Efficient Stateful PIR Using Format-Preserving Encryption

Pranav Shriram Arunachalaramanan
University of Illinois Urbana-Champaign

Ling Ren
University of Illinois Urbana-Champaign

Abstract

Recently, Stateful Private Information Retrieval (PIR) has emerged as a promising new paradigm of PIR. Despite significant recent progress, state-of-the-art schemes in this paradigm still suffer from practical inefficiencies in communication, computation, and/or client storage. In this work, we construct a new stateful PIR scheme called HarmonyPIR that achieves efficient communication, computation, and client storage. From a technical standpoint, we build on the recent work of Wang and Ren (EUROCRYPT 25) and propose a new hint organization that uses only a single random permutation. The random permutation can be instantiated using either AES or the recently standardized FF1 Format-Preserving Encryption, yielding two variants of HarmonyPIR. Our new scheme achieves up to two orders of magnitude better amortized computation and up to five times better amortized communication than state-of-the-art schemes, both in the single-server and two-server setting.

1 Introduction

Private information retrieval (PIR) is a protocol between a server holding a public database and a client wishing to retrieve database entries from the server, without the server learning which entries are retrieved. PIR is a fundamental cryptographic primitive and finds widespread application in a variety of problems, such as anonymous communication [2, 32, 38], secure advertising [4, 22, 29, 45], password check [47], safe browsing [31], privacy-preserving media streaming [23], and many more.

PIR was proposed in the work of Chor et. al. [8]. Their work defined PIR in the standard setting, where the server stores the database in its original form and the client does not store any information about the database locally. A long list of subsequent works [1, 5, 6, 14, 36, 37, 41, 49] have significantly improved the efficiency of PIR in this setting. However, PIR schemes in the standard setting suffer from a fundamental barrier that the server must operate on every database entry for each client *query*. If the server does not operate on a specific entry during a query, the server learns that the client is not interested in that entry, violating the privacy guarantee of PIR.

Stateful PIR is a recent and promising direction to overcome this barrier. A stateful PIR scheme consists of an *offline* phase and an *online* phase. During the offline phase, the client and the server jointly perform a preprocessing step that allows the client to sample information regarding the database called *hints*. The client stores the hints locally. During the online phase, the client uses the hints to retrieve database entries efficiently. Very recently, stateful PIR has been considered for clients to privately retrieve *blockchain state* [17, 18].

A long list of works [3, 9–11, 19, 21, 24, 34, 35, 42, 43, 48, 52] have made significant progress in improving the computation, communication, and storage costs of stateful PIR. However, state-of-the-art schemes [3, 43, 48] still suffer from major practical inefficiencies. For example, the state-of-the-art practical single-server stateful PIR scheme of Ren et al. [43] requires large client storage in its default setting. For a 32 MB database consisting of 2^{20} entries of 32 Bytes each, the client needs to store 6.25 MB of data, which is about 20% of the database size. While its client storage can be reduced with alternative parameter choices, doing so blows up the communication and computation costs proportionally. On the other hand, the scheme of Wang and Ren [48], which builds on the *permutation-based hints* approach introduced by [35], requires only 0.07 MB of client storage for the same 32 MB database above. But its client computation is hundreds of times higher than that of [43]. Balanced PIR [3] strikes a balance between client storage and client computation, but has costly communication.

In summary, no existing stateful PIR scheme achieves good practical efficiency in all aspects. Our goal is to construct such a stateful PIR scheme that is efficient in *all three major metrics*: communication, computation, and client storage.

Perhaps surprisingly, our construction builds on the permutation-based hints approach [35, 48], which so far has been dismissed as being purely theoretical [3, 20] due to its use of *small-domain pseudorandom permutation (PRP)* [39], a

Table 1: Approximate and relative amortized computation and communication of various single-server stateful PIR schemes.

Scheme	Computation	Communication	Efficient Updates?
Ren et al. [43]	$\sim 15\times$	$\sim 3\times$	✗
WR PIR [48]	$\sim 100\times$	$1\times$	✓
Balanced PIR [3]	$1\times$	$\sim 5\times$	✗
HarmonyPIR0	$\sim 15\times$	$\sim 1\times$	✓
HarmonyPIR1	$1\times$	$\sim 1\times$	✓

cryptographic primitive believed to be costly in practice. Our work disproves this belief. Through new PIR techniques and improved small-domain PRP constructions, we obtain practically efficient stateful PIR schemes using the permutation-based hints approach.

1.1 Our Contributions

We construct a stateful PIR scheme called HarmonyPIR that has efficient computation, communication, and client storage. Building on WR PIR [48], HarmonyPIR uses a new organization of hints that only requires a *single* small-domain PRP. We instantiate the small-domain PRP in two ways, resulting in the following two variants of HarmonyPIR.

HarmonyPIR1. Empirical Format-Preserving Encryption (FPE) schemes offer an efficient way to instantiate small-domain PRPs. One such scheme is the *FF1* FPE, recently standardized by NIST [15]. HarmonyPIR1 uses FF1 as the small-domain PRP. For strong security guarantees, HarmonyPIR1 requires the database to contain at least half a million entries, per NIST recommendation [15]. We remark that WR PIR, despite having a similar structure to our scheme, cannot leverage FF1 for many practical database sizes. We discuss this in more detail in Section 2.

HarmonyPIR0. We also present a less efficient variant called HarmonyPIR0, whose security relies only on the time-tested Advanced Encryption Standard (AES) [16] block cipher. Specifically, HarmonyPIR0 uses a small-domain PRP whose security is provably reduced to an underlying *Pseudorandom Function* (PRF), which can be instantiated using AES. We observe that the strongest form of PRP security is not required for HarmonyPIR0. We use a weaker PRP from Hoang et al. [26], which has 50% fewer PRF (AES) calls than the PRP [39] used in WR PIR [48]. We also propose improvements to the PRP of Hoang et al. PRP [26] to reduce its PRF (AES) calls by another 75%. This part may be of independent interest.

Efficiency. We compare the efficiency of HarmonyPIR with state-of-the-art schemes WR PIR [48], Ren et al. [43], and Balanced PIR [3]. In the comparison, all schemes are normalized to use a similar amount of client storage.

In the single-server setting, the amortized computation of HarmonyPIR1 is roughly $100\times$ better than WR PIR [48], 15–$20\times$ better than Ren et al. [43], and comparable to Balanced PIR [3]. The amortized computation of HarmonyPIR0 is 15–$20\times$ worse than that of HarmonyPIR1 and is comparable to Ren et al. [43]. The amortized communication of HarmonyPIR (both variants) is $3.2\times$ better than Ren et al. [43], $5.2\times$ better than Balanced PIR [3], and comparable to WR PIR [48].

In short, single-server HarmonyPIR1 is significantly better than all the other schemes, but uses a newly standardized cryptographic construction. HarmonyPIR0 offers better communication but worse computation compared to state-of-the-art schemes [3, 43]. Since communication is generally more costly than computation in real-world deployments, we believe HarmonyPIR0 offers a better trade-off than prior works under the same standard cryptographic assumption.

In the two-server setting, if the database has small entries (e.g., 32 bytes), HarmonyPIR provides advantages over state-of-the-art schemes [3, 43, 48] similar to those in the single-server setting. For large entry sizes, HarmonyPIR1 still has the lowest computation cost, but Ren et al. [43] has the best communication cost.

Database updates. When a database entry is updated, HarmonyPIR offers an efficient way to update client hints with just $\tilde{O}(1)$ work (both communication and computation), a property inherited from permutation-based hints [35, 48]. In contrast, most schemes outside the permutation-based hints approach [3, 9, 10, 33, 42, 43, 52] do not support efficient updates. In particular, the two state-of-the-art baselines [3, 43] require $\tilde{O}(\sqrt{N})$ client computation per update.

2 Technical Overview

We give an overview of how single-server HarmonyPIR is constructed; two-server HarmonyPIR will follow naturally from this construction. Our starting point is the PIR scheme of Wang and Ren [48] (WR PIR). We now give a brief recap of their scheme, with an emphasis on the details that are relevant to this overview.

2.1 A Brief Recap of WR PIR [48]

The database DB of N entries is divided into T *partitions*, each of size $M = N/T$ consecutive database entries.

Offline phase. During the offline phase, the client samples hints according to a *hint table* consisting of T rows and $2M$ columns. Row i of the hint table contains the M database indices from partition i plus M empty cells. The $2M$ values in each row are permuted according to a *random permutation*.

The client computes one hint parity for each column in the hint table. Specifically, hint parity j is the combined parity of database entries at the indices in column j of the hint table.

Online phase. Given a query index q, the client first finds the column in the hint table that contains q (notice that only a single location in the hint table contains q). The client will use the indices in this column and the hint parity of this column to retrieve $\mathsf{DB}[q]$. To do so, the client computes a *request* consisting of the indices in this column, except that the query index is replaced with a random value from its row in the hint table. The client sends the request to the server and receives a *response* consisting of the database entries at the indices in the request. The client computes the queried database entry $\mathsf{DB}[q]$ using the response and the column hint parity.

The client then relocates each value in the column used for the query to a random empty cell within its row, and the hint parities are updated accordingly. The column used for the query is then deleted from the hint table.

Notice that after M queries, there are no more empty cells left in the hint table, and the client cannot make any more queries. In this case, the client reruns the offline phase with a new hint table (which uses different random permutations) before making further queries.

Relocation data structure and efficiency. WR PIR requires an efficient data structure to track the contents of the hint table. Storing the hint table naïvely would incur $\tilde{O}(N)$ storage (the $\tilde{O}$ notation hides the index size and entry size). They design a new data structure called the *relocation data structure* to instantiate the hint table using only $\tilde{O}(M)$ storage. Importantly, the hint table requires T random permutations, each of size $2M$, which are instantiated using the small-domain PRP of Morris and Rogaway [39].

Both communication and computation of WR PIR is $\tilde{O}(T)$. However, client computation is expensive in practice as it involves $O(T)$ invocations of a costly small-domain PRP [39].

2.2 Our Construction

The key idea. Our PIR scheme HarmonyPIR uses a new hint organization, which requires only a *single random permutation* of size $2N$. This allows us to efficiently instantiate the random permutation in HarmonyPIR through two ways: i) using an empirical FPE called FF1 [15], and ii) using the small-domain PRP from Hoang et al. [26] (with several new improvements), to get the two variants of our scheme HarmonyPIR1 and HarmonyPIR0.

It is important to note that an instantiation of the random permutation using FF1 is not possible in WR PIR. Recall that WR PIR requires T random permutations, each of size $2M$. To instantiate the random permutations using FF1, the size of the permutation ($2M$) must be at least a million for strong security per NIST recommendation [15]. In other words, M must be at least half a million. Recall that communication and computation in WR PIR depend on T, whereas client storage depends on M. To balance the two metrics, it is natural to choose $T \approx M \approx \sqrt{N}$ since $M \cdot T = N$. This means the database size N must at least be a quarter trillion, which rules out many practical application scenarios.

We now describe at a high level the construction of HarmonyPIR.

The hint row. Hints in HarmonyPIR are sampled according to a list of $2N$ cells, called the *hint row*. The database indices $\{0,1,2,\ldots,N-1\}$, hereafter denoted as $[N]$, appear exactly once in the hint row, and the remaining N cells are empty. The $2N$ values are randomly permuted according to a random permutation. For any given T, the cells in the hint row are partitioned into $M = 2N/T$ *segments*, of size T, where each segment $i \in [M]$ consists of the cells in the range $P_i = \{i \cdot T, i \cdot T + 1, \ldots, i \cdot T + T - 1\}$. We remark that the parameters T and M play similar roles in our scheme and in the WR scheme: M is the number of hints the client samples, and T determines the communication and computation.

We will now describe a PIR scheme when the client stores the entire hint row. Notice that this would require $\tilde{O}(N)$ client storage. In our actual scheme HarmonyPIR, we instead adapt the relocation data structure from the work of Wang and Ren [48] to achieve $\tilde{O}(M)$ client storage for the hint row. However, the following PIR scheme conveys the key ideas that are used in the construction of HarmonyPIR.

Offline phase. During the offline phase, the client computes a list H of hint parities, one for each segment of the hint row. The parity for segment i is $\mathsf{H}[i] = \bigoplus_{j \in P_i} \mathsf{DB}[\mathsf{HR}[j]]$, where HR denotes the hint row.

Online phase. In the online phase, given a query index q, the client finds the segment s containing q in the hint row. The client then uses the values in segment s of the hint row and the hint parity $\mathsf{H}[s]$ to fulfill its query. In more detail, let Q denote the request. All the values in segment s are first added to Q. However, if Q is sent to the server as such, the protocol would be clearly insecure, as the query index would always appear in the request. Instead, the query index must appear in the request with the same probability as any other index. To achieve this, with a certain probability (which we specify in Section 5), the query index in Q is replaced with a value chosen at random from the rest of the hint row. This way, the request appears to be a random selection of T values from the hint row to the server. We illustrate request construction in Figure 1.

The client sends Q to the server and receives a *response* R from the server consisting of the database entries at the non-empty indices in Q. Observe that the client now has all the database entries that were used to compute $\mathsf{H}[s]$, except $\mathsf{DB}[q]$. Then, the client can compute $\mathsf{DB}[q]$ as the combined parity of $\mathsf{H}[s]$ and the database entries at indices in segment s (except $\mathsf{DB}[q]$).

The server has now seen the indices in segment s (except the query index). If the client makes another query for an index in s, the request would leak information to the server. To maintain privacy, the client relocates the values in segment

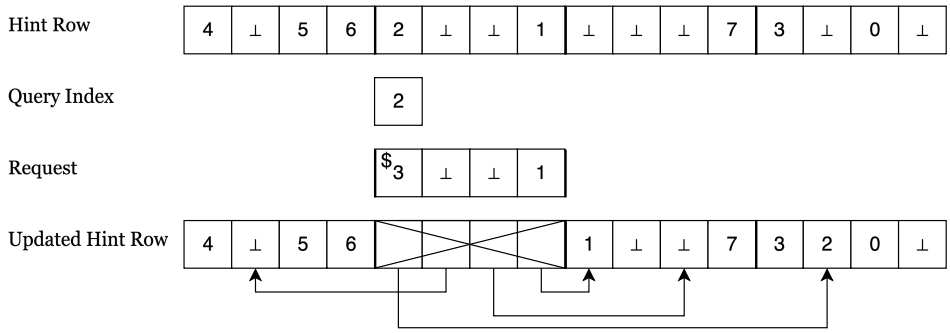


Figure 1: This depicts a hint row for a database of size 8 with $T = 4$, the request for the query index 2, and the updated hint row after the request.

s to random empty cells in the hint row, one at a time. We illustrate the relocations in Figure 1. The hint parities H are updated according to these relocations: each hint $\mathsf{H}[i]$ must match the combined parity of database entries at indices in segment i of the hint row. The segment s in the hint row and the hint parity $\mathsf{H}[s]$ are then deleted. Like WR PIR, after $M/2$ queries, there are no more empty cells left in the hint row, and the client reruns the offline phase with a new hint row.

Summary. The main novelty of our scheme is a new hint organization that does not rely on partitions and requires only a single (pseudo-)random permutation. For this to work, we adapt the relocation data structure in [48]. Then, the remainder of the PIR scheme follows somewhat naturally and bears similarities to prior works at a high level. Specifically, the query phase is similar to [35, 48] and the relocation phase is similar to [48]. However, we present several new low-level optimizations that are critical to achieving practical efficiency. In the end, both variants of HarmonyPIR have the same communication cost as WR PIR [48], but significantly improve computation. HarmonyPIR1 improves client computation by a λ factor, and HarmonyPIR0 improves by a large constant factor over that of WR PIR.

3 Model and Preliminaries

3.1 Stateful PIR

In Private Information Retrieval (PIR), a server holds a database DB of N entries, and a client has a *query index* q. At the end of the protocol, the client should retrieve the q-th entry in DB (correctness) and the server does not learn any information about q (privacy).

We now formally define stateful PIR, first in the single-server setting. A stateful PIR scheme consists of two phases of execution: i) an offline phase, and ii) an online phase. During the offline phase, the client and the server perform preprocessing together, and the client samples useful information regarding the database called *hints*. The client then stores the hints locally. During the online phase, the client has a sequence of query indices and executes a sequence of *queries* to retrieve the database entries at the query indices. The interaction between the client and the server during each query is as follows:

Given a query index, the client constructs a *request* Q and sends the request to the server. The server computes a *response* R using Q and sends it back to the client. The client then computes an *answer* using R.

For any sequence of W queries with query indices $q_0, q_2, \ldots, q_{W-1}$, a stateful PIR scheme must satisfy the following conditions of correctness and privacy.

Correctness. The answers computed by the client must match the database entries at the query indices, except with negligible (in the security parameter) probability.

Privacy. The view of the server/adversary in the PIR protocol contains "no information" about the client's query indices.

More formally, consider an experiment in which the adversary, acting as the server, interacts with the client to execute a sequence of W queries. Each query index is *adaptively* chosen by the adversary based on its view (which contains previous requests) and handed to the client. The PIR protocol satisfies privacy if there is a simulator that can take the place of the client in this experiment to interact with the adversary but never receive the query indices, such that the adversary's view is computationally indistinguishable in the two cases. A full formal definition of stateful PIR is in Section A.1.

The two-server setting. The model and definitions of single-server stateful PIR extend naturally to the two-server setting. Let S_0 and S_1 be two non-colluding servers, each holding a copy of the database. The client interacts with server S_0 to obtain hints and with server S_1 to execute queries. The correctness definition remains the same, and the privacy definition must hold against an adversary that can choose to corrupt either server.

3.2 Random Permutation and Small-domain Pseudorandom Permutation (PRP)

Random permutation. For any N', let $[N']$ denote the set $\{0,1,2,\ldots,N'-1\}$. A *random permutation* is a bijective function that maps inputs in a domain $[N']$ to random outputs in $[N']$. We refer to N' as the *size* of the random permutation.

Small-domain PRP. We first define a small-domain PRP. Given a mapping $P:[N']\times\{0,1\}^\lambda\to[N']$, for any $k\in\{0,1\}^\lambda$, let $P(*,k)$, denoted as P_k, be a permutation and P_k^{-1} be its inverse permutation. Let $\Pi(N')$ be the set of all permutations of size N'. At a high level, P is a q-secure small-domain PRP if no adversary making at most q queries to its oracles can distinguish oracle access to (P_k,P_k^{-1}), for a randomly sampled k, from oracle access to (g,g^{-1}) for a uniformly random $g\in\Pi(N')$. We give the full formal definition in Section A.2.

3.3 Relocation Data Structure

In this subsection, we present the details of the relocation data structure DS from the work of Wang and Ren [48] that are relevant to our construction.

Definition. Let $[N']$ denote the set of values $\{0,1,2,\ldots,N'-1\}$. The relocation data structure DS stores a list of $2N'$ cells, where each cell contains a value in the set $S=[N']\cup\{\perp\}$. Each value in $[N']$ appears exactly once in DS, and the remaining N' cells are empty (denoted $\perp$). DS supports operations to access the value in a cell (Access), find the cell containing a value (Locate), and relocate the value in a cell to a random empty cell (Relocate). DS uses a list C to store *relocation history*, i.e., the list of cells from which values have been relocated. To elaborate, the interface of Access, Locate and Relocate are as follows.

1. $\mathsf{Access}(c)\to v$: Given a cell $c\in[2N']\backslash C$, returns the value $v\in S$ in it. Access does not update C.
2. $\mathsf{Locate}(v)\to c$: Given a value $v\in[N']$, returns the cell $c\in[2N']\backslash C$ that contains v. Locate does not update C.
3. $\mathsf{Relocate}(c)$: Given a cell c, moves the value in c to a random empty cell that is not in C. In addition, Relocate adds c to C.

For convenience, when we say that a cell c has relocated, we mean that a value has been relocated *from* cell c to a random empty cell.

DS must satisfy the following properties of correctness and randomness invariance.

Correctness.

1. Initially, and after every Relocate operation, each value in $[N']$ appears exactly once in DS'.
2. Initially, and after every Relocate, Access and Locate are inverses of each other. That is, for all $v\in[N']$, $c=\mathsf{DS.Locate}(v)$, *if and only if* $v=\mathsf{DS.Access}(c)$.
3. A Relocate operation moves the value from a given cell to an empty cell that has not been relocated, and all other cells remain unchanged. Consider any Relocate operation: $\mathsf{DS.Relocate}(c)$ on some cell $c\in[2N']\backslash C$. Let DS^* be used to denote the relocation data structure after relocation, and $v\leftarrow\mathsf{DS.Access}(c)$ and $c^*\leftarrow\mathsf{DS}^*.\mathsf{Locate}(v)$. Then, i) $c^*\notin C$ and $\mathsf{DS.Access}(c^*)=\perp$, and ii) $\mathsf{DS.Locate}(v')=\mathsf{DS}^*.\mathsf{Locate}(v')$ for all $v'\in[N']\backslash\{v\}$.

Randomness Invariance. At a high level, this property guarantees that after any number of Relocate operations, the values in the cells of the relocation data structure are permuted randomly, even when conditioned on the values at the cells that were previously relocated. We describe this formally below.

Let $U_{N',q}$ be a multiset containing all the values in $[N']$ and q empty symbols ($\perp$). For any multiset X, let $\mathcal{P}(X)$ denote the uniform distribution over the set of all permutations of the values in X, and $\mathcal{P}_T(X)$ denote the distribution of T values sampled without replacement from X.

Given a relocation data structure DS with $2N'$ cells, let R be any list of N' cells of DS. Relocate operations are performed on the cells in R, one by one. DS satisfies randomness invariance if the following holds:

- Let DS^i denote the relocation data structure *before* the i-th ($i\in[N']$) relocation operation, and C^i denote the relocation history of DS^i. The list of values in DS^i: $A_i=[c\in[2N']\backslash C^i,\mathsf{DS}^i.\mathsf{Access}(c)]$ follows the distribution $\mathcal{P}(U_{N',N'-i})$. This holds *even* when conditioned on the values which have been relocated: $[DS^0.\mathsf{Access}(R[0]),\ldots,DS^{i-1}.\mathsf{Access}(R[i-1])]$ (only applicable for $i\geq 1$).

History data structure. The construction of DS from the work of Wang and Ren [48] uses a simple *history data structure* Hist to track relocation history C; given a position in C, Hist supports finding the cell in that position (index-based lookup), and given a cell, Hist supports returning the position which contains the cell in C (value-based lookup). Hist simply uses a list L to support index-based lookup of C and a map M to support value-based lookup of C. The details of this simple data structure are in Algorithm 1.

Summary of DS. DS samples a random permutation P of size $2N'$ to determines the initial contents of the cells in DS. The cells $P(0),P(1),P(2),\ldots,P(N'-1)$ are assigned the values $0,1,2,\ldots,N'-1$, respectively, and the cells $P(N'),P(N'+1),\ldots,P(2N'-1)$ are empty. We call the values $N',N'+1,\ldots,2N'-1$ as "empty values". DS also initializes the history data structure Hist (Algorithm 1).

Relocate operation on any cell moves the value in the cell to the "next" empty cell. Specifically, the i-th Relocate operation on any cell c, relocates the value in that cell to the

Algorithm 1 History Data Structure Hist

State: L: List of relocated cells, M: Map between each relocated cell to its position in L

Hist.Init()

1: Initialize L as an empty list and M as an empty map

Hist.Append(c)

2: L.append(c)

3: $M[c] \leftarrow |L|-1$

Hist[p]: **return** $L[p]$

Hist^{-1}[c]: **return** $M[c]$

cell containing the i-th empty value, i.e., the cell containing $N'+i-1$. To relocate the value in a cell c, the Relocate operation simply performs Hist.Append(c).

Access and Locate use Hist and P to satisfy their functionalities. We give full details on Access and Locate in Section B for completeness, but understanding those details of DS is *not required*. To understand our work, it is sufficient to know the following: i) functionalities of Access, Locate, and Relocate, ii) construction of Hist, and iii) Access and Locate use *only* on the history Hist and the permutation P along with their inputs to achieve their required functionalities.

Lemma 1. *Relocation data structure* DS *[48] satisfies correctness and randomness invariance as defined in Section 3.3.*

4 A Restricted Relocation Data Structure

In this section, we define a "restricted" variant of the relocation data structure adapted from Section 3.3, and describe an efficient construction for it. Looking ahead, we will use this restricted relocation data structure in our PIR scheme.

We first introduce a new notion of *segments* for the relocation data structure DS from Section 3.3, and a new operation called RelocateSegment. For any value of a parameter T, DS is partitioned into chunks of T contiguous cells called *segments*. In particular, for any $s \in [2N'/T]$, segment s holds the list of cells $[s \cdot T, s \cdot T+1, s \cdot T+2, \ldots, s \cdot T+T-1]$. We now define a new operation called RelocateSegment as follows:

- RelocateSegment(s): Given a segment s, RelocateSegment relocates the values in cells $[s \cdot T, s \cdot T+1, \ldots, s \cdot T+T-1]$, one after the other to random empty cells. The cells in segment s are added to the relocation history C.

One could trivially implement RelocateSegment on a segment s using a sequence of Relocate operations: Relocate($s \cdot T$), Relocate($s \cdot T+1$), ..., Relocate($s \cdot T+T-1$). However, observe that using the construction of the relocation data structure in Section 3.3, this would require $O(R \cdot T)$ storage to perform R RelocateSegment operations; this is because each Relocate operation adds an entry to the list L and map M of the history data structure Hist.

Algorithm 2 History Data Structure for Segment Relocations Hist$'$

State S: List of relocated segments, M: Map from relocated segment to its position in S, T: Segment Size

Hist$'$.Init(T)

1: Initialize S as an empty list and M as an empty map.

2: Hist.$T \leftarrow T$

Hist$'$.Append(s)

3: Append s to S

4: $M[s] \leftarrow |S|-1$

Hist$'$[p]

5: $s \leftarrow \lfloor p/T \rfloor$

6: **return** $S[s] \cdot T + (p \mod T)$

Hist$'^{-1}$ [c]

7: $s \leftarrow \lfloor c/T \rfloor$

8: **return** $M[s] \cdot T + (c \mod T)$

Consider a restricted variant of the relocation data structure that only allows Access, Locate, and RelocateSegment operations (does not allow Relocate operations). We denote this restricted relocation data structure as DS$'$. To support lookup of relocation history C in DS$'$, it is sufficient for its history data structure to store the relocated segments because cells within segments are relocated in a deterministic order. We describe such a simple history data structure Hist$'$ in Algorithm 2.

The RelocateSegment operation on a segment s of DS$'$ simply adds s to the list of relocated segments (using Hist$'$.Append(s) in Algorithm 2). Then, observe that DS$'$ only needs $O(R)$ client storage (rather than $O(R \cdot T)$) for performing R RelocateSegment operations, by using Hist$'$.

The Access and Locate functions of DS$'$ are the same as DS. They rely on index-based and value-based lookup of relocation history C, which is supported by Hist$'$.

Lemma 2. DS$'$ *satisfies correctness and randomness invariance as defined in Section 3.3.*

Proof. We can view every RelocateSegment operation in DS$'$ as a sequence of Relocate operations. Then, correctness and randomness invariance of DS$'$ follow from correctness and randomness invariance of DS (Lemma 1). □

5 HarmonyPIR

Recall the high-level idea of our single-server HarmonyPIR construction from Section 2. Using the restricted relocation data structure DS$'$, we first describe, prove the security of, and

Algorithm 3 Algorithms in HarmonyPIR Scheme

Client State DS': Restricted Relocation Data Structure, H: Hint parities, M: Number of hint parities, T: Segment Size, N: Number of database entries

PIR.Offline(N,T)

Initialize DS' with $2N$ cells and segment size T.
Initialize a list H of M hint parities to all zeros.
Stream the database from the server. For each k-th entry:

- $c \leftarrow \mathsf{DS}'.\mathsf{Locate}(k)$
- $\mathsf{H}[\lfloor c/T \rfloor] \leftarrow \mathsf{H}[\lfloor c/T \rfloor] \oplus \mathsf{DB}[k]$

PIR.Online(q)

Request

$c \leftarrow \mathsf{DS}'.\mathsf{Locate}(q)$
$s \leftarrow \lfloor c/T \rfloor$, $r \leftarrow c \mod T$
Initialize a list Q of size T
$\forall i \in [T], \mathsf{Q}[i] \leftarrow \mathsf{DS}'.\mathsf{Access}(s \cdot T + i)$
Let $|C|$ denote the number of relocated cells.
Flip a random Bernoulli coin χ with $p = T/(N-|C|)$.
if χ is tails **then**
  Initialize a multiset $L \leftarrow U_{N,N-|C|} \setminus \mathsf{Q}$.
  $\mathsf{Q}[r] \xleftarrow{\$} L$
end if

Response

Sends Q to the server and receive a response R of the database entries at indices in Q

Answer

$A \leftarrow \mathsf{H}[s] \bigoplus_{i \in [T] \setminus r} \mathsf{R}[i]$

Relocate

$\mathsf{DS}'.\mathsf{RelocateSegment}(s)$
$\forall i \in [T] \setminus \{r\}, d_i \leftarrow \lfloor \mathsf{DS}'.\mathsf{Locate}(\mathsf{Q}[i])/T \rfloor$
$\forall i \in [T] \setminus \{r\}, \mathsf{H}[d_i] \leftarrow \mathsf{H}[d_i] \oplus \mathsf{R}[i]$
$d_r \leftarrow \lfloor \mathsf{DS}'.\mathsf{Locate}(q)/T \rfloor$
$\mathsf{H}[d_r] \leftarrow \mathsf{H}[d_r] \oplus A$

analyze the full constructions of single-server HarmonyPIR. We then discuss two different ways to instantiate the random permutation used in the scheme. Finally, we show how the construction and the analysis can be naturally extended to the two-server setting.

5.1 Offline Phase

During the offline phase, given the database size N and a parameter T, the client initialises a relocation data structure DS' (Section 4) with $2N$ cells and segment size T. Recall that such a DS' consists of the database indices $[N]$ each appearing exactly once, and N empty cells.

The client computes a list of $M = 2N/T$ hint parities, one hint parity for each segment of DS'. Specifically, the i-th hint parity, denoted as $\mathsf{H}[i]$, is computed as the combined parity of the database entries at the indices in segment i of DS'. To do this, the hint parities H are first initialized to 0. Then, the client streams the database entry by entry from the server, and as it streams the k-th entry $\mathsf{DB}[k]$, the client adds $\mathsf{DB}[k]$ to the appropriate hint parity. That is, the client first finds the cell containing index k as $c \leftarrow \mathsf{DS}'.\mathsf{Locate}(k)$, computes c's segment $s = \lfloor c/T \rfloor$, and updates the hint parity $\mathsf{H}[s]$ to $\mathsf{H}[s] \oplus \mathsf{DB}[k]$. The hint parities are stored locally by the client. The full offline algorithm, which is executed by the client, is detailed in Algorithm 3.

5.2 Online Phase

Querying the database. Given a query index q, the client first locates q in DS': q is in cell $c = \mathsf{Locate}(q)$, which is in segment $s = \lfloor c/T \rfloor$; further, q is in the r-th cell of its segment where $r = c \mod T$.

First, the client copies the values in segment s of DS' into Q, i.e., $\mathsf{Q}[i]$ is set to the i-th value of segment s. However, observe that the query index q is guaranteed to be included, so sending Q as is to the server would be clearly insecure. Instead, q should appear in the request with the same probability as any other index. This probability is $p = T/(N-|C|)$ ($|C|$ is the number of relocated cells of DS') since T out of the remaining $N-|C|$ cells are selected. Thereby, to ensure privacy, q must be excluded from the request with probability $1-p$. To do this, let L be the multiset containing the remaining values in DS', i.e., $L = U_{N,N-|C|} \setminus \mathsf{Q}$. The client tosses a Bernoulli coin, which lands on heads with probability $p = T/(N-|C|)$. If this coin lands on tails (which happens with probability $1-p$), the client replaces the query index q in $\mathsf{Q}[r]$ with a uniformly random value in L. We prove this satisfies privacy in Theorem 2.

The client sends the request Q to the server. The server returns a response R which consists of the database entries at the non-empty indices in Q. Recall that the hint $\mathsf{H}[s]$ was computed as the combined parity of the database entries at indices in segment s of DS'. The client has obtained from the server the database entries at indices in segment s except $\mathsf{DB}[q]$. The client can compute $\mathsf{DB}[q]$ as the combined parity of $\mathsf{H}[s]$ and $\bigoplus_{i \in [T] \setminus r} \mathsf{R}[i]$ (the client uses $\mathsf{R}[i] = 0$ for all empty $\mathsf{Q}[i]$).

Relocating segment s. From the request Q, the server has learned the indices in segment s of DS'. To ensure privacy, the client refreshes DS' by relocating the values in segment s to random empty locations in DS'. To do this, the client simply executes $\mathsf{DS}'.\mathsf{RelocateSegment}(s)$.

Then, the hint parities need to be updated according to these relocations; each hint parity of a DS' segment must match the combined parity of the database entries at indices in that segment. Notice that the values originally in segment s are: i) the request values $\mathsf{Q}[i]$ for all $i \in [T] \setminus r$, and ii) the query

index q. The client finds the segments $d_0, d_1, \ldots, d_{T-1}$ that now contain each of these values using Locate as follows:

$$d_i = \begin{cases} \lfloor \mathsf{Locate}(\mathsf{Q}[i])/T \rfloor, & i \neq r \\ \lfloor \mathsf{Locate}(q)/T \rfloor, & i = r \end{cases}$$

The client then updates the hint parities of these segments by adding the database entries relocated into them:

$$\mathsf{H}[d_i] = \begin{cases} \mathsf{H}[d_i] \oplus \mathsf{R}[i], & i \neq r \\ \mathsf{H}[d_r] \oplus \mathsf{DB}[q] & i = r \end{cases}$$

Notice that the client can perform these updates locally, as the client retrieved each $\mathsf{R}[i]$ from the server and computed $\mathsf{DB}[q]$ locally (using the response and the hint parity). We also describe a more efficient method to perform segment relocation in Section C. This achieves a constant factor improvement in client computation over the current method. We defer this explanation to the appendix because it requires the reader to understand how Access and Locate are implemented.

The full online algorithm is detailed in Algorithm 3.

After each query, T of the empty cells in DS' are filled up. After $M/2$ queries, there will be no more empty cells left in DS', and the client cannot make queries anymore. Hence, the offline procedure needs to be executed again by the client before making more queries.

Handling Database Updates. Whenever a database entry at some index i gets modified, the server sends the combined parity of the original database entry $\mathsf{DB}[i]$ and the modified database entry $\mathsf{DB}'[i]$ to the client. Let us denote this value as $\mathsf{diff} = \mathsf{DB}[i] \oplus \mathsf{DB}'[i]$. The client changes the computed hint parities according to this modification; to do this, client finds the segment containing index i in DS' as $s = \lfloor \mathsf{DS}'.\mathsf{Locate}(i)/T \rfloor$ and changes the segment's hint parity $\mathsf{H}[s]$ to $\mathsf{H}[s] \oplus \mathsf{diff}$ (which amounts to removing the old value $\mathsf{DB}[i]$ and adding in the new value $\mathsf{DB}'[i]$).

5.3 Analysis

Lemma 3. *(Hint parities are correct) Before any query, let S denote the set of relocated segments in DS'. For all $s \in [M] \setminus S$, $\mathsf{H}[s]$ is equal to the parity of the database entries at the indices in segment s of DS'.*

Proof. We can show this by induction. Before the first query, each database index is at a unique location in DS' and can be found using Locate (the first two correctness conditions in Section 3.3); the client finds this location and updates the corresponding hint parity. Hence, the invariant in the lemma holds right after the offline phase.

For the inductive step, suppose the invariant holds before query j. After query j, indices in some segment s are relocated; the client finds the destination cells of the relocated indices (using Locate) and updates the corresponding hint parities. From the third correctness condition of DS' (Section 3.3), updating these hint parities preserves the invariant as the other cells of DS' remain unchanged. □

Theorem 1 (Correctness)**.** HarmonyPIR *satisfies correctness.*

Proof. Lemma 3 establishes the invariant that, for the selected hint segment s in any query, the hint parity $\mathsf{H}[s]$ equals the combined parity of the database entries at indices in cells $[s \cdot T, s \cdot T + 1, \ldots, s \cdot T + T - 1]$ of DS'.

Then, for any query index q, it is easy to see that $\mathsf{DB}[q]$ equals the combined parity of the hint $\mathsf{H}[s]$ and the database entries at indices in segment s except $\mathsf{DB}[q]$. This is precisely what the client computes; the client has $\mathsf{H}[s]$ stored locally and retrieves from the server the database entries at indices in segment s (except $\mathsf{DB}[q]$). □

Lemma 4. *For any sequence of $M/2$ queries to indices $q_0, q_1, \ldots, q_{M/2-1}$, each chosen adaptively based on the adversary's view, the request Q_i for the i-th query follows the distribution $\mathcal{P}_T(U_{N,N-T\cdot i})$, conditioned on the prior requests $\mathsf{Q}_0, \ldots, \mathsf{Q}_{i-1}$ and the query indices $q_0, \ldots, q_i$.*

We give the intuition here, and the full proof is at Section D. Observe from the request construction procedure that the first request Q to the server appears to be a selection of T random values from the values in DS', using the fact that the values in DS' are arranged randomly. For any values of a segment s of DS' that the server sees in a request, the values in this segment are relocated to random empty locations in DS', and the segment s is deleted. Therefore, from the server's perspective, any request appears to be a selection of T random values from the remaining values in DS'.

Theorem 2 (Privacy)**.** HarmonyPIR *satisfies privacy.*

Proof. We construct a simulator for the server using Lemma 4; the i-th request of the client is simulated using the distribution $\mathcal{P}_T(U_{N,N-T\cdot i})$. Such a simulation works for the adaptive adversary in Section 3.1 because Lemma 4 holds for any sequence of query indices. □

Intantiating the random permutation using a PRP. Recall the formal definition of small-domain PRP from Section 3.2. WR PIR [48] uses the $(2N)$-secure small-domain PRP of Morris and Rogaway [39] to instantiate its random permutation. However, we observe that it suffices to use a small-domain PRP that is (N)-secure. Roughly, this is because in both WR PIR and HarmonyPIR, the server (which is the adversary) obtains PRP evaluations at only half of the points in the domain. Specifically, the client makes only $M/2$ queries using the same DS' in HarmonyPIR, after which it reruns the offline phase with a different DS' (which will use a different permutation). During these queries, the server/adversary only sees the values at N $(= MT/2)$ cells of DS', and hence only

learns the values at N of the $2N$ positions in the underlying permutation. We prove that an (N)-secure small-domain PRP is sufficient for HarmonyPIR as follows:

Theorem 3. HarmonyPIR *satisfies privacy (Section 3.1) when the random permutation of size $2N$ (used in DS') is instantiated using an N-secure small-domain PRP.*

Proof. We prove in Section F that an adversary cannot differentiate whether the client is using an N-secure small-domain PRP or a random permutation to instantiate DS'. The lemma then follows from Theorem 2. □

Theorem 4 (Efficiency)**.** *Consider any database consisting of N entries each of size w bits, and any given parameter T. Over at least $M/2 = N/T$ queries,* HarmonyPIR *has:*

- *client storage of $O(Mw + M\log N)$ bits*
- *amortized communication of $O(Tw + T\log N)$ bits*
- *amortized client computation of $O(T)$ small-domain PRP evaluations and $O(T)$ w-bit XORs.*
- *amortized server computation of T lookups of w-bit entries.*

We prove this theorem in Section E.

5.4 Variants of HarmonyPIR

We instantiate the random permutation used by DS' in HarmonyPIR using a small-domain PRP (Section 3.2) in two different ways to obtain the following two variants of HarmonyPIR. The security of both variants follows from Theorem 3.

HarmonyPIR1. An empirical FPE scheme called FF1 [15] is assumed to be a $(2N-2)$-secure small-domain PRP for a permutation of size $2N$. We can securely instantiate the random permutation used by HarmonyPIR using FF1 [15], to get HarmonyPIR1. While practical attacks exist on FF1 on very small domain sizes [7, 12, 13, 25, 27], the NIST FF1 recommendation [15][Appendix A.2] specifies that for a domain of size at least 10^6, the resources required for best-known attacks on FF1 are prohibitive. Thus, to use FF1, the size of the hint row must be at least 10^6, which means that the database in HarmonyPIR must contain at least half a million entries.

HarmonyPIR0. We also instantiate the random permutation used by HarmonyPIR using our (N)-secure variant of the small-domain PRP of Hoang et al. [26], to get HarmonyPIR0. Unlike FF1, which is a relatively new cryptographic construction, this small-domain PRP is provably secure under the assumption that the well-established Advanced Encryption Standard (AES) is secure. We describe our improved (N)-secure variant of the small-domain PRP of Hoang et al. [26] in Section 6.

5.5 Two server HarmonyPIR

HarmonyPIR in the two-server setting is very similar.

Offline phase. The offline server S_0 (rather than the client) executes the offline algorithm in Algorithm 3, and sends DS' and the hint parities to the client. The client stores these locally. Notice that this significantly reduces the offline communication compared to single-server HarmonyPIR.

Online phase. The online phase of two-server HarmonyPIR is identical to that of single-server HarmonyPIR. Specifically, the client and the online server S_1 execute the online algorithm in Algorithm 3.

Analysis. It is easy to see that the correctness analysis (Theorem 1) directly applies to the two-server setting. The privacy analysis (Theorem 2) applies to a corrupt *online server*. Privacy against a corrupt *offline server* is straightforward: every $M/2$ queries, the simulator simply "pings" the offline server to obtain hints. The two instantiations of the random permutation described in Section 5.4 also apply to the two-server setting. Two-server HarmonyPIR has (almost) the same asymptotic efficiency as single-server HarmonyPIR (Theorem 4); for completeness, we detail this in Theorem 7 of Section G.1.

6 A Small-domain PRP for HarmonyPIR0

In HarmonyPIR0., we instantiate the random permutation using the small-domain PRP from the work of Hoang et al. [26], which satisfies the following:

Theorem 5 ([26])**.** *For any $N \geq 1$ and the security parameter λ, there exists a small-domain PRP P of size $2N$ that is (N)-secure, and the algorithms for P_k and P_k^{-1} both run in time $\Theta(\log N + \lambda)$.*

We now describe the small-domain PRP of Hoang et al. [26], and then present our improved variant of their construction, which we use in HarmonyPIR0.

6.1 The construction of Hoang et al. [26]

Random permutation as a card shuffle. We can view a random permutation from the lens of a card shuffle. To see this, consider N' cards, numbered from 0 to $N'-1$, where card i is initially at position i. The cards are then randomly shuffled. After the shuffle, the final position of card i is the evaluation of some random permutation P on input i. We will now use the card shuffle lens to describe the small-domain PRP of Hoang et al. [26].

The card shuffle of Hoang et al. [26] happens in r *rounds*. In each round $i \in [r]$, first, a *round key* $K[i]$ and a *round function* F_i are sampled as follows: $K_i \xleftarrow{\$} [N']$ and $F_i \xleftarrow{\$} \{f : [N'] \to \{0,1\}\}$. In round $i \in [r]$, cards at position X and $X \oplus K[i]$ are paired with each other and are swapped if and only if $F_i(\min(X, X')) = 1$.

Algorithm 4 The small-domain PRP of Hoang et al. [26]

Input X: Input to permutation, N': Domain Size, r: Number of rounds, K: A list of r round keys, F: A list of r round functions

for $i \in [r]$ **do**
  $X' \leftarrow X \oplus K[i]$
  $X_{min} \leftarrow \mathsf{min}(X, X')$
  if $F_i(X_{min}) = 1$ **then**
    $X \leftarrow X'$
  end if
end for
return X

Tracking a single card. The shuffle of Hoang et al. [26] supports efficiently finding the final position of a single card without tracking the positions of other cards. Recall that this is equivalent to evaluating a random permutation on a specific input.

Specifically, consider card X, which is initially at the position X. In each round $i \in [r]$, the card at position X is paired with the card at position $X' = X \oplus K[i]$, where $K[i]$ is the round key. Then, if the round function $F_i(\mathsf{min}(X', X))$ evaluates to 1, the card at position X is moved to X', following which, X is updated to be X'. After r rounds, the algorithm outputs X. The details of this simple algorithm are in Algorithm 4.

Practically, the round functions and the round keys can be instantiated using AES [16] with a key k. Specifically, the round function $F_i(x)$ is $\mathsf{AES}_k(\text{“function”}||i||x)$ and the round key $K[i]$ is $\mathsf{AES}_k(\text{“key”}||i)$. $\Theta(\log N' + \lambda)$ rounds of shuffle are needed to construct the $(N'/2)$-secure small-domain PRP required by HarmonyPIR (Theorem 3).

6.2 Our variant of Hoang et al.

High-level idea. The round functions F_i used in the shuffle of Hoang et al. [26] (Algorithm 4) need a single random bit, whereas the instantiation of F_i using AES outputs λ bits. Our variant utilizes the λ random bits more effectively, thereby reducing the total number of AES calls needed per invocation of the permutation.

Let X be the position of a given card at the start of round i in the shuffle of Hoang et al. (Algorithm 4). We make a key observation about the final position of this card after β rounds; we call a set of β rounds a *phase*. We first define a *phase group* G; let L be the list of round keys for rounds i up to $i+\beta-1$, and S be the power set of L; for each set $s \in S$, the combined parity of the keys in s and X is added to a list G. Observe that the card in position X at the start of round i can only go to a position in the phase group G after β rounds. This is because positions that differ from the card's position by the round key are also in G. And due to the same reason, the cards in any of the positions in G at the start of round i only end up at positions in G after β rounds.

Algorithm 5 Our optimized variant of the Small-domain PRP from Hoang et al. [26]

Input X: Input to permutation, N: Domain Size, r: Number of rounds, β: Phase size, k: AES key, K: A list of r round keys

for $i \in [r/\beta]$ **do**
  $G \leftarrow \mathsf{getPhaseGroup}(X, i)$
  $f \leftarrow \mathsf{AES}_k(\text{“function”}||i||G[0])$
  for $j \in [\beta]$ **do**
    $X' \leftarrow X \oplus G[j]$
    $X_{min} \leftarrow \mathsf{min}(X, X')$
    $p \leftarrow$ The smallest $k \in [\beta]$ such that $G[k]$ is X_{min}
    if $p \cdot \beta + j$ **then**
      $X \leftarrow X'$
    end if
  end for
end for
return X

getPhaseGroup(X, i)
$S \leftarrow \mathsf{powerset}(K[i \cdot \beta, i \cdot \beta + \beta - 1])$
$G \leftarrow []$
for $s \in S$ **do**
  $G.\mathsf{append}(X \oplus \bigoplus_{k \in S} k)$
end for
return $\mathsf{sort}(G)$

Using this observation, we use a *single* AES call to decide the outputs of round functions for positions in G during the β rounds. To do this, AES is evaluated on the smallest value in a sorted G, and for any two positions in G that get paired, a specific bit in the AES output is assigned to be the output of the round function.

In more detail, AES is first computed as $f = \mathsf{AES}_k(\text{“function”}||i||G[0])$. Each of the positions in the sorted list G are assigned β bits of f. The j-th ($j \in [\beta]$) bit assigned to a position can serve as its round function in round $i+j$. Specifically, for any two positions X and X' of G that get paired in the round $i+j$, $j \in [\beta]$, let p be the location of $X_{min} = \mathsf{min}(X, X')$ in G. Then, the round function is the j-th bit assigned to X_{min}, which is the $(p\beta + j)$-th bit of f. Observe that the round function is an independent random bit as required by the shuffle of Hoang et al. (Algorithm 4).

Tracking a single card. The algorithm for tracking a single card at a position X is a natural extension of Algorithm 4. In each phase, for a given position X, the round keys, the phase group G, and the round functions are computed. Then, the rounds within the phase follow the same procedure as the original shuffle. After the completion of all phases, the final position X is returned. We describe this algorithm in Algorithm 5.

For such a shuffle, observe that $2^\beta \cdot \beta$ must be smaller than the output of AES (which is 128 bits). This is because each

of the 2^β values in a phase group G is assigned a set of β bits from the output of a *single* AES invocation. Hence, we choose $\beta = 4$.

To evaluate the small-domain PRP on a single input, Algorithm 5 only requires r/β AES invocations. For comparison, Algorithm 4 needs r AES invocations. Therefore, our improved variant achieves a $4\times$ reduction in the number of AES invocations.

Storing round keys. In HarmonyPIR0, the client can precompute the round keys of the small-domain PRP in Algorithm 5 in the offline phase and store them locally. This amounts to $O(\lambda(\log N + \lambda))$ client storage. We note that this is not an option for WR PIR [48] because their scheme uses T instances of small-domain PRPs (one for each row of the hint table, refer to Section 2). Storing the round keys for all these small-domain PRPs would blow up the client storage $O(\lambda T(\log N + \lambda))$, which would defeat WR PIR's purpose of achieving efficient client storage. As a result, WR PIR needs to make $(r \log N)/128$ additional AES calls to compute round keys (as each AES call computes round keys for $128/\log N$ rounds) per small-domain PRP evaluation.

7 Implementation and Evaluation

7.1 Implementation Details

We implement both variants of HarmonyPIR using C++ in 900 lines of code. The implementation is available at HarmonyPIR (https://anonymous.4open.science/r/HarmonyPIR-13DB). We use the AES implementation in the CryptoPP library.

7.2 Experimental Setup

Baseline. We select the following schemes as our baseline. We use $\tilde{O}$ to hide the factors of $\log N$ and w (i.e., sizes of the indices and entries), but not the security parameter λ, in the asymptotics of the following schemes:

- The PIR scheme by Ren et al. [43] (RMS PIR): This scheme achieves $\tilde{O}(\sqrt{N})$ communication and computation while requiring $\tilde{O}(\lambda\sqrt{N})$ client storage, where λ is typically 80. We use their C++ implementation[1].
- Balanced PIR [3]: This scheme achieves $\tilde{O}(\sqrt{N})$ communication and expected $\tilde{O}(\sqrt{N})$ computation while requiring $\tilde{O}(\lambda'\sqrt{N})$ client storage, where $\lambda' < \lambda$. However, this comes at the practical cost of more expensive communication since both the request communication and response communication are $\tilde{O}(\sqrt{N})$. We use their C++ implementation[2].
- The PIR scheme by Wang and Ren [48] (WR PIR): This scheme achieves $O(\sqrt{N})$ communication and expected $\tilde{O}(\sqrt{N})$ computation while requiring only $\tilde{O}(\sqrt{N})$ client storage. However, its client computation is costly due to the use of the small-domain PRP of Morris and Rogaway [39]. We use a C++ implementation of this scheme from the work of Balanced PIR [3].

[1] https://github.com/renling/S3PIR
[2] https://github.com/PranavShriram/Balanced-PIR

Execution Environment. We run all our experiments on a MacBook Pro laptop that has the Apple M4 Pro processor with 24 GB RAM.

Methodology. We compare HarmonyPIR (both single-server and two-server) with each of the baselines separately. We let each baseline scheme use its preferred client storage, and tune HarmonyPIR to use that much (in fact, typically smaller) client storage. This methodology favors the baselines, sometimes quite significantly. This is because HarmonyPIR has a tight space-time tradeoff [50] (i.e., it trades off storage smoothly for communication and computation), whereas the baseline schemes [3, 43] do not.

We perform each comparison for databases consisting of $2^{20}, 2^{24}$ and 2^{28} entries, with 32 byte entries. We observe a similar comparison for entry sizes up to 256 bytes in the single-server setting (we omit these results).

7.3 Evaluation

7.3.1 Single-server HarmonyPIR

We compare single-server HarmonyPIR with WR PIR [48] in Table 2, RMS PIR [43] in Table 3, and Balanced PIR [3] in Table 4. We report communication, computation, and client storage costs of all schemes. For communication and computation, we report the offline cost, the online cost, and the amortized cost, which is the offline cost per query plus the online cost of each query. For offline computation, we report the time taken for the first offline phase. We mark metrics that are efficient in green and inefficient in yellow.

All the schemes stream the entire database during the offline phase, following which the offline communication of all these schemes is identical and equal to the database size. This is a common drawback of most schemes in the single-server stateful setting; schemes that circumvent this drawback suffer from other inefficiencies [28, 41].

Comparison with WR PIR [48] (Table 2). The communication and server computation in HarmonyPIR and WR PIR are identical; they differ only in client computation, due to reasons we have described before. Concretely, the amortized computation of HarmonyPIR1 is two orders of magnitude better than the amortized computation of WR PIR, and the amortized computation of HarmonyPIR0 is $6\times$ better than the amortized computation of WR PIR.

Comparison with RMS PIR [43] (Table 3). The amortized computation of HarmonyPIR1 is $15\times$ to $20\times$ faster than the

Table 2: Comparison of single-server HarmonyPIR with single-server WR PIR [48].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)	Amortized per query Comm. (KB)	Amortized per query Compute (ms)
WR PIR [48]	2^{19} 32-byte entries	0.06	16	17.1	13	65.9	29	82.6
HarmonyPIR0	16 MB in total	0.06	16	3.9	13.5	8.9	29.5	12.7
HarmonyPIR1		0.06	16	0.26	13.5	0.52	29.5	0.77
WR PIR [48]	2^{23} 32-byte entries	0.27	256	296.6	52	288.6	116	361
HarmonyPIR0	256 MB in total	0.27	256	76	54	39.8	118	58.4
HarmonyPIR1		0.27	256	3.9	54	2.3	118	3.3
WR PIR [48]	2^{27} 32-byte entries	1.06	4096	5228	208	1281	464	1600
HarmonyPIR0	4096 MB in total	1.06	4096	1421	216	190.9	472	277.6
HarmonyPIR1		1.06	4096	63.7	216	9.36	472	13.2

Table 3: Comparison of single-server HarmonyPIR with single-server RMS PIR [43].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)	Amortized per query Comm. (KB)	Amortized per query Compute (ms)
RMS PIR [43]	2^{19} 32-byte entries	5.625	16	1.95	1.1	0.12	1.5	0.17
HarmonyPIR0	16 MB in total	4.25	16	3.9	0.2	0.09	0.46	0.15
HarmonyPIR1		4.25	16	0.26	0.2	0.006	0.46	0.01
RMS PIR [43]	2^{23} 32-byte entries	22.5	256	31.2	4.1	0.5	5.7	0.7
HarmonyPIR0	256 MB in total	17	256	76	0.84	0.42	1.8	0.7
HarmonyPIR1		17	256	3.89	0.84	0.02	1.8	0.03
RMS PIR [43]	2^{27} 32-byte entries	90	4096	527.6	16.1	2.16	22.5	3
HarmonyPIR0	4096 MB in total	68	4096	1421	3.4	2.7	7.4	4
HarmonyPIR1		68	4096	75	3.4	0.1	7.4	0.2

amortized computation of RMS PIR, and the amortized computation of HarmonyPIR0 is comparable to the amortized computation of RMS PIR. Further, the amortized communication of HarmonyPIR is 3× better than the amortized communication of RMS PIR.

HarmonyPIR is more efficient than RMS PIR because it has communication and computation proportional to $\sqrt{N}/64$ (as $T = \sqrt{N}/64$), whereas RMS PIR has communication and computation proportional to $\sqrt{N}$.

Comparison with Balanced PIR [3] (Table 4). Balanced PIR has comparable amortized computation to HarmonyPIR1 and 16× better amortized computation than the amortized computation of HarmonyPIR0. The online and amortized communication of HarmonyPIR is 10× and 5× better than the online and amortized communication of Balanced PIR.

7.3.2 Two server HarmonyPIR

We compare two-server HarmonyPIR with RMS PIR [43] in Table 5. In this comparison, we assume that each offline phase, except the first, is executed in parallel with the preceding online period. Hence, the "offline" refers to the first offline phase, and "online" includes making a query *and* obtaining extra hints. For databases consisting of 32-byte entries, two-server HarmonyPIR1 has 5× better online compute cost and 1.5× better online communication cost even when we give two-server RMS PIR 1.75× larger client storage.

The other two baseline schemes did not specify two-server variants in the original papers [3, 48]. We compare with their natural two-server extensions in the appendix.

8 Related Work

Private information retrieval was introduced in the work of Chor et. al. [8]. In the standard setting without preprocessing or client states, PIR schemes face a fundamental efficiency barrier: the server must operate on every entry in the database for each query; otherwise, it leaks information about the client's query index [6].

Stateful PIR. Stateful PIR, which was introduced by Patel, Persiano, and Yeo [42], is a recent and promising approach to overcoming this barrier; in their work [42], the goal was to improve the server computation by a constant factor. The

Table 4: Comparison of single-server HarmonyPIR with single-server Balanced PIR [3].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)	Amortized per query Comm. (KB)	Amortized per query Compute (ms)
BalancedPIR [3]	2^{19} 32-byte entries	0.6	16	4.9	17	0.08	19.3	0.09
HarmonyPIR0	16 MB in total	0.53	16	3.9	1.7	0.8	3.7	1.2
HarmonyPIR1		0.53	16	0.26	1.7	0.048	3.7	0.08
BalancedPIR [3]	2^{23} 32-byte entries	2.5	256	85	68	0.37	77.1	0.35
HarmonyPIR0	256 MB in total	2.1	256	76	6.8	3.4	14.8	5.7
HarmonyPIR1		2.1	256	3.89	6.8	0.19	14.8	0.3
BalancedPIR [3]	2^{27} 32-byte entries	10	4096	1526	272	1.57	308.6	1.8
HarmonyPIR0	4096 MB in total	8.5	4096	1421	27	18.7	59	29.5
HarmonyPIR1		8.5	4096	66	27	0.85	59	1.35

Table 5: Comparison of two-server Harmony PIR with two-server RMS PIR [43].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)
RMS PIR [43]	2^{19} 32-byte entries	3.5	2.5	1	1.1	0.1
HarmonyPIR0	16 MB in total	2	2	3.9	0.6	0.3
HarmonyPIR1		2	2	0.26	0.6	0.02
RMS PIR [43]	2^{23} 32-byte entries	13.75	10	23.2	4.1	0.5
HarmonyPIR0	256 MB in total	8.25	8	76	2.25	1.68
HarmonyPIR1		8.25	8	4.1	2.25	0.08
RMS PIR [43]	2^{27} 32-byte entries	55	40	395	16	2.2
HarmonyPIR0	4096 MB in total	33	32	1421	9	7
HarmonyPIR1		33	32	68	9	0.3

recent works of Simple PIR [24] and Frodo PIR [11] also require $O(N)$ server computation with improved constant factors. Compared to the other schemes we will discuss next, their advantage is that the offline preprocessing needs to be executed only once, after which the client can make any number of queries.

Corrigan-Gibbs and Kogan [10] designed the first two-server stateful PIR scheme with sublinear amortized computation, which was later extended to the single-server setting [9]. However, their scheme requires κ parallel invocations of a PIR scheme, which is costly in practice. Building on the *partition-based hints* approach introduced in TreePIR [33], the PIR schemes of Zhou et al. [52] and Ren et al. [43] avoid parallel repetitions and achieve $O(\sqrt{N})$ communication and $O(\sqrt{N})$ computation, using $(\lambda\sqrt{N})$ client storage (asymptotics in this section are measured in words of size $\log N + w$). Improving the *permutation-based* hints introduced in [35], the PIR scheme of Wang and Ren [48] achieves $O(\sqrt{N})$ communication and $O(\sqrt{N})$ computation with only $O(\sqrt{N})$ client storage, meeting a lower bound by Yeo [50]. However, their scheme requires costly client computation. Balanced PIR [3] achieves efficient computation and client storage, but at the cost of worse communication.

While our focus is on practical schemes, we briefly mention a few theoretical works for completeness. A line of works [33, 51] improve communication asymptotically using theoretical cryptographic tools such as programmable PRFs, which do not have known practical instantiations. The PIR scheme of Ghoshal et al. [21] achieves $O(N^{1/4})$ communication but incurs significant blowups to client storage and offline server computation. Fisch et al. [19] give a method that achieves sublinear offline communication with a single server, but requires substantial server computational resources (hundreds of thousands of GPUs) to match the offline computation of other schemes.

Small-domain PRP. A line of work improves the construction of small-domain PRPs [26, 39, 40, 44, 46]. The construction of Hoang et al. [26] requires $O(\log N + \lambda)$ evaluations of a pseudorandom function (PRF) to construct a $(N/2)$-secure small-domain PRP. The start-of-the-art scheme by Morris and Rogaway [30] requires $O(\log^2 N + \lambda)$ evaluations of a PRF to construct a (N)-secure small-domain PRP, but it is overkill for our purpose.

Acknowledgements

We thank Dan Boneh for pointing us to Format Preserving Encryption. This research was funded in part by the National Science Foundation award CNS-2246386 and gifts from Google.

Ethical Considerations

To the best of our knowledge, there are no ethical considerations that arise from this work. This work develops new cryptographic techniques for stateful PIR, which reduces the privacy concerns of clients performing public data retrieval. All experiments are conducted on synthetic data.

Open Science

Our work uses the following artifact: https://anonymous.4open.science/r/HarmonyPIR-13DB/README.md. The artifact can be accessed by simply clicking on the link. It contains the implementation of HarmonyPIR, which we used to benchmark our work. A detailed description of executing the code can be found by accessing the link.

References

[1] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. Pir with compressed queries and amortized query processing. In *2018 IEEE symposium on security and privacy (SP)*, pages 962–979. IEEE, 2018.

[2] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, 2016.

[3] Pranav Shriram Arunachalaramanan and Ling Ren. Single-server stateful pir with verifiability and balanced efficiency. *Cryptology ePrint Archive*, 2025.

[4] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. Obliviad: Provably secure and practical online behavioral advertising. In *2012 IEEE Symposium on Security and Privacy*, pages 257–271. IEEE, 2012.

[5] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and J-F Raymond. Breaking the o (n/sup 1/(2k-1)/) barrier for information-theoretic private information retrieval. In *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.*, pages 261–270. IEEE, 2002.

[6] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *20th Annual International Cryptology Conference (CRYPTO)*, pages 55–73. Springer, 2000.

[7] Mihir Bellare, Viet Tung Hoang, and Stefano Tessaro. Message-recovery attacks on feistel-based format preserving encryption. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 444–455, 2016.

[8] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[9] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2022.

[10] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*, pages 44–75. Springer, 2020.

[11] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023.

[12] F Betül Durak and Serge Vaudenay. Breaking the ff3 format-preserving encryption standard over small domains. In *Annual international cryptology conference*, pages 679–707. Springer, 2017.

[13] F Betül Durak and Serge Vaudenay. Generic round-function-recovery attacks for feistel networks over small domains. In *International Conference on Applied Cryptography and Network Security*, pages 440–458. Springer, 2018.

[14] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *Journal of the ACM (JACM)*, 63(4):1–15, 2016.

[15] Morris Dworkin. Recommendation for block cipher modes of operation. *NIST special publication*, 800, 2025. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38Gr1.2pd.pdf.

[16] Morris J Dworkin, Elaine Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. Advanced encryption standard (aes). 2001.

[17] Ethereum Foundation. Ethereum privacy: Private information retrieval, 2025. URL: https://pse.dev/blog/ethereum-privacy-pir.

[18] Ethereum Foundation. Plinko pir tutorial, 2025. URL: https://vitalik.eth.limo/general/2025/11/25/plinko.html.

[19] Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. Thorpir: single server pir via homomorphic thorp shuffles. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1448–1462, 2024.

[20] Ashrujit Ghoshal, Mingxun Zhou, Bo Peng, and Elaine Shi. Zelda: Efficient multi-server preprocessing pir with unconditional security. *Cryptology ePrint Archive*, 2025.

[21] Ashrujit Ghoshal, Mingxun Zhou, and Elaine Shi. Efficient pre-processing pir without public-key cryptography. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 210–240. Springer, 2024.

[22] Matthew Green, Watson Ladd, and Ian 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, 2016.

[23] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, 2016.

[24] Alexandra Henzinger, Matthew M Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, 2023.

[25] Viet Tung Hoang, David Miller, and Ni Trieu. Attacks only get better: how to break ff3 on large domains. In *Annual International Conference on the theory and Applications of cryptographic techniques*, pages 85–116. Springer, 2019.

[26] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Annual Cryptology Conference*, pages 1–13. Springer, 2012.

[27] Viet Tung Hoang, Stefano Tessaro, and Ni Trieu. The curse of small domains: new attacks on format-preserving encryption. In *Annual International Cryptology Conference*, pages 221–251. Springer, 2018.

[28] Alexander Hoover, Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Plinko: single-server pir with efficient updates via invertible prfs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 3–33. Springer, 2025.

[29] Ari Juels. Targeted advertising... and privacy too. In *Cryptographers' Track at the RSA Conference*, pages 408–424. Springer, 2001.

[30] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. Optimal rate private information retrieval from homomorphic encryption. *Proc. Priv. Enhancing Technol.*, 2015.

[31] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX security symposium (USENIX Security 21)*, pages 875–892, 2021.

[32] Albert Hyukjae Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. 2015.

[33] Arthur Lazzaretti and Charalampos Papamanthou. Near-optimal private information retrieval with preprocessing. In *Theory of Cryptography Conference*, pages 406–435. Springer, 2023.

[34] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *Annual International Cryptology Conference*, pages 284–314. Springer, 2023.

[35] Arthur Lazzaretti and Charalampos Papamanthou. Single pass {Client-Preprocessing} private information retrieval. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5967–5984, 2024.

[36] Ming Luo, Feng-Hao Liu, and Han Wang. Faster fhe-based single-server private information retrieval. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1405–1419, 2024.

[37] Samir Jordan Menon and David J Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947. IEEE, 2022.

[38] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. Pir-tor: Scalable anonymous communication using private information retrieval. In *20th USENIX security symposium*, 2011.

[39] Ben Morris and Phillip Rogaway. Sometimes-recurse shuffle: almost-random permutations in logarithmic expected time. In *Annual International Conference on the*

Theory and Applications of Cryptographic Techniques, pages 311–326. Springer, 2014.

[40] Ben Morris, Phillip Rogaway, and Till Stegers. How to encipher messages on a small domain: deterministic encryption and the thorp shuffle. In *Annual International Cryptology Conference*, pages 286–302. Springer, 2009.

[41] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2292–2306, 2021.

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

[43] Ling Ren, Muhammad Haris Mughees, and I Sun. Simple and practical amortized sublinear private information retrieval using dummy subsets. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 1420–1433, 2024.

[44] Thomas Ristenpart and Scott Yilek. The mix-and-cut shuffle: small-domain encryption secure against n queries. In *Annual Cryptology Conference*, pages 392–409. Springer, 2013.

[45] Sacha Servan-Schreiber, Kyle Hogan, and Srinivas Devadas. Adveil: A private targeted advertising ecosystem. *Cryptology ePrint Archive*, 2021.

[46] Emil Stefanov and Elaine Shi. Fastprp: Fast pseudorandom permutations for small domains. *Cryptology ePrint Archive*, 2012.

[47] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, et al. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, 2019.

[48] Zhikun Wang and Ling Ren. Single-server client preprocessing pir with tight space-time trade-off. *Cryptology ePrint Archive*, 2024.

[49] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. *Journal of the ACM (JACM)*, 55(1):1–16, 2008.

[50] Kevin Yeo. Lower bounds for (batch) pir with private preprocessing. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 518–550. Springer, 2023.

[51] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 395–425. Springer, 2023.

[52] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: Extremely simple, single-server pir with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 55–55. IEEE Computer Society, 2023.

A Formal Definitions

A.1 Stateful PIR

Definition 1. *A stateful PIR scheme is* private *if there exists a probabilistic polynomial time simulator* Sim *such that for every probabilistic polynomial time adversary* $\mathcal{A}$ *acting as the server, every database* DB *of N entries, and every* $W = \mathrm{poly}(\lambda)$*, the views of* $\mathcal{A}$ *in the following two experiments are computationally indistinguishable.*

- Real World: *The client* $C(N,W)$ *interacts with* $\mathcal{A}(1^\lambda, \mathsf{DB}, W)$ *to execute a sequence of W queries. Each query index* q_i *is chosen adaptively by* $\mathcal{A}$ *based on its view.*
- Ideal World: *The simulator* $\mathsf{Sim}(N,W)$ *interacts with* $\mathcal{A}$ *in place of the client. Each query index* q_i *is chosen adaptively by* $\mathcal{A}$ *based on its view, but* Sim *is not given any query index.*

That is,

$$\Big|\Pr\big[\mathcal{A}^{C(N,W)}(1^\lambda, \mathsf{DB}, W) = 1\big] - \Pr\big[\mathcal{A}^{\mathsf{Sim}(N,W)}(1^\lambda, \mathsf{DB}, W) = 1\big]\Big| \leq \mathsf{negl}(\lambda).$$

A.2 Small-domain PRP

We formally define small-domain PRP. Given a mapping $P : [N'] \times \{0,1\}^\lambda \to [N']$, for any $k \in \{0,1\}^\lambda$, let $P(*,k)$, denoted as P_k, be a permutation and P_k^{-1} be its inverse permutation. Let $\Pi(N')$ be the set of all permutations of size N'. Then, P is a q-secure small-domain PRP if, for any probabilistic polynomial time adversary $\mathcal{A}$ that makes at most q queries to its oracle,

$$\Big|\Pr_{k \xleftarrow{\$} \{0,1\}^\lambda}[\mathcal{A}^{P_k, P_k^{-1}}(1^\lambda) = 1] - \Pr_{g \xleftarrow{\$} \Pi(N')}[\mathcal{A}^{g, g^{-1}}(1^\lambda) = 1]\Big| \leq \mathsf{negl}(\lambda)$$

where negl is a negligible function.

Algorithm 6 Relocation Data Structure DS

State N': Number of indices, T: Segment Size, P: Permutation, Hist: History Data Structure (Construction 1)

DS.Init(N')

Sample P to be a uniformly random permutation with domain $[2N]$
Hist.Init()
DS.$N' \leftarrow N'$

DS.Access(c) $\rightarrow v$

while $\mathsf{Hist}[P^{-1}(c)-N] \neq \perp$ **do**
  $c \leftarrow \mathsf{Hist}[P^{-1}(c)-N']$
end while
if $P^{-1}(c) < N'$ **then**
  return $P^{-1}(c)$
else
  return $\perp$
end if

DS.Locate(v) $\rightarrow c$

$c \leftarrow P(v)$
while $\mathsf{Hist}^{-1}[c] \neq \perp$ **do**
  $c \leftarrow P(N' + \mathsf{Hist}^{-1}[c])$
end while
return c

DS.Relocate(c)

Hist.Append(c)

B Relocation Data Structure (Wang and Ren [48])

Recall the relocation data structure DS from Section 3.3. We give the pseudocode of the Access and Locate functions in Algorithm 6 and describe them below.

Access. To Access the value in a cell c, we need to check if a value from another cell has relocated to cell c (may happen if c initially contains an empty value). Specifically, if $\mathsf{Hist}[P^{-1}(c)-N] \neq \perp$, the value in cell $\mathsf{Hist}[P^{-1}(c)-N]$ has relocated to the cell c, and we set c to $\mathsf{Hist}[P^{-1}(c)-N]$. This process is repeated until a cell c that *does not* contain a relocated value (in other words, $\mathsf{Hist}[P^{-1}(c)-N] = \perp$) is found. Finally, the initial value of cell c is returned (as that value got relocated to the input cell through a chain of relocations). Precisely, $P^{-1}(c)$ is returned if $P^{-1}(c) < N'$, or $\perp$ otherwise.

Locate. To Locate to the cell containing a value v, we first compute the cell containing v at initialization of DS, which is $c \leftarrow P(v)$. This cell could have relocated, in which case v would be in a different cell. Specifically, if $\mathsf{Hist}^{-1}[c] \neq \perp$, the value has relocated to cell $c \leftarrow P(N' + \mathsf{Hist}^{-1}[c])$. This process is repeated until a cell c that has not been relocated is found, and the cell c is returned.

Algorithm 7 Improved Hint Relocation in HarmonyPIR

Relocate

$m \leftarrow |S|$
RelocateSegment(s)
for $i \in [T]$ **do**
  $d_i \leftarrow \lfloor \mathsf{Locate}(N + mT + i)/T \rfloor$
  if $i \neq r$ **then**
    $\mathsf{H}[d_i] = \mathsf{H}[d_i] \oplus \mathsf{R}[i]$
  else
    $\mathsf{H}[d_r] = \mathsf{H}[d_r] \oplus A$
  end if
end for

C Optimizing Segment Relocation

We first modify the definition of Locate (Section 3.3) to support finding cells containing even empty values, not just values in $[N']$. The new definition is as follows:

- $\mathsf{Locate}(\mathsf{v}) \rightarrow c$: Given a value $v \in [2N']$, Locate returns the cell $c \in [2N'] \backslash C$ that contains v. Locate does not update C.

In contrast, the definition of Wang and Ren [48] only allows v to be in $[N']$. We remark that this new definition does not modify the construction of Locate from Wang and Ren [48], and in fact better captures what their construction achieves.

Using this modified definition, we can now improve the Relocate step in HarmonyPIR (Algorithm 3). Let $m = |S|$ be the number of relocated segments of DS$'$. To relocate the values in segment s, the client executes DS$'$.RelocateSegment(s) as in the original scheme. The hint parities now need to be updated according to these relocations; that is, the j-th hint parity matches the combined parity of database entries at values in segment j of DS$'$. To perform these updates, the client first locates the cells that contain the values in segment s after the segment relocation. Instead of invoking Locate on the values in segment s (as in the original scheme), the client invokes Locate on the empty values to which they have relocated. Recall that during the j-th cell relocation, the value in a cell of DS$'$ is relocated to the cell containing the value $N + j - 1$. Then, the value in the i-th cell of segment s has relocated to the cell $\mathsf{Locate}(N + m \cdot T + i)$, which is in segment $d_i = \lfloor \mathsf{Locate}(N + m \cdot T + i)/T \rfloor$. The client updates hints as follows: for $i \in [T] \backslash r$, $\mathsf{H}[d_i] = \mathsf{H}[d_i] \oplus \mathsf{R}[i]$ and $\mathsf{H}[d_r] = \mathsf{H}[d_r] \oplus A$. This method of finding the cells containing the relocated segment indices achieves a constant-factor improvement in client computation.

D Privacy of HarmonyPIR

Lemma 5 (Full proof of Lemma 4)**.** *For any sequence of $M/2$ queries to indices $q_0, q_1, \ldots, q_{M/2-1}$, each chosen adaptively*

based on the adversary's view, the request Q_i for the i-th query follows the distribution $\mathcal{P}_T(U_{N,N-T\cdot i})$, conditioned on the prior requests $\mathsf{Q}_0,\ldots,\mathsf{Q}_{i-1}$ and the query indices $q_0,\ldots,q_i$.

Notation. Let $n_i = 2N - T\cdot i$ be the number of cells of DS' that have not been relocated before the i-th query.

Consider a list of values $v = (v[0],\ldots,v[T-1])$ with $v[j] \in [N]\cup\{\perp\}$, $\mu_j(v)$, where each value is drawn from the set $U_{N,N-T\cdot i}$ without replacement. Then, for a random variable X following the distribution $\mathcal{P}_T(U_{N,N-T\cdot i})$, we compute $\Pr[X = v]$.

Consider the following definition:

$$\mu_j(v) = \begin{cases} 1, & v[j]\in[N], \\ (N - T\cdot i) - |\{\, j' < j : v[j'] = \perp\,\}|, & v[j] = \perp, \end{cases}$$

The above defines a quantity $\mu_j(v)$, that is i) 1, when $v[j]$ is an index, and ii) the difference between the total number of empty cells in DS' and the number of empty cells in v before position j, when $v[j]$ is $\perp$. The utility of defining this quantity will become apparent in the next step. Suppose we want to compute $\Pr[X = v]$:

$$\begin{aligned}\Pr[X = v] &= \Pr[X[0] = v[0]]\cdot\Pr[X[1] = v[1] \mid X[0] = v[0]] \\ &\quad\cdot\Pr[X[2] = v[2] \mid X[0] = v[0], X[1] = v[1]]\cdot \\ &\quad\ldots\Pr[X[T-1] = v[T-1] \mid X[0] = v[0],\ldots]] \\ &= \frac{\mu_0(v)}{n_i}\cdot\frac{\mu_1(v)}{n_i-1}\cdots\frac{\mu_T(v)}{n_i-T+1}\end{aligned} \tag{1}$$

If we define $\Pi(v) = \prod_{j\in[T]}\mu_j(v)$ and $(a)_T = a(a-1)\cdots(a-T+1)$. Then,

$$\Pr[X = v] = \frac{\Pi(v)}{(n_i)_T}.$$

Proof. Suppose $i-1$ queries have already been completed. From the randomness invariance of DS', the values in the non-relocated cells of DS' follow $\mathcal{P}(U_{N,N-T\cdot i})$ conditioned on prior requests and query indices. We now compute the distribution of Q_i conditioned on $C = [\mathsf{Q}_0,\mathsf{Q}_1,\ldots,\mathsf{Q}_{i-1},q_0,q_1,\ldots,q_{i-1},q_i]$. Consider any list v of T values drawn without replacement from the set $U_{N,N-T\cdot i}$ (v essentially denotes any possible request). Recall the algorithm used in request construction. We compute the probability that Q_i takes the value v in two cases.

Case 1: q_i does not appear in v. For this case to occur: i) χ must be tails, which occurs with probability $\frac{n_i-T}{n_i}$ ii) the values in v must match a random selection of values without replacement from the remaining n_i-1 values of DS' (excluding q_i), which occurs with probability $\frac{\Pi(v)}{(n_i-1)_T}$. Then:

$$\Pr[\mathsf{Q}_i = v \mid C] = \frac{n_i-T}{n_i}\cdot\frac{\Pi(v)}{(n_i-1)_T} = \frac{\Pi(v)}{(n_i)_T}.$$

Case 2: q_i appears in v at position ρ. For this case to occur: i) r must match ρ, which occurs with probability $1/T$, ii) χ must be heads, which occurs with probability T/n_i, and iii) the remaining $T-1$ cells of s must match the remaining values of v, which occurs with probability $\Pi(v)/(n_i-1)_{T-1}$. Then:

$$\Pr[\mathsf{Q}_i = v \mid C] = \frac{1}{T}\cdot\frac{T}{n_i}\cdot\frac{\Pi(v)}{(n_i-1)_{T-1}} = \frac{\Pi(v)}{(n_i)_T}.$$

In both cases, Q_i conditioned on C follows the distribution $\mathcal{P}_T(U_{N,N-T\cdot i})$, completing the proof. □

E Efficiency of single-server HarmonyPIR

Theorem 6 (Proof of Theorem 4)**.** *Consider any database consisting of N entries each of size w bits, and any given parameter T. Over at least $M/2 = N/T$ queries,* HarmonyPIR *has:*

- *client storage of $O(Mw + M\log N)$ bits*
- *amortized communication of $O(Tw + T\log N)$ bits*
- *amortized client computation of $O(T)$ small-domain PRP evaluations and $O(T)$ w-bit XORs.*
- *amortized server computation of T lookups of w-bit entries.*

Proof. **Client storage.** The client stores a Hist' data structure that can contain $M/2$ segments, which requires $O(M\log N)$ bits, and M hint parities, which requires $O(Mw)$ bits.

Amortized communication. The client streams N w-bit entries in the offline phase; the offline communication gets amortized over $M/2$ queries, make it $O(Tw)$ per query. For each query in the online phase, the client sends T values each of $\log N$ bits (the request), and gets T entries of each of w bits (the response).

Amortized client computation. In the offline phase, the client performs $O(N)$ Locate operations and $O(N)$ XORs of w-bit entries. Each Locate and Access operation involves $O(1)$ small-domain PRP evaluations in expectation [48, Lemmas A.1 and A.2]. Then, the amortized offline client computation over $M/2$ queries involves $O(T)$ small-domain PRP evaluations and $O(T)$ XORs of w-bit entries.

For each query in the online phase, the client performs $2T$ Locate operations, 1 Access operation, and 1 RelocateSegment operation; these take $O(T)$ small-domain PRP evaluations in expectation. Further, the client performs T XOR operations on w-bit entries.

Amortized server computation. During the offline phase, the server looks up all N entries in the database, which, when amortized over $M/2$ queries, gives T lookups per query. For each query in the online phase, the server looks up T entries in the database. □

F Instantiating the random permutation in HarmonyPIR using a small-domain PRP

Lemma 6. *Consider the real-world interaction of stateful PIR from Section 3.1, restricted to one offline phase and the $M/2$ queries that follow it. For any database* DB *which is a list of N entries, let $\mathcal{C}(f, f^{-1}, N, M/2)$ denote the* HarmonyPIR *client whose relocation data structure* DS′ *is instantiated with the permutation f of size $2N$, and let the adversary $\mathcal{A}$ be the server. If P is an N-secure small-domain PRP of size $2N$, then for every probabilistic polynomial time $\mathcal{A}$,*

$$\Big| \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} \left[\mathcal{A}^{\mathcal{C}(P_k, P_k^{-1}, N, M/2)}(1^\lambda, \mathsf{DB}, M/2) = 1\right] - \Pr_{g \xleftarrow{\$} \Pi(2N)} \left[\mathcal{A}^{\mathcal{C}(g, g^{-1}, N, M/2)}(1^\lambda, \mathsf{DB}, M/2) = 1\right] \Big| \leq \mathsf{negl}(\lambda).$$

Proof. If $\mathcal{A}$ is the offline server of two-server HarmonyPIR, it is straightforward to see that the lemma holds. We now prove the lemma when $\mathcal{A}$ is the online server of two-server HarmonyPIR or the only server in single-server HarmonyPIR.

We show that a distinguisher $\mathcal{A}$ between $\mathcal{C}(P_k, P_k^{-1})$ and $\mathcal{C}(g, g^{-1})$ yields a distinguisher $\mathcal{D}$ between oracle access to (P_k, P_k^{-1}) and to (g, g^{-1}) making at most N queries, contradicting the security of the small-domain PRP (Section 3.2).

The challenger Ch samples either $(f, f^{-1}) = (P_k, P_k^{-1})$ or $(f, f^{-1}) = (g, g^{-1})$ and grants $\mathcal{D}$ oracle access to it. $\mathcal{D}$ plays the role of the client towards $\mathcal{A}$ and outputs whatever $\mathcal{A}$ outputs.

Offline phase. (Only in the single-server case) $\mathcal{D}$ simply streams the database from $\mathcal{A}$. Note that it does not need to compute hint parities as they do not influence client requests.

Online phase. To simulate the online phase, $\mathcal{D}$ keeps track of all the values in relocated segments in a list C $\mathcal{A}$ selects query indices and sends them to $\mathcal{D}$, and $\mathcal{D}$ must respond with requests. To construct requests, $\mathcal{D}$ must first find the segment containing the query index, and access all the values in the segment. Given a query index q_j, $\mathcal{D}$ finds the cell containing q_j as follows:

- If q_j is not a value in C, return $f(q_j)$ by making a call to Ch.
- Otherwise, suppose q_j be at position k in C; then, it would have relocated to the cell containing the value $N+k-1$. Recurse with the value $N+k-1$.

Having found the cell c containing q_j, $\mathcal{D}$ needs to find values in all other cells of the segment $s = \lfloor c/T \rfloor$. To find the value in a cell c' of s:

1. Compute $f^{-1}(c')$ by making a call to Ch.
2. If $f^{-1}(c') \in [N]$ return $f^{-1}(c')$.
3. Otherwise, $f^{-1}(c') \in [2N] \backslash [N]$, and some other value could have relocated to it.
   - If $f^{-1}(c') - (N-1) > |C|$, return $\perp$.
   - Otherwise, find the value in position $f^{-1}(c') - (N)$ of C and return to step (2) with $f^{-1}(c') = C[f^{-1}(c') - (N)]$.

Given all the values in the segment s, it is straightforward to see how the request can be constructed without invoking any permutation operations. $\mathcal{D}$ then sends this request to $\mathcal{A}$.

Observe that to compute the request, $\mathcal{D}$ only has to make T oracle queries. Then, to compute all requests in an online phase, $\mathcal{D}$ only has to make $MT/2 = N$ oracle queries, which it is allowed to make. Finally, $\mathcal{D}$ outputs to Ch whatever $\mathcal{A}$ outputs. Then:

$$\left| \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} [\mathcal{A}^{\mathcal{C}(P_k, P_k^{-1})}(1^\lambda)] - \Pr_{g \xleftarrow{\$} \Pi(2N)} [\mathcal{A}^{\mathcal{C}(g, g^{-1})}(1^\lambda)] \right| = \left| \Pr_{k \xleftarrow{\$} \{0,1\}^\lambda} [\mathcal{D}^{P_k, P_k^{-1}}(1^\lambda)] - \Pr_{g \xleftarrow{\$} \Pi(2N)} [\mathcal{D}^{g, g^{-1}}(1^\lambda)] \right| \leq \mathsf{negl}(\lambda)$$

, where the last inequality follows from the security of the small-domain PRP.

□

Lemma 7. *Consider the real-world interaction of stateful PIR from Section 3.1. For any database* DB *which is a list of N entries, and any sequence of $W = \mathsf{poly}(\lambda)$ queries, let $\mathcal{C}_{\mathsf{PRP}}(N, W)$ denote the* HarmonyPIR *client whose relocation data structure* DS′ *is instantiated with small-domain PRPs of size $2N$ and let $\mathcal{C}_{\mathsf{RP}}(N, W)$ denote the* HarmonyPIR *client whose relocation data structure* DS′ *is instantiated with random permutations of size $2N$. Then for every probabilistic polynomial time $\mathcal{A}$ that is the server,*

$$\Big| \Pr\left[\mathcal{A}^{\mathcal{C}_{\mathsf{PRP}}(N,W)}(1^\lambda, \mathsf{DB}, W) = 1\right] - \Pr\left[\mathcal{A}^{\mathcal{C}_{\mathsf{RP}}(N,W)}(1^\lambda, \mathsf{DB}, W) = 1\right] \Big| \leq \mathsf{negl}(\lambda).$$

Proof. As in the previous lemma, we consider the case where $\mathcal{A}$ is the online server of two-server HarmonyPIR or the only server in single-server HarmonyPIR. When $\mathcal{A}$ is the offline server, the lemma is straightforward.

We use a standard hybrid argument combined with Lemma 6. The number of online phases executed is $E = 2W/M$. For $i \in [0, E]$, define hybrid $\mathcal{H}_i$ to be an execution in which the client uses a randomly sampled PRP for the first i online phases and a random permutation for the remaining offline phases. Observe that our goal is to bound the probability that an adversary distinguishes $\mathcal{H}_E$ and $\mathcal{H}_0$. Consider any two consecutive hybrids $\mathcal{H}_j$ and $\mathcal{H}_{j+1}$. The probability that an adversary distinguishes these two hybrids is at most $\mathsf{negl}(\lambda)$, as otherwise, we can construct a distinguisher to Lemma 6.

Specifically, this can be done by instantiating the $i+1$-th permutation using the permutation provided to the client in Lemma 6.

Then, by applying the triangle inequality we get that the probability that the adversary distinguishes $\mathcal{H}_E$ and $\mathcal{H}_0$ is at most $E\mathsf{negl}(\lambda)$, which gives the lemma. □

G Two server HarmonyPIR

G.1 Efficiency

Theorem 7. *(Efficiency) Consider any database consisting of N entries each of size w bits, and any given parameter T. Over at least $M/2 = N/T$ queries, two server* HarmonyPIR *has:*

- *client storage of $O(Mw + M\log N)$ bits.*
- *amortized communication of $O(Tw + T\log N)$ bits.*
- *amortized client computation of $O(T)$ evaluations of a small-domain PRP and $O(T)$ XORs of w bit entries.*
- *amortized offline server computation is $O(T)$ evaluations of a small-domain PRP and $O(T)$ XORs of w bit entries.*
- *amortized online server computation of T lookups to entries of size w bits in a list of N entries.*

Proof.

Client Storage. The client stores a Hist' data structure with at most $M/2$ segments, which requires $O(M\log N)$ bits, and M hint parities, which requires $O(Mw)$ bits.

Amortized Communication. The client obtains M w-bit hint parities from the offline server; the offline communication gets amortized over $M/2$ queries, make it $O(w)$ per query. For each query in the online phase, the client sends T values each of $\log N$ bits (the request), and gets T entries of each of w bits (the response).

Amortized Client Computation. In the offline phase, the client simply obtains M w-bit hint parities from the offline server.

For each query in the online phase, the client performs $2T$ Locate operations, 1 Access operation, and 1 $\mathsf{RelocateSegment}$ operation; these take $O(T)$ small-domain PRP evaluations in expectation. Further, the client performs T XOR operations on w-bit entries.

Amortized Offline Server Computation. To sample hints, the offline server performs $O(N)$ Locate operations and $O(N)$ XORs of w-bit entries.

Amortized Offline Server Computation. The online server looks up T entries in the database for each query. □

G.2 Comparison of two-server HarmonyPIR

We now compare two-server HarmonyPIR with two-server WR PIR [48] and Balanced PIR [3].

Comparison with two-server WR PIR [48] (Table 6). When compared to WR PIR, two-server HarmonyPIR1 has $125\times$ better online computation cost and comparable online communication cost.

Comparison with two-server Balanced PIR [3] (Table 7). When compared to Balanced PIR, two-server HarmonyPIR1 has $5\times$ better online communication cost and comparable online computation cost.

Table 6: Comparison of two-server Harmony PIR with two-server WR PIR [48].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)
WR PIR [48]	2^{19} 32-byte entries	0.06	0.06	17.1	13	88
HarmonyPIR0	16 MB in total	0.06	0.06	3.9	13.5	10.5
HarmonyPIR1	16 MB in total	0.06	0.06	0.26	13.5	0.7
WR PIR [48]	2^{23} 32-byte entries	0.25	0.25	296	52	325
HarmonyPIR0	256 MB in total	0.25	0.25	76	54	54
HarmonyPIR1	256 MB in total	0.25	0.25	4.1	54	2.6
WR PIR [48]	2^{27} 32-byte entries	1	1	5228	208	869
HarmonyPIR0	4096 MB in total	1	1	1421	216	158
HarmonyPIR1	4096 MB in total	1	1	68	216	6.9

Table 7: Comparison of two-server Harmony PIR with two-server Balanced PIR [3].

	Database Parameters	Client Storage (MB)	Offline Comm. (MB)	Offline Compute (s)	Online Comm. (KB)	Online Compute (ms)
BalancedPIR [3]	2^{19} 32-byte entries	0.31	0.38	17.3	17	0.28
HarmonyPIR0	16 MB in total	0.25	0.25	3.9	3.3	3
HarmonyPIR1	16 MB in total	0.25	0.25	0.25	3.3	0.2
BalancedPIR [3]	2^{23} 32-byte entries	1.5	1.3	288	68	1.2
HarmonyPIR0	256 MB in total	1	1	76	13.5	17
HarmonyPIR1	256 MB in total	1	1	4.2	13.5	0.81
BalancedPIR [3]	2^{27} 32-byte entries	6	5.1	4615	272	2.8
HarmonyPIR0	4096 MB in total	4	4	1421	54	50
HarmonyPIR1	4096 MB in total	4	4	67	54	2.2