# Near-Optimal Private Information Retrieval with Preprocessing

ARTHUR LAZZARETTI
*Yale University*

CHARALAMPOS PAPAMANTHOU
*Yale University*

**Abstract**

In Private Information Retrieval (PIR), a client wishes to access an index $i$ from a public $n$-bit database without revealing any information about $i$. Recently, a series of works starting with the seminal paper of Corrigan-Gibbs and Kogan (EUROCRYPT 2020) considered PIR with *client preprocessing* and *no additional server storage*. In this setting, we now have protocols that achieve $\tilde{O}(\sqrt{n})$ (amortized) server time and $\tilde{O}(1)$ (amortized) bandwidth in the two-server model (Shi et al., CRYPTO 2021) as well as $\tilde{O}(\sqrt{n})$ server time and $\tilde{O}(\sqrt{n})$ bandwidth in the single-server model (Corrigan-Gibbs et al., EUROCRYPT 2022). Given existing lower bounds, a single-server PIR scheme with $\tilde{O}(\sqrt{n})$ (amortized) server time and $\tilde{O}(1)$ (amortized) bandwidth is still feasible, however, to date, no known protocol achieves such complexities. In this paper we fill this gap by constructing the first single-server PIR scheme with $\tilde{O}(\sqrt{n})$ (amortized) server time and $\tilde{O}(1)$ (amortized) bandwidth. Our scheme achieves near-optimal (optimal up to polylogarithmic factors) asymptotics in every relevant dimension. Central to our approach is a new cryptographic primitive that we call an *adaptable pseudorandom set*: With an adaptable pseudorandom set, one can represent a large pseudorandom set with a succinct fixed-size key $k$, and can both add to and remove from the set a constant number of elements, by manipulating the key $k$, while maintaining its concise description as well as its pseudorandomness (under a certain security definition).

## 1 Introduction

In private information retrieval (PIR), a server holds a public database $\mathsf{DB}$ represented as an $n$-bit string and a client wishes to retrieve $\mathsf{DB}[i]$ without revealing $i$ to the server. PIR has many applications in various systems with advanced privacy requirements [SCH+21, AS16, BKMP12, KCG21, GCM+16] and comprises a foundational computer science and cryptography problem, with connections to primitives such as oblivious transfer [DCMO00] and locally-decodable codes [Yek10, KKYM20], among others. PIR can be naively realized by downloading the whole $\mathsf{DB}$ for each query, which is prohibitive for large databases. To achieve more efficient protocols, the two-server assumption [CKGS98] was introduced, where $\mathsf{DB}$ is replicated in two, non-colluding servers. *For the rest of the paper we use* **1PIR** *to refer to single-server PIR and* **2PIR** *to refer to two-server PIR.* Clearly, 1PIR is much more challenging than 2PIR, but also more useful; it is hard to ensure two servers do not collude and remain available in practice [MCR21, BBG+20]. Also, many connections between PIR and other primitives are shown only for 1PIR [DCMO00, Yek10].

**Sublinear time 2PIR.** Preliminary PIR works [KO97, CKGS98, BI01, Yek08, Efr12, DG16, Lip05, GR05, DC14, KLL+15, LP17] featured linear server time and sublinear bandwidth. To reduce server time, several works [BIM00, DCIO01, GMP16, DvDF+16, GCM+16, ACLS18] proposed *preprocessing PIR*. These approaches require a prohibitive amount of server storage due to a large server-side data structure. Recently a new type of preprocessing PIR with *offline client-side preprocessing* was proposed by Corrigan-Gibbs and Kogan [CGK20]. Introduced as 2PIR, their scheme has sublinear server time and *no additional server storage*—the preprocessing phase outputs just a few bits stored at the client. A simplified, stripped-down[1] version of their protocol, involving three parties, **client**, **server**$_1$ and **server**$_2$, is given below.

- *Offline phase.* **client** sends $\sqrt{n}$ random index sets $S_1, \ldots, S_{\sqrt{n}}$ of size $\sqrt{n}$ each to **server**$_1$ and **server**$_1$ returns database parities $p_1, \ldots, p_{\sqrt{n}}$, where $p_i = \oplus_{j \in S_i} \mathsf{DB}[j]$. These database parities, along with the respective index sets, are then stored by **client** locally.

- *Online phase (query to index i).* In Step 1, **client** finds a local set $S_j$ that contains $i$ and sends $S'_j = S_j \setminus \{i\}$ to **server**$_2$. In Step 2, **server**$_2$ returns parity $p'_j$ of $S'_j$, and **client** computes $\mathsf{DB}[i] = p_j \oplus p'_j$. In the final step, **client** generates a fresh random set $S^*_j$ that contains $i$, sends $S^*_j \setminus \{i\}$ to **server**$_1$, gets back its parity $p^*_j$, and replaces $(S_j, p_j)$ with $(S^*_j, p^*_j \oplus \mathsf{DB}[i])$ (We note that the final step is crucially needed to maintain the distribution of the sets at the server side and ensure security of future queries.)

While the complexities of the above protocol are linear (such as client storage and bandwidth), Corrigan-Gibbs and Kogan [CGK20] achieved $\tilde{O}(\sqrt{n})$ complexities by introducing the notion of *pseudorandom sets*: Instead of sending the sets in plaintext, the client sends a Pseudorandom Permutation (PRP) key so that the server can regenerate the sets as well as check membership efficiently. However, the first step of the online phase above requires removing element $i$ from the set $S_j$. This cannot be done efficiently with a PRP key, so Corrigan-Gibbs and Kogan [CGK20] sent $S_j \setminus \{i\}$ in plaintext, incurring $O(\sqrt{n}\log n)$ online bandwidth. In a followup work, Shi et al. [SACM21] addressed this issue. They use no PRPs and construct their sets via *privately-puncturable pseudorandom functions* [BKM17, CC17]. Their primitive allows element removal without key expansion in the online phase, thus keeping a short set description, yielding $\tilde{O}(1)$ bandwidth.

**Compiling 2PIR into 1PIR.** The original protocol by Corrigan-Gibbs and Kogan [CGK20], their follow-up work [KCG21], as well as Shi et al.'s polylog bandwidth protocol [SACM21], are all 2PIR protocols. Henzinger et. al. [CGHK22] showed how to port the 2PIR protocols by Corrigan-Gibbs and Kogan [CGK20, KCG21] into a 1PIR scheme with the same (amortized) $\tilde{O}(\sqrt{n})$ complexities. Their main technique, is to transform their initial 2PIR scheme [CGHK22] into another 2PIR scheme that *avoids communication with*

[1]In particular, in Step 1 of the actual protocol's online phase, the client sends $S_j \setminus \{i\}$ with probability $1 - 1/\sqrt{n}$ and $S_j \setminus \{r\}$, for a random element $r$, with probability $1/\sqrt{n}$, to ensure no information is leaked about $i$. Also, $\omega(\log \lambda)$ parallel executions are required to guarantee overwhelming correctness in $\lambda$, e.g., when no set $S_j$ can be found that contains $i$.

Table 1: Comparison with related work. Server time and bandwidth are amortized (indicated with $^*$). All schemes presented have $\tilde{O}(\sqrt{n})$ client time, $\tilde{O}(\sqrt{n})$ client space and no additional server space.

| scheme | model | server time$^*$ | bandwidth$^*$ | assumption |
|---|---|---|---|---|
| [CGHK22] | 1PIR | $\tilde{O}(\sqrt{n})$ | $\tilde{O}(\sqrt{n})$ | LWE |
| [SACM21] | 2PIR | $\tilde{O}(\sqrt{n})$ | $\tilde{O}(1)$ | LWE |
| Theorem 5.2 | 1PIR | $\tilde{O}(\sqrt{n})$ | $\tilde{O}(1)$ | LWE |

**server**$_1$ *in the online phase*. We call such a 2PIR protocol 2PIR+. Then, they use fully-homomorhpic encryption (FHE) [Gen09] to execute both offline and online phases on the same server, yielding 1PIR. To build the crucial 2PIR+ protocol, they make two simple modifications of the high-level protocol presented before: (i) In the offline phase, instead of preprocessing $\sqrt{n}$ sets, they preprocess $\sqrt{n} + Q$ sets, where $Q = \sqrt{n}$ is the number of queries they wish to support; (ii) In the final step of the online phase, instead of picking a fresh random set $S_j^*$ and then communicating with **server**$_1$, they use a preprocessed set $S_h$ from above, *avoiding communication with* **server**$_1$ *in the online phase*. Crucially, $S_h$ must then be updated to contain $i$. After $Q$ queries there are no more preprocessed sets left and the offline phase is run again, maintaining the same amortized complexity.

Based on the above, it seems that a natural approach to construct a sublinear-time, polylog-bandwidth 1PIR scheme (which is the central contribution of this paper) would be to apply the same trick of preprocessing an additional $Q$ random sets to the Shi et al. protocol [SACM21]. But this strategy runs into a fundamental issue: We would have to ensure that, in Step 3 of the online phase, when we use one of the preprocessed sets, $S_h$, to replace the set that was just consumed to answer query $i$, the set key corresponding to $S_h$ *would have to be updated to contain* $i$. However, this is not supported in the current construction of pseudorandom sets by Shi et al. [SACM21]—one can only remove elements, but not add. Our work capitalizes on this observation.

**Technical highlight: Adaptable pseudorandom sets.** A substantial part of our contribution is to define and construct an *adaptable pseudorandom set* supporting *both* element removal and addition. In fact, our technique can support addition and removal of a constant number of elements. At a high level, our primitive can be used as follows. Key generation outputs a succinct key $sk$ representing the set. Along with algorithms for enumeration of $sk$ and membership checking in $sk$, we define algorithms for removing an element $x$ from the set defined by $sk$ and adding an element $x$ into the set defined by $sk$, both of which output the updated set's new key $sk'$. We believe that this primitive can also be of independent interest outside of PIR.

**Our final 2PIR+ and 1PIR protocols.** Armed with adaptable pseudorandom sets, a high-level description of our new 2PIR+ scheme is as follows. Below, APRS denotes "adaptable pseudorandom set".

- *Offline phase.* **client** sends $\sqrt{n} + Q$ APRS keys $sk_1$, ..., $sk_{\sqrt{n}+Q}$ to **server**$_1$ and **server**$_1$ returns database parities $p_1, \ldots, p_{\sqrt{n}+Q}$ where $p_i = \oplus_{j \in sk_i} \mathsf{DB}[j]$. The database parities are then stored locally by **client**, together with the respective APRS keys.

- *Online (query to index i).* First, **client** finds APRS key $sk_j$ that contains $i$, **removes** $i$ from $sk_j$ and sends $sk'_j$ to **server**$_2$. Then **server**$_2$ returns parity $p'_j$ of $sk'_j$, and **client** computes $\mathsf{DB}[i] = p_j \oplus p'_j$. Finally, **client adds** $i$ into key $sk_h$ (for some $h > \sqrt{n}$) and replaces $(sk_j, p_j)$ with $(sk_h, p_h \oplus \mathsf{DB}[i])$.

The above 2PIR+ protocol requires more work to ensure a small probability of failure and that the server's view is uniform. Also, again, we can convert the above 2PIR+ protocol to 1PIR with sublinear complexities, using FHE [CGHK22]. Note that using FHE naively for 1PIR would incur $\tilde{O}(n)$ server time—thus combining FHE with our above 2PIR+ protocol yields a much better (sublinear) FHE-based 1PIR instantiation.

**Our result and comparison with related work.** As we discussed, if we require the server time to be sublinear (with no additional storage), the most bandwidth-efficient 2PIR protocol is the one by Shi et al. [SACM21]. However, when we consider 1PIR, the Corrigan-Gibbs, Henzinger and Kogan [CGHK22] construction increases the bandwidth from polylogarithmic to $O(\sqrt{n}\log n)$.

In this paper, we fill this gap. Our result (Theorem 5.2) provides the first 1PIR protocol with *sublinear amortized server time and polylogarithmic amortized bandwidth*.

We note that our scheme is optimal up to polylogarithmic factors in every dimension, given known lower bounds for client-dependent preprocessing PIR [BIM00, CGK20, CGHK22]. For client-independent preprocessing PIR, Persiano and Yeo [PY22] showed the product of online bandwidth and amortized server time should be linear, and existing constructions are either very far from realizing this bound [BIM00, GHPS22] or use infeasible primitives such as Virtual Blackbox Obfuscation [BIPW17]. For a comparison with prior sublinear-server-time-no-additional-server-storage schemes, see Table 1.

**Concurrent work.** We note independently the notion of 1PIR with polylogarithmic bandwidth and sublinear server time was studied by Zhou et al. [ZLTS22], whose work appeared on the crypto eprint archive subsequent to a submission of an earlier manuscript of this work.

**Notation.** We use the abbreviation PPT to refer to probabilistic polynomial time. Unless otherwise noted, we define a negligible function $\texttt{negl}(\cdot)$ to be a function such that for every polynomial $p(\cdot)$, $\texttt{negl}(\cdot)$ is less than $1/p(\cdot)$. We fix $\lambda \in \mathbb{N}$ to be a security parameter. We will also use the notation $1^z$ or $0^z$ to represent 1 or 0 repeated $z$ times. For any vector or bitstring $V$, we index $V$ using the notation $V[i]$ to represent the $i$-th element or $i$-th bit of $V$, indexed from 0. We will also use the notation $V[i:]$ to denote $V$ from the $i$-th index onwards. We use $x||y$ to denote the concatenation of bitsring $x$ and bitstring $y$. We use $S \sim D$ to denote that $S$ is "sampled from distribution" $D$. We use the notation $[x, y]$ to represent the set $\{x, x+1, \ldots, y-1\}$. Finally, we use $\tilde{O}(\cdot)$ to denote the big-O notation that ignores polylogarithmic terms and any polynomial terms in the security parameter $\lambda$.

## 2 Background: PIR, Puncturable Functions and Puncturable Sets

We now introduce definitions for 2PIR. We consider 2PIR protocols where only one server (the second one) participates in the online phase. We refer to these protocols as 2PIR+. We also introduce privately-puncturable PRFs [BKM17] and privately-puncturable pseudorandom sets [CGK20, SACM21], both crucial for our work. Moving forward, "PRF" stands for "pseudorandom function" and "PRS" stands for "pseudorandom set".

**Definition 2.1** (2PIR+ scheme)**.** *A 2PIR+ scheme consists of three stateful algorithms (***server**$_1$*,* **server**$_2$*,* **client***) with the following interactions.*

- ***Offline:*** **server**$_1$ *and* **server**$_2$ *receive the security parameter* $1^\lambda$ *and an* $n$*-bit database* $\mathsf{DB}$*.* **client** *receives* $1^\lambda$*.* **client** *sends one message to* **server**$_1$ *and* **server**$_1$ *replies with one message.*

- ***Online:*** *For each query* $x \in \{0, \ldots, n-1\}$*,* **client** *sends one message to* **server**$_2$ *and* **server**$_2$ *responds with one message. In the end,* **client** *outputs a bit* $b$*.*

**Definition 2.2** (2PIR+ correctness)**.** *A 2PIR+ scheme is correct if its honest execution, with any database* $\mathsf{DB} \in \{0,1\}^n$ *and any polynomial-sized sequence of queries* $x_1, \ldots, x_Q$*, returns* $\mathsf{DB}[x_1], \ldots, \mathsf{DB}[x_Q]$ *with probability* $1 - \texttt{negl}(\lambda)$*.*

**Definition 2.3** (2PIR+ privacy)**.** *A 2PIR+ scheme (***server**$_1$*,* **server**$_2$*,* **client***) is private if there exists a PPT simulator* $\texttt{Sim}$*, such that for any algorithm* $serv_1$*, no PPT adversary* $\mathcal{A}$ *can distinguish the experiments below with non-negligible probability.*

- ***Expt***$_0$*:* **client** *interacts with* $\mathcal{A}$ *who acts as* **server**$_2$ *and* ***server***$_1^*$ *who acts as the* **server**$_1$*. At every step* $t$*,* $\mathcal{A}$ *chooses the query index* $x_t$*, and* **client** *is invoked with input* $x_t$ *as its query.*

- ***Expt***$_1$*:* `Sim` *interacts with* $\mathcal{A}$ *who acts as* **server**$_2$ *and* ***server***$_1^*$ *who acts as the* **server**$_1$*. At every step* $t$*,* $\mathcal{A}$ *chooses the query index* $x_t$*, and* `Sim` *is invoked with no knowledge of* $x_t$*.*

We note that in the above definition our adversary $\mathcal{A}$ can deviate arbitrarily from the protocol. Intuitively the privacy definition implies that queries made to **server**$_2$ will appear random to **server**$_2$, assuming servers do not collude (as is the case in our model). Also, note that the above definition only captures privacy for **server**$_2$ since by Definition 2.1, **server**$_1$ interacts with **client** *before* the query indices are picked.

**Privately-puncturable PRFs.** A puncturable PRF is a PRF $F$ whose key $k$ can be punctured at some point $x$ in the domain of the PRF, such that the output punctured key $k_x$ reveals nothing about $F_k(x)$ [GGM84]. A privately-puncturable PRF is a puncturable PRF where the punctured key $k_x$ also reveals *no* information about the punctured point $x$ (by re-randomizing the output $F_k(x)$). Privately-puncturable PRFs can be constructed from standard $\mathsf{LWE}$ (learning with errors assumption) [BKM17, BTVW17, CC17] and can be implemented to allow puncturing on $m$ points at once [BKM17]. We now give the formal definition.

**Definition 2.4** (Privately-puncturable PRF [BKM17])**.** *A privately-puncturable PRF has four algorithms: (i)* $\mathsf{Gen}(1^\lambda, L, m) \to sk$*: Outputs secret key* $sk$*, given security parameter* $\lambda$*, input length* $L$ *and number of points to be punctured* $m$*; (ii)* $\mathsf{Eval}(sk, x) \to b$*: Outputs the evaluation bit* $b \in \{0,1\}$*, given* $sk$ *and input* $x$*; (iii)* $\mathsf{Puncture}(sk, P) \to sk_P$*: Outputs punctured key* $sk_P$*, given* $sk$ *and set* $P$ *of* $m$ *points for puncturing; (iv)* $\mathsf{PEval}(sk_P, x) \to b$*: Outputs the evaluation bit* $b \in \{0,1\}$*, given* $sk_P$ *and* $x$*.*

There are three properties we require from a privately-puncturable PRF: First, *functionality preservation*, meaning that $\mathsf{PEval}(sk_P, x)$ equals $\mathsf{Eval}(sk, x)$ for all $x \notin P$. Second, *pseudorandomness*, meaning that the values $\mathsf{Eval}(sk, x)$ at $x \in P$, appear pseudorandom to the adversary that has access to $sk_P$, as long as the adversary cannot query $\mathsf{PEval}(sk_P, x)$ for $x \in P$, in which case it is trivial to distinguish. Third, *privacy with respect to puncturing*, meaning that the punctured key $sk_P$ does not reveal anything about the set of points that was punctured. Formal definitions are in Appendix A ( Definitions A.2, A.1, A.3).

It is important to note here that we will be using a privately-puncturable PRF with a *randomized* puncturing algorithm. Although initial constructions were deterministic [BKM17], Canetti and Chen [CC17] show how to support randomized puncturing without extra assumptions and negligible extra cost.

**Privately-puncturable PRSs.** A privately-puncturable PRS is a set that contains elements drawn from a given distribution $\mathbb{D}_n$. The set can be represented succinctly with a key $sk$. Informally, one can "puncture" an element $x$, producing a new key that represents a set without $x$. Privatety-puncturable PRSs were first introduced by Corrigan-Gibbs and Kogan [CGK20] and were further optimized by Shi et al. [SACM21] (See Figure 3 in Appendix C for the [SACM21] construction.) The formal definition is as follows.

**Definition 2.5** (Privately-puncturable PRS [CGK20, SACM21])**.** *A privately-puncturable PRS has four algorithms: (i)* $\mathsf{Gen}(1^\lambda, n) \to (msk, sk)$*: Outputs a set key* $sk$ *and a master key* $msk$*, given security parameter* $\lambda$ *and the set domain* $\{0, \ldots, n-1\}$*; (ii)* $\mathsf{EnumSet}(sk) \to S$*: Outputs set* $S$ *given* $sk$*; (iii)* $\mathsf{InSet}(sk, x) \to b$*: Outputs a bit* $b$ *denoting whether* $x \in \mathsf{EnumSet}(sk)$*; (iv)* $\mathsf{Resample}(msk, x) \to sk_x$*: Outputs a secret key* $sk_x$ *for a set generated by* $sk$*, with* $x$*'s membership resampled.*[2]

We require three properties from a privately-puncturable PRS: First, *pseudorandomness with respect to a distribution* $\mathbb{D}_n$, meaning that $\mathsf{Gen}(1^\lambda, n)$ generates a key that represents a set whose distribution is indistinguishable from $\mathbb{D}_n$. Second, *functionality preservation with respect to resampling*, informally meaning that the set resulting from resampling should be a subset of the original set. This means we can

[2]Previously this was called "puncture". We rename it to "resample" for ease of understanding and consistency with our work.

only resample elements already in the set. Third, *security in resampling*, states that for some $(msk, sk)$ output by $\mathsf{Gen}(1^\lambda, n)$, $sk$ is computationally indistinguishable from a key $sk'_x$ where $(msk', sk')$ is a key output by calling $\mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk, x) \to 1$ and $sk'_x$ is the output of $\mathsf{Resample}(msk', x)$. Formal definitions can be found in Appendix A (Definitions A.6, A.4, A.5).

**Privately-puncturable PRSs from privately-puncturable PRFs.** Shi et al. [SACM21] constructed a privately-puncturable PRS from a privately-puncturable PRF. Let $F$ be a privately-puncturable PRF and let $x \in \{0,1\}^{\log n}$ be an element of the set domain. We provide the intuition behind the construction. Consider that we require both concise description and fast membership testing. One first approach to constructing a PRS could be to define $x \in S$ iff $F.\mathsf{Eval}(sk, x)$ is 1. Resampling $x$ would then be equivalent to puncturing $F$'s key at point $x$. Unfortunately, this approach creates sets proportional to the size of the PRF domain, which is undesirable for our application; we want sets of size approximately $\sqrt{n}$. To deal with this problem, one can add additional constraints with respect to suffixes of $x$. In other words, define $x \in S$ iff $F.\mathsf{Eval}(sk, x[i:])$ equals 1, for all $i = 1, \ldots, m$, where $m = \log n/2$. Recall $x[i:]$ denotes the suffix of bitstring $x$ starting at position $i$. Puncturing in this case would require puncturing at $m$ points. While this approach generates sets of expected size $\sqrt{n}$, it introduces too much dependency between elements in the set: Elements with shared suffixes are very likely to be together in the set. To deal with this, Shi et al. [SACM21] changed the construction as follows. Let $B$ be an integer greater than 0. Then, let $z = 0^B || x$. We say that $x \in S$ iff

$$F.\mathsf{Eval}(sk, z[i:]) = 1, \text{ for all } i = 1, \ldots, m + B\,.$$

For clarity we provide a small example here. Suppose $n = 16$ and that we want to check the membership of element 7 for set $S$. First, we represent 7 with $\log 16 = 4$ bits, $7_2 = 0111$. Next, we append $B = 4$ zeros to the front of the bitstring, so that we have the string 00000111. Now, we say that $7 \in S$ iff

$$F.\mathsf{Eval}(sk, 00000111) = 1 \wedge F.\mathsf{Eval}(sk, 0000111) = 1 \wedge F.\mathsf{Eval}(sk, 000111) = 1$$
$$\wedge\, F.\mathsf{Eval}(sk, 00111) = 1 \wedge F.\mathsf{Eval}(sk, 0111) = 1 \wedge F.\mathsf{Eval}(sk, 111) = 1\,.$$

Note that adding these $B$ extra checks decreases dependency between elements proportional to $2^B$, since it adds bits unique to each element. As a tradeoff, it decreases the size of the set proportional to $2^B$. By picking $B = \lceil 2 \log \log n \rceil$, we maintain the set size to be $\sqrt{n}/\log^2 n$ while having small enough dependency between elements—which can be addressed. We give an overview of our remaining algorithms below.

*Set enumeration.* Naively, set enumeration would take $n(m+B)$ time, since it requires checking membership for each element in $\{0, \ldots, n-1\}$. However, Shi et al. [SACM21] observed that due to the light dependency introduced, we can enumerate the set in expected time $\tilde{O}(\sqrt{n})$.

*Resampling.* To resample an element $x$ from the set $S$, we puncture the PRF key at the $M = m + B = \log n/2 + 2 \log \log n$ points that determine $x$'s membership by running

$$sk_x \leftarrow F.\mathsf{Puncture}(sk, \{z[i:]\}_{i=1,\ldots,M})\,.$$

By the pseudorandomness of $F$, this will resample $x$'s membership in $S$ and $x$ will not be in the set defined by $sk_x$ with probability $1 - 1/2^M = 1 - 1/\sqrt{n} \log^2 n$. Clearly, we do not remove elements from the set with overwhelming probability. Aside from that, there is still dependency among elements, and puncturing $x$ may also remove other elements in $S$ with some small probability. Shi et al. [SACM21] resolve this by bounding these probabilities to less than 1/2 and running $\lambda$ parallel executions of the protocol and taking a majority. Looking ahead, we will require this too.

*Key generation.* By Definition 2.5, key generation for a privately-puncturable PRS outputs two keys, key $sk$ that represents the initial set and key $msk$ that is used for puncturing. To output $msk$, we simply call $F.\mathsf{Gen}(1^\lambda, L, M)$. To output $sk$, we pick a set $P$ of $M$ "useless" strings of $L = \log n + B$ bits that start with the 1 bit and output a second key $sk \leftarrow F.\mathsf{Puncture}(msk, P)$. The reason for that is to ensure that resampled keys keys are indisinguishable from freshly sampled keys as required by the "security in resampling" property. Therefore we artificially puncture $msk$ in a way that does not affect the set of elements represented by it, yet we change its format to be the same as a set key resampled at a given point.

*Efficiency and security.* To summarize, the scheme described above by Shi et al. [SACM21] has the following complexities: Algorithms $\mathsf{Gen}$, $\mathsf{InSet}$ and $\mathsf{Resample}$ run in $\tilde{O}(1)$ time. All keys have $\tilde{O}(1)$ size. Algorithm $\mathsf{EnumSet}$ runs in expected $\tilde{O}(\sqrt{n})$ time. It satisfies Definitions A.4 and A.5 assumming privately-puncturable PRFs (that satisfy Definitions A.2, A.1, A.3).

## 3 Preliminary 2PIR+ Protocol

We first design a preliminary 2PIR+ protocol (Figure 1) that helps with the exposition of our final protocol. In this preliminary 2PIR+ protocol the client has linear local storage and the communication is amortized $\tilde{O}(\sqrt{n})$. Later, we will convert this 2PIR+ scheme into a space and communication-efficient 2PIR+ protocol (by using our new adaptable PRS primitive of Section 4) that will yield our final 1PIR scheme. Crucially, the analysis of the preliminary protocol is almost the same as that of our final PIR protocol in Section 5.

**Overview of our preliminary protocol.** Our preliminary protocol works as follows. During the *preprocessing* phase, the client constructs a collection $\mathsf{T}$ of $\ell = \sqrt{n}\log^3 n$ "primary" sets and a collection $\mathsf{Z}$ of an additional $\sqrt{n}$ "reserve" sets. All sets are sampled from a fixed distribution $\mathbb{D}_n$ over the domain $\{0,\ldots,n-1\}$. While we can use any distribution for our preliminary protocol, we use a specific one that will serve the use of PRSs in Section 5. Both $\mathsf{T}$ and $\mathsf{Z}$ are sent to **server**$_1$ and client receives the hints back, as explained in the introduction. Client stores locally the collections $\mathsf{T}$ and $\mathsf{Z}$ along with the hints. This is the main difference with our final protocol, where we will be storing keys instead of the sets themselves. To query an index $x$ during the *query* phase, the client finds some $T_j = (S_j, p_j)$ in $\mathsf{T}$ such that that $S_j$ contains $x$, "removes" $x$ and sends the new set to **server**$_2$. Then **server**$_2$ computes the parity of the new set and sends the parity back, at which point the client can compute $\mathsf{DB}[x]$, by xoring **server**$_2$'s response with $p_j$. As we will see, element removal in this context means resampling the membership of $x$ via a `Resample` algorithm introduced below. To ensure the set distribution of $\mathsf{T}$ does not change across queries, our protocol has a *refresh* phase, where element $x$ is "added", to the next available reserve set, via an `Add` algorithm introduced below. The protocol allows for $\sqrt{n}$ queries and achieves amortized sublinear server time over these $\sqrt{n}$ queries. After $\sqrt{n}$ queries, we re-run the offline phase.

The above protocol can fail with constant probability, as we will analyze in Lemma 3.1 below. To avoid this, as we indicate at the top of Figure 1, we run $\log n \log\log n$ parallel instances of the protocol and take the majority bit as the output answer. We now continue with the detailed description of the building blocks (such as algorithms `Resample` and `Add`) that our protocol uses.

**Sampling distribution $\mathbb{D}_n$.** For our preliminary protocol we are using the same distribution as the one induced by the PRS construction by Shi et al. [SACM21] described in Section 2. This will help us seamlessly transition to our space-efficient protocol in Section 5. To sample a set $S$ with elements from the domain $\{0,\ldots,n-1\}$ we define, for all $x \in \{0,\ldots,n-1\}$,

$$x \in S \Leftrightarrow \texttt{RO}(z[i:]) = 1 \text{ for all } i \in 1,\ldots,m+B\,,$$

where we recall that $m = \log n/2, B = 2\log\log n$ and $z = 0^B||x$. Also, $\texttt{RO} : \{0,1\}^* \rightarrow \{0,1\}$ denotes a random oracle. We use the random oracle for exposition only—our final construction does not need one. Note for our preliminary protocol, the adversary cannot call the $\texttt{RO}$ function or otherwise all the sets would revealed. We also define $\mathbb{D}_n^x$ to be a distribution where a set $S$ is sampled from $\mathbb{D}_n$ *until* $x \in S$.

- Run $\log n \log\log n$ instances of the protocol below, output the majority bit $maj$ in Step 4 of Query.
- Use $maj$ as $\mathsf{DB}[x]$ in Step 2 of Refresh.

Offline phase: Preprocessing

1. **client** samples $\ell + \sqrt{n}$ sets from $\mathbb{D}_n$
$$S_1, \ldots, S_{\ell+\sqrt{n}}\,,$$
where $\ell = \sqrt{n}\log^3 n$.
2. **client** sends sets $S_1, \ldots, S_{\ell+\sqrt{n}}$ to **server**$_1$ and **server**$_1$ returns a set of bits $p_1, \ldots, p_{\ell+\sqrt{n}}$, where
$$p_i = \oplus_{j\in S_i}\mathsf{DB}[j]\,.$$
3. **client** stores pairs of sets/hints
$$\mathsf{T} = \{T_j = (S_j, p_j)\}$$
$$\mathsf{Z} = \{Z_k = (S_k, p_k)\}\,,$$
where $j \in [\ell]$ and $k \in [\ell+1, \ell+\sqrt{n}]$.

Online phase: Query (input is index $x \in \{0, \ldots, n-1\}$)

1. **client** finds the first $T_j = (S_j, p_j)$ in $\mathsf{T}$ such that $x \in S_j$. If such $T_j$ is not found, set $j = |\mathsf{T}| + 1$ and $T_j = (S_j, p_j)$ where $S_j \sim \mathbb{D}_n^x$ and $p_j$ is uniform bit.
2. **client** sends $S' = \texttt{Resample}(S_j, x)$ to **server**$_2$.
3. **server**$_2$ returns $r = \bigoplus_{k\in S'} \mathsf{DB}[k]$.
4. **client** computes $\mathsf{DB}[x] = r \oplus p_j$.

Online phase: Refresh (executed when $j \le |\mathsf{T}|$)

1. Let $Z_0 = (S_0, p_0)$ be the first item from $\mathsf{Z}$.
2. Let $S_0^* = \texttt{Add}(S_0, x)$, and
$$p_0^* = p_0 \oplus (\mathsf{DB}[x] \wedge (x \notin S_0))\,.$$
3. **client** sets $T_j = (S_0^*, p_0^*)$, where $T_j$ was consumed earlier, and removes $Z_0$ from $\mathsf{Z}$.

Figure 1: Our preliminary 2PIR+ protocol. With $n$ we denote the size of the database $\mathsf{DB}$ and $[\ell] = [1, \ell]$.

**Functions with respect to $\mathbb{D}_n$.** We define two functions with respect to the distribution $\mathbb{D}_n$ —these functions will be needed to describe our preliminary scheme. To define these functions, we first introduce what it means for two elements to be *related.*

**Definition 3.1.** *Function* $\texttt{Related}(x, y)$*, where* $x, y \in \{0, \ldots, n-1\}$*, returns a bit* $b \in \{0, 1\}$ *where* $b = 1$ *(in which case we say that* $x$ *is* related *to* $y$*) iff* $x$ *and* $y$ *share a suffix of length* $> \log n/2$ *in their binary representation.*

For example $\texttt{Related}(1000001, 1100001) = 1$ and $\texttt{Related}(1000001, 1101111) = 0$. Equipped with this, we define our two functions.

- $\texttt{Resample}(S, x) \to S'$: Define $z = 0^B||x$ for some $x \in S$. We sample a uniform bit for each suffix of $z$, $z[i:]$, for $i \in 1, \ldots, m + B$. For each $y \in S$ such that $\texttt{Related}(x, y)$ (including $x$), we check if any suffix of $y$ was mapped to 0, and if so, remove it from $S$ and return this new set.
- $\texttt{Add}(S, x) \to S'$: This function essentially "reprograms" the random oracle such that $\texttt{RO}(z[i:]) = 1$ for all $i = 1, \ldots, m + B$, where $z = 0^B||x$. This may also affect membership of other elements $y \in \{0, \ldots, n-1\}$ not is $S$ related to $x$ with some probability. For us it will suffice that for most of executions, $\texttt{Add}(S, x) = S \cup \{x\}$. We bound the probability of this formally in Appendix B.

**Efficiency analysis.** Our preliminary protocol in Figure 1 is rather inefficient. In particular, while the online server time is $\tilde{O}(\sqrt{n})$, client storage and computation is $\tilde{O}(n)$ and bandwidth is $\tilde{O}(\sqrt{n})$. Also, the preliminary protocol supports $\sqrt{n}$ queries, after which we need to re-run the offline phase.

**Correctness proof.** As we mentioned before, our basic protocol that does not run parallel instances, has constant failure probability, less than $1/2$. We prove this through Lemma 3.1.

**Lemma 3.1** (Correctness of protocol with no repetitions)**.** *Consider the protocol of Figure 1 with no repetitions and fix a query $x_i$. The probability that the returned bit* $\mathsf{DB}[x_i]$ *in Step 4 of* Query *is incorrect, assuming* $\mathsf{DB}[x_{i-1}]$ *used in Step 2 of* Refresh *is correct with overwhelming probability, is less than* $1/2$.

We give an overview of the intuition of the proof here and defer the full proof of Lemma 3.1 to Appendix B. We distinguish two cases. For the first query $x_1$, there are three cases where our protocol can fail. The first failure occurs if we cannot find an index $j$ in $\mathsf{T}$ such that $x \in S_j$ for $T_j = (S_j, p_j)$ (Step 1 of Query). We can bound this failure by $1/n$. The second failure occurs when our `Resample` function does not remove $x$. This happens with probability $1/\sqrt{n}\log^2 n$. The third failure case occurs when we remove $x$, but also remove an element other than $x$ within `Resample`. This can bounded by $1/2\log n$.

For every other query $x_i$, $i$ greater than 1, we must consider an additional failure case which occurs when, in the Refresh phase, we add an element other than $x$ within `Add`—which we can also bound by $1/2\log n$. Computing the final bound requires more work. It requires showing that Refresh only incurs a very small additional error probability to subsequent queries, which can also be bounded at the query step. We argue this formally in our proof of Theorem 3.1.

*Amplifying correctness via repetition.* In order to increase the correctness of our scheme, we run $k$ parallel instances of our protocol and set the output bit in Step 3 of Query to be equal to the majority of $\mathsf{DB}[x]$ over these $k$ instance. We run Refresh with the correct $\mathsf{DB}[x]$ computed in Query so that we can apply Lemma 3.1. Let $C$ be the event, where, over $k$ instances of our preliminary PIR scheme, more than $\frac{k}{2}$ instances output the correct $\mathsf{DB}[x]$. Using a standard lower-tail Chernoff bound, we have that, if $p > 1/2$ is the probability $\mathsf{DB}[x]$ is correct, $C$'s probability $> 1 - \exp(-\frac{1}{2p}k(p - \frac{1}{2})^2)$ which is overwhelming for $k = \omega(\log n)$, satisfying Definition 2.2. The same technique is used in our final PIR scheme.

**Privacy proof.** We now show that our preliminary PIR protocol satisfies privacy, per Definition 2.3. Proving privacy relies on two properties which follow in a straightforward manner from the construction of $\mathbb{D}_n$.

Property 1: Let $S \sim \mathbb{D}_n^x$ and $S' \sim \mathbb{D}_n$. Then `Resample`$(S, x)$ and $S'$ are computationally indistinguishable.

Property 2: Let $S \sim \mathbb{D}_n$ and $S' \sim \mathbb{D}_n^x$. Then `Add`$(S, x)$ and $S'$ are computationally indistinguishable.

For the first query, we pick an entry $T_j = (S_j, p_j)$ from $\mathsf{T}$ whose $S_j$ contains the index $x$ we want to query. Since $S_j$ is the first set in $\mathsf{T}$ to contain $x$, $S_j \sim \mathbb{D}_n^x$. By Property 1, since what **server**$_2$ sees is $S' =$ `Resample`$(S_j, x)$, $S'$ is indistinguishable from a random set drawn from $\mathbb{D}_n$, and therefore, the query reveals nothing about the query index $x$ to **server**$_2$.

For every other query, we argue that the Refresh step maintains the distribution of $\mathsf{T}$. Note that after a given set $S_j$ is used, re-using it for the same query *or* a different query could create privacy problems. That is why after each query, we must replace $S_j$ with an identically distributed set. By Property 2, $S_j$ and `Add`$(S_0, x)$ are identically distributed. Then, the swap maintains the distribution of sets in $\mathsf{T}$ and therefore the view of **server**$_2$ is also simulatable without $x$. These arguments form the crux of the proof of Theorem 3.1; we provide the full proof in Appendix B.

**Theorem 3.1** (Preliminary 2PIR+ protocol)**.** *The 2PIR+ scheme in Figure 1 is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i) $\tilde{O}(n)$ client storage $\tilde{O}(n)$ client time; (ii) $\tilde{O}(\sqrt{n})$ amortized server time and no additional server storage; (iii) $\tilde{O}(\sqrt{n})$ amortized bandwidth.*

## 4 Adaptable Pseudorandom Sets

In this section, we introduce the main primitive required for achieving our result, an *adaptable pseudorandom set*. The main difference from a privately-puncturable PRS introduced in Section 2 is the support for the "add" procedure, as well as any constant number of additions or removals, as opposed to a single removal. This will eventually allow us to port the protocol from Section 3 into a 1PIR protocol that has much improved complexities, such as sublinear client storage and polylogarithmic communication. We now give the formal definition and then we present a construction that satisfies our definition.

**Definition 4.1** (Adaptable PRS). *An adaptable PRS has five algorithms: (i)* $\mathsf{Gen}(1^\lambda, n) \to (msk, sk)$*: Outputs our set's key* $sk$ *and master key* $msk$*, given security parameter* $\lambda$ *and set domain* $\{0, \ldots, n-1\}$*; (ii)* $\mathsf{EnumSet}(sk) \to S$*: Outputs set* $S$ *given* $sk$*; (iii)* $\mathsf{InSet}(sk, x) \to b$*: Outputs bit* $1$ *iff* $x \in \mathsf{EnumSet}(sk)$*; (iv)* $\mathsf{Resample}(msk, sk, x) \to sk_x$*: Outputs secret key* $sk_x$ *that corresponds to an updated version of the set (initially generated by* $sk$*) after element* $x$ *is resampled; (v)* $\mathsf{Add}(msk, sk, x) \to sk^x$*: Outputs secret key* $sk^x$ *that corresponds to an updated version of the set (initially generated by* $sk$*) after element* $x$ *is added.*

Note that our interface differs from privately-puncturable PRSs introduced in Section 2 in that our resample and add operations are dependent on both $msk$ and $sk$; we will see why below.

**Security definitions for adaptable PRSs.** Our adaptable PRS must satisfy five definitions. Three of them, *functionality preservation with respect to resampling*, *pseudorandomness with respect to a distribution* $\mathbb{D}_n$ and *security in resampling* are identical to the equivalent definitions from privately-puncturable PRSs, namely Definitions A.6, A.4, A.5 in Appendix A. We give two additional definitions in Appendix B (definitions A.8 and A.7) that relate to addition. First, *functionality preservation with respect to addition*, meaning that adding always yields a superset of the original set and can only cause elements related to $x$ (which are few) to be added to the set. Second, *security in addition*, meaning that generating fresh keys until we find one where $x$ belongs to the set is equivalent to generating one fresh key and then adding $x$ into it.

**Problems with enabling both addition and removal.** The first step towards supporting addition is finding a way to simulate the "Add" function from Section 3. In particular, to add element $x$, we can run the puncture operation many times until a punctured key of a set containing $x$ is output. This will take $\tilde{O}(\sqrt{n})$ time[3]

However, since the PRF's puncturing is only defined for $m$ points, we would not be able to remove an element $y$ after we add an element $x$. To deal with this issue, one idea is to instantiate our PRS with a privately-puncturable PRF that allows puncturing at $2m$ points. Again, this is problematic since our PRF interface only allows puncturing $2m$ points *at once*, and not *on demand* but we do need adding and removing elements on demand. For example, in our preliminary protocol of Section 3, when we add an element into a set $S$ at the end of query $i$ as part of $\mathsf{Refresh}$, we do not know which element we will be potentially removing from $S$ at the beginning of query $i+1$ since this depends on the next query. It also requires special attention to deal with puncturing and adding related elements, since we would have to puncture the key on the same point twice, an operation that is undefined.

**Intuition of our construction: Introduce an additional key.** Our core idea is to use two keys $sk[1]$ and $sk[2]$ and define the evaluation on the suffixes that determines membership as the XOR of $F.\mathsf{Eval}(sk[1], \cdot)$ and $F.\mathsf{Eval}(sk[2], \cdot)$. In this way, we can add to one key, and resample the other, independently. Note that this idea can support any fixed number of additions or resamplings (removals), by adding extra PRF keys. We present a summary of our construction below. The detailed implementation is in Figure 4 in Appendix C.

*Key generation.* Let $F$ be a privately-puncturable PRF. For key generation, we run $F.\mathsf{Gen}$ twice, outputting $msk[1]$ and $msk[2]$. After puncturing on $m$ "useless" points (for reasons we explained in Section 2), we output $sk[1]$ and $sk[2]$. And finally we output $sk = (sk[1], sk[2])$ and $msk = (msk[1], msk[2])$.

*Set membership and enumeration.* For each $x \in \{0, \ldots, n-1\}$ we define

$$x \in S \Leftrightarrow F.\mathsf{Eval}(sk[1], z[i:]) \oplus F.\mathsf{Eval}(sk[2], z[i:]) = 1 \text{ for all } i = 1, \ldots, m+B\,,$$

where we recall $m = \log n/2, B = 2\log\log n$ and $z = 0^B||x$. For enumeration, we use the same algorithm as Shi et al. [SACM21], with the difference that evaluation is done as the XOR of two evaluations, as above.

[3]Importantly, our constuction requires a *rerandomizable* privately-puncturable PRF, as mentioned before in Section 2.

• Run $\log n \log\log n$ instances of the protocol below, output the majority bit $maj$ in Step 4 of Query.
• Use $maj$ as $\mathsf{DB}[x]$ in Step 2 of Refresh.

Offline phase: Preprocessing

1. **client** generates $\ell + \sqrt{n}$ PRSet keys
$$(msk_1, sk_1), \ldots, (msk_{\ell+\sqrt{n}}, sk_{\ell+\sqrt{n}})$$
with $\mathsf{Gen}(1^\lambda, n), \ell = \sqrt{n}\log^3 n$.
2. **client** sends keys $sk_1, \ldots, sk_{\ell+\sqrt{n}}$ to **server**$_1$ and **server**$_1$ returns a set of bits $p_1, \ldots, p_{\ell+\sqrt{n}}$, where
$$p_i = \oplus_{j \in \mathsf{EnumSet}(sk_i)} \mathsf{DB}[j]\,.$$
3. **client** stores pairs of keys/hints
$$\mathsf{T} = \{T_j = (msk_j, sk_j, p_j)\}\,,$$
$$\mathsf{Z} = \{Z_k = (msk_k, sk_k, p_k)\}\,,$$
where $j \in [\ell]$ and $k \in [\ell+1, \ell+\sqrt{n}]$.

Online phase: Query (input is index $x \in \{0, n-1\}$)

1. **client** finds the first $T_j = (msk_j, sk_j, p_j)$ in $\mathsf{T}$ such that $\mathsf{InSet}(sk_j, x) = 1$. If such $T_j$ is not found, set $j = |\mathsf{T}| + 1$ and $T_j = (msk_j, sk_j, p_j)$ where $\mathsf{Gen}(1^\lambda, n) \to (msk_j, sk_j)$ and $p_j$ is uniform bit.
2. **client** sends $sk' \leftarrow \mathsf{Resample}(msk_j, sk_j, x)$ to **server**$_2$.
3. **server**$_2$ returns $r = \bigoplus_{k \in \mathsf{EnumSet}(sk')} \mathsf{DB}[k]$.
4. **client** computes $\mathsf{DB}[x] = r \oplus p_j$.

Online phase: Refresh (executed when $j \leq |\mathsf{T}|$)

1. Let $Z_0 = (msk_0, sk_0, p_0)$ be the first item from $\mathsf{Z}$.
2. Let $sk_0^* \leftarrow \mathsf{Add}(msk_0, sk_0, x)$ and
$$p_0^* = p_0 \oplus (\mathsf{DB}[x] \wedge (\neg \mathsf{InSet}(x, sk_0)))\,.$$
3. **client** sets $T_j = (msk_0, sk_0^*, p_0^*)$, where $T_j$ was consumed earlier, and removes $Z_0$ from $\mathsf{Z}$.

Figure 2: Our 2PIR+ protocol for $n$-bit DB using adaptable PRS ($\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}, \mathsf{Add}$).

*Resampling.* Resampling works exactly as resampling in privately-puncturable PRSs (by calling $F.\mathsf{Puncture}$) and uses, without loss of generality, $msk[2]$ as input. The output replaces only the second part of $sk$—thus we require $sk$ as input so that we can output the first part intact.

*Addition.* To add an element $x$, we call $F.\mathsf{Puncture}$ on input $msk[1]$ repeatedly until $x$'s membership test succeeds. Naively, this takes $\tilde{O}(\sqrt{n})$ time, but we show in the Appendix how to reduce this to $\tilde{O}(1)$ by leveraging the puncturable PRF used. Our final theorem is Theorem 4.1. Our final construction and the proof can be found in Appendix C.

**Theorem 4.1** (Adaptable PRS construction)**.** *Assuming* LWE*, the scheme in Figure 4 satisfies correctness, pseudorandomness with respect to $\mathbb{D}_n$ (Definition A.4), functionality preservation in resampling and addition (Definitions A.6 and A.8), security in resampling and addition (Definitions A.5 and A.7), and has the following complexities: (i) keys $sk$ and $msk$ have $\tilde{O}(1)$ size; (ii) membership testing, resampling and addition take $\tilde{O}(1)$ time; (iii) enumeration takes $\tilde{O}(\sqrt{n})$ time.*

## 5 More Efficient 2PIR+ and Near-Optimal 1PIR

We now use adaptable PRSs introduced in the previous section to build a more efficient 2PIR+ scheme (one with $\tilde{O}(\sqrt{n})$ client storage and $\tilde{O}(1)$ communication complexity) which can be compiled, using FHE, into a 1PIR scheme with the same complexities, as we explained in the introduction. The main idea is to replace the actual sets, stored by the client in their entirety in our preliminary protocol, with PRS keys that support succinct representation, addition and removal. In particular, our proposed protocol in Figure 2 is identical to our preliminary protocol in Figure 1 except for the following main points: (i) In the offline phase, instead of sampling sets from $\mathbb{D}_n$, we generate keys $(msk, sk)$ for adaptable PRSs that correspond to sets of the same distribution $\mathbb{D}_n$. (ii) In the online phase, we run `Resample` and `Add` defined in the adaptable PRS. These have exactly the same effect in the output set, except the operations are done on the set key not the set. (iii) We can check membership efficiently using $\mathsf{InSet}$. We now introduce Theorem 5.1.

**Theorem 5.1** (Efficient 2PIR+ protocol)**.** *Assuming* $\mathsf{LWE}$*, the 2PIR+ scheme in Figure 2 is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i)* $\tilde{O}(\sqrt{n})$ *client storage and* $\tilde{O}(\sqrt{n})$ *client time; (ii)* $\tilde{O}(\sqrt{n})$ *amortized server time and no additional server storage; (iii)* $\tilde{O}(1)$ *amortized bandwidth.*

**Unlimited queries.** Our scheme can handle $\sqrt{n}$ queries but can be extended to unlimited queries: We just rerun the offline phase after all secondary sets are used. This maintains the complexities from Theorem 5.1.

**From 2PIR+ to 1 PIR with same complexities.** As detailed in [CGHK22], we can port our 2PIR+ to 1PIR by merging **server**$_1$ and **server**$_2$ and executing the work of **server**$_1$ using FHE. We require a symmetric key FHE scheme that is *gate-by-gate* [CGHK22], where *gate-by-gate* means that encrypted evaluation runs in time $\tilde{O}(|C|)$ for a circuit of size $|C|$. As noted in [CGHK22], this is a property of standard FHE based on LWE [BV11, GSW13]. With this, we can use a batch parity Boolean circuit $C$ that, given a database of size $n$ and $l$ lists of size $m$, $C$ computes the parity of the lists in $\tilde{O}(l \cdot m + n)$ time [CGHK22]. Our main result, Theorem 5.2, is as follows.

**Theorem 5.2** (Near-Optimal 1PIR protocol)**.** *Assuming* $\mathsf{LWE}$*, there exists an 1PIR scheme that is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i)* $\tilde{O}(\sqrt{n})$ *client storage and* $\tilde{O}(\sqrt{n})$ *client time; (ii)* $\tilde{O}(\sqrt{n})$ *amortized server time and no additional server storage; (iii)* $\tilde{O}(1)$ *amortized bandwidth.*

Our proof for both theorems introduced are located in Appendix D, but follow closely from our adaptable pseudorandom set and the proof from our preliminary protocol, along with the tools introduced above.

## References

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, May 2018. ISSN: 2375-1207.

[AS16] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, OSDI'16, pages 551–569, USA, November 2016. USENIX Association.

[BBG$^+$20] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, pages 1253–1269, New York, NY, USA, October 2020. Association for Computing Machinery.

[BI01] Amos Beimel and Yuval Ishai. Information-Theoretic Private Information Retrieval: A Unified Construction. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 2076, pages 912–926. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. Series Title: Lecture Notes in Computer Science.

[BIM00] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In Mihir Bellare, editor, *Advances in Cryptology — CRYPTO 2000*, Lecture Notes in Computer Science, pages 55–73, Berlin, Heidelberg, 2000. Springer.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can We Access a Database Both Locally and Privately? pages 662–693, November 2017.

[BKM17] Dan Boneh, Sam Kim, and Hart Montgomery. Private Puncturable PRFs from Standard Lattice Assumptions. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, Lecture Notes in Computer Science, pages 415–445, Cham, 2017. Springer International Publishing.

[BKMP12] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *2012 IEEE Symposium on Security and Privacy*, pages 257–271, May 2012. ISSN: 2375-1207.

[BTVW17] Zvika Brakerski, Rotem Tsabary, Vinod Vaikuntanathan, and Hoeteck Wee. Private Constrained PRFs (and More) from LWE. In Yael Kalai and Leonid Reyzin, editors, *Theory of Cryptography*, Lecture Notes in Computer Science, pages 264–302, Cham, 2017. Springer International Publishing.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Phillip Rogaway, editors, *Advances in Cryptology – CRYPTO 2011*, volume 6841, pages 505–524. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. Series Title: Lecture Notes in Computer Science.

[CC17] Ran Canetti and Yilei Chen. Constraint-Hiding Constrained PRFs for NC$$^1$$from LWE. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017*, Lecture Notes in Computer Science, pages 446–476, Cham, 2017. Springer International Publishing.

[CGHK22] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II*, pages 3–33, Berlin, Heidelberg, May 2022. Springer-Verlag.

[CGK20] Henry Corrigan-Gibbs and Dmitry Kogan. Private Information Retrieval with Sublinear Online Time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, Lecture Notes in Computer Science, pages 44–75, Cham, 2020. Springer International Publishing.

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

[DC14] Changyu Dong and Liqun Chen. A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In Mirosław Kutyłowski and Jaideep Vaidya, editors, *Computer Security - ESORICS 2014*, volume 8712, pages 380–399. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science.

[DCIO01] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal Service-Providers for Private Information Retrieval. *Journal of Cryptology*, 14(1):37–74, January 2001.

[DCMO00] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single Database Private Information Retrieval Implies Oblivious Transfer. In Bart Preneel, editor, *Advances in Cryptology — EUROCRYPT 2000*, Lecture Notes in Computer Science, pages 122–138, Berlin, Heidelberg, 2000. Springer.

[DG16] Zeev Dvir and Sivakanth Gopi. 2-Server PIR with Subpolynomial Communication. *Journal of the ACM*, 63(4):1–15, November 2016.

[DvDF$^+$16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: 13th International Conference on Theory of Cryptography, TCC 2016. *Theory of Cryptography - 3th International Conference, TCC 2016-A, Proceedings*, pages 145–174, 2016. Publisher: Springer.

[Efr12] Klim Efremenko. 3-Query Locally Decodable Codes of Subexponential Length. *SIAM Journal on Computing*, 41(6):1694–1703, January 2012.

[GCM$^+$16] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 91–107, USA, March 2016. USENIX Association.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing - STOC '09*, page 169, Bethesda, MD, USA, 2009. ACM Press.

[GGM84] Oded Goldreich, S. Goldwasser, and S. Micali. How to Construct Random Functions (Extended Abstract). In *FOCS*, 1984.

[GHPS22] Daniel Günther, Maurice Heymann, Benny Pinkas, and Thomas Schneider. {GPU-accelerated} {PIR} with {Client-Independent} Preprocessing for {Large-Scale} Applications. pages 1759–1776, 2022.

[GMP16] Sanjam Garg, Payman Mohassel, and Charalampos Papamanthou. TWORAM: Efficient Oblivious RAM in Two Rounds with Applications to Searchable Encryption. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, Lecture Notes in Computer Science, pages 563–592, Berlin, Heidelberg, 2016. Springer.

[GR05] Craig Gentry and Zulfikar Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming*, volume 3580, pages 803–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.

[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, *Advances in Cryptology – CRYPTO 2013*, Lecture Notes in Computer Science, pages 75–92, Berlin, Heidelberg, 2013. Springer.

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

[KKYM20] Koki Kazama, Akira Kamatsuka, Takahiro Yoshida, and Toshiyasu Matsushima. A Note on a Relationship between Smooth Locally Decodable Codes and Private Information Retrieval. In *2020 International Symposium on Information Theory and Its Applications (ISITA)*, pages 259–263, October 2020. ISSN: 2689-5854.

[KLL$^+$15] Aggelos Kiayias, Nikos Leonardos, Helger Lipmaa, Kateryna Pavlyk, and Qiang Tang. Optimal Rate Private Information Retrieval from Homomorphic Encryption. *Proceedings on Privacy Enhancing Technologies*, 2015(2):222–243, June 2015.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 364–373, Miami Beach, FL, USA, 1997. IEEE Comput. Soc.

[Lip05] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In *Proceedings of the 8th international conference on Information Security*, ISC'05, pages 314–328, Berlin, Heidelberg, September 2005. Springer-Verlag.

[LP17] Helger Lipmaa and Kateryna Pavlyk. A Simpler Rate-Optimal CPIR Protocol. In *Financial Cryptography and Data Security, 2017*, 2017.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response Efficient Single-Server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 2292–2306, New York, NY, USA, November 2021. Association for Computing Machinery.

[PY22] Giuseppe Persiano and Kevin Yeo. Limits of Preprocessing for Single-Server PIR. Technical Report 235, 2022.

[SACM21] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *Advances in Cryptology - CRYPTO*, 2021.

[SCH$^+$21] Sudheesh Singanamalla, Suphanat Chunhapanya, Jonathan Hoyland, Marek Vavruša, Tanya Verma, Peter Wu, Marwan Fayed, Kurtis Heimerl, Nick Sullivan, and Christopher Wood. Oblivious DNS over HTTPS (ODoH): A Practical Privacy Enhancement to DNS. *Proceedings on Privacy Enhancing Technologies*, 2021(4):575–592, October 2021.

[Yek08] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. *Journal of the ACM*, 55(1):1–16, February 2008.

[Yek10] Sergey Yekhanin. *Locally Decodable Codes and Private Information Retrieval Schemes*. Information Security and Cryptography. Springer, Berlin, Heidelberg, 2010.

[ZLTS22] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. Optimal Single-Server Private Information Retrieval. *ePrint IACR*, 2022.

## A Definitions

In this section we give relevant definitions that our primitives satisfy.

### A.1 Definitions for Privately-puncturable PRFs

**Definition A.1** (Pseudorandomness for privately-puncturable PRFs)**.** *A privately-puncturable PRF scheme (*$\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval}$*) satisfies* pseudorandomness *if no PPT admissible adversary $\mathcal{A}$ can distinguish between the following experiments (An adversary is admissible if it never queries elements in $P$ on the original $sk$, and always picks a set $P$ of size $m$.)*

- $\mathsf{Gen}(\lambda, L, m) \to sk$, $\mathcal{A}(\lambda) \to P$, $\mathsf{Puncture}(sk, P) \to sk_P$;
  $\mathcal{A}^{\mathsf{Eval}(sk,.)}$ *is given* $(sk_P, \{\mathsf{Eval}(sk, x)\}_{x \in P})$.
- $\mathsf{Gen}(\lambda, L, m) \to sk$, $\mathcal{A}(\lambda) \to P$, $\mathsf{Puncture}(sk, P) \to sk_P$;
  $\mathcal{A}^{\mathsf{Eval}(sk,.)}$ *is given* $(sk_P, \{R_i\}_{i=1,\ldots,m})$*, where $R_i$ are sampled uniformly at random.*

**Definition A.2** (Functionality preservation in puncturing for privately-puncturable PRFs)**.** *A privately-puncturable PRF scheme (*$\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval}$*) satisfies* functionality preservation *if for any PPT adversary $\mathcal{A}$ that outputs set of $m$ points $P$ of length $\leq L$ each, there exists a negligible function $\mathtt{negl}(\cdot)$ such that, for the following experiment*

- $P \leftarrow \mathcal{A}(1^\lambda)$, $sk \leftarrow \mathsf{Gen}(1^\lambda, L, m)$, $sk_P \leftarrow \mathsf{Puncture}(sk, P)$;
- $x \leftarrow \mathcal{A}^{\mathsf{Eval}(sk,\cdot)}(sk_P)$.

*it holds that*

$$\Pr[(x \notin P) \wedge (\mathsf{Eval}(sk, x) \neq \mathsf{PEval}(sk_P, x))] \leq \mathtt{negl}(\lambda)\,.$$

**Definition A.3** (Privacy with respect to puncturing for privately-puncturable PRFs)**.** *A privately-puncturable PRF scheme (*$\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval}$*) satisfies* privacy with respect to puncturing *if for any PPT admissible adversary $\mathcal{A}$, experiments* $\mathrm{Expt}^0(\lambda, L, m)$ *and* $\mathrm{Expt}^1(\lambda, L, m)$ *are computationally indistinguishable (An adversary is admissible if it never queries elements in $P_1 \cup P_2 - P_1 \cap P_2$ [Babis: please check] on the original $sk$, and always picks sets of size $m$.) Experiment* $\mathrm{Expt}^b(\lambda, L, m)$ *is defined as follows.*

- $\mathsf{Gen}(\lambda, L, m) \to sk$, $\mathcal{A}(\lambda) \to (P_0, P_1)$, $\mathsf{Puncture}(sk, P_b) \to sk_{P_b}$;
- $b' \leftarrow \mathcal{A}^{\mathsf{Eval}(sk,\cdot)}(sk_{P_b})$.

### A.2 Definitions for Privately-Puncturable PRSs

We give here definitions for privately-puncturable PRSs as defined in previous work by Shi et. al. [SACM21].

**Definition A.4** (Pseudorandomness with respect to some distribution $\mathbb{D}_n$ for privately-puncturable PRSs)**.** *A privately-puncturable PRS scheme* ($\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}$) *satisfies* pseudorandomness with respect to some distribution $\mathbb{D}_n$ *if the distribution of* $\mathsf{EnumSet}(sk)$*, where $sk$ is output by* $\mathsf{Gen}(\lambda, n)$*, is indistinguishable from a set sampled from $\mathbb{D}_n$.*

**Definition A.5** (Security in resampling for privately-puncturable PRSs)**.** *A privately-puncturable PRS scheme (*$\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}$*) satisfies* security in resampling *if, for any $x \in \{1, \ldots, n-1\}$, the following two distributions are computationally indistinguishable.*

- *Run* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$*, output* $sk$*.*
- *Run* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$ *until* $\mathsf{InSet}(sk, x) \to 1$*, output* $sk_x = \mathsf{Resample}(msk, x)$*.*

**Definition A.6** (Functionality preservation in resampling for privately-puncturable PRSs)**.** *We say that a privately-puncturable PRS scheme* ($\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}$) satisfies functionality preservation in resampling *with respect to a predicate* $\texttt{Related}$ *if, with probability* $1 - \texttt{negl}(\lambda)$ *for some negligible function* $\texttt{negl}(.)$*, the following holds. If* $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ *and* $\mathsf{Resample}(msk, x) \to sk_x$ *where* $x \in \mathsf{InSet}(sk)$ *then*

1. $\mathsf{EnumSet}(sk_x) \subseteq \mathsf{EnumSet}(sk)$*;*
2. $\mathsf{EnumSet}(sk_x)$ *runs in time no more than* $\mathsf{EnumSet}(sk)$*;*
3. *For any* $y \in \mathsf{EnumSet}(sk) \setminus \mathsf{EnumSet}(sk_x)$*, it must be that* $\texttt{Related}(x, y) = 1$*.*

## A.3 Additional Definitions for Adaptable PRSs

Our adaptable PRS primitive will satisfy Definitions A.4, A.5 and A.6, as well as the counterparts for Definitions A.5 and A.6 (that account for the "add" functionality), which we state here.

**Definition A.7** (Security in addition for adaptable PRSs)**.** *We say that an adaptable PRS scheme (*$\mathsf{Gen}$*,* $\mathsf{EnumSet}$*,* $\mathsf{InSet}$*,* $\mathsf{Resample}$*,* $\mathsf{Add}$*) satisfies* security in addition *if, for any* $x \in \{0, \ldots, n-1\}$*, the following two distributions are computationally indistinguishable.*

- *Run* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$ *until* $\mathsf{InSet}(sk, x) \to 1$*, output* $sk$*.*
- *Run* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$*, output* $sk^x \leftarrow \mathsf{Add}(msk, sk, x)$*.*

**Definition A.8** (Functionality preservation in addition for adaptable PRS)**.** *We say that an adaptable PRS scheme* ($\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}, \mathsf{Add}$) *satisfies* functionality preservation in addition *with respect to a predicate* $\texttt{Related}$ *if, with probability* $1 - \texttt{negl}(\lambda)$ *for some negligible function* $\texttt{negl}(.)$*, the following holds. If* $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ *and* $\mathsf{Add}(msk, sk, x) \to sk^x$ *then*

- $\mathsf{EnumSet}(sk) \subseteq \mathsf{EnumSet}(sk^x)$*;*
- *For all* $y \in \mathsf{EnumSet}(sk^x) \setminus \mathsf{EnumSet}(sk)$ *it must be that* $\texttt{Related}(x, y) = 1$*.*

# B Correctness Lemmata

First, we prove Lemma 3.1. Then, we proceed to prove Theorem 3.1.

**Lemma 3.1** (Correctness of protocol with no repetitions)**.** *Consider the protocol of Figure 1 with no repetitions and fix a query* $x_i$*. The probability that the returned bit* $\mathsf{DB}[x_i]$ *in Step 4 of* $\mathsf{Query}$ *is incorrect, assuming* $\mathsf{DB}[x_{i-1}]$ *used in Step 2 of* $\mathsf{Refresh}$ *is correct with overwhelming probability, is less than* $1/2$*.*

*Proof.* Recall that we fix $B = 2 \log \log n$. As alluded to in Section 3, we can split our failure probability in three cases:

- Case 1: $x_i$ is not in any primary set that was preprocessed.
- Case 2: The resampling does not remove $x_i$.
- Case 3: Resampling removes *more* that just $x_i$ from the set.

*Case 1:*
We first note that, from our distribution $\mathbb{D}_n$, for any $x \in \{0, \dots, n-1\}$, we have that, for $S \sim \mathbb{D}_n$,

$$\begin{aligned}\Pr[x \in S] &= \left(\frac{1}{2}\right)^{\frac{1}{2}\log n + B} \\ &= \frac{1}{\sqrt{n}}\left(\frac{1}{2}\right)^{B} \\ &= \frac{1}{2^B\sqrt{n}}\,.\end{aligned}$$

Then note that the expected size of $S$ is the sum of the probability of each element being in the set, i.e.,

$$\begin{aligned}\mathbb{E}\left[|S|\right] &= \mathbb{E}\left[\sum_{x=0}^{n-1}\frac{1}{2^B\sqrt{n}}\right] \\ &= \sum_{x=0}^{n-1}\mathbb{E}\left[\frac{1}{2^B\sqrt{n}}\right] \\ &= \frac{\sqrt{n}}{2^B} \leq \frac{\sqrt{n}}{(\log n)^2}\,.\end{aligned}$$

We can conclude that the desired probability is

$$\begin{aligned}\Pr[x \notin \cup_{i\in[1,l]}S_i] &= \left(1 - \frac{1}{\sqrt{n}(\log n)^2}\right)^{\sqrt{n}(\log n)^3} \\ &= \left(\frac{1}{e}\right)^{\log n} \leq \frac{1}{n}\,,\end{aligned}$$

where $\ell = \sqrt{n}\log^3 n$ and $S_1, \dots, S_\ell \sim (\mathbb{D}_n)^\ell$.
*Case 2:*
Assuming there is a set $S$ such that $x_i \in S$, by construction of `Resample`, it is easy to see that the probability that $x_i$ is not removed from $S$ is equivalent to a Bernoulli variable that is 1 with probability $p = \frac{1}{\sqrt{n}\cdot 2^B}$, since we toss $1/2\log n + B$ coins, and $x$ is not removed only if all of these coins evaluate to 1. Therefore

$$\begin{aligned}\Pr[x_i \in \texttt{Resample}(S, x_i)] &= \frac{1}{\sqrt{n}\cdot 2^B} \\ &\leq \frac{1}{\sqrt{n}\log^2 n}\,.\end{aligned}$$

*Case 3:*
Note that for any $k$ less than $\log n$, there are exactly $2^{\log n - k} - 1$, or less than $2^{\log n - k}$ strings in $\{0,1\}^{\log n}$, that are different than $x$ share a suffix of length $\geq k$ with $x$. Note that since $x$ is in the set, for any $k$, the probability that a string $y$ that has a common suffix of length exactly $k$ with $x$ is included in the set is the chance that its initial $B$ bits *and* its remaining bits not shared with $x$ evaluate to 1, namely, for any $k$ less than $\log n$ and $y = \{0,1\}^k || x[k:]$ we have that

$$\Pr[y \in S] = \frac{1}{2^B 2^{\log n - k}}\,.$$

Let $N_k$ be the set of strings in the set that share a longest common suffix with $x$ of length $k$. Then, since we know that there are at most $2^{\log n - k}$ such strings, we can say that for any $k$, the expected size of $N_k$ is

$$\begin{aligned}\mathbb{E}\left[|N_k|\right] &\leq \mathbb{E}\left[\sum_{x=1}^{2^{\log n-k}} \frac{1}{2^B 2^{\log n - k}}\right] \\ &= \sum_{x=1}^{2^{\log n-k}} \mathbb{E}\left[\frac{1}{2^B 2^{\log n-k}}\right] \\ &= 2^{\log n-k}\frac{1}{2^B 2^{\log n-k}} = \frac{1}{2^B}.\end{aligned}$$

Then, for our construction, where we only check prefixes for $k$ greater than $\log n$, we can find that the sum of the expected size of $N_k$, for each such $k$ is

$$\begin{aligned}\mathbb{E}\left[\sum_{k=\frac{1}{2}\log n+1}^{\log n-1} |N_k|\right] &= \sum_{k=\frac{1}{2}\log n+1}^{\log n-1} \mathbb{E}\left[|N_k|\right] \\ &\leq \left(\frac{1}{2}\log n - 1\right)\frac{1}{2^B} \\ &= \frac{\log n - 2}{2(\log n)^2} \leq \frac{1}{2\log n}.\end{aligned}$$

Clearly, we can bound the probability of removing an element along with $x_i$ by the probability that there exists a related element to $x_i$ in the set, by previous discussion in Section 3. Then, given each bound above, assuming that the previous query was correct and that the refresh phase maintains the set distribution, we see that the probability that the returned bit $\mathsf{DB}[x_i]$ is incorrect for query step $i$ is

$$\begin{aligned}\Pr[\mathsf{DB}[x_i] \text{ is incorrect}] &\leq \frac{1}{n} + \frac{1}{\sqrt{n}\log^2 n} + \frac{1}{2\log n} \\ &\leq \frac{3}{2\log n} < \frac{1}{3},\end{aligned}$$

for $n \geq 32$. ∎

Now we introduce a new lemma that will help us prove Theorem 3.1. This lemma will bound the probability that `Add` does not work as expected. The intuition here is that, just like `Resample` can remove elements (already in the set) related to the resampled element, `Add` can add elements (not in the set) related to the added element. Below, we are bounding the number of elements *that are not* $x$ and are expected to be added to the set when we add $x$. As we explained in Section 3, this is a "failure case", since it means that our set will not be what we expect.

**Lemma B.1** (Adding related elements)**.** *For $S \sim \mathbb{D}_n$, and any $x \in \{0, \ldots, n-1\}$, the related set $S_{almost,x}$ is defined as*

$$S_{almost,x} = \{y \mid y \in \mathtt{Add}(S, x) \setminus (S \cup \{x\})\}.$$

*Then the expected size of $S_{almost,x}$ is at most $\frac{1}{2\log n}$.*

*Proof.* Note that for any $k$ less than $\log n$, there are less than $2^{\log n - k}$ strings in $\{0,1\}^{\log n}$ that share a suffix of length greater than or equal to $k$ with $x$ that do not equal $x$. The probability that a string $y$ that has a common suffix of exactly $k$ with $x$ is included in $S_{almost,x}$ is the chance that its initial $B$ bits *and* its remaining bits not shared with $x$ evaluate to 1. Namely, let us say that

$$S_{almost,x} = \bigcup N_k \,,$$

for any $k \in \mathbb{N}$ that is less than $\log n$ and more than $(1/2)\log n$. We define each $N_k$ as

$$N_k = \{y : y = \{0,1\}^k || x[k:]\} \,.$$

Since this is the same size as the $N_k$ in Case 3 of Lemma 3.1, and we are iterating over the same $k$, the expected size of $S_{almost,x}$ is

$$\mathbb{E}\left[|S_{almost,x}|\right] \leq \frac{1}{2\log n} \,.$$

■

We are now equipped with all the tools we need to prove Theorem 3.1. We restate it for completeness, and prove it below.

**Theorem 3.1** (Preliminary 2PIR+ protocol)**.** *The 2PIR+ scheme in Figure 1 is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i) $\tilde{O}(n)$ client storage $\tilde{O}(n)$ client time; (ii) $\tilde{O}(\sqrt{n})$ amortized server time and no additional server storage; (iii) $\tilde{O}(\sqrt{n})$ amortized bandwidth.*

*Proof.* We first prove privacy of the scheme, then proceed to prove correctness. The asymptotics follow by construction and were argued in Section 3.
*Privacy.* Privacy for **server**$_1$ is trivial. It only ever sees random sets generated completely independent of the queries and is not interacted with online. We present the privacy proof for **server**$_2$ below.

Privacy with respect to **server**$_2$, as per our definition, must be argued by showing there exists a stateful algorithm `Sim` that can run without knowledge of the query and be indistinguishable from an honest execution of the protocol, from the view of any PPT adversary $\mathcal{A}$ acting as **server**$_2$ for any protocol **server**$_1^*$ acting as **server**$_1$. First, we note that the execution of the protocol between **client** and **server**$_2$ is independent of **client**'s interaction with **server**$_1$. **client** generates sets and queries **server**$_1$ in the offline phase for their parity. Although this affects correctness of each query, it does not affect the message sent to **server**$_2$ at each step of the online phase, since this is decided by the sets, generated by **client**. Then, we can rewrite our security definition, equivalently, disregarding **client**'s interactions with **server**$_1$.

We want to show that for any query $q_t$ for $t \in [1, Q]$, $q_t$ leaks no information about the query index $x_t$ to **server**$_2$, or that interactions between **client** and **server**$_2$ can be simulated with no knowledge of $x_t$. To do this, we show, equivalently, that the following two experiments are computationally indistinguishable.

- **Expt**$_0$: Here, for each query index $x_t$ that **client** receives, **client** interacts with **server**$_2$ as in our PIR protocol.

- **Expt**$_1$ In this experiment, for each query index $x_t$ that **client** receives, **client** *ignores* $x_t$, samples a fresh $S \sim \mathbb{D}_n$ and sends $S$ to **server**$_2$.

First we define an intermediate experiment **Expt**$_1^*$.

- **Expt**$_1^*$ : For each query index $x_t$ that **client** receives, **client** samples $S \sim \mathbb{D}_n^{x_t}$. **client** sends $S' =$ `Resample`$(S, x_t)$ to the **server**$_2$.

By Property 1 defined in Section 3, $S'$ is computationally indistinguishable from a fresh set sampled from $\mathbb{D}_n$. Therefore, we have that **Expt**$_1^*$ and **Expt**$_1$ are indistinguishable. Next, we define another intermediate experiment **Expt**$_0^*$ to help in the proof.

- **Expt**$_0^*$: Here, for each query index $x_t$ that **client** receives, **client** interacts with **server**$_2$ as in our PIR protocol, except that on the refresh phase after each query, instead of picking a table entry $B_k$ = $(S_k, P_k)$ from our secondary sets and running $S'_k = \texttt{Add}(S_k, x_t)$, we generate a new random set $S \sim \mathbb{D}_n^{x_t}$ and replace our used set with $sk$ instead.

First, we note that by Property 2 defined in Section 3, it follows directly that **Expt**$_0$ and **Expt**$_0^*$ are computationally indistinguishable. Now, we continue to show that **Expt**$_0^*$ and **Expt**$_1^*$ are computationally indistinguishable. At the beginning of the protocol, right after the offline phase, the client has a set of $|T|$ primary sets picked at random. For the first query index, $x_1$, we

- either pick an entry $(S_j, p_j) \in T$ from these random sets where $x_1 \in S_j$;
- or, if the step above fails, we run $S_j \sim \mathbb{D}_n^{x_1}$.

Then, we send to **server**$_2$ $S'_j = \texttt{Resample}(S_j, x)$. Note that the second case is trivially equivalent to generating a random set with $x_1$ and resampling it at $x_1$. But in the first case, note that $T$ holds a sets sampled from $\mathbb{D}_n$ in order. As a matter of fact, looking at it in this way, $S_j$ is the first output in a sequence of samplings that satisfies the constraint of $x$ being in the set. Then, if we consider just the executions from 1 to $j$, this means that picking $S_j$ is equivalent to sampling from $\mathbb{D}_n^{x_1}$, by definition. Then, by Property 1, it follows that the set that the server sees in the first query is indistinguishable from a freshly sampled set.

It follows from above that for the first query, $q_1$, **Expt**$_0^*$ is indistinguishable from **Expt**$_1^*$. To show that this holds for all $q_t$ for $t \in [1, Q]$ we show, by induction, that after each query, we refresh our set table $T$ to have the same distribution as initially. Then, by the same arguments above, it will follow that every query $q_t$ in **Expt**$_0^*$ is indistinguishable from each query in **Expt**$_1^*$.

*Base Case.* Initially, our table $T$ is a set of $|T|$ random sets sampled from $\mathbb{D}_n$ independently from the queries, offline.

*Inductive Step.* After each query $q_t$, the smallest table entry $(S_j, p_j)$ such that $x_t \in S_j$ is replaced with a set sampled from $\mathbb{D}_n^{x_t}$. Since the sets are identically distributed, then it must be that the table of set keys $T$ maintains the same distribution after each query refresh.

Since our set distribution is unchanged across all queries, then using the same argument as for the first query, each query $q_t$ from **client** will be indistinguishable from a freshly sampled set to **server**$_2$. Then, we can say that **Expt**$_1^*$ is indistinguishable from **Expt**$_0^*$. This concludes our proof for experiment indistinguishability. Since we have defined a way to simulate our protocol *without* access to each $x_t$, it follows that we satisfy **server**$_2$ privacy for any PPT non-uniform adversary $\mathcal{A}$.

*Correctness.* To show correctness, we consider a slightly modified version of the scheme: After the refresh phase has used the auxiliary set $(S_j, p_j)$, the client stores $(S_j, p_j, z_j)$, where $z_j$ is the element that was added to $S_j$ as part of the protocol—for the sets that have not been used, we simply set $z_j = null$. Note that the rest of the scheme functions exactly as in Figure 1 and therefore never uses $z_j$. It follows, then, that the correctness of this modified scheme is exactly equivalent to the correctness of the scheme we presented. Note that the query phase will fail to output the correct bit only on the following four occasions:

- Case 1: $x_i$ is not in any primary set that was preprocessed.
- Case 2: The resampling does not remove $x_i$.
- Case 3: Resampling removes *more* that just $x_i$ from the set.

- Case 4: Parity is incorrect because `Add` added a related element during the refresh phase.

*Case 1:*
From the privacy proof above, we know that refreshing the sets maintains the primary set distribution. Then, we can use the same argument as in Lemma 3.1 and say that, for a query $x_i$, for all $i \in \{1, \ldots, Q\}$, we have

$$\Pr[x_i \notin \cup_{j\in[1,l]} S_j] = \left(\frac{1}{e}\right)^{\log n} \leq \frac{1}{n}\,.$$

*Case 2:*
Since `Resample` is independent from the set (just tossing random coins), we can again re-use the proof of Lemma 3.1 and say that, for any $x_i$, for all $i \in \{1, .., Q\}$, we have

$$\Pr[x_i \in \texttt{Resample}(S, x_i)] \leq \frac{1}{\sqrt{n}(\log n)^2}\,.$$

*Case 3:*
Case 3 requires us to look into our modified scheme. For the initial primary sets, the probability of removing an element related to the query is exactly the same as in Case 3 for our Lemma 3.1. However, for sets that were refreshed, we need to consider the fact that these are not freshly sampled sets, in fact, they are sets that were sampled and then had an `Add` operation performed on them. For a given query $x_i$, let $S_j$ be the first set in $T$ that contains $x_i$. Let us denote `PuncRel` to be the event that we remove *more* than just $x_i$ when resampling $S_j$ on $x_i$. We split the probability of `PuncRel` as

$$\begin{aligned}\Pr[\texttt{PuncRel}] = {} & \Pr[\texttt{PuncRel} \mid \texttt{Related}(x_i, z_j) = 1 \wedge x_i \neq z_j] \times \Pr[\texttt{Related}(x_i, z_j) = 1 \wedge x_i \neq z_j] \\ & \cup \Pr[\texttt{PuncRel} \mid \texttt{Related}(x_i, z_j) = 0 \vee x_i = z_j] \times \Pr[\texttt{Related}(x_i, z_j) = 0 \vee x_i = z_j]\,.\end{aligned}$$

The first term corresponds to the case where the added element in a previous refresh phase, $z_j$, is related to the current query element, $x_i$. Note that if $x_i$ equals $z_j$, we get the same distribution as the initial $S_j$ by Property 2 in Section 3. Then, we consider only the case where $z_j$ does not equal $x_i$. Note that we can bound

$$\Pr[\texttt{Related}(x_i, z_j) = 1 \wedge x_i \neq z_j] \leq \Pr[\texttt{Related}(S_j, z_j) = 1] \leq \frac{1}{2\log n}\,.$$

Above, we use $\texttt{Related}(S_j, z_j)$ to denote the probability that there is any related element to $z_j$ (not equal to $z_j$) in $S_j$. We can bound this event by Lemma 3.1 (see Case 3). Then, we have

$$\Pr[\texttt{PuncRel} \mid \texttt{Related}(x_i, z_j) = 1 \wedge x_i \neq z_j] \times \Pr[\texttt{Related}(x_i, z_j) = 1 \wedge x_i \neq z_j] \leq \frac{1}{2\log n}\,.$$

For the second term of our initial equation, since $\texttt{Related}(x_i, z_j)$ is 0 or $x_i$ equals $z_j$, note that our probability of resampling incorrectly is either *independent* of $z_j$, since $z_j$ does not share any prefix with $x_i$ and therefore the resampling cannot affect $z_j$ or its related elements in any way, by definition; *or* it is identical to the probability of the initial set, by Property 2. Therefore, we have that the probability of removing a related element is at most the probability of removing a related element in the original set, which by Lemma 3.1 is

$$\Pr[\texttt{PuncRel} \mid \texttt{Related}(x_i, z_j) = 0 \vee x_i = z_j] \leq \frac{1}{2\log n}.$$

And, therefore, it follows that

$$\Pr[\texttt{PuncRel} \mid \texttt{Related}(x_i, z_j) = 0 \vee x_i = z_j] \times \Pr[\texttt{Related}(x_i, z_j) = 0 \vee x_i = z_j] \leq \frac{1}{2\log n}\,.$$

Finally, we have

$$\Pr[\texttt{PuncRel}] \leq \frac{1}{2\log n} + \frac{1}{2\log n} \leq \frac{1}{\log n}\,.$$

*Case 4:*
Lastly, we have the case that query $x_i$ is incorrect because the parity $p_j$ from the set $S_j$ where we found $x_i$ is incorrect. This will only happen when we added elements related to $z_j$ when adding $z_j$ during the refresh phase. We denote this event $\texttt{AddRel}$. By Lemma B.1, we have that

$$\Pr[\texttt{AddRel}] \leq \frac{1}{2\log n}\,.$$

We can conclude that at each query $x_i$, $i \in \{1, \dots, Q\}$, assuming the previous query was correct, it follows that the probability of a query being incorrect, such that the output of the query does not equal $\mathsf{DB}[x_i]$, is:

$$\begin{aligned}\Pr[\text{incorrect query}] &\leq \frac{1}{n} + \frac{1}{\sqrt{n}\log^2 n} + \frac{1}{\log n} + \frac{1}{2\log n}\\ &\leq \frac{2}{\log n} \leq \frac{1}{3} \text{ for } n > 405.\end{aligned}$$

Because at each step we run a majority vote over $\omega(\log n)$ parallel instances, we can guarantee that, since our failure probability is less than $\frac{1}{2}$, each instance will get back the correct $\mathsf{DB}[x_i]$ with overwhelming probability. ■

## C PRS Contructions and Proofs

This section presents a construction and proof for the Adaptable PRS, as introduced and defined in Section 4. We present a construction of our Adaptable PRS in Figure 4. We also present the previous PRS construction by [SACM21] in Figure 3 for contrast. In the proof, we use a function $\texttt{time}\colon f(\cdot) \to \mathbb{N}$ that takes in a function $f(\cdot)$ and output the number of calls made in $f(\cdot)$ to any PRF function. We also prove Theorem 4.1 for our construction in Figure 4. We re-state it below for recall:

**Theorem 4.1** (Adaptable PRS construction)**.** *Assuming* $\mathsf{LWE}$*, the scheme in Figure 4 satisfies correctness, pseudorandomness with respect to* $\mathbb{D}_n$ *(Definition A.4), functionality preservation in resampling and addition (Definitions A.6 and A.8), security in resampling and addition (Definitions A.5 and A.7), and has the following complexities: (i) keys* $sk$ *and* $msk$ *have* $\tilde{O}(1)$ *size; (ii) membership testing, resampling and addition take* $\tilde{O}(1)$ *time; (iii) enumeration takes* $\tilde{O}(\sqrt{n})$ *time.*

*Proof.* We begin the proof by showing that our scheme in Figure 4 satisfies the definitions in Appendix A. We then argue efficiencies.

*Correctness and pseudorandomness with respect to* $\mathbb{D}_n$*.* Correctness follows from our construction and functionality preservation of the underlying PRF. Pseudorandomness follows from pseudorandomness of the underlying PRF (Definition A.2). Both incur a negligible probability of failure in $\lambda$, inherited from the underlying PRF.

```
Let $B = 2\log\log n$, $m = \frac{1}{2}\log n + B$.
• $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ :
    1. Let $msk \leftarrow PRF.\mathsf{Gen}(1^\lambda, \log n + B, m)$.
    2. Let $P$ be a set of $m$ arbitrary distinct strings in $\{0,1\}^{\log n + B}$ that start with a 1-bit.
    3. Let $sk = PRF.\mathsf{Puncture}(msk, P)$.
    4. output $(sk, msk)$.
• $\mathsf{EnumSet}(sk) \to S$ :
    1. Let $Z_{\frac{1}{2}\log n}$ be all bit-strings in $\ell \in \{0,1\}^{\frac{1}{2}\log n}$ such that $PRF.\mathsf{PEval}(sk, l) = 1$.
    2. Then, For $i$ in $\{\frac{1}{2}\log n + 1, \ldots, \log n\}$:
        (a) Set $Z_{i+1}$ to be any string of the form $b||\ell$ where $b \in \{0,1\}$, $\ell \in Z_i$ and PRF.$\mathsf{Eval}(sk, b||\ell) = 1$.
    3. Return $S = \{\ell : \ell \in Z_{\log n} \wedge PRF.\mathsf{PEval}(sk, 0^k||\ell) = 1\}$ for $k \in \{1, \ldots, B\}$.
• $\mathsf{InSet}(sk, x) \to b$ :
    1. Let $z = 0^B||x$.
    2. output 1 if $PRF.\mathsf{PEval}(sk, z[i:]) = 1$ for $i \in \{1, m\}$, otherwise output 0.
• $\mathsf{Resample}(msk, x) \to sk_x$ :
    1. Let $z = 0^B||x$, $Z = \{z[i:]\}$ for $i \in \{1, m\}$.
    2. Let $sk_x =$ PRF.$\mathsf{Puncture}(msk, Z)$.
    3. Return $sk_x$.
```

Figure 3: PRS Construction by Shi et al. [SACM21].

```
Let $B = 2\log\log n$, $m = \frac{1}{2}\log n + B$.
• $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ :
    1. Let $msk_1 \leftarrow$ PRF.$\mathsf{Gen}(1^\lambda, \log n + B, m)$, $msk_2 \leftarrow$ PRF.$\mathsf{Gen}(1^\lambda, \log n + B, m)$.
    2. Let $P_1, P_2$ be two sets of random $\left(\frac{1}{2}\log n + B\right)$ strings in $\{0,1\}^{\log n + B}$ that start with a 1-bit.
    3. Let $sk_1 =$ PRF.$\mathsf{Puncture}(msk_1, P_1)$, $sk_2 =$ PRF.$\mathsf{Puncture}(msk_2, P_2)$.
    4. output $(sk, msk) = ((sk_1, sk_2), (msk_1, msk_2))$.
• $\mathsf{Eval}(sk, x) \to b$ :      % internal function used to simplify algorithms
    1. Return PRF.$\mathsf{PEval}(sk[1], x) \oplus$ PRF.$\mathsf{PEval}(sk[2], x)$.
• $\mathsf{EnumSet}(sk) \to S$ :
    1. As in Figure 3 but using Eval instead of PRF.Eval.
• $\mathsf{InSet}(sk, x) \to b$ :
    1. Let $z = 0^B || x$.
    2. Output 1 if $\mathsf{Eval}(sk, z[i:]) = 1$ for $i \in \{1, m\}$, otherwise output 0.
• $\mathsf{Resample}(msk, sk, x) \to sk$ :
    1. Let $z = 0^B || x$, $Z = \{z[i:]\}$ for $i \in \{1, m\}$.
    2. Let $sk_x =$ PRF.$\mathsf{Puncture}(msk[2], Z)$.
    3. Return $(sk[1], sk_x)$.
• $\mathsf{Add}(msk, sk, x) \to sk$:
    1. Write $x \in \{0,1\}^{\log n}$ as a binary string.
    2. Define $z = 0^B || x$, $Z = \{z[i:]\}$ for $i \in \{1, m\}$.
    3. While true:      % puncture until we find $sk_x$ such that $\mathsf{Eval}(sk, x)$ equals 1.
        (a) Let $sk_x =$ PRF.$\mathsf{Puncture}(msk[1], Z)$.
        (b) If $\mathsf{InSet}((sk_x, sk[2]), x)$, output $(sk_x, sk[2])$.
```

Figure 4: Our Adaptable PRS Implementation.

*Functionality preservation in resampling and addition.* Assuming pseudorandomness and functionality preservation of the underlying PRF (Definition A.1 and Definition A.2), our PRS scheme satisfies the properties of Functionality Preservation in Addition.

For $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk, x)$, and $sk_x \leftarrow \mathsf{Punc}(msk, sk, x)$:

- From construction, $\mathsf{EnumSet}(sk_x) \subseteq \mathsf{EnumSet}(sk)$, since puncturing strings that evaluate to 1 can only reduce the size of the set (since we only resample elements in the set).
- From the point above, and construction of our $\mathsf{EnumSet}$, it follows that $\mathtt{time}(\mathsf{EnumSet}(sk)) \geq \mathtt{time}(\mathsf{EnumSet}(sk_x))$.
- By construction of our resampling operation and `Related` function, it must be that

$$y \in \mathsf{EnumSet}(sk) \setminus \mathsf{EnumSet}(sk_x) \leftrightarrow \mathtt{Related}(x, y) = 1.$$

Also, for any $n, \lambda \in \mathbf{N}$,$x \in \{0, \ldots, n-1\}$, for $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$, $sk^x \leftarrow \mathsf{Add}(msk, sk, x)$ we note that:

- By construction, $\mathsf{EnumSet}(sk) \subseteq \mathsf{EnumSet}(sk^x)$ since since we only ever make 0s into 1s.
- By the converse of same argument as Functionality Preservation in Resampling above, it follows that

$$y \in \mathsf{EnumSet}(sk^x) \setminus \mathsf{EnumSet}(sk) \leftrightarrow \mathtt{Related}(x, y) = 1.$$

Therefore, our scheme satisfies Functionality preservation in resampling and addition.
*Security in resampling.* We show that our scheme satisfies Definition A.5 below, assuming pseudorandomness and privacy w.r.t. puncturing of the underlying PRF (Definition A.1 and Definition A.3, respectively).

To aid in the proof, we define an intermediate experiment, $\mathbf{Expt}_1^*$, defined as:

- $\mathbf{Expt}_1^*$: Run $\mathsf{Gen}(\lambda, n) \rightarrow (sk, msk)$, and return $sk_x \leftarrow \mathsf{Resample}(msk, sk, x)$.

For each $sk$ output by $\mathsf{Gen}$, $sk = (sk[1], sk[2])$, two keys of $m$-puncturable PRFs. First, we show indistinguishability between $\mathbf{Expt}_1^*$ and $\mathbf{Expt}_0$:

Assume that there exists a distinguisher $D_0$ than can distinguish $\mathbf{Expt}_1^*$ and $\mathbf{Expt}_0$. Let us say that $D_0$ outputs 0 whenever it is on $\mathbf{Expt}_0$ and 1 when it is on $\mathbf{Expt}_1^*$. Then, we can construct a $D_0^*$ with access to $D_0$ that breaks the privacy w.r.t. puncturing of the PRF (as in Definition A.3) as follows, for any $x \in \{0, \ldots, n-1\}$:

```
Let $m = \frac{1}{2}\log n + B$, $L = \log n + B$, $z = 0^B||x$.
$D_0^*(m, L, z)$:
  1. Define $P_0 = \{z[i :]\}_{i \in [1,m]}$ and let $P_1, P_2$ be a set of $m$ random points of length $L$ starting with a 1-bit.
  2. Send $P_0, P_1$ to the privacy w.r.t. puncturing experiment and get back $sk_{P_b}$ and oracle access to PRF.$\mathsf{Eval}(sk_{P_b}, \cdot)$.
  3. Run PRF.$\mathsf{Gen}(1^\lambda, L, m) \rightarrow sk$, PRF.$\mathsf{Puncture}(sk, P_2) \rightarrow sk_{P_2}$.
  4. Set secret key $sk' = (sk_{P_2}, sk_{P_b})$.
  5. Return $D_0(sk')$.
```

Note that in the case where $b$ equals 0, the experiment is exactly equivalent to $D_0$'s view of $\mathbf{Expt}_0$, since $sk'$ is two random $m$-privately-puncturable PRF keys punctured and $m$ points starting with a 1-bit. Also, when $b$ is 1, $D_0$'s view is exactly equivalent to **Expt**$_1^*$, since we pass in two random $m$-privately-puncturable PRF keys, one punctured at $m$ points starting with a 1-bit, and the other at $\{z[i:]\}_{i\in[1,m]}$, with no constraints on whether $x$ was in the set before or after the puncturings. Then, since $D_0$'s view is exactly the same as its experiment, it will distinguish between both with non-negligible probability, and whatever it outputs, by construction, will be the correct guess for $b$ with non-negligible probability.

Now we proceed to show that **Expt**$_1^*$ and **Expt**$_1$ are indistinguishable, assuming pseudorandomness of the underlying PRF. Now, assume there exists a distinguisher $D_1$ that can distinguish between **Expt**$_1^*$ and **Expt**$_1$ with non-negligible probability. Then, we can construct a distinguisher $D_1^*$ that uses $D_1$ to break the pseudorandomness of the underlying PRF (as in Definition A.1) as follows, for any $x \in \{0, \dots, n-1\}$:

Let $m = \frac{1}{2}\log n + B$, $L = \log n + B$, $z = 0^B||x$.

$\underline{D_1^*(m, L, z):}$

1. Send $P = \{z[i:]\}_{i\in[1,m]}$ to the PRF pseudorandomness experiment, get back $sk_P$ and a set of $m$ bits $\{M_i\}_{i\in[1,m]}$.
2. Let $P_1$ be a set of $m$ random bit strings of length $L$ starting with a 1-bit. Run PRF.$\mathsf{Gen}(1^\lambda, L, m) \to sk$, PRF.$\mathsf{Puncture}(sk, P_1) \to sk_{P_1}$. Let $sk' = (sk_{P_1}, sk_P)$.
3. If $\forall i \in [1,m]$, PRF.$\mathsf{PEval}(sk_{P_1}, z[i:]) \oplus M_i = 1$, output $D_1(sk')$, else output a random bit.

Note that in the case $D_1$'s view in the case where the evaluations as described above all output 1 is exactly its view in distinguishing between our **Expt**$_1$ and **Expt**$_1^*$. With probability $\frac{1}{2}$, it is given a punctured key where $x$ was an element of the original set, and with probability $\frac{1}{2}$ it is given a punctured key where $x$ was sampled at random. Then, in this case, it will be able to distinguish between the two with non-negligible by assumption, and therefore distinguish between the real and random experiment for pseudorandomness of the PRF. Since the probability of having all the evaluations output 1 is non-negligible, then we break the pseudorandomness of the PRF. By contraposition, then, assuming pseudorandomness of the PRF, it must be that **Expt**$_1$ and **Expt**$_1^*$ are indistinguishable. This concludes our proof.

*Security in addition.* We now show that our scheme satisfies Definition A.7, assuming privacy w.r.t. puncturing of the underlying PRF (Definition A.3). Assume there exists a distinguisher $D$ that can distinguish between these two with non-negligible probability. Then, we can construct a distinguisher $D^*$ that breaks privacy w.r.t. puncturing of the PRF as follows, for any $x \in \{0, \dots, n-1\}$:

Let $m = \frac{1}{2}\log n + B$, $L = \log n + B$, $z = 0^B||x$.

$\underline{D^*(m, L, z):}$

1. Define $P_0 = \{z[i:]\}_{i\in[1,m]}$ and let $P_1, P_2$ be two sets of random $m$ points of length $L$ starting with a 1-bit.
2. Send $P_0, P_1$, to the privacy w.r.t. puncturing experiment and get back $sk_{P_b}$ and oracle access to PRF.$\mathsf{Eval}(sk_{P_b}, \cdot)$.
3. Run PRF.$\mathsf{Gen}(1^\lambda, L, m) \to sk$, PRF.$\mathsf{Puncture}(sk, P_2) \to sk_{P_2}$ .
4. Set our secret key $sk' = (sk_{P_b}, sk_{P_2})$.
5. **If** $\mathsf{InSet}(sk', x)$, output $D(sk')$, **else** output a random bit.

Consider the case where $x \in \mathsf{EnumSet}(sk')$:

- If $P_0$ was punctured, $D$'s view is exactly equivalent to **Expt**$_0$ in his experiment, since in $\mathsf{Add}$ we output a secret key $sk = (sk[1], sk[2])$ where $sk[1]$ is punctured at $x$, $sk[2]$ is punctured at $m$ random points starting with a 1, and $\mathsf{InSet}(sk, x)$ returns true.

- If $P_1$ was punctured, $D$'s view is exactly equivalent to **Expt**$_1$ in his experiment, by construction of $\mathsf{Gen}$, $P_1$ and $P_2$, the $sk$ outputted is equivalent to a key outputted by $\mathsf{Gen}(1^\lambda, n)$ where $\mathsf{InSet}(sk, x)$ returns true.

We conclude that, conditioned on $\mathsf{InSet}(sk_{P_b}, x)$ returning true, $D$'s view of the experiment is exactly equivalent to the experiment from our Definition A.7, and therefore it will be able to distinguish between whether $P_0$ and $P_1$ was punctured with non-negligible probability. If we fix a random $sk[2]$, the probability

$$\Pr\left[\mathsf{InSet}(sk', x) = true\right] = \frac{1}{\sqrt{n}} > \mathtt{negl}(n).$$

Then, the algorithm $D^*$ we constructed will break the privacy w.r.t. puncturing of the PRF with non-negligible probability. By contraposition, assuming privacy w.r.t. puncturing, security in addition holds.
*Efficiencies.* Efficiency for our $\mathsf{Gen}$,$\mathsf{InSet}$ and $\mathsf{Resample}$ follow from the construction and efficiencies for our underlying PRF. The two efficiencies which we will show are $\mathsf{EnumSet}$ and $\mathsf{Add}$.

Note that in $\mathsf{EnumSet}$, the step 1 takes $\tilde{O}(\sqrt{n})$ time to evaluate every string of size $\frac{\log n}{2}$, then, by pseudorandomness of the PRF, at each subsequent step we only ever keep $\sqrt{n}$ strings since half are eliminated. Since there are a logarithmic number of steps, we can say that $\mathsf{EnumSet}$ runs in probabilistic $\tilde{O}(\sqrt{n})$ time. We can also bound $\mathsf{EnumSet}$ to have a deterministic bound by incurring an additional small error for the protocol (in the case that it doesn't conclude). We show this formally in Appendix E.

For $\mathsf{Add}$, by pseudorandomness of the PRF, our construction will take probabilistic $\tilde{O}(\sqrt{n})$ time. By exploring the construction of our primitive, we reduce this to $\tilde{O}(1)$. We prove this in Corollary 1 below. Similar to the enumeration, we present a way to bound the execution deterministically without affecting overall correctness (see Appendix E). ■

In Corollary 1, we present a proof that we can run our Add function in expected $\tilde{O}(1)$ time, as alluded to in the above. Although this is not necessary for the efficiencies claimed in Theorem 4.1, it shows a significant improvement to the $\mathsf{Add}$ function; we note, however, that this speed-up only applies to the privately-puncturable PRF construction from Boneh et al. [BKM17].

**Corollary 1** (Efficient Add)**.** *Our construction can use the privately-puncturable PRF primitive to run* $\mathsf{Add}$ *run in* $\tilde{O}(1)$ *time.*

*Proof.* Note that an $m$-privately-puncturable PRF is just the xor of $m$ 1-privately-puncturable PRF keys abstracted away. By using this property, we can make $\mathsf{Add}$ more efficient, from $\tilde{O}(\sqrt{n})$ time to $\tilde{O}(1)$ by puncturing each point individually checking if it is mapped to 1, rather than attempting to get them all right at once. We replace the step 2 in the protocol with the steps as follows:

1. write $msk[1]$ as $\{msk[1]_p\}_{p\in[1,m]}$.
2. write $sk[1]$ as $\{sk[1]_p\}_{p\in[1,m]}$.
3. **For** $i$ in $[1, m]$:
   (a) $p = \mathsf{Eval}(sk, z[i:])$.
   (b) $p_{i,old} = \text{PRF}^1.\mathsf{PEval}(sk[1]_i, z[i:])$ .
   (c) $sk'_i = \text{PRF}^1.\mathsf{Puncture}(msk[1]_i, z[i:])$.
   (d) $p_{i,new} = \text{PRF}^1.\mathsf{PEval}(sk'_i, z[i:])$.
   (e) **If** $p \oplus p_{i,old} \oplus p_{i,new} \neq 1$ return to (c).
4. Let $sk' = \{sk'_i\}_{i\in[1,\frac{1}{2}\log n+B]}$ .

It follows in a straightforward manner from construction of the PRF punctured at multiple points that our algorithm for addition presented earlier and the one using this technique output indistinguishable keys. This changes $\mathsf{Add}$'s run-time from expected $\tilde{O}(\sqrt{n})$ to expected run-time $\tilde{O}(1)$. ■

## D Constructions and Proofs for Section 5

In this section, we give proof for our scheme presented in Figure 2. Then, we give a full construction and proof for the 1PIR scheme from Theorem 5.2.

### D.1 Theorem 5.1

We introduce a small lemma that will aid us in our task. This lemma, intuitively, tells us that our APRS's $\mathsf{Resample}$ is exactly equivalent to our `Resample` operation defined in Section 3 (incurring some negligible probability of failure), in other words, the new evaluations of punctured points are pseudorandom and completely independent of the previous evaluations before puncturing.

**Lemma D.1** (Randomness in resampling)**.** *In some distribution $\mathbb{D}_n$, the following two distributions are computationally indistinguishable for $m = \frac{1}{2}\log n + B$ and any $x \in \{0, \ldots, n-1\}$:*

- **Expt**$_0$: Run $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ until $\mathsf{InSet}(sk, x)$, run $sk_x \leftarrow \mathsf{Resample}(msk, sk, x)$, Return the tuple $(\mathsf{EnumSet}(sk), x \in \mathsf{EnumSet}(sk_x))$.
- **Expt**$_1$: Run $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$ until $\mathsf{InSet}(sk, x)$, sample some boolean $b$ from Bernoulli($\phi$) where $\phi = 2^{-m}$. Output the tuple $(\mathsf{EnumSet}(sk), \text{Bernoulli}(\phi))$.

Note that $x \in \mathsf{EnumSet}(sk)$ denotes a boolean denoting true or false for the expression.

*Proof.* Let us define an intermediary experiment to aid in the proof.

- **Expt**$_0^*$: Run $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$ once and let $sk^x \leftarrow \mathsf{Add}(msk, sk, x)$. Return the tuple $(\mathsf{EnumSet}(sk^x), x \in \mathsf{EnumSet}(sk))$.

Note that by our two security properties above, it follows that **Expt**$_0$ and **Expt**$_0^*$ are indistinguishable, since generating a key until finding one with $x$ is equivalent to adding, and sampling a key with $x$ and resampling $x$ is indistinguishable from sampling a fresh key. Well, by pseudorandomness of the PRF, it follows that $x \in \mathsf{EnumSet}(sk)$ for a fresh $sk$ is indistinguishable from Bernoulli($\phi$) expect with negligible probability, and by security in addition as we saw above generating a set key until we find one with $x$ is indistinguishable from generating a fresh key and adding x. Then, it follows that **Expt**$_1$ and **Expt**$_0^*$ are also indistinguishable, and this concludes our proof. ■

Now, we have all the tools we need to prove Theorem 5.1. We restate it here:

**Theorem 5.1** (Efficient 2PIR+ protocol)**.** *Assuming* $\mathsf{LWE}$*, the 2PIR+ scheme in Figure 2 is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i) $\tilde{O}(\sqrt{n})$ client storage and $\tilde{O}(\sqrt{n})$ client time; (ii) $\tilde{O}(\sqrt{n})$ amortized server time and no additional server storage; (iii) $\tilde{O}(1)$ amortized bandwidth.*

*Proof.* For brevity, we recall previous proofs. The efficiencies follow in a straightforward manner by construction in unison with Theorem 4.1. Correctness and privacy follow in the very same manner as Theorem 3.1, except we require the properties from Theorem 4.1 along with Lemma D.1 to ensure privacy and correctness, since we are dealing with short set keys, not plaintext sets. Specifically, we require Definition A.4 to ensure that our sets satisfy the same distribution as $\mathbb{D}_n$ introduced in Section 3. We require

Definition A.6, Definition A.5 and Lemma D.1 to ensure that our 'Resample' algorithm satisfied property 1 as introduced in Section 3. Note Lemma D.1 ensures for us that our privacy proof remains valid, since it says that the outputs of the resample function are *independent* of $msk$, and therefore the server's view of our distribution is entirely considitoned on the $sk$ it sees and not dependent on $msk$. We require Definition A.8 and Definition A.7 to show that our 'Add' satisfies property 2 as described in Section 3. Other than this, the proof maps exactly as Theorem 3.1, except we also incur a negligible probability of failure inherent from drawing our distribution with a PRF and not an idealized model. ■

## D.2 Theorem 5.2

As mentioned in Section 5, in order to port our two-server PIR scheme in Figure 2 to single server, we require two building blocks:

- *Batch Parity Circuit* We will use a batch parity boolean circuit $C$. Given any database of size $n$ and $l$ lists of size $m$, $C$ computes the parity of the $l$ lists in $\tilde{O}(l * m + n)$ time. A construction for such $C$ was idealized and proved in [CGHK22] [4].
- *Gate-by-gate FHE* We require the existence of a symmetric key FHE scheme ($\mathsf{Gen}$,$\mathsf{Enc}$,$\mathsf{Dec}$,$\mathsf{Eval}$) that is *gate-by-gate* (as defined in [CGHK22]), where *gate-by-gate* means $\mathsf{Eval}$ runs in time $\tilde{O}(|C|)$ for a circuit of size $|C|$. As noted in [CGHK22], this is a property of standard FHE schemes [BV11, GSW13].

To use this for our new PIR algorithm that uses sets with variable size, we require three adjustments:

1. We must modify the circuit to have an $n-$th index hardcoded to 0. The reasoning for this will be clear from item 2. This clearly does not affect correctness, efficiency or privacy of the scheme.
2. We must modify our PRS's $\mathsf{EnumSet}$ algorithm to output a list instead of a set, and pad any list of size strictly less than $\frac{\sqrt{n}}{\log n}$ with the index $n$ until the list is of size $\frac{\sqrt{n}}{\log n}$, and output $\perp$[5] if any set list is of size strictly greater than $\frac{\sqrt{n}}{\log n}$. The algorithm then incurs an expected additional $\log n$ run-time.

The reasoning for this is to use $|C|$ as a black-box, which requires a fixed sized list. With this modified PRS enumeration algorithm and circuit $C$, along with the FHE scheme introduced in 5 we finally construct our single server PIR scheme in Figure 5.

On step 3 of the offline phase, if the client *does not* find $\ell + \sqrt{n}$ sets that were evaluated correctly $p_i$ not equal to $\perp$, it just runs each online phase with a freshly sampled set key and outputs $\mathsf{DB}[x]$ equals 0. We bound the probability of this happening in the following Lemma:

**Lemma D.2** (Set Size Bound). *Let* $\texttt{LargeSet}(\cdot,\cdot)$ *be a function that takes in a list L and number w, and outputs a bit b that is 1 if the size of L is strictly greater than w, and 0 otherwise. Let* $s = \ell + Q$ *for some* $\ell, Q \in \mathbb{N}$*. Then, for* $S_1, \ldots, S_{s\cdot(\log n)^2} \sim (\mathbb{D}_n)^{s\cdot(\log n)^2}$,

$$\Pr\left[\left(\sum_{i=1}^{s\cdot(\log n)^2} \texttt{LargeSet}\left(S_i, \frac{\sqrt{n}}{\log n}\right)\right) > s\cdot(\log n)\right] < \frac{1}{\log n}.$$

[4]This circuit does not have polylog depth and therefore we require a circular-security assumption on FHE. This is well accepted be true [Gen09].

[5]We denote $\perp$ to mean an 'empty' or null value.

Run $k = \log n \log\log n$ instances of the following scheme. Let $\ell = \sqrt{n}(\log n)^3$, $s = (\ell + \sqrt{n})(\log n)^2$.

**Offline phase**

- **client** generates $s$ PRS keys $(msk_1, sk_1), \ldots, (msk_s, sk_s)$ each with $\mathsf{Gen}(1^\lambda, n)$ .
- **client** encrypts all the secret keys, FHE.$\mathsf{Enc}(sk_1), \ldots, FHE.\mathsf{Enc}(sk_s) \rightarrow (esk_1, \ldots, esk_s)$ and sends these to **server**$_1$.
- **server**$_1$ runs FHE.$\mathsf{Eval}(\mathsf{EnumSet}(esk_i))$ on each $esk_i, i \in [1, s]$ and gets back $s$ sets $S_1, \ldots, S_s$, where it will be clear which $S_i = \perp$ from the size.
- **server**$_1$ evaluates the parity of each set under FHE using $C$ and computes $ep_1, \ldots, ep_s$. For each set key $sk_i$, if $\mathsf{EnumSet}(sk_i) = \perp$, **server**$_1$ sets $ep_i = \perp$, and sends these to **client**
- **client** decrypts each $ep_i$ using FHE.$\mathsf{Dec}$ into the parity $p_i$ and stores the first $\ell$ hints where $p_i \neq \perp$ in $T = \{T_j = ((msk_j, sk_j), p_j)\}_{j \in [1,\ell]}$, and the next $\sqrt{n}$ hints that $p_i \neq \perp$ in $B = \{B_k = ((msk_k, sk_k), p_k)\}_{k \in [\ell+1, \ell+\sqrt{n}]}$.

**Online Phase:** Invoked with some index $x \in \{0, .., n-1\}$.

• Query

1. **client** finds smallest $j$ s.t. $T_j = ((msk_j, sk_j), p_j) \in T$ and $\mathsf{InSet}(sk_j, x)$. If no such $j$ is found, we let $j = |T| + 1$, run $\mathsf{Gen}(1^\lambda, n, x) \rightarrow (sk_j, msk_j)$, let p$_j$ be a uniform random bit.
2. **client** sends $sk' = \mathsf{Resample}(msk_j, sk_j, x)$ to **server**$_1$, that returns $r = \oplus_{k \in \mathsf{EnumSet(sk')}} \mathsf{DB}[k]$.
3. **client** computes $\mathsf{DB}[x] = r \oplus p_j$.
4. **client** computes $\mathsf{DB}[x]'$ to be the majority vote of the computed $\mathsf{DB}[x]$ over the $k$ instances.

• Refresh (only run if $j \leq |T|$)

1. **client** gets $B_k = ((msk_k, sk_k), p_k)$ be the first item from set $B$.
2. **client** computes $sk_k^x = \mathsf{Add}(msk_k, sk_k, x)$.
3. **client** sets $T_j = ((msk_k, sk_k^x), p_k \oplus (\mathsf{DB}[x]' \wedge \mathsf{InSet}(sk_k, x)))$, where $T_j$ was the entry consumed by the query earlier, and also sets $B = B \setminus B_k$.

Figure 5: Our 1PIR protocol using an Adaptable PRSet ($\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Resample}, \mathsf{Add}$).

We provide the proof in Appendix D.3. From this Lemma, we see that this restriction incurs an additional correctness failure of $1/\log n$ compared to our normal scheme. Note that for larger $n$, we can potentially tighten this bound and require less additional sets.

The online phase runs in exactly the same way as our scheme in Figure 2. Now, we set out to prove that our scheme satisfies Theorem 5.2. We re-state it below.

**Theorem 5.2** (Near-Optimal 1PIR protocol)**.** *Assuming* $\mathsf{LWE}$*, there exists an 1PIR scheme that is correct (per Definition 2.2) and private (per Definition 2.3) and has: (i)* $\tilde{O}(\sqrt{n})$ *client storage and* $\tilde{O}(\sqrt{n})$ *client time; (ii)* $\tilde{O}(\sqrt{n})$ *amortized server time and no additional server storage; (iii)* $\tilde{O}(1)$ *amortized bandwidth.*

*Proof.* The efficiencies follow from the efficiencies in the scheme in Figure 2, except for extra polylogarithmic factors and $\lambda$ factors incurred by using $C$ and FHE in the offline phase, along with the extra number of preprocessed sets. Neither of these affect the complexity of our scheme when examined under $\tilde{O}(\cdot)$.

Privacy for the scheme follows from the security of the FHE scheme and privacy of the scheme in Figure 2 (Theorem 5.1).

Correctness follows from the correctness proof from Theorem 5.1 and Lemma D.2, along with correctness of $C$ and the FHE scheme we use. Note that for each single copy scheme, we incur exactly the same errors as in the 2PIR scheme, with the addition of the of an extra small error probability (less than or equal to $\frac{1}{\log n}$ for any $n$ greater than 4, as shown in Lemma D.2) offline when we do not have the right amount of sets. It is clear to see that this extra factor does not take the correctness probability of the single copy scheme to be greater than $\frac{1}{2}$ and therefore the same arguments from Theorem 5.1 hold. ■

## D.3 Lemma Proof

Below we give the proof for the Lemma used in this section.

**Lemma D.2** (Set Size Bound)**.** *Let* $\mathtt{LargeSet}(\cdot,\cdot)$ *be a function that takes in a list L and number $w$, and outputs a bit b that is 1 if the size of L is strictly greater than $w$, and 0 otherwise. Let $s = \ell + Q$ for some $\ell, Q \in \mathbb{N}$. Then, for $S_1, \ldots, S_{s\cdot(\log n)^2} \sim (\mathbb{D}_n)^{s\cdot(\log n)^2}$,*

$$\Pr\left[\left(\sum_{i=1}^{s\cdot(\log n)^2} \mathtt{LargeSet}\left(S_i, \frac{\sqrt{n}}{\log n}\right)\right) > s\cdot(\log n)\right] < \frac{1}{\log n}.$$

*Proof.* From Lemma 3.1, we know that the expected size of $S$ is $\frac{\sqrt{n}}{(\log n)^2}$. Then, by a simple Markov bound:

$$\Pr\left[|S| > \frac{\sqrt{n}}{\log n}\right] < \frac{1}{\log n}.$$

Then, over $s\cdot\log n$ sets sampled independently from $\mathbb{D}_n$, $S_1, \ldots, S_{s(\log n)}$, by linearity of expectation:

$$\mathbb{E}\left[\sum_{i=1}^{s\cdot\log n} \mathtt{LargeSet}\left(S_i, \frac{\sqrt{n}}{\log n}\right)\right] < s.$$

Now, if we just apply the Markov bound again:

$$\Pr\left[\left(\sum_{i=1}^{s\cdot(\log n)^2} \mathtt{LargeSet}\left(S_i, \frac{\sqrt{n}}{\log n}\right)\right) > s\cdot(\log n)\right] < \frac{1}{\log n}.$$

■

## E Deterministic Time Bounds

We discuss below how get deterministic time bounds for our randomized algorithms used, $\mathsf{EnumSet}$ and $\mathsf{Add}$.

**EnumSet**. To get deterministic run-time for $\mathsf{EnumSet}$, we can cap the server enumeration time to be at most $6\sqrt{n}(\log n)^3$ function calls, after which it can output a random bit as the set parity. From a standard Markov argument, we see that this incurs an additional $\frac{1}{\log n}$ error per copy, which will be handled by the Chernoff bound. It is clear to see that this does not affect privacy for the servers, and only slightly affects correctness in a way that still leaves it overall greater than $\frac{1}{2}$ for any relevant $n$.

**Add**. To get deterministic run-time for $\mathsf{Add}$, we can cap the execution similarly as above, with $2(\log n)^2$ iterations. We note that in order for this change not to affect privacy of the scheme, we must take precautions to change the order of steps in our PIR scheme to ensure privacy. To make this change, we must run step 1 and 2 of the Refresh phase along with step 1 of the Query phase. If either of these fails to execute correctly, then we send to the server $(sk, \_) \leftarrow \mathsf{Gen}(1^\lambda, n)$ and **client** sets $\mathsf{DB}[x]$ to be a uniform random bit. This introduces a small probability of correctness failure but does not affect privacy since we send a random set to **server**$_2$. If we do not take this precaution of running the steps of the Refresh phase upfront, then our PIR scheme for concrete performance bounds would potentially breach privacy in the case that $\mathsf{Add}$ fails.