod-ed by the new domain size to get appropriate values, as long as the original output of the PRF was large enough to keep the biases negligible. However, we still have to guarantee that the element in the constrained row of each column is an element from the constrained set S. Unlike a PRP, we cannot invert the PRF and find the correct index to start evaluating. Instead, the algorithm will use an augmented description for the columns that consists of the key k along with a *displacement* v. The displacement v is computed to ensure that the r-th unique value produced by each column PRF results in a specified index of T corresponding to a value from the constrained set S. Thus, the expansion algorithm will set the element in row 1 of a column described by $(\mathcal{K}, v)$ as the one in position $\mathsf{F}(\mathcal{K}, 1) + v$ of table T (with wrap-around if necessary). The algorithm chooses v so that $\mathsf{F}(\mathcal{K}, r) + v$ is the current index in T of the constraint set S that is destined for row r of the column being computed. The reconstruction algorithm is straightforward: the partition is created column by column, using the key to instantiate the PRP and encrypt forward to create pointers to which the displacement v is added to get index for the table T. After each column, the server removes the used values from the table and the PRF works on a smaller domain for the next column.

The algorithm described above requires maintaining an ordered table of the *unused* elements, that is the elements that have not been assigned to a part yet, and this would require linear storage. However, we observe that only the ranks of the unused elements of the constraint set need to be maintained. For each element s_i from the constraint set, the algorithm maintains the number of unused elements that are smaller than s_i. Indeed, after generating a column, the algorithm only needs to check that the column contains the rank of exactly one unused constraint subset element. Therefore, we only explicitly store the rank of all constraint subset elements that are used. Once a new column is obtained, the algorithm updates the ranks of the remaining unused items. As a consequence, this algorithm requires $O(k)$ storage to maintain the ranks and an additional $O(m)$ storage to explicitly store the generated column.

Private Batched Sum Retrieval We present several different private batched sum retrieval protocols with both practical and theoretical interest. Recall as input, the protocol receives c subsets $S_1, \ldots, S_c \subseteq [n]$. The protocol with the best performance on practical database sizes is also the simplest scheme, which simply downloads the entire database using n records of communication.

A more complex algorithm makes use of a private batched retrieval (PBR) scheme [7, 22, 26, 30]. The client constructs the set of records used in the c sums, $S = S_1 \cup \ldots \cup S_c$. Additional records are added to S until S has $|S_1| + \ldots + |S_c|$ records. Afterwards, the client simply executes a PBR scheme to retrieve all records in S and adds the records locally. This scheme requires only $O(|S| \cdot \mathsf{poly}\log(n))$ records of communication and $O(n)$ CPU computation. This scheme makes sense for scenarios where network costs are significantly higher than computation costs. In Appendix E, we present more protocols of theoretical interest that use only $O(c \cdot \mathsf{poly}\log(n))$ records of communication.

Paper Organization In Section 3, we present the definitions of oblivious partitioning, private batched sum retrieval and private stateful information retrieval. We present our OCP, PBSR and PSIR constructions in Sections 4, 5 and 6 respectively. The results of our experiments are shown in Section 7.

3 Definitions

We suppose the database D contains n records denoted $\mathsf{B}_1, \ldots, \mathsf{B}_n$. We will interchangeably use the terms records and blocks. We use $[n]$ to denote the set $\{1, \ldots, n\}$. For a two-party protocol `Prot`, the writing $(y_1, y_2) \leftarrow \texttt{Prot}(x_1, x_2)$ denotes that when the protocol is played by party 1 with input x_1 and party 2 with input x_2 then party 1 receives y_1 as output whereas party 2 receives y_2.

For an algorithm A that interacts with the server, we denote the transcript by $\mathsf{Trans}[y \leftarrow A(x)]$ as all the information revealed to the server by executing A to compute output y on input x. The transcript includes all data uploaded to the server as well as the sequence of data accesses performed when executing A.

3.1 Private Stateful Information Retrieval

In this section, we define the notion of *Private Stateful Information Retrieval* (PSIR). Roughly speaking, PSIR is an extension of the classical notion of a single-server PIR [35] in which the client keeps a state between queries. Before any query can be issued, the client initializes its state by executing the Init protocol with the server. As in PIR, the query process of a PSIR consists of a single client-to-server message followed by a single server-to-client message. The client will use the server's response and their current state to recover the record desired. For the sake of clarity, we split this last step into two distinct parts: an Extract algorithm executed by the client to extract the record queried from the server's reply, and an UpdateState algorithm jointly executed by the client and server to update the client's state. In our construction, the UpdateState protocol consumes part of the client's state and, if needed, re-executes the Init protocol. We stress that the server of a PSIR does not have a state just like in the original notion of a single-server PIR.

Definition 1 (PSIR)**.** A *Stateful Information Retrieval* PSIR = (PSIR.Init, PSIR.Query, PSIR.Reply, PSIR.Extract, PSIR.UpdateState) consists of the following components

- $(\mathsf{st}, \perp) \leftarrow \mathsf{PSIR.Init}((1^\lambda, 1^c), (1^\lambda, D = (\mathsf{B}_1, \ldots, \mathsf{B}_n)))$: a protocol executed by a client and the server. The client takes as input the security parameter λ and the parameter c that describes the number of records that can be stored by the client. The server takes as input the security parameter λ and the database $D = (\mathsf{B}_1, \ldots, \mathsf{B}_n)$. The client's output is its initial state st and the server receives no output. Protocol Init is run when a client enrolls into the system or when a new state is needed by a client because the state has been lost for a crash.

- $(\mathsf{st}', \mathsf{Query}) \leftarrow \mathsf{PSIR.Query}(q, \mathsf{st})$: an algorithm executed by the client that takes as input index $q \in [n]$ and its current state st and outputs Query to be sent to the server. In addition, the client's state is updated to st'.
- $\mathsf{Reply} \leftarrow \mathsf{PSIR.Reply}(\mathsf{Query}, D = (\mathsf{B}_1, \dots, \mathsf{B}_n))$: an algorithm executed by the server that takes as input Query computed by a client and the database and returns Reply.
- $\mathsf{B} \leftarrow \mathsf{PSIR.Extract}(\mathsf{Reply}, \mathsf{st})$: an algorithm executed by the client to extract a record from the server's Reply to the query. Note that the client's state is not updated.
- $(\mathsf{st}, \perp) \leftarrow \mathsf{PSIR.UpdateState}((\mathsf{st}), (D))$: a protocol executed by a client and the server that updates the client's state.

such that the following *Correctness* condition is satisfied.

Correctness For all λ, for all $n \le \mathsf{poly}(\lambda)$, for all $c \le \mathsf{poly}(\lambda)$, for any set H of honest players, for any database $D = (B_1, \dots, B_n)$ of size n and for any sequence $\mathbb{Q} = ((q_1, u_1), \dots, (q_l, u_l))$ of length $l = \mathsf{poly}(\lambda)$ we have

$$\Pr\left[\mathbf{Expt}^{\mathsf{PSIR}}_{\mathsf{Correctness}}(\lambda, c, \mathsf{H}, D, \mathbb{Q}) \neq 1\right] \le \mathsf{negl}(\lambda)$$

where the experiment $\mathbf{Expt}^{\mathsf{PSIR}}_{\mathsf{Correctness}}$ is defined as follows

```
$\mathbf{Expt}^{\mathsf{PSIR}}_{\mathsf{Correctness}}(\lambda, c, \mathsf{H}, D, \mathbb{Q})$;
For all $u \in \mathsf{H}$
  $(\mathsf{st}_u, \perp) \leftarrow \mathsf{PSIR.Init}((1^\lambda, 1^c), (1^\lambda, D))$;
For $i \leftarrow 1, \dots, l$:
  $(\mathsf{st}_{u_i}, \mathsf{Query}_i) \leftarrow \mathsf{PSIR.Query}(q_i, \mathsf{st}_{u_i})$;
  $\mathsf{Reply}_i \leftarrow \mathsf{PSIR.Reply}(\mathsf{Query}_i, D)$;
  $R_i \leftarrow \mathsf{PSIR.Extract}(\mathsf{Reply}_i, \mathsf{st}_{u_i})$;
  $(\mathsf{st}_{u_i}, \perp) \leftarrow \mathsf{PSIR.UpdateState}((\mathsf{st}_{u_i}), (D)]$;
Output 1 iff for all $i \in [l]$ $R_i = B_{q_i}$;
```

To properly define the notion of security for a a stateful information retrieval we need the concept of sequences *compatible* with respect to a set of corrupted players.

Definition 2. Two sequences $\mathbb{Q}^0 = ((q_1^0, u_1^0), \dots, (q_l^0, u_l^0))$ and $\mathbb{Q}^1 = ((q_1^0, u_1^1), \dots, (q_l^0, u_l^1))$ are *compatible* with respect to set C of users if

1. $u_i^0 = u_i^1$, for all $i = 1, \dots, l$;
2. $q_i^0 = q_i^1$, for all i such that $u_i^0 = u_i^1 \in \mathsf{C}$.

We are now ready to present our static notion of security.

Definition 3. A Stateful Information Retrieval is a *Private Stateful Information Retrieval* (PSIR) if for all λ, for all $n \le \mathsf{poly}(\lambda)$, for all $c \le \mathsf{poly}(\lambda)$, for any sets H of honest players and C of corrupted players, for any database $D = (B_1, \dots, B_n)$ of size n, and for any two compatible sequences, $\mathbb{Q}^0$ and $\mathbb{Q}^1$ with respect to C of length $l = \mathsf{poly}(\lambda)$ and for all PPT adversary $\mathcal{A}$, we have that

$$\left|\Pr[\mathbf{Expt}^{\mathsf{PSIR}}_{\mathcal{A}}(\lambda, \mathsf{C}, \mathsf{H}, D, \mathbb{Q}^0) = 1] - \Pr[\mathbf{Expt}^{\mathsf{PSIR}}_{\mathcal{A}}(\lambda, \mathsf{C}, \mathsf{H}, D, \mathbb{Q}^1) = 1]\right| < \mathsf{negl}(\lambda),$$

where experiment $\mathbf{Expt}^{\mathsf{PSIR}}_{\mathcal{A}}$ is defined as follows

```
$\mathbf{Expt}^{\mathsf{PSIR}}_{\mathcal{A}}(\lambda, c, \mathsf{H}, \mathsf{C}, D, \mathbb{Q})$;
$\mathsf{View} \leftarrow \bigcup_{u \in \mathsf{H} \cup \mathsf{C}} \mathsf{Trans}[(\mathsf{st}_u, \perp) \leftarrow \mathsf{PSIR.Init}((1^\lambda, 1^c), (1^\lambda, D))]$;
$\mathsf{View} \leftarrow \mathsf{View} \cup \left(\bigcup_{u \in \mathsf{C}} \mathsf{st}_u\right)$;
Parse $\mathbb{Q}$ as $\mathbb{Q} = ((q_1, u_1), \ldots, (q_l, u_l))$;
For $i \leftarrow 1, \ldots, l$:
  $(\mathsf{st}_{u_i}, \mathsf{Query}_i) \leftarrow \mathsf{PSIR.Query}(q_i, \mathsf{st}_{u_i})$;
  $\mathsf{View} \leftarrow \mathsf{View} \cup \{\mathsf{Query}_i\}$;
  $\mathsf{Reply}_i \leftarrow \mathsf{PSIR.Reply}(\mathsf{Query}_i, D)$;
  $B_{q_i} \leftarrow \mathsf{PSIR.Extract}(\mathsf{Reply}_i, \mathsf{st}_{u_i})$;
  $\mathsf{View} \leftarrow \mathsf{View} \cup \mathsf{Trans}[(\mathsf{st}_{u_i}, \perp) \leftarrow \mathsf{PSIR.UpdateState}((\mathsf{st}_{u_i}), (D)]$;
  If $u_i \in \mathsf{C}$ then $\mathsf{View} \leftarrow \mathsf{View} \cup \{\mathsf{st}_{u_i}\}$;
Output $\mathcal{A}(\mathsf{View})$;
```

If we enforce the restriction that each user's state is empty and that both Init and $\mathsf{UpdateState}$ are empty functions, we attain a security definition for PIR where the server may non-adaptively corrupt a set of users.

Adaptive corruption. In the security definition above, the set of corrupted users C is fixed at the beginning. We can also present a security definition against adaptive corruption by means of the following game between an $\mathcal{A}$ and a challenger $\mathcal{CH}$. The games starts with $\mathcal{CH}$ executing $\mathsf{PSIR.Init}$ for all users u and by picking a random bit η. Then at each step i, $\mathcal{A}$ can decide to corrupt a new player u. The initial state of st_u of u is then given to $\mathcal{A}$ along with all the random coin tosses of u, and all the replies received by u from $\mathcal{CH}$. In addition, $\mathcal{A}$ may issue a query on behalf of user u_i, in which case $\mathcal{A}$ outputs (q_i^0, q_i^1, u_i) and $\mathcal{CH}$ executes query q_i^η for user u_i and gives $\mathcal{A}$ the view of the server and, if $u_i \in \mathsf{C}$, the updated state st_{u_i} of u_i. At the end of the game, $\mathcal{A}$ outputs its guess η' for η. We say that $\mathcal{A}$ *wins* if $\eta = \eta'$ and sequences $\mathbb{Q}^0 = ((q_1^0, u_1), \ldots, (q_l^0, u_l))$ and $\mathbb{Q}^1 = ((q_1^1, u_1), \ldots, (q_l^1, u_l))$ are compatible with respect the set of corrupted players C at the end of the game. A PSIR scheme is an *adaptively private* stateful information retrieval if for any PPT adversary $\mathcal{A}$ the probability that $\mathcal{A}$ wins is negligibly in λ close to $1/2$.

It can be shown that our main construction presented in Section 6 is also adaptively private.

3.2 Oblivious Constrained Partitions

In this section, we define the concept of a *Oblivious Constrained Partition* OCP. An OCP consists of two algorithms ($\mathsf{OCP.GenerateSeed}$, $\mathsf{OCP.ExtractPartition}$). Algorithm $\mathsf{OCP.GenerateSeed}$ takes integers n, k and the *constraint* subset $S \subseteq [n]$ of size k and returns the pair $(\mathcal{K}, r)$ such that $\mathcal{K}$ is the description of a partition $(P_1, \ldots, P_m)$ of $[n]$ into $m := n/k$ ordered parts of size k and r is such that $S_r = S$. Algorithm $\mathsf{OCP.ExtractPartition}$ takes $\mathcal{K}$ and expands it into the partition $(P_1, \ldots, P_m)$. We require that the construction hides the constraint subset S used to sample $(P_1, \ldots, P_m)$, in the sense that an adversary that sees the description of a partition has no information on which of the parts of the partition was used as constraint subset in the generation of the partition. In particular, an adversary given the seed $\mathcal{K}$ cannot determine whether an element q belongs to the constraint subset or not.

Definition 4 (Constrained Partition)**.** A *Constrained Partition* $\mathsf{OCP} = (\mathsf{OCP.GenerateSeed}, \mathsf{OCP.ExtractPartition})$ consists of the following two PPT algorithms:

- $(\mathcal{K}, r) \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, S)$: an algorithm that takes as input security parameter λ, integers n and *constraint* subset $S \subseteq [n]$ of size k, such that k divides n, and outputs a seed $\mathcal{K}$ and an integer r.
- $(P_1, \ldots, P_m) \leftarrow \mathsf{OCP.ExtractPartition}(\mathcal{K})$: an algorithm that takes as input a seed $\mathcal{K}$ and outputs subsets $(P_1, \ldots, P_m)$, with $m = n/k$.

that satisfy the following two properties:

- Correctness. For every n, k, λ and for every $S \subseteq [n]$ of size k, if $(\mathcal{K}, r) \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, k, S)$ and $(P_1, \ldots, P_m) \leftarrow \mathsf{OCP.ExtractPartition}(\mathcal{K})$ then $(P_1, \ldots, P_m)$ is a partition of $[n]$ into m parts each of size k and $P_r = S$.

- Obliviousness. For all λ, all $n = \mathsf{poly}(\lambda)$ and k such that k divides n, for any two elements $q^0, q^1 \in [n]$ and for any PPT adversary $\mathcal{A}$,

$$\left|\Pr[\mathbf{Expt}^{\mathsf{OCP}}_{\mathcal{A}}(\lambda, n, k, q^0) = 1] - \Pr[\mathbf{Expt}^{\mathsf{OCP}}_{\mathcal{A}}(\lambda, n, k, q^1) = 1]\right| < \mathsf{negl}(\lambda),$$

where $\mathbf{Expt}^{\mathsf{OCP}}_{\mathcal{A}}$ is defined as follows

$\mathbf{Expt}^{\mathsf{OCP}}_{\mathcal{A}}(\lambda, n, k, q)$;
Randomly choose $S' \subseteq [n]$ of size $k-1$; Set $S = S' \cup \{q\}$; $(\mathcal{K}, r) \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, S)$; Output $\mathcal{A}(\mathcal{K})$;

3.3 Private Batched Sum Retrieval

A *private batched sum retrieval* is an algorithm that computes the sums of c subsets $S_1, \ldots, S_c$ of n elements from a field $\mathbb{F}$ stored by a potentially adversarial server. In such a scheme, the server should not learn the subsets for which the algorithm is computing the partial sums. In Section 5, we present several different batched sum retrieval schemes with various bandwidth and computational overheads. To analyze communication, we count the number of field elements that are transferred by the algorithm. For computational costs, we count the number of field operations performed.

Definition 5. (Batched Sum Retrieval) A *batched sum retrieval* scheme PBSR is an algorithm that takes as input c subsets, $S_1, \ldots, S_c \subseteq [n]$ and accesses n field elements $D = (\mathsf{B}_1 \ldots, \mathsf{B}_n) \in \mathbb{F}^n$ stored on a server and outputs c sums, $O_1, \ldots, O_c$, where $O_i = \sum_{j \in S_i} \mathsf{B}_j$ for all $i = 1, \ldots, c$.

A batched sum retrieval scheme is *private* if for all λ, for all $n = \mathsf{poly}(\lambda)$, for any $c = \mathsf{poly}(\lambda)$ and for any pair of sequences of subsets $(S_1, \ldots, S_c)$ and $(S'_1, \ldots, S'_c)$ such that $|S_i| = |S'_i|$ for all $i \in [c]$ and for any PPT adversary $\mathcal{A}$,

$$\left|\Pr[\mathbf{Expt}^{\mathsf{PBSR}}_{\mathcal{A}}(\lambda, n, S_1, \ldots, S_c) = 1] - \Pr[\mathbf{Expt}^{\mathsf{PBSR}}_{\mathcal{A}}(\lambda, n, S'_1, \ldots, S'_c) = 1]\right| < \mathsf{negl}(\lambda),$$

where $\mathbf{Expt}^{\mathsf{PBSR}}_{\mathcal{A}}$ is defined as follows

$\mathbf{Expt}^{\mathsf{PBSR}}_{\mathcal{A}}(\lambda, n, S_1, \ldots, S_c)$;
$\chi \leftarrow \mathsf{Trans}[(O_1, \ldots, O_c) \leftarrow \mathsf{PBSR}(S_1, \ldots, S_c, D)]$; Output $\mathcal{A}(\chi)$;

4 Oblivious Constrained Partitions

In this section, we give a construction for OCP. We refer the reader to Section 2, for an informal description of the algorithm. Our construction uses a family of pseudorandom functions, $\mathsf{F} = \{\mathsf{F}(\mathcal{K}, \cdot)\}_{\mathcal{K} \in \{0,1\}^\star}$. For a λ-bit key $\mathcal{K}$, $\mathsf{F}(\mathcal{K}, \cdot)$ is a function from $\{0,1\}^\lambda$ to $\{0,1\}^\lambda$. We assume F is a random oracle for security.

We start by describing two subroutines that will be useful in our construction. Subroutine $\mathsf{ExtractSubset}$ takes as input integers u and m and seed $\mathcal{K}$ and outputs a sequence T consisting of m distinct values in $[u]$. The subsets T is obtained by evaluating $\mathsf{F}(\mathcal{K}, \cdot)$ starting from 0 until m distinct elements are obtained. Subroutine $\mathsf{ExtractSubset}$ is also used in our construction of PSIR of Section 6. Subroutine $\mathsf{ExtractCondSubset}$ instead outputs a succinct description of a sequence T of m distinct elements of $[u]$ subject to the constrain that the r-th element is equal to a given element x. $\mathsf{ExtractSubset}$ and $\mathsf{ExtractCondSubset}$ are described in Figure 2. We also use procedure $\mathsf{FisherYates}(1^m, \mathcal{K}_k)$ that samples a random permutation of $[m]$ using the Fisher-Yates algorithm and using $\mathsf{F}(\mathcal{K}_k, \cdot)$ as the source of randomness. We omit a description of this standard algorithm.

We are now ready for a formal description of our construction OCP.

Construction 6 (OCP)**.** We describe an oblivious constrained partition scheme $\mathsf{OCP} = (\mathsf{OCP.GenerateSeed}, \mathsf{OCP.ExtractPartition})$. Our construction OCP uses $\mathsf{FisherYates}$ and $\mathsf{ExtractCondSubset}$ as subroutines.

```
$T \leftarrow \mathsf{ExtractSubset}(\mathcal{K}, u, 1^m)$

  1. Initialize $T$ to be an array of size $m$.
  2. Initialize seen to be an empty set.
  3. Initialize $\mathsf{next} \leftarrow 0$.
  4. While $\mathsf{seen.size} < m$:
       $\ell \leftarrow F(\mathcal{K}, \mathsf{next}) \bmod u$.
       If $\ell \notin \mathsf{seen}$:
         Insert $\ell$ into seen.
         Append $\ell$ to $T$.
       Increment next by one.
  5. return $T$.
```

```
$(T, \mathcal{K}, v) \leftarrow \mathsf{ExtractCondSubset}(1^\lambda, u, 1^m, r, x)$

  1. Generate random seed $\mathcal{K} \leftarrow \{0,1\}^\lambda$.
  2. Execute $T \leftarrow \mathsf{ExtractSubset}(\mathcal{K}, u, 1^m)$.
  3. Compute $v \leftarrow T[r] - x \bmod u$.
  4. Set $T[l] \leftarrow T[l] - v \bmod u$.
  4. Return $(T, (\mathcal{K}, v))$.
```

Figure 2: Description of ExtractSubset and ExtractCondSubset.

OCP.GenerateSeed. This algorithm outputs a compact description of a partition $\mathcal{P} = (P_1, \ldots, P_m)$ of $[n]$ into $m := n/k$ parts of size k such that one of the parts is the constraint subset S received in input. The algorithm sees the partition as a matrix with m rows, one for each of the m parts of the partition, and k columns . Columns $T_1, \ldots, T_k$ are constructed one at a time and the j-th partition P_j will consist of the j-th element of each column. At the start of the algorithm, all elements of $[n]$ are *unused* and elements become *used* as they are assigned to a part by the algorithm when columns are constructed. As we have explained in the informal description of the algorithm in Section 2, a column is not directly specified by its element but only by an index to a table of sorted unused elements.

$((\mathcal{K}_1, v_1) \ldots, (\mathcal{K}_k, v_k)) \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, S)$

1. Set $k = |S|$ and write S as $S = (s_1, \ldots, s_k)$ with $s_1 < s_2 < \cdots < s_k$.

   Pick random $r \in [m]$ and a random permutation τ of $[k]$.

   The algorithm will guarantee that the r-th element of each column will be an element of S. Specifically, $s_{\tau(\ell)}$ of S will appear as the r-th element of column T_ℓ, for $\ell = 1, \ldots, k$.

2. Allocate array `unused` of k Boolean values and initialize `unused`$[i] =$ `True`, for $i = 1, \ldots, k$.

   The algorithm maintains the invariant that `unused`$[i] =$ `True` *if and only if s_i, the i-th smallest element of S, is unused. Clearly, when the algorithm starts no element of S, actually no element of $[n]$, has been assigned to a part yet. Note that the algorithm only needs to keep track of which elements of S are unused.*

3. Allocate array rank of k integers and initialize $\mathsf{rank}[\ell] = s_\ell$, for $\ell = 1, \ldots, k$.

   The algorithm maintains the invariant that $\mathsf{rank}[\ell]$ is equal to the number of unused elements of $[n]$ that are smaller than or equal to s_ℓ. Since at the beginning all elements of $[n]$ are unused, $\mathsf{rank}[\ell]$ is initialized to be s_ℓ.

4. For each $\ell \leftarrow 1, \ldots, k-1$:

   The algorithm constructs the first $k-1$ columns. For efficiency reasons, the last column, T_k, is constructed using an ad-hoc procedure.

   (a) Set $(T_\ell, (\mathcal{K}, v)) \leftarrow \mathsf{ExtractCondSubset}(1^\lambda, u := n - m \cdot (\ell - 1), 1^m, r, x := \mathsf{rank}[\tau(\ell)])$.
   Column T_ℓ does not contain explicitly specifies the m elements but only their current ranks with respect to the $u := n - m \cdot (\ell - 1)$ still unused elements. Procedure ExtractCondSubset *also guarantees*

```
    that the $r$-th element of the column is equal to $s_{\tau(\ell)}$ by enforcing $T_\ell[r]$ to be equal to the current
    rank $x$ of $s_{\tau(\ell)}$.
    Next the algorithm updates the ranks of the unused elements of $S$ and checks if $T_\ell$ as returned by
    ExtractCondSubset contains unused elements of $S$ other than $s_\tau(\ell)$.
  (b) Initialize newRank to store the updated ranks as if $T_\ell$ were to be the $\ell$-th column.
  (c) For all elements $s_i \in S$ such that unused$[i]$ = True:
      i. Let $j$ be the largest integer such that rank$[i] \geq T_\ell[j]$.
      ii. If rank$[i] = T_\ell[j]$ and $i \neq \ell$ then go to Step 4a.
          In this case, $T_\ell[j]$ is equal to the current rank, rank$[i]$, of $s_i$ and which means that the column
          contains an unused element of $S$ other than $\ell$. The algorithm thus restart by selecting another
          candidate column for $T_\ell$.
      iii. Set newRank$[i] \leftarrow$ rank$[i] - (j-1)$.
          Note that $j-1$ elements of $T_\ell$ that are smaller than rank$[i]$ will be used and thus the rank of
          $s_i$ is adjusted accordingly.
  (d) Set $\mathcal{K}_\ell \leftarrow \mathcal{K}$ and $v_\ell \leftarrow v$, set unused$[\tau(\ell)]$ = False and update rank $\leftarrow$ newRank.
      $T_\ell$ can be used as the $\ell$-th column subset.

5. Randomly select $\mathcal{K}_k \leftarrow \{0,1\}^\lambda$, $T_k \leftarrow$ FisherYates$(1^m, \mathcal{K}_k)$ and set $v_k \leftarrow T_k[r] -$ rank$[\tau(k)]$.

6. Output $((\mathcal{K}_1, v_1) \ldots, (\mathcal{K}_k, v_k))$.
```

OCP.ExtractPartition. This algorithm takes the succinct representation, $((\mathcal{K}_1, v_1), \ldots, (\mathcal{K}_k, v_k))$ of a partition of $[n]$ into m parts of size k and outputs the explicit partition.

$(P_1, \ldots, P_m) \leftarrow$ OCP.ExtractPartition$(n, (\mathcal{K}_1, v_1), \ldots, (\mathcal{K}_k, v_k))$

```
1. Initialize $P_1, \ldots, P_m$ to be empty arrays.

2. Initialize $T \leftarrow [n]$ as an ordered set.

3. For $i \leftarrow 1, \ldots, k-1$:
  (a) Execute $T_i \leftarrow$ ExtractSubset$(|T|, m, \mathcal{K}_i)$.
  (b) For each $j \leftarrow 1, \ldots, m$:
          Append the $(T_i[j] - v_i)$-th smallest item in $T$ to $P_j$.
  (c) For each $j \leftarrow 1, \ldots, m$:
          Remove $(T_i[j] - v_i)$-th smallest item from $T$.

4. Execute $T_k \leftarrow$ FisherYates$(n/k, \mathcal{K}_k)$.

5. For each $j \leftarrow 1, \ldots, m$:
       Retrieve the $(T_k[j] - v_k)$-th smallest item in $T$ and append it to $P_j$.

6. Output $(P_1, \ldots, P_m)$.
```

We now prove the efficiency and security of our OCP construction. Our proofs will use the Chernoff Bounds that are reviewed in Appendix B.

Lemma 7. For $m = \omega(\log n)$, an invocation of ExtractSubset by OCP.GenerateSeed performs $O(m)$ PRF evaluations except with probability negligible in n.

Proof. At each iteration of the loop for ℓ that starts at Step 4, there are exactly $u = n - m \cdot (\ell - 1)$ unused elements and this value is passed to ExtractCondSubset at Step 4a and, eventually, to ExtractSubset to generate a subset of size m.

Note that $u \geq 2m$ in all invocations and therefore each F evaluations has probability at least $1/2$ of being distinct from all previous evaluations. We consider $p := 2(1 + 2\epsilon) \cdot m$ independent evaluations of F, $X_1, \ldots, X_p$ such that $X_i = 1$ if and only if the i-th evaluation is distinct from all previous evaluations. We set $X = X_1 + \ldots + X_p$ as the total number of successes and observe that $\mu = E[X] \geq p/2 = (1+2\epsilon) \cdot m$. Note ExtractSubset only requires m successes to terminate and thus the probability that ExtractSubset performs more than p evaluations is at most $\Pr[X \leq m] \leq \Pr[X \leq (1-\epsilon)\mu]$. Assuming that ϵ is small enough so that $(1 + 2\epsilon)(1 - \epsilon) \geq 1$ and by using Chernoff bounds, we obtain that this probability is $\mathsf{negl}(n)$. $\square$

Lemma 8. *A subset output by* ExtractCondSubset *intersects the constraint subset at exactly one element witg probability at least $1/e$.*

Proof. Fix any $\ell \in [k - 1]$ for which we are focusing our analysis. After fixing the r-th element of T_ℓ in Step 4a, we can view the remaining $m - 1$ elements as being randomly drawn from the set of the remaining $u - 1$ unused elements. The sampling is considered successful if we avoid the remaining $k - \ell$ elements of the constraint subset S. As a result, the probability that T_ℓ intersects only at the r-th element is

$$\begin{aligned}&\left(1 - \frac{k-\ell}{u-1}\right) \cdot \left(1 - \frac{k-\ell}{u-2}\right) \cdot \ldots \cdot \left(1 - \frac{k-\ell}{u-(m-1)}\right)\\ &\geq \left(1 - \frac{k-\ell}{u-m}\right)^m \geq \left(1 - \frac{k-\ell}{m(k-\ell)}\right)^m \approx \frac{1}{e}.\end{aligned}$$

$\square$

Theorem 9. OCP.GenerateSeed *requires $O(n)$ PRF evaluations except with negligible probability in n when $m, k = \omega(\log n)$ as well as storage of $O(m + k)$ integers.*

Proof. OCP.GenerateSeed requires storage of the ranks of the k elements of the constraint subset as well as the m elements of T_ℓ at each loop iteration.

By Lemma 8, a subset output by ExtractCondSubset has probability $1/e$ of passing the test of the loop starting at Step 4.c. We consider $p := e(1 + 3\epsilon) \cdot k$ independent invocations of ExtractCondSubset we would expect $\mu = (1 + 3\epsilon)k$ invocations to be successful. Therefore, by Chernoff bounds, the probability that OCP.GenerateSeed performs more than $(1 - \epsilon)\mu \geq (1 + \epsilon)k$ invocations is negligible, provided that ϵ is small enough so that $(1 + 3\epsilon)(1 - \epsilon) \geq (1 + \epsilon)$. The Theorem follows since, by Lemma 7, each invocation of ExtractCondSubset requires $O(m)$ PRF evaluations except with negligible probability, $\square$

Theorem 10. *Construction 6 is an oblivious constrained partition according to Definition 4 when function F is modeled as a random oracle.*

Proof. The theorem follows from the fact that the constraint set, as all the other subsets of the partition, is randomly selected and thus cannot be distinguished from the others. $\square$

In Appendix D, a more complex OCP scheme requiring $O(k)$ client storage for OCP.GenerateSeed is presented.

5 Private Batched Sum Retrieval

In this section, we present StreamPBSR, a simple batch sum retrieval scheme that downloads all the field elements. While requiring $O(n)$ bandwidth and computation, StreamPBSR can be used to construct a single-server multiple user PIR with sublinear encrypted operations later. The idea is very simple: each record is downloaded and added to the appropriate partial sums. Before streaming, the server could compress the database to reduce network costs.

Construction 11. $(O_1, \ldots, O_c) \leftarrow \mathsf{StreamPBSR}(S_1, \ldots, S_c)$

```
1. Initialize $O_1 \leftarrow 0, \ldots, O_c \leftarrow 0$.
2. For $i = 1, \ldots, n$:
    (a) For all $j \in [c]$ such that $i \in S_j$:
        i. $O_j \leftarrow O_j + \mathsf{B}_i$
3. Output $(O_1, \ldots, O_c)$.
```

Theorem 12. StreamPBSR is a private batched sum retrieval scheme according to Definition 5. In addition, the algorithm uses $O(n)$ bandwidth, $O(\sum_i |S_i|)$ additions, and $O(c)$ memory.

Proof. The algorithm accesses all the field elements in order independent of the input subsets. □

In Appendix D, we present several more communication efficient algorithms of more theoretical interest.

6 Private Stateful Information Retrieval

In this section, we present our construction of PSIR that uses as subprotocols an oblivious constrained partition scheme OCP, a private batched sum scheme PBSR, and a single-server PIR PIR. Our construction is parametrized by two integers (c, k). For a choice (c, k) of the parameters, PSIR has the following performance in terms of client memory required and bandwidth:

- Client memory: the client memory must be large enough to contain c records;
- Bandwidth per query amortized over c queries: k integers in the range $[n]$, k seeds of λ bits, and $n/c + \mathsf{PIR}(n/k)$ blocks, where $\mathsf{PIR}(N)$ is the bandwidth of the underlying single-server PIR PIR for a database of N records.

Except for the PIR operations, the server and client only need to perform plaintext operations or pseudo-random function evaluations.

A client initializes their state by executing PBSR to download c sums where each sum is over a random subset of $k - 1$ records, which will be the client's side information. To retrieve a record, the client iterates through its stored record sums or side information until finding a sum that does not include the desired query record as an addend. If all unused side information contains the desired query record, then our PSIR scheme will fail. However, we show that at least $(1 - \epsilon)c$ queries will succeed for all constant $0 < \epsilon < 1$ and $k \leq n/2$. After exactly $(1 - \epsilon)c$ queries, all clients will re-initialize the state (and side information) by running another PBSR protocol.

After finding side information that does not include the query record, the client executes OCP with the side information and the query record as the constraint subset. In addition, the client executes PIR to construct a PIR request corresponding to retrieving the constraint subset. The request from PIR and seed from OCP are sent to the server. The server reconstructs the partition and adds all records with each of the $m := n/k$ parts. The server executes PIR on the m record sums and returns the result to the client. The client simply decrypts the sum of side information and the query record and subtracts the side information to retrieve the query record. The formal description of PSIR is below.

Construction 13 (PSIR)**.** We describe the algorithms that constitute PSIR that uses an adjustable parameter k and a constant $0 < \epsilon < 1$.

$\mathsf{PSIR.Init}$ This protocol is jointly executed by the client and server to compute the client's initial state st. The database D is provided as input to the server. Algorithm $\mathsf{ExtractSubset}$ is described in Section 4.

$(\mathsf{st}, \perp) \leftarrow \mathsf{PSIR.Init}((1^\lambda, 1^c), (1^\lambda, D = (\mathsf{B}_1, \ldots, \mathsf{B}_n)))$

1. Client randomly selects $\mathcal{K}_1, \ldots, \mathcal{K}_c \leftarrow \{0,1\}^\lambda$ and sets
   - $S_1 \leftarrow \mathsf{ExtractSubset}(\mathcal{K}_1, n, 1^{k-1})$
   - ...
   - $S_c \leftarrow \mathsf{ExtractSubset}(\mathcal{K}_c, n, 1^{k-1})$.
2. Client and Server jointly set $(O_1, \ldots, O_c) \leftarrow \mathsf{PBSR}(S_1, \ldots, S_c)$.
3. Client sets $\mathsf{count} \leftarrow 1$ and $\mathsf{next} \leftarrow 1$.
4. Output $\mathsf{st} = (\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots, \mathcal{K}_c)$.

$\mathsf{PSIR.Query}$ This algorithm is executed by the client to retrieve the q-th record of the database. It takes as input the state st and the index q and returns an updated state st and a Query that is sent to the server.

$(\mathsf{st}, \mathsf{Query}) \leftarrow \mathsf{PSIR.Query}(q, \mathsf{st})$

1. Write st as $\mathsf{st} = (\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots, \mathcal{K}_c)$.
2. Let p be the smallest integer $p \geq \mathsf{next}$ such that $q \notin S_p$ and set $\mathsf{next} = p + 1$ and $\mathsf{count} = \mathsf{count} + 1$. If no such integer p exists, then abort.
3. Set $S \leftarrow S_p \cup \{q\}$.
4. Execute $(\mathcal{K}, r) \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, S)$.
5. Execute $(\mathsf{PIRQuery}, \mathsf{PIRKey}) \leftarrow \mathsf{PIR.Query}(r)$.
6. Update $\mathsf{st} = (\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots, \mathcal{K}_c,\ ((\mathcal{K}, r),\ \mathsf{PIRKey}))$.
7. Output $\mathsf{Query} = (\mathcal{K}, \mathsf{PIRQuery})$.

$\mathsf{PSIR.Reply}$ This algorithm is executed by the server to compute $\mathsf{Response}$ to the client's Query.

$\mathsf{Reply} \leftarrow \mathsf{PSIR.Reply}(\mathsf{Query}, D = (\mathsf{B}_1, \ldots, \mathsf{B}_n))$

1. Write Query as $\mathsf{Query} = (\mathcal{K}, \mathsf{PIRQuery})$ and set $m = n/k$.
2. Set $(P_1, \ldots, P_m) \leftarrow \mathsf{OCP.ExtractPartition}(n, \mathcal{K})$.
3. Compute $T_i \leftarrow \sum_{j \in P_i} \mathsf{B}_j$ for $i = 1, \ldots, m$.
4. Output $\mathsf{Response} = \mathsf{PIR.Reply}(\mathsf{PIRQuery}, T_1, \ldots, T_m)$.

$\mathsf{PSIR.Extract}$ This algorithm is executed by the client to recover the q-th block from $\mathsf{Response}$ obtained from server and the $\mathsf{QueryKey}$ associated with the Query sent to the server.

$B_q \leftarrow \mathsf{PSIR.Extract}(\mathsf{Response}, \mathsf{st})$

1. Parse st as

   $(\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots,\ \mathcal{K}_c,\ ((\mathcal{K}, r),\ \mathsf{PIRKey}))$.
2. Execute $\mathsf{B}_q \leftarrow \mathsf{PIR.Extract}(\mathsf{Response}, \mathsf{PIRKey}) - O_r$.
3. Output B_q.

$\mathsf{PSIR.UpdateState}$ This protocol is executed by the client, possibly interacting with the server, to update the client's state. This can be executed offline by the client by itself unless it requires the server's help to re-instate an initial state.

$(\mathsf{st}, \perp) \leftarrow \mathsf{PSIR.UpdateState}((\mathsf{st}), (D))$

1. Parse st as

   $(\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots, \mathcal{K}_c, ((\mathcal{K}, r), \mathsf{PIRKey}))$.

2. Set st as $\mathsf{st} = (\mathsf{count}, \mathsf{next}, O_1, \ldots, O_c, \mathcal{K}_1, \ldots, \mathcal{K}_c)$.

3. If $\mathsf{count} > (1 - \epsilon)c/2$ then

   Execute protocol $(\mathsf{st}, \perp) \leftarrow \mathsf{PSIR.Init}((1^\lambda, 1^c), (1^\lambda, D))$.

4. Output st.

We next prove that if a client has local storage large enough to hold c blocks then, except with negligible probability, the initial state can be used for $\Omega(c)$ queries. We use the following lemma on the tail of the sum of geometric distributions (for a proof, see Theorem 2.1 of [32]).

Lemma 14. Let $X_1, \ldots, X_n$ be n geometrically distributed random variables with success probability $p_1, \ldots, p_n$ and define X to be $X := \sum_{i=1}^n X_i$. Then, for every $\epsilon > 0$,

$$\Pr[X \geq (1 + \epsilon)\mu] \leq e^{-p_\star \mu(\epsilon - \ln(1+\epsilon))},$$

where $\mu = \mathbb{E}[X] = \sum_i 1/p_i$ and $p_\star = \min_i p_i$.

Lemma 15. Algorithm $\mathsf{PSIR.Query}$ fails with probability negligible in n whenever $c = \omega(\log n)$ and $k \leq n/2$.

Proof. $\mathsf{PSIR.Query}$ fails only if all c subsets are consumed by fewer than $(1-\epsilon)c/2$ queries. By $\mathsf{PSIR.Init}$, each S_p is a randomly chosen subset of $[n]$ of cardinality $k-1$ and thus, for any fixed $q \in [n]$, the probability that $q \notin S_p$ is $(n - (k-1))/n$, which is at least $1/2$ since $k \leq n/2$. Therefore the number of subsets consumed by each invocation of $\mathsf{PSIR.Query}$ is geometrically distributed with probability of success at least $1/2$. The lemma follows by applying the bound given by Lemma 14. □

Theorem 16. Construction 13 reduces private record retrieval to a single PIR execution on a database of n/k records while all other operations are strictly plaintext or cryptographic hashes as well as communicating k seeds and k integers. In addition, the client must download (offline) $O(n/c)$ records amortized over $\Omega(c)$ queries.

Proof. The cost of communicating k seeds and k integers derive from the seeds derived from $\mathsf{OCP.GenerateSeed}$. Besides the operations of PIR, all other client or server operations are cryptographic hash evaluations and adding or moving blocks around (plaintext). For amortized communication costs, the client streams n records to construct side information which is used over $O(c)$ queries. Therefore, a total of $O(n/c)$ amortized offline communication is required. □

Theorem 17. Construction 13 is a private stateful information retrieval scheme according to Definition 1.

Proof. We note that $\mathsf{PSIR.Init}$ and $\mathsf{PSIR.UpdateState}$ are independent of the input subsets by the security of the underlying PBSR scheme. As a result, the adversary's view is independent of each honest client's state. $\mathsf{PSIR.Query}$ outputs Query which is viewed by the adversary. $\mathsf{PSIR.Query}$ uses only the input q and the client's state. Query consists of $\mathsf{PIRQuery}$ and the seed generated by OCP. By the security of the underlying PIR, $\mathsf{PIRQuery}$ is generated independently from the input q. Similarly, $\mathcal{K}$ generated by $\mathsf{OCP.GenerateSeed}$ is independent from the input q. As a result, Query is independent from q. As a result, the execution of $\mathsf{PSIR.Reply}$ as well its transcript is independent from q as it only takes Query and the database as input. Therefore, the adversary's view is independent of the query sequence for all honest clients. □

In Appendix C, we present several techniques to amortize the costs of $\mathsf{PSIR.UpdateState}$.

6.1 Discussion

In this section, we will discuss various parts of our PSIR construction.

Online to Offline. A major contribution of our PSIR scheme is moving online costs to an offline phase. Not all computation and network costs are identical. A charging phone downloading at night connected to WiFi is almost free in comparison to a phone downloading during the day over cellular networks. PIR expenses all bandwidth at query time whereas our PSIR scheme moves costs to cheaper offline costs overcoming a significant practical obstacle.

Use of StreamPBSR. At first look, StreamPBSR might seem really inefficient to download the entire database to construct the client's state. However, it turns out that StreamPBSR is very efficient in terms of private retrieval storage schemes that must handle many queries. For existing single-server PIR constructions, a couple hundreds of PIR queries requires communication more than streaming an entire database.

In practice, StreamPBSR is competitive when compared to asymptotically more efficient methods that use homomorphic encryption and batch codes in an attempt to decrease communication costs (see Appendix D). These techniques require significant computation so that the amortized computations costs become similar or worse than existing single-server PIR constructions even though they have better asymptotic performance. On the other hand, StreamPBSR uses linear communication costs with very smaller hidden constants giving much better practical efficiency.

Initializing State. Another issue that arises is that initializing a client's state before performing query requires streaming the entire database. Many clients may wish to perform queries immediately after joining the protocol. In this case, there are several options that the client may use. First, the client may just perform standard PIR queries while waiting for client state to finish initializing. On the other hand, the client may use another private batched sum retrieval schemes described in Appendix E.1 to initialize state for a small number of queries. The client may use this smaller state for immediate queries while waiting for the large client state to be initialized. The first small number of queries for each client will be slow as the client waits for its state initialization to complete (analogous to a cache warming up).

7 Experimental Evaluation

In this section we report on the experimental evaluation of PSIR that we have conducted. We start by describing our experimental setup and our choice of parameters for the experiments. We then consider two types of implementations of PSIR: the first is on top of the C++ APIs provided by the open sourced XPIR [2] implementation and relies on Ring-LWE and the second considers PIR based on Paillier's encryption scheme [42].

Using the experiments, we attempt to answer three important questions. First, can we construct an PSIR scheme with less concrete costs than the current best PIR construction, SealPIR [7]? Second, what are the benefits of an PSIR with XPIR and SealPIR compared to generic XPIR and SealPIR? Finally, what is the latency of our PSIR schemes in several different settings?

7.1 The Experimental Setup

Our experiments are conducted using two identical machines, one for the client and one for the server. The machines are Ubuntu PCs with 12 cores, 3.5 GHz Intel Xeon E5-1650 and 32 GB of RAM. All reported results (except the estimate costs for SealPIR) have standard deviations less than 10% of the means. The cost of network resources are determined at the application layer. The client and server implementations of PSIR are built using the gRPC [4] library.

	SealPIR			SealPSIR			PaillierPSIR		
Database Size (n)	2^{16}	2^{18}	2^{20}	2^{16}	2^{18}	2^{20}	2^{16}	2^{18}	2^{20}
Client State (KB)	N/A	N/A	N/A	**129**	**258**	**524**	**129**	**258**	**524**
Client CPU (sec)									
Query	< 0.01	< 0.01	< 0.01	< 0.01*	< 0.01*	< 0.01*	0.01	0.02	0.04
Extract	< 0.01	< 0.01	< 0.01	< 0.01*	< 0.01*	< 0.01*	0.11	0.11	0.11
OCP.GenerateSeed	N/A	N/A	N/A	0.04	0.12	0.51	0.04	0.12	0.51
total	**< 0.01**	**< 0.01**	**< 0.01**	**< 0.05***	**< 0.13***	**< 0.52***	**0.16**	**0.25**	**0.66**
Server CPU (sec)									
Reply	0.41	1.20	6.50	0.05*	0.10*	0.20*	0.74	0.91	1.24
OCP.ExtractPartition	N/A	N/A	N/A	0.02	0.11	0.51	0.02	0.11	0.51
total	**0.41**	**1.20**	**6.50**	**0.07***	**0.21***	**0.71***	**0.76**	**1.02**	**1.75**
Network (KB)									
Query	64	64	64	64	64	64	2	4	8
Answer	256	256	256	256	256	256	18	18	18
OCP seed	N/A	N/A	N/A	10	20	40	10	20	40
online total	**320**	**320**	**320**	**330**	**340**	**360**	**30**	**42**	**66**
StreamPBSR	N/A	N/A	N/A	47	92	178	47	92	178
amortized total	**320**	**320**	**320**	**377**	**432**	**538**	**77**	**134**	**244**

Table 2: Microbenchmarks for network and CPU costs. The left column results are reported from [7]. The middle column is extrapolated estimates from [7] with asterisks indicating estimates. The right column are our experimental results.

Database. Our experimental database contains $n = 2^{16}, 2^{18}$, 2^{20} records of 288 bytes to match the experiments of previous works [7, 8]. Each database record will be chosen uniformly at random. This is a pessimistic choice for our StreamPBSR scheme since the database cannot be compressed. For real-world databases, compression could save up to 75% of network costs especially for large databases. As we shall see, even with this very pessimistic choice, our PSIR scheme is more efficient than previous PIR schemes.

Primitives. Our PSIR construction relies on three primitives: PBSR, OCP and PIR. In all our experiments we have used StreamPBSR as our PBSR scheme. As far as Oblivious Constrained Partition is concerned, we implemented our construction from Section 4 using the OpenSSL implementation of SHA256 as the basis of our function F. All seeds are 16 bytes long. OCP.ExtractPartition requires an ordered set implementation that requires efficient querying for the i-th smallest element. We implement a red-black tree that maintains the number of elements in each node's left and right subtree to efficiently query for the i-th smallest element in logarithmic time.

Parameters. PSIR inherits parameter k from OCP and k determines the number of seeds that must be transferred during the online phase of PSIR and the size of the PIR database that must be constructed by the server to answer the client's query. In our experiments, we fix $k = 2\sqrt{n}$ where n is the number of records. We choose the client to store approximately 125, 250 and 500 KB of state for database with 2^{16}, 2^{18} and 2^{20} records respectively. In our experiments, the client will store slightly less than the above chosen values. With records of 288 bytes, we choose the client to hold $c \in \{425, 850, 1725\}$ records of side information for each respective record size.

We note our client state size assumptions are reasonable since they are significantly smaller than the network costs of XPIR and similar to the network costs of SealPIR. The cost of adding plaintext records within parts for our PSIR schemes is reported with the underlying PIR protocol.

In order to hide the actual number of queries served with the given memory, in PSIR we stop after serving $(1 - \epsilon)c/2$ queries, independently from the number of the still unused side information available in client memory. We prove that, except with negligible probability, c records of side information suffice to serve $(1-\epsilon)c/2$ queries, for any $\epsilon > 0$ and sufficiently large c. We have performed experimental evaluation to determine the correct number of queries our PSIR schemes will service before running PSIR.Init to re-initialize client state. We consider database sizes of $n \in \{2^{16}, 2^{18}, 2^{20}\}$ with the parameters $k = 2\sqrt{n}$. In our experiments, we pick c random subsets of $k-1$ records as the client's side information. We consider random query sequences and determine the maximum number of queries that may be serviced by the c blocks of side information. We run the above experiment 1,000,000 times and report on the average and minimum queries serviced.

Database Size (n)	2^{16}	2^{18}	2^{20}
Parameter (c)	425	850	1725
Average Number of Queries Serviced	421.97	846.57	1722.52
Minimum Number of Queries Serviced	420	844	1719

We see that, on average, less than 5 blocks of side information are wasted. Furthermore, the minimum encountered over all 1,000,000 experiments is greater than $c-6$ in all three scenarios. Therefore, we empirically choose that $c-25$ queries will be serviced by our PSIR scheme before running PSIR.Init to re-initialize client state.

Latency. To understand the total time required to perform a private retrieval, we measure the latency of retrieving one 288 byte element from databases of $n \in \{2^{16}, 2^{18}, 2^{20}\}$ records. We set up our client and server machines on the same LAN network and utilize the Linux Traffic Control [1] to configure the bandwidth on both machines. We consider the following three scenarios also used in [7]: querying between two datacenters, using a home network and using a mobile network. We configure our machines to maintain 800 Mbps for queries between data centers. For the home and mobile network, we choose 18.7 Mbps and 10 Mbps for each setting based on Akamai's latest reports [3]. The results are reported in Figure 3. Amortized latency refers to online latency and the latency of PSIR.Init and PSIR.UpdateState spread over queries. PaillierPSIR and XPSIR have smaller latency than XPIR.

7.2 Cost of Ring-LWE based PSIR

Our first implementation of PSIR is in C++ on top of the APIs provided by the open sourced XPIR [2] implementation.

We choose the parameters for Ring-LWE based FHE schemes used by XPIR based on the analysis by Albrecht *et al.* [6]. Similar parameters are used in previous instantiations of XPIR [5] and SealPIR [7]. For XPIR, we use 2048 degree polynomials and 60 bit coefficient modulus. In particular, we use the recommended modulus of the form $2^{61} - i \cdot 2^{14} + 1$ for different i. We will pack $\alpha = 14$ records to be downloaded and fit α records into a single ciphertext. Therefore, XPIR will view the database as n/α records of $288 \cdot \alpha$ bytes.

We construct PSIR using the XPIR scheme with $d = 2$ levels of recursion, which we denote XPSIR. In addition, we estimate costs of PSIR with SealPIR (SealPSIR) using previously reported results [7]. We note our experimental setup is worse than the one chosen in [7] in terms of number of cores, RAM size and CPU version. The means of 10 experiment runs are reported for XPSIR in Table 3. Our estimated costs for SealPSIR are shown in Table 2. The costs of OCP are shown in Appendix A.

XPSIR. The results for XPSIR show up to 12x reductions in server CPU, 7x reductions in online network costs and 6x reductions in total amortized network costs compared to XPIR. For the 1M items case, XPSIR introduces up a half second increase in client CPU, but offsets the gain by decreasing server by at least 4.5 seconds.

SealPSIR. Our estimates for SealPSIR indicate up to 10x reduction in server CPU costs since the PSIR framework reduces the client database to $\sqrt{n}/2$ records. In exchange, SealPSIR introduces a 13% increase in online network costs, a 68% increase in total amortized network costs and a significant increase in client CPU.

	XPIR ($d = 2$)			XPIR ($d = 3$)			XPSIR		
Database Size (n)	65,536	262,144	1,048,576	65,536	262,144	1,048,576	65,536	262,144	1,048,576
Client State (KB)	N/A	N/A	N/A	N/A	N/A	N/A	**129**	**258**	**524**
Client CPU (sec)									
XPIR query	0.04	0.08	0.15	0.01	0.02	0.04	< 0.01	< 0.01	0.01
XPIR extract	< 0.01	< 0.01	< 0.01	0.02	0.02	0.02	< 0.01	< 0.01	< 0.01
OCP.GenerateSeed	N/A	N/A	N/A	N/A	N/A	N/A	0.04	0.12	0.51
total	$<$ **0.05**	$<$ **0.09**	$<$ **0.16**	**0.03**	**0.04**	**0.06**	$<$ **0.05**	$<$ **0.13**	$<$ **0.53**
Server CPU (sec)									
XPIR	0.35	1.37	5.08	0.55	2.02	6.47	0.01	0.01	0.01
OCP.ExtractPartition	N/A	N/A	N/A	N/A	N/A	N/A	0.01	0.11	0.51
total	**0.35**	**1.37**	**5.08**	**0.55**	**2.02**	**6.47**	**0.02**	**0.12**	**0.52**
Network (KB)									
XPIR query	4,456	8,913	17,891	1,638	2,556	4,129	295	393	557
XPIR answer	262	590	590	1,998	2,097	2,097	262	262	262
OCP seed	N/A	N/A	N/A	N/A	N/A	N/A	10	20	40
online total	**4,718**	**9,503**	**18,481**	**3,636**	**4,653**	**6,226**	**567**	**675**	**895**
StreamPBSR	N/A	N/A	N/A	N/A	N/A	N/A	47	92	178
amortized total	**4,718**	**9,503**	**18,481**	**3,636**	**4,653**	**6,226**	**614**	**767**	**1,073**

Table 3: Microbenchmarks for network and CPU costs for XPIR with $d \in \{2, 3\}$ levels of recursion and PSIR using XPIR ($d = 2$).

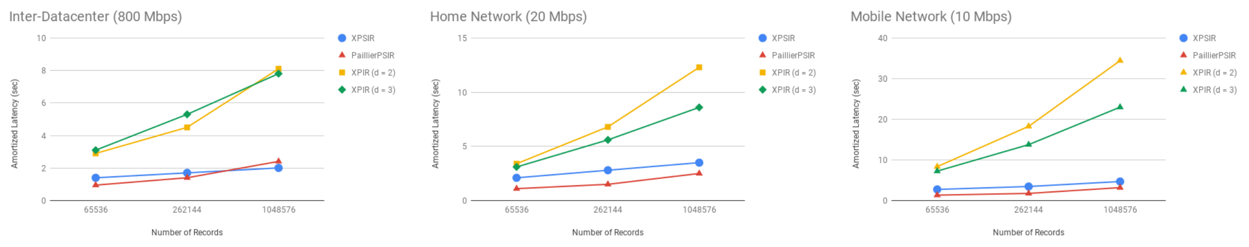
However, the client CPU increase is half a second while 6 seconds are saved in server CPU. Furthermore, 80% of the extra network costs can be performed offline at cheaper, non-busy times.

7.3 Cost of PaillierPSIR

In the previous section, we show that our estimates for SealPSIR indicate decreases in server CPU and online network costs but increases in total amortized network costs. Is it possible to construct a PSIR scheme that can decrease the total amortized network while reducing server CPU and online network costs significantly? We answer in the affirmative.

We construct a library that constructs PIR based on the Paillier cryptosystem [42] and build a PSIR scheme on top of the Paillier PIR library. We denote this PSIR scheme as PaillierPSIR. Traditionally, PIR schemes built from Paillier enjoy the advantages of small network costs but suffer from extremely large server CPU costs. However, PaillierPSIR reduces the online PIR request to a small database where the Paillier-built PIR is feasible. We study the time needed by the client to generate queries and extract responses as well as the time needed by the server to generates responses from queries. In addition, we examine network costs. The results can be seen in the right column of Table 2 and compared to SealPIR (the best, previous construction) seen in the left column of Table 2.

Optimizations to Paillier. The Paillier cryptosystem [42] is a partially homomorphic encryption system. Paillier has two important properties which enable its use for PIR: homomorphic plaintext absorption and homomorphic addition. Homomorphic plaintext absorption is the property that given an encryption, $\mathsf{Enc}(\mathcal{K}, m)$, and a plaintext, p, then $\mathsf{Enc}(\mathcal{K}, m)^p = \mathsf{Enc}(\mathcal{K}, mp)$. Homomorphic addition is the property that for any two ciphertexts, $\mathsf{Enc}(\mathcal{K}, m_1)$ and $\mathsf{Enc}(\mathcal{K}, m_2)$, then $\mathsf{Enc}(\mathcal{K}, m_1)\mathsf{Enc}(\mathcal{K}, m_2) = \mathsf{Enc}(\mathcal{K}, m_1 + m_2)$. Paillier can be used to perform PIR requests. For a database, $p_1, \ldots, p_n$, of n items of b bits, the client uploads $\mathsf{Enc}(\mathcal{K}, m_1)$, ..., $\mathsf{Enc}(\mathcal{K}, m_n)$ where only one $m_i = 1$ corresponding to retrieving the i-th item. The server needs to compute the value $\prod_{j \in [n]} \mathsf{Enc}(\mathcal{K}, m_j)^{p_j}$. The trivial way to compute a product of powers is to perform n exponentiations and n multiplications. Bernstein [10] surveys several techniques that improve the



(a) Inter-Datacenter. (b) Home Network. (c) Mobile Network.

Figure 3: The amortized latency for queries under different settings.

computational efficiency of this problem. We implement Straus's algorithm [48] as a faster way to compute the product of powers. For parameter ρ, we only require $(n/\rho)\cdot(2^\rho+b)$ multiplications and $(n/\rho)\cdot b$ squarings. For a description of Straus's algorithm, see Algorithm 14.88 in [39].

Network costs. PaillierPSIR benefits from significant reductions in online network costs as well as a modest reduction in total amortized network costs. PaillierPSIR uses 4.8-10.5x less online network costs and 1.3-4.5x less total amortized network costs compared to SealPIR. The main gains derive from replacing RLWE-based PIR schemes with the Paillier PIR scheme. Furthermore, the majority of amortized network costs may be performed offline.

CPU costs. For 1M items, PaillierPSIR reduces server CPU costs by more than 3.7x compared to SealPIR. However, PaillierPSIR introduces an increase in client CPU costs which we believe is a worthwhile tradeoff. In concrete terms, client CPU increases by 660 milliseconds while server CPU decreases by more than 4 seconds.

8 Conclusions

In this paper, we present PSIR, an extension of PIR, that is able to utilize the large amounts of storage available to applications on client devices. Unlike other stateful primitives, we design PSIR such that several important practical properties of PIR are maintained. In particular, PSIR ensures that simultaneous querying capability to large groups of independent clients, query privacy for a server colluding with clients and the ability for stateless clients to enroll in the system using only interaction with the server. By using client state, PSIR reduces the number of public-key operations that dominate practical costs to be sub-linear in the database size.

For concrete gains, we show that PaillierPSIR scheme (PSIR using PaillierPIR) is able to significantly reduce server CPU, online and amortized total network costs compared to SealPIR (the current best PIR construction). In addition, we show that instantiating PSIR with XPIR and SealPIR can also significantly reduce server CPU. For PSIR with XPIR, significant network cost are also enjoyed while PSIR with SealPIR increases network costs. Due to our PSIR scheme, the majority of network costs may be moved to offline processing.

References

[1] Introduction to Linux traffic control. `http://tldp.org/HOWTO/Traffic-Control-HOWTO/intro.html`, 2006.

[2] XPIR: Private information retrieval for everyone. `https://github.com/XPIR-team/XPIR`, 2015.

[3] Q1 2017 State of the internet - connectivity report. `https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf`, 2017.

[4] gRPC - an RPC library and framework. `https://github.com/grpc/grpc`, 2018.

[5] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, 2016(2):155–174, 2016.

[6] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.

[7] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. Cryptology ePrint Archive, Report 2017/1142, 2017. https://eprint.iacr.org/2017/1142.

[8] S. Angel and S. T. Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, pages 551–569, 2016.

[9] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Annual International Cryptology Conference*, pages 55–73. Springer, 2000.

[10] D. J. Bernstein. Pippenger's exponentiation algorithm. 2002.

[11] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. *Proceedings on Privacy Enhancing Technologies*, 2015(2):4–24, 2015.

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

[13] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[14] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011.

[15] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *Theory of Cryptography Conference*, pages 694–726. Springer, 2017.

[16] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[17] R. Cheng, W. Scott, B. Parno, A. Krishnamurthy, and T. Anderson. Talek: a private publish-subscribe protocol. Technical report, Technical Report. University of Washington, 2016.

[18] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, pages 41–50. IEEE, 1995.

[19] I. Damgård and M. Jurik. A generalisation, a simplication and some applications of Paillier's probabilistic public-key system. In *International Workshop on Public Key Cryptography*, pages 119–136. Springer, 2001.

[20] P. M. Fenwick. A new data structure for cumulative frequency tables. *Software: Practice and Experience*, 24(3):327–336, 1994.

[21] R. A. Fisher and F. Yates. *Statistical tables for biological, agricultural and medical research (3rd ed.)*. Oliver & Boyd, 1948.

[22] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *International Colloquium on Automata, Languages, and Programming*, pages 803–815. Springer, 2005.

[23] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 182–194. ACM, 1987.

[24] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[25] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1591–1601. ACM, 2016.

[26] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *International Workshop on Public Key Cryptography*, pages 107–123. Springer, 2010.

[27] T. Gupta, N. Crooks, W. Mulhern, S. T. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *NSDI*, pages 91–107, 2016.

[28] A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. Private anonymous data access. Cryptology ePrint Archive, Report 2018/363, 2018. `https://eprint.iacr.org/2018/363`.

[29] R. Henry. Polynomial batch codes for efficient IT-PIR. *Proceedings on Privacy Enhancing Technologies*, 2016(4):202–218, 2016.

[30] 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*, pages 262–271. ACM, 2004.

[31] M. Jakobsson and A. Juels. Addition of El Gamal plaintexts. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 346–358. Springer, 2000.

[32] S. Janson. Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135:1 – 6, 2018.

[33] S. Kadhe, B. Garcia, A. Heidarzadeh, S. E. Rouayheb, and A. Sprintson. Private information retrieval with side information. *arXiv preprint arXiv:1709.00112*, 2017.

[34] N. P. Karvelas, A. Peter, and S. Katzenbeisser. Blurry-ORAM: A multi-client oblivious storage architecture. *IACR Cryptology ePrint Archive*, 2016:1077, 2016.

[35] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Foundations of Computer Science, 1997. Proceedings., 38th Annual Symposium on*, pages 364–373. IEEE, 1997.

[36] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.

[37] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 341–358. IEEE, 2015.

[38] T. Mayberry, E.-O. Blass, and G. Noubir. Multi-client oblivious RAM secure against malicious servers. Cryptology ePrint Archive, Report 2015/121, 2015. `https://eprint.iacr.org/2015/121`.

[39] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC press, 1996.

[40] B. Morris, P. Rogaway, and T. Stegers. How to encipher messages on a small domain. In *Advances in Cryptology - CRYPTO 2009, 29th Annual International Cryptology Conference*, volume 5677 of *Lecture Notes in Computer Science*, pages 286–302. Springer, 2009.

[41] D. E. Muller. Application of Boolean algebra to switching circuit design and to error detection. *Transactions of the IRE Professional Group on Electronic Computers*, (3):6–12, 1954.

[42] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 223–238. Springer, 1999.

[43] S. Patel, G. Persiano, M. Raykova, and K. Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. Cryptology ePrint Archive, Report 2018/373, 2018. `https://eprint.iacr.org/2018/373`.

[44] M. Pătraşcu and E. D. Demaine. Tight bounds for the partial-sums problem. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 20–29. Society for Industrial and Applied Mathematics, 2004.

[45] I. Reed. A class of multiple-error-correcting codes and the decoding scheme. *Transactions of the IRE Professional Group on Information Theory*, 4(4):38–49, 1954.

[46] T. Ristenpart and S. Yilek. The mix-and-cut shuffle: Small-domain encryption secure against n queries. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 392–409, 2013.

[47] D. Stinson, R. Wei, and M. B. Paterson. Combinatorial batch codes. *Advances in Mathematics of Communications*, 3(1):13–27, 2009.

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

[49] J. Zhang, W. Zhang, and D. Qiao. MU-ORAM: Dealing with stealthy privacy attacks in multi-user data outsourcing services. *IACR Cryptology ePrint Archive*, 2016:73, 2016.

A Cost of OCP

To evaluate the cost of our OCP scheme described in Section 4, we perform benchmarks to measure the CPU costs of executing GenerateSeed and ExtractPartition as well as the size of the seed transmitted between the client and server by our PSIR scheme.

For our experiment, we consider randomly chosen constraint subsets of size $k = 2\sqrt{n}$ for various values of database sizes $n \in \{2^{16}, 2^{18}, 2^{20}\}$. We report results on the CPU costs for executing OCP.GenerateSeed to construct the succinct representation and for executing OCP.ExtractPartition to construct the explicit partition. The experiments are run 100 times with the means reported below.

Database Size (n)	65,536	262,144	1,048,576
Client CPU (ms)			
GenerateSeed	37.39	124.18	511.80
Server CPU (ms)			
ExtractPartition	23.31	109.23	512.27
Network (KB)			
Seed Size	10	20	40

We note that the network costs of OCP is significantly better than the trivial algorithm of explicitly describing the entire partition. For example, the communication required to explicitly send the partition for $n = 2^{20}$ database records is approximately 2.5 MB, which is significantly larger than the 40 KB used by our OCP scheme.

B Chernoff Bounds

Chernoff Bounds are a concentration equality for a sequence of many independent events with binary random variable outcomes.

Theorem 18 (Chernoff Bounds)**.** Let $X = X_1 + \ldots + X_n$ where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$ and all X_i are independent. Let $\mu = \mathsf{E}[X] = p_1 + \ldots + p_n$. Then,

1. $\Pr[X \geq (1+\epsilon)\mu] \leq \exp(-\frac{\epsilon^2\mu}{2+\epsilon})$.
2. $\Pr[X \leq (1-\epsilon)\mu] \leq \exp(-\frac{\epsilon^2\mu}{2})$.

We get the following corollary for restricted parameters.

Theorem 19. Let $X = X_1 + \ldots + X_n$ where $X_i = 1$ with probability p_i and $X_i = 0$ with probability $1 - p_i$ and all X_i are independent. Let $\mu = \mathsf{E}[X] = p_1 + \ldots + p_n$. For constant $0 < \epsilon < 1$ and if $\mu = \mathsf{E}[X] = p_1 + \ldots + p_n = \Omega(\log^2 n)$, then

1. $\Pr[X \geq (1+\epsilon)\mu] = \mathsf{negl}(n)$.
2. $\Pr[X \leq (1-\epsilon)\mu] = \mathsf{negl}(n)$.

C PSIR Amortization

In this section, we present several useful techniques for distributing the cost of PSIR.UpdateState for some important settings.

C.1 Online Work to Offline Processing

One interpretation of our PSIR scheme can be the ability to offload the online cost of queries to offline preprocessing. For example, let's suppose that a client will only query 5 times per day. During the night before (or any other off-peak periods), the client will gather a sufficient amount of side information using a PBSR scheme, which will be used when performing queries online the next day. By viewing the PBSR scheme as offline preprocessing, we are able to reduce the cost of online queries. As a result, an PSIR scheme can be viewed as a technique to move online work to offline processing.

C.2 Worse Case to Average Case Cost

A natural theoretical question regularly asked is whether the worst case of a query may be reduced to be equivalent to the average (amortized) cost over a large number of queries. By evenly distributing costs over queries, there will be no unexpected jumps in resource requirements or unexpected latency increases. Furthermore, clients and the server do not need to plan for resource allocations that are ever too large. Reducing worst case cost to average case costs in our PSIR scheme reduces to being able to distribute the costs of the underlying PBSR scheme over many queries. The costs of StreamPBSR, the scheme used in our instantiations in Section 7, can be trivially distributed over any set of $(1-\epsilon)c$ queries. The client will simply keep a pointer to the last database record downloaded. For each query, the client will perform exactly $\frac{1}{(1-\epsilon)c}$ work of StreamPBSR. This involves downloading $\frac{n}{(1-\epsilon)c}$ records and adding them to the appropriate side information. Note, this requires the client to store two sets of side information. One is being used to service queries and the other set is being built by StreamPBSR to be used once the current set of side information is exhausted.

D Oblivious Constrained Partition

We present an OCP scheme that only requires the client to store $O(k)$ integers instead of $O(n/k+k)$ integers required by the OCP scheme of Section 4.

D.1 Online Partial Sums Data Structure

In this section, we present previous results on online partial sums data structure.

Definition 20 (Online Partial Sums Data Structure)**.** An online partial sums data structure PartialSums consisting of the following algorithms:

- $D \leftarrow \mathsf{PartialSums.Init}(v_1, \ldots, v_n)$: an algorithm that takes as input an array of n values, $v_1, \ldots, v_n$, and outputs a database D for the values $v_1, \ldots, v_n$.
- $D' \leftarrow \mathsf{PartialSums.Update}(D, i, \delta)$: an algorithm that takes as input an index $i \in [n]$ and updates $v_i \leftarrow v_i + \delta$ in D and outputs a new version of the database, D'.
- $s \leftarrow \mathsf{PartialSums.Sum}(D, i)$: an algorithm that takes as input an index $i \in [n]$ and outputs the sum $s \leftarrow v_1 + \ldots + v_i$.
- $i \leftarrow \mathsf{PartialSums.Select}(D, v)$: an algorithm that takes as input a value v and outputs i such that $\mathsf{PartialSums.Sum}(i-1) < v \leq \mathsf{PartialSums.Sum}(i)$.

Fenwick [20] presented an online partial sums data structure without the Select operation, which required $O(\log n)$ operations using $O(n)$ storage. Using binary search, Select could be done using $O(\log^2 n)$ operations. Pătraşcu and Demaine [44] present tight bounds for an online partial sums data structure that supports all operations in $O(\log n)$ operations using $O(n)$ storage.

D.2 Space-Efficient OCP Scheme

In our new construction, we select a random subset by using succinctly describable pseudo-random permutation [36] instead of a hash function. Specifically, for pseudo-random family of permutations $\gamma(\cdot, \cdot)$ over $[n]$, we pick a random seed $\mathcal{K}$ and output the set $T = \{\ \gamma(\mathcal{K}, 1),\ \ldots,\ \gamma(\mathcal{K}, k)\}$. Note that now we have direct access to the set T; that is, to determine the r-th element of T, only the value $\gamma(\mathcal{K}, r)$ needs to be computed. The previous subset generation algorithm from Section 4 required computing all of $T_1, \ldots, T_{r-1}$ before computing T_r. As a result, we do not need to keep all elements of T stored to determine the r-th element. This will completely removes the $O(n/k)$ storage requirement.

However, we now require a more complex data structure to maintain the ranks of the unused items in the constraint subset since the generated subsets will not be explicitly stored. In particular, we will use online partial sums data structures, described in Appendix D.1, to store ranks. The items of the constraint subset are stored in sorted order such that the partial sum up to an index i will be equal to the rank of i-th largest member of the constraint subset. When initializing the data structure, we will store the differences between adjacent elements of the sorted constraint subset to ensure that rank can be retrieved by performing a partial sum query. Removing an element from the set of unused elements requires decreasing the rank of all constraint subset elements that are larger than the removed element by one. This can be achieved by simply subtracting one from the index of the smallest item in the constraint subset that is larger than the element to be removed. As a result, the rank of all constraint subset elements larger will also decrease by one. Finding the smallest element from the constraint subset larger than the removed element requires a single PartialSums.Select operation while retrieving the rank and updating an entry requires a single PartialSums.Sum and PartialSums.Update operation respectively. The entire data structure only requires storing a single entry for each constraint subset item meaning storage requirements are $O(k)$.

Construction 21. We describe a pseudorandom partition with a fixed partition OCP = (OCP.GenerateSeed, OCP. ExtractPartition).

$\mathsf{OCP.GenerateSeed}$. This algorithm constructs a key $\mathcal{K}$ which describes a partitioning of $[n]$ into $\frac{n}{k}$ parts such S is one of the parts where n, k and S are all inputs.

```
$\mathcal{K} \leftarrow \mathsf{OCP.GenerateSeed}(1^\lambda, 1^n, S)$
  1. Client locally sorts $S$ and sets $k = |S|$.
  2. Set $v_1 = S[1]$;
  3. For $i = 2, \ldots, k$, set $v_i = S[i] - S[i-1]$.
  4. Client locally constructs online partial sums data structure $D_{\mathsf{Sum}} \leftarrow \mathsf{PartialSums.Init}(v_1, \ldots, v_k)$. Note
     that, as a result, it holds that $\mathsf{PartialSums.Sum}(D_{\mathsf{Sum}}, i) = S[i]$ for all $i \in [k]$.
  5. Randomly select $r \in [n/k]$ and a permutation $\tau$ over $[k]$. The algorithm will embed $S$ into the $r$-th
     row of the matrix in the order described by $\tau$.
  6. Initialize $\ell \leftarrow 1$.
  7. While $\ell \leq k$:
     (a) Initialize $\mathsf{rank}_\ell \leftarrow \mathsf{PartialSums.Sum}(\tau(\ell))$.
     (b) Set $\texttt{unused} := n - (n/k)(\ell - 1)$.
     (c) Randomly select a $\lambda$-bit seed for a pseudorandom permutation $\gamma$ over the set $[\texttt{unused}]$.
     (d) Initialize $\mathsf{pivot} \leftarrow \gamma^{-1}(\mathsf{rank}_\ell) - r$.
     (e) For $i = \mathsf{pivot}, \ldots, \mathsf{pivot} + (n/k) - 1 \mod \texttt{unused}$:
         i. Execute $j \leftarrow \mathsf{PartialSums.Select}(D_{\mathsf{Sum}}, \gamma(i))$.
         ii. If $\mathsf{PartialSums.Sum}(D_{\mathsf{Sum}}, j) = \gamma(i)$ and $\gamma(i) \neq \mathsf{rank}_\ell$ then:
             A. Go back to Step 7c.
     (f) For $i = \mathsf{pivot}, \ldots, \mathsf{pivot} + (n/k) - 1 \mod \texttt{unused}$:
         i. Execute $j \leftarrow \mathsf{PartialSums.Select}(D_{\mathsf{Sum}}, \gamma(i))$.
         ii. Execute $D_{\mathsf{Sum}} \leftarrow \mathsf{PartialSums.Update}(D_{\mathsf{Sum}}, j, -1)$.
     (g) Initialize $\gamma_\ell \leftarrow \gamma$.
     (h) Initialize $\mathsf{pivot}_\ell \leftarrow \mathsf{pivot}$.
     (i) Increment $\ell$ by 1.
  8. Return description of partition $(\gamma_1, \ldots, \gamma_k, \mathsf{pivot}_1, \ldots, \mathsf{pivot}_k)$.
```

$\mathsf{OCP.ExtractPartition}$. This algorithm takes as input the description of a partition and expands it into an explicit description of the partition.

```
$(P_1, \ldots, P_{n/k}) \leftarrow \mathsf{OCP.ExtractPartition}(\gamma_1, \ldots, \gamma_k, \mathsf{pivot}_1, \ldots, \mathsf{pivot}_k)$.
  1. Construct a vector of $n$ integers, $v = (1, \ldots, 1)$.
  2. Execute $D_{\mathsf{Sum}} \leftarrow \mathsf{PartialSums.Init}(v)$.
  3. For $i = 1, \ldots, n/k$:
     (a) For $j = 1, \ldots, k$:
         i. Initialize $\ell_j \leftarrow \mathsf{PartialSums.Select}(\gamma_i(\mathsf{pivot}_i + j))$.
     (b) Set $P_i \leftarrow \{\ell_j\}_{j \in [k]}$.
     (c) For $j = 1, \ldots, k$:
         i. Execute $D_{\mathsf{Sum}} \leftarrow \mathsf{PartialSums.Update}(D_{\mathsf{Sum}}, \ell_j, -1)$.
  4. Return $(P_1, \ldots, P_{n/k})$.
```

E Private Batched Sum Retrieval

In this section, we present several PBSR schemes that have better network costs than StreamPBSR. However, each of these schemes introduce significant CPU costs prohibiting them from practical use.

E.1 Constant Bandwidth From Homomorphic Encryption

We present a constant bandwidth PBSR scheme built from homomorphic encryption that has both an homomorphic plaintext absorption and homomorphic addition (see Section 7 for a more detailed explanation). This scheme will upload c vectors of n homomorphic encryptions of either $k-1$ encryptions of 1 corresponding to the addends and all remaining encryptions are 0. For each vector, the server will simply absorb each database record into the corresponding homomorphic encryption and homomorphically add all ciphertexts together. Once receiving the final ciphertext, the client decrypts to retrieve the sum of its $k-1$ chosen records. This technique requires only $O(c)$ records to download, $O(n)$ ciphertext to upload and $O(nc)$ server computation. This technique is useful when records are very large. The number of records downloaded is minimized at the cost of larger uploads of ciphertexts of size λ.

E.2 Private Batched Retrieval

Private batched retrieval is a generalization of private information retrieval where any number of blocks can be downloaded privately instead of just a single block as described in private information retrieval. Formally, private batched retrieval considers downloading m blocks from a database of n blocks, where each block consists of B bits with security parameter λ. Private batched retrieval has been studied by many previous works [7, 22, 26, 30]. We now present BatchPBSR using any private batched retrieval scheme.

Construction 22. $(O_1, \ldots, O_c) \leftarrow \mathsf{BatchPBSR}(S_1, \ldots, S_c)$

1. Initialize $O_1 \leftarrow 0, \ldots, O_c \leftarrow 0$.
2. Compute $S = S_1 \cup \ldots \cup S_c$. If $|S| < ck$, pad with any arbitrary set of extra blocks until S is of size ck.
3. Execute a private batched retrieval scheme to download S.
4. For all $\mathsf{B}_i \in S$:
   (a) For all $j \in [c]$ such that $i \in S_j$:
      i. $O_j \leftarrow O_j + \mathsf{B}_i$
5. Return $(O_1, \ldots, O_c)$.

Theorem 23. BatchPBSR is a private batched sum retrieval scheme according to Definition 5.

In particular, we can use the work of Groth *et al.* [26] which presents a private batched retrieval with information theoretically optimal communication complexity of $O(m \log n + \lambda + mB)$ to retrieve m records. A more practical approach might use the private batched retrieval from cuckoo hashing by Angel *et al.* [7]. In either case, BatchPBSR is more communication efficient than StreamPBSR but at the cost of more computational costs.

E.3 Batch Codes and Homomorphic Encryption

Batch codes [7, 8, 29, 30, 47] are a technique used by many private batched retrieval to reduce the problem into running a private information retrieval scheme. Formally, a batch code is parameterized by the values (n, N, k, m, t). A database of n blocks, denoted by x, is encoded using the encoding function $C(x)$ into k buckets such that a total number of N blocks appear in all k buckets. Batch codes ensure the following important property: for any subset $S \subseteq [n]$ with at most m items, the set $\{\mathsf{B}_i\}_{i \in S}$ may be retrieved by reading at most t items in each of the k buckets. The majority of works on batch codes [30, 7] consider the case $t = 1$. In particular, we will use cuckoo batch codes described by Angel *et al.* [7].

We now present BatchCodePBSR, which will use a $(n, n' = O(n), m' = O(m), m, 1)$-batch code as well as a private information retrieval scheme using additively homomorphic encryption [31, 42, 19, 14, 13]. We assume that the database has already been encoded using the chosen batch code.

Construction 24. $(O_1, \ldots, O_c) \leftarrow \mathsf{BatchCodePBSR}(S_1, \ldots, S_c)$.

1. Let $S = S_1, \ldots, S_c$ and we set $m = ck$ as the number of blocks we wish to retrieve.
2. Perform a PIR request to each of the m' buckets to retrieve the block from the bucket necessary to decode S. Do not download the results of the PIR query, instead store them on the server as $r_1, \ldots, r_{O(m)}$. As a result, each of $(O_1, \ldots, O_c)$ is now a sum of a subset of the results of the $O(m)$ PIR queries.
3. For $i = 1, \ldots, c$:
   (a) Construct a vector $(e_1, \ldots, e_{O(m)})$ where e_j is an encryption of 1 if and only if O_i requires the block from the i-th bucket as part of the sum and e_j is an encryption of 0 otherwise.
   (b) Upload $(e_1, \ldots, e_{O(m)})$.
   (c) Server homomorphically computes $O_i = e_1 \cdot r_1 + \ldots + e_{m'} r_{m'}$ and returns O_i to the client.
   (d) Client decrypts O_i.
4. Return $(O_1, \ldots, O_c)$.