Efficient Pre-processing PIR Without Public-Key Cryptography

Ashrujit Ghoshal Mingxun Zhou Elaine Shi*

Carnegie Mellon University

Abstract

Classically, Private Information Retrieval (PIR) was studied in a setting without any pre-processing. In this setting, it is well-known that 1) public-key cryptography is necessary to achieve non-trivial (i.e., sublinear) communication efficiency in the single-server setting, and 2) the total server computation per query must be linear in the size of the database, no matter in the single-server or multi-server setting. Recent works have shown that both of these barriers can be overcome if we are willing to introduce a pre-processing phase. In particular, a recent work called Piano showed that using only one-way functions, one can construct a single-server preprocessing PIR with $\widetilde{O}(\sqrt{n})$ bandwidth and computation per query, assuming $\widetilde{O}(\sqrt{n})$ client storage. For the two-server setting, the state-of-the-art is defined by two incomparable results. First, Piano immediately implies a scheme in the two-server setting with the same performance bounds as stated above. Moreover, Beimel et al. showed a two-server scheme with $O(n^{1/3})$ bandwidth and $O(n/\log^2 n)$ computation per query, and one with $O(n^{1/2+\epsilon})$ cost both in bandwidth and computation — both schemes provide information theoretic security.

In this paper, we show that assuming the existence of one-way functions, we can construct a two-server preprocessing PIR scheme with $\widetilde{O}(n^{1/4})$ bandwidth and $\widetilde{O}(n^{1/2})$ computation per query, while requiring only $\widetilde{O}(n^{1/2})$ client storage. We also construct a new single-server preprocessing PIR scheme with $\widetilde{O}(n^{1/4})$ *online* bandwidth and $\widetilde{O}(n^{1/2})$ *offline* bandwidth and *computation* per query, also requiring $\widetilde{O}(n^{1/2})$ client storage. Specifically, the online bandwidth is the bandwidth required for the client to obtain an answer, and the offline bandwidth can be viewed as background maintenance work amortized to each query. Our new constructions not only advance the theoretical understanding of preprocessing PIR, but are also concretely efficient because the only cryptography needed is pseudorandom functions.

1 Introduction

Private Information Retrieval (PIR), originally formulated by Chor, Goldreich, Kushilevitz, and Sudan [CGKS95], studies the following important problem. Imagine that a server holds a public database denoted $\mathsf{DB} \in \{0,1\}^n$. A client with small local storage wants to query the database, while hiding its queries from the server. PIR has wide applications in practice. For example, it enables private contact discovery [DRRT18, Con], privacy-preserving light-weight clients for cryptocurrencies, private DNS queries [SACM21, ZPSZ24, Fea], private web search [HDCG+23], and so on.

Classical PIR without pre-processing. A naïve solution for PIR is to have the client linearly scan through the entire database for each query. Unfortunately, this would incur linear bandwidth. A series of works spanning over two decades [CGKS95, Cha04, GR05, CMS99, CG97, KO97, Lip09,

*Author ordering is randomized.

OS07, Gas04, BFG03, SC07, OG11, MCG+08, MG07] starting with Chor et al. [CGKS95] showed how to construct PIR with non-trivial bandwidth. Specifically, in the single-server setting, it is well-known that with various cryptographic assumptions (e.g., Φ-hiding, LWE, Damgård-Jurik, DDH, QR), we can achieve $\widetilde{O}_\lambda(1)$ bandwidth per query [CMS99, HHCG+22, MW22, DGI+19] where $\widetilde{O}_\lambda(\cdot)$ hides polylogarithmic factors and the dependence on the security parameter λ. In the two-server setting, Dvir and Gopi [DG16] showed that information-theoretic PIR is possible with $n^{O(\sqrt{\log\log n/\log n})}$ bandwidth per query. All the aforementioned works studied PIR in the *classical setting without preprocessing*. More specifically, the classical setting assumes that the server stores only the original database, and need not store per-client state. Unfortunately, the classical setting suffers from the following inherent limitations.

1. First, Beimel, Ishai, and Malkin [BIM00] proved that any classical PIR scheme without preprocessing must suffer from linear (in n) server computation per query. Intuitively, if there is any location that the server does not look at during a query, then the client cannot be asking for that location.

2. Second, in a single-server setting, it is known that any PIR scheme with non-trivial (i.e., sublinear) bandwidth would imply oblivious transfer [DCMO00], i.e., some form of public-key cryptography is needed.

Pre-processing sublinear PIR without public-key cryptography. To get around the aforementioned barriers, earlier works have suggested a pre-processing model. Specifically, we consider a model with a one-time pre-processing phase upfront, followed by an *unbounded* number of queries. The pre-processing model was first proposed by Beimel, Ishai, and Malkin [BIM00] and Corrigan-Gibbs and Kogan [CK20].

In a pre-processing model, we focus on exploring the efficiency of PIR with *sublinear computation*, and *without the use of public-key cryptography* — to get around the aforementioned barriers, In particular, the requirement of sublinear computation is especially important in practical scenarios where the database is large — for example, in a private DNS application, the database can be several hundred Gigabytes. Further, we restrict ourselves to Minicrypt (i.e., allowing PRFs but not using any public-key cryptography) — this is not only motivated by theoretical interest, but also the promise of concretely faster constructions since modern processors have hardware acceleration for AES operations.

Earlier works showed that under a global, server-side preprocessing, one can overcome the linear computation barrier [BIM00, CK20, CHK22, ZLTS23, LP23, LP22, SACM21, LMW23, ZPSZ24]; and further, assuming a client-specific preprocessing, we can overcome both of the aforementioned barriers [ZPSZ24, MIR23]. So far, we know the following pre-processing PIR schemes which enjoy sublinear pre-processing without the use of public-key cryptography. Beimel et al. [BIM00] showed by leveraging a global server-side pre-processing, it is possible to construct an information theoretic 2-server PIR with $O(n^{1/3})$ bandwidth, $O(n/\log^2 n)$ computation, while consuming $O(n^2)$ server storage. The same work also showed an incomparable information-theoretic scheme with $O(n^{1/2+\epsilon})$ cost in both computation and bandwidth, while incurring $n^{1+\epsilon'}$ server storage where ϵ' is a constant dependent on ϵ.

Under a client-specific pre-processing model, the recent work Piano [ZPSZ24] and the subsequent work of Mughees et al. [MIR23] constructed single-server PIR schemes with $\widetilde{O}(\sqrt{n})$ bandwidth and server computation, $\widetilde{O}_\lambda(\sqrt{n})$ client computation per query, while consuming $\widetilde{O}_\lambda(\sqrt{n})$ client storage — interestingly, their schemes relied only on pseudorandom functions (PRFs) and do not make use of public-key cryptography. Since classical (single-server) PIR implies oblivious transfer [DCMO00],

Table 1: **Comparison of single-server and two-server pre-processing PIR schemes**. Any single-server scheme immediately implies a two-server result with the same performance bounds. n is the size of the database and m is the number of clients. The server space counts only the extra storage needed on top of storing the original database.

Scheme	Assumpt.	Compute	Comm.	Space		# servers	Concrete eff.
				client	server		
With public-key cryptography							
[CHK22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
[ZLTS23, LP22]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(m \cdot n)^*$	1	✗
[LMW23]	Ring-LWE	$\mathsf{poly}((\log n)^{1/\epsilon})$	$\mathsf{poly}((\log n)^{1/\epsilon})$	0	$n^{1+\epsilon}$	1	✗
[SACM21]	LWE	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✗
[LP23]	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
Our work	Various	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$ offline $\widetilde{O}_\lambda(1)$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
Without public-key cryptography							
[BIM00]	None	$O(n/\log^2 n)$	$O(n^{1/3})$	0	$O(n^2)$	2	✗
[BIM00]	None	$O(n^{1/2+\epsilon})$	$O(n^{1/2+\epsilon})$	0	$O(n^{1+\epsilon'})^{**}$	2	✗
[ZPSZ24, MIR23]	OWF	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓
Our work	OWF	$O_\lambda(\sqrt{n})$	$O_\lambda(n^{1/4})$	$\widetilde{O}_\lambda(\sqrt{n})$	0	2	✓
Our work	OWF	$O_\lambda(\sqrt{n})$	$O(\sqrt{n})$ offline $O_\lambda(n^{1/4})$ online	$\widetilde{O}_\lambda(\sqrt{n})$	0	1	✓

$*$: In the unbounded query setting, some earlier works [ZLTS23, LP22, CHK22] require that the next pre-processing is persistently piggybacked on the current window of $O(\sqrt{n})$ operations, and the pre-processing consumes $O_\lambda(n)$ server space per client to evaluate under FHE an $\widetilde{O}(n)$-sized circuit containing a sorting network.

$**$: $\epsilon' > 0$ depends on ϵ.

Piano [ZPSZ24] and Mughees et al. [MIR23]'s results also separate pre-processing PIR from classical PIR (in the single-server setting) from a theoretical perspective.

1.1 Our Results

We show new results that improve the state of our understanding regarding pre-processing PIR. In all of our constructions, the server only needs to store the original database and need not store any per-client state.

Main result 1. First, we construct a two-server pre-processing PIR scheme with asymptotically better bandwidth than prior work, relying only on the existence of PRFs (which is equivalent to the existence of one-way functions). Our result is stated in the following theorem.

Theorem 1.1 (Two-server pre-processing PIR with improved bandwidth)**.** *Assume the existence of one-way functions. There exists a two-server pre-processing PIR scheme with $O_\lambda(n^{1/4})$ bandwidth and $O_\lambda(n^{1/2})$ computation per query, while incurring $\widetilde{O}_\lambda(n^{1/2})$ client storage.*

In comparison with the prior work of Beimel et al. [BIM00], our Theorem 1.1 achieves significant asymptotic improvements in both bandwidth, computation, and server-side storage. On the other hand, we need to assume one-way functions whereas Beimel et al. [BIM00]'s schemes are information theoretic; further, we additionally require $\widetilde{O}(\sqrt{n})$ space on each client. However, our construction that gives Theorem 1.1 is simple and concretely efficient, which is another advantage over Beimel et al.

Main result 2. Second, we construct a new pre-processing PIR scheme in the single-server setting that improves the *online* bandwidth in comparison with the state-of-the-art. In this theorem, we differentiate between online bandwidth and offline bandwidth. The online bandwidth is the bandwidth necessary for the client to obtain an answer to its query, so it matters to the response time of the client. The offline bandwidth is the cost of background maintenance work amortized to each query, and is not on the critical path of the client's response time.

Theorem 1.2 (Single-server preprocessing PIR with improved online bandwidth)**.** *Assume the existence of one-way functions. There exists a single-server pre-processing PIR scheme with $O_\lambda(n^{1/4})$ online bandwidth, $O(n^{1/2})$ offline bandwidth, $O_\lambda(n^{1/2})$ server computation and $\widetilde{O}_\lambda(n^{1/2})$ client computation per query, while incurring $\widetilde{O}(n^{1/2})$ client storage.*

In comparison with the state-of-the-art scheme Piano, Theorem 1.2 improves the online bandwidth cost from $\widetilde{O}(\sqrt{n})$ to $\widetilde{O}_\lambda(n^{1/4})$, while keeping all other costs the same. Moreover, recall that earlier works [CK20, CHK22], proved the time-space product lower bound, showing that the product of the client space and the online server time has to be at least linear in n. In this sense, Theorem 1.2 is tight (upto polylogarithmic factors) in terms of this time-space product.

Additional results. While our main results focus on constructions in Minicrypt, if we are willing to assume classical PIR with $\widetilde{O}_\lambda(1)$ bandwidth (which is known from various assumptions such as LWE, Φ-hiding, Damgård-Jurik, DDH, QR) [CMS99, HHCG+22, MW22, DGI+19], our techniques would then give rise to a concretely efficient single-server PIR scheme with $\widetilde{O}_\lambda(1)$ *online* bandwidth, $\widetilde{O}(\sqrt{n})$ offline bandwidth and computation per query, consuming $\widetilde{O}_\lambda(\sqrt{n})$ client storage. In comparison, although the earlier works by Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] claim to achieve polylogarithmic (online and offline) bandwidth, their schemes suffer from a significant drawback, that is, the server would have to persistently store at least n amount of state per client! Specifically, Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] require the pre-processing phase of the next epoch be piggybacked on the queries of the current epoch; however, their pre-processing phase requires that the server allocate at least n amount of space per client, to perform homomorphic evaluation of a circuit which is super-linear in size. So far, in the unbounded query setting, it is not known how to get polylogarithmic overall bandwidth (including offline and online) per query *under any assumption*, assuming that the server stores only the original database. We state this additional result in the following theorem.

Theorem 1.3. *Assume the existence of a classical single-server PIR scheme (i.e., without pre-processing) that enjoys $\widetilde{O}_\lambda(1)$ bandwidth per query. Then, there exists a single-server pre-processing PIR scheme with $\widetilde{O}_\lambda(1)$ online bandwidth, $\widetilde{O}_\lambda(\sqrt{n})$ computation, $\widetilde{O}(\sqrt{n})$ offline bandwidth, and requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage.*

1.2 Technical Highlights

The earlier work of Shi et al. [SACM21] showed that assuming the existence of a privately puncturable PRF [BLW17, BKM17, CC17, BTVW17], one can construct an efficient 2-server pre-processing PIR scheme with $\widetilde{O}_\lambda(\sqrt{n})$ computation per query and requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage. Further, the communication per query is only polylogarithmically larger than the size of a punctured key, which can be as small as $\widetilde{O}_\lambda(1)$ using known constructions [BLW17, BKM17, CC17, BTVW17]. Unfortunately, the only known techniques for constructing a privately puncturable PRF [BLW17, BKM17, CC17, BTVW17] requires two layers of fully homomorphic encryption, and it is not known whether privately puncturable PRFs can be built from only one-way functions. The elegant TreePIR work of Lazzaretti and Papamanthou [LP23] showed how to replace the privately puncturable PRF

with a weaker primitive called a "weak privately puncturable PRF". Unfortunately, their approach relies on recursing on a classical PIR scheme for a $\sqrt{n}$-sized database, and because this database is *dynamically constructed* during the scheme, it is not possible to pre-process it. Therefore, Lazzaretti and Papamanthou [LP23]'s techniques fundamentally also require public-key cryptography.

Privately programmable pseudorandom set with list decoding. Our main contribution is to come up with a new abstraction called a *Privately Programmable Pseudorandom Set with List Decoding* (PPPS). Given a PPPS key sk, we can expand the key sk to a pseudorandom set denoted $\mathsf{Set}(\mathsf{sk})$ of size $\sqrt{n}$. Further, deciding whether any element in $\{0, 1, \ldots, n-1\}$ is in the set takes only constant time. Importantly, we can call a $\mathsf{Program}$ algorithm to program sk such that the new set is almost the same as the original $\mathsf{Set}(\mathsf{sk})$, except that the one element in the set is now changed to another specified element. The programmed key does not leak information about which element is programmed.

Our notion of PPPS is otherwise very similar to the earlier work of Zhou et al. [ZLTS23], except that we make a relaxation on the correctness when decoding a programmed key — this relaxation is the crucial reason why we can construct it from only one-way functions, whereas Zhou et al. [ZLTS23]'s construction relies on LWE. More specifically, we do not require that one can correctly recover the programmed set given a programmed key sk'. Instead, we allow *list-decoding*, that is, given a programmed key sk', decoding outputs a list of candidate sets, among which one must be the true programmed set. Moreover, the list-decoding of our PPPS construction is structured, allowing succinct representation and efficient representation.

Using only one-way functions, we construct a PPPS scheme with list decoding for a pseudorandom set of size $\sqrt{n}$, where the programmed key has size $O_\lambda(n^{1/4})$.

Using such a PPPS scheme, we show how to get a two-server scheme with $O_\lambda(n^{1/4})$ communication and $\widetilde{O}_\lambda(n^{1/2})$ computation per query, using only $\widetilde{O}_\lambda(n^{1/2})$ client space (Theorem 1.1). Unlike TreePIR [LP23], our scheme need not recurse on a classical PIR scheme, and thus we do not need public-key operations.

A new broken hint technique. To get our single-server scheme (Theorem 1.2), we encounter some further challenges. In particular, it would have been easy to make the scheme work if our PPPS scheme supported programming a key twice at two points. Specifically, in our construction, when the client consumes a pseudorandom set (represented by sk) in the hint table containing the current query x, it needs to replace the replaced entry with another randomly sampled PPPS key sk subject to containing the query x. One way to achieve this is to fetch an unconsumed key from a backup table, and program the key to contain x. However, later, when the client consumes this already-programmed key in another query y, it needs to program the point y to some other random point in order not to leak the query y.

Unfortunately, our PPPS construction does not support programming twice. Interestingly, earlier works [ZLTS23, LP22] also encountered a similar challenge of needing to program a key twice, but there it was resolved using different techniques that relied on the LWE assumption, which would not work in our setting.

The way we resolve the problem is to introduce a new technique of allowing *broken* entries in the hint table. Basically, if the client consumes some PPPS key sk in the hint table, it simply replaces the consumed sk with a new entry sampled according to the desired distribution (required for privacy). However, since the client did not perform any preparation work during the pre-processing phase for this new entry, consuming this entry later in a new query would result in an incorrect answer, i.e., the replaced entry is *broken*. Fortunately, we can amplify correctness through repetition. We defer the details to the subsequent technical sections.

Other applications of the broken hint technique. The broken hint technique can also lead

to other interesting applications. For example, recall that TreePIR is a 2-server pre-processing scheme [LP23]. With our new broken hint technique, we can convert TreePIR to a single-server scheme which enjoys the efficiency stated in Theorem 1.3.

Further improvements. The approach of using broken entries introduces a super-logarithmic blowup in the bandwidth and computation costs, due the repetition needed for correctness amplification. In Appendix B, we suggest an improved scheme that avoids this super-logarithmic blowup and gets us the tighter bounds stated in Theorem 1.2, but the resulting scheme is somewhat more complex to describe.

2 Formal Definitions

Single-server pre-processing PIR. We first define a single-server pre-processing PIR scheme. A single-server pre-processing PIR scheme consists of two stateful algorithms: the client and the server. The scheme consists of the two following phases.

1. **Pre-processing :** The pre-processing is run only once at the beginning. The client receives no input, while the server receives a database $\mathsf{DB} \in \{0,1\}^n$ as input. The client and server interact and the client may store some information in its local storage. We refer to this information as "hints".

2. **Queries:** This phase is repeated for every index x of the DB that the client wants to read. For every query, the client sends a single message to the server, and the server responds with a single message. The client then performs some computation and outputs an answer β.

Correctness. Given a database DB with entries indexed by $0, 1, \ldots, n-1$, correctness entails that the query for an index $x \in \{0, 1 \ldots, n-1\}$ by the client in the query phase, results in an answer $\mathsf{DB}[x]$ (the x-th bit of DB) output by the client. Formally, correctness requires that for any security parameter $\lambda \in \mathbb{N}$, for any n, q which are polynomially bounded in λ, there exists a negligible function negl such that for any database $\mathsf{DB} \in \{0,1\}^n$, for any sequence of queries $x_1, x_2, \ldots, x_q \in \{0, 1, \ldots, n-1\}$, an honest execution of the PIR scheme with DB and queries $x_1, x_2, \ldots, x_q$ returns all correct answers with probability at least $1 - \mathsf{negl}(\lambda)$.

Privacy. Privacy of a PIR scheme entails that for any index x queried by the client to the server, the view of the server must not leak information about the query x. Formally, we define the privacy of a single-server PIR scheme as follows.

A single-server PIR scheme satisfies privacy if and only if there exists a probabilistic polynomial-time simulator $\mathsf{Sim}(1^\lambda, n)$ such that for any probabilistic polynomial-time adversary $\mathcal{A}$ acting as the server, any polynomially bounded n and q, any database $\mathsf{DB} \in \{0,1\}^n$, $\mathcal{A}$'s views in the following two experiments are computationally indistinguishable:

- Real: an honest client interacts with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$ who acts as the server and may arbitrarily deviate from the prescribed protocol. In every query step $t \in [q]$, $\mathcal{A}$ may adaptively choose the next query $x_t \in \{0, 1, \ldots, n-1\}$ for the client, and the client is invoked with x_t as input.

- Ideal: the simulated client $\mathsf{Sim}(1^\lambda, n)$ interacts with $\mathcal{A}(1^\lambda, n, \mathsf{DB})$ who acts as the server and may arbitrarily deviate from the prescribed protocol. In every query step $t \in [q]$, $\mathcal{A}$ may adaptively choose the next query $x_t \in \{0, 1, \ldots, n-1\}$ for the client, and the client is invoked with x_t as input.

Two-server pre-processing PIR. In the two-server setting, there are two non-colluding servers, and the client may interact with both servers in both the preprocessing and query phases. The two servers do not interact with each other.

Correctness is defined in the same way as the single-server setting. For privacy, we want the definition of the single-server setting to hold for each individual server.

Additional notation. In the formal sections later, for clarity we distinguish between a statistical security parameter denoted κ and a computational security parameter denoted λ.

3 Privately Programmable Pseudorandom Set with List Decoding

3.1 Definition

Distribution of set $\mathcal{D}_n$. We want to construct a pseudorandom set whose distribution emulates a set $S \subset \{0, 1, \ldots, n-1\}$ of size $\sqrt{n}$ sampled from the following distribution denoted $\mathcal{D}_n$ — we assume that n is a perfect forth ($n^{1/4}$ is an integer):

- Divide the n elements into $\sqrt{n}$ chunks indexed with $0, 1, \ldots, \sqrt{n}-1$, where chunk i contains the elements $[\ell \cdot \sqrt{n},\ (\ell+1) \cdot \sqrt{n} - 1]$
- For each chunk $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, sample a random offset $\delta_\ell \xleftarrow{\$} \{0, 1, \ldots, \sqrt{n}-1\}$.
- Output the following set $S := \{\ell \cdot \sqrt{n} + \delta_\ell\}_{\ell \in \{0,1,\ldots,\sqrt{n}-1\}}$.

Offset representation of a set. For convenience, in the rest of the section, we will always use an offset representation of a set, i.e., we will represent a set as

$$S := \{\delta_0, \ldots, \delta_{\sqrt{n}-1}\}$$

where each $\delta_i \in \{0, \ldots, \sqrt{n}-1\}$ represents the relative offset of the i-th element inside the i-th chunk.

Privately programmable pseudorandom set with list decoding. We introduce a new abstraction called a privately programmable pseudorandom set with list decoding that we utilize in our PIR constructions that follow. Intuitively, this primitive provides an algorithm to generate a secret key that represents pseudorandom subset of $\{0, \ldots, n-1\}$ with a specific distribution. Further, the primitive allows, given a key for a pseudorandom set, to produce a key for pseudorandom set that is the same as the starting set except for being programmed at a particular location with a specified value – with the guarantee that the new key does not reveal the programmed location. Moreover, this primitive has a list decoding algorithm, that given a programmed key outputs a list of sets such that one of them is the one is the correct set that the key represents.

Formally, define a privately programmable pseudorandom set (PPPS) with list decoding which emulates the distribution $\mathcal{D}_n$:

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$: takes in the security parameter 1^λ, the size of the set n, and outputs a secret key sk.
- $S \leftarrow \mathsf{Set}(\mathsf{sk})$: takes in a secret key sk, and expands it to a random set S of size $\sqrt{n}$. We sometimes write $\mathsf{Set}(\mathsf{sk})[i]$ to denote the element in the i-th chunk for this set.
- $\mathsf{sk}', i \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$: takes in a secret key sk, a chunk identifier $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, a desired offset δ_ℓ within the specified chunk ℓ, and outputs a programmed key sk', and some auxiliary information i that indicates which of the decoded set will be correct.

- $\{S_0, \ldots, S_{L-1}\} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$: takes in a programmed key sk' and outputs a list of sets $S_0, S_2, \ldots, S_{L-1}$, such that one of them is the correctly programmed set corresponding to the key sk'.

Correctness. Correctness requires that for any $\lambda, n \in \mathbb{N}$, for any $\ell, \delta_\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, the following holds with probability 1: let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, $\mathsf{sk}', i \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, it must be that S_i is equal to the $\mathsf{Set}(\mathsf{sk})$ but replacing the ℓ-th element with δ_ℓ instead.

Pseudorandomness. We say that a PPPS scheme emulates $\mathcal{D}_n$ iff the following two distributions are computationally indistinguishable:

- Sample $S \stackrel{\$}{\leftarrow} \mathcal{D}_n$ and output S;
- Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, output $\mathsf{Set}(\mathsf{sk})$.

Private programmability. We require that there exists a probabilistic polynomial time simulator Sim such that for any $n \in \mathbb{N}$ that is a perfect square and polynomially bounded in λ, any $\ell \in \{0, 1, \ldots, \sqrt{n}-1\}$, any index x that belongs to the ℓ-th chunk, the outputs of the following experiments be computationally indistinguishable:

- Real. Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$ subject to $x \in \mathsf{Set}(\mathsf{sk})$, let $\delta_\ell \stackrel{\$}{\leftarrow} \{0, 1, \ldots, \sqrt{n}-1\}$, and let $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, output sk'.
- Ideal. Output $\mathsf{Sim}(1^\lambda, n)$.

Efficiency. In our PIR scheme later, we need a programmed key sk' to have size at most $O_\lambda(n^{1/4})$. Further, the size of the decoded list $L = n^{1/4}$. Naïvely, since each set has size $\sqrt{n}$, it would take $n^{3/4}$ space to represent the L decoded sets. However, we want our scheme to satisfy a non-trivial notion of efficiency, that is, it takes only $O(\sqrt{n})$ space to represent all L decoded sets. Specifically, the compression is possible because the L decoded sets are correlated.

3.2 Construction

Intuition. In our construction, we will divide the $\sqrt{n}$ *chunks* into $n^{1/4}$ *superblocks* where the i-th superblock contains the i-th group of $n^{1/4}$ consecutive chunks.

To program a PPPS key in some chunk ℓ with the specified offset $\widetilde{\delta}$, we first expand the PPPS key to $n^{1/4}$ superblock keys denoted $k_0, \ldots, k_{n^{1/4}-1}$. Let i be the superblock corresponding to chunk ℓ. We then replace k_i with a randomly sampled superblock key $\widetilde{k}_i$. We than expand k_i into $n^{1/4}$ offsets denoted $\delta_0, \ldots, \delta_{n^{1/4}-1}$, one corresponding to each chunk contained in the i-th superblock. Suppose chunk ℓ corresponds to the j-th chunk within the i-th superblock. We then replace δ_j with the desired $\tilde{\delta}$. The programmed key is the combination of $k_0, \ldots, k_{i-1}, \widetilde{k}_i, k_{i+1}, \ldots, k_{n^{1/4}-1}$, and $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$. Given this programmed key, we do not know which superblock should contain the expanded offsets $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$. However, we can generate a list of $n^{1/4}$ candidate sets by plugging in the offsets $\delta_0, \ldots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \ldots, \delta_{n^{1/4}-1}$ into each of the $n^{1/4}$ superblocks. One of them must be the true programmed set.

Detailed PPPS construction. Henceforth, let $\mathsf{PRF}_1 : \{0,1\}^\lambda \times \{0,1\}^{\frac{\log n}{4}} \to \{0,1\}^\lambda$, and $\mathsf{PRF}_2 : \{0,1\}^\lambda \times \{0,1\}^{\frac{\log n}{4}} \to \{0,1\}^{\frac{\log n}{2}}$ be two pseudorandom functions.

- $\mathsf{Gen}(1^\lambda, n)$: Sample a PRF_1 key sk and output sk.
- $\mathsf{Set}(\mathsf{sk})$:
  1. First, expand sk to $n^{1/4}$ superblock keys:
  $$\forall i \in \{0, \dots, n^{1/4} - 1\}: \;\; k_i = \mathsf{PRF}_1(\mathsf{sk}, i) \tag{1}$$
  2. Next, for each superblock $i \in \{0, \dots, n^{1/4} - 1\}$, compute the pseudorandom offset for each of its $n^{1/4}$ chunks, that is:
  $$\forall i, j \in \{0, \dots, n^{1/4} - 1\}: \;\; \delta_{i,j} = \mathsf{PRF}_2(k_i, j) \tag{2}$$
  3. Define the alias $\delta_{i \cdot n^{1/4} + j} := \delta_{i,j}$, and output $S := \{\delta_\ell\}_{\ell \in \{0, \dots, \sqrt{n}-1\}}$.
- $\mathsf{Program}(\mathsf{sk}, \ell, \widetilde{\delta})$:
  1. Expand sk to $n^{1/4}$ superblock keys denoted $k_0, \dots, k_{n^{1/4}-1}$ as in Equation (1).
  2. Let $i := \lfloor \ell / n^{1/4} \rfloor$ be the superblock containing the ℓ-th chunk, let $j := \ell \mod n^{1/4}$ be the index of chunk ℓ within superblock i.
  3. Sample a fresh PRF key $\widetilde{k}_i$ to replace k_i with.
  4. For $j' \in \{0, 1, \dots, n^{1/4} - 1\}$, compute $\delta_{j'} = \mathsf{PRF}_2(k_i, j')$.
  5. Output the following:
  $$\mathsf{sk}' := \begin{pmatrix} (k_0, \dots, k_{i-1}, \widetilde{k}_i, k_{i+1}, \dots, k_{n^{1/4}-1}), \\ (\delta_0, \dots, \delta_{j-1}, \widetilde{\delta}, \delta_{j+1}, \dots, \delta_{n^{1/4}-1}), \end{pmatrix}, \quad i$$
- $\mathsf{ListDecode}(\mathsf{sk}')$:
  1. Parse $\mathsf{sk}' = (\{k_i\}_{i \in \{0, \dots, n^{1/4}-1\}}, \{\delta_j^*\}_{j \in \{0, \dots, n^{1/4}-1\}})$.
  2. $\forall i, j \in \{0, \dots, n^{1/4} - 1\}$, compute $\delta_{i,j}$ like in Equation (2), let S be the matrix $S := \{\delta_{i,j}\}_{i,j \in \{0,1,\dots,n^{1/4}-1\}}$.
  3. For $i \in \{0, \dots, n^{1/4} - 1\}$, let S_i be the same as S except for substituting the i-th row with $\{\delta_j^*\}_{j \in \{0, \dots, n^{1/4}-1\}}$. In other words,
  $$S_i := \begin{pmatrix} \delta_{0,0}, & \dots, & \delta_{0,n^{1/4}-1}, \\ \dots, & \dots, & \dots, \\ \delta_{i-1,0}, & \dots, & \delta_{i-1,n^{1/4}-1}, \\ \delta_0^*, & \dots, & \delta_{n^{1/4}-1}^*, \\ \delta_{i+1,0}, & \dots, & \delta_{i+1,n^{1/4}-1}, \\ \dots, & \dots, & \dots, \\ \delta_{n^{1/4}-1,0}, & \dots, & \delta_{n^{1/4}-1,n^{1/4}-1}, \end{pmatrix}$$
  4. Output $(\mathsf{Flatten}(S_0), \dots, \mathsf{Flatten}(S_{n^{1/4}-1}))$ where $\mathsf{Flatten}$ outputs the vector obtained from concatenating all rows of the matrix,

Size of programmed key and efficiency of ListDecode. Clearly, the programmed key sk' output by $\mathsf{Program}$ has size $O_\lambda(n^{1/4})$. It is also easy to have a succinct representation of size $O(\sqrt{n})$ of all

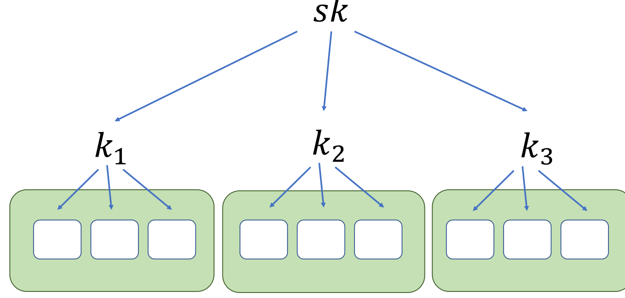
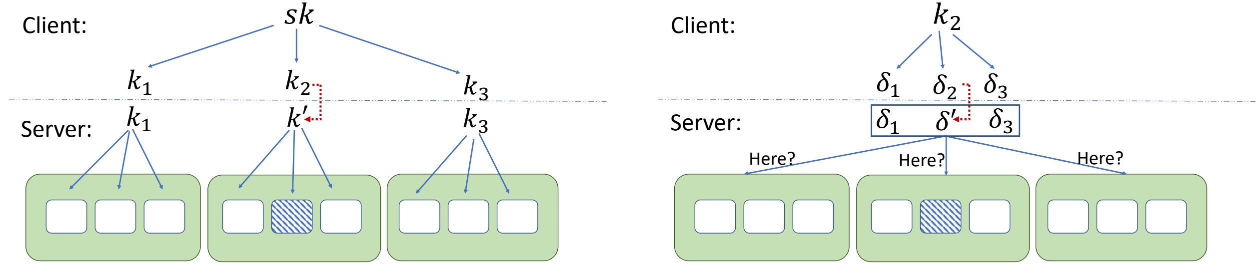


Figure 1: Two-layer set representation. The first layer key expands to $n^{1/4}$ superblock keys. Each superblock key further expands to $n^{1/4}$ offsets, one for each chunk in the superblock.



Step 1: The client expands the PPPS key to $n^{1/4}$ superblock keys and replaces the key corresponding to x's superblock with a random key.

Step 2: The client expands the replaced superblock key to $n^{1/4}$ offsets and replaces x's offset to a random one. The server constructs the candidate sets by plugging these offsets into every superblock.

Figure 2: Illustration about how PPPS is used in our PIR schemes. The client programs the key and the server will decode the list of candidate sets.

$n^{1/4}$ candidate sets output by ListDecode. Specifically, one can first compute the common set S of size $\sqrt{n}$ (we abuse the notation that this set is derived from flattening the matrix S in ListDecode). Then, the symmetric difference between the i-th candidate set and the common set S is just $2n^{1/4}$ elements (those elements in the i-th superblocks). So the succinct representation (and hence the efficient algorithm) of ListDecode takes $O_\lambda(\sqrt{n})$ space and time.

Efficient set membership. The above construction also supports $O_\lambda(1)$-time set membership query. Given a secret key sk that has not been programmed, to check if some element $x \in \mathsf{Set}(\mathsf{sk})$ or not, one simply has to check

$$\mathsf{PRF}_2(\mathsf{PRF}_1(\mathsf{sk}, \lfloor \ell / n^{1/4} \rfloor), \quad \ell \bmod n^{1/4}) \stackrel{?}{=} x \bmod n^{1/2} \text{ where } \ell = \lfloor x / n^{1/2} \rfloor$$

3.3 Proof of Correctness

To see correctness, let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, let $\mathsf{sk}', i^* \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$. Recall that sk' can be parsed as $\mathsf{sk}' = (\{k_i\}_{i \in \{0,\ldots,n^{1/4}-1\}}, \{\delta_i^*\}_{i \in \{0,\ldots,n^{1/4}-1\}})$, and by construction, we know that $i^* = \lfloor \ell / n^{1/4} \rfloor$ is the index of the superblock that contains the chunk ℓ. Let $j^* := \ell \mod n^{1/4}$. Let $S_\emptyset := \mathsf{Set}(\mathsf{sk})$, and we can view $S_\emptyset$ as a $n^{1/4} \times n^{1/4}$ matrix. The correct programmed set S^* is $S_\emptyset$ but replacing the element at index (i^*, j^*) with δ_ℓ.

Below, we show that the set S_i output by ListDecode is the same as S^*. By construction, in the $\mathsf{ListDecode}(\mathsf{sk}')$ algorithm, the intermediate set S is the same as $S_\emptyset$ except for the i^*-th row.

Further, the i^*-th row of $S_\emptyset$ is the same as $\{\delta_j^*\}_{j\in\{0,\dots,n^{1/4}-1\}}$ but replacing the j^*-th element with δ_ℓ. Additionally, the S_i output by $\mathsf{ListDecode}$ is obtained by replacing the i^*-th row of S with $\{\delta_i^*\}_{i\in\{0,\dots,n^{1/4}-1\}}$.

3.4 Proof of Security

We now prove pseudorandomness and private programmability assuming the security of the underlying PRF_1 and PRF_2.

Pseudorandomness. Pseudorandomness follows directly from the pseudorandomness of the underlying PRFs.

Private programmability. We can consider the following sequence of hybrid experiments. Fix an arbitrary chunk identifier ℓ and an index x that belongs to the ℓ-th chunk. Throughout, let $i^* = \lfloor \ell/n^{1/4} \rfloor$, let $j^* = \ell \mod n^{1/4}$.

Experiment Real. Recall the definition of the real experiment. Sample a PRF key sk such that $\mathsf{PRF}_2(\mathsf{PRF}_1(\mathsf{sk}, i^*), j^*) = x \mod \sqrt{n}$. Let $\delta_\ell \stackrel{\$}{\leftarrow} \{0, 1, \dots, \sqrt{n}-1\}$, and let $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$, output sk'.

Experiment Hyb. Same as Real except with the following modification: when executing the $\mathsf{Program}(\mathsf{sk}, \ell, \delta_\ell)$ algorithm instead of using the $k_0, \dots, k_{n^{1/4}-1}$ keys that are expanded using $\mathsf{PRF}_1(\mathsf{sk}, \cdot)$, sample $k_0, \dots, k_{n^{1/4}-1}$ at random subject to $\mathsf{PRF}_2(k_{i^*}, j^*) = x \mod \sqrt{n}$.

Lemma 3.1. *Suppose that* PRF_1 *is secure. Then,* Hyb *is computationally indistinguishable from* Real.

Proof. Suppose there is an efficient adversary $\mathcal{A}$ that can distinguish Real and Hyb with non-negligible probability. We can construct the following efficient reduction $\mathcal{B}$ which can distinguish a PRF from a random function with non-negligible probability. Basically, $\mathcal{B}$ is interacting with its own challenger who either answers queries using a PRF or using a truly random function. $\mathcal{B}$ will query its own challenger on the inputs $0, 1, \dots, n^{1/4}-1$, and it will obtain $k_0, \dots, k_{n^{1/4}-1}$ from its challenger. It will check if k_{i^*} satisfies the relation $\mathsf{PRF}(k_{i^*}, j^*) = x \mod \sqrt{n}$. If not, $\mathcal{B}$ aborts and outputs 0. Otherwise, it runs the $\mathsf{Program}$ algorithm where it plugs in the terms $k_0, \dots, k_{n^{1/4}-1}$ as the superblock keys. It gives the resulting sk' to $\mathcal{A}$. Henceforth, we use $b = 0$ to denote the world in which $\mathcal{B}$'s challenger uses a truly random function, and we use $b = 1$ to denote the world in which $\mathcal{B}$'s challenger uses a randomly sampled PRF function. We use the notation $\Pr_b[\cdot]$ to denote the probability of events in world $b \in \{0, 1\}$. Let G be the good event that the relation $\mathsf{PRF}(k_{i^*}, j^*) = x \mod \sqrt{n}$ is satisfied.

$$\Pr_b[\mathcal{B} \text{ outputs } 1] = 0 \cdot \Pr_b[\overline{G}] + \Pr_b[\mathcal{A} \text{ outputs } 1|G] \cdot \Pr_b[G]$$

We know that $\Pr_0[G] = 1/\sqrt{n}$ which is non-negligible. If the PRF is secure, then it must be that $|\Pr_1[G] - \Pr_0[G]| \le \mathsf{negl}(\lambda)$ due to a straightforward reduction to PRF security. Therefore, we have that

$$\begin{aligned}
&\left|\Pr_1[\mathcal{B} \text{ outputs } 1] - \Pr_0[\mathcal{B} \text{ outputs } 1]\right| \\
=& \left|\Pr_1[\mathcal{A} \text{ outputs } 1|G] \cdot \Pr_1[G] - \Pr_0[\mathcal{A} \text{ outputs } 1|G] \cdot \Pr_0[G]\right| \\
\ge& \left|\Pr_1[\mathcal{A} \text{ outputs } 1|G] - \Pr_0[\mathcal{A} \text{ outputs } 1|G]\right| \cdot \frac{1}{\sqrt{n}} - \mathsf{negl}(\lambda)
\end{aligned}$$

Observe also that in world 0, conditioned on G, $\mathcal{A}$'s view in the experiment is identically distributed as Hyb. In world 1, conditioned on G, $\mathcal{A}$'s view in the experiment is identically distributed as Real. Therefore, the term

$$\left|\Pr_1[\mathcal{A} \text{ outputs } 1|G] - \Pr_0[\mathcal{A} \text{ outputs } 1|G]\right|$$

represents $\mathcal{A}$'s advantage in distinguishing Real and Hyb. We can now conclude that if $\mathcal{A}$ can distinguish Real and Hyb with non-negligible probability, then $\mathcal{B}$ can break PRF security with non-negligible probability. ■

Experiment Ideal. The Ideal experiment is almost the same as Hyb except with the following modification: when outputting the sk', instead of using the $\delta_0, \ldots, \delta_{j^*-1}, \delta_{j^*+1}, \delta_{n^{1/4}-1}$ terms derived from evaluating $\mathsf{PRF}_2(k_{i^*}, \cdot)$ at the points $0, 1, \ldots, j^* - 1, j^* + 1, \ldots, n^{1/4} - 1$, we now sample $\delta_0, \ldots, \delta_{j^*-1}, \delta_{j^*+1}, \delta_{n^{1/4}-1}$ at random from $\{0, \ldots, n^{1/2} - 1\}$ instead.

Observe that in the Ideal experiment, we no longer make use of knowledge of the query x. Therefore, the description of the Ideal experiment also uniquely specifies the simulator Sim we want to construct.

Lemma 3.2. *Suppose that* PRF_2 *is secure. Then,* Ideal *is computationally indistinguishable from* Hyb*.*

Proof. It suffices to show that the following to probability ensembles are computationally indistinguishable for any fixed $\delta^* \in \{0, 1, \ldots, \sqrt{n} - 1\}$, and $j^* \in \{0, 1, \ldots, n^{1/4} - 1\}$.

1. Distr_0: Output a randomly sampled vector $\delta_0, \ldots, \delta_{n^{1/4}-1} \in \{0, 1, \ldots, \sqrt{n} - 1\}^{n^{1/4}}$.
2. Distr_1: Sample a PRF key k subject to $\mathsf{PRF}_2(k, j^*) = \delta^*$. Sample $\delta' \in \{0, 1, \ldots, \sqrt{n}-1\}$ at random. For $j \in \{0, 1, \ldots, n^{1/4} - 1\}$, compute $\delta_j = \mathsf{PRF}_2(k, j)$. Output $\delta_0, \ldots, \delta_{j^*-1}, \delta', \delta_{j^*+1}, \ldots, \delta_{n^{1/4}-1}$.

If there is an efficient adversary $\mathcal{A}$ that can distinguish between the above Distr_0 and Distr_1 with non-negligible probability, we can construct an efficient reduction $\mathcal{B}$ that can distinguish whether it is interacting with a random oracle or a randomly chosen PRF function. Basically, $\mathcal{B}$ sends the inputs $0, \ldots, n^{1/4} - 1$ to the oracle it is interacting with, and gets back $\delta_0, \ldots, \delta_{n^{1/4}-1}$. If $\delta_{j^*} \neq \delta^*$, then $\mathcal{B}$ aborts and outputs 0. Otherwise, it replaces δ_{j^*} with a random value from $\{0, \ldots, n^{1/2} - 1\}$ and gives the resulting vector to $\mathcal{A}$. Suppose $\mathcal{B}$ is interacting with a random oracle, then conditioned on the good event $\delta_{j^*} = \delta^*$, $\mathcal{A}$'s view is identically distributed as Distr_0. On the other hand, suppose $\mathcal{B}$ is interacting with a PRF, then conditioned on the good event $\delta_{j^*} = \delta^*$, $\mathcal{A}$'s view is identically distributed as Distr_1. The rest of the proof can be completed due to a similar probability calculation as Theorem 3.1. ■

4 Our Two-Server PIR Scheme

4.1 Construction

Detailed algorithm for bounded, random queries. We describe the detailed construction for $Q = \sqrt{n} \log \kappa \cdot \alpha$ random, distinct queries in Figure 3. We can easily extend such a scheme to support unbounded, arbitrary queries using known techniques [ZLTS23]. For completeness, we explain how the extension works shortly after.

Two-server scheme for $Q = \sqrt{n} \log \kappa \cdot \alpha$ queries

Offline preprocessing.

- *Hint table.* Let $M_1 = \sqrt{n} \log \kappa \cdot \alpha(\kappa)$. For each $i \in [M_1]$, sample a fresh PPPS key sk_i, send sk_i to the right server and receive a parity $p_i := \oplus_{j \in \mathsf{Set}(\mathsf{sk}_i)} \mathsf{DB}[j]$ back. Let $T := \{(\mathsf{sk}_i, p_i)\}_{i \in [M_1]}$ denote the client's *hint table.*
- *Replacement entries.* For each chunk $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, repeat the following $M_2 = 3 \log \kappa \cdot \alpha(\kappa)$ times: sample a random index $r_1 \in \{0, \ldots, n-1\}$ in chunk ℓ, send r_1 to the left server, and receive $\mathsf{DB}[r_1]$. Store the tuple $(r_1, \mathsf{DB}[r_1])$.

  Similarly, for each chunk $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, repeat the following M_2 times: sample a random index r_2 in chunk ℓ, send r_2 to the right server, and receive $\mathsf{DB}[r_2]$. Store the tuple $(r_2, \mathsf{DB}[r_2])$.

Query for index $x \in \{0, 1, \ldots, n-1\}$.

1. **Step 1: (Client Querying)**
   - Find the first entry (sk, p) in the hint table such that $x \in \mathsf{Set}(\mathsf{sk})$. [a]
   - Find the first unconsumed replacement entries $(r_1, \mathsf{DB}[r_1])$ retrieved from right server, such that r_1 is in $\mathsf{chunk}(x)$. [b]
   - $(\mathsf{sk}'_1, j_1) \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), r_1 \bmod \sqrt{n})$.
   - Send sk_1 to the left server.
2. **Step 2: (Client Reconstructing)**
   - Receive $(\beta_{0,1}, \ldots, \beta_{n^{1/4}-1,1})$ from the left server.
   - Save the answer as $y = p \oplus \mathsf{DB}[r_1] \oplus \beta_{j_1,1}$.
3. **Step 3: (Client Refreshing)**
   - Sample sk_2 such that $x \in \mathsf{Set}(\mathsf{sk}_2)$.
   - Find the first unconsumed replacement entries $(r_2, \mathsf{DB}[r_2])$ retrieved from left server, such that r_2 is in $\mathsf{chunk}(x)$.
   - $(\mathsf{sk}'_2, j_2) \leftarrow \mathsf{Program}(\mathsf{sk}_2, \mathsf{chunk}(x), r_2 \bmod \sqrt{n})$.
   - Send sk'_2 to right server.
   - Receive $(\beta_{0,2}, \ldots, \beta_{n^{1/4}-1,2})$ from the right server.
   - Replace the hint (sk, p) with $(\mathsf{sk}_2, \mathsf{DB}[r_2] \oplus \beta_{j_2,2} \oplus y)$ in the table.
4. **Server Responding: (Same for Left and Right Server)**
   - Upon receiving sk', compute $(S_0, S_1, \ldots, S_{n^{1/4}-1}) \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$.
   - Return $(\beta_0, \ldots, \beta_{n^{1/4}-1})$ to the client where $\beta_i = \oplus_{i \in S_b} \mathsf{DB}[i]$.

[a] In a rare case, if not found, let sk be a freshly sampled PPPS key subject to $x \in \mathsf{Set}(\mathsf{sk})$, and let $p = 0$.

[b] In a rare case, if such an r_1 is not found, let it be a random index in $\mathsf{chunk}(x)$, and use 0 whenever $\mathsf{DB}[r_1]$ is needed later.

Figure 3: Two-server preprocessing PIR with $O_\lambda(n^{1/4})$ communication, $O_\lambda(n^{1/2})$ computation based on PRFs.

Efficiency. Observe that the list decoding produces $n^{1/4}$ candidate sets each of size $\sqrt{n}$. Naively, expanding all sets and computing their corresponding parities takes $O_\lambda(n^{3/4})$ time. We are still going to rely on the fact that $\mathsf{ListDecode}$ has an $O(\sqrt{n})$-size succinct representation to optimize the computation. Recall that we can first compute the common set of size $\sqrt{n}$, and then the symmetric difference between each possible decoding set and the common set will only contain $2n^{1/4}$ elements. Therefore, to compute the parities for all $n^{1/4}$ possible sets, we first compute the parity for the common set S, which takes $O_\lambda(\sqrt{n})$ time. Then, it takes $O_\lambda(n^{1/4})$ time to enumerate the symmetric difference between the i-th set and the common set. So the total computation time will be $O_\lambda(\sqrt{n})$.

Supporting unbounded, arbitrary queries. For completeness, we review the techniques described in previous works for upgrading the scheme for Q random, distinct queries to a scheme supporting unbounded, arbitrary queries. We can easily get rid of the distinct query assumption in the following way: we require the client to store a local cache of size Q for caching the most recent Q queries. If the client wants a repeated query, it can lookup in the cache and make a distinct fake query.

Further, we can assume that the queries are random without loss of generality as follows: we can let the client and the servers agree on a small-domain pseudorandom permutation (PRP) [RY13, HMR12a] (which is implied by one-way functions [HMR12b]) upfront and the server can permute the database according to the PRP. Another option is let one of the servers build the database as a key-value storage and use a cuckoo hash table [PR04, Yeo23] directly based on a PRF to locate the queries, and share it with the other server. Notice that, in both implementations, the client can still make queries adaptively depending on the real query sequence and the responses, which is sufficient for practical usage. Then, as long as the client makes the queries independent of the randomness of the PRP/PRF, those queries can be considered as uniformly random. This assumption is only needed for the correctness.

Lastly, we can remove the bounded Q query assumption as follows: we use a pipelining trick suggested in earlier works [ZLTS23, ZPSZ24]. Essentially, we can spread the pre-processing for the next window of Q queries over the current window of Q queries.

Theorem 4.1. *Let $\alpha(\kappa)$ be any superconstant function. Suppose that $\mathsf{PRF}_1, \mathsf{PRF}_2$ are secure pseudorandom functions, and that n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The two-server scheme in Figure 3 that supports $Q = \sqrt{n}\log\kappa \cdot \alpha$ random, distinct queries is private, and correct with probability $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(.)$. Further, it achieves the following performance bounds:*

- *$O_\lambda(\sqrt{n}\log\kappa\alpha(\kappa))$ client storage and no additional server storage;*
- ***Pre-processing Phase:***
  - *$O_\lambda(n\log\kappa \cdot \alpha)$ server time and $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$ client time;*
  - *$O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$ communication;*
- ***Query Phase:***
  - *$O_\lambda(\sqrt{n})$ expected client time and $O_\lambda(\sqrt{n})$ server time per query;*
  - *$O_\lambda(n^{1/4})$ communication per query.*

Therefore, the amortized communication per query is $O_\lambda(n^{1/4})$, and the amortized server computation and expected client computation per query is $O_\lambda(\sqrt{n})$.

Proof. We defer the privacy and correctness proofs to Section 4.2 and Section 4.3 respectively. Here, we focus on proving the efficiency claims.

The client stores $M_1 = \sqrt{n}\log\kappa \cdot \alpha$ number of PPPS keys, and $M_2 = 3\log\kappa \cdot \alpha$ number of replacement entries per chunk. Therefore, the space required is $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$.

During the offline phase, the client sends M_1 PPPS keys to the right server, and sends M_2 indices per chunk to either server for constructing replacement entries. Therefore, the offline communication is bounded by $O_\lambda(\sqrt{n}\log\kappa \cdot \alpha)$. The right server needs to expand the sets for each PPPS key received and evaluate the xor-sums. Both servers need to return the bits for the replacement entries. Therefore, the total server computation is bounded by $O_\lambda(n\log\kappa \cdot \alpha)$.

During the query phase, the client sends one programmed PPPS key to each server, and the size of a programmed key is at most $O_\lambda(n^{1/4})$. Each server sends back the xor-sums of $n^{1/4}$ candidate sets. Each candidate set has size $n^{1/2}$, however, all $n^{1/4}$ candidate sets has a succinct representation of size only $n^{1/2}$, and server can compute this succinct representation in time $O_\lambda(n^{1/2})$. Further, it is not hard to see that due to the structure of the candidate sets, the server can compute all $n^{1/4}$ xor-sums in time only $O(n^{1/2})$. Therefore, the servers' running time is bounded by $O_\lambda(n^{1/2})$ during each query query. The client needs to find a matched hint, and compute $O(1)$ xor operations during each query. Its running time is dominated by the cost of finding a matched hint, which can be done by invoking the set membership operation for each of the M_1 hints. Using Theorem 4.2 and the pseudorandomness property of the PPPS, the expected number of hints checked until a key sk such that $\mathsf{Set}(\mathsf{sk})$ contains the current query is found is $O(\sqrt{n})$. The expected number of tries till success is $\sqrt{n}$. Therefore, the client's expected running time per query is upper bounded by $O_\lambda(\sqrt{n})$. ■

4.2 Privacy Proof

Suppose that the underlying PPPS scheme satisfies private programmability. Below, we prove the privacy of our two-server PIR scheme.

In the pre-processing phase, the server sends the sets $\mathsf{Set}(\mathsf{sk}_i)$ only to the right server, thereby no information about these sets is leaked to the left server. Similarly, no information about the indices r_1 is leaked to the right server and no information about the indices r_2 is leaked to the left server. We will first the lemma about the distribution of client's hint table, when the adversary controls either of the left or right server.

Lemma 4.2. *Recall that in each time step t, the adversary $\mathcal{A}$ adaptively chooses a query $x_t \in \{0, 1, \ldots, n-1\}$ for the client. At the end of each time step t, the client's hint table is distributed as a table of size M_1 where each entry is a freshly sampled PPPS key, even when conditioned on $\mathcal{A}$'s view so far.*

Proof. Suppose the above statement holds at the end of time step $t-1$. We prove that it still holds at the end of time step t. Since the hint table is distributed as a fresh randomly sampled table even when conditioned on $\mathcal{A}$'s view at the end of $t-1$, we may henceforth assume an arbitrary fixed query x_t. The distribution of the hint table before the t-th query can be equivalently rewritten as:

- First, sample the decision whether any of the M_1 entries contains the current query x_t, and if so, which is the first entry (denoted i^*) that contains x_t. If not found, we assume $i^* = M_1 + 1$.
- For each $i < i^*$, sample a random PPPS key subject to not containing x_t.
- For each $i = i^*$, sample a random PPPS key subject to containing x_t.
- For each $i > i^*$, sample a random PPPS key.

Using the above interpretation, it is easy to see that the distribution of the hint table after the t-th query is unaltered no matter which of the two servers $\mathcal{A}$ controls. ■

4.2.1 Left Server Privacy

We first construct the following simulator for proving left server privacy.

Simulator construction.

- During the pre-processing phase, for each chunk ℓ, sample M_2 random indices belonging to ℓ, send them to $\mathcal{A}$.
- During each query, call the simulator of the PPPS scheme which outputs sk', send sk' to $\mathcal{A}$.

Indistinguishability of Real and Ideal. We now prove the indistinguishability of the Real and Ideal for both the servers assuming the private programmability of the underlying PPPS scheme.

First, due to Theorem 4.2, we can equivalently rewrite the Real experiment for the right server as follows: at the end of each time step, resample the entire hint table freshly at random before continuing to answer more queries. As a result, the view of $\mathcal{A}$ who controls the right server is distributed as:

- *Pre-processing phase.* For each chunk ℓ, send M_2 random indices in chunk ℓ to $\mathcal{A}$.
- *Each time step t.*
  - sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n} - 1\}$.
  - call $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
  - send sk' to $\mathcal{A}$.

One way to see this is to think of the distribution of the table as the equivalent distribution in the proof of Theorem 5.2. Further, observe that each $r_2 \bmod \sqrt{n}$ in the scheme is distributed randomly from the perspective of the left server, since they were only sent to the right server during the pre-processing phase.

Therefore, the rest of the proof follows due to a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the right server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

4.2.2 Right Server Privacy

We first construct the following simulator for proving right server privacy.

Simulator construction.

- During the pre-processing phase, send M_1 randomly sampled PPPS keys to $\mathcal{A}$. Further, for each chunk ℓ, sample M_2 random indices in ℓ, send them to $\mathcal{A}$.
- During each query, call the simulator of the PPPS scheme which outputs sk', send sk' to $\mathcal{A}$.

Indistinguishability of Real and Ideal. We now prove the indistinguishability of the Real and Ideal for both the servers assuming the private programmability of the underlying PPPS scheme. The view of $\mathcal{A}$ who controls the right server is distributed as:

- *Pre-processing phase.* Sample M_1 random PPPS keys, and send them to $\mathcal{A}$. Further, for each chunk ℓ, send M_2 random indices in chunk ℓ to $\mathcal{A}$.
- *Each time step t.*
  - sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n} - 1\}$.
  - call $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
  - send sk' to $\mathcal{A}$.

To see the above, observe that each $r_1 \bmod \sqrt{n}$ in the scheme is distributed randomly from the perspective of the right server, since they were only sent to the left server during the pre-processing phase.

Therefore, the rest of the proof follows due to a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the right server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

4.3 Correctness Proof

We show that with $Q = \sqrt{n} \log \kappa \cdot \alpha$ random, distinct queries, the probability of ever having correctness error is negligibly small. An error can happen if one of the following bad events take place:

- *No matched hint.* During some query for x, no hint is found that contains the query x.
- *Depleting replacement entries.* During some query for x, there is no more replacement entry of the form $(r_1, \mathsf{DB}[r_1])$ or $(r_2, \mathsf{DB}[r_2])$ corresponding to $\mathsf{chunk}(x)$.

Below, we show that the probability of each bad event during a window of Q random, distinct queries is negligibly small.

Probability of no matched hint. Due to Theorem 4.2, for any fixed time step t, we can assume the client's hint table contains freshly sampled PPPS keys and is independent of the current query x_t. Due to the pseudorandomness property of the PPPS, the sets generated by the keys in the hint table are computationally indisintinguishable from M_1 sets independently sampled from the distribution $\mathcal{D}_n$. Below we calculate the probability that a fixed element x_t is not in any of the M_1 sets sampled independently from $\mathcal{D}_n$ — the probability that x_t is not contained in any entry in the client's hint table can only be negligibly different.

The probability that one set sampled from $\mathcal{D}_n$ contains x_t is $1/\sqrt{n}$. Therefore, the probability that none of the M_1 sets contains x_t is $(1 - 1/\sqrt{n})^{M_1}$, and given the choice of $M_1 = \sqrt{n} \log \kappa \alpha(\kappa)$ where $\alpha(\kappa)$ is a super-constant function, this probability is negligibly small in κ.

Finally, taking a union bound over all polynomially many time steps, the probability of ever not having a matched hint is negligibly small in κ.

Probability of depleting replacement entries. One can only deplete the replacement entries of some chunk ℓ if the chunk ℓ is encountered more than M_2 times. With Q random distinct queries, each query will hit a random chunk. The expected number of hits per chunk is therefore $Q/\sqrt{n} = \log \kappa \cdot \alpha$. By the Chernoff bound, the probability that the number of visits to some fixed chunk ℓ exceeds $M_2 = 3 \log \kappa \cdot \alpha$ is negligibly small in κ as long as $\alpha(\kappa)$ is a super-constant function.

Finally, taking a union bound over all chunks and all polynomially many time steps, the probability of ever depleting replacement entries of any chunk is negligibly small in κ.

Single-Server Scheme for $Q = \sqrt{n}/2$ Queries [a]

Notation. κ denotes a *statistical* security parameter, λ denotes a computational security parameter. We use $\alpha(\kappa)$ to denote an arbitrarily small super-constant function.

Preprocessing.

- Client samples $M_1 = 2\sqrt{n}\log\kappa \cdot \alpha(\kappa)$ master PPPS keys denoted $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \in \{0,1\}^\lambda$. Initialize the parities $p_1, \ldots, p_{M_1}$ to zeros.
- Client downloads the whole DB from the server in a streaming way: when the client has the j-th chunk $\mathsf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$:
  - *Update primary table*: for $i \in [M_1]$, let $p_i \leftarrow p_i \oplus \mathsf{DB}[\mathsf{Set}(\mathsf{sk}_i)[j]]$.
  - *Store replacement entries*: sample and store $M_2 = 3\log\kappa \cdot \alpha(\kappa)$ tuples of the form $(r, \mathsf{DB}[r])$ where r is a random index from the j-th chunk.
  - Delete $\mathsf{DB}[j\sqrt{n} : (j+1)\sqrt{n}]$ from the local storage.
- At this moment, let $T := \{(\mathsf{sk}_i, p_i)\}_{i\in[M_1]}$ denote the client's *hint table*. Mark all the hints as "good".

Query for index $x \in \{0, 1, \ldots, n-1\}$.

1. Client: For each matched entry (sk_i, p_i) such that $x \in \mathsf{Set}(\mathsf{sk}_i)$ in the hint table, do the following unless there are already $M_3 = 3\log\kappa \cdot \alpha$ matched entries:
   - For the first good (i.e., non-broken) matched entry, find the first unconsumed replacement entry $(r, \mathsf{DB}[r])$ for $\mathsf{chunk}(x)$. [b]
   - Otherwise, sample a random index r in $\mathsf{chunk}(x)$.
   - $(\mathsf{sk}', i^*) \leftarrow \mathsf{Program}(\mathsf{sk}_i, \mathsf{chunk}(x), r \bmod \sqrt{n})$.
   - Send sk' to the server, and receive $\{\beta_i\}_{i\in\{0,\ldots,n^{1/4}-1\}}$ from the server.
   - For the first good matched entry, save the answer $p_i \oplus \beta_{i^*} \oplus \mathsf{DB}[r]$.
   - Sample a fresh PPPS key $\mathsf{sk}_{\mathrm{new}}$ subject to $x \in \mathsf{Set}(\mathsf{sk})$, and replace the consumed entry (sk_i, p_i) with $(\mathsf{sk}_{\mathrm{new}}, 0)$ and mark the entry as *broken*.
2. Client: If less than M_3 keys are sent in the previous step, send more dummy programmed keys to the server until there are M_3 keys sent [c].
3. Client: Output the saved answer. If no answer was saved, output 0.
4. Server: for each sk' received, let $S_0, \ldots, S_{n^{1/4}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$. For each $i \in \{0, \ldots, n^{1/4}-1\}$, send the xor-sum $\oplus_{j\in S_i}\mathsf{DB}[j]$ to the client[d].

[a] We first present the scheme supporting distinct and random queries. As mentioned, these restrictions can be removed by applying PRP and local caching.

[b] If not found, treat it as the otherwise case.

[c] The dummy key is constructed as sampling a random PPPS key sk subject to $x \in \mathsf{Set}(\mathsf{sk})$ and call $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), \delta')$.

[d] We use the normal representation of the set S_i and not the offset representation.

Figure 4: Our single-server pre-processing PIR scheme.

5 Our Single-Server PIR Scheme

5.1 Construction

Notation. For $x \in \{0, 1,] \ldots, n-1\}$, we define $\mathsf{chunk}(x) := \lfloor x/n^{1/2} \rfloor$ and $\mathsf{superblock}(x) := \lfloor \mathsf{chunk}(x)/n^{1/4} \rfloor$. We assume $(\mathsf{Gen}, \mathsf{Set}, \mathsf{Program}, \mathsf{Decode})$ is a PPPS scheme over the distribution $\mathcal{D}_n$ as described in Section 3.

Detailed algorithm for bounded, random queries. In Figure 4, we describe our algorithm which supports $Q = \sqrt{n}/2$ random and distinct queries. It is well-known how to upgrade such an algorithm to support an *unbounded* number of *arbitrary* queries [ZLTS23]. For completeness, we briefly describe the upgrade shortly after.

Efficiency. Observe that although the list decoding produces $n^{1/4}$ candidate sets each of size $\sqrt{n}$, all $n^{1/4}$ sets can actually be represented using only $O(\sqrt{n})$ space. Further, computing the parities of all sets takes only $O(\sqrt{n})$ time. This is because all $n^{1/4}$ sets are derived from some common set S, but replacing offsets within each of the $n^{1/4}$ superblocks with another random vector $\delta_0, \ldots, \delta_{n^{1/4}-1}$. We give a full efficiency analysis in the proof of Theorem 5.1.

Supporting unbounded, arbitrary queries. We can easily get rid of the distinct query assumption in the following way: we can require the client to store a local cache of size Q for caching the most recent Q queries. If the client wants a repeated query, it can lookup in the cache and make a dummy query.

Further, we can assume that the queries are random without loss of generality as follows: we can let the client and the server agree on a pseudorandom permutation (PRP) [RY13, HMR12a] upfront and the server can permute the database according to the PRP. Another option is let the server build the database as a key-value storage and use a cuckoo hash table [PR04, Yeo23] directly based on a PRF to locate the queries. Notice that, in both implementations, the client can still make queries adaptively depending on the real query sequence and the responses, which is sufficient for practical usage. Then, as long as the client makes the queries independent of the randomness of the PRP/PRF, those queries can be considered as uniformly random. This assumption is only needed for the correctness.

Lastly, we can remove the bounded Q query assumption as follows: the straightforward way is that once the client finishes a window of Q queries, the client and the server reruns the preprocessing phase again, using fresh randomness. The drawback is that the client has to wait a long time before starting the next window. As previous work pointed out [ZPSZ24, ZLTS23], we can easily avoid this drawback through a simple pipelining trick, by spreading the preprocessing work of the next Q window over the current Q window of queries.

Theorem 5.1. *Let $\alpha(\kappa)$ be any super-constant function. Suppose that $\mathsf{PRF}_1, \mathsf{PRF}_2$ are secure pseudorandom functions, and n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The single-server scheme in Figure 4 that supports $\sqrt{n}/2$ random, distinct queries is private, and correct with probability $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ for some negligible function $\mathsf{negl}(\cdot)$. Further, it achieves the following performance bounds:*

- $O_\lambda(\sqrt{n} \log \kappa \cdot \alpha)$ *client storage and no additional server storage;*
- ***Pre-processing Phase:***
  - $O(n)$ *server time and* $O_\lambda(n \log \kappa \cdot \alpha)$ *client time;*
  - $O(n)$ *communication;*

- ***Query Phase:***
  - $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha)$ *expected client time and* $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha)$ *server time per query;*
  - $O_\lambda(n^{1/4}\log\kappa\cdot\alpha)$ *communication per query.*

Therefore, the amortized online communication per query is $O_\lambda(n^{1/4}\log\kappa\cdot\alpha)$*, the amortized offline communication per query is* $O(\sqrt{n})$*, the amortized client and server computation per query is* $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha)$*.*

Proof. We defer the privacy and the correctness proof to Section 5.2 and Section 5.3 respectively. The client only stores $M_1 = 2\sqrt{n}\log\kappa\alpha$ λ-bit keys and store in total $\sqrt{n}\cdot M_2 = 3\sqrt{n}\log\kappa\cdot\alpha$ index-value pairs. So the storage is $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha)$. The pre-processing phase's performance bounds follow straightforward by the algorithm descriptions.

For the query phase, the client first enumerates all M_1 hints to find x, which takes $O_\lambda(\sqrt{n}\log\kappa\cdot\alpha)$ time. For all the $M_3 = \Theta(\log\kappa\cdot\alpha)$ found hints, the $\mathsf{Program}$ algorithm takes $O_\lambda(n^{1/4})$ client computation. For the server, during the query phase, the client sends M_3 programmed PPPS key to the server, the size of which is $O_\lambda(n^{1/4})$. The server sends back the xor-sum of $n^{1/4}$ candidate sets for each key after running $\mathsf{ListDecode}$. Even though each candidate set has size $n^{1/2}$, all the candidate sets have a succinct representation of size $n^{1/2}$ and server can compute this representation in time $O_\lambda(n^{1/2})$. Further, as observed in Theorem 4.1, due to the structure of the candidate sets, the sever can compute all the $n^{1/4}$ xor-sums in time only $O(n^{1/2})$. Hence, the server's running time $O_\lambda(\sqrt{n})$.

The client needs to find M_3 matched hints, compute $O(\log\kappa\cdot\alpha)$ xor operations during each query and for the matched hints, it needs to sample fresh PPPS key $\mathsf{sk}_{\mathsf{new}}$ subject to $x \in \mathsf{Set}(\mathsf{sk}_{\mathsf{new}})$. The client's computation time is dominated by this sampling step. We consider the expected computation time: each key in the hint table will have $1/\sqrt{n}$ probability to be replaced in this and each sampling takes $O_\lambda(\sqrt{n})$ expected time to finish using Lemma 5.2 and pseudorandomness of PPPS, the expected number of keys checked until a key sk such that $\mathsf{Set}(\mathsf{sk})$ contains the current query is found is $O(\sqrt{n})$. So the total expected time for the query phase is $O_\lambda(\sqrt{n}\log\kappa\alpha)$ per query. The server time is $O_\lambda(\sqrt{n}(\log\kappa\cdot\alpha))$ per query. The online communication per query is $O_\lambda(n^{1/4}(\log\kappa\cdot\alpha))$. ■

5.2 Privacy Proof

Suppose that the underlying PPPS scheme satisfies private programmability. Below, we prove the privacy of our single-server PIR scheme.

In the pre-processing phase, the server observes a single scan over the database, and thus no information is leaked. The rest of the proof will therefore focus on the query phase.

Lemma 5.2. *Recall that in each time step* t*, the adversary* $\mathcal{A}$ *adaptively chooses a query* $x_t \in \{0, 1, \ldots, n-1\}$ *for the client. At the end of each time step* t*, the client's hint table is distributed as a table of size* M_1 *where each entry is a freshly sampled PPPS key, even when conditioned on* $\mathcal{A}$*'s view so far.*

Proof. Suppose the above statement holds at the end of time step $t-1$. We prove that it still holds at the end of time step t. Since the hint table is distributed as a fresh randomly sampled table even when conditioned on $\mathcal{A}$'s view at the end of $t-1$, we may henceforth assume an arbitrary fixed query x_t. The distribution of the hint table before the t-th query can be equivalently rewritten as:

- First, sample the indices of the entries (henceforth denoted I) that contain the query x_t. Specifically, each $i \in [M_1]$ is chosen into the set I independently with probability $1/\sqrt{n}$.
- For each $i \notin I$, sample a random PPPS key subject to not containing x_t.
- For each $i \in I$, sample a random PPPS key subject to containing x_t.

Using the above interpretation, it is easy to see that the distribution of the hint table after the t-th query is unaltered. ■

Simulator construction and the Ideal experiment. Consider the following simulator construction which does not make use of the queries: in every time step t, call the simulator Sim of the PPPS scheme, and let the output be sk'. Send sk' to the server.

Indistinguishability of Real and Ideal. We now prove the indistinguishability of the Real and Ideal assuming the private programmability of the underlying PPPS scheme.

First, due to Theorem 5.2, we can equivalently rewrite the Real experiment as follows: at the end of each time step, resample the entire hint table freshly at random before continuing to answer more queries. As a result, the messages sent to $\mathcal{A}$ in each time step t is distributed as

Repeat M_3 times:

- sample a PPPS key sk at random subject to containing the query x_t; sample δ at random from $\{0, \ldots, \sqrt{n} - 1\}$.
- call $\mathsf{sk}', _ \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x_t), \delta)$;
- send sk' to $\mathcal{A}$.

One way to see this is to think of the distribution of the table as the equivalent distribution in the proof of Theorem 5.2.

Therefore, the rest of the proof follows due to a straightforward hybrid argument where we replace the programmed keys (denoted sk' earlier) sent to the server in all time steps one by one with a simulated key, relying on the private programmability of the underlying PPPS.

5.3 Correctness Proof

For the correctness analysis, we may assume that every set is sampled independently from $\mathcal{D}_n$. Due to the pseudorandomness property of the PPPS, this will only affect the correctness probability by a negligible amount.

Recall that we have a window of $Q = \sqrt{n}/2$ random, distinct queries. There are only two bad events that can cause correct failure: 1) the client cannot find a good hint that contains the query index; 2) the client runs out of replacements in a chunk.

We first analyze the second bad event, i.e., depleting replacement entries For every query, at most one replacement entry is consumed. Therefore, the second bad event only happens when the client makes more than M_2 queries in one chunk. The analysis is the same as analysis of depleting replacement entries in the proof of correctness for our 2-server scheme (see Section 4.3).

The first bad event, i.e., no matched good hint, can only arise from the following events: 1) there are no good hints left that match the query; 2) there are more than M_3 matched hints but the first M_3 matched hints are all broken.

Fix a sequence of query $x_1, \ldots, x_m$ and consider the error probability for x_m. Consider the initial hint table in which each entry represents a random set sampled from $\mathcal{D}_n$. If a hint in the

initial hint table contains x_m and does not contain $x_1, \ldots, x_{m-1}$, this hint will remain good until the query for x_m. We have that, for any hint

$$\Pr[\text{ the hint contains } x_m] = 1/\sqrt{n} \; .$$

Further, conditioned on a hint containing x_m, if x_i and x_m are not in the same chunk, then the hint contains x_i with probability $1/\sqrt{n}$ because each chunk has its own randomness. Moreover, if x_i and x_m are in the same chunk, this hint definitely will not contain x_i. Hence, we have that

$$\begin{aligned} &\Pr\left[\text{ hint is broken} \;\middle|\; \text{the hint contains } x_m\right] \\ \leq &\Pr\left[\exists i \in \{1, \ldots, m-1\} \text{ hint contains } x_i \;\middle|\; \text{the hint contains } x_m\right] \\ \leq &(m-1)/\sqrt{n} \leq Q/\sqrt{n} \leq 1/2 \; . \end{aligned}$$

Therefore,

$$\begin{aligned} &\Pr[\text{ hint is good and contains } x_m] \\ \geq &\Pr\left[\text{ hint is good} \;\middle|\; \text{the hint contains } x_m\right] \cdot \Pr[\text{ the hint contains } x_m] \geq 1/(2\sqrt{n}) \; . \end{aligned}$$

Then, the probability that there is no good hint matching x_m in the table is at most

$$\begin{aligned} &\left(1 - \frac{1}{2\sqrt{n}}\right)^{M_1} = \left(1 - \frac{1}{2\sqrt{n}}\right)^{2\sqrt{n} \ln \kappa \cdot \alpha(\kappa)} \\ \leq &(1/e)^{\ln \kappa \alpha(\kappa)} = \kappa^{-\alpha(\kappa)}. \end{aligned}$$

Now let us argue that for query x_m, the probability of more than M_3 hints being matched is small. Due to the Lemma 5.2, we may assume that at the beginning of each query, the hint table contains freshly and independently chosen sets. Each set sampled from $\mathcal{D}_n$ contains the query with probability $1/\sqrt{n}$. The expected number of hints that match the query is therefore $M_1/\sqrt{n} = 2 \log \kappa \cdot \alpha$. Using the Chernoff bound, we have that the probability of more than more than $M_3 = 3 \log \kappa \cdot \alpha$ hints being matched is bounded by a negligibly small function in κ.

Finally, we can apply a union bound over all $\sqrt{n}/2$ queries and conclude that the probability of the first bad event (i.e., no matched good hint) ever happening is negligibly small in κ.

6 Additional Related Works

Now we review some additional related works.

PIR with linear computation cost. The classical single-server PIR setting does not have preprocessing and the server is required to store the database without further encodings. Beimel, Ishai and Malkin [BIM00] proved that the per-query time of any PIR scheme in the classical model has to be $\Omega(n)$. Early theoretical works [CMS99, Cha04, GR05] improve the communication cost of PIR to polylogarithmic. Some later works [MBFK16, AYA+21, MW22] focus on improving the concrete computation time, mainly by reducing the number of public key cryptographic operations.

Within the scope of linear-computation cost PIR, recent schemes [PPY18, MCR21, DPC22, HHCG+22] also utilize preprocessing to improve the concrete performance. In particular, SimplePIR [HHCG+22] achieves the fastest computation time and claims that the online query time in their implementation is already limited by the server's memory I/O speed, instead of the computation.

Nonetheless, all these schemes have linear online time per query, which hinders them from being practical when scaling to large databases.

Batch PIR. In batch PIR schemes [IKOS04, AS16, ACLS18, MR22, LLWR22], the client is assumed to have a Q-size batch of queries simultaneously. The client can make a batched query and the server's computation cost for the whole batch is still $O(n)$, while the amortized computation cost for each query is reduced to $\widetilde{O}(n/Q)$. The main drawback of batch PIR schemes is that the client cannot make adaptive queries. Also, batch PIR schemes usually need some form of homomorphic encryption evaluation on the whole database, making the query latency significantly large. See the comparison in [HHCG+22].

Preprocessing PIR with sublinear computation cost. Biemel, Ishai and Malkin [BIM00] introduced PIR with preprocessing in order to circumvent the $\Omega(n)$ lower-bound on server side computation. They showed that the servers could compute polynomially many bits offline that reduces the online computation for the servers. Specifically, for any $\epsilon > 0, k \geq 2$, they gave a k-server protocol that has $O(n^{1/2k-1})$ communication and $O(n/(\epsilon \log n)^{2k-2})$ online server computation, and needs $O(n^{1+\epsilon})$ extra server storage. In order to reduce the online server computation further, they gave a second construction which for any $\epsilon > 0$ achieves $O(n^{1/2+\epsilon})$ online server computation and communication, and requires $O(n^{1+\epsilon'})$ extra server storage, where ϵ' depends on ϵ. While the second construction reduces online server computation significantly, the extra server storage is concretely very large, e.g., to achieve $n^{0.6}$ server side computation, the server must store roughly $n^{3.2}$ extra bits. In practice, this kind of overhead is prohibitively large.

Our scheme can be categorized as the "client-preprocessing" model, where the client will store some database-dependent hints and use those hints to make efficient online query. In fact, most of the recent schemes follow the same design framework proposed by Corrigan-Gibbs and Kogan [CK20]: the client will store many sets and their corresponding parities as hints and later on the client will find a local set S that contains the online query x, then make a carefully-designed query to learn the parity of $S/\{x\}$. The initial scheme [CK20] only supports one query, and subsequent works extend the idea to support multiple queries. In the two-server setting, Shi et al. [SACM21], Checklist [KCG21] and TreePIR [LP23] improve the communication cost to $\widetilde{O}_\lambda(1)$ while the amortized computation cost is $\widetilde{O}_\lambda(\sqrt{n})$. TreePIR is also concretely efficient. In the single-server setting, Corrigan-Gibbs, Henzinger and Kogan [CHK22] provide a scheme that has $\widetilde{O}_\lambda(\sqrt{n})$ amortized computation and communication cost using FHE. Zhou et al. [ZLTS23] and Lazzaretti and Papamanthou [LP22] further improve the communication cost to $\widetilde{O}_\lambda(1)$, relying on privately puncturale PRF. We introduce a single-server scheme that does not rely on the heavy cryptographic primitives, i.e., privately puncturable PRF in Appendix A.

All these aforementioned schemes require cryptographic assumptions that imply PKC, such as DDH and LWE. Recnetly, Piano [ZPSZ24] and a subsequent work from Mughees, I, and Ren [MIR23] introduce concretely efficient schemes based on the OWF assumption, while achieving optimal client-storage and online communication tradeoff (both are $\widetilde{O}_\lambda(\sqrt{n})$) and $\widetilde{O}_\lambda(\sqrt{n})$ amortized communication cost per query. Our work preserves the optimal tradeoff and improves the online communication cost to $\widetilde{O}_\lambda(n^{1/4})$.

Doubly Efficient PIR (DEPIR), also known as the "server-proprocessing model", requires the server first preprocesses the database and store additional bits, and subsequently, the server may answer queries with sublinear computation time. Earlier DEPIR schemes [CHR17, BIPW17] rely on non-standard assumptions such as oblivious locally decodable code (OLDC) or virtual-blackbox obfuscation (VBB). Recently, a breakthrough work from Lin, Mook and Wichs [LMW23] constructs a DEPIR scheme based on the standard Ring-LWE assumption.

References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*, 2018.

[AS16] Sebastian Angel and Srinath TV Setty. Unobservable communication over fully untrusted infrastructure. In *OSDI*, volume 16, pages 551–569, 2016.

[AYA+21] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI2021, July 14-16, 2021*, 2021.

[BFG03] Richard Beigel, Lance Fortnow, and William I. Gasarch. A nearly tight bound for private information retrieval protocols. *Electronic Colloquium on Computational Complexity (ECCC)*, 2003.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *CRYPTO*, pages 55–73, 2000.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *TCC*, 2017.

[BKM17] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, pages 415–445, 2017.

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In *PKC*, 2017.

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private constrained prfs (and more) from LWE. In *TCC*, 2017.

[CC17] Ran Canetti and Yilei Chen. Constraint-hiding constrained PRFs for NC^1 from LWE. In *EUROCRYPT*, pages 446–476, 2017.

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval. In *STOC*, 1997.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[Cha04] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.

[CHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Eurocrypt*, 2022.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In *TCC*, 2017.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999.

[Con] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/.

[DCMO00] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In *EUROCRYPT*, 2000.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-server pir with subpolynomial communication. *J. ACM*, 63(4), 2016.

[DGI+19] Nico Döttling, Sanjam Garg, Yuval Ishai, Giulio Malavolta, Tamer Mour, and Rafail Ostrovsky. Trapdoor hash functions and their applications. In *Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III*, 2019.

[DPC22] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/981, 2022. https://eprint.iacr.org/2022/981.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018.

[Fea] Nick Feamster. Oblivious DNS deployed by Cloudflare and Apple. https://medium.com/noise-lab/oblivious-dns-deployed-by-cloudflare-and-apple-1522ccf53cab.

[Gas04] William I. Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.

[HDCG+23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, , and Nickolai Zeldovich. Private web search with Tiptoe. In *29th ACM Symposium on Operating Systems Principles (SOSP)*, Koblenz, Germany, October 2023.

[HHCG+22] 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. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[HMR12a] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, pages 1–13. Springer, 2012.

[HMR12b] Viet Tung Hoang, Ben Morris, and Phillip Rogaway. An enciphering scheme based on a card shuffle. In *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2012.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[KCG21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, August 2021.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, 1997.

[Lip09] Helger Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC*, 2009.

[LLWR22] Jian Liu, Jingyu Li, Di Wu, and Kui Ren. Pirana: Faster (multi-query) pir via constant-weight codes. Cryptology ePrint Archive, Paper 2022/1401, 2022. https://eprint.iacr.org/2022/1401.

[LMW23] Wei-Kai Lin, Ethan Mook, and Daniel Wichs. Doubly efficient private information retrieval and fully homomorphic ram computation from ring lwe. In *STOC*, 2023.

[LP22] Arthur Lazzaretti and Charalampos Papamanthou. Single server pir with sublinear amortized time and polylogarithmic bandwidth. Cryptology ePrint Archive, Paper 2022/830, 2022. https://eprint.iacr.org/2022/830.

[LP23] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. In *CRYPTO*, 2023.

[MBFK16] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. Xpir: Private information retrieval for everyone. *Proceedings on Privacy Enhancing Technologies*, pages 155–174, 2016.

[MCG+08] Carlos Aguilar Melchor, Benoit Crespin, Philippe Gaborit, Vincent Jolivet, and Pierre Rousseau. High-speed private information retrieval computation on GPU. In *Proceedings of the 2008 Second International Conference on Emerging Security Information, Systems and Technologies*, SECURWARE '08, pages 263–272, Washington, DC, USA, 2008. IEEE Computer Society.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server pir. In *CCS*. Association for Computing Machinery, 2021.

[MG07] Carlos Aguilar Melchor and Philippe Gaborit. A lattice-based computationally-efficient private information retrieval protocol. *IACR Cryptology ePrint Archive*, 2007:446, 2007.

[MIR23] Muhammad Haris Mughees, Sun I, and Ling Ren. Simple and practical amortized sublinear private information retrieval. Cryptology ePrint Archive, Paper 2023/1072, 2023.

[MR22] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. Cryptology ePrint Archive, Paper 2022/1262, 2022. https://eprint.iacr.org/2022/1262.

[MW22] Samir Jordan Menon and David J. Wu. Spiral: Fast, high-rate single-server PIR via FHE composition. In *IEEE S&P*, 2022.

[OG11] Femi G. Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography*, pages 158–172, 2011.

[OS07] Rafail Ostrovsky and William E. Skeith, III. A survey of single-database private information retrieval: techniques and applications. In *PKC*, pages 393–411, 2007.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2018.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[RY13] Thomas Ristenpart and Scott Yilek. The mix-and-cut shuffle: small-domain encryption secure against n queries. In *Advances in Cryptology–CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, pages 392–409. Springer, 2013.

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In *CRYPTO*, 2021.

[SC07] Radu Sion and Bogdan Carbunar. On the computational practicality of private information retrieval. In *Network and Distributed Systems Security Symposium (NDSS)*, 2007.

[Yeo23] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. *arXiv preprint arXiv:2306.11220*, 2023.

[ZLTS23] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal single-server private information retrieval. In *EUROCRYPT*, 2023.

[ZPSZ24] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server pir with sublinear server computation. In *IEEE S& P*, 2024.

Supplementary Materials

A Sublinear Preprocessing PIR with $\widetilde{O}(1)$ Online Communication from Stronger Assumptions

Previously, we focused on designing preprocessing PIR schemes that rely only on one way functions (OWF). In this section, we show that the "broken hint" technique used in our one-server construction Section 5 can also be applied to the state-of-the-art two-server scheme, TreePIR [LP23]. This gives us a single-server scheme with $\widetilde{O}_\lambda(1)$ online communication cost, $O(\sqrt{n})$ offline communication, and $\widetilde{O}_\lambda(\sqrt{n})$ computation per query, assuming the existence of a classical single-server PIR scheme with polylogarithmic bandwidth.

A.1 Privately Puncturable Pseudorandom Set with List Decoding

The elegant TreePIR work [LP23] constructs a Privately Puncturable Pseudorandom Set with List Decoding (henceforth denoted PPPS^-). Specifically, their implied PPPS^- also emulates the same set distribution $\mathcal{D}_n$ which we inherit in our paper (see Section 3.1), and it supports the following operations:

- $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$: takes in the security parameter 1^λ, the size of the universe n, and outputs a secret key sk representing a set.
- $S \leftarrow \mathsf{Set}(\mathsf{sk})$: takes in a secret key sk and outputs a set S of size $\sqrt{n}$.
- $\mathsf{sk}', i \leftarrow \mathsf{Puncture}(\mathsf{sk}, \ell)$: takes in a secret key sk, a chunk index ℓ, and outputs a punctured key sk' which removes the element in the ℓ-th chunk from the set, as well as auxiliary information i that indicates which of the list decoded answers later is the correct answer.
- $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$: takes in a punctured key sk', and outputs a set of candidate sets $S_0, \ldots, S_{L-1}$. It is guaranteed that one of them is the correctly punctured set.

A PPPS^- scheme needs to satisfy the following properties:

1. **Correctness.** For any λ, n, any $\ell \in \{0, \ldots, \sqrt{n} - 1\}$, the following must hold with probability 1: let $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$, $\mathsf{sk}', i \leftarrow \mathsf{Puncture}(\mathsf{sk}, \ell)$, $S_0, \ldots, S_{L-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, it must be that S_i is equal to $\mathsf{Set}(\mathsf{sk})$ but with the ℓ-th element removed.
2. **Pseudorandomness.** Defined in the same way as in Section 3.1.
3. **Private puncturability.** There exists a probabilistic polynomial-time simulator Sim such that for any n that is polynomially bounded in λ, for any $x \in \{0, \ldots, n - 1\}$, the following two distributions are computationally indistinguishable:
   - Real: Sample $\mathsf{sk} \leftarrow \mathsf{Gen}(1^\lambda, n)$ subject to $x \in \mathsf{Set}(\mathsf{sk})$, let $\mathsf{sk}', _ \leftarrow \mathsf{Puncture}(\mathsf{sk}, \mathsf{chunk}(x))$, output sk'.
   - Ideal: Let $\mathsf{sk}' \leftarrow \mathsf{Sim}(1^\lambda, n)$, output sk'.

TreePIR [LP23] implies a PPPS^- scheme constructed from PRFs, satisfying not only the above properties but also the following efficiency requirements:

- *Fast membership*: testing whether an element $x \in \{0, \ldots, n - 1\}$ is in the set or not takes $O_\lambda(1)$ time.

Single-Server Scheme for $Q = \sqrt{n}/2$ Queries

Notation. κ denotes a *statistical* security parameter, λ denotes a computational security parameter. We use $\alpha(\kappa)$ to denote an arbitrarily small super-constant function.

Preprocessing. Same as in Figure 4 but no more need for replacement entries.

Query for index $x \in \{0, 1, \ldots, n-1\}$.

1. For each entry (sk_i, p_i) in the client's hint table, if $x \in \mathsf{Set}(\mathsf{sk}_i)$ (henceforth called a matched entry), do the following unless there are already $M_3 = 3 \log \kappa \cdot \alpha$ matched entries:
   - Let $\mathsf{sk}', i^* \leftarrow \mathsf{Puncture}(\mathsf{sk}_i, \mathsf{chunk}(x))$. Send sk' to the server.
   - Server calls $S_0, \ldots, S_{\sqrt{n}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$, and for each $i \in \{0, \ldots, \sqrt{n}-1\}$, it computes $\beta_i := \oplus_{j \in S_i} \mathsf{DB}[j]$.
   - Client and server run a classical PIR scheme on the database $(\beta_0, \ldots, \beta_{\sqrt{n}-1})$, and the client retrieves β_{i^*}.
   - If the matched entry is good, client saves the answer $p_i \oplus \beta_{i^*}$.
   - Client samples a fresh PPPS^- key sk_{new} subject to $x \in \mathsf{Set}(\mathsf{sk})$, and replaces the consumed entry (sk_i, p_i) with $(\mathsf{sk}_{\text{new}}, 0)$ and mark the entry as *broken*.
2. Let cnt be the number of matched entries in the previous step. If $\mathsf{cnt} < M_3$, then client repeats the following $M_3 - \mathsf{cnt}$ times: sample a random PPPS^- key sk subject to $x \in \mathsf{Set}(\mathsf{sk})$, call $\mathsf{sk}', _ \leftarrow \mathsf{Puncture}(\mathsf{sk}, \mathsf{chunk}(x))$, and send sk' to the server. Invoke a classical PIR scheme with the server to retrieve any index in $\{0, \ldots, \sqrt{n}-1\}$, and ignore the answer received.
3. Client outputs any saved answer. If no answer was saved, output 0.

Figure 5: Our single-server pre-processing PIR scheme with $\widetilde{O}(1)$ online bandwidth from a classical PIR scheme.

- *Small punctured key*: a punctured key has size $\widetilde{O}_\lambda(1)$.
- *Efficient list decoding*: Although $\mathsf{ListDecode}$ outputs $\sqrt{n}$ candidate sets, all $\sqrt{n}$ candidate sets has a succinct representation of size $O(\sqrt{n})$; moreover, the $\mathsf{ListDecode}$ algorithm runs in $O_\lambda(\sqrt{n})$ time.

A.2 A Single-Server PIR Scheme with Polylogarithmic Online Communication

Given a PPPS^- scheme with the aforementioned security and efficiency requirements, we can construct a single-server pre-processing PIR scheme as in Figure 5.

We can prove the following theorem about this construction.

Theorem A.1. *Assume the existence of a classical PIR scheme with linear computation, storage and polylogarithmic communication. We can construct a PIR scheme supporting $\sqrt{n}/2$ queries that is private and achieves the following performance bounds:*

- $O_\lambda(\sqrt{n} \log \kappa \alpha(\kappa))$ *client storage and no additional server storage;*
- ***Pre-processing Phase:***
  - $O_\lambda(n \log \kappa \cdot \alpha)$ *client time and* $O_\lambda(n \log \kappa \cdot \alpha)$ *server time;*

- $O(n)$ *communication;*

- ***Query Phase:***
  - $\widetilde{O}_\lambda(\sqrt{n})$ *expected client time and server time per query;*
  - $\widetilde{O}_\lambda(1)$ *communication per query.*

Further, assuming that n is bounded by $\mathsf{poly}(\lambda)$ *and* $\mathsf{poly}(\kappa)$*, all the $Q = \sqrt{n}/2$ queries in the scheme in Section A will be answered correctly with probability at least* $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ *for some negligible function* $\mathsf{negl}(\cdot)$*.*

Proof. The proofs about efficiency is nearly the same as the proof of Theorem 5.1 – the main differences being that the PPPS^- replaces the PPPS here and that the classical PIR scheme that is run during the query phase needs to be taken into account for the storage, computation and communication. Since the storage and computation of the classical scheme is linear, and the communication polylogarithmic, the overall complexities do not change asymptotically. Also, we note in the context of the PPPS^- scheme that the parities of the $\sqrt{n}$ sets can be computed by the server in $O(\sqrt{n})$ time as shown in [LP23].

The proof of correctness is the same as the analysis in Section 5.3, assuming correctness of the underlying classical PIR scheme. The privacy proof is also nearly the same as the privacy proof in Section 5.2 except the following couple changes: 1) the PIR's simulator additionally calls the underlying classical PIR's simulator; and 2) instead of calling the simulator of PPPS scheme, the PIR's simulator now calls the simulator of the PPPS^- scheme. ■

B Removing the Repetition Overhead in the Single-Server Scheme

Since our earlier single-server scheme in Section 5 employs the broken hints technique, there is a superlogarithmic factor multiplicative overhead due to the repetition needed for correctness compared to the two-server scheme in Section 4. In this section, we describe an improved single-server pre-processing PIR scheme that avoids this super-logarithmic repetition overhead. For this optimized version, we cannot use the PPPS scheme in a completely blackbox manner.

Why previous schemes do not worry about broken hints. First, we need to understand why in the previous single-server schemes like Piano [ZPSZ24], the client does not have to worry about broken hints. Notice that in the two-server scheme, the client will replace the consumed hint with a resampled hint that contains the current query, and interact with another server to fetch the correct parity for this new hint. This ensures that all hints are useful. In the single-server setting, the client cannot rely on the additional server to dynamically fetch the parity for the resampled hint. To fix this, Piano requires the client to prepare polylogarithmic *backup* hints for each of the $\sqrt{n}$ chunks. The backup hints for the i-th chunk will record the parity of the set and also the value of the item in the i-th chunk. After querying for x, the client will replace the consumed hint with a backup hint from x's chunk and lazily mark that i-th element of the hint to be the current query x, which maintains the distribution of the hint table. Since the values for the original i-th element and x are known to the client, the correct parity for this edited hint can be derived by the client locally, avoiding the issue of broken hints. Then, the client only needs to find the first hint that contains x, instead of finding all matched hints and making multiple queries based on those hints, hoping that

one of them will be good. [1] Now, after the client expands the key to a whole set, the client can easily enforce the marking (if necessary) by just changing the corresponding element.

We will try to follow the backup hint and the lazy-marking idea. There is one challenge remaining: in PIANO, the client can expand the whole set and *possibly change two elements obliviously* before sending the set to the server. The client not only needs to change the offset in the current query's chunk to ensure the privacy, it also needs to change the offset in the lazy marked element's chunk to reflect the correct history. Our main construction in Section 3 only allows us to program one location. How should we modify the scheme to support programming at two points?

One way to achieve that is by privately programmable pseudorandom function (PPPRF). PPPRF allows us to directly program on the key. This is nearly the same idea in two previous single-server PIR schemes [ZLTS23, LP22]. Unfortuantely, this primitive is only known under strong cryptographic assumptions like LWE, and still only exists in theoretical literatures – it is completely impractical at the current stage.

Our solution: rejection sampling. We observe that all we need to do is just to "maintain" the right distribution of the hint table and the keys sent to the server. If we can monitor the change in the distribution, we can rejection-sample the PRF keys according to the right distribution and avoid the difficulty of doing private programming on the key level.

Maintaining the distribution for the local hint table is relatively simple and is already achieved by the lazy-marking technique. For the local hint table, what we actually care about is the distribution of the actual sets, regardless how we represent them. Assume that the client already queries for y and knows $\mathsf{DB}[y]$. Now the client consumes a hint that contains y, and replaces it with a hint from those backup hints prepared for $\mathsf{chunk}(y)$. Suppose the key for the backup hint is sk. $\mathsf{Set}(\mathsf{sk})$ may not contain y, but we can mark y alongside the key, such that whenever we need to do membership testing on this key, we will consider y is already programmed into the set. This maintains the local table's distribution.

The more tricky case is how we handle the keys that the client sends to the server. Suppose client is now querying for x, and it finds the first hint that contains x. Also, it notices that the hint is marked with y. Recall that in the PPPS programming algorithm, the client will expand the master key into $n^{1/4}$ keys and expand the keys corresponding to the superblock of x to $n^{1/4}$ offsets. Now there are two cases:

1. *x and y are in the same superblock.* This is a relatively simple case. As the normal programming step in the PPPS, we will replace the sub-key corresponding to the superblock of x (the same as y) with a uniformly random sub-key. However, when we expand the original sub-key to $n^{1/4}$ offsets, we need to replace x's offset with a random one, and also manually set the offset corresponding to y's chunk to y's offset, reflecting that we enforce y to be included. No rejection sampling is required in this case.

   For the answer, since we are just replacing the original element in y's chunk to y, the influence of such change should be corrected. We can require the client to store the database value for the i-th element for all the backup hints dedicated for the i-th chunk. Then, if this promoted hint sk is used, we can correct the answer by xoring the influence value, $\mathsf{DB}[\mathsf{Set}(\mathsf{sk})[\mathsf{chunk}(y)]] \oplus \mathsf{DB}[y]$.

2. *x and y are in the different superblocks.* This is the more involved case. In the programming, we will again replace the sub-keys corresponding to x's superblock with a uniformly random

[1] A natural question is why the client cannot simply pick the first good hint. Observe that it would be equivalent to deleting a consumed hint from the table because a broken hint will never be used again. Then, it would skew the distribution of the hint table because the remaining hints are less likely to contain the past queries, which leads to privacy leakage.

Augmented Query Protocol Based on a Single Hint[a]

Client's input:

- A master PPPS key sk assuming $x \in \mathsf{Set}(\mathsf{sk})$;
- The parity of $\mathsf{Set}(\mathsf{sk})$ is assumed to be p_{sk};
- A random index r in x's chunk such that the $\mathsf{DB}[r]$ is assumed to be known.

1. **Step 1: (Client)**
   (a) Send $(\mathsf{sk}', i) \leftarrow \mathsf{Program}(\mathsf{sk}, \mathsf{chunk}(x), r \bmod \sqrt{n})$ to the server;
2. **Query Step 2: (Server)** Upon receivin sk':
   (a) Parse sk' as
   $$(k_0, \ldots, k_{n^{1/4}-1}), (\delta_0, \ldots, \delta_{n^{1/4}-1}).$$
   (b) Let $S_0, \ldots, S_{n^{1/4}-1} \leftarrow \mathsf{ListDecode}(\mathsf{sk}')$;
   (c) For $i \in \{0, \ldots, n^{1/4} - 1\}$:
      i. Let $\beta_i = \oplus_{x \in S_i} \mathsf{DB}[x]$;
      ii. Expand k_i to $n^{1/4}$ offsets and consider them being the corresponding index offsets in chunk $\{i \cdot n^{1/4}, \ldots, (i+1) \cdot n^{1/4} - 1\}$. Compute the xor-sum of the database values for those indices as
      $$\alpha_i = \oplus_{j \in \{0, \ldots, n^{1/4}-1\}} \mathsf{DB}[(j + in^{1/4})\sqrt{n} + \mathsf{PRF}_2(k_i, j)].$$
   (d) Return $(\alpha_0, \ldots, \alpha_{n^{1/4}-1})$, $(\beta_0, \ldots, \beta_{n^{1/4}-1})$ to the client.
3. **Step 3: (Client)**
   Upon receiving $(\alpha_0, \ldots, \alpha_{n^{1/4}-1}), (\beta_0, \ldots, \beta_{n^{1/4}-1})$:
   - Save $(\alpha_0, \ldots, \alpha_{n^{1/4}} - 1)$ if the answer needs to be corrected later.
   - Return $p_{\mathsf{sk}} \oplus \beta_i \oplus \mathsf{DB}[r]$ as the answer.

[a] The augmented part is highlighted in red.

Figure 6: The augmented query protocol based on a single hint. The augmentation is mainly about returning the the parities of those superblocks corresponding to the $n^{1/4}$ sub-keys and will be used in Figure 7.

Optimized Single-Server Scheme for $Q = \sqrt{n}/2$ Queries [a]

Notation. κ denotes a *statistical* security parameter, λ denotes a computational security parameter. We use $\alpha(\kappa)$ to denote an arbitrarily small super-constant function.

Preprocessing.

- In addition to the previous algorithm:
  - Client samples $M_2 = 3 \log \kappa\alpha(\kappa)$ backup keys $\mathsf{sk}_{i,1}, \ldots, \mathsf{sk}_{i,M_2}$ for chunk i;
  - For each backup hint, Client stores $\mathsf{sk}_{i,j}$ and the following information:
    - The parity for the whole set: $p_{i,j} = \oplus_{x \in \mathsf{Set}(\mathsf{sk}_{i,j})} \mathsf{DB}[x]$;
    - The parity for those items within the superblock that contains the i-th chunk: $b_{i,j} = \oplus_{x \in \mathsf{Set}(\mathsf{sk}_{i,j})} \mathbf{1}\left[\mathsf{superblock}(x) = \left\lfloor \frac{i}{n^{1/4}} \right\rfloor\right] \mathsf{DB}[x]$ [b]
    - The DB-value of the i-th item: $\mathsf{DB}[\mathsf{Set}(\mathsf{sk}_{i,j})[i]]$.

Query for index $x \in \{0, 1, \ldots, n-1\}$.

1. **Query:**
   (a) Client finds the first matched hint (sk_i, p_i) such that $x \in \mathsf{Set}(\mathsf{sk}_i)$ and the hint does not have a positive constraint y that $\mathsf{chunk}(y) = \mathsf{chunk}(x)$.
   (b) If there is no positive constraint on this hint: proceed the subroutine Figure 6 as usual.
   (c) If there is a positive constraint $+y$ on this hint:
      i. If $\mathsf{superblock}(y) = \mathsf{superblock}(x)$, execute the subroutine, except that
         - After the client programs the key and gets $n^{1/4}$ sub-keys and offsets, replaces the $(\mathsf{chunk}(y) \mod n^{1/4})$-th offset with y's offset;
         - Let the return value be v, mark the answer as $v \oplus \mathsf{DB}[y] \oplus \mathsf{DB}[\mathsf{Set}(\mathsf{sk}_i)[\mathsf{chunk}(y)]]$.
      ii. If $\mathsf{superblock}(y) \neq \mathsf{superblock}(x)$, execute the subroutine, except that:
         - After the client has the programmed key of $n^{1/4}$ sub-keys and offsets, replace the $\mathsf{superblock}(y)$-th key with a rejection-sampled key k'. The rejection sampling key k' should satisfy all constraints related to the matched hint $(+y, -z_1, \ldots, -z_t)$ if k' controls $\mathsf{superblock}(y)$ (as specified in Section B).
         - Let the return value be v, mark the answer as $v \oplus \alpha_{\mathsf{superblock}(y)} \oplus b$, where $\alpha_{\mathsf{superblock}(y)}$ is the corresponding parity of superblock controled by the rejection-sampled sub-key and b is the original parity for that superblock, known by preprocessing.

2. **Refresh:**
   - Client adds the constraint $-x$ for the first $i-1$ entries.
   - Client replaces the matched hint with an unconsumed hint from the $\mathsf{chunk}(x)$'s backup hint group. Clean all other constraints and only mark $+x$ for the i-th entry.

[a] For clarity, we present the scheme supporting distinct and random queries.
[b] $\mathbf{1}[A]$ is 1 when A is true and 0 otherwise.

Figure 7: The optimized sublinear single server preprocessing PIR protocol introduced in Section B. The protocol uses a subroutine defined in Fig. 6 as a sub-routine.

sub-key and replace the offsets in x's chunk with a random one. However, we need to ensure that the key corresponding to y's superblock will expand to the correct offset for y's chunk, to ensure that y is included. Now, we will rejection-sample a sub-key k for y's superblock, such that it expands to the correct offset of y. This is what we refer to as a **positive** constraint, and we write this constraint as $+y$. Additionally, to maintain the right distribution, there are also **negative** sampling constraints. Observe that if the i-th hint entry is the first one to contains x, it means that from the client's perspective, the first $i-1$ entries do not contain x. Suppose between the query y and query x, there are some other queries. After making those queries, the client also knows that the current hint entry does not contain some queries $z_1, \ldots, z_t$. We write down the negative constraints as $-z_1, \ldots, -z_t$. Then, when we do the rejection sampling for the sub-key for y's superblock, we will consider all the constraints: $+y, -z_1, \ldots, -z_t$, until we find a sub-key k that satisfies all the constraints. This ensures that the resampled sub-key for y's superblock has the right distribution.

We also need to argue that the rejection-sampling is efficient. The positive constraint is satisfied with probability $1/\sqrt{n}$. Conditioned on the positive constraint being satisfied, for each negative constraint, it is satisfied with probability at least $1 - 1/\sqrt{n}$. We only aim to support $Q = \sqrt{n}/2$ queries within one window, so there are no more than $\sqrt{n}/2$ negative constraints. By union bound, satisfying all negative constraints happens with probability at least $1/2$. Therefore, we conclude that a freshly sampled sub-key satisfies all the constraints with probability at least $1/2\sqrt{n}$, and the expected sample time is $O(\sqrt{n})$.

A noticable detail is that when we draw a new fresh key, we first check the positive constraint. Only when the positive constraint is satisfied, we check the negative constraints. The positive constraint is satisfied with probability $1/\sqrt{n}$ but only takes $O(1)$ time to check. Conditioned on the positive constraint being satisfied, the negative constraints take $O(\sqrt{n})$ to check but are satisfied with probability at least $1/2$. So even including the constraint checking time, the expected sampling time is still $O(\sqrt{n})$.

For the answer, since we are essentially replacing the original superblock to a rejection-sampled one. To remove the influence of this step, we can require the client to store the parity for the corresponding superblock controlling the i-th chunk for all the backup hints dedicated for the i-th chunk. Then, if this promoted hint sk is used, we can correct the answer by xor-ing the parity for the original superblock (known by preprocessing) and the parity for the rejection-sampled superblock. This is why we need an augmented version of the response protocol for the server Figure 6 – we need the server to tell the client about the parity of the rejection-sampled superblock.

We provide the pseudocode of this construction in Figure 7.

Theorem B.1. *Let $\alpha(\kappa)$ be any superconstant function. Assume n is bounded by $\mathsf{poly}(\lambda)$ and $\mathsf{poly}(\kappa)$. The PIR scheme in Figure 7 that supports $\sqrt{n}/2$ queries is private, correct with probability $1 - \mathsf{negl}(\lambda) - \mathsf{negl}(\kappa)$ and achieves the following performance bounds (the optimized parts are underlined):*

- $O_\lambda(\sqrt{n}\log\kappa\alpha(\kappa))$ *client storage and no additional server storage;*
- ***Pre-processing Phase:***
  - $O_\lambda(n\log\kappa\alpha(\kappa))$ *client time and* $O(n)$ *server time;*
  - $O(n)$ *communication;*
- ***Query Phase:***

– $\underline{O_\lambda(\sqrt{n})}$ *expected client time* and $\underline{O_\lambda(\sqrt{n})}$ *server time per query;*

– $\underline{O_\lambda(n^{1/4})}$ *communication per query.*

Therefore, the amortized online communication per query is $O_\lambda(n^{1/4})$*, the amortized offline communication per query is* $O(\sqrt{n})$*, the amortized client computation is* $O_\lambda(\sqrt{n}\log\kappa\alpha)$ *and the server computation is* $O_\lambda(\sqrt{n})$*.*

Proof. The correctness proof will be similar to the correctness proof for the main scheme described in Section 4.3.

We defer the privacy proof to later.

The additional storage for the client will be those M_2 backup hints per chunk, which results in total of $O(\lambda\sqrt{n}\log\kappa\alpha(\kappa))$ space. Notice that the client does not directly store the negative constraints, otherwise the storage will be blown up. The client can directly store the history of those matched hint entry index (which takes $O(\sqrt{n}\log n)$ space) and also the generation time label of each hint. Whenever the client finds the first matched hint, say entry i, it finds all the previous queries that are older than the current hint's generation time and whose matched hint entry indices are larger than i. Those history queries will be the negative constraints. By doing this, maintaining all the constraints only takes $O(\sqrt{n}\log n)$ space.

The pre-processing phase's performance bounds are obvious by the algorithm description.

For the query phase, the client first finds the first hint that contains x. Each hint contain x with probability $1/\sqrt{n}$, so the expected time is $O(\lambda\sqrt{n})$. The subroutine in Figure 6 takes $O(\lambda n^{1/4})$ client computation and $O(\lambda\sqrt{n})$ server computation, while consuming $O(\lambda n^{1/4})$ communication cost. The refresh phase takes only $O(1)$ time (if the client maintains the negative constraint using the technique mentioned above). So the expected client time is $O(\lambda\sqrt{n})$ per query. The server time is $O(\lambda\sqrt{n})$ per query. The online communication per query is $O(\lambda n^{1/4})$ per query. ■

B.1 Privacy Proof

Proof. We will prove this theorem via a sequence of hybrids. The first hybrid in the sequence will capture the real experiment in the privacy definition of single server PIR, and the last hybrid will capture the ideal experiment. We first define the hybrid Real^*:

- *Pre-processing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \stackrel{\$}{\leftarrow} \{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  – The client finds the first matched key sk_i in the hint table, that is, $x \in \mathsf{Set}(\mathsf{sk}_i)$ and if there's a positive constraint $+y$ on this entry, x is not in the same chunk as y.
  – If the entry i has no positive constraint: execute the subroutine.
  – If the entry i has a positive constraint $+y$:
    * If y and x are in the same superblock: execute the subroutine, with the difference that when the client is expanding the sub-key corresponding to the superblock of x to $n^{1/4}$ offsets, replace the offset of y's location to y's offset.
    * If y and x are not in the same superblock: execute the subroutine with the difference that when the client is expanding the master key to $n^{1/4}$ sub-keys, replace the sub-key corresponding to y's superblock to a rejection sampled key k', such that when k' is considered as the key for that superblock, the expanded set should satisfy all constraints related to entry i;

– Then, the client replaces the entry sk_i with a freshly-sampled key sk', but deletes other constraints and marks constraint $+x$ for entry i;
– The client also marks constraints $-x$ for entry 1 to entry $i-1$.

This hybrid is identically distributed as the actual experiment that we just remove all parts unrelated to the privacy proof.

We define Hyb_1 by removing PRF_1 from Real^*: for each sampled master key sk, the client will directly sample $n^{1/4}$ random sub-keys and store them in the local storage. When the client executes the programming step in the subroutine, it no longer has to expand the master key to $n^{1/4}$ sub-keys.

Hyb_1 is computationally indistinguishable from Real^* due to a straightforward reduction to the pseudorandomness of PRF_1.

Since we already replaces all master keys with $n^{1/4}$ sub-keys, the later hybrids no longer need to expand a master key to $n^{1/4}$ sub-keys in the subroutine.

We define Hyb_2 as following: Hyb_2 is the same as Hyb_1, except that whenever the client finds the matched hint for query x, it will resample all those $n^{1/4}$ sub-keys **according to all the marked negative constraints and also the new constraint** $+x$**.** Notice that the history-dependent positive constriaint (usually written as $+y$) is not considered in this new resample step and only enforced just before the client sending messages to the server.

Now we argue that Hyb_2 is identically distributed as Hyb_1.

Claim B.2. *The view of the adversary in* Hyb_2 *is identically distributed as in* Hyb_1*.*

Proof. For all Q queries, lets consider the matched hint index vector $\mathbf{I} = (i_1, \ldots, i_Q)$ where i_t denotes the client finds i_t as the matched hint in the t-th round. Notice that the recorded constraints at t-th round are completely determined by $i_1, \ldots, i_{t-1}$.

Now we add the matched index vector to the view of the adversarys, and we will argue that even with the augmented view, the view of the adversary are still identically distributed in the two experiments.

First, observe that the two experiments have exactly the same way to generate the matched index vector. We only need to argue that, the selected entry in each round has the same distribution in both experiments, and then the messages observed by the adversary should have the same distribution.

We claim that conditioned on the same $i_1, \ldots, i_{t-1}$, the selected entry in round t have the same distributions in the two experiments.

In Hyb_1, we can equivalently consider the client first uniformly sample the M_1 primary hints and extra $t-1$ replaced hints. Then, given the queries $x_1, \ldots, x_{t-1}$, the client finds the matched hint and hence derives $i_1, \ldots, i_{t-1}$ and those selected entry.

In Hyb_2, we can equivalently consider the client first observes the queries $x_1, \ldots, x_{t-1}$ and samples $i_1, \ldots, i_{t-1}$ from the marginal distribution (conditioned on $x_1, \ldots, x_{t-1}$ and all the initial M_1 hints and the $t-1$ replacement hints are random). Then, the client samples the selected entry for each round by deriving the constraints from $i_1, \ldots, i_{t-1}$ and does rejection sampling.

A key observation is that in Hyb_1, the adversary cannot observe what the hint table is before round t. Then, from the adversary's perspective, the selected entry's distribution is the *posterior* distribution of them after observing $i_1, \ldots, i_{t-1}$. Moreover, that *posterior* distribution is exactly the same as the distribution the client uses to generate the selected entry (and even the whole table) in round t (i.e., containing the query x_t and some negative constraints $-z_1, -z_2, \ldots$) in Hyb_2. Therefore, from the adversary's perspective, the selected hints in the two experiment share the same distribution, and thus the views are identically distributed.

■

We define Hyb_3 as following: Hyb_3 is the same as Hyb_2, except that whenever the client finds the matched hint for query x, it will resample that entry also considering the history-dependent positive constraint. Hyb_2 does not consider the history-dependent positive constraint because the history-dependent constraint is only marked locally and is only enforced before sending the message. That is, it will include the positive constraints of including the current query x and a history query y (if necessary) and all other negative constraints. We will prove that Hyb_3 is computationally indistinguishable from Hyb_2.

Claim B.3. Hyb_3 *is computationally indistinguishable from* Hyb_2*.*

Proof. For each query x_t, there are three cases for the selected hint entry: 1) it does not have a history-dependent positive constraint; 2) it has a history-dependent positive constraint $+y$, but y and x are not in the same superblock; 3) it has a history-dependent positive constraint $+y$ such that y and x are in the same superblock.

The first case is trivially identical in the two experiments. For the second case, since the client will always resample the sub-key for y's superblock before sending messages to the server, it does not matter whether the original hint contains y or not (which only affect the sub-key for y's superblock). The final case is that the current query x and the positive constraint y are in the same superblock. We can view Hyb_2 and Hyb_3 be the case where we sample the sub-key in that particular superblock according to the same constraints, except that Hyb_3 will have one additional constraint that the offset in y's chunk should be y's offset. Then the client will expand the sampled sub-key to $n^{1/4}$ offsets and change the offset in y's chunk to y's offset. Due to the pseudorandomness of PRF_2, the two distributions are computationally indistinguishable.

■

Now, define Hyb_4 the same as Hyb_3, except that in the query phase, after the client resample the selected hint entry, the client will not do the processing steps related to the history-dependent positive constraint.

For clarity, we write the full definition of Hyb_4:

- *Pre-processing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \xleftarrow{\$} \{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client finds the first matched keys sk_i in the hint table, that is, $x \in \mathsf{Set}(\mathsf{sk}_i)$ and if there's a positive constraint $+y$ on this entry, x are not in the same chunk as y.
  - Resample a key sk subject to all the constraints recorded for this entry (e.g. including the current query x and the history query y, while satisfying some negative constraints $-z_1, \ldots$).
  - Execute the subroutine based on this resampled key.
  - Then, the client replaces the entry sk_i with a freshly-sampled key sk', but deletes other constraints and marks constraint $+x$ for entry i;
  - The client also marks constraints $-x$ for entry 1 to entry $i-1$.

Hyb_4 is identically distributed as Hyb_3. There are only two different cases in Hyb_4 and Hyb_3:

- The history-dependent positive constraint y is in the same superblock as the current query x. Notice that the client already resamples the sub-key subject to y is included, so the offset in $\mathsf{chunk}(y)$ is already y's offset;

- The history-dependent positive constraint y are not in the same superblock as the current query x. In Hyb_3, we already have another step to resample the sub-key correponding to y's superblock. However, notice that the two resampling steps are independent and subject to exactly the same constraints, so removing the latter one preserves the distribution.

Now we define Hyb_5 as following:

- *Pre-processing phase.* $\mathcal{A}$ receives the streaming signal. The client samples $\mathsf{sk}_1, \ldots, \mathsf{sk}_{M_1} \xleftarrow{\$} \{0,1\}^\lambda$.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client finds the first matched keys sk_i in the hint table, that is, $x \in \mathsf{Set}(\mathsf{sk}_i)$.
  - Execute the subroutine with this matched key.
  - Then, the client replaces the entry sk_i with a freshly-sampled key sk' subject to $x \in \mathsf{Set}(\mathsf{sk}')$.

Notice that in Hyb_5, the client does not record any constraint and does not resample the matched hint.

Claim B.4. *The adversary's views in* Hyb_4 *and* Hyb_5 *is computatonally indistinguishable.*

Proof. This argument is similar to the argument for Claim B.2. We can still consider adding the matched index vector $\mathbf{I}$ to the adversary's views and prove the augmented views are indistinguishable.

We first argue that the distributions of $\mathbf{I}$ are computationally indistinguishable in the two experiments. Observe that given any time, if we expand all the local hints to sets, the distributions of those sets are computationally indistinguishable. The only difference between the two experiments is that in Hyb_4, the client replaces the matched hint with a uniformly random new hint and lazily marks the current query x to the entry in Hyb_4, while in Hyb_5, the client directly does rejection-sampling to sample a new hint containing x. These two sets are computationally indistinguishable due to the pseudorandomness of the PRF_2.

Conditioned on the same matched indices $i_1, \ldots, i_t$, we only need to argue the selected entry in round t has the same distribution in both experiments, and then the messages observed by the adversary should have the same distribution.

In Hyb_4, we can equivalently consider the client first observes the queries $x_1, \ldots, x_{t-1}$ and sample $i_1, \ldots, i_t$ from the marginal distribution (conditioned on $x_1, \ldots, x_t$, those initial M_1 hints are uniformly random, and the extra $t-1$ replacement hints are sampled conditioned on containing the corresponding queries). Then, the client samples the selected entry for this round by deriving the constraints from $i_1, \ldots, i_t$ and does rejection sampling.

In Hyb_5, we can equivalently consider the client first generates M_1 primary hints and also the $t-1$ replacement hints given the queries $x_1, \ldots, x_{t-1}$. Then, the client finds the matched hint in each round and hence derives $i_1, \ldots, i_t$ and chooses those selected entries.

Again, the key observation is that in Hyb_5, the adversary cannot observe what the hint table is before round t. Then, from the adversary's perspective, the whole table's distribution is the *posterior* distribution after observing $i_1, \ldots, i_t$. Moreover, that *posterior* distribution is exactly the same as the distribution the client recorded as the form of constraints.

Therefore, from the adversary's perspective, the selected entries in the two experiment share the same distribution conditioned on the same matched indice $i_1, \ldots, i_t$. So we can conclude that the adversary's view in the two experiments are computationally indistinguishable. ■

Finally, we define the hybrid Ideal:

- *Pre-processing phase.* $\mathcal{A}$ receives the streaming signal.
- *Query phase.* For each round t, $\mathcal{A}$ chooses a query x_t:
  - The client will
    * Independently sample $k_0, \ldots, k_{n^{1/4}-1} \xleftarrow{\$} \{0,1\}^\lambda$;
    * Independently sample $r_0, \ldots, r_{n^{1/4}-1} \xleftarrow{\$} \{0, \ldots, \sqrt{n}-1\}$.
    * Send $(k_0, \ldots, k_{n^{1/4}-1}), (r_0, \ldots, r_{n^{1/4}-1})$ to the server.

The argument that Ideal and Hyb_5 are computationally indistinguishable is nearly the same as the proof in Section 5.2 ■