Optimal Single-Server Private Information Retrieval

Mingxun Zhou[1] Wei-Kai Lin[2] Yiannis Tselekounis[1] Elaine Shi[1*]

[1]Carnegie Mellon University
[2]Northeaster University

Abstract

We construct a single-server pre-processing Private Information Retrieval (PIR) scheme with optimal bandwidth and server computation (up to poly-logarithmic factors), assuming hardness of the Learning With Errors (LWE) problem. Our scheme achieves amortized $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation and $\widetilde{O}_\lambda(1)$ bandwidth per query, completes in a single roundtrip, and requires $\widetilde{O}_\lambda(\sqrt{n})$ client storage. In particular, we achieve a significant reduction in bandwidth over the state-of-the-art scheme by Corrigan-Gibbs, Henzinger, and Kogan (Eurocrypt'22): their scheme requires as much as $\widetilde{O}_\lambda(\sqrt{n})$ bandwidth per query, with comparable computational and storage overhead as ours.

1 Introduction

Imagine that a server holds a large public database DB indexed by $0, 1, \ldots, n-1$, e.g., the repository of DNS entries or a collection of webpages. A client wants to fetch the i-th entry of the database. Although the database is public, the client wants to hide which entry it is interested in. Chor, Goldreich, Kushilevitz, and Sudan [CGKS95, CKGS98] first formulated this problem as Private Information Retrieval (PIR), and since then, a long line of works have focused on constructing efficient PIR schemes [CG97,Cha04,GR05,CMS99,KO97,Lip10,OS07,Gas04,DG16,PR93,DIO98, BLW17,BGI16,PPY18,IKOS04,Hen16,HH17,ACLS18,IKOS06,LG15,DHS14,CK20,CHK22,KC21, dCP22].

The good news is that PIR schemes with *poly-logarithmic* bandwidth are well-known [CG97, Cha04, GR05, CMS99, KO97, Lip10, OS07, PR93, BLW17, BGI16, PPY18, DG16, IKOS04, Hen16], either in the single-server or multi-server settings. The bad news is that in the classical PIR setting *without pre-processing*, all known schemes suffer from prohibitive server computation overhead: the server(s) must (in aggregate) perform computation that is linear in the database size n to answer each query. Intuitively, if there is an entry that the server does not look at, it leaks information that the client is not interested in that entry. Beimel, Ishai, and Malkin [BIM00] formalized this intuition into an elegant lower bound, showing that any PIR scheme without pre-processing must incur $\Omega(n)$ server computation per query.

Recognizing this inherent limitation, Beimel et al. [BIM00] introduce a new model for PIR that allows *pre-processing*, and they were the first to show that the linear-computation lower bound can be circumvented with the help of pre-processing. Subsequently, a line of works further explored PIR in the preprocessing model [CK20, CHK22, PY22, SACM21], culminating in the recent works by Corrigan-Gibbs, Henzinger, and Kogan [CHK22] and by Shi et al. [SACM21]. Corrigan-Gibbs,

[*] Author ordering is randomized. The work was done when Wei-Kai Lin was a post-doctoral researcher at CMU.

Table 1: **Comparison of single-server PIR schemes.** Q is the batch size for batch PIR, m is the number of clients, n is the database size, and $\epsilon \in (0, 1)$ is some suitable constant. "BW" means bandwidth per query. "CRA" means the composite residuosity assumption, ϕ-hiding is a number-theoretic assumption described in [CMS99], "OLDC" means oblivious locally decodable codes, and "VBB" means virtual-blackbox obfuscation.

Scheme	**Assumpt.**	**Adaptive**	**BW**	**Per-query time**		**Extra space**	
				Client	Server	Client	Server
Standard [Cha04, GR05, CMS99]	CRA or ϕ-hiding or LWE	✓	$\widetilde{O}(1)$	$\widetilde{O}(1)$	$O(n)$	0	0
Batch PIR [ACLS18, IKOS04]	same as above	×	$\widetilde{O}(1)$	$\widetilde{O}(1)$	$O(\frac{n}{Q})$	0	0
[CHR17, BIPW17]	OLDC	✓	n^ϵ	n^ϵ	n^ϵ	$O(1)$	mn
[BIPW17]	OLDC, VBB	✓	n^ϵ	n^ϵ	n^ϵ	0	n
[CK20]	LWE	✓	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(n)$	$\widetilde{O}_\lambda(\sqrt{n})$	0
[CHK22]	LWE	✓	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0
Ours	LWE	✓	$\widetilde{O}_\lambda(1)$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	$\widetilde{O}_\lambda(\sqrt{n})$	0

Henzinger, and Kogan [CHK22] proved that in the single-server and pre-processing setting, we can construct a PIR scheme with amortized $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation per query, while requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage. Here, we use $\widetilde{O}_\lambda(\cdot)$ to hide $\mathsf{poly}(\lambda, \log n)$ factors, where λ is the security parameter. Corrigan-Gibbs et al. [CHK22] also showed that their scheme achieves optimality up to $\mathsf{poly}\log$ factors in terms of server computation, assuming $\widetilde{O}(\sqrt{n})$ client storage. Unfortunately, their scheme suffers from $\widetilde{O}_\lambda(\sqrt{n})$ bandwidth overhead which is significantly worse than classical PIR schemes without pre-processing. On the other hand, Shi et al. [SACM21] showed that in a setting with two non-colluding servers, we can construct a PIR scheme that incurs only $\widetilde{O}_\lambda(1)$ online bandwidth and $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation per query, while requiring $\widetilde{O}_\lambda(\sqrt{n})$ client storage. Both of these schemes support *unbounded* number of queries after a one-time pre-processing, and the cost of the pre-processing is amortized to each query.

While the two schemes [SACM21, CHK22] achieve similar server and client computation overhead, Shi et al. [SACM21] has the advantage that it achieves $\widetilde{O}_\lambda(1)$ online bandwidth — although unfortunately, this is achieved at the price of requiring two non-colluding servers. Notably, Shi et al.'s scheme is known to be optimal up to $\mathsf{poly}\log$ factors even in the two-server setting, in terms of bandwidth and server computation, assuming that the client can only download roughly $\sqrt{n}$ amount of data during the offline pre-processing phase [CK20].

Given the state of the art, we ask whether we can achieve the best of both worlds. Specifically, we ask the following natural question — the same open question was also raised by Corrigan-Gibbs et al. in their recent work [CHK22]:

> Can we construct a *single-server* pre-processing PIR scheme that achieves (near) *optimality* in both *server computation* and *bandwidth*?

1.1 Our Contributions

We provide an affirmative answer to the aforementioned question by proving the following theorem:

Theorem 1.1. *Assume that the Learning With Errors (LWE) assumption holds. Then, there exists a single-server pre-processing PIR scheme that achieves amortized $\widetilde{O}_\lambda(1)$ bandwidth, $\widetilde{O}_\lambda(\sqrt{n})$ server and client computation per query, and requires $\widetilde{O}_\lambda(\sqrt{n})$ client storage.*

More specifically, in our scheme, there is a one-time pre-processing phase with the same overheads in all dimensions as Corrigan-Gibbs [CHK22] (up to $\mathsf{poly}\log$ factors). During the offline pre-processing, the client and the server engage in $\widetilde{O}_\lambda(\sqrt{n})$ communication, the server performs $\widetilde{O}_\lambda(n)$ computation, and the client performs $\widetilde{O}_\lambda(\sqrt{n})$ computation. In Theorem 1.1 above, the cost of the pre-processing is *amortized* to the subsequent queries. After the one-time pre-processing, we can support an unbounded number of queries, and for each query, we incur the same costs as stated in Theorem 1.1, in the *worst case*. Our actual construction makes use of two cryptographic primitives: fully homomorphic encryption (FHE) [Gen09, GSW13] and privately programmable pseudorandom functions [BLW17, PS18, KW21], both of which have known instantiations assuming LWE.

Near optimality. Our scheme is optimal up to $\mathsf{poly}\log$ factors in terms of server computation and bandwidth, in light of the lower bounds proven in recent works [CK20, CHK22]. Specifically, Corrigan-Gibbs and Kogan [CK20] showed that for any pre-processing PIR scheme where the server stores only the original database, it must be that $C \cdot T \geq \Omega(n)$ where C is the bandwidth incurred during the offline pre-processing and T is the online server time per query. The recent work of Corrigan-Gibbs, Henzinger, and Kogan [CHK22] proved that for any pre-processing PIR scheme that supports unbounded number of dynamic queries and assuming the server stores only the original database, it must be that $S \cdot T \geq \Omega(n)$ where S is client's storage and T is the online server time per query.

Although in the main body we focus on the special case where the parameters S and T are balanced, in Appendix B, we discuss how to achieve a smooth tradeoff between S and T. In particular, for any function $f(n) \in [\log^c n, n/\log^c n]$ for some suitable positive constant c, we give a scheme that requires only $\widetilde{O}_\lambda(f(n))$ client space, and achieves $\widetilde{O}_\lambda(n/f(n))$ online server and client time per query, and $\widetilde{O}_\lambda(1)$ bandwidth per query. Therefore, we achieve near optimality for every choice of client space.

Comparison with prior schemes. Table 1 compares our scheme against various prior works. We focus on schemes in the *single-server* setting, and for pre-processing PIR schemes, we amortize the pre-processing overhead over an unbounded number of subsequent queries. Among these schemes, batch PIR schemes [ACLS18, IKOS04, Hen16] must have a large batch size of Q to achieve the stated amortized performance, and fail in the scenario when the queries are generated *adaptively* and arrive one by one. We discuss additional related work in Section 1.2.

1.2 Additional Related Work

We now review some additional related work. Besides being first to define PIR with pre-processing, Beimel et al. [BIM00] additionally showed how to construct a preprocessing PIR with polylogarithmic online bandwidth assuming polylogarithmically many non-colluding servers, and $\mathsf{poly}(n)$ server storage. Unlike our work as well as the recent works by Corrigan-Gibbs et al. [CHK22, CK20], the scheme by Beimel et al. [BIM00] employs a *public* pre-processing, where the pre-processing results in no client-side secret state. In fact, in their scheme [BIM00], the server pre-processes the database, resulting in a $\mathsf{poly}(n)$-sized encoding of the database which is then stored by the server. The very recent work of Persiano and Yeo [PY22] proved that for any PIR scheme with *public* pre-processing, it must be that $T \cdot R \geq \Omega(n \log n)$ where T is the server computation per query and R is size of the additional state computed by the public pre-processing. In comparison, our work considers a *private* pre-processing model, i.e., at the end of the pre-processing, the client stores some secret state not seen by the server. This model matches well with a "subscription model" in practice. For example, every client that needs private DNS service can subscribe with the provider, and during subscription, they perform the one-time pre-processing.

Besides the single-server PIR scheme from FHE mentioned in Table 1, the work of Corrigan-Gibbs and Kogan [CK20] also propose another scheme assuming only linearly homomorphic encryption, which requires $O(n^{2/3})$ bandwidth and client computation and $O(n)$ server computation per query, as well as $O(n^{2/3})$ client storage. Further, the work of Corrigan-Gibbs, Henzinger, and Kogan [CHK22] additionally suggests a single-server PIR scheme assuming only linearly homomorphic encryption, incurring $O(\sqrt{n})$ bandwidth and client computation, and $O(n^{3/4})$ server computation per query, requiring $O(n^{3/4})$ client storage.

Hamlin et al. [HOWW19] suggested a related notion called *private anonymous data access* (PANDA). PANDA is a form of preprocessing PIR which requires an additional *third-party trusted setup* besides the client and the servers; and moreover, the server storage and time grow w.r.t. the number of corrupt clients. In applications (e.g., private DNS) that involve a potentially unbounded number of mutually distrustful clients, PANDA schemes would be unsuitable.

A line of works have explored the concrete efficiency of PIR schemes [ACLS18, MCR21, KC21, PPY18, GI14, MW22]. In particular, the work of Angel et al. [ACLS18] relies on batching to amortize the linear server computation over a batch of queries. Kogan and Corrigan-Gibbs [KC21] gives a practical instantiation of the two-server pre-processing PIR scheme described in their earlier work [CK20], with a new trick that removes the k-fold parallel repetition. For their private blocklist application, it turns out that the database is somewhat small, and therefore, they are willing to incur $\Theta(n)$ client-side computation per online query, in exchange for logarithmic bandwidth. The work of Patel et al. [PPY18] explores how to rely on a stateful client to improve the concrete performance of PIR schemes. Our work focuses on the asymptotical overhead, and we leave it to future work to consider concretely efficient instantiations that preserve our asymptotical performance.

Some works have considered achieving sublinear server time by relaxing the security definition to differential privacy. Toledo et al. [TDG16] improved the server time to sublinear with this relaxation, assuming a large number of servers are available. Albab et al. [AIVG22] also considered the differential privacy notion, and they can achieve sublinear amortized server computation in a batched setting.

Independent work. Subsequent to our work, Lazaretti and Papamanthou [LP22] proposed a similar construction. The main difference in their construction is that they claim to rely only on privately *puncturable* PRFs and we rely on privately *programmable* PRFs. However, inside their scheme, they are effectively using rejection sampling to construct a programmable PRF from a puncturable PRF — earlier work has pointed out that this approach will only work if the privately puncturable PRF satisfies rerandomizability [CC17]. Therefore, for Lazaretti and Papamanthou's scheme [LP22] to work, they need to rely on a rerandomizable privately puncturable PRF like what Canetti and Chen [CC17] suggested. Additionally, their privacy proof (in their Eprint version dated 2022-06-23) appears slightly incomplete but likely fixable. In particular, in the inductive argument in their privacy proof in their Section B.1, they argue that the sk part of the client's table is indistinguishable from randomly sampled secret keys (for the hard puncturing key). To prove the PIR scheme secure, they actually need to show that the client's table is indistinguishable form randomly sampled keys, not just for the sk part, but actually for the pair $(\mathsf{msk}, \mathsf{sk})$. This is because the server's view actually depends on the msks in the client's table. While it is outside the scope of our paper to complete their proof, we think changing the security definition of their pseudorandom sets to include the msk, and reproving their pseudorandom sets secure under this new definition should lend to fixing this issue.

2 Technical Roadmap

2.1 Starting Point: Optimal 2-Server Scheme By Shi et al.

2.1.1 An Inefficient Toy Scheme

Our starting point is the nearly optimal 2-server scheme by Shi et al. [SACM21], and we will explore how to coalesce the two servers into one. To understand their scheme, it helps to start out with the following toy scheme which is a slight variant of the strawman schemes described in recent works [CK20,SACM21]. Henceforth, we use the notations Right and Left to denote two non-colluding servers. Let $\mathcal{D}_n$ be some distribution from which we can sample random sets of expected size $\sqrt{n}$ — at this moment, the reader need not care what exactly the distribution $\mathcal{D}_n$ is.

Inefficient Toy 2-Server Scheme: Single-Copy Version

Offline preprocessing. (DB$[k]$ *denotes the k-th bit of the database*)

- Client samples $\sqrt{n}$ sets $S_1, S_2, \ldots, S_{\sqrt{n}} \subseteq \{0, 1, \ldots, n-1\}$ from the distribution $\mathcal{D}_n$.
- Client sends the resulting sets $S_1, \ldots, S_{\sqrt{n}}$ to Left. For each set $j \in [\sqrt{n}]$, Left responds with the parity bit $p_j := \oplus_{k \in S_j} \mathsf{DB}[k]$ of indices in the set.
- Client stores the hint table $T := \{T_j := (S_j, p_j)\}_{j \in [\sqrt{n}]}$.

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

- **Query:** (Client ⇔ Right)
  1. Find an entry $T_j := (S_j, p_j)$ in its hint table T such that $x \in S_j$. Let $S^* := S_j$ if found, else let S^* be a fresh random set containing x.
  2. Send the set $S := \mathbf{ReSamp}(S^*, x)$ to Right, where $\mathbf{ReSamp}(S^*, x)$ outputs a set almost identical to S^*, except that the coins used to determine x's membership are re-tossed.
  3. Upon obtaining a response $p := \oplus_{k \in S} \mathsf{DB}[k]$ from Right, output the candidate answer $\beta' := p_j \oplus p$ or $\beta' := 0$ if no such T_j was found earlier.
  4. Client obtains the true answer $\beta := \mathsf{DB}[x]$ — the full scheme will repeat this single-copy scheme $k = \omega(\log \lambda)$ times, and β is computed as a majority vote among the k candidate answers, which is guaranteed to be correct except with negligible probability.
- **Refresh** (Client ⇔ Left)
  1. Client samples a random set S' and sends S' to Left.
  2. Left responds with $p' := \oplus_{k \in S'} \mathsf{DB}[k]$. Let $\widetilde{p} = p' \oplus \beta$ if $x \notin S'$, else let $\widetilde{p} = p'$. If a table entry T_j containing x was found and consumed earlier, Client replaces T_j with $(S' \cup \{x\}, \widetilde{p})$.

In this 2-server toy scheme, during the offline phase, the client samples $\sqrt{n}$ sets each of expected size $\sqrt{n}$ from some distribution $\mathcal{D}_n$. It downloads the parities of all these sets from the Left server. It stores all these sets as well as the parity of each set in a local hint table. During the online phase, to query an index $x \in \{0, 1, \ldots, n-1\}$, the client looks up its hint table and finds a set S^* that contains x, whose parity is p_j. It then resamples the coins that determine whether x is in the set or not. It sends the resampled set to the Right server, which returns the client the parity p'. The client computes $\beta' = p' \oplus p_j$ as the candidate answer. If we choose the distribution $\mathcal{D}_n$ carefully, then, with significant probability, the $\mathbf{ReSamp}(x)$ will *remove the element x from the set, without adding or removing any other element.* In this case, the candidate answer β' would be correct. If we can ensure that each single copy has 2/3 correctness probability, then we can amplify the correctness probability to $1 - \mathsf{negl}(\lambda)$ through parallel repetition using $\omega(\log \lambda)$ copies and majority voting. Finally, once we consume a hint from the table, we need to replenish it. To achieve this, the client samples a random set S', and obtains its parity p' from the Left server. The client replaces the consumed entry with the set $S' \cup \{x\}$ and its parity which can be computed knowing p' and $\beta = \mathsf{DB}[x]$.

Privacy. Privacy w.r.t. the Left server is easy to see. Basically, the Left server sees $\sqrt{n}$ random sets sampled from $\mathcal{D}_n$ during the offline phase, and during each online query, it sees an additional random set also sampled from $\mathcal{D}_n$. Privacy w.r.t. the Right server can be proven using an inductive argument. Initially, the client's hint table consists of $\sqrt{n}$ random sets sampled independently from $\mathcal{D}_n$. Suppose that at the end of the i-th query the client's hint table satisfies the above distribution. Then, during the i-th query that requests some index $x \in \{0, 1, \ldots, n-1\}$, if some hint (S_j, p_j) is matched, i.e., $S_j \ni x$, then, the distribution of S_j is the same as sampling from $\mathcal{D}_n$ subject to containing x. Therefore, the set sent to the Right server, i.e., $\mathbf{ReSamp}(S_j)$ has the same distribution as sampling at random from $\mathcal{D}_n$. Further, notice that the client replaces the consumed entry with another set sampled at random subject to containing x. Thus, at the end of the i-th query, the client's hint table still has $\sqrt{n}$ independent and identically distributed (i.i.d.) sets sampled from $\mathcal{D}_n$.

Inefficiency of the toy scheme. In the toy scheme, both the server and the client perform roughly $\sqrt{n}$ computation per query. However, the online bandwidth to each of the two servers is roughly $\sqrt{n}$, and the client storage is $O(n)$.

2.1.2 Compressing the Bandwidth and Client Storage

Pseudorandom sets with private ReSamp. Shi et al. [SACM21] suggested an idea to improve the efficiency of the toy scheme in the two-server setting. To achieve this, they introduce a cryptographic object called a pseudorandom set (PRSet), allowing us to succinctly represent a pseudorandom set of size roughly $\sqrt{n}$ with a short key of $\mathsf{poly}(\lambda)$ bits. In this way, the client can store a key in place of each set, and send a key to the server in place of the full description of a set. Their PRSet scheme must support the following operations:

- $\mathsf{sk} \leftarrow \mathbf{Gen}(1^\lambda, n)$: samples a key sk that generates a pseudorandom set emulating the distribution $\mathcal{D}_n$;
- $S \leftarrow \mathbf{Set}(\mathsf{sk})$: given a key sk, enumerate the set S;
- $\mathbf{Member}(\mathsf{sk}, x)$: test if an element $x \in \{0, 1, \ldots, n-1\}$ is in $\mathbf{Set}(\mathsf{sk})$;

- $\mathsf{sk}' \leftarrow \mathbf{ReSamp}(\mathsf{sk}, x)$: given a key sk, generates a related key sk' that effectively resamples the coins that are used to determine whether x is in the set or not, while preserving all other coins[1];

Designing such a PRSet scheme turns out to be non-trivial, since we need to satisfy the following properties simultaneously.

- *Privacy of* **ReSamp***.* The resampled key output by $\mathbf{ReSamp}(\mathsf{sk}, x)$ must hide the point x that is being resampled.
- *Efficient membership test and set enumeration.* The membership test algorithm $\mathbf{Member}(\mathsf{sk}, x)$ must complete in $\widetilde{O}_\lambda(1)$ running time and the set enumeration algorithm $\mathbf{Set}(\mathsf{sk})$ must complete in $\widetilde{O}_\lambda(\sqrt{n})$ time.

Shi et al. [SACM21] show how to rely on a privately puncturable pseudorandom function [CC17, BKM17, BTVW17] to construct a PRSet scheme that supports a private **ReSamp** operation. Further, to satisfy efficient membership test and efficient set enumeration simultaneously, they carefully crafted a distribution $\mathcal{D}_n$ that the PRSet scheme emulates. Notably, whether two elements are in the set may not be independent in the distribution $\mathcal{D}_n$. Such weak dependence between elements brings additional possibilities of errors. In particular, $\mathbf{ReSamp}(\mathsf{sk}, x)$ *may accidentally remove other elements besides x.* If $\mathbf{ReSamp}(\mathsf{sk}, x)$ either fails to remove x or ends up removing additional elements besides x, the resulting PIR scheme would be incorrect. Shi et al. [SACM21] made sure that the probability of such error is small, such that each single copy of the PIR scheme still has $2/3$ correctness.

Optimal 2-server PIR scheme. With such a PRSet scheme, we can easily modify the aforementioned toy scheme to compress the client storage and bandwidth [SACM21]. Specifically, during the offline phase, the client sends $\sqrt{n}$ PRSet keys to the Left server. The Left server uses the set enumeration algorithm **Set** to enumerate the sets and sends the client their parity bits. The client now stores a hint table where each entry is of the form (sk_i, p_i), where sk_i is a PRSet key that can be used to generate a set of size roughly $\sqrt{n}$, and p_i is the parity bit as before. During an online query for $x \in \{0, 1, \ldots, n-1\}$, the client finds an sk^* in its hint table such that $\mathbf{Member}(\mathsf{sk}^*, x) = 1$, and sends the outcome of $\mathbf{ReSamp}(\mathsf{sk}^*, x)$ to the Right server. If such a key is not found, the client simply samples a random $\mathsf{sk}' \leftarrow \mathbf{Gen}(1^\lambda, n)$ and sends it to the server. The client computes the candidate answer the same way as before. What is most interesting is how to perform the refresh operation to replenish the consumed key. This is achieved in the following manner:

- Sample $\mathsf{sk}' \leftarrow \mathbf{Gen}(1^\lambda, n)$ subject to $\mathbf{Member}(\mathsf{sk}', x) = 1$, and send the outcome of $\mathbf{ReSamp}(\mathsf{sk}', x)$ to the Left server.
- The Left server enumerates the set using the **Set** algorithm and sends the client the parity bit p'. The client replaces the consumed entry with $(\mathsf{sk}', p' \oplus \beta)$ where $\beta = \mathsf{DB}[x]$ is the true answer to the current query.

2.2 Highlights of Our Construction and Proof Techniques

Corrigan-Gibbs and Kogan [CK20] proposed an FHE-based technique to compile a two-server pre-processing PIR scheme into a single-server scheme, and the technique was further extended by Corrigan-Gibbs, Henzinger, and Kogan [CHK22] — this technique is remotely related to techniques

[1] Shi et al. [SACM21] referred to **ReSamp** as **Punct** since the operation is implemented by calling the puncturing operation of the underlying privately puncturable PRF.

for converting multi-prover proof systems into single-prover proof systems [ABOR00, BMW98, DHRW16, DNR16, KRR13]. The idea is to get rid of the Left server and redirect the queries originally destined for the Left server instead to the Right server, but now encrypted under a fully homomorphic encryption (FHE) scheme. The server now evaluates the answers to the query through homomorphic evaluation. Unfortunately, this compilation technique is incompatible with Shi et al. [SACM21]. The technicality arises from the fact that FHE evaluation relies on *circuit* as the computation model, whereas the sublinear server computation time of Shi et al. [SACM21] relies on the *RAM* model (since dynamic memory accesses are needed). Recall that every time the server receives a pseudorandom set key, it needs to expand the key to a set of size $\widetilde{O}(\sqrt{n})$, and retrieve the parity of the database bits at precisely these indices. On a RAM, this computation costs $\widetilde{O}(\sqrt{n})$, but now that the key is encrypted under FHE, using a circuit to homomorphically evaluate this computation would require an $\Omega(n)$-sized circuit — this defeats our goal of having sublinear server time.

Fortunately, the following critical observation, first made by Corrigan-Gibbs et al. [CHK22], saves the day.

Observation. Although homomorphically evaluating the parity of a single set takes a linear-sized circuit, we can batch-evaluate the parity bits of $\Theta(\sqrt{n})$ sets in a circuit of size $\widetilde{O}(n)$, leveraging oblivious sort. With batch evaluation, the amortized cost per set is only $\widetilde{O}(\sqrt{n})$.

Idea 1: Batched refresh operations. The above batching idea allows us to compile the offline phase of Shi et al. [SACM21] without suffering from the RAM-to-circuit conversion blowup (ignoring poly-logarithmic factors). However, the online phase is problematic, since Shi et al. requires that the client talks to the Left server to perform a refresh operation every time it makes a query.

Our first idea is inspired by Corrigan-Gibbs et al. [CHK22]. Instead of performing refreshes individually, we can group them into $Q = \sqrt{n}$-sized batches. We first consider a bounded scheme that supports only $Q = \sqrt{n}$ queries — in this way, we can hope to front-load all Q refresh operations upfront during the pre-processing phase. It is easy to get an unbounded scheme given a bounded scheme. We can simply rerun the offline setup every Q queries, and amortize the cost of the periodic setup over each query — in fact, it is also not hard to deamortize the periodic setup and spread the work across time.

In summary, through batching the refresh operations, we can hope to achieve $\widetilde{O}_\lambda(\sqrt{n})$ amortized server computation per refresh operation.

Idea 2: a pseudorandom set scheme supporting Add and ReSamp. If we front-load all Q refresh operations upfront during the offline pre-processing, a new technicality arises. Recall that during a query for $x \in \{0, 1, \ldots, n-1\}$, we must replenish the consumed entry with a set sampled subject to containing the queried element x. During the offline pre-processing, however, we do not have foreknowledge of x. Therefore, we can only hope to sample (pseudo-)random sets (represented by keys) during the offline pre-processing, and add the element x to the set during the online phase.

This means that we need a new PRSet that supports not only **ReSamp**, but also an **Add** operation. Specifically, given a PRSet key sk, the client should be able to call $\mathsf{sk}' \leftarrow \mathbf{Add}(\mathsf{sk}, x)$ and then call $\mathsf{rsk} \leftarrow \mathbf{ReSamp}(\mathsf{sk}', y)$, and send the resulting rsk to the server. For privacy, the resulting rsk must hide both x and y. To construct such a PRSet scheme, we need a cryptographic primitive called privately programmable pseudorandom functions [BLW17, PS18, KW21], which is stronger than the privately puncturable pseudorandom functions employed by Shi et al.

New proof techniques. For the optimal two-server scheme of Shi et al. [SACM21], they have a relatively simple privacy proof. In comparison, our privacy proof is much more involved, and we need new techniques to make the privacy proof work. At a high level, the challenges in the privacy proof arise due to the way the probability analysis is interwined with the cryptography. Our main new idea in the privacy proof is to introduce a *lazy sampling* technique[2] that provides an alternative way to view how the client generates the key to send to the server — called the "frontend" in our proof. In particular, during the scheme, the client scans through its primary table and checks if each key contains the current query x. Whenever such a check is made and the answer is no, it creates a constraint on the entry, i.e., the entry should not contain x. Whenever an entry is matched during a query x, a constraint is created that the entry should contain x. If the entry was previously promoted from the backup table, these constraints can also be modified accordingly. Thus, we can imagine that the client maintains a set of constraints in this way, and defer the actual sampling of the key to send to the server to the very last moment, subject to the set of constraints that have been maintained on the matching entry. With this lazy sampling view, we can decouple the *frontend* (i.e., how the client interacts with the server) from the *backend* (i.e., how the client maintains its local primary table), and switch their distributions one by one in the subsequent hybrids. In our actual proof later, the frontend and the backend diverge at some point when we switch to the lazy sampling view, and eventually, after switching both the backend and the frontend, they would converge again, i.e., the distribution of the key sent to the server matches the distribution of the matched entry (after some post-processing) again. At this moment, we can undo the lazy sampling view, and continue to complete the proof.

Another technicality in our proof arises from the fact that the form of the standard security definition of privately puncturable PRF is not in a convenient form we can easily use in our proof. For this reason, we introduce a *key technical lemma* (Section 6.2) that is closer to the form we want. We repeatedly apply this key technical lemma when making the switches between our hybrid experiments.

To help the reader understand the technicalities of our privacy proof and our new ideas, we give an informal proof roadmap in Section 6.1.

3 Preliminaries

3.1 Privately Programmable Pseudorandom Functions

Intuitively, a privately programmable pseudorandom function [BLW17, PS18, KW21] is a pseudorandom function (PRF) with one extra capability: it allows one to create a *programmed key* that forces the PRF's outcomes in at most L distinct input points $\{x_i\}$ to be a set of pre-determined values $\{v_i\}$. For security, we want to guarantee the privacy of the programmed inputs. Specifically, if the set of output values $\{v_i\}$ are randomly chosen, then the programmed key should not leak more information about the set of input points programmed. Further, the programmed key should not leak the original PRF's evaluation outcomes at the programmed inputs prior to the programming.

3.1.1 Syntax

Let $\mathcal{X}$ denote the input domain and let $\mathcal{V}$ denote the output range, whose sizes may depend on the security parameter λ. A programmable pseudorandom function is a tuple (**Gen**, **Eval**, **Prog**, **PEval**) of efficient, possibly randomized algorithms with the following syntax:

[2] Our lazy sampling is remotely reminiscent of the delayed sampling technique of Bartusek and Khurana [BK22].

```
$\underline{\mathsf{RealPPRF}_{\mathcal{A}}(1^\lambda, L)}$:

  $P := \{(x_i, v_i)\}_{i\in[L']} \leftarrow \mathcal{A}(1^\lambda, L)$
                  $//$ *require:* $L' \leq L$
  $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$
  $\mathsf{sk}_P \leftarrow \mathbf{Prog}(\mathsf{msk}, P)$
  $\mathsf{sk}_P \rightarrow \mathcal{A}$
  **repeat**
    $x \leftarrow \mathcal{A}$
    $\mathbf{Eval}(\mathsf{msk}, x) \rightarrow \mathcal{A}$
  **until** $\mathcal{A}$ halts
```

```
$\underline{\mathsf{IdealPPRF}_{\mathcal{A},\mathsf{Sim}}(1^\lambda, L)}$:

  $P := \{(x_i, v_i)\}_{i\in[L']} \leftarrow \mathcal{A}(1^\lambda, L)$
                  $//$ *require:* $L' \leq L$
  $\mathsf{sk}_P \leftarrow \mathsf{Sim}(1^\lambda, P, L)$
  $\mathsf{sk}_P \rightarrow \mathcal{A}$
  **repeat**
    $x \leftarrow \mathcal{A}$
    If $x \notin \{x_i\}_{i\in[L']}$ then $\mathbf{PEval}(\mathsf{sk}_P, x) \rightarrow \mathcal{A}$
    Else $v \xleftarrow{\$} \mathcal{V}, v \rightarrow \mathcal{A}$
  **until** $\mathcal{A}$ halts
```

Figure 1: The real and ideal experiments for simulation security.

- $\mathbf{Gen}(1^\lambda, L)$: given the security parameter λ and an upper bound, L, on the number of programmable inputs, output a master secret key msk.
- $\mathbf{Eval}(\mathsf{msk}, x)$: given the master secret key msk and an input $x \in \mathcal{X}$, output the evaluation result $v \in \mathcal{V}$ on the input x.
- $\mathbf{Prog}(\mathsf{msk}, P = \{(x_i, v_i)\})$: given the master secret key msk and a set P containing up to L pairs $(x_i, v_i) \in \mathcal{X} \times \mathcal{V}$, where all x_i's must be distinct, output a programmed key sk_P.
- $\mathbf{PEval}(\mathsf{sk}_P, x)$: given a programmed key sk_P and an input $x \in \mathcal{X}$, output the evaluation outcome, $v \in \mathcal{V}$, over the input x.

Correctness of programming. A programmable function satisfies correctness if for all λ, $L = \mathsf{poly}(\lambda) \in \mathbb{N}$, all sets of up to L pairs $P := \{(x_i, v_i)\} \subseteq \mathcal{X} \times \mathcal{V}$ (with distinct x_is), we have the following:

1. For every $i \in [|P|]$,

$$\Pr\left[\mathbf{PEval}(\mathsf{sk}_P, x_i) \neq v_i \;\middle|\; \begin{array}{c} \mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L) \\ \mathsf{sk}_P \leftarrow \mathbf{Prog}(\mathsf{msk}, P) \end{array} \right] \leq \mathsf{negl}(\lambda), \text{ and}$$

2. For any x' not in P, we have

$$\Pr\left[\mathbf{PEval}(\mathsf{sk}_P, x') \neq \mathbf{Eval}(\mathsf{msk}, x') \;\middle|\; \begin{array}{c} \mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L) \\ \mathsf{sk}_P \leftarrow \mathbf{Prog}(\mathsf{msk}, P) \end{array} \right] \leq \mathsf{negl}(\lambda).$$

We note that Peikert and Shiehian [PS18] did not define the second correctness condition above, but their proof shows that the second condition also holds.

3.1.2 Security Definitions

Definition 3.1 (Simulation security)**.** A programmable function is *simulation secure*, if there is a probabilistic polynomial-time (PPT) simulator Sim such that for any PPT adversary $\mathcal{A}$ and any polynomial $L(\lambda)$,

$$\left\{\mathsf{RealPPRF}_{\mathcal{A}}(1^\lambda, L)\right\}_{\lambda\in\mathbb{N}} \overset{c}{\approx} \left\{\mathsf{IdealPPRF}_{\mathcal{A},\mathsf{Sim}}(1^\lambda, L)\right\}_{\lambda\in\mathbb{N}},$$

```
$\underline{\mathsf{RealPPRFPriv}_{\mathcal{A}}(1^\lambda, L)}$:

  $\{x_i\}_{i\in[L']} \leftarrow \mathcal{A}(1^\lambda, L)$
            // require: $L' \leq L$
  $\{v_i\}_{i\in[L']} \xleftarrow{\$} \mathcal{V}$
  $P := \{(x_i, v_i)\}_{i\in[L']}$
  $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$, $\mathsf{sk} \leftarrow \mathbf{Prog}(\mathsf{msk}, P)$
  $\mathsf{sk} \rightarrow \mathcal{A}$
```

```
$\underline{\mathsf{IdealPPRFPriv}_{\mathcal{A},\mathsf{Sim}}(1^\lambda, L)}$:

  $\{x_i\}_{i\in[L']} \leftarrow \mathcal{A}(1^\lambda, L)$
            // require: $L' \leq L$
  $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$
  $\mathsf{sk} \rightarrow \mathcal{A}$
```

Figure 2: The real and ideal experiments for private programmability.

where $\mathsf{RealPPRF}$ and $\mathsf{IdealPPRF}$ are the respective views of $\mathcal{A}$ in the executions of Figure 1 and "$\overset{c}{\approx}$" denotes computational indistinguishability.

Definition 3.2 (Private programmability)**.** A programmable function is *privately programmable*, if there is a PPT simulator Sim such that for any PPT adversary $\mathcal{A}$ and any polynomial $L(\lambda)$,

$$\left\{\mathsf{RealPPRFPriv}_{\mathcal{A}}(1^\lambda, L)\right\}_{\lambda\in\mathbb{N}} \overset{c}{\approx} \left\{\mathsf{IdealPPRFPriv}_{\mathcal{A},\mathsf{Sim}}(1^\lambda, L)\right\}_{\lambda\in\mathbb{N}},$$

where $\mathsf{RealPPRFPriv}$ and $\mathsf{IdealPPRFPriv}$ are the respective views of $\mathcal{A}$ in the executions of Figure 2.

Last but not the least, we define an additional security property, i.e., the ordinary pseudorandomness notion for the PRF. We prove that pseudorandomness is implied by private programmability — however, defining this notion explicitly will facilitate our proofs later.

Definition 3.3 (Pseudorandomness)**.** We say that a programmable pseudorandom function satisfies pseudorandomness iff for every probabilistic polynomial-time adversary $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that the following holds:

$$\left|\Pr[\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L) : \mathcal{A}^{\mathbf{Eval}(\mathsf{msk},\cdot)} = 1] - \Pr[\mathsf{rf} \xleftarrow{\$} \mathcal{RF} : \mathcal{A}^{\mathsf{rf}(\cdot)} = 1]\right| \leq \mathsf{negl}(\lambda),$$

where $\mathcal{RF}$ denotes the family of random functions that map the input domain $\mathcal{X}$ to the output range $\mathcal{V}$.

Fact 3.4. *Suppose that a programmable PRF scheme satisfies private programmability, then it also satisfies pseudorandomness.*

Proof. Let q be the maximum number of queries made by the pseudorandomness adversary $\mathcal{A}$. We consider a sequence of hybrids $\mathsf{H}_0, \mathsf{H}_1, \ldots, \mathsf{H}_q$. In H_j where $j \in \{0, 1, \ldots, q\}$, for the first j *distinct* queries made by $\mathcal{A}$, return to $\mathcal{A}$ truly random answers, and for the remaining queries, return the outcomes of the PRF evaluation. If $\mathcal{A}$ makes any repeat query, it always gets the same answer as before.

It suffices to show that no probabilistic polynomial-time $\mathcal{A}$ can distinguish H_i and H_{i+1} for any $i \in \{0, 1, \ldots, q-1\}$. To show this, consider an intermediate hybrid H'_i. In H'_i, the first i distinct queries are answered with true randomness, and the remaining queries are answered using a simulated key generated by $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$.

We first show that H_{i+1} is computationally indistinguishable from H'_i. Suppose that there is an efficient adversary $\mathcal{A}$ that can distinguish H'_i and H_{i+1}. We can construct an efficient reduction $\mathcal{B}$ that breaks the private programmability of the underlying PRF. $\mathcal{B}$ answers the first i distinct queries from $\mathcal{A}$ using true randomness. When $\mathcal{A}$ submits the $(i+1)$-th distinct query x_{i+1}, $\mathcal{B}$ submits $\{x_{i+1}\}$ to its own challenger. It gets back from its challenger sk. For all remaining queries x_j for $j \in [i+1, q]$, it returns $\mathbf{PEval}(\mathsf{sk}, x_j)$ to answer to $\mathcal{A}$. If $\mathcal{B}$ is playing $\mathsf{RealPPRFPriv}$, then $\mathcal{A}$'s view is statistically indistinguishable from H_{i+1} (where the negligible statistical failure comes from the "correctness of programming" failure probability), else if $\mathcal{B}$ is playing $\mathsf{IdealPPRFPriv}$, then $\mathcal{A}$'s view is identically distributed as H'_i.

Next, we show that H'_i is computationally indistinguishable from H_i. Consider H''_i in which all but the first i queries are answered using a key sk generated as follows: $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$, $\mathsf{sk} \leftarrow \mathbf{Prog}(\mathsf{msk}, \emptyset)$. H_i is statistically indistinguishable from H''_i due to the correctness of the programmable PRF. H''_i is computationally indistinguishable from H'_i through a straightforward reduction to the private programmability of the PRF.

Summarizing the above, H_i is computationally indistinguishable from H_{i+1} and this suffices for proving the claim. □

3.1.3 Construction

In our syntax and security definitions above, we want the programmable PRF to support programming *at most* L inputs. By contrast, Peikert and Shiehian [PS18] gave a construction of privately programmable PRFs where the **Prog** function must program *exactly* L inputs. Similarly, in their security definitions, the admissible adversary $\mathcal{A}$ is required to satisfy $L' = L$ (as opposed to $L' \leq L$ in our case).

Given a privately programmable PRF construction that programs exactly L inputs, we now show how to construct a new scheme that allows programming *up to* L inputs. In our PIR construction later, we want the PRF's input domain to contain all strings of length up to some parameter $\ell \in \mathbb{N}$. We use the notation $\{0,1\}^{\leq \ell}$ to denote all strings of length up to ℓ.

Let $\mathsf{PRF}' := (\mathbf{Gen}', \mathbf{Eval}', \mathbf{Prog}', \mathbf{PEval}')$ denote a privately programmable PRF whose input domain is $\mathcal{X}' = \{0,1\}^{\leq \ell+1}$, i.e., all strings of length up to $\ell + 1$, and whose output range is $\mathcal{V}$, supporting programming exactly L inputs. We now construct a privately programmable PRF scheme denoted PRF whose input domain is $\mathcal{X} = \{0,1\}^{\leq \ell}$, i.e., all strings of length up to ℓ, and whose output range is $\mathcal{V}$, i.e., the same as that of PRF'.

- $\mathbf{Gen}(1^\lambda, L)$: let $\mathsf{msk} \leftarrow \mathbf{Gen}'(1^\lambda, L)$, and output msk;
- $\mathbf{Eval}(\mathsf{msk}, x)$: output $\mathbf{Eval}'(\mathsf{msk}, x||0)$;
- $\mathbf{Prog}(\mathsf{msk}, P = \{(x_i, v_i)\}_{i \in [L']})$:
  - choose $L - L'$ distinct strings of length at most $\ell + 1$ that end with 1, denoted $x'_1, \ldots, x'_{L-L'}$;
  - for $j \in [L - L']$, choose $v_j \stackrel{\$}{\leftarrow} \mathcal{V}$ at random;
  - call $\mathsf{sk} \leftarrow \mathbf{Prog}'(\mathsf{msk}, \{(x_i||0, v_i)\}_{i \in [L']} \cup \{(x'_j, v_j)\}_{j \in [L-L']})$, and output sk.
- $\mathbf{PEval}(\mathsf{sk}, x)$: let $v \leftarrow \mathbf{PEval}(\mathsf{sk}, x||0)$ and output v.

Claim 3.5. *Suppose that the underlying programmable* PRF' *that maps* $\{0,1\}^{\ell+1}$ *to* $\mathcal{V}$ *satisfies correctness, simulation security, and private programmability. Then, the above* PRF *which maps* $\{0,1\}^{\ell}$ *to* $\mathcal{V}$ *also satisfies correctness, simulation security, and private programmability.*

We defer the proof of the above claim to Appendix E.1.

We can use Peikert and Shiehian [PS18]'s scheme (based on LWE) as our the underlying privately puncturable PRF to instantiate Claim 3.5. The schem by Boyle et al. [BGIK22] is not suitable for our application, since their evaluation time is quasilinear in the input domain size which would lead to super-linear server computation.

3.2 Single-Server Private Information Retrieval

We define a single-server private information retrieval (PIR) scheme in the pre-processing setting. In a single-server PIR scheme, we have two stateful machines called the client and the server. The scheme consists of two phases:

- **Offline setup.** The offline setup phase is run only once upfront. The client receives nothing as input, and the server receives a database $\mathsf{DB} \in \{0,1\}^n$ as input. The client sends a single message to the server, and the server responds with a single message.
- **Online queries.** This phase can be repeated multiple times. Upon receiving an index $x \in \{0,1,\ldots,n-1\}$, the client sends a single message to the server, and the server responds with a single message. The client performs some computation and outputs an answer $\beta \in \{0,1\}$.

Correctness. Given a database $\mathsf{DB} \in \{0,1\}^n$, where the bits are indexed $0,1,\ldots,n-1$, the correct answer for a query $x \in \{0,1,\ldots,n-1\}$ is the x-th bit of DB.

For correctness, we require that for any q, n, that are polynomially bounded in λ, there is a negligible function $\mathsf{negl}(\cdot)$, such that for any database $\mathsf{DB} \in \{0,1\}^n$, for any sequence of queries $x_1, x_2, \ldots, x_q \in \{0,1,...,n-1\}$, an honest execution of the PIR scheme with DB and queries $x_1, x_2, \ldots, x_q$, returns all correct answers with probability $1 - \mathsf{negl}(\lambda)$.

Privacy. We say that a single-server PIR scheme satisfies privacy, iff there exists a probabilistic polynomial-time simulator Sim, such that for any probabilistic polynomial-time adversary $\mathcal{A}$ acting as the server, $\mathcal{A}$'s views in the following two experiments are computationally indistinguishable:

- Real: an honest client interacts with $\mathcal{A}$ who acts as the server and may arbitrarily deviate from the prescribed protocol. In every online step t, $\mathcal{A}$ 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;
- Ideal: the simulated client Sim interacts with $\mathcal{A}$ who acts as the server. In every online $\mathcal{A}$ may adaptively choose the next query $x_t \in \{0,1,\ldots,n-1\}$, and Sim is invoked without receiving x_t.

3.3 The Distribution $\mathcal{D}_n$

For convenience, we often write $x \in \{0,1,\ldots,n-1\}$ as a binary string, i.e., $x \in \{0,1\}^{\log n}$.

Our pseudorandom set emulates the same distribution $\mathcal{D}_n$ that was defined earlier in Shi et al. [SACM21]. Specifically, to define the distribution $\mathcal{D}_n$, imagine that we have a random oracle $\mathsf{RO}(\cdot) : \{0,1\}^* \to \{0,1\}$ that is sampled at random upfront — our actual PRSet scheme later will replace the RO with a PRF so our construction does not need an RO. Henceforth, let $B := \lceil 2 \log\log n \rceil$. An element $x \in \{0,1\}^{\log n}$ is in the set iff for every $i \in [\frac{\log n}{2} + B]$, $\mathsf{RO}\left((0^B || x)[i:]\right)$ returns 1 — in other words, if hashing every sufficiently long suffix of the string $0^B || x$ using the random oracle RO gives back 1. Throughout the paper, we write $\log = \log_2$, and assume that $\log n$ is an even integer — this is without loss of generality since we can always round it up to an even number incurring only constant blowup.

Efficient membership test and set enumeration. One important observation about the distribution $\mathcal{D}_n$ is that the decisions regarding whether two elements x and y are in the set or not can be weakly dependent — as Shi et al. [SACM21] pointed out, this property is important for simultaneously ensuring efficient membership test and efficient set enumeration. Clearly, to test if an element $x \in \{0,1\}^{\log n}$ is in the set or not, we only need to make $\frac{\log n}{2} + B$ calls to the RO.

Enumerating all elements in the set can be accomplished by making roughly $\sqrt{n} \cdot \mathsf{poly} \log n$ calls to RO with at least $1 - o(1)$ probability. Let $\ell \geq \frac{1}{2} \log n + 1$, and let Z_ℓ be the set of all strings z of length exactly ℓ, such that using RO to "hash" all suffixes of z of length at least $\frac{1}{2} \log n + 1$, outputs 1. To enumerate the set generated by RO, we can start with $Z_{\frac{1}{2}\log n+1}$ which takes at most $2^{\frac{1}{2}\log n+1}$ calls to generate. Then, for each $\ell := \frac{1}{2} \log n + 2$ to $\log n$, we will generate Z_ℓ from $Z_{\ell-1}$. This can be accomplished by enumerating all elements $z' \in Z_{\ell-1}$, and checking whether $\mathsf{RO}(0||z') = 1$ and $\mathsf{RO}(1||z') = 1$. Finally, for every element $z \in Z_{\log n}$, we check if it is the case that for every $j \in [B]$, $0^j||z$ hashes to 1. If so, the element z is in the set.

Useful properties of $\mathcal{D}_n$. We will need to use the following useful facts about the distribution $\mathcal{D}_n$ all of which were proven by Shi et al. [SACM21].

Fact 3.6. *For any fixed $x \in \{0, 1, \dots, n-1\}$, $\Pr_{S \overset{\$}{\leftarrow} \mathcal{D}_n}[x \in S] = \frac{1}{\sqrt{n} \cdot 2^B}$. Moreover, $\mathbb{E}_{S \overset{\$}{\leftarrow} \mathcal{D}_n}[|S|] \leq \frac{\sqrt{n}}{\log^2 n}$.*

Henceforth, let $\mathcal{D}_n^{+x}$ be the following distribution: sample $S \overset{\$}{\leftarrow} \mathcal{D}_n$ subject to $x \in S$. Given $x, y \in \{0,1\}^{\log n}$, we say that x and y are *related*, if they share a common suffix of length at least $\frac{1}{2} \log n + 1$. Given a set $S \subseteq \{0, 1, \dots, n-1\}$, let $N_{\text{related}}(S, x)$ be the number of elements in S that are related to x.

Fact 3.7 (Number of related elements in sampled set)**.** *Fix an arbitrary element $x \in \{0, 1, \dots, n-1\}$. Then,*

$$\mathbb{E}_{S \overset{\$}{\leftarrow} \mathcal{D}_n^{+x}}[N_{\text{related}}(S, x)] \leq \frac{1}{\log n}$$

Fact 3.8 (Coverage probability)**.** *Let $m \geq 6\sqrt{n} \cdot \log^3 n$. For any fixed $x \in \{0, 1, \dots, n-1\}$, $\Pr_{S_1,\dots,S_m \overset{\$}{\leftarrow} \mathcal{D}_n^m}[x \notin \cup_{i \in [m]} S_i] \leq 1/n$.*

Henceforth, let $\mathsf{EnumTime}(\mathsf{RO})$ denote the number of RO calls made by the aforementioned set enumeration algorithm to enumerate the set generated by RO.

Fact 3.9 (Efficient set enumeration)**.** *Suppose that $n \geq 4$. For any fixed $x \in \{0, 1, \dots, n-1\}$,*

$$\Pr_{\mathsf{RO} \overset{\$}{\leftarrow} \mathcal{D}_n^{+x}}\left[\mathsf{EnumTime}(\mathsf{RO}) > 6\sqrt{n} \log^5 n\right] \leq 1/\log n$$

4 Privately Programmable Pseudorandom Set

4.1 Definition

In our Privately Programmable Pseudorandom Set (PRSet) scheme, we can sample a key sk that defines a pseudorandom set. We can support two operations on the key: we can call $\mathbf{Add}(\mathsf{sk}, x)$ to force x to be added to the set, we can also call $\textbf{ReSamp}(\mathsf{sk}, x)$ to cause the decision whether x is in the set or not to be resampled. The key output by a **ReSamp** operation is said to be *final*, i.e., we cannot perform any more operations on it. By contrast, keys output by either **Gen** or **Add** are said to be *intermediate*, i.e., we can still perform more operations on them. Henceforth, we use the notation rsk to denote a final key and sk to denote an intermediate key. Jumping ahead, later in our PIR scheme, the client always sends to the server a final key during an online query; however, the client locally stores a set of intermediate keys.

- $\mathsf{sk} \leftarrow \textbf{Gen}(1^\lambda, n)$: given the security parameter 1^λ and the universe size n, samples a secret key sk;
- $S \leftarrow \textbf{Set}(\mathsf{rsk})$: a deterministic algorithm that outputs a set S given a final secret key rsk;
- $b \leftarrow \textbf{Member}(\mathsf{sk}, x)$: given an intermediate secret key sk and an element $x \in \{0, 1, \ldots, n-1\}$, output a bit indicating whether $x \in \textbf{Set}(\mathsf{sk})$;
- $\mathsf{sk}_{+x} \leftarrow \textbf{Add}(\mathsf{sk}, x)$: given an intermediate secret key sk and an element $x \in \{0, 1, \ldots, n-1\}$, output a secret key sk_{+x} such that $x \in \textbf{Set}(\mathsf{sk}_{+x})$;
- $\mathsf{rsk}_{-x} \leftarrow \textbf{ReSamp}(\mathsf{sk}, x)$: given an intermediate secret key sk and an element $x \in \{0, 1, \ldots, n-1\}$, output a final key rsk_{-x} that "resamples" the decision whether x is in the set or not.

We note that a PRSet scheme is parametrized by a family of distributions $\mathcal{D}_n$. The pseudorandom set generated by the PRSet scheme should emulate the distribution $\mathcal{D}_n$ — we will define this more formally shortly.

Jumping ahead, later in our application, for each PRSet key sampled using **Gen**, we perform at most one **Add** operation on the key before we perform **ReSamp** and obtain a final key.

Efficiency requirements. Our PRSet scheme samples pseudorandom sets of size roughly $\sqrt{n}$. We want an efficient set enumeration algorithm $\textbf{Set}(\mathsf{rsk})$ that takes time roughly $\sqrt{n}$ (rather than linear in n). Additionally, we want that the membership test $\textbf{Member}(\mathsf{sk}, x)$ to complete in polylogarithmic time.

Remark 4.1. We do not give security definitions to our PRSet. Jumping ahead, the privacy proof of our PIR scheme actually opens up the PRSet scheme and relies on the properties of the underlying PRF directly. Nonetheless, abstracting out the PRSet helps to make the description of our PIR scheme conceptually cleaner.

4.2 Construction

We now present our PRSet construction. As mentioned, we assume that for each key sampled through **Gen**, at most one **Add** operation can be performed on the key before we call **ReSamp** which produces a final key.

Intuition for our PRSet. In our pseudorandom set, we simply replace the RO with a PRF function, such that its description can be compressed using a short key.

Our pseudorandom set supports two additional operations:

- The $\textbf{Add}(\mathsf{sk}, x)$ operation modifies the secret key sk such that the element $x \in \{0, 1\}^{\log n}$ is forced to be in the set. In our construction, this is done in the most naïve way: simply attach the element x to the secret key. This will be fine in our PIR construction since the intermediate key generated by **Add** is stored only on the client side and never sent to the server. Therefore, we do not need the resulting key to hide the point x that is added.

- The $\mathbf{ReSamp}(\mathsf{sk}, x)$ operation takes in an intermediate key that is either the output of **Gen** or the output of a previous **Add** operation, and it resamples the decision whether the element $x \in \{0,1\}^{\log n}$ is in the set or not. In our PIR scheme later, this resampled key will be sent to the server during online queries. Therefore, we want the resulting key to hide not only the element x that is being resampled, but also the element x' that was added earlier should the input key sk be the result of a previous $\mathbf{Add}(\_, x')$ operation.

  In our construction, this is accomplished in the following way. First, we sample at random the answers $\{v_i\}_{i \in [\frac{\log n}{2} + B]}$ — we want to force the PRF's evaluation at points $\{(0^B || x)[i :]\}_{i \in [\frac{\log n}{2} + B]}$ to be the values $\{v_i\}_{i \in [\frac{\log n}{2} + B]}$. Next, if the input key sk is the result of a previous $\mathbf{Add}(\_, x')$ operation, for any point $(0^B || x')[i :]$ where $i \in [\frac{\log n}{2} + B]$, if $(0^B || x')[i :] \neq (0^B || x)[i :]$, then we want to force the PRF's evaluation on $(0^B || x')[i :]$ to be 1. Finally, we call the underlying PRF's **Prog** function, to force the aforementioned outcomes on all the relevant points. Clearly, the total number of constraints to be forced is at most $L = 2(\frac{\log n}{2} + B)$.

Detailed construction. We describe our PRSet construction below.

PRSet **Scheme**

Parameters: $B := \lceil 2 \log \log n \rceil$, $L = 2(\frac{\log n}{2} + B)$.

- $\mathsf{sk} \leftarrow \mathbf{Gen}(1^\lambda, n)$: call $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$, and output $\mathsf{sk} := (\mathsf{msk}, \bot)$.
- $S \leftarrow \mathbf{Set}(\mathsf{rsk})$: Same as the set enumeration algorithm in Section 3.3, except that the calls to $\mathsf{RO}(\cdot)$ are now replaced with calls to $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, \cdot)$.
- $b \leftarrow \mathbf{Member}(\mathsf{sk}, x)$:
  1. Parse $\mathsf{sk} := (\mathsf{msk}', x')$. Write $x \in \{0,1\}^{\log n}$ as a binary string and let $z := 0^B || x$. If $x' \neq \bot$, write $x' \in \{0,1\}^{\log n}$ as a binary string and let $z' := 0^B || x'$.
  2. Output 1 if for every $i \in [\frac{\log n}{2} + B]$, the following holds: either $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}', z[i :]) = 1$ or ($x' \neq \bot$ and $z[i :] = z'[i :]$). Else, output 0.
- $\mathsf{sk}_{+x} \leftarrow \mathbf{Add}(\mathsf{sk}, x)$: parse $\mathsf{sk} := (\mathsf{msk}', \bot)$, and output $\mathsf{sk}_{+x} := (\mathsf{msk}', x)$.
- $\mathsf{rsk}_{-x} \leftarrow \mathbf{ReSamp}(\mathsf{sk}, x)$:
  1. Parse $\mathsf{sk} := (\mathsf{msk}', x')$, and write $x \in \{0,1\}^{\log n}$ as a binary string and let $z := 0^B || x$.
  2. Sample uniformly random $v \xleftarrow{\$} \{0,1\}^{\frac{\log n}{2} + B}$, and let $P := \{(z[i :], v[i])\}_{i \in [\frac{\log n}{2} + B]}$.
  3. If $x' \neq \bot$, do the following. Write $x' \in \{0,1\}^{\log n}$ as a binary string, and let $z' := 0^B || x'$. For $i \in [\frac{\log n}{2} + B]$, if $z'[i :] \neq z[i :]$, add the constraint $(z'[i :], 1)$ to the set P.
  4. Compute $\mathsf{rsk}_{-x} \leftarrow \mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}', P)$, and output rsk_{-x}.

Additional helpful notations. In our PIR scheme later, we will only need to call set enumeration for final keys rsk. Therefore, our algorithm description above defines $\mathbf{Set}(\mathsf{rsk})$ only for final keys. However, in our proofs and narratives, it helps to define the set associated with an intermediate key sk as well — however, in this case we need not worry about the running time of $\mathbf{Set}(\mathsf{sk})$. This is defined in the most natural manner:

- If $\mathsf{sk} = (\mathsf{msk}, \bot)$ is the direct output of $\mathbf{Gen}(1^\lambda, n)$, then $\mathbf{Set}(\mathsf{sk})$ is defined just like in Section 3.3 except that calls to $\mathsf{RO}(\cdot)$ are replaced with $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$;

- If $\mathsf{sk} = (\mathsf{msk}, x)$ is the output of an earlier **Add** operation, then **Set**(sk) is defined just like in Section 3.3 except that calls to $\mathsf{RO}(\cdot)$ are replaced with the following outcomes: 1) we force the outcomes to be 1 at the input points $\{(0^B||x)[i:]\}_{i \in [\frac{\log n}{2}+B]}$; and 2) for all other inputs, we call $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ to obtain the outcome.

Performance bounds. **Gen**($1^\lambda, n$) takes $\mathsf{poly}(\lambda, \log n)$ time. Due to Fact 3.9, **Set**(rsk) takes $\sqrt{n} \cdot \mathsf{poly}\log(\lambda, n)$ time with $1 - 1/\log n$ probability. **Member**(sk, x) takes $\mathsf{poly}(\lambda, \log n)$ time. **Add**(sk, x) takes constant time. **ReSamp**(sk, x) takes $\mathsf{poly}(\lambda, \log n)$ time.

Circuit for set enumeration. Later in our PIR scheme, during the offline phase, the server needs to perform set enumeration under fully homomorphic encryption. Therefore, we need to describe how to perform set enumeration in circuit. We will describe a circuit construction of size at most $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$ which obtains as input a final key rsk, and outputs a set $S = \{(x_1, b_1), (x_2, b_2), \ldots\}$ of size at most $2\sqrt{n}\log^2 n$ with distinct x's, and a bit bSucc indicating success. We want to ensure that if $\mathsf{bSucc} = \mathsf{True}$, then the set generated is correct in the following sense:

- for every $(x, 1) \in S$, x is in the correct set defined by $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, \cdot)$; and
- for every element x in the set defined by $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, \cdot)$, the pair $(x, 1)$ appears in S.

Our circuit construction emulates the set enumeration algorithm of Section 3.3. Our circuit construction works as follows — henceforth we use the term "hash" to mean the computing outcome of $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, \cdot)$:

Circuit for set enumeration $\mathsf{CSetEnum}$

1. Let $\mathsf{bSucc} = \mathsf{True}$.
2. For every $x \in \{0,1\}^{\frac{1}{2}\log n+1}$, let $b_x = \mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, x)$. Output an array containing $\{(x, b_x)\}_{x \in \{0,1\}^{\frac{1}{2}\log n+1}}$.
3. Obliviously sort above array such that entries with $b_x = 1$ are moved to the front. Truncate the array at length $2\sqrt{n}\log^2 n$ elements, and if the truncation removes any string that hash to 1, set $\mathsf{bSucc} = \mathsf{False}$. Let $Z_{\frac{1}{2}\log n+1}$ be the resulting truncated array, where each entry is of the form (x, b_x).
4. For $\ell = \frac{1}{2}\log n + 2$ to $\log n$, do the following:
   - For each $(x, b_x) \in Z_{\ell-1}$, if $b_x = 1$, write down $(0||x, \mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, 0||x))$ and $(1||x$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}$, $1||x))$; else write down $(0||x, 0)$ and $(1||x, 0)$.
   - Oblivious sort the resulting array such that all entries marked with 1 move to the front. Truncate the resulting array at length exactly $2\sqrt{n}\log^2 n$. If the truncation removes any string that hash to 1, set $\mathsf{bSucc} = \mathsf{False}$. Let Z_ℓ denote the resulting array where each entry is of the form (x, b_x).
5. For every $(x, b_x) \in Z_{\log n}$, check if it is the case that for every $j \in [B]$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, 0^j||x) = 1$. If so, write down (x, b_x), else, write down $(x, 0)$. Output the resulting array as well as bSucc.

Fact 4.2. *Using the AKS sorting network [AKS83] or the bitonic sorting network [Bat68] to realize the oblivious sort, the above algorithm can be implemented with a circuit of size* $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$.

Proof. The proof is straightforward given the fact that the AKS sorting circuit has size $O(n' \log n')$ for sorting n' elements, and the bitonic sorting network has size $O(n' \log^2 n')$. Also, note that each $\mathbf{PEval}(\mathsf{rsk}, \cdot)$ consumes $\mathsf{poly}(\lambda, \log n)$ gates to implement. □

For correctness, we will imagine that the above algorithm is run where $\mathsf{PRF}.\mathbf{PEval}(\mathsf{rsk}, \cdot)$ is replaced with calls to a random oracle RO — we denote the resulting algorithm as $\mathsf{CSetEnum}^{\mathsf{RO}}$. Note that we do not care about the computational model when stating the correctness probability.

Fact 4.3. *Suppose that $n \geq 4$. For any $x \in \{0, 1, \dots, n-1\}$,*

$$\Pr_{\mathsf{RO} \xleftarrow{\$} \mathcal{D}_n^{+x}} \left[\mathsf{CSetEnum}^{\mathsf{RO}} \text{ } \textit{outputs} \text{ } \mathsf{bSucc} = \mathsf{True} \right] \geq 1 - 1/\log n,$$

Moreover,

$$\Pr_{\mathsf{RO} \xleftarrow{\$} \mathcal{D}_n} \left[\mathsf{CSetEnum}^{\mathsf{RO}} \text{ } \textit{outputs} \text{ } \mathsf{bSucc} = \mathsf{True} \right] \geq 1 - 1/\log n$$

Proof. $\mathsf{CSetEnum}^{\mathsf{RO}}$ is a direct implementation of the set enumeration algorithm in Section 3.3 except that we truncate each Z_ℓ to size exactly $2\sqrt{n} \log^2 n$. Shi et al. [SACM21] proved that no matter whether RO is sampled from $\mathcal{D}_n^{+x}$ or $\mathcal{D}_n$, with $1 - 1/\log n$ probability, the following good event holds: for all $\ell \in [\frac{\log n}{2} + 1, \log n]$, $|Z_\ell| \leq 2\sqrt{n} \log^2 n$ — see the proof of Lemma 6.4 in their paper. The algorithm outputs $\mathsf{bSucc} = 1$ as long as the above good event holds. □

5 PIR Scheme

We now describe a PIR scheme that supports a bounded number of queries denoted Q. Given this scheme, we can compile it to a scheme that supports unbounded number of queries by performing the offline setup phase every Q queries, and amortizing this cost over the Q queries.

Intuition. In the offline setup phase, the client chooses $\widetilde{O}(Q)$ keys each of which defines a pseudorandom set of size roughly $\sqrt{n}$. It encrypts these keys under a fully homomorphic encryption (FHE) scheme, and sends the encrypted keys to the server. Through homomorphic evaluation, the server enumerates the sets and computes the encrypted parity (i.e., an encryption of $\oplus_{x \in S} \mathsf{DB}[x]$) for each of these sets S, and returns the encrypted parities to the client. The client decrypts the parities, and stores each set's key as well as its parity. These sets are divided into two parts: the last Q entries are called the *backup* sets or entries, and the remaining are called the *primary* sets or entries. The primary entries are used for answering queries, whereas the backup entries are later promoted to become primary entries as they get consumed. Henceforth, we also use the terms primary table and backup table to refer to the tables that store all primary entries and backup entries, respectively.

In the online phase, whenever the client wants to make a query for the database's value at index $x \in \{0, 1, \dots, n-1\}$, it finds the first primary set (sk_i, p_i) such that $\mathbf{Set}(\mathsf{sk}_i)$ contains the query x. It then resamples the decision whether x is in the set or not, and obtains a programmed key. It sends this programmed key to the server, which calls the set enumeration algorithm to enumerate the set S generated by the key. The server then returns the parity p of the set S to the client. The client computes $p_i \oplus p$ as the candidate answer to the query. Since the resampling operation removes the element x from the set with high probability, the candidate answer is correct with high probability. The correctness probability can be further boosted by repeating the same scheme k times and taking the majority vote among the k copies.

Detailed construction. We describe the detailed construction below.

PIR Scheme for $Q = \sqrt{n}$ queries

Run $k = \omega(\log \lambda)$ parallel copies of the single-copy scheme described below.

Offline phase:

- **Client**: // let $\mathsf{lenT} := 6\sqrt{n} \cdot \log^3 n$
  - $\mathsf{fsk} \leftarrow \mathsf{FHE}.\mathbf{Gen}(1^\lambda)$;
  - For $i \in [k \cdot (\mathsf{lenT} + Q)]$ where $k = \omega(\log \lambda)$, $\mathsf{sk}_i \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$, $\overline{\mathsf{sk}}_i \leftarrow \mathsf{FHE}.\mathbf{Enc}(\mathsf{fsk}, \mathsf{sk}_i)$;
  - Send $(\overline{\mathsf{sk}}_1, \ldots, \overline{\mathsf{sk}}_{k \cdot (\mathsf{lenT}+Q)})$ to the server.
- **Server**:
  - For $i \in [k \cdot (\mathsf{lenT} + Q)]$, $(\overline{S}_i, \overline{\mathsf{bSucc}_i}) \leftarrow \mathsf{FHE}.\mathbf{Eval}(\mathsf{CSetEnum}, \overline{\mathsf{sk}}_i)$;
  - $\{\overline{p}_i\}_{i \in [k \cdot (\mathsf{lenT}+Q)]} \leftarrow \mathsf{FHE}.\mathbf{Eval}(\mathsf{CBatchParity}, \overline{S}_1, \ldots, \overline{S}_{k \cdot (\mathsf{lenT}+Q)})$, where the $\mathsf{CBatchParity}$ circuit is described below. Send $\{\overline{p}_i, \overline{\mathsf{bSucc}_i}\}_{i \in [k \cdot (\mathsf{lenT}+Q)]}$ to the client.
- **Client**:
  - for $i \in [k \cdot (\mathsf{lenT} + Q)]$, $p_i \leftarrow \mathsf{FHE}.\mathbf{Dec}(\mathsf{fsk}, \overline{p}_i)$; $\mathsf{bSucc}_i \leftarrow \mathsf{FHE}.\mathbf{Dec}(\mathsf{fsk}, \overline{\mathsf{bSucc}_i})$;
  - choose a subset $I \subseteq [k \cdot (\mathsf{lenT} + Q)]$ of size exactly $\mathsf{lenT} + Q$ such that for any $i \in I$, $\mathsf{bSucc}_i = \mathsf{True}$ — if not enough such entries are found, simply abort. Copy $\{(\mathsf{sk}_i, p_i)\}_{i \in I}$ to a table.

  We call the last Q entries of the above table the *backup* table, henceforth renamed to $T^* := \{(\mathsf{sk}^*_i, p^*_i)\}_{i \in [Q]}$. We call the remaining lenT entries the *primary* table, henceforth renamed to $T := \{(\mathsf{sk}_i, p_i)\}_{i \in [\mathsf{lenT}]}$.

Online query for index $x \in \{0, \ldots, n-1\}$:

- **Client**:
  - Sample $\mathsf{sk} \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$ and append the entry $(\mathsf{sk}, 0)$ to the table T of primary sets;
  - Find the first entry (sk_i, p_i) in T such that $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_i, x) = 1$;
  - Compute $\mathsf{rsk} \leftarrow \mathsf{PRSet}.\mathbf{ReSamp}(\mathsf{sk}_i, x)$ and send rsk to the server.
- **Server**: Compute $S \leftarrow \mathsf{PRSet}.\mathbf{Set}(\mathsf{rsk})$, and return the parity bit p of the set S to the client. If the set enumeration algorithm has not completed even after making $6\sqrt{n}\log^5 n$ calls to the underlying PRF's $\mathbf{PEval}(\mathsf{rsk}, \cdot)$ function, then return $p = 0$ to the client.
- **Client**: let $\beta' := p \oplus p_i$ be the candidate answer of the current copy, and remove the last entry of T.

  Recall that there are k parallel instances, and let β be the majority vote among the candidate answers of all k copies. Now, let (sk^*_j, p^*_j) denote the next available backup set and perform the following:

- let $\mathsf{sk}' \leftarrow \mathsf{PRSet}.\mathbf{Add}(\mathsf{sk}^*_j, x)$; let $p' := p^*_j \oplus \beta$ if $\mathbf{Member}(\mathsf{sk}^*_j, x) = 0$, else let $p' := p^*_j$;
- let $T_j := (\mathsf{sk}', p')$, and mark the backup entry (sk^*_j, p^*_j) as unavailable.

The circuit $\mathsf{CBatchParity}$**.** The circuit $\mathsf{CBatchParity}$ takes $S_1, S_2, \ldots, S_{k\cdot(\mathsf{lenT}+Q)}$ as input, where for $j \in [k \cdot (\mathsf{lenT} + Q)]$, S_j contains exactly $2\sqrt{n}\log^2 n$ entries of the form (x, b_x) — specifically, $b_x = \mathsf{True}$ implies that x is the j-th set and $b_x = \mathsf{False}$ implies x is not in the j-th set[3]. The circuit outputs $k \cdot (\mathsf{lenT} + Q)$ parity bits $p_1, \ldots, p_{k\cdot(\mathsf{lenT}+Q)}$ of each of the $k \cdot (\mathsf{lenT} + Q)$ sets.

The circuit can be constructed as follows using oblivious sort:

1. Let $\mathsf{DB} \in \{0,1\}^n$ be the server's database, let $\mathbf{D} := ((0, \mathsf{DB}[0]), (1, \mathsf{DB}[1]), ..., (n-1, \mathsf{DB}[n-1]))$.
2. For $j \in [k \cdot (\mathsf{lenT} + Q)]$, let $\mathbf{X}_j = \{(x, b_x, j)\}_{x \in S_j}$
3. Obliviously sort the array $\mathbf{Y} := \mathbf{D}||\mathbf{X}_1|| \ldots ||\mathbf{X}_{k\cdot(\mathsf{lenT}+Q)}$, such that each entry of the form $(x, \mathsf{DB}[x])$ is followed by all tuples of the form (x, b_x, j). Henceforth, we call a tuple of the form (x, b_x, j) a consumer.
4. In a linear scan, all consumers receive the $\mathsf{DB}[x]$ they are requesting. At this moment, each consumer entry is updated to $(x, b_x, j, \mathsf{DB}[x])$.
5. Use a circuit that mirrors the oblivious sort circuit in Step 3, and reverse-routes the $\mathsf{DB}[x]$ values back to the position where it came from. As a result, each consumer entry of the form $(x, b_x, j) \in \mathbf{Y}$ receives $\mathsf{DB}[x]$.
6. At this moment, we have an array of the form $\mathbf{X}'_1|| \ldots ||\mathbf{X}'_{k\cdot(\mathsf{lenT}+Q)}$, where each $\mathbf{X}'_j$ contains exactly $2\sqrt{n}\log^2 n$ entries of the form $(x, b_x, j, \mathsf{DB}[x])$. In a linear scan, we can compute for each $j \in [k \cdot (\mathsf{lenT} + Q)]$, the parity bit
$$p_j = \oplus_{(x, b_x, j, \mathsf{DB}[x]) \in \mathbf{X}'_j}(b_x \cdot \mathsf{DB}[x])$$

It is not hard to see that if we instantiate the oblivious sort using either AKS [AKS83] or bitonic sort [Bat68], and given $\mathsf{lenT} = 6\sqrt{n}\log^3 n$ and $Q = \sqrt{n}$, the above circuit has size $O(n \cdot \mathsf{poly}\log n)$.

Performance bounds. We now analyze the performance bounds of our Q-bounded PIR construction. We may plug in $k = \log^{1.1} n$ since any super-logarithmic function will work. In the analysis below, the k parameter is absorbed in the $\mathsf{poly}\log n$ term, so it does not show up explicitly.

- *Offline phase.* During the offline phase, the client's computation and bandwidth are upper bounded by $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$. The server's computation is upper bounded by $n \cdot \mathsf{poly}(\lambda, \log n)$.
- *Online phase.* The bandwidth is $\mathsf{poly}(\lambda, \log n)$. The client's computation is $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$. The server's computation is also $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$.

Supporting unbounded number of queries and deamortization. To extend the scheme from Q-bounded to supporting an unbounded number of queries, we just need to rerun the offline phase every $Q = \sqrt{n}$ queries. For the scheme with unbounded queries, the amortized bandwidth per query is $\mathsf{poly}(\lambda, \log n)$, the amortized client and server computation per query is $\sqrt{n} \cdot \mathsf{poly}(\lambda, \log n)$.

This periodic offline setup can be deamortized very easily. Specially, upfront, we perform the offline setup for $2Q$ queries. During the i-th window of Q queries, we perform the offline setup for

[3] This input format is inherited from the output format of the circuit $\mathsf{CSetEnum}$.

the $(i+2)$-th window of Q queries, and so on. This way, when the $(i+2)$-th window of Q queries starts, the corresponding offline setup will be ready. With deamortization, there is a factor of 2 blowup in storage. There is no additional blowup in terms of amortized computational cost.

6 Privacy Proof

Recall that privacy for a single-server PIR scheme was defined earlier in Section 3.2. We now prove that our PIR scheme in Section 5, when instantiated with the PRSet scheme in Section 4.2, satisfies privacy, as stated in the following theorem.

Theorem 6.1 (Privacy of our PIR scheme)**.** *Suppose that the FHE scheme employed satisfies semantic security, and that the underlying programmable PRF scheme satisfies correctness, private programmability, and simulation security. Then, the PIR scheme in Section 5, when instantiated with the* PRSet *scheme in Section 4.2, satisfies privacy.*

In the remainder of this section, we will prove the above theorem.

6.1 Proof Roadmap

A key insight in our privacy proof is to rely on a *lazy sampling* technique to decompose the *backend* and the *frontend* of a complicated randomized experiment, where the *backend* refers to the primary table stored by the client, and the *frontend* refers to the message the clients sends to the server during each query. Below, we explain the proof intuition, and the formal proofs can be found in Section 6.2 and Appendix C.3.

We start from the real-world experiment, where the client interacts with the server like in the real-world scheme. First, in Hyb_1, we replace the FHE ciphertexts the client sends to the server in the offline phase with encryptions of 0. Therefore, henceforth we will not be worried about these FHE ciphertexts, and we will focus on what happens in the online phase. In our full proof in Appendix C.3, the key is how to get from Hyb_2 to Hyb_6, which are described below.

Table 2: Hyb_2 and Hyb_6.

Hybrid	**Backend** promoted key during query y	**Frontend** during query x
Hyb_2	$\mathsf{msk} \leftarrow \mathbf{Gen}$, $\mathsf{sk} := (\mathsf{msk}, y)$	• find $\mathsf{sk} := (\mathsf{msk}, y)$ in T s.t. msk contains x after adding y if $y \neq \perp$ • program msk s.t. $\mathsf{suffixes}(x)$ are resampled and if $y \neq \perp$, $\mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$ forced to 1
Hyb_6	$\mathsf{msk} \leftarrow \mathbf{Gen}$ s.t. $y \in \mathbf{Set}(\mathsf{msk})$	• find msk in T s.t. $x \in \mathbf{Set}(\mathsf{msk})$, • program msk s.t. $\mathsf{suffixes}(x)$ are resampled

If we can get to Hyb_6, the rest of the proof can be completed in a similar manner as Shi et al. [SACM21]'s proof. Therefore, the key is how to get from Hyb_2 to Hyb_6. To accomplish this, we introduce a lazy sampling idea to "decouple" the backend and the frontend in our proof.

Hyb_3**: introduce lazy sampling.** We define a hybrid experiment Hyb_3 that is an equivalent rewrite of Hyb_2 by lazy sampling in the following sense.

1. *Backend: maintain constraints on each entry in T that defines the a-posteriori distribution.* Let $\mathbf{I} = \{i_1, i_2, \ldots, i_q\}$ be the indices of the entries that are matched during each of the $q \leq Q$ queries so far. The client maintains the a-posteriori distribution of each entry of the primary table T conditioned on the local observation $\mathbf{I}$.

   To maintain the a-posteriori distribution, the client maintains a set of constraints of the form $\langle -x \rangle$, $\langle +x \rangle$, $\langle +y : -x \rangle$, or $\langle +y : +x \rangle$ on each entry. A negative constraint of the form $\langle -x \rangle$ means that this entry was not promoted from the backup table, and we have checked that x is not in the set generated by the key, during some query for x. A negative constraint of the form $\langle +y : -x \rangle$ means that this entry was promoted from the backup table during a query for y, and we have checked that after forcing y to be in the set, x is not in the set generated by the key. The positive constraints $\langle +x \rangle$ and $\langle +y : +x \rangle$ are similarly defined but requiring x to be in the set.

   During an online query for some x, the client sequentially scans through the current entries of T. For each entry j, it samples from the a-posteriori distribution to decide if j should be the match. Depending on the decision, it adds either a negative or positive constraint to the current entry.

2. *Frontend: lazy sampling from the a-posteriori distribution.* Whenever the client is about to send a key to the server, it performs lazy sampling of the key based on the a-posteriori distribution on the entry that the client has maintained. More specifically, there are two cases depending on whether the matched entry comes from the backup table or not : 1) it samples a key from the correct a-posteriori distribution, calls **ReSamp** and sends the resulting key to the server; 2) it samples a key from the correct a-posteriori distribution, calls both **Add** and **ReSamp**, and then sends the resulting key to the server.

In our proof, we show that except with negligible probability, the constraints maintained on any entry can be satisfied with inverse polynomial probability for a randomly sampled key.

Hyb_4**: switch the backend.** Next, in Hyb_4, we change the backend to be like in Hyb_6, and the client uses the resulting table T to decide which entries are matched during each query, and just like in Hyb_3, the client maintains a set of constraints on each entry of the table, such that the frontend can perform lazy sampling according to the a-posteriori distribution when interacting with the server. Note that this change technically affects the distribution of the matched entries during each query, and thus affects the distribution of the server's view. Fortunately, using the security of the privately programmable PRF, we can prove that even when we make this change on the backend, the server's view remains computationally indistinguishable[4].

Hyb_4 **to** Hyb_6**: switch the frontend.** Next, from Hyb_4 to Hyb_6, we change the way the frontend performs the lazy sampling from the method of Hyb_3 to the method of Hyb_6. To complete this proof, we do it in two steps using Hyb_5 as a stepping stone. In Hyb_4, after lazy sampling a key according to the maintained constraints, we program $\mathsf{suffixes}(x)$ to be random values and if $y \neq \perp$, we program $\mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$ to be 1. In Hyb_5, we remove all the programming and replace it with rejection sampling of simulated keys. In Hyb_6, we introduce back the part of the programming, and we

[4] Note that we need NOT prove that the joint distribution of the backend and the frontend are computationally indistinguishable, we only need to prove that the frontend, i.e., server's view is computationally indistinguishable.

program only $\mathsf{suffixes}(x)$ to be random values, while the part $\mathsf{suffixes}(y)\backslash\mathsf{suffixes}(x)$ being forced to be 1 is achieved through rejection sampling. To show that Hyb_4 and Hyb_5 are computationally indistinguishable and that Hyb_5 and Hyb_6 are computationally indistinguishable, we need to make use of the security property of the privately programmable PRF. Some technicalities arise in this proof, since the security definitions of the privately programmable PRF are not in a form that we can use conveniently here. Therefore, as a key stepping stone, we introduce a *key technical lemma* (see Section 6.2), that will help us prove the transitions between Hyb_4 and Hyb_5, and between Hyb_5 and Hyb_6 more easily. Further, this key technical lemma can be proven using the security definitions of the privately programmable PRF.

Hyb_6**: convergence of backend and frontend.** One important observation is that in Hyb_b, the frontend and the entry found in the table during each query have the same distribution (modular some post-processing). Therefore, in this step, the backend and the frontend converge again, and this is why we can undo the lazy sampling at this point, and Hyb_6 can be equivalently viewed as in Table 2.

6.2 Technical Lemma for Privately Programmable PRF

We shall consider a programmable PRF whose output range is binary, i.e., $\{0, 1\}$. Henceforth, we use the notation $\mathsf{pred}^X(\mathsf{msk})$ to denote an event that looks at the outputs of $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ at all inputs in X, and outputs either 0 or 1. We say that $\mathsf{pred}^X(\cdot)$ is an *admissible* event, iff 1) for a randomly sampled $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$, it returns 1 with probability at least $1/p(\lambda)$ for some polynomial function $p(\cdot)$; and 2) pred is polynomial-time checkable.

Lemma 6.2 (Strong privacy of programmable PRF)**.** *Let* PRF *be a programmable PRF with a binary output range, and suppose that* $L = O(\log \lambda)$*. Suppose that* PRF *satisfies private programmability and simulation security. Then, there exists a probabilistic polynomial-time simulator* Sim *such that the following two experiments are computationally indistinguishable to any probabilistic polynomial-time adversary.*

- $\mathsf{RealPPRFStrong}(1^\lambda)$*:*
  - $X, X', \{v_x\}_{x\in X'}, \mathsf{pred}^{X\cup X'} \leftarrow \mathcal{A}(1^\lambda, L)$ *s.t.* $|X| + |X'| \leq L$*,* $X \cap X' = \emptyset$*, and* $\mathsf{pred}^{X\cup X'}(\cdot)$ *is admissible;*
  - *for* $x \in X$*, let* $v_x \overset{\$}{\leftarrow} \mathcal{V}$*; let* $P := \{(x, v_x)\}_{x\in X\cup X'}$*;*
  - *sample* $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$ *subject to* $\mathsf{pred}^{X\cup X'}(\mathsf{msk}) = 1$*, and let* $\mathsf{sk} \leftarrow \mathbf{Prog}(\mathsf{msk}, P)$*;*
  - $\mathsf{sk} \rightarrow \mathcal{A}$*;*
- $\mathsf{IdealPPRFStrong}(1^\lambda)$*:*
  - $X, X', \{v_x\}_{x\in X'}, \mathsf{pred}^{X\cup X'} \leftarrow \mathcal{A}(1^\lambda, L)$ *s.t.* $|X| + |X'| \leq L$*,* $X \cap X' = \emptyset$*, and* $\mathsf{pred}^{X\cup X'}(\cdot)$ *is admissible;*
  - *sample* $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$ *subject to the constraint that for any* $x \in X'$*,* $\mathbf{PEval}(\mathsf{sk}, x) = v_x$*;*
  - $\mathsf{sk} \rightarrow \mathcal{A}$*.*

In the real experiment $\mathsf{RealPPRFStrong}$, we sample a random key subject to some admissible predicate on X and X', and then program X to be random and program X' to be values of the adversary $\mathcal{A}$'s choice (e.g., all 1s). The lemma states that the real experiment $\mathsf{RealPPRFStrong}$ is computationally indistinguishable from an ideal experiment $\mathsf{IdealPPRFStrong}$ where we simply sample a random simulated key subject to the set of points X' evaluating to the choices specified by $\mathcal{A}$. Note that in $\mathsf{IdealPPRFStrong}$, we do not perform any programming at all, and replace it with rejection sampling that checks if the set of points in X' evaluate to the choices specified by $\mathcal{A}$.

The intuition is the following. In the real experiment, we sample an msk subject to some predicate pred. The observation is that it does not matter what predicate pred we check, since we eventually reprogram the points in $X \cup X'$, and recall that we require the predicate pred to only look at the PRF's outcomes on $X \cup X'$. Effectively, the reprogramming cancels the effect of the sampling subject to a predicate pred that looks at only $X \cup X'$. In fact, the distribution of the final programmed key is indistinguishable from the ideal experiment, where we simply sample a simulated key that evaluates to adversary-specified values on the set X'.

Deferred Materials

We defer the full privacy proof, the correctness proof of our PIR scheme, how to tune the tradeoff between client storage and the online computation, as well as additional preliminaries to the appendices.

Acknowledgment

This work is in part supported by a grant from ONR, a gift from Cisco, NSF awards under grant numbers CIF-1705007, 2128519 and 2044679, and a Packard Fellowship.

References

[ABOR00] William Aiello, Sandeep N. Bhatt, Rafail Ostrovsky, and Sivaramakrishnan Rajagopalan. Fast verification of any remote procedure call: Short witness-indistinguishable one-round proofs for NP. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *ICALP 2000*, volume 1853 of *LNCS*, pages 463–474. Springer, Heidelberg, July 2000.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.

[AIVG22] Kinan Dak Albab, Rawane Issa, Mayank Varia, and Kalman Graffi. Batched differentially private information retrieval. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3327–3344. USENIX Association, August 2022.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *15th ACM STOC*, pages 1–9. ACM Press, April 1983.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *AFIPS*, 1968.

[BGI16] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1292–1303. ACM Press, October 2016.

[BGIK22] Elette Boyle, Niv Gilboa, Yuval Ishai, and Victor I. Kolobov. Programmable distributed point functions. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part IV*, volume 13510 of *LNCS*, pages 121–151. Springer, Heidelberg, August 2022.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg, August 2000.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 662–693. Springer, Heidelberg, November 2017.

[BK22] James Bartusek and Dakshita Khurana. Cryptography with certified deletion. Cryptology ePrint Archive, Report 2022/1178, 2022. https://eprint.iacr.org/2022/1178.

[BKM17] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 415–445. Springer, Heidelberg, April / May 2017.

[BLW17] Dan Boneh, Kevin Lewi, and David J. Wu. Constraining pseudorandom functions privately. In Serge Fehr, editor, *PKC 2017, Part II*, volume 10175 of *LNCS*, pages 494–524. Springer, Heidelberg, March 2017.

[BMW98] Ingrid Biehl, Bernd Meyer, and Susanne Wetzel. Ensuring the integrity of agent-based computations by short proofs. In *International Workshop on Mobile Agents*, pages 183–194. Springer, 1998.

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private constrained PRFs (and more) from LWE. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 264–302. Springer, Heidelberg, November 2017.

[CC17] Ran Canetti and Yilei Chen. Constraint-hiding constrained PRFs for NC^1 from LWE. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 446–476. Springer, Heidelberg, April / May 2017.

[CG97] Benny Chor and Niv Gilboa. Computationally private information retrieval (extended abstract). In *29th ACM STOC*, pages 304–313. ACM Press, May 1997.

[CGKS95] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, October 1995.

[Cha04] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 50–61. Springer, Heidelberg, July 2004.

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

Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 3–33. Springer, Heidelberg, May / June 2022.

[CHR17] Ran Canetti, Justin Holmgren, and Silas Richelson. Towards doubly efficient private information retrieval. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 694–726. Springer, Heidelberg, November 2017.

[CK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Heidelberg, May 2020.

[CKGS98] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, November 1998.

[CMS99] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.

[dCP22] Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part I*, volume 13275 of *LNCS*, pages 150–179. Springer, Heidelberg, May / June 2022.

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

[DHRW16] Yevgeniy Dodis, Shai Halevi, Ron D. Rothblum, and Daniel Wichs. Spooky encryption and its applications. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 93–122. Springer, Heidelberg, August 2016.

[DHS14] Daniel Demmler, Amir Herzberg, and Thomas Schneider. Raid-pir: Practical multi-server pir. In *CCSW*, 2014.

[DIO98] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal service-providers for database private information retrieval (extended abstract). In Brian A. Coan and Yehuda Afek, editors, *17th ACM PODC*, pages 91–100. ACM, June / July 1998.

[DNR16] Cynthia Dwork, Moni Naor, and Guy N. Rothblum. Spooky interaction and its discontents: Compilers for succinct two-message argument systems. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part III*, volume 9816 of *LNCS*, pages 123–145. Springer, Heidelberg, August 2016.

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

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *41st ACM STOC*, pages 169–178. ACM Press, May / June 2009.

[GI14] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 803–815. Springer, Heidelberg, July 2005.

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 75–92. Springer, Heidelberg, August 2013.

[Hen16] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, October 2016.

[HH17] Syed Mahbub Hafiz and Ryan Henry. Querying for queries: Indexes of queries for efficient and expressive IT-PIR. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1361–1373. ACM Press, October / November 2017.

[HOWW19] Ariel Hamlin, Rafail Ostrovsky, Mor Weiss, and Daniel Wichs. Private anonymous data access. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 244–273. Springer, Heidelberg, May 2019.

[IKOS04] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004.

[IKOS06] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *47th FOCS*, pages 239–248. IEEE Computer Society Press, October 2006.

[KC21] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021*, pages 875–892. USENIX Association, August 2021.

[KO97] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, October 1997.

[KRR13] Yael Tauman Kalai, Ran Raz, and Ron D. Rothblum. Delegation for bounded space. In Dan Boneh, Tim Roughgarden, and Joan Feigenbaum, editors, *45th ACM STOC*, pages 565–574. ACM Press, June 2013.

[KW21] Sam Kim and David J. Wu. Watermarking cryptographic functionalities from standard lattice assumptions. *Journal of Cryptology*, 34(3):28, July 2021.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg, January 2015.

[Lip10] Helger Lipmaa. First CPIR protocol with data-dependent computation. In Donghoon Lee and Seokhie Hong, editors, *ICISC 09*, volume 5984 of *LNCS*, pages 193–210. Springer, Heidelberg, December 2010.

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

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response efficient single-server PIR. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2292–2306. ACM Press, November 2021.

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, high-rate single-server PIR via FHE composition. In *2022 IEEE Symposium on Security and Privacy*, pages 930–947. IEEE Computer Society Press, May 2022.

[OS07] Rafail Ostrovsky and William E. Skeith III. A survey of single-database private information retrieval: Techniques and applications (invited talk). In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC 2007*, volume 4450 of *LNCS*, pages 393–411. Springer, Heidelberg, April 2007.

[PPY18] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1002–1019. ACM Press, October 2018.

[PR93] Pavel Pudlák and Vojtech Rödl. Modified ranks of tensors and the size of circuits. In *25th ACM STOC*, pages 523–531. ACM Press, May 1993.

[PS18] Chris Peikert and Sina Shiehian. Privately constraining and programming PRFs, the LWE way. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part II*, volume 10770 of *LNCS*, pages 675–701. Springer, Heidelberg, March 2018.

[PY22] Giuseppe Persiano and Kevin Yeo. Limits of preprocessing for single-server PIR. In *SODA*, pages 2522–2548. SIAM, 2022.

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 641–669, Virtual Event, August 2021. Springer, Heidelberg.

[TDG16] Raphael R. Toledo, George Danezis, and Ian Goldberg. Lower-cost ϵ-private information retrieval. *PETS*, 2016.

A Optimizing the Polylogarithmic Factors

In the main body, we did not care about optimizing the exact polylogarithmic factors in the asymptotical bounds. We can tighten the analyses and get the following amortized costs where $\alpha(\cdot)$ can be any super-constant function, and λ and λ' denote the computational and statistical security parameters, respectively:

- *client computation*: $O_\lambda(\sqrt{n} \cdot \log^2 n \cdot \log \lambda') \cdot \alpha(\lambda')$,
- *server computation*: $O_\lambda(\sqrt{n} \cdot \log n \cdot \log \lambda') \cdot \alpha(\lambda')$,
- *bandwidth*: $O_\lambda(\log n \cdot \log \lambda') \cdot \alpha(\lambda')$, and

- *client storage*: $O_\lambda(\sqrt{n} \cdot \log^2 n \cdot \log(\lambda')) \cdot \alpha(\lambda')$.

In the above, the $\log(\lambda') \cdot \alpha(\lambda')$ terms arise from the super-logarithmic copies of the PIR scheme. The notation $O_\lambda(\cdot)$ hides a computational security parameter λ related to the strength of the cryptographic primitives including FHE and privately programmable PRF. In the bandwidth and storage costs, one $\log n$ factor arises from the fact that we need to program $\log n$ points. The evaluation overhead for the programmable PRF need not suffer from this extra $\log n$ factor using the techniques in Appendix B of Shi et al. [SACM21].

To get this tighter analysis, basically we need to do the following: 1) choose parameters like Section 5.4 of Shi et al. [SACM21]; 2) amortize the setup over $\sqrt{n}\log n$ queries rather than $\sqrt{n}$ queries; and 3) use linear-size oblivious compaction circuit in $\mathsf{CSetEnum}$ rather than oblivious sorting.

Below we give more details about the parameters. Like Section 5.4 of Shi et al. [SACM21], we can set $B = \log\log n + C$ for some suitably large constant C. The expected size of a pseudorandom set is $O(\sqrt{n}/\log n)$. Both the primary set and backup set have size $O(\sqrt{n}\log n)$. For the $\mathsf{CSetEnum}$ circuit, we can budget $C' \cdot \sqrt{n}$ capacity per level (for some suitable constant C') to store the currently surviving candidates; in this way, it is not hard to show that the probability of overflow in $\mathsf{CSetEnum}$ is negligibly small. To show this, we first show through the Chernoff bound that except with negligible probability, the number of surviving candidates at the first level is at most $\sqrt{n} + n^{0.3}$. Now, conditioned on the above being true, we can show that except with negligible probability, the number of surviving candidates at the next level is at most $\sqrt{n} + 2 \cdot n^{0.3}$. We can now apply this argument for the logarithmically many levels. Note that the same argument also proves that except with negligible probability, the pseudorandom set size is at most $\sqrt{n}$.

B Smooth Tradeoff Between Space and Time

Throughout the paper, we focused on the special case when the client storage is $\widetilde{O}_\lambda(\sqrt{n})$, and the server and client computation is also $\widetilde{O}_\lambda(\sqrt{n})$ per query. Observe that the lower bound $S \cdot T \geq \Omega(n)$ by Corrigan-Gibbs et al. [CHK22] suggests a possible tradeoff between the client space S and the server/client computation per query T. Indeed, we can tune the parameters of our scheme to trade off the two parameters. The parameter choices are similar to Appendix A of Shi et al. [SACM21].

Suppose that we want the client's storage to be $\widetilde{O}_\lambda(f(n))$ for some function $f(n)$, and we want to guarantee $\widetilde{O}_\lambda(n/f(n))$ server/client computation per query. Moreover, suppose that $f(n) \in [\log^c n, \frac{n}{\log^c n}]$ for some suitable positive constant c. We can set the probability that any element $x \in \{0, 1, \ldots, n-1\}$ is included in the set to be $\frac{1}{f(n)\log^2 n}$. This can be accomplished by applying the PRF to any suffix of $0^B||x$ of length at least $\log n - \log f(n) + 1$, and checking that the outcomes are all 1. We can set the $\mathsf{lenT} = f(n)\log^3 n$ to make sure that Fact 3.8 still holds [SACM21]. As argued by Shi et al. [SACM21], the expected set enumeration time is now $O(\frac{n}{f(n)}\log n)$, and in the set enumeration algorithm, we can cap the number of calls to the PRF at $O(\frac{n}{f(n)} \cdot \log^5 n)$. Finally, we will set the batching parameter $Q = f(n)$.

With these parameters, the offline server time is $\widetilde{O}_\lambda(n)$ and the offline client time and bandwidth are $\widetilde{O}_\lambda(f(n))$. The online server and client time per query is $\widetilde{O}_\lambda(n/f(n))$, and the per-query bandwidth is $\widetilde{O}_\lambda(1)$. Since we need to perform the offline pre-processing every Q queries, we can amortize the cost of the offline phase over the Q queries. As a result, the amortized server and client time per query is $\widetilde{O}_\lambda(n/f(n))$, and the per-query bandwidth is $\widetilde{O}_\lambda(1)$.

C Full Privacy Proof

C.1 Proof of Lemma 6.2

Proof of Lemma 6.2. We consider the following intermediate hybrid experiment called Hyb. Hyb is almost the same as $\mathsf{RealPPRFStrong}$ except that when we sample the msk, it is sampled at random rather than subject to the constraint that $\mathsf{pred}^{X \cup X'}(\mathsf{msk}) = 1$.

Claim C.1. *Suppose that the programmable PRF satisfies simulation security. Then,* $\mathsf{RealPPRFStrong}$ *is computationally indistinguishable from* Hyb.

Proof. Let Sim' be the simulator as in the definition of simulation security. Through a straightforward reduction to simulation security, Hyb is computationally indistinguishable from the following hybrid experiment denoted H:

- $X, X', \{v_x\}_{x \in X'}, \mathsf{pred}^{X \cup X'} \leftarrow \mathcal{A}(1^\lambda, L)$ s.t. $|X| + |X'| \leq L$, $X \cap X' = \emptyset$, and $\mathsf{pred}^{X \cup X'}(\cdot)$ is admissible;
- for $x \in X$, let $v_x \overset{\$}{\leftarrow} \mathcal{V}$; let $P := \{(x, v_x)\}_{x \in X \cup X'}$;
- $\mathsf{sk} \leftarrow \mathsf{Sim}'(1^\lambda, P, L)$;
- $\mathsf{sk} \rightarrow \mathcal{A}$;

It suffices to show that H is computationally indistinguishable from $\mathsf{RealPPRFStrong}$. We show that if there is an efficient adversary $\mathcal{A}$ that can distinguish H and $\mathsf{RealPPRFStrong}$ with non-negligible probability, we can construct an efficient reduction $\mathcal{B}$ that breaks the simulation security of the PRF scheme with non-negligible probability. Specifically, $\mathcal{B}$ waits till $\mathcal{A}$ submits $X, X', \{v_x\}_{x \in X'}, \mathsf{pred}^{X \cup X'}$, it then chooses v_x at random for $x \in X$, and lets $P := \{(x, v_x)\}_{x \in X \cup X'}$. It gives P to its own challenger. It obtains a key sk from its own challenger. It then queries its challenger on the inputs $X \cup X'$, and checks if pred holds over the outcomes. If so, it gives sk to $\mathcal{A}$ and outputs the same guess as $\mathcal{A}$. Otherwise, it outputs a random guess.

If $\mathcal{B}$ is playing in the experiment $\mathsf{RealPPRF}$ with its own challenger, then, conditioned on $\mathsf{pred}^{X \cup X'}$ being true, $\mathcal{A}$'s view is identically distributed as $\mathsf{RealPPRFStrong}$. Else, if $\mathcal{B}$ is playing in the experiment $\mathsf{IdealPPRF}$ with its own challenger, then, conditioned on $\mathsf{pred}^{X \cup X'}$ being true, $\mathcal{A}$'s view is identically distributed as H.

Let p be the probability that the predicate $\mathsf{pred}^{X \cup X'}$ holds if $\mathcal{B}$ is playing in the experiment $\mathsf{RealPPRF}$, and let p' be the probability that the predicate $\mathsf{pred}^{X \cup X'}$ holds if $\mathcal{B}$ is playing in the experiment $\mathsf{IdealPPRF}$ with its challenger. Since $\mathsf{pred}^{X \cup X'}$ is admissible, p must be a non-negligible function. It must be that $|p - p'| \leq \mathsf{negl}(\lambda)$ since otherwise we can easily construct an efficient adversary that distinguishes $\mathsf{RealPPRF}$ and $\mathsf{IdealPPRF}$ with non-negligible probability.

Therefore, if $\mathcal{A}$ has a non-negligible advantage in distinguishing $\mathsf{RealPPRFStrong}$ and H, $\mathcal{B}$ has a non-negligible advantage in distinguishing $\mathsf{RealPPRF}$ and $\mathsf{IdealPPRF}$. □

Claim C.2. *Suppose that the programmable PRF satisfies private programmability. Then* Hyb *is computationally indistinguishable from* $\mathsf{IdealPPRFStrong}$*, where the simulator* Sim *is the same as in the private programmability definition.*

Proof. We show that if there is an efficient adversary $\mathcal{A}$ that can distinguish Hyb and $\mathsf{IdealPPRFStrong}$ with non-negligible probability, then we can construct an efficient reduction $\mathcal{B}$ that can break private programmability with non-negligible probability. $\mathcal{B}$ obtains $X, X', \{v_x\}_{x \in X'}, \mathsf{pred}^{X \cup X'}$ from $\mathcal{A}$. It then forwards $X \cup X'$ to its own challenger, and obtains sk from its own challenger. It then checks if it is the case that for every $x \in X'$, $\mathbf{PEval}(\mathsf{sk}, x) = v_x$. If so, it forwards sk to $\mathcal{A}$, and outputs whatever $\mathcal{A}$ outputs. Otherwise, $\mathcal{B}$ outputs a random guess. If $\mathcal{B}$ is playing the game $\mathsf{RealPPRF}$ with its own challenger, then conditioned on $\mathcal{A}$ receiving sk from $\mathcal{B}$, $\mathcal{A}$'s view is identically distributed as in Hyb. Else, if $\mathcal{B}$ is playing $\mathsf{IdealPPRF}$ with its challenger, then conditioned on $\mathcal{A}$ receiving sk from $\mathcal{B}$, $\mathcal{A}$'s view is identically distributed as $\mathsf{IdealPPRFStrong}$.

Let p be the probability that $\mathcal{B}$ forwards sk to $\mathcal{A}$ when $\mathcal{B}$ is playing $\mathsf{RealPPRF}$, and let p' be the corresponding probability when $\mathcal{B}$ is playing $\mathsf{IdealPPRF}$. Since $L \leq O(\log \lambda)$ and the PRF has a binary output domain, we know that $p \geq 1/\mathsf{poly}(\lambda)$. Moreover, $|p' - p| \leq \mathsf{negl}(\lambda)$ since otherwise, we can easily construct an efficient adversary that distinguishes between $\mathsf{RealPPRF}$ and $\mathsf{IdealPPRF}$ with non-negligible probability. Therefore, if $\mathcal{A}$ has non-negligible advantage in distinguishing Hyb and $\mathsf{IdealPPRFStrong}$, then $\mathcal{B}$ has non-negligible advantage in distinguishing $\mathsf{RealPPRF}$ and $\mathsf{IdealPPRF}$. □

□

C.2 Useful Facts about the Distribution $\mathcal{D}_n$

We define the following helpful notation where $x \in \{0,1\}^{\log n}$:

$$\mathsf{suffixes}(x) := \{(0^B || x)[i :]\}_{i \in [\frac{\log n}{2} + B]}$$

We first describe a couple useful facts which will later be used in our hybrid sequence.

Fact C.3. *Consider two arbitrary elements $x, y \in \{0, 1, \ldots, n-1\}$ which may be different or the same. There is a polynomial function $\mathsf{poly}(\cdot)$ such that $\Pr_{S \xleftarrow{\$} \mathcal{D}_n}[x, y \in S] \geq 1/\mathsf{poly}(n)$. Or equivalently, let $\mathsf{RO}(\cdot)$ denote a random oracle. Then, there is some polynomial function $\mathsf{poly}(\cdot)$, such that the following event happens with at least $1/\mathsf{poly}(n)$ probability: $\mathsf{RO}(\cdot)$ outputs 1 on every input from $\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y)$.*

Proof. The proof is straightforward. Since there are at most $2(\frac{\log n}{2} + B)$ points that we want to force to be 1, the probability that this happens is at least

$$\frac{1}{2^{2(\frac{\log n}{2} + B)}} \geq \frac{1}{n \log^5 n}$$

for sufficiently large n. □

Intuitively, the following fact states that given the distribution $\mathcal{D}_n$, conditioned on one element x or two elements $x, y \in \{0, 1, \ldots, n-1\}$ being in the set, the probability that up to $\sqrt{n}$ other elements are not in the set must be at least inverse polynomial.

Fact C.4. *Consider two arbitrary elements $x, y \in \{0, 1, \ldots, n-1\}$ which may be different or the same, and $Q' \leq \sqrt{n}$ other elements $x_1, \ldots, x_{Q'}$ such that $x_j \neq x$ and $x_j \neq y$ for any $j \in [Q']$. Then, there is some polynomial function $\mathsf{poly}(\cdot)$, such that*

$$\Pr_{S \xleftarrow{\$} \mathcal{D}_n}[\forall j \in [Q'] : x_j \notin S | x, y \in S] \geq 1/\mathsf{poly}(n)$$

Or equivalently, let $\mathsf{RO}(\cdot)$ denote a random oracle. Then, there is some polynomial function $\mathsf{poly}(\cdot)$, such that the following event happens with at least $1/\mathsf{poly}(n)$ probability over the choice of RO: for every $j \in [Q']$, $\mathsf{RO}(\cdot)$ does not always output 1 over the input set $\mathsf{suffixes}(x_j) \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$.

Proof. It suffices to prove the lemma for the worst case $Q' = Q = \sqrt{n}$. For convenience, define the following event Ev_j^- for $j \in [Q]$:

Ev_j^- : $\mathsf{RO}(\cdot)$ does not always output 1 over the input set $\mathsf{suffixes}(x_j)\backslash(\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$

Let $\mathsf{size}(x_j) = |\mathsf{suffixes}(x_j)\backslash(\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))|$, then $\Pr[\mathsf{Ev}_j^-] \geq 1 - 1/2^{\mathsf{size}(x_j)}$. Note also that for any Ev_j^-, any set $I \subseteq [n]$ such that $j \notin I$, it holds that

$$\Pr[\mathsf{Ev}_j^- | \{\mathsf{Ev}_{j'}^-\}_{j' \in I}] \geq \Pr[\mathsf{Ev}_j^-]$$

Therefore, we have that

$$\Pr[\mathsf{Ev}_1^-, \ldots, \mathsf{Ev}_Q^-] \geq \prod_{j \in [Q]} \Pr[\mathsf{Ev}_j^-]$$

Observe that as long as $x_j \neq x$ and $x_j \neq y$, $\mathsf{size}(x_j) > B$. Further, there are at most 4 choices of x_j such that $\mathsf{size}(x_j) = B+1$, at most 8 choices of x_j such that $\mathsf{size}(x_j) = B+2$, at most 16 choices of x_j such that $\mathsf{size}(x_j) = B+3$, and so on. Therefore, we have the following where $\ell = \frac{\log n}{2}$:

$$\begin{aligned}
&\Pr[\mathsf{Ev}_1^-, \ldots, \mathsf{Ev}_Q^-] \geq \prod_{j \in [Q]} \Pr[\mathsf{Ev}_j^-] \\
\geq &\left(1 - \frac{1}{2^{B+1}}\right)^4 \cdot \left(1 - \frac{1}{2^{B+2}}\right)^8 \cdot \left(1 - \frac{1}{2^{B+3}}\right)^{16} \cdot \ldots \cdot \left(1 - \frac{1}{2^{B+\ell-1}}\right)^{2^\ell} \\
\geq &\left(1 - \frac{2}{\log^2 n}\right)^\ell \geq 1 - \frac{1}{\log n}
\end{aligned}$$

□

C.3 Sequence of Hybrid Experiments

To prove Theorem 6.1, we define a sequence of hybrid experiments and show that the adversary $\mathcal{A}$'s views in every pair of adjacent hybrids are either identically distributed or computationally indistinguishable.

Experiment Real**.** Same as the real-world execution where an honest client interacts with $\mathcal{A}$ acting as the server. Henceforth, we may assume that during the online queries, the client skips the steps of FHE decryption and computing the answer to the query. However, it still deletes the last entry of the table T (which was added earlier during the online query); further, it still promotes the next available backup entry to a primary entry. Note that locally skipping the FHE decryption and computation of the answer does not affect the distribution of the messages the client sends to the server $\mathcal{A}$.

Experiment Hyb_1**.** Experiment Hyb_1 is almost the same as Real except that during the offline setup phase, the client replaces all FHE ciphertexts sent to the server with encryptions of 0.

Claim C.5. *Suppose that the FHE scheme satisfies semantic security. Then,* Hyb_1 *is computationally indistinguishable from* Real*.*

Proof. Hyb_1 is computationally indistinguishable from the real experiment following in a straightforward manner from the semantic security of FHE.

□

Experiment Hyb_2. Experiment Hyb_2 is the same as Hyb_1 except that we modify the client to record some extra information as it interacts with the server. Specifically, for each entry in the primary table, the client maintains a set of constraints. Initially, the constraint sets are all empty. During each online query for the index $x \in \{0, 1, \ldots, n-1\}$, the client performs the following — below the text in blue denotes the additional actions taken by the client:

- Sample $\mathsf{sk} \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$ and append the entry $(\mathsf{sk}, 0)$ to the table T of primary sets; record the additional constraint $\langle +x \rangle$ for this entry;
- Find the first entry (sk_i, p_i) such that $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_i, x) = 1$;
  - for every entry $j < i$ in T, if the entry (sk_j, p_j) was earlier promoted from a backup set during a query for y, record the additional constraint $\langle +y : -x \rangle$; else record the additional constraint $\langle -x \rangle$ for this entry.
  - For the i-th entry (sk_i, p_i), if the entry (sk_i, p_i) was earlier promoted from a backup set during a query for y, record the additional constraint $\langle +y : +x \rangle$; else record the additional constraint $\langle +x \rangle$ for this entry.
- The rest of the client's algorithm is the same as in Hyb_1, except that when the client promotes a backup entry to the primary table T, it records the fact that the entry was promoted during a query for the index x, and empties the constraint set associated with the relevant entry.

Intuitively, the constraint $\langle +x \rangle$ (or $\langle -x \rangle$, resp.) mean that x should (or should not, resp.) be contained in the set; the constraint $\langle +x \rangle$ means that x should be contained in the set. The constraint $\langle +y : +x \rangle$ (or $\langle +y : -x \rangle$) means that after calling **Add** to force-add the element y, the element x should (or should not, resp.) be in the set.

Hyb_2 is clearly identically distributed as Hyb_1 since recording the extra information does not affect the distribution.

Experiment Hyb_3. Experiment Hyb_3 is the same as Hyb_2 except that during the online phase, whenever the client is about to send the key of a pseudorandom set to the server, instead of sending the key computed by the honest algorithm, the client resamples a fresh key corresponding to the desired constraints, performs a corresponding **Add** operation on it if the key was promoted from a backup key, and then performs a **ReSamp** operation. More concretely, whenever the client is about to send a key for a pseudorandom set to the server during an online query, it instead sends the following "lazy-sampled" key:

- If the entry found in the table T was earlier promoted from a backup entry during a query for y, then, repeat: let $\mathsf{sk} \leftarrow \mathbf{Gen}(1^\lambda, n)$ and let $\mathsf{sk}_{+y} \leftarrow \mathbf{Add}(\mathsf{sk}, y)$, until the following constraints are satisfied:
  - for every constraint of the form $\langle +y : -x \rangle$ recorded for this entry, it must be that $x \notin \mathbf{Set}(\mathsf{sk}_{+y})$;
  - for every constraint of the form $\langle +y : +x \rangle$ recorded for this entry, it must be that $x \in \mathbf{Set}(\mathsf{sk}_{+y})$;

  Now, call $\mathsf{rsk} \leftarrow \mathbf{ReSamp}(\mathsf{sk}_{+y}, x)$ where x is the current query, and send rsk to the server.
- Else, sample $\mathsf{sk} \leftarrow \mathbf{Gen}(1^\lambda, n)$ subject to the following constraints:
  - for every constraint of the form $\langle -x \rangle$ recorded for this entry, it must be that $x \notin \mathbf{Set}(\mathsf{sk})$;

- for every constraint of the form $\langle +x \rangle$ recorded for this entry, it must be that $x \in \mathbf{Set}(\mathsf{sk})$;

Now, call $\mathsf{rsk} \leftarrow \mathbf{ReSamp}(\mathsf{sk}, x)$ where x is the current query, and send rsk to the server.

Claim C.6. Hyb_3 *is identically distributed as* Hyb_2*.*

Proof. In Hyb_3, we first sample an initial batch of pseudorandom set keys, and we use these keys to decide which primary entry is matched during each query. Let $\mathbf{I} := (i_1, i_2, \ldots, i_Q)$ be the random variables that denote the index of the entry matched during each of the Q queries, and let $\mathbf{x} := x_1, x_2, \ldots, x_Q$ be the Q queries. The experiment Hyb_3 is essentially maintaining the *a-posteriori* distribution of each primary entry conditioned on having observed the choices of $\mathbf{I}$ and $\mathbf{x}$. Only when the client needs to send the key to the server (possibly after performing an **Add** operation on it and always after performing a **ReSamp** operation), we perform the actual sampling of the key from the *a-posteriori* distribution. Further, if the key was promoted from a backup key, we perform a corresponding **Add** operation; and we always perform a **ReSamp** operation on the lazily sampled key before sending it to the server. Therefore, the server $\mathcal{A}$'s views in Hyb_3 and Hyb_2 are identically distributed — in fact, this holds even if we allow $\mathcal{A}$ to observe the choice of $\mathbf{I}$ (which $\mathcal{A}$ does not observe in the actual experiment). □

Claim C.7. *Suppose $Q = \sqrt{n}$. Except with negligible probability,* Hyb_3 *completes in polynomial-time. In other words, the lazy sampling of keys can be accomplished in polynomial time except with negligible probability.*

Proof. Observe that if the entry found in T was earlier promoted from a backup entry during a query for y, then there is exactly one positive constraint of the form $\langle +y : +x \rangle$, and at most Q negative constraints of the form $\langle +y : -x' \rangle$. Similarly, if the entry found in T was not promoted from a backup entry, then there is exactly one positive constraint of the form $\langle +x \rangle$, and at most Q negative constraints of the form $\langle -x' \rangle$.

The remainder of the proof follows in a straightforward fashion from the pseudorandomness of the underlying programmable PRF and due to Fact C.3 and Fact C.4. □

Experiment Hyb_4**.** Hyb_4 is otherwise the same as Hyb_3 except with the following modification. Recall that in Hyb_3, upon receiving a new query x, we scan the primary table T, and for each key $\mathsf{sk} \in T$, we check if $\mathbf{Member}(\mathsf{sk}, x) = 1$. If a key sk was promoted from the backup table during an earlier query y, the check is performed by calling $\mathsf{PRF}.\mathbf{Eval}(\mathsf{sk}, \cdot)$ but forcing the outcomes on $\mathsf{suffixes}(y)$ to be 1. In Hyb_4, however, we make the following change:

- Whenever a backup key is promoted to become a primary key during a query for y, we replace the key with a resampled one, i.e., sample $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to $y \in \mathbf{Set}(\mathsf{msk})$. In other words, we are sampling a programmable PRF key subject to the constraint that its outcomes on any input from the set $\mathsf{suffixes}(y)$ must be 1.

Claim C.8. *Suppose that the programmable PRF satisfies private programmability, simulation security, and correctness. Then,* Hyb_3 *is computationally indistinguishable from* Hyb_4*.*

Proof. We consider the following hybrid sequence. Experiment H is otherwise the same as Hyb_3, except that when we promote a backup key to the primary table, we replace the consumed key with a key sk sampled from the following distribution: sample $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$, $\mathsf{sk} \leftarrow \mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}, \{(z, 1)\}_{z \in \mathsf{suffixes}(x)})$, where x is the current query. Due to the correctness of the PRF, Hyb_3 is statistically indistinguishable from H.

Experiment H' is otherwise the same as H, except that when we promote a backup key to the primary table, we replace the consumed key with a key sk sampled from the following distribution: sample $\mathsf{sk} \leftarrow \mathsf{PRF}.\mathsf{Sim}(1^\lambda, L)$, subject to $x \in \mathbf{Set}(\mathsf{sk})$, where Sim is the simulator in the private programmability definition. H' is computationally indistinguishable from H due to Lemma 6.2.

Experiment H'' is otherwise the same as H', except that when we promote a backup key to the primary table, we replace the consumed key with a key sk sampled from the following distribution: sample $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$, $\mathsf{sk} \leftarrow \mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}, \emptyset)$ subject to $x \in \mathbf{Set}(\mathsf{sk})$, where x is the current query. H'' is computationally indistinguishable from H' due to the private programmability of the PRF.

Finally, H'' is statistically indistinguishable from Hyb_4, due to the correctness of the programmable PRF. □

Experiment Hyb_5. Hyb_5 is otherwise the same as Hyb_4 except with the following modification when performing lazy-sampling of the pseudorandom set key. Let x be the current query, and let Sim be the simulator in the private programmability definition.

- If the entry found in T was earlier promoted from the backup table during a query for y, then repeat: $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$ until sk satisfies the following constraints, and send sk to the server:
  - $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ outputs 1 on any input from the set $\mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$;
  - for every negative constraint of the form $\langle +y : -x' \rangle$ that is recorded for this entry, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ does not output all 1s on the input set $\mathsf{suffixes}(x') \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$.
- Else, repeat: $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$ until sk satisfies the following constraints, and send sk to the server:
  - for every negative constraint of the form $\langle -x' \rangle$ that is recorded for this entry, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ does not output all 1s on the input set $\mathsf{suffixes}(x') \backslash \mathsf{suffixes}(x)$.

Notably, in Hyb_5, we no longer perform **Add** or **ReSamp** operations on the key sent to the server $\mathcal{A}$.

Claim C.9. *Suppose that the underlying programmable PRF satisfies private programmability and simulation security. Then, Hyb_5 is computationally indistinguishable from Hyb_4.*

Proof. Hyb_4 can be equivalently viewed as the following: during a query for index x, when we compute the lazy-sampled key, do the following depending on which case:

- *Case 1: the lazy-sampled key was promoted from the backup table during an earlier query for index y.* In this case, there is exactly one positive constraint of the form $\langle +y : +x \rangle$, and there can be at most Q negative constraints of the form $\langle +y : -x' \rangle$.

  Therefore, the lazy-sampled key has the following distribution: sample a random $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to the following constraints:

  - $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ evaluates to 1 on $\mathsf{suffixes}(x) \backslash \mathsf{suffixes}(y)$; and
  - For each negative constraint of the form $\langle +y : -x' \rangle$, $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ does not evaluate to all 1s on the input set $\mathsf{suffixes}(x') \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$.

  Finally, we call **Prog** to program the resulting msk to force the outcomes at $\mathsf{suffixes}(x)$ to be random, and the outcomes at $\mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$ to be 1, and send the programmed key to the server $\mathcal{A}$.

- *Case 2: the lazy-sampled key was not promoted from the backup table.* In this case, there is exactly one positive constraint of the form $\langle +x \rangle$, and there can be at most Q negative constraints of the form $\langle -x' \rangle$. Therefore, the lazy-sampled key is sampled at random $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$, subject to the following constraints:
  - $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ evaluates to 1 on $\mathsf{suffixes}(x)$; and
  - For each negative constraint of the form $\langle +y : -x' \rangle$, $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ does not evaluate to all 1s on the input set $\mathsf{suffixes}(x') \backslash \mathsf{suffixes}(x)$.

  Finally, we call **Prog** to program the resulting msk to force the outcomes at $\mathsf{suffixes}(x)$ to be random, and send the programmed key to the server $\mathcal{A}$.

To show that Hyb_4 is computationally indistinguishable from Hyb_5, we can consider a sequence of hybrid experiments denoted $\mathsf{H}_0, \ldots, \mathsf{H}_Q$. In H_i where $i \in \{0, 1, \ldots, Q\}$, for the first i queries, we use the method of Hyb_5 to sample the key sent to the server, and for the remaining queries, we use the method of Hyb_4 to sample the key sent to the server. It suffices to show that every pair of adjacent hybrids are computationally indistinguishable.

We show that if there is an efficient adversary $\mathcal{A}$ that can distinguish H_i and H_{i+1} where $i \in \{0, 1, \ldots, Q-1\}$, we can construct an efficient reduction $\mathcal{B}$ which can break the strong privacy of the underlying PRF (see Lemma 6.2) which is implied by private programmability and simulation security. Basically, $\mathcal{B}$ acts as the client and interacts with $\mathcal{A}$ like in H_i, except that for the i-th query, when it is about to send the key to $\mathcal{A}$, it performs the following instead:

- *Case 1: the i-th query wants to lazy-sample a key that was promoted from the backup table during an earlier query for the index y.* $\mathcal{B}$ sends its challenger $X = \mathsf{suffixes}(x)$, $X' = \mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$, $\{v_x = 1\}_{x \in X'}$, and a predicate $\mathsf{pred}^{X \cup X'}$ that wants the original unprogrammed PRF to output 1 at $\mathsf{suffixes}(x) \backslash \mathsf{suffixes}(y)$. $\mathcal{B}$ obtains from its challenger some PRF key sk. If for every negative constraint of the form $\langle +y : -x' \rangle$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ does not evaluate to all 1s on $\mathsf{suffixes}(x') \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$, then $\mathcal{B}$ forwards sk to $\mathcal{A}$, and outputs the same guess as $\mathcal{A}$. Otherwise, $\mathcal{B}$ aborts outputting a random guess. If $\mathcal{B}$ is in the game $\mathsf{RealPPRFStrong}$, and $\mathcal{B}$ does not abort outputting a random guess, then $\mathcal{A}$'s view is identically distributed as in H_i. If $\mathcal{B}$ is in the game $\mathsf{IdealPPRFStrong}$, and $\mathcal{B}$ does not abort outputting a random guess, then $\mathcal{A}$'s view is identically distributed as in H_{i+1}.

  Let p be the probability that $\mathcal{B}$ does not outputting a random guess when it is playing $\mathsf{RealPPRFStrong}$, and let p' be the corresponding probability when it is playing $\mathsf{IdealPPRFStrong}$. Due to Fact C.4 and the pseudorandomness of the PRF, $p \geq 1/\mathsf{poly}(\lambda, n)$. Further, it must be that $|p' - p| \leq \mathsf{negl}(\lambda)$ since otherwise we can easily construct an adversary that can distinguish $\mathsf{RealPPRFStrong}$ and $\mathsf{IdealPPRFStrong}$ with non-negligible probability. Therefore, if $\mathcal{A}$ has non-negligible advantage in distinguishing H_i and H_{i+1}, then $\mathcal{B}$ has non-negligible advantage in distinguishing $\mathsf{RealPPRFStrong}$ and $\mathsf{IdealPPRFStrong}$.
- *Case 2*: The proof of Case 2 is similar to Case 1 except that now, we replace $\mathsf{suffixes}(y)$ with $\emptyset$.

□

Experiment Hyb_6. Hyb_6 is otherwise the same as Hyb_5, except the following modification when performing lazy-sampling of the pseudorandom set key. Let x be the current query, and let Sim be the simulator in the private programmability definition.

- If the entry found in T was earlier promoted from the backup table during a query for y, then repeat: $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$ until msk satisfies the following constraints:

- $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ outputs 1 on any input from the set $\mathsf{suffixes}(y) \cup \mathsf{suffixes}(x)$;
- for every negative constraint of the form $\langle +y : -x' \rangle$ that is recorded for this entry, $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ does not output all 1s on the input set $\mathsf{suffixes}(x') \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$.

- Else, repeat: $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$ until sk satisfies the following constraints, and send sk to the server:
  - for every negative constraint of the form $\langle -x' \rangle$ that is recorded for this entry, $\mathsf{PRF}.\mathbf{Eval}(\mathsf{msk}, \cdot)$ does not output all 1s on the input set $\mathsf{suffixes}(x') \backslash \mathsf{suffixes}(x)$, and moreover it outputs 1 on $\mathsf{suffixes}(x)$.

Now, call $\mathsf{sk} \leftarrow \mathbf{Prog}(\mathsf{msk}, \{(z, r_z)\}_{z \in \mathsf{suffixes}(x)})$ where all r_z's are sampled independently at random, and send sk to the server.

Claim C.10. *Suppose that the PRF satisfies private programmability and simulation security. Then,* Hyb_6 *is computationally indistinguishable from* Hyb_5*.*

Proof. The proof is very similar to that of Claim C.9. To show that Hyb_5 is computationally indistinguishable from Hyb_6, we can consider a sequence of hybrid experiments denoted $\mathsf{H}_0, \ldots, \mathsf{H}_Q$. In H_i where $i \in \{0, 1, \ldots, Q\}$, for the first i queries, we use the method of Hyb_5 to sample the key sent to the server, and for maining queries, we use the method of Hyb_6 to sample the key sent to the server. It suffices to show that every pair of adjacent hybrids are computationally indistinguishable.

We show that if there is an efficient adversary $\mathcal{A}$ that can distinguish H_i and H_{i+1} where $i \in \{0, 1, \ldots, Q-1\}$, we can construct an efficient reduction $\mathcal{B}$ which can break the strong privacy of the underlying PRF (see Lemma 6.2) which is implied by private programmability and simulation security. Basically, $\mathcal{B}$ acts as the client and interacts with $\mathcal{A}$ like in H_i, except that for the i-th query, when it is about to send the key to $\mathcal{A}$, it performs the following instead:

- *Case 1: the i-th query wants to lazy-sample a key that was promoted from the backup table during an earlier query for the index y.* $\mathcal{B}$ sends its challenger $X = \mathsf{suffixes}(x)$, $X' = \emptyset$, and a predicate $\mathsf{pred}^{X \cup X'}$ that wants the original unprogrammed PRF to output 1 at $\mathsf{suffixes}(x)$. $\mathcal{B}$ obtains from its challenger some PRF key sk. If for every negative constraint of the form $\langle +y : -x' \rangle$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ does not evaluate to all 1s on $\mathsf{suffixes}(x') \backslash (\mathsf{suffixes}(x) \cup \mathsf{suffixes}(y))$, and moreover, it evaluates to all 1s on the input set $\mathsf{suffixes}(y) \backslash \mathsf{suffixes}(x)$, then $\mathcal{B}$ forwards sk to $\mathcal{A}$, and outputs the same guess as $\mathcal{A}$. Otherwise, $\mathcal{B}$ aborts outputting a random guess. If $\mathcal{B}$ is in the game $\mathsf{RealPPRFStrong}$, and $\mathcal{B}$ does not abort outputting a random guess, then $\mathcal{A}$'s view is identically distributed as in H_i. If $\mathcal{B}$ is in the game $\mathsf{IdealPPRFStrong}$, and $\mathcal{B}$ does not abort outputting a random guess, then $\mathcal{A}$'s view is identically distributed as in H_{i+1}.

  Let p be the probability that $\mathcal{B}$ does not outputting a random guess when it is playing $\mathsf{RealPPRFStrong}$, and let p' be the corresponding probability when it is playing $\mathsf{IdealPPRFStrong}$. Due to Fact C.4 and the pseudorandomness of the PRF, $p \geq 1/\mathsf{poly}(\lambda, n)$. Further, it must be that $|p' - p| \leq \mathsf{negl}(\lambda)$ since otherwise we can easily construct an adversary that can distinguish $\mathsf{RealPPRFStrong}$ and $\mathsf{IdealPPRFStrong}$ with non-negligible probability. Therefore, if $\mathcal{A}$ has non-negligible advantage in distinguishing H_i and H_{i+1}, then $\mathcal{B}$ has non-negligible advantage in distinguishing $\mathsf{RealPPRFStrong}$ and $\mathsf{IdealPPRFStrong}$.

- *Case 2*: The proof of Case 2 is similar to Case 1 except that now, we replace $\mathsf{suffixes}(y)$ with $\emptyset$.

□

Fact C.11. Hyb_6 *is identically distributed as the following process. During the offline phase, send FHE encryptions of* 0 *to the server. During each query, the client appends a key sampled at random from* $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ *subject to containing* x *at the end of the table* T*. The client finds in* T *the first entry* msk *whose set contains* x*. The client sends* $\mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}, \{(z, r_z)\}_{z \in \mathsf{suffixes}(x)})$ *to the server where* r_z*'s are independent random bits. The client then replaces the consumed entry with a fresh key sampled from* $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ *subject to containing* x*, and deletes the last entry of the table.*

Proof. By definition, the above process has the same local table distribution as in Hyb_6. Recall that in Hyb_6, the client samples another key subject to the corresponding constraints, programs the key and sends it to the server. In the above process, the client directly programs the key stored in the local table and sends it to the server. In Hyb_6, conditioned on the matched indices $\mathbf{I}$, the matched entry in the table has the same a-posteriori distribution as the key we lazily sample which we then program and send to the server. Therefore, Hyb_6 can be equivalently rewritten as the randomized process in Fact C.11. □

Experiment Ideal. In the Ideal experiment, during the offline phase, the client sends FHE encryptions of 0 to the server. During each online query, whenever the client needs to send the server $\mathcal{A}$ some key, it simply samples a key $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$ at random where Sim is the same simulator as in the private programmability definition, and sends sk to the server.

Claim C.12. *Suppose that the PRF satisfies private programmability. Then* Hyb_6 *is computationally indistinguishable from* Ideal*.*

Proof. We can consider a sequence of hybrids denoted $\mathsf{H}_0, \mathsf{H}_1, \ldots, \mathsf{H}_Q$. In H_i, for the first i queries we do the following:

- The client sends the server a random simulated key sampled from $\mathsf{sk} \leftarrow \mathsf{Sim}(1^\lambda, L)$.

For the remaining queries, the client does the following just like in Hyb_6 (see Fact C.11):

- Let x be the current query. The client appends a key sampled at random from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to containing x at the end of the table T. The client finds the first entry msk whose set contains x. The client sends $\mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}, \{(z, r_z)\}_{z \in \mathsf{suffixes}(x)})$ to the server where r_z's are independent random bits. The client then replaces the consumed entry with a fresh key sampled from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to containing x, and deletes the last entry of the table.

It suffices to prove that H_i and H_{i+1} are computationally indistinguishable for every $i \in \{0, 1, \ldots, Q-1\}$. First, we prove the following fact.

Fact C.13. *In* H_i*, conditioned on the server's view at the beginning of the* $(i+1)$*-th query, the keys contained in the client's primary table* T *are identically distributed as sampling independent keys from* $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$*.*

Proof. From an information theoretic perspective, the server learns no information during the first i queries. Therefore, we can prove the claim by induction. The statement is true initially before any query is made. Now, suppose the statement is true at the end of the $(i'-1)$-th query where $i' \leq i$, we prove that the statement is still true at the end of the i'-th query. It is easy to see this if we view the distribution of lenT randomly sampled simulated keys as the following distribution:

- First, sample the index $j^* \in [\mathsf{lenT}+1]$ which is the first entry that contains the queried element x. Note that since we appended to the table T a key sampled from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to containing the current query x before searching through T, a satisfying key is guaranteed to be found.

- For any $j < j^*$, sample the j-th key at random from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to not containing x; sample the j^*-th key at random from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to containing x; and finally, for $j > j^*$, sample j-th key at random from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$.

Now, during the i'-th query, we consume the j^*-th entry, and replace it with a key freshly sampled from $\mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to containing x. Therefore, this does not change the distribution.

□

Therefore, in H_i, during the $(i+1)$-th query for some index x, the matched key has the following distribution: sample $\mathsf{msk} \leftarrow \mathbf{Gen}(1^\lambda, L)$ subject to containing x. Thus, the key returned to the server is identically distributed as: sample $\mathsf{msk} \leftarrow \mathsf{PRF}.\mathbf{Gen}(1^\lambda, L)$ subject to $x \in \mathbf{Set}(\mathsf{msk})$ where x is the current query, call $\mathsf{sk} \leftarrow \mathsf{PRF}.\mathbf{Prog}(\mathsf{msk}, \{(z, r_z)\}_{z \in \mathsf{suffixes}(x)})$ where the r_z's are independently sampled random bits. Due to the private programmability of the PRF, we can replace the key sent to the server during the $(i+1)$-th query with the outcome of $\mathsf{Sim}(1^\lambda, L)$, which gives us H_{i+1}, and the adversary $\mathcal{A}$ will not be able to distinguish the two except with negligible probability. □

D Correctness Proof

We now prove the correctness of our PIR scheme.

Offline phase. Due to Fact 4.3 and the pseudorandomness of the PRF, for each pseudorandom set key sk, the probability that $\mathsf{CSetEnum}$ returns $\mathsf{bSucc} = \mathsf{True}$ is at least $1 - 1/\log n$. Therefore, the probability that among $k = \omega(\log \lambda)$ copies, no copy returns $\mathsf{bSucc} = \mathsf{True}$ is negligibly small. Therefore, the probability that the offline phase cannot find $\mathsf{lenT} + Q$ copies with $\mathsf{bSucc} = \mathsf{True}$ is negligibly small. Due to the correctness of the FHE, during the offline phase, the client obtains the correct parities for all $\mathsf{lenT} + Q$ pseudorandom sets except with negligible probability.

Online phase. Given the above, to prove the correctness of our PIR scheme, it suffices to show the following: assume that to start with, the client is storing the correct parity bits for all of the pseudorandom sets. Then, each single copy of the PIR scheme is correct with probability at least 2/3. If so, due to the standard Chernoff bound, when we do majority voting among $k = \omega(\log \lambda)$ copies, the majority vote is correct with all but $\mathsf{negl}(\lambda)$ probability.

Experiment CReal. Same as the real-world experiment running a single copy of the PIR scheme, except that at the end of each query, we force the client's parity bits to be all correct (even if the client may have computed an incorrect parity bit).

Experiment CIdeal. Consider the following experiment where the client stores each set using a random oracle RO_j rather than a pseudorandom key.

Ideal correctness experiment CIdeal

Offline phase. Client generates $\mathsf{lenT} + Q$ random oracles denoted $\{\mathsf{RO}_j\}_{j \in [\mathsf{lenT}+Q]}$, where each RO_j defines a random set. Let $\mathsf{Label}_j = \perp$ for $j \in [\mathsf{lenT}]$. The client obtains the correct parity bit p_j for each random set. The $\mathsf{lenT} + Q$ random oracles are divided into lenT primary entries which form the primary table T, as well as Q backup entries.

Online query for $x \in \{0, 1, \ldots, n-1\}$.

- Client overwrites the $(\mathsf{lenT} + 1)$-th entry of the primary table T to be the following random oracle: sample a fresh RO^* and force RO^*'s outcomes at $\mathsf{suffixes}(x)$ to be 1.
- Client finds the first primary entry j such that the set defined by RO_j contains x. If the entry found is the last entry of T, return $\mathsf{ErrNotFound}$.
- If $y := \mathsf{Label}_j \neq \perp$ and the set generated by RO_j contains other elements related to y, return $\mathsf{ErrParity}$.
- Client resamples RO_j's outcomes at the points $\mathsf{suffixes}(x)$. If this resampling ends up removing some element $x' \neq x$ from the set, or it does not remove x itself from the set, return $\mathsf{ErrReSampFail}$. If the resampled RO_j takes more than $6\sqrt{n}\log^5 n$ RO calls to enumerate the set, then return $\mathsf{ErrTimeOut}$.
- Client takes the first unconsumed backup entry denoted RO^*, forces RO^*'s outputs at $\mathsf{suffixes}(x)$ to be 1, and then uses the resulting random oracle to replace the j-th entry of T. Further, set $\mathsf{Label}_j = x$. Return $\mathsf{Success}$.

Intuitively, $\mathsf{ErrNotFound}$ characterizes the probability that the queried element is not in any of the primary sets; $\mathsf{ErrReSampFail}$ represents the probability that resampling at point x either fails to remove x from the set, or it removes some other element related to x from the set; $\mathsf{ErrParity}$ represents the probability that when we promoted a backup entry to the primary table by forcefully adding some element y, it caused some element(s) related to y to be added to the set — in this case, the parity associated with this entry could be incorrect. Finally, $\mathsf{ErrTimeOut}$ represents the probability of an error caused by the set enumeration timing out.

Let $\mathsf{Wrong}^{i,\mathsf{CReal}}(x_1, \ldots, x_Q)$ denote the event that upon the query sequence $x_1, \ldots, x_Q$, the client computes the incorrect answer during the i-th query in experiment CReal. Let $\mathsf{Wrong}^{i,\mathsf{CIdeal}}(x_1, \ldots, x_Q)$ denote the event that upon the query sequence $x_1, \ldots, x_Q$, the client returns either $\mathsf{ErrNotFound}$ or $\mathsf{ErrReSampFail}$ or $\mathsf{ErrTimeOut}$ during the i-th query in experiment CIdeal.

Claim D.1. *Assume that the programmable PRF satisfies pseudorandomness and correctness. Then, there exists a negligible function* $\mathsf{negl}(\cdot)$ *such that for any* $x_1, \ldots, x_Q \in \{0, 1, \ldots, n-1\}$, *for any* $i \in [Q]$,

$$\Pr[\mathsf{Wrong}^{i,\mathsf{CReal}}(x_1, \ldots, x_Q)] \leq \Pr[\mathsf{Wrong}^{i,\mathsf{CIdeal}}(x_1, \ldots, x_Q)] + \mathsf{negl}(\lambda)$$

Proof. Experiment CIdeal makes the following modifications to CReal: 1) replaces PRF evaluations to RO calls, and 2) remove all instructions not related to correctness. Note that the event $\mathsf{Wrong}^{i,\mathsf{CReal}}$ depends only on the evaluation outcomes of the PRF and does not depend on the PRF key itself. Therefore, the claim follows in a straightforward fashion from the pseudorandomness and the correctness of the underlying programmable PRF. □

Claim D.2. *For any* $x_1, \ldots, x_Q \in \{0, 1, \ldots, n-1\}$, *for any* $i \in [Q]$, $\Pr[\mathsf{Wrong}^{i,\mathsf{CIdeal}}(x_1, \ldots, x_Q)] \leq 1/3$.

Proof. To prove the claim, we will make use of the following fact:

Fact D.3. *In* CIdeal*, the* RO_j *found during each query has the following distribution: sample an* RO *at random subject to containing* x*. Moreover, at the end of each query, the table* T *(ignoring the* $(\mathsf{lenT}+1)$*-th entry) has the same distribution as* $\mathsf{lenT}+Q$ *independently sampled* RO*s.*

Proof. We can prove the fact using a similar argument as Fact C.13, except that now, each entry of the table T is a random oracle instead of a key sampled from $\mathsf{Sim}(1^\lambda, L)$. □

We can now bound the probability of each type of error.

ErrNotFound. Due to Fact D.3 and Fact 3.8, $\Pr[\mathsf{ErrNotFound}] \le 1/n$.

ErrReSampFail. Due to Fact D.3 and Fact 3.7, the probability that there exists another element related to x in the chosen RO_j is upper bounded by $1/\log n$. The probability that resampling fails to remove x from the set is $1/(\sqrt{n}\,\mathsf{poly}\log n)$. Thus, $\Pr[\mathsf{ErrReSampFail}] \le 2/\log n$.

ErrTimeOut. Due to Fact D.3 and Fact 3.9, $\Pr[\mathsf{ErrTimeOut}] \le 1/\log n$.

ErrParity. This is the most complicated error to bound. To bound the probability of $\mathsf{ErrParity}$, we may equivalently consider the following experiment which is obtained from CIdeal, but removing all other errors we do not care about right now.

Experiment $\mathsf{CIdealParity}$

// same experiment as the one in Lemma 7.7 in Shi et al. [SACM21]

Offline setup. For $j = 1$ to lenT: sample a random oracle RO and let $T_j := \mathsf{RO}$. Set $\mathsf{Label}_j := \bot$.

Online query for index $x \in \{0, 1, \dots, n-1\}$**.**

a) Sample a new RO^* such that the associated set contains x. Append RO^* to the table T as the last entry, and mark its label $\mathsf{label}(T_{\mathsf{lenT}+1}) := \bot$.

b) Let $T_j := \mathsf{RO}_j$ be the smallest entry in the table T such that the set generated by RO_j contains x.

c) If $y := \mathsf{Label}_j \ne \bot$ and the set generated by RO_j contains other elements related to y, then return $\mathsf{ErrParity}$.

d) Sample a new RO' such that the generated set contains x. Overwrite $T_j := \mathsf{RO}'$ and set $\mathsf{Label}_j := x$.

e) Remove the last entry from T and return $\mathsf{Success}$.

We can show that in the above experiment, the probability that the i-th query returns $\mathsf{ErrParity}$ is upper bounded by $2/\log n$ using exactly the same approach as Shi et al. [SACM21]. Specifically, in the above experiment, for the i-th query to return $\mathsf{ErrParity}$, there are two cases:

1. The i-th query for index x_i finds an entry with the $y = \mathsf{Label}_j \ne \bot$, and x_i is related to y.
2. The i-th query for index x_i finds an entry with the $y = \mathsf{Label}_j \ne \bot$, and x_i is not related to y.

In the proof of Lemma 7.7 of Shi et al. [SACM21], they argue that due to Fact D.3 and Fact 3.7, the probability of the first case happening is upper bounded by the probability that a random RO subject to containing x_i also contains another element related to x_i, which is upper bounded by $1/\log n$. Through a more complicated argument, they also show that the probability of the second case happening is also upper bounded by $1/\log n$.

Therefore, we have that $\Pr[\mathsf{ErrParity}] \leq 2/\log n$. □

E Additional Preliminaries

E.1 Proof of Claim 3.5

Proof of Claim 3.5. We prove the properties one by one.

Correctness. If x is one of the programmed inputs, then correctness of **PEval** over x follows directly from the correctness of **PEval** of the underlying PRF'. Now consider the case when x is not one of the programmed inputs. Observe that **Prog** algorithm may program the underlying PRF' at $L' - L$ strings of length at most $\ell + 1$ ending with 1. However, calling $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, x)$ results in calling $\mathsf{PRF}'.\mathbf{PEval}(\mathsf{sk}, x||0)$, and $x||0$ cannot be a programmed input for the underlying PRF'. Therefore, correctness of **PEval** over a non-programmed input x follows from the correctness of the underlying PRF'.

Simulation security. $\mathsf{PRF}.\mathsf{Sim}(1^\lambda, P, L)$ is constructed as follows:

- Parse $P = \{(x_i, v_i\}_{i\in[L']}$. Choose $L - L'$ strings of length at most $\ell + 1$ ending with 1, denoted $\{x'_j\}_{j\in[L-L']}$. Choose $v'_j \xleftarrow{\$} \mathcal{V}$ for $j \in [L - L']$.
- Output $\mathsf{PRF}'.\mathsf{Sim}(1^\lambda, \{x_i||0, v_i\}_{i\in[L']} \cup \{x'_j, v'_j\}_{j\in[L-L']}, L)$.

Suppose there is an efficient adversary $\mathcal{A}$ that can break the simulation security of PRF with non-negligible probability, we can construct an efficient reduction $\mathcal{B}$ that breaks the simulation security of the underlying PRF' with non-negligible probability. When $\mathcal{A}$ submits $\{x_i, v_i\}_{i\in[L']}$ where $L' \leq L$, $\mathcal{B}$ chooses $L - L'$ strings of length at most $\ell + 1$ ending with 1, denoted $\{x'_j\}_{j\in[L-L']}$. $\mathcal{B}$ also chooses $v'_j \xleftarrow{\$} \mathcal{V}$ for $j \in [L - L']$. $\mathcal{B}$ submits to its own challenger $\{x_i||0, v_i\}_{i\in[L']} \cup \{x'_j, v'_j\}_{j\in[L-L']}$, and obtains sk from its challenger. It forwards sk to $\mathcal{A}$. Now, whenever $\mathcal{A}$ queries the point $x \in \{0,1\}^{\leq \ell}$, $\mathcal{B}$ forwards $x||0$ to its own challenger, obtains v, and forwards it to $\mathcal{A}$. Finally, $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs.

If $\mathcal{B}$ is playing the game $\mathsf{RealPPRF}$ for the underlying PRF', then $\mathcal{A}$'s view is identically distributed as in $\mathsf{RealPPRF}$ for PRF. If $\mathcal{B}$ is playing the game $\mathsf{IdealPPRF}$ for the underlying PRF', then $\mathcal{A}$'s view is identically distributed as in $\mathsf{IdealPPRF}$ for PRF.

Private programmability. Let $\mathsf{PRF}.\mathsf{Sim}(1^\lambda, L) = \mathsf{PRF}'.\mathsf{Sim}(1^\lambda, L)$. Suppose there is an efficient adversary $\mathcal{A}$ that can break the private programmability of PRF with non-negligible probability, we can construct an efficient reduction $\mathcal{B}$ that breaks the private programmability of the underlying PRF' with non-negligible probability. When $\mathcal{A}$ submits $\{x_i\}_{i\in[L']}$ where $L' \leq L$, $\mathcal{B}$ chooses $L - L'$ strings of length at most $\ell + 1$ ending with 1, denoted $\{x'_j\}_{j\in[L-L']}$. $\mathcal{B}$ submits to its own challenger $\{x_i||0\}_{i\in[L']} \cup \{x'_j\}_{j\in[L-L']}$, and obtains sk from its challenger. It then outputs whatever $\mathcal{A}$ outputs.

If $\mathcal{B}$ is playing $\mathsf{RealPPRFPriv}$ of the underlying PRF', then $\mathcal{A}$'s view is identically distributed as $\mathsf{RealPPRFPriv}$ of PRF. If $\mathcal{B}$ is playing $\mathsf{IdealPPRFPriv}$ of the underlying PRF', then $\mathcal{A}$'s view is identically distributed as $\mathsf{IdealPPRFPriv}$ of PRF. □

E.2 Fully Homomorphic Encryption

A fully homomorphic encryption scheme (FHE) with respect to a class of circuits $\mathcal{C}$ is a tuple $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Eval}, \mathbf{Dec})$ of efficient, possibly randomized algorithms, with the following syntax:

- $\mathbf{Gen}(1^\lambda)$: receives the security parameter λ and outputs a secret key fsk.
- $\mathbf{Enc}(\mathsf{fsk}, m)$: receives a secret key fsk and message m, and outputs a ciphertext c.
- $\mathbf{Eval}(\mathsf{Circ}, c_1, \ldots, c_d)$: receives a circuit $\mathsf{Circ} \in \mathcal{C}$ with d inputs, as well as d ciphertexts and outputs a ciphertext c.
- $\mathbf{Dec}(\mathsf{fsk}, c)$: receives a secret key fsk and ciphertext c, and outputs a plaintext m.

Correctness. Let $\mathcal{C}$ be a class of circuits, Circ be an arbitrary element in $\mathcal{C}$, and d be the input size of Circ. A FHE scheme $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Eval}, \mathbf{Dec})$ is correct with respect to $\mathcal{C}$, if $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is a correct encryption scheme, and there is a negligible function $\mathsf{negl}(\cdot)$ such that for every security parameter λ, for all messages $m_1, \ldots, m_d$, for any $\mathsf{Circ} \in \mathcal{C}$, the following holds with at least $1 - \mathsf{negl}(\lambda)$ probability: $\mathsf{fsk} \leftarrow \mathbf{Gen}(1^\lambda)$, for $i \in [d]$, $c_i \leftarrow \mathbf{Enc}(\mathsf{fsk}, m_i)$, $c' \leftarrow \mathbf{Eval}(\mathsf{Circ}, c_1, \ldots, c_d)$, then, it must be that $\mathbf{Dec}(\mathsf{fsk}, c') = \mathsf{Circ}(m_1, \ldots, m_d)$.

Semantic security. We say that an FHE scheme $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Eval}, \mathbf{Dec})$ is semantically secure iff $(\mathbf{Gen}, \mathbf{Enc}, \mathbf{Dec})$ is semantically secure.