Single Server Private Information Retrieval with Sublinear Amortized Time and Polylogarithmic Bandwidth

ARTHUR LAZZARETTI
Yale University

CHARALAMPOS PAPAMANTHOU
Yale University

June 23, 2022

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 this index. Recently, a series of works starting with the seminal paper of Corrigan-Gibbs et al. (Eurocrypt 2020) have introduced offline-online PIR schemes with $\tilde{O}(\sqrt{n})$ (amortized) server time, $\tilde{O}(\sqrt{n})$ (amortized) bandwidth and no additional storage at the server, in both the single-server and two-server models. As a followup to this work, Shi et al. (CRYPTO 2021) further decreased the bandwidth to polylogarithmic, but only in the two-server model. In this paper we fill this gap by constructing a single-server PIR with $\tilde{O}(\sqrt{n})$ (amortized) server time and polylogarithmic bandwidth. Central to our approach is a new cryptographic primitive that we call *extended puncturable pseudorandomn set*: With an extended puncturable pseudorandom set, one can represent a random set succinctly (e.g., with a fixed-size key), and can, at the same time both add and remove elements from the set, by manipulating the key. This extension improves previously-proposed constructions that supported only removal, and could have further applications. We acknowledge our work has limitations; more work is required to bring our ideas closer to practice, due to the use of cryptographic primitives such as FHE (only in the offline phase) and LWE-based privately-puncturable PRFs. However, our protocol yields the best asymptotic complexities in single-server PIR to date and we believe it is an important step towards eventually building a practical PIR scheme.

1 Introduction

In private information retrieval (PIR), we consider the scenario where a server holds a public database DB represented as a string of n bits and a client wishes to retrieve $\mathsf{DB}[i]$ without revealing i to the server. PIR has found many applications in various systems with advanced privacy requirements such as oblivious DNS [42], instant messaging [3], advertising [4], web browsing [34] and media consumption [31]. It is well-known that PIR can be realized trivially by requesting all the bits of DB for each query, which is prohibitive for very large n, both in terms of bandwidth as well as server time. To achieve protocols with better efficiency, the community has also considered the problem under the two-server assumption [35], where the database DB is replicated in two, non-colluding servers. *For the rest of the paper we use* **1PIR** *to refer to the single-server version of the problem and* **2PIR** *to refer to the two-server version of the problem.* Clearly, 1PIR is a much more challenging problem than 2PIR, but also more useful; it is non-trivial to ensure two servers do not collude and remain highly-available in practice [38, 9].

Sublinear bandwidth 1PIR and 2PIR. The initial work by Chor et al. [18] presented a 2PIR protocol that maintained the information-theoretic security of the trivial download-all approach while reducing

Table 1: Comparison between the state-of-the-art 1PIR scheme and ours. Server time, client time and bandwidth are amortized (indicated with *). Note that all schemes have *no additional server storage*.

scheme	server time	client space	client time	bandwidth	model	assumption
[19]	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(\sqrt{n})^*$	1PIR	FHE
[41]	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(1)^*$	2PIR	LWE
Theorem 6.1	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(\sqrt{n})$	$\tilde{O}(\sqrt{n})^*$	$\tilde{O}(1)^*$	1PIR	FHE and LWE

bandwidth to $O(n^{1/3})$—the server time was, however, still linear. Later, a 1PIR protocol was introduced by Kushilevitz and Ostrovsky [35], relying on the assumption of a computationally-bounded adversary. They were able to achieve $O(n^{\epsilon})$ communication complexity for any $\epsilon > 0$, but both the client and the server were required to perform polynomial amount of work. A series of inspiring works made relevant improvements in bandwidth in both 2PIR [7, 43, 25, 24] and 1PIR [36, 28, 23, 32, 37].

Sublinear time 2PIR. Most preliminary PIR works featured a server that performs linear work. To address the problem of linear server time in 2PIR, Beimel et al. [8] proposed using preprocessing to construct a data structure on the server side that could alleviate the server's work. This approach, however, required a prohibitive amount of storage. Followup works [22, 26, 21, 31, 2] run into similar issues. Recently, this line of work was revived by Corrigan-Gibbs et al. [20] and Kogan and Corrigan-Gibbs [34] who proposed new 2PIR protocols with sublinear server time and no additional storage (inspired by the private keyword search protocol in [12]). The former work examines the idea of an offline-online PIR. Offline-online PIR is a special case of the PIR with preprocessing [8], where the preprocessed bits are stored client-side. The proposed scheme [20] allows for unlimited adaptive queries after a single expensive offline phase. The main protocol is a 2PIR protocol and involves three parties, **client**, **server**$_1$ and **server**$_2$. At a high level, it has five main steps.

Offline

1. **client** sends $\sqrt{n}$ random index sets $S_1, \ldots, S_{\sqrt{n}}$ to **server**$_1$ and **server**$_1$ returns database parities $p_1, \ldots, p_{\sqrt{n}}$, where
$$p_i = \oplus_{j \in S_i} \mathsf{DB}[j]\,.$$
2. **client** locally stores index sets and database parities.

Online (query to index i)

1. **client** finds set S_j that contains i and sends $S'_j = S_j \setminus \{i\}$ to **server**$_2$.
2. **server**$_2$ returns parity p'_j of S'_j, and **client** computes
$$\mathsf{DB}[i] = p_j \oplus p'_j\,.$$
3. **client** generates a fresh random set S^*_j that contains i, sends S^*_j to **server**$_1$, gets back its parity p^*_j, and replaces (S_j, p_j) with (S^*_j, p^*_j).

Later we will go into more detail as to how exactly we deal with efficiency issues and failure cases. Note here that while it seems that complexities of the above protocol are linear (such as client storage and bandwidth), Corrigan-Gibbs et al. [20] achieved $O(\sqrt{n})$ complexities by introducing the notion of *pseudorandom sets*: Instead of sending the sets in plaintext, the client sends a PRP key that allows the server to regenerate the sets (and also to check membership efficiently). However, note that the first step of the online phase of the Corrigan-Gibbs et al. [20] scheme requires removing an element i from the set S_j, even when the set is represented by a PRP key. Unfortunately, this cannot be done efficiently with a

PRP key, so they ended up communicating the $S_j \setminus \{i\}$ in plaintext, incurring $O(\sqrt{n}\log n)$ bandwidth online.

In a followup work, Shi et al. [41] addressed this issue. They showed how to use privately-puncturable PRFs [12, 10] to build *puncturable pseudorandom sets*, with keys that allow one to remove an element without expanding the key, thus keeping the short description of the set in the online phase. Armed with this primitive, they were able to construct a 2PIR protocol with polylogarithmic bandwidth as well as sublinear server time and $O(\sqrt{n} \cdot \mathsf{poly}(\log n))$ (amortized) complexities for all the other complexity measures.

Compiling 2PIR into 1PIR. As we discussed, the sublinear-time protocols by Corrigan-Gibbs et al. [20] and Shi et al. [41] were in the two-server model. So a natural question was whether one can achieve the same complexities in the single server model. Indeed, Corrigan-Gibbs, Henzinger and Kogan [19] showed how to port the 2PIR scheme by Corrigan-Gibbs et al. [20] into a PIR scheme with the same (amortized) complexities ($O(\sqrt{n})$ server time and $O(\sqrt{n})$ bandwidth). Their technique, inspired by [1] first transforms their initial 2PIR scheme [19] into another 2PIR scheme that *avoids communication with* **server**$_1$ *during the online phase*. To achieve this they make two crucial modifications of the high-level protocol presented before.

1. In the first step of the offline phase, instead of preprocessing $\sqrt{n}$ sets, they preprocess $\sqrt{n}+Q$ sets, where Q is the number of adaptive queries they wish to support.

2. In the third step of the online phase, instead of picking a fresh random set S_j^* and then communicating with **server**$_1$, they use an already preprocessed set from above, *avoiding communication with* ***server***$_1$ *during the online phase*.

We highlight here, that after Q queries have been performed in the above, the initially-preprocessed sets are exhausted, and the offline phase must be executed again, which does not increase the amortized complexity of the protocol.

To turn the above 2PIR protocol into a 1PIR protocol, they develop a circuit, encrypted with Fully Homomorphic Encryption [27] (FHE) that computes the $\sqrt{n}+Q$ hints in the offline phase in near-linear time without revealing information about the sets, allowing the online phase to be computed on the same server. Crucially, since all FHE preprocessing takes place offline, the online queries are just as efficient as in the two-server scenario. The costly FHE preprocessing during "downtime" is a tradeoff for fast PIR performance during the online phase.

Unfortunately, turning the Shi et al. [41] construction into an 1PIR protocol using the same trick of preprocessing an additional Q random sets and then applying homomorphic encryption runs into a fundamental issue: We need to ensure that, in Step 3 of the online phase, when we use one of the preprocessed sets, S^*, to replace the set that was just consumed to answer query i, the PRP key corresponding to S^* *must be updated to contain* i. However, this is not supported in the current construction of puncturable pseudorandom sets by Shi et al. [41]—one can only remove elements, but not add. In fact, the scheme by [41] converted to 1PIR would require linear server time per query. Our main result, described below, capitalizes on this observation.

Our result. 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. [41]. However, when we consider the 1PIR model, the Corrigan-Gibbs et al. [20] construction increases the bandwidth from polylogarithmic to $O(\sqrt{n}\log n)$. In this paper, we focus on filling this gap in 1PIR literature. We construct a **1PIR** protocol with *sublinear amortized server time and polylogarithmic amortized bandwidth for adaptive queries*. We note that our scheme is optimal up to polylogarithmic factors in every dimension except for client time, given known lower bounds [8, 20, 19]. For a detailed comparison with prior sublinear-server-no-additional-server-storage schemes, see Table 1.

Technical highlight: Puncturable pseudorandom sets with "hard puncture". As we mentioned above, the missing property that seems to be crucial for compiling Shi et al.'s 2PIR protocol into 1PIR with the same (amortized) complexities is a puncturable pseudorandom set that also supports *adding* an element into the set. Our main contribution is a formal definition and a construction of this primitive: an "extended" puncturable pseudorandom set that allows addition . We note that our new primitive might potentially have other applications.

At an abstract level, one can use our new primitive as follows. A key generation algorithm generates a succinct key sk that represents the random set. Along with algorithms for enumeration of sk and membership checking in sk, we define algorithms for puncturing element x out of sk (removing) and hard puncturing element x into sk (adding), which output a new key sk' representing the updated sets (he actual interface is a little different—see Section 4). In our construction, we allows sk to be punctured once and hard punctured once. From the construction, it is also easy to see how to extend this to any constant number of either operation. Armed with this cryptographic primitive, we first develop an improved 2PIR scheme where only one server is needed online and that achieves polylogarithmic amortized bandwidth and sublinear amortized server time. At a high level, our new 2PIR scheme works as follows (in the following we write PRS key to refer to "pseudorandom set key").

Offline

1. **client** sends $\sqrt{n} + Q$ PRS keys $sk_1, \ldots, sk_{\sqrt{n}+Q}$ to **server**$_1$ and **server**$_1$ returns database parities $p_1, \ldots, p_{\sqrt{n}+Q}$ where
$$p_i = \oplus_{j \in sk_i} \mathsf{DB}[j] \,.$$
2. **client** locally stores PRS keys and database parities.

Online (query to index i)

1. **client** finds PRS key sk_j that contains i, **punctures** i out of sk_j and sends sk'_j to **server**$_2$.
2. **server**$_2$ returns parity p'_j of sk'_j, and **client** computes
$$\mathsf{DB}[i] = p_j \oplus p'_j \,.$$
3. **client hard punctures** i into key sk_h (for the next available sk_h where $h > \sqrt{n}$) and replaces (sk_j, p_j) with $(sk_h, p_h \oplus \mathsf{DB}[i])$.

Given the above 2PIR scheme, we can convert it to a 1PIR scheme, using a batch-parity retrieval boolean circuit encrypted using FHE to run Step 1 of the offline phase. For that, we use a modified FHE circuit from [19]. We stress here the most expensive part of our protocol, running FHE encryption, is performed offline (e.g., during downtime), and therefore does not affect the online time of our 1PIR scheme. We also note that keeping a client state allows us to circumvent the recent strong lower bound result [39] that proves that we cannot have sublinear server time using logarithmic sized hints in single-server PIR.

1.1 Limitations of our scheme

Our 1PIR scheme does require the use of FHE in the same way that the Corrigan-Gibbs et al. scheme [19] did (we stress that FHE is not used during the online phase; all FHE preprocessing can be performed during "downtime"). Also, our proposed pseudorandom set with "hard puncture" is using privately-puncturable PRFs that can be constructed from the Learning with Errors [40] (LWE) assumption [12, 10]. There are currently no implementations of this primitive. It is also an open problem to construct privately-puncturable PRFs from more efficient primitives.

1.2 Paper outline

We give background on privately-puncturable PRFs and pseudorandom sets in Section 2. We give our preliminary PIR protocol that embodies the main ideas of our approach in Section 3. We introduce pseudorandom sets with the new "hard puncture" primitive in Section 4. We provide the 2-server version of our final PIR protocol in Section 5. We show how to port our 2PIR scheme into 1PIR using FHE (including how to design the FHE circuit) in Section 6. We conclude with some open problems in Section 7.

1.3 Notation

We use the $\tilde{O}(\cdot)$ notation as $O(\cdot)$ notation that hides factors related to the security parameter and poly-logarithmic factors. We denote with $negl(\cdot)$ an arbitrary negligible function.

1.4 Concurrent work

We note independently the notion of 1PIR with polylogarithmic bandwidth and sublinear server time was studied by Zhou, Lin, Tselekounis and Shi [44], whose work appeared subsequent to a submission of an earlier manuscript of this work.

2 Background

In this section, we introduce necessary cryptographic primitives that are crucial for our work: privately puncturable pseudorandom functions [10] and puncturable pseudorandom sets [20, 41].

2.1 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 $F_k(x)$. A simple construction for a puncturable PRF has a logarithmic-sized key and follows from the tree-construction of PRFs in [30]. Since this work, many other works have expanded on this concept of constraining PRFs [6, 13, 14, 33, 11, 5]. Recently, attention has been turned to a new type of puncturable PRF called a *privately* puncturable PRF, which is a puncturable PRF where the punctured key k_x, in addition to its properties as a puncturable PRF, also reveals no information about the punctured point x (by re-randomizing the output $F_k(x)$). This problem is considerably more challenging. Until recently, the only known constructions were instantiated from multilinear maps [12]. In 2017, it was proven that we can get these from standard LWE assumptions [10, 15, 17]

The implementation of privately puncturable PRFs given in [10] allows for puncturing a single point. For this section, we denote this PRF^1. However, we need a privately puncturable PRF punctured at a group of points, of size poly-logarithmic in n (where n is the size of the database), as we will see shortly. To construct an m-privately-puncturable PRF, denoted PRF^m, one can generate m PRF^1 keys and have the evaluation of PRF^m be the XOR of individual PRF^1 evaluations [10]. This allows us to puncture m independent points by puncturing each PRF^1 key at a different point. Therefore, a PRF^m exists assuming LWE. In a later section, we will see how exploring the specifics of this construction allows us to achieve better efficiency in one of our Puncturable PRSet algorithms. We henceforth denote PRF^m as simply PRF.

Definition 2.1 (Privately puncturable PRF [41, 10]). *A privately puncturable PRF scheme consists of the following four algorithms:*

- $\mathsf{Gen}(\lambda, L, m) \to sk$: *Samples a secret key* sk*, given a security parameter* λ*, a max message length* L *and number of points to be punctured,* m.
- $\mathsf{Eval}(sk, x) \to b$: *Takes in a secret key* sk *and an input* x *of bit-length* $\leq L$ *and outputs a result* $b \in \{0, 1\}$.
- $\mathsf{Puncture}(sk, P) \to sk_P$: *Takes in a secret key* sk *and a set of* m *points* P*, each of length* $\leq L$*, outputs a punctured key* sk_P.
- $\mathsf{PEval}(sk_P, x) \to b$: *Takes in a punctured key* sk_P *and an input* x *of bit-length* $\leq L$ *and outputs a result* $b \in \{0, 1\}$.

Note that the key size for the PRF based on the PRF[1] introduced by [10] is *linear* in m. Since n is poly-logarithmic in n it does not greatly affect the asymptotic efficiency of our scheme. For every PRF scheme defined by the above algorithms, we require the following three properties (one for correctness and two for security):

Definition 2.2 (Functionality preservation for a privately puncturable PRF scheme)**.** *A privately puncturable PRF scheme* ($\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval}$) *satisfies* functionality preservation *if for any polynomial* L *and* m*, for any non-uniform PPT adversary* $\mathcal{A}$ *that outputs set of* m *points* P *of length* $\leq L$ *each, there exists a negligible function* $negl()$ *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

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

Intuitively, the above definition says that puncturing the secret key on a specific set of points maintains the values of the function for all other points in the domain.

Definition 2.3 (Pseudorandomness for a privately puncturable PRF scheme)**.** *A PRF scheme satisfies* pseudorandomness *if, no PPT admissible adversary* A *can distinguish between being in either of the following experiments (we define admissible adversaries* A *as adversaries that never query the elements in its set* P *below on the original* sk*, and always picks a set* P *of size* m*).*

- $\mathit{Gen}(\lambda, L, m) \to sk$*,* $A(\lambda) \to P$*,* $\mathit{Puncture}(sk, P) \to sk_P$*,*
  $A^{\mathsf{Eval}(sk,.)}$ *is given* $(sk_P, \{\mathsf{Eval}(sk, x)\}_{x \in P})$.
- $\mathit{Gen}(\lambda, L, m) \to sk$*,* $A(\lambda) \to P$*,* $\mathit{Puncture}(sk, P) \to sk_P$*, sample uniform* $R_i \in \{0, 1\}$ *for* $i \in \{1, \ldots, m\}$*,* $A^{\mathsf{Eval}(sk,.)}$ *is given* $(sk_P, \{R_i\}_{i \in m})$.

This definition states that the values of the original function, F_k, at the punctured points, appear pseudorandom to the adversary that has access to the punctured key, as long as the adversary does not explicitly query the value of the original function at any point in set P, in which case it is trivial to distinguish.

Definition 2.4 (Privacy w.r.t. puncturing for a privately puncturable PRF)**.** *A PRF scheme satisfies* privacy with respect to puncturing *if, for any PPT admissible adversary* A*, experiments* $\mathrm{Expt}^0(\lambda, L, m)$ *and* $\mathrm{Expt}^1(\lambda, L, m)$ *are computationally indistinguishable. Experiment* $\mathrm{Expt}^b(\lambda, L, m)$ *is defined as follows:*

$\mathit{Gen}(\lambda, L, m) \to sk$*,* $A(\lambda) \to P_0, P_1$*,* $\mathit{Puncture}(sk, P_b) \to sk_{P_b}$*:* $A^{\mathit{Eval}(sk,\cdot)}(sk_{P_b}) \to b'$.

This definition states that the punctured key sk_P does not reveal anything about the set P that was punctured.

2.2 Puncturable Pseudorandom Sets

We now give a definition for puncturable pseudorandom sets (PRSets) by [41], first introduced in [20].

Definition 2.5 (Puncturable PRSet [[41, 20]). *A Puncturable PRSet scheme consists of the four following algorithms:*

- $\mathsf{Gen}(1^\lambda, n) \to (msk, sk)$*: Sample a random key to represent our puncturable random set and a puncturing master key* msk*.*
- $\mathsf{EnumSet}(sk) \to S$*: Enumerates the set from the set key.*
- $\mathsf{InSet}(sk, x) \to b$*: Outputs a bit* b *denoting whether* $x \in$ ***EnumSet***(sk)*.*
- $\mathsf{Punc}(msk, x) \to sk_x$*: Outputs a secret key that represents the set generated by* sk *punctured at element* x*.*

Note that we need both an msk and sk because our security and correctness is defined with respect to the enumeration key, while the puncturing is done with the master key. Leaking the master key breaches privacy of the PRSet. We now introduce some properties for the Puncturable PRSet scheme.

Definition 2.6 (Pseudorandomness with respect to some distribution $\mathbb{D}_n$ for Puncturable PRSets). *A Puncturable PRSet scheme* $(\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Punc})$ *satisfies* pseudorandomness w.r.t. some distribution $\mathbb{D}_n$ *if for any* λ, n*, if we take* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$*, for some distribution* $\mathbb{D}_n$*, the distribution of* $\mathsf{EnumSet}(sk)$ *is indistinguishable from a set sampled from* $\mathbb{D}_n$*.*

Definition 2.7 (Security in puncturing for Puncturable PRSets). *A Puncturable PRSet scheme* $(\mathsf{Gen}, \mathsf{EnumSet}, \mathsf{InSet}, \mathsf{Punc})$ *satisfies* security in puncturing *if the following two distributions are computationally indistinguishable for any* $x \in \{1, \ldots, n-1\}$*:*

- *Run* $\mathsf{Gen}(\lambda, n) \to (sk, msk)$*, output* sk*.*
- $\mathsf{Gen}(\lambda, n) \to (sk, msk)$ *until* $x \in \mathsf{Set}(sk)$*, output* sk_x = ***Puncture***(msk, x)*.*

2.3 Puncturable Pseudorandom Sets from Privately Puncturable PRFs

Here, we present a summary of the method that Shi et al. [41] use to generate, puncture, and evaluate membership in a PRSet using privately puncturable PRFs. We refer the reader to the paper [41] for a more in-depth analysis of their scheme. Note that their construction does not allow adding an element into the set (as we will see, our proposed PRSet scheme draws inspiration from their scheme to provide addition). In order to describe their scheme, we use a privately puncturable PRF, PRF = (Gen, Eval, Puncture, PEval) as defined in Section 2.1.

Key generation. We defer discussion of key generation to the end, for now, assume we have an sk sampled through PRF.Gen$(1^\lambda, L, m)$ as in Section 2.1. The values we need for m and L will become clear as we dive into the construction.

Set membership. From our PRF, we want to construct a pseudorandom set S. The initial attempt is to define an element $x \in \{0,1\}^{\log n}$ to be in the set *iff* for every suffix of x of size $\geq \frac{1}{2}\log n$ evaluates to 1. If we denote $x[i:]$ as the bit representation of x from the i-th bit onwards, then we can rewrite this as:

$$x \in S \Leftrightarrow PRF.\mathsf{Eval}(sk, x[i:]) = 1, \text{ for all } i = 1, \ldots, \frac{1}{2}\log n - 1\,.$$

If the PRF satisfies pseudorandomness, then it follows that, with only this requirement,

$$\Pr[x \in S] = \frac{1}{2^{\frac{1}{2}\log n}} = \frac{1}{\sqrt{n}}\,,$$

which is a suitable set size for our purposes, as we will see in the next section. Unfortunately, this approach also 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, we change the construction to add an additional constraint. Let B be some natural number. Then, let $z = 0^B||x$. We say that:

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

Note that adding this requirement 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 as well. By picking B carefully as $\lceil 2\log\log n \rceil$, it allows maintaining the set size to be $\sqrt{n}/\log^2 n$ while having a small enough dependency between elements which can be addressed.

Also, by defining membership like this, it follows that we can check set membership in $\tilde{O}(1)$ time, as was the initial requirement.

Set enumeration. Naively, set enumeration would take $n(\frac{1}{2}\log n + B)$ time, checking membership for each element in $\{0, \ldots, n-1\}$. However, what [41] point out is that, due to the light dependency introduced, we can enumerate the set in expected time $\tilde{O}(\sqrt{n})$. The key idea is to start from the

$$2^{\frac{1}{2}\log n + 1} = 2 \cdot \sqrt{n}$$

strings in $\{0,1\}^{\frac{1}{2}\log n+1}$, and keep strings s with PRF.$\mathsf{Eval}(sk, s) = 1$. We then iterate over from $m = \frac{1}{2}\log n + 2$ to $\log n$ by appending 1 and 0 to each remaining set of strings from the last step and keep only those that evaluate to 1. Since each step requires order of $\sqrt{n}$ calls to the PRF in expectation, this set enumeration runs in expected time $\tilde{O}(\sqrt{n})$. Since we use a similar enumeration algorithm, we refer the reader to our construction in Section 4 for a complete description and Appendix A for a formal proof.

Set puncturing. Up to now, all the operations we looked at could have been done with a regular PRF. Unsurprisingly, the private puncturing operation of the PRF is required only for puncturing the set. To puncture an element x from the set S, we puncture the PRF key at the $m = \frac{1}{2}\log n + B$ points that determine x's membership. By the pseudorandomness of our PRF, this will resample x's membership in S. Let $z = 0^B||x$. We run

$$sk_x \leftarrow \text{PRF.}\mathsf{Puncture}(sk, \{z[i:]\}_{i\in\{1,\ldots,m\}})\,.$$

Note that x will be in the set defined by sk_x with probability

$$1 - \frac{1}{2^{\frac{1}{2}\log n + B}} = 1 - \frac{1}{\log^2 n\sqrt{n}}\,.$$

This new key is of different format than our unpunctured key, and cannot be used in the same set membership and set enumeration functions. Also, for the security properties we described above, we want a key for a set *with* x punctured at x to be indistinguishable from a key generated through **Gen**. This cannot hold since the unpunctured key is of different format. We show how to address this issue below.

Key Generation and short set description. To deal with the problems described above, Gen outputs two keys.

1. A key generated by PRF.$\mathsf{Gen}(1^\lambda, L, m)$ as described above, denoted our *master key* msk, which will be used only for puncturing.
2. We pick a set P of m strings of size $L = \log n + B$ that start with the 1 bit and output a second key

$$sk \leftarrow \text{PRF.}\mathbf{Puncture}(msk, P)\,.$$

Note that no element in P affects set enumeration, so the set denoted by msk and sk is the same. However, outputting both allows us not only to define every PRSet function with respect to punctured keys (use PRF.PEval for set membership and enumeration), but also helps us satisfy our security in puncturing property defined above. As seen above, each set can be described fully by a secret key and punctured secret key from the underlying privately puncturable PRF, each of which have size $\tilde{O}(1)$ by definition. This allows the description of the initial set and/or punctured set succinctly. In our scheme, we take from this approach to create keys that support both puncturing and addition. We formalize this in Section 4.

Efficiency. To summarize, the PRSet scheme achieves the following efficiency requirements:

- Gen runs in time $\tilde{O}(1)$ and outputs a key of size $\tilde{O}(1)$.
- EnumSet runs in $\tilde{O}(\sqrt{n})$.
- InSet runs in $\tilde{O}(1)$.
- Punc runs is $\tilde{O}(1)$ and outputs a key of size $\tilde{O}(1)$.

Additional considerations. Clearly, we do not remove elements from the set with overwhelming probability through puncturing. Aside from that, there is also still dependency among elements, and puncturing x may also remove some other element in S. Shi et al. [41] bound these probabilities in a way that allows their final PIR scheme to work. We manage the same feat for our scheme, using a similar approach.

3 Preliminary PIR Protocol

We first design a preliminary 2PIR protocol where the client interacts with **server**$_1$ during the offline phase only. During the online phase, the client only interacts with **server**$_2$ (As we showed in the introduction, we will be able to convert this scheme later into a 1PIR protocol.) The protocol we propose here 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. In a later section, we generalize this to Q queries, and explain the nuances on the choice of Q. For now, we do not use any of the tools discussed in Section 2, because none provide us exactly what we need, but we will look at how they fit in later.

3.1 Building Blocks

In our naive protocol, we pre-compute the parity of sets sampled from a distribution $\mathbb{D}_n$ as hints that will later aid us in achieving fast, private online queries. We define below a sampling distribution $\mathbb{D}_n$ for which our preliminary scheme satisfies privacy and correctness (to a certain extent, as will be argued later) and functions that work with respect to this distribution $\mathbb{D}_n$.

3.1.1 Sampling Distribution

We define the distribution $\mathbb{D}_n$ in a similar way as in [41]. Note that the distribution is defined very carefully in order to achieve fast enumeration, fast membership testing, and short descriptions for sets of size close to $\sqrt{n}$. These three properties are not achievable in unison when sampling from naive distributions, as was seen in [19, 41]. In order to allow for these three properties, we introduce light dependency between elements in the set. For the definition of the distribution, we construct a set S as follows. sampled from $\mathbb{D}_n$ by sampling from an Random Oracle $\texttt{RO}$ for each set. We note that our final scheme does *not* use a random oracle and this is for expositional purposes only. Let $m = \frac{1}{2}\log n, B = 2\log\log n$. Set $z = 0^B||x$, then, for all $x \in \{0, \ldots, n-1\}$, we have:

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

Note that if we use the technique in 2.3, we can enumerate this set in $\tilde{O}(\sqrt{n})$ time. Now, we also define $\mathbb{D}_n^x$ to be a distribution where you sample a set S from $\mathbb{D}_n$ *until* it outputs a set where $x \in S$.

3.1.2 Functions with respect to $\mathbb{D}_n$

We define two functions we will need for the preliminary scheme. These are defined with respect to the distribution $\mathbb{D}_n$. For these functions, we first need to define what it means for two elements to be *related*.

Definition 3.1. *We define the function* $\texttt{Related}(x, y) \to b$ *to take in two elements* $x, y \in \{0, \ldots, n-1\}$ *and return a bit* $b \in \{0, 1\}$ *where* $b = 1$ *if and only if* x *and* y *share a suffix of length* $> \frac{1}{2}\log n$ *in their binary representation.*

For example $\texttt{Related}(1000001, 1100001) = 1$ while $\texttt{Related}(1000001, 1101111) = 0$. Equipped with this, we define two functions with respect to $\mathbb{D}_n$.

- $\texttt{Resample}(S, x) \to S$: Define $z = 0^B||x$. 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.
- $\texttt{Add}(S, x) \to S'$: To evaluate this function, we require access to the $\texttt{RO}(\cdot)$ used to evaluate the set. Then, this function re-samples the set from $\texttt{RO}(\cdot)$, except it replace any evaluation to a suffix of x with the output 1. Note that this will add x to the set, but also potentially add some related elements.

  Informally, on a correct execution, $\texttt{Add}(S, x)$ outputs $S \cup \{x\}$. On some executions it will output a set $S' \supset S \cup \{x\}$.

Note that, by construction, $\texttt{Resample}$ and $\texttt{Add}$ can only affect the input element x and elements *related* to x. We define below a function and set we will need bound this.

Also note that by construction of our scheme, for $S \sim \mathbb{D}_n$:

- $\forall x \in S$, $S' = \texttt{Resample}(S, x) \subseteq S$.
- For any $x \in \{0, \ldots, n-1\}$, $S \subseteq S' = \texttt{Add}(S, x)$.

We will look at how to bound the occurrence of adding or removing related elements in the correctness proof.

3.2 Preliminary Protocol

As we mentioned, we have two *non-colluding* servers, denoted **server**$_1$ and **server**$_2$. We show our protocol in Figure 1. Our protocol supports $\sqrt{n}$ queries and runs in amortized $\tilde{O}(\sqrt{n})$ server time. It uses the distribution $\mathbb{D}_n$ and functions defined with respect to it above. We will argue correctness later in the section, but as a precursor, we will say that this preliminary scheme is not correct with overwhelming probability. In fact, for any $n > 100$, its correctness probability is more than 0.9 for the first query and progressively smaller for the following queries. Discussion on how to address this is present at the end of this section.

Preliminary Protocol

Parameters: Set $\ell = \sqrt{n}\log^3 n$.
Offline phase: Preprocessing

1. **client** samples $\ell + \sqrt{n}$ sets $S_1, \ldots, S_{\ell+\sqrt{n}} \sim (\mathbb{D}_n)^{\ell+\sqrt{n}}$.
2. **client** sends sets $S_1, \ldots, S_{\ell+\sqrt{n}}$ to **server**$_1$ and gets back 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

$$\mathsf{T} = \{T_j = (S_j, p_j)\}_{j\in[1,\ell]}$$

and

$$\mathsf{B} = \{B_k = (S_k, p_k)\}_{k\in[\ell+1,\ell+\sqrt{n}]}\,.$$

Online Phase: called with some index $x \in \{0, \ldots, n-1\}$
• Query

1. **client** finds $T_j = (S_j, p_j)$ in 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:
   - $S_j \sim \mathbb{D}_n^x$.
   - p_j is a uniform bit.
2. **client** sends $S' = \mathtt{Resample}(S_j, x)$ to **server**$_2$, and **server**$_2$ returns $r = \bigoplus_{k\in S'} \mathsf{DB}[k]$.
3. **client** computes $\mathsf{DB}[x] = r \oplus p_j$.

• Refresh (locally on **client**—executes only when $j \leq |\mathsf{T}|$)

1. Let $B_1 = (S_k, p_k)$ be the first item from set B.
2. Let $S_k^* = \mathtt{Add}(S_k, x)$, and $p_k^* = p_k \oplus (\mathsf{DB}[x] \wedge (x \notin S_k))$.
3. Set $T_j = (S_k^*, p_k^*)$, where T_j was the entry consumed by the query earlier, and removes B_1 from B.

Figure 1: Our preliminary 2PIR protocol.

3.3 Criticality of addition

What allows our protocol to run for $\sqrt{n}$ queries as a single-server protocol after the initial offline phase is the fact that we use information from previous queries along with our preprocessed information to add the element that was queried to a fresh random set and update its parity. Without addition, it would require storing order of n sets to find a suitable replacement each primary set used when considering $\sqrt{n}$ adaptive queries. Addition allows us to achieve the same functionality with only $\sqrt{n}$ extra sets. Looking ahead, one of our main contributions is developing a pseudorandom set scheme with short description that allows for addition *without* increasing the size of the key. This is what helps us achieve polylogarithmic communication online on 1PIR.

3.4 Analysis

We now proceed to look at the efficiency, privacy and correctness of this scheme, in this order. We defer formal proofs for the real scheme, and instead argue intuition on each that will be helpful when diving into the proofs. To analyze the scheme we need the follwing Lemma:

Lemma 3.1 (Set Size)**.** *The expected size of a set* $S \sim \mathbb{D}_n$ *is* $\mathbb{E}[|S|] \leq \frac{\sqrt{n}}{\log^2 n}$.

We defer the proof to Appendix A.

3.4.1 Efficiency

We now analyze the various complexities of our preliminary protocol.

Server time. For the server time, we consider the work done by the servers jointly. $\textbf{server}_1$ does expected work of $\tilde{O}(n)$ offline and no work online. $\textbf{server}_2$ does no work offline and expected $O(\sqrt{n})$ work per query. These follow from enumeration time and Lemma 3.1. Over $\sqrt{n}$ queries, this means that $\textbf{server}_2$ will perform $O(n)$ work in expectation. So our total joint work for both servers is $\tilde{O}(n)$. Over $\sqrt{n}$ queries, our amortized server time is $\tilde{O}(\sqrt{n})$.

Server storage. Concerning server storage, both servers have to store DB, of size n bits, but incur no extra storage.

Client time. Offline, **client** performs $\tilde{O}(n)$ work to generate $\ell + \sqrt{n}$ sets of size $\sqrt{n}$ and send them to the server plaintext. Online, the **client** has to find the set with the corresponding i, which takes expected time $O(\sqrt{n})$ (the expected number of elements to check) for each of our ℓ primary sets, totalling $\tilde{O}(n)$. On Section 3.5 we examine techniques that will allow this efficiency to be worst-case and not amortized and not require sorting offline. Our final scheme presents $\tilde{O}(\sqrt{n})$ client time.

Client Storage. As it is, our **client** has to store $\ell + \sqrt{n}$ sets. Each set has expected $\sqrt{n}/\log^2 n$. This is proved in the following lemma: elements (see Lemma 3.1) of size $\log n$ each, and their corresponding parities, which means that the client storage for our preliminary scheme is $\tilde{O}(n)$. We improve this by using a PRF F and storing the relevant instead of the set, but this requires fine tuning to work in sync correctly with `Add` and Resample. What we do in our final scheme is similar to this, and we improve this to $\tilde{O}(\sqrt{n})$.

Bandwidth. From Lemma 3.1, we get the following. Offline, we incur an expected $\tilde{O}(n)$ bandwidth to send all sets to the $\textbf{server}_1$, and then receive back the corresponding parities. Online, we incur an additional expected $O(\sqrt{n})$ bandwidth to send the set S'_j to $\textbf{server}_2$. Over $\sqrt{n}$ queries, we achieve $\tilde{O}(\sqrt{n})$ amortized bandwidth. In our final scheme, we reduce this to $\tilde{O}(1)$ amortized bandwidth through our new primitive.

3.4.2 Privacy

We now examine privacy with respect to each server. Proving privacy relies mainly on two properties.

1. For any $S \sim \mathbb{D}_n^x$, $S_x = \texttt{Resample}(S, x)$, $S' \sim \mathbb{D}_n$, S_x and S' are computationally indistinguishable.

2. For any $S \sim \mathbb{D}_n$, $S^x = \texttt{Add}(S, x)$, $S' \sim \mathbb{D}_n^x$, S^x and S' are computationally indistinguishable.

These two follow in a straightforward manner from the construction of $\mathbb{D}_n$. For our real scheme, we need to prove a harder version of these two properties, presented in Section 4.1, since we communicate a set key to the server.

*Privacy with respect to **server**$_1$.* Note that **server**$_1$ only ever sees $\ell + \sqrt{n}$ independent random sets. The sets reveal nothing about future queries to it, and it is never accessed online, so its privacy is easy to see. Any deviation from the protocol by **server**$_1$ can only affect correctness of the scheme, and not privacy.

*Privacy with respect to **server**$_2$.* The privacy argument for **server**$_2$ is more delicate. We split the argument in two parts:

- For the initial query, we pick a set with the index we want to query, x, from our primary sets. Given a table T of randomly sampled sets, by construction of T (since j is the smallest set with x, S_j can be viewed as generating sets until we find one with x) and *property 1* above, $S' = \texttt{Resample}(S_j, x)$ is indistinguishable from a random set from $\mathbb{D}_n$, and therefore, the query reveals nothing about the query index x to **server**$_2$.
- In order to argue that every query also reveals nothing, we claim that the 'refresh' step maintains the distribution of 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. Then, after each query, we must replace S_j with an identically distributed set. Conditioned on our choice of x, by property 2, S_j and $\texttt{Add}(S_k, x)$ are identically distributed. Then, the swap maintains the distribution of our primary sets where every set is unknown to the server. By induction on our primary set distribution, each query is equivalent to the first, and none of them reveal anything about the queried indexes to **server**$_2$.

3.4.3 Correctness

We denote our PIR protocol to be **correct** when, for every query x_i for $i \in [1, \sqrt{n}]$, **client** outputs $\mathsf{DB}[x_i]$, the x_i-th bit of the database, with probability $(1 - negl(\lambda))$ for any n polynomially bounded in λ.

The correctness calculated is based on the assumption that the servers are running the protocol correctly. We do not guarantee correctness in the case of deviation by either server, since **client** has no mechanism to verify responses.

Offline Phase Correctness. Assuming the server follows the protocol, the offline phase will be correct by construction. The client generates the random sets, gets their parity and saves the sets and parities to use online.

Online Phase Correctness. For each query in the online phase, we have four failure points. We defer proofs of the lemmas presented here to Appendix A.

- *Failure case 1*: Note that for a query to x, if we cannot find a in index j in T such that $x \in S_j$ for $T_j = (S_j, p_j)$ (Step 1), and send to the server a random set instead, the algorithm *fails*. As is clear to see, the parity computed on the following steps will be incorrect. We can bound this as follows:

  Lemma 3.2 (Primary set coverage)**.** *Let $S_1, \ldots, S_\ell \sim (\mathbb{D}_n)^\ell$. For any $x \in \{0, \ldots, n-1\}$, the probability of not finding x over these sets,* $\Pr[x \notin \cup_{i \in [1,l]} S_i] \leq \frac{1}{n}$.

- *Failure case 2*: The second failure case is on step two of the query phase, when our $\texttt{Resample}$ function does not remove x. From construction of our sets and resample, we can see that this happens with probability $1/\sqrt{n}\log^2 n$. Note that this failure case is crucial for privacy, since it guarantees that **server**$_2$ only sees sets that look random.

- *Failure case 3*: The third failure case happens when we remove an element other than x within `Resample`, thus yielding incorrect final parity. We bound this as follows:

  Lemma 3.3 (Related elements in S)**.** *For any $x \in \{0, \ldots, n-1\}$, for $S \sim \mathbb{D}_n^x$, the expected number of elements in S related to x, $\mathbb{E}[|\text{Related}(S, x)|] \leq \frac{1}{2\log n}$.*

- *Failure case 4*: Analogous to the previous failure case, we denote failure case 4 as the case when we add an element other than x within `Add`. We bound this as follows:

  Lemma 3.4 (Related elements almost in S)**.** *For a randomly sampled set $S \sim \mathbb{D}_n$, and any $x \in \{0, \ldots, n-1\}$, we define a related set $S_{almost,x}$, for $y \in \{0, \ldots, n-1\}$:*

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

  Then, we can bound the expected size of the set $S_{almost,x}$ as: $\mathbb{E}[|S_{almost,x}|] \leq \frac{1}{2\log n}$.

We first look at the correctness for the *first query* only. Since *Failure case 4* does not affect the first query, we ignore it for now. Then, from a simple probability bound, we can say that our total probability of failure for the first query is (the probability that either event happens) for $n > 100$:

$$\frac{1}{n} + \frac{1}{\sqrt{n}\log^2 n} + \frac{1}{2\log n} < 0.1$$

Therefore can say that the probability of our scheme being correct for the first query is greater than $\frac{1}{10}$ for any relevant n, for the first query. Note that our error propagates to the next query and makes it more likely to fail. Over $\sqrt{n}$ queries, the correctness rapidly decreases. We will look at how to deal with propagation of error and transform this into a scheme with overwhelming correctness probability across all queries below.

3.5 Improvements

3.5.1 Correctness Improvements

As is done in [41] and in classical literature for randomized algorithms, in order to increase the correctness of our scheme, we can treat our output as a random bit with $p > \frac{1}{2}$ chance of correctness, and drive our correctness up by running k parallel instances our protocol and overwriting the output bit $\mathsf{DB}[x]$ with the majority of $\mathsf{DB}[x]$ over these k instances in step 2 of our query stage. Let us denote C to be the event, where, over k instances of our preliminary PIR scheme, more than $\frac{k}{2}$ of them output the correct bit. Using a standard lower-tail Chernoff bound, we have that, with p being the probability of $\mathsf{DB}[x]$ being correct, that the probability of C is

$$\Pr[C] \geq 1 - e^{-\frac{1}{2p}k(p-\frac{1}{2})^2}.$$

To get $\Pr[C] \geq 1 - negl(n)$ for some negligible function in our security parameter λ, we need for $e^{\frac{1}{2p}k(p-\frac{1}{2})^2} \geq negl(\lambda)$. If we pick $k = \log(\lambda)\log(\log(\lambda))$, then we get

$$\Pr[C] \geq 1 - e^{-\frac{1}{2p}(\log\lambda\log(\log(\lambda)))(p-\frac{1}{2})^2} \geq 1 - (\log(\lambda))^{(-\log\lambda)(\frac{1}{2p}(p-\frac{1}{2})^2)}.$$

For $p > \frac{1}{2}$, the latter term on the exponent is a constant strictly greater than 0. Therefore, since $(\log(\lambda))^{-\log\lambda}$ is negligible in λ, we conclude that by introducing repetition poly-logarithmic in λ, and our preliminary PIR scheme, we are able to construct a PIR scheme that is correct with overwhelming probability. In fact, we picked a concrete k to showcase, but this would work for any k in $\omega(\log\lambda)$. We will use the same technique in our full-fledged PIR scheme.

We stress that it is crucial to take the majority vote after each *query* step and run refresh with the $\mathsf{DB}[x]'$ that is correct with overwhelming probability. From our correctness analysis above, this will work for the first query, but does not take care of a propagating error from our *Failure case 4*. We bound correctness of our scheme for all queries by bounding the probability that an entry from T_j has incorrect parity. We defer this argument for our real scheme, but by bounding failures to be less than $\frac{1}{2}$, we use k parallel instances with the same argument as above to achieve overwhelming correctness probability in through $\sqrt{n}$ queries.

After $\sqrt{n}$ queries, we re-run the scheme (since we run out of backup sets).

3.5.2 Efficiency Improvements

As we saw in Section 2.3, there exists a scheme [41] based on privately puncturable PRFs [12] that allows for resampling an element in the set through private puncturing within the key, allowing us to send this 'punctured key' of size $O(\mathbf{poly}\log n)$ to the server to enumerate the set, while guaranteeing that no information about the punctured element is revealed by the punctured key. We note, however, that their solution cannot be applied to our scenario, since there is no good way to add an element to the set and then resample another element. Even if we do implement `Add` using `Resample` as we do here, their PRSet implementation only allows for one operation. This begs the question on whether we can achieve efficient pseudorandom sets that can support both addition and deletion, while maintaining a short description. Throughout the rest of the work, we go through how to define efficient pseudorandom sets that support both resampling and adding, our implementation, and the consequences of such construction (Section 4). From it, we formally define an efficient offline-online PIR scheme that communicates with only one scheme online (Section 5), and then show how to port this scheme to a single-server offline-online scheme with fast online queries (Section 6). We defer most proofs to the Appendix.

4 Extended Puncturable Pseudorandom Sets

In this section, we introduce our main primitive that is required for achieving our result, an *extended puncturable pseudorandom set*. We first give the definition that can accommodate both additions and removals of elements and then we present our construction that satisfies this definition.

4.1 Definitions

Definition 4.1 (Extended Puncturable PRSet)**.** *An Extended Puncturable PRSet is defined by five algorithms.*

- $\mathsf{Gen}(\lambda, n) \to (sk, msk)$*: Sample a master secret key* msk *and a secret key* sk*.*
- $\mathsf{EnumSet}(sk) \to S$*: Outputs a set* S *given a secret key* sk*.*
- $\mathsf{InSet}(sk, x) \to b$*: Outputs a bit* b *denoting whether* $x \in \mathsf{EnumSet}(sk)$*.*
- $\mathsf{Punc}(msk, sk, x) \to sk_x$*: Outputs a secret key* sk_x *punctured at* x*.*
- $\mathsf{HardPunc}(msk, sk, x) \to sk^x$ *Outputs a secret key* sk^x *hard punctured at* x*.*

Note that our interface differs from the original construction by [41] not only in the extra functionality, but also in the overlapping puncture function. The new functionality requires our puncture and hard puncture operation to be dependent both on our master key and enumeration key. We will see this in more detail below. Our new Puncturable PRSet with addition must satisfy the following security properties. Note here that Punc and HardPunc have equivalent functionalities to the `Resample` and `Add` presented in Section 3, respectively. We now define properties required for the extended puncturable PRSet.

Definition 4.2 (Pseudorandomness with respect to some distribution $\mathbb{D}_n$ for extended puncturable PRSets)**.** *An Extended Puncturable PRSet scheme* (Gen, EnumSet, InSet, Punc, HardPunc) *satisfies* pseudorandomness w.r.t. some distribution $\mathbb{D}_n$ *if we take* $\textsf{Gen}(\lambda, n) \to (sk, msk)$*. Then, For some distribution* $\mathbb{D}_n$*, the distribution of* $\textsf{EnumSet}(sk)$ *is computationally indistinguishable from a set sampled at random from* $\mathbb{D}_n$ *for any* λ, n*.*

Definition 4.3 (Security in Puncturing for Extended Puncturable PRSets)**.** *An Extended Puncturable PRSet scheme* (Gen, EnumSet, InSet, Punc, HardPunc) *satisfies* Security in Puncturing *if the following two distributions are computationally indistinguishable for any* $x \in \{0, \ldots, n-1\}$*:*

- ***Expt***$_0$*: Run* $\textsf{Gen}(\lambda, n) \to (sk, msk)$*. Output* sk*.*
- ***Expt***$_1$*:* $\textsf{Gen}(\lambda, n) \to (sk, msk)$ *until* $x \in \textsf{EnumSet}(sk)$*,* $sk_x = \textsf{Puncture}(msk, sk, x)$*, output* sk_x*.*

This is a straightforward property that says that a set picked to have a specific element x, punctured at x, is identically distributed and indistinguishable from a set sampled at random.

Definition 4.4. *An Extended Puncturable PRSet scheme* (Gen, EnumSet, InSet, Punc, HardPunc) *satisfies* Security in Hard Puncturing *if the following two distributions to be computationally indistinguishable for any* $x \in \{0, \ldots, n-1\}$*:*

- ***Expt***$_0$*: Run* $\textsf{Gen}(\lambda, n) \to (sk, msk)$ *and return* $sk^x \leftarrow \textsf{HardPunc}(msk, x)$*.*
- ***Expt***$_1$*: Run* $\textsf{Gen}(\lambda, n) \to (sk, msk)$ *until* $x \in \textsf{EnumSet}(sk)$*. Return* sk*.*

This experiment guarantees that generating fresh keys until we find one where x belongs to the set is equivalent to generating one fresh key and hard puncturing x into it. We will see later that this will be a key property in allowing us to only communicate with one server during the online phase.

4.2 PRSets with One Addition or Removal

From the construction in Section 2.3, we examine how to make modifications to allow for *addition*. One straightforward way to approach this is to attempt to puncture an element *into* the set. All things equal, since the puncture operation is randomized, we can run puncture the same key many times until it outputs a punctured key where we have added the element to the set, like we did using the `Add` function in the preliminary protocol. Indeed, this approach allows us to add an element to a set with expected $\tilde{O}(\sqrt{n})$ operations. However, since our puncturing is only defined for m points, and we cannot 're-puncture' punctured keys (we only puncture from the master key), this precludes removal. We also would like to run this faster, since $\tilde{O}(\sqrt{n})$ is considerably more costly than the puncturing operation, which can be done in $\tilde{O}(1)$.

4.3 Allowing Both Addition and Removal

From our insight on addition, we attempt to include both addition and removal by instantiating our PRSet with a privately puncturable PRF punctured at $2m$ points. This, however, introduces several problems. Given our interface with privately puncturable PRFs, we cannot partition our puncturing (puncture some elements now, others later) without overcomplicating our PRSet scheme and introducing many security liabilities. We would have to add and remove at the same time, which is not only inviable in many use-cases, but also introduces its own set of complications to the PRSet scheme. For example, if the element we want to add and the element we want to remove are related, then we would be puncturing less than $2m$ points, and it is also not clear how to prioritize overlapping points.

Seemingly at a standstill, we borrow from the idea in [10], that defines an m-privately-puncturable PRF from the xor of m 1-privately-puncturable PRFs. In our case, we decide to introduce an *extra* m-privately-puncturable PRF key to our set. In this way, we can hard puncture from one key, and puncture from the other, independently. We look at how this works in an simplified model below, and then delve into our actual implementation. We note that from this idea, it would also not be a large leap to define a PRSet that supports any fixed number of additions or removals, by adding extra PRF keys.

4.4 Sampling Distribution

To satisfy *Pseudorandomness w.r.t. some distribution*, we must show that our scheme models some distribution. Let us model a distribution $\mathbb{D}'_n$ for sets generated by this new idea. We now use a PRF F instead of the $\texttt{RO}(\cdot)$ to sample a set. For each set, we sample *two* uniform random keys, k_0 and k_1 Then, we generate the sets as in Section 3, except we replace each evaluation of a string using $\texttt{RO}(\cdot)$ by an evaluation using $F_{k_0}(\cdot) \oplus F_{k_1}(\cdot)$. Note that by the randomness preserving property of the xor operation, this distribution is a permutation of the one $\mathbb{D}_n$ we defined in Section 3 (incurring some negligible probability of failure in the security parameter), and so the set and element distribution we showed for $\mathbb{D}_n$ is identical for $\mathbb{D}'_n$. Also, set enumeration will run in the same asymptotic time, with an additional constant factor of 2 to account for the requirement of two evaluations per point. We then re-define every operation as follows:

Set Membership. Let our set membership for each element $x \in \{0, \dots, n-1\}$ be determined as was discussed earlier with slight adjustment. Take $z = 0^B||x$. Then,

$$x \in S \Leftrightarrow \text{ for all } i = 1, \dots, \frac{1}{2}\log n + B \text{ it is } F_{k_0}(z[i:]) \oplus F_{k_1}(z[i:]) = 1\,.$$

Resampling. In this model, when running `Resample(S,x)`, we resample the output of the relevant suffixes only for k_1. Again, by the randomness preserving property of xor, this is exactly equivalent to the `Resample` presented in Section 3.

Adding. In this model, when running `Add(S,x)`, we resample the output of the relevant suffixes only for F_{k_0} until we find S' s.t. $x \in S'$. Again, by the randomness preserving property of xor, this is exactly equivalent to the `Add` presented in Section 3.

Distribution. From the arguments above, we note that $\mathbb{D}'_n$ and $\mathbb{D}_n$ model the same distribution, except for the number of calls needed to enumerate the set, which are related by a factor of 2.

4.5 Functionality preservation

As in the scheme by [41], we require our scheme to satisfy some notion of correctness with respect to puncturing. More specifically, in order to prove correctness of the PIR scheme, it is very helpful to properly bound what can and cannot go wrong in puncturing. For our construction, we have the added constraint of requiring an analog correctness notion for hard puncturing. We need to ensure that the algorithms satisfy very specific properties. For the definitions below, we define $\texttt{time}(f(\cdot))$ as a function that maps a function to its runtime. We also use a modified version of the function defined in [41], $\texttt{Related}(x, y) : \{0,1\}^{\log n} \times \{0,1\}^{\log n} \to \{0,1\}$, which tells us whether two bit-strings x, y are related or not for any $x, y \in \{0,1\}^{\log n}$.

Definition 4.5 (Functionality Preservation in Puncturing)**.** *We say that an extended puncturable PRSet (Gen,EnumSet,InSet,Punc,HardPunc) satisfies functionality preservation in puncturing if, given some* `Related` *function,* $\forall \lambda, n$, *the following hold with overwhelming probability in* λ*, for any* $x \in \{0, \dots, n-1\}$*:*

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

- $\mathsf{EnumSet}(sk_x) \subseteq \mathsf{EnumSet}(sk)$.
- $\texttt{time}(\mathsf{EnumSet}(sk)) \geq \texttt{time}(\mathsf{EnumSet}(sk_x))$.
- $y \in \mathsf{EnumSet}(sk) \setminus \mathsf{EnumSet}(sk_x) \leftrightarrow \texttt{Related}(x, y) = 1$.

Definition 4.6 (Functionality Preservation in Hard Puncturing)**.** *We say that an extended puncturable PRSet (Gen,EnumSet,InSet,Punc,HardPunc) satisfies functionality preservation in hard puncturing if, given some* $\texttt{Related}$ *function,* $\forall \lambda, n$*, the following hold with overwhelming probability in* λ *for any* $x \in \{0, \ldots, n-1\}$*:*

For $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$*,* $sk^x \leftarrow \mathsf{HardPunc}(msk, sk, x)$*:*

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

Note that we do not bound the runtime of $\mathsf{EnumSet}(sk^x)$ to anything. This is because we already need to prove the runtime for a set sampled with x, and as we saw from the security properties above, it must be that such set is indistinguishable from a set with x hard punctured.

4.6 Extended Puncturable PRSets Implementation

As was observed in [41], it is a difficult task to balance, within a pseudorandom set: element independence, efficient set enumeration and membership testing, short descriptions and puncturing.

Our implementation idea for PRSets with addition, inspired by [41] and [10], comes from the keen observation that puncturing an element x into the set happens with the same probability as generating a set with x, and that these two tasks seemed analogous. The main challenge was balancing a construction that was simple enough to have clear security properties, but also refined enough that would allow our PRSets to manage both an addition *and* a removal.

We define a function Eval as a private interface to be used within our algorithms only, to simplify their description. Also note that we assume a random PRF.$\mathsf{Puncture}$ that generates randomness instead of using randomness defined in PRF.Gen. This is shown in [10]. Then, from the scheme presented in Figure 2, we derive the following theorem, reliant on the LWE assumption:

Theorem 4.1 (Extended PRSet)**.** *The scheme in Figure 2 satisfies correctness, the security definitions defined in Section 4.1, the functionality preservation definitions in Section 4.5 and has the following complexities:*

- $\tilde{O}(1)$ *sk and msk size.*
- $\tilde{O}(1)$ *membership-testing time.*
- $\tilde{O}(\sqrt{n})$ *enumeration time.*
- $\tilde{O}(1)$ *puncturing time.*
- $\tilde{O}(\sqrt{n})$ *hard puncturing time.*

We provide the proofs in Appendix A.

Our PRSet Construction

Let $B = 2\log\log n$, $m = \frac{1}{2}\log n + B$.

• $\mathsf{Gen}(1^\lambda, n) \to (sk, msk)$:

1. Let $msk_1 \leftarrow \text{PRF}.\mathsf{Gen}(1^\lambda, \log n + B, m)$, $msk_2 \leftarrow \text{PRF}.\mathsf{Gen}(1^\lambda, \log n + B, m)$.
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 $sk_1 = \text{PRF}.\mathsf{Puncture}(msk_1, P_1)$, $sk_2 = \text{PRF}.\mathsf{Puncture}(msk_2, P_2)$.
4. output $(sk, msk) = ((sk_1, sk_2), (msk_1, msk_2))$.

• $\mathsf{Eval}(sk, x)^* \to b$:

1. Return $\text{PRF}.\mathsf{PEval}(sk[1], x) \oplus \text{PRF}.\mathsf{PEval}(sk[2], x)$.

• $\mathsf{EnumSet}(sk)$:

1. Let $B_{\frac{1}{2}\log n}$ be all bit-strings in $l \in \{0,1\}^{\frac{1}{2}\log n}$ such that $\mathsf{Eval}(sk, l) = 1$.
2. Then, For i in $\{\frac{1}{2}\log n + 1, \ldots, \log n\}$:
   (a) $B_i = \{1||l : l \in B_{i-1} \wedge \mathsf{Eval}(sk, 1||l) = 1\} \cup \{0||l : l \in B_{i-1} \wedge \mathsf{Eval}(sk, 1||l) = 1\}$.
3. Output $B_{\log n + B} = \{l : l \in B_{\log n} \wedge \mathsf{Eval}(sk, 0^k||l) = 1\}$ for $k \in \{1, \ldots, B\}$.

• $\mathsf{InSet}(sk, x) \to b$:

1. Let $z = 0^B||x$.
2. output 1 if $\mathsf{Eval}(sk, z[i:]) = 1$ for $i \in \{1, m\}$, otherwise output 0.

• $\mathsf{Punc}(msk, sk, x) \to sk$:

1. Let $z = 0^B||x$, $Z = \{z[i:]\}$ for $i \in \{1, m\}$.
2. Let $sk_x = \text{PRF}.\mathsf{Puncture}(msk[2], Z)$.
3. Return $(sk[1], sk_x)$.

• $\mathsf{HardPunc}(msk, sk, x) \to sk^{**}$:

1. write $x \in \{0,1\}^{\log n}$ as a binary string, and define $z = 0^B||x$, $Z = \{z[i:]\}$ for $i \in \{1, m\}$.
2. While `true`:
   (a) Let $sk_x = \text{PRF}.\mathsf{Puncture}(msk[1], Z)$.
   (b) If $\mathsf{InSet}((sk_x, sk[2]), x)$, output $(sk_x, sk[2])$.

Figure 2: Our Extended PRSet Implementation.

4.6.1 Efficiency Improvements

Note that we can make HardPunc more efficient, from $\tilde{O}(\sqrt{n})$ time to $\tilde{O}(1)$ by breaking up $msk[1]$ and $sk[1]$ into the corresponding m PRF^1 keys that define them, where PRF^1 is a privately puncturable PRF instantiated the same way as our interface but with $m = 1$. We then each point individually and check if it is what we want, 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 $sk' = \{sk'_i\}_{i\in[1,\frac{1}{2}\log n+B]}$.

Note that it follows in a straightforward manner from construction of the PRF punctured at multiple points that our algorithm for hard puncture presented earlier and the one using this technique output indistinguishable keys. This shifts HardPunc's run-time from expected $\tilde{O}(\sqrt{n})$ to expected run-time $\tilde{O}(1)$. For the rest of the paper, we use the former algorithm for simplicity, and because it does not change the asymptotic runtime of our client, but using a Markov Inequality we can bound the runtime of this new algorithm to $\tilde{O}(1)$ with high probability.

4.6.2 Concrete Bounds

We prove concrete bounds for our scheme to run correctly with high probability in the Appendix (specifically Appendix A and Appendix B.3). For our PIR Scheme, we can use those to get concrete performance bounds while maintaining overwhelming correctness.

5 Our New PIR Protocol

First, we define the security definition of our new two-server PIR protocol that communicates with only one server online.

5.1 Definitions

We define our PIR as a collection of three stateful algorithms denoted as **server**$_1$ and **server**$_2$, and **client**. All three players receive the parameters 1^λ and n before the start. We define the interactions below:

- **Offline**: **server**$_1$ and **server**$_2$ receive the same n-bit database DB and **client** receives nothing. **client** sends one message to **server**$_1$, **server**$_1$ replies with one message.
- **Online**: For each query, **client** receives an index $x \in \{0, \ldots, n-1\}$. **client** sends one message to **server**$_2$, which responds with one message. In the end, **client** outputs a bit b.

Correctness

We define correctness with respect to the output of the Q queries made by the PIR scheme. We say that our PIR scheme is correct iff $\exists$ a negligible function $negl()$ s.t for any database $\mathsf{DB} \in \{0,1\}^n$, and any sequence of queries $x_1, x_2, \ldots, x_Q \in \{0, 1, \ldots, n-1\}$, an honest execution of our PIR scheme with DB and queries $x_1, x_2, \ldots, x_Q$ returns $\mathsf{DB}[x_1]$, $\mathsf{DB}[x_2]$, ..., $\mathsf{DB}[x_Q]$ with probability $1 - negl(\lambda)$, for any n, Q polynomially bounded in λ, where $\mathsf{DB}[i]$ is the i-th bit of DB.

Privacy

Below we define the privacy for each server in our PIR scheme.

server$_1$ *privacy:* **server**$_1$ only interacts with **client** offline, where it answers parities for sets *independent* of the future queries. Then, it is trivial to see that no adversarial algorithm for **server**$_1$ can learn anything about the queries. We therefore do not consider its privacy in our model, only correctness.

server$_2$ *privacy:* We say that we satisfy **server**$_2$ privacy holds if $\exists$ a p.p.t., stateful algorithm `Sim`, such that for any algorithm **server**$_1$*, no non-uniform p.p.t. adversary `A` can distinguish between the following experiments with non-negligible probability:

- **Expt**$_0$: **client** interacts with `A` who acts as the **server**$_2$ and deviates arbitrarily from the protocol, and **server**$_1$* who acts as the **server**$_1$. At every step t, `A` chooses the query index x_t, and **client** is invoked with input x_t as its query, $x_t \in \{0, \ldots, n-1\}$
- **Expt**$_1$: `Sim` interacts with `A` who acts as the **server**$_2$ and deviates arbitrarily from the protocol, and **server**$_1$* who acts as the **server**$_1$. At every step t, `A` chooses the query index x_t, and `Sim` is invoked with no knowledge of x_t.

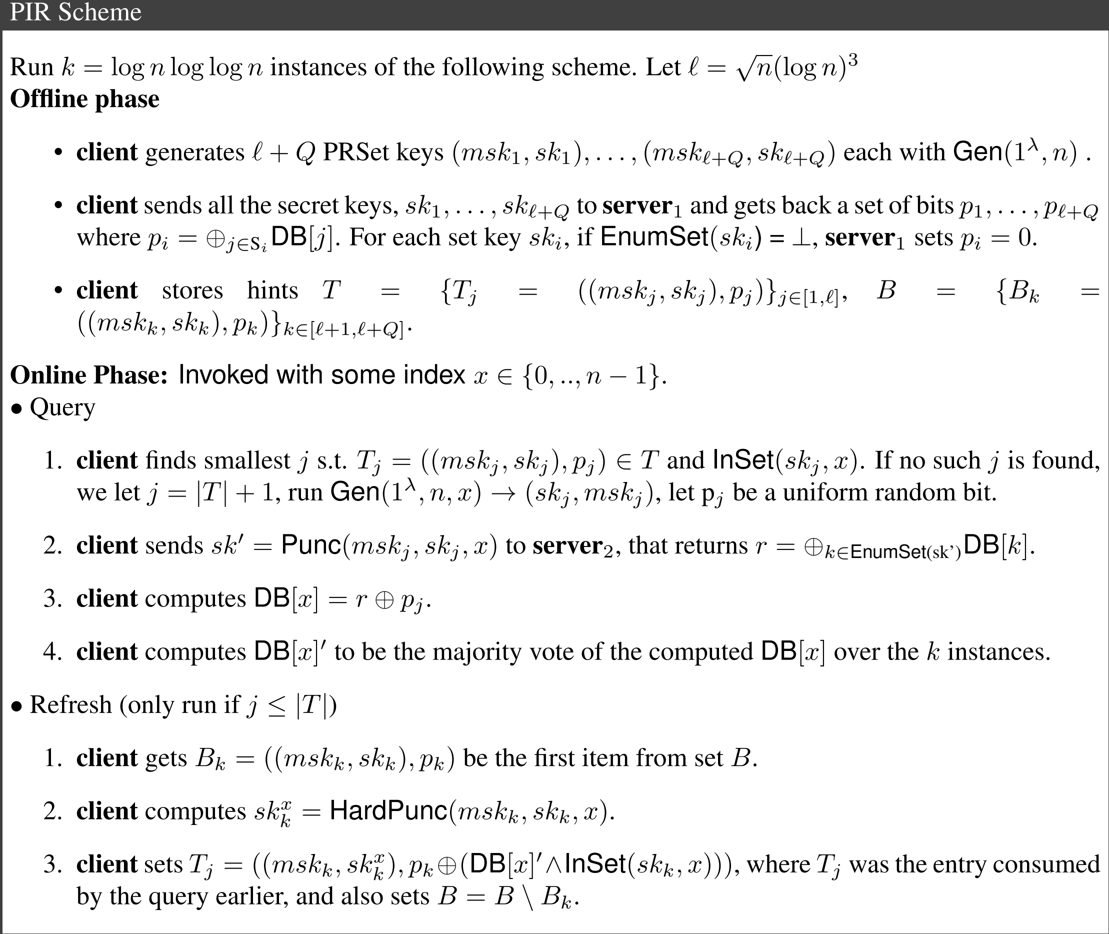
Intuitively all this says that the queries we make to **server**$_2$ will look random to **server**$_2$. It also means that although privacy will hold regardless of deviation from the servers (assuming no collusion). This is not the case for correctness.

5.2 Our Protocol

Now, we introduce our two-server PIR protocol with a single server online phase with near-optimal communication that allows for Q queries offline and has amortized sublinear server time. In a later section we will see how to transform this to a single-server PIR protocol with near-optimal communication and amortized sublinear server time. As in [41], we require $\omega(\log \lambda)$ parallel instances of the following scheme and a majority vote for correctness. The online phase supports up to Q queries, after which we have to re-run the offline phase. Note that our final client bit $\mathsf{DB}[x]$ is the most common $\mathsf{DB}[x]$ over our parallel repetitions. For concrete efficiency, we pick our number of parallel repetitions to be equal to $\log n \log \log n$. As we saw earlier, this choice gives us overwhelming correctness probability of a correct majority vote, by the Chernoff bound.

5.2.1 Requiring Only One Server Online

We re-iterate the idea that requiring only one server online is a big step forward in itself. In any practical setting, this allows one server's availability to be very scarce, meaning that in adversarial settings where we have only one reliable server, and the other one is being compromised or has to go through maintainance, the functionality of our PIR scheme is unaffected. This scheme will also be indispensable for the construction of our final single-server PIR scheme.


PIR Scheme

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

Offline phase

- **client** generates $\ell + Q$ PRSet keys $(msk_1, sk_1), \ldots, (msk_{\ell+Q}, sk_{\ell+Q})$ each with $\mathsf{Gen}(1^\lambda, n)$.
- **client** sends all the secret keys, $sk_1, \ldots, sk_{\ell+Q}$ to **server**$_1$ and gets back a set of bits $p_1, \ldots, p_{\ell+Q}$ where $p_i = \oplus_{j \in S_i} \mathsf{DB}[j]$. For each set key sk_i, if $\mathsf{EnumSet}(sk_i) = \perp$, **server**$_1$ sets $p_i = 0$.
- **client** stores hints $T = \{T_j = ((msk_j, sk_j), p_j)\}_{j \in [1,\ell]}$, $B = \{B_k = ((msk_k, sk_k), p_k)\}_{k \in [\ell+1, \ell+Q]}$.

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 $sk' = \mathsf{Punc}(msk_j, sk_j, x)$ to **server**$_2$, 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 $sk_k^x = \mathsf{HardPunc}(msk_k, sk_k, x)$.
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 3: Our 2PIR protocol.

5.2.2 Construction

Our scheme is constructed in Figure 3.

For simplicity, we define $\mathsf{Gen}(1^\lambda, n, x)$ to run normal Gen until it outputs an sk where $\mathsf{InSet}(sk, x) = 1$. For optimized performance bounds, we can substitute this for generating one set and hard puncturing x. With our faster hard puncture algorithm, this is much faster than generating sets until finding one with x, and with the probability of failure introduced being the probability we fail to hard puncture.

With our scheme, we prove the following theorem, reliant on the LWE assumption:

Theorem 5.1 (PIR Protocol)**.** *The scheme in Figure 3 satisfies privacy and correctness as defined in Section 5.1 and has the following complexities:*

- $\tilde{O}(\sqrt{n} + Q)$ *client storage and no additional server storage.*
- $\tilde{O}(n/Q + \sqrt{n})$ *amortized server time.*
- $\tilde{O}(\sqrt{n})$ *client time.*

- $\tilde{O}(\sqrt{n}/Q)$ *amortized bandwidth.*

We defer the proof to Appendix B, but it will follow similarly to what was sketched in Section 3, except more involved.

5.2.3 Analysis

Note that our scheme here is very similar to the naive scheme presented in Section 3, with pinpoint modifications to account for either issues that we had discussed initially, or possible caveats from our PRSet definition and implementation. We see that for each copy of our scheme, our offline phase runs in $\tilde{O}(n)$ server time and $\tilde{O}(\sqrt{n})$, and each query runs in $\tilde{O}(\sqrt{n})$ client and server time, with $\tilde{O}(1)$ bandwidth. After Q queries, we have to re-run the offline phase, but as we will see shortly, an ideal choice of Q will amortize our server time to be sublinear any number of queries.

Our sources of error stem from when we cannot find a set with x for a query, when $\mathsf{EnumSet}$ takes too long to run, or when Punc or $\mathsf{HardPunc}$ do not work as expected. We will bound these formally in the next section, but for now notice that the hardest error to deal with will be when $\mathsf{HardPunc}$ does not run as expected. This is because it happens *after* the majority vote for $\mathsf{DB}[x]$ and therefore potentially propagates this error to a future query with non-negligible probability. We can bound the error for each query by then bounding the probability, for each $q \in Q$, that the parity for T_j we find was incorrect. We deal with this formally when we prove correctness for our PIR Scheme in Appendix B.

5.2.4 Choice of Q

We define Q as the number of queries that our scheme can make after running the offline phase one. It can also be seen as the number of queries allowed by our scheme before we have to re-run the offline phase. Note that for any $k > 0$, a choice of Q in the order of $n^{1/k}$ allows us to achieve sublinear amortized server time over Q queries. However, note that for any $k \leq 1$, the client storage is of order $\tilde{O}(n)$. In fact, any $k < 2$ increases the asymptotic client storage required by the initial primary sets. Also note that for any $k > 2$, we reduce the space required (although not asymptotically, since that is lower bounded by the number of primary sets we need), but also reduce the number of queries we can amortize the offline phase cost across. In fact, from this, it is clear to see that the optimal choice for balancing amortized server time and client space is $k = 2$, with $Q = \sqrt{n}\mathbf{poly}\log n$, although specific implementations might benefit from picking a different order of Q within the range discussed.

5.2.5 Choice of B

Our choice of B follows directly from the choice of B in [41], and allows us to achieve sets of reasonable size and light dependency, while maintaining fast enumeration. We do not discuss it further here.

5.2.6 Deterministic and optimized bounds for our scheme

To get a protocol with deterministic performance requires bounding the number of tries for both $\mathsf{Gen}(1^\lambda, n, x)$ and $\mathsf{HardPunc}$. When doing this, 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 (more discussion in Appendix B.3), but clearly 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 $\mathsf{HardPunc}$ fails.

The optimized bounds discussed in [41] translate in a straight-forward manner to our PRSet and PIR scheme, so we do not discuss it further here.

6 Porting into a Single-Server Scheme

We now porting our two-server PIR scheme that communicates with only one server online to a single-server PIR scheme.

6.1 Requirements

In order to port our two-server PIR scheme in Figure 3 to single server, we require two building blocks:

6.1.1 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 [19].

6.1.2 Gate-by-gate FHE

We require the existence of a symmetric key FHE scheme (Gen,Enc,Dec,Eval) that is *gate-by-gate* (as defined in [19]), where *gate-by-gate* means Eval runs in time $\tilde{O}(|C|)$ for a circuit of size $|C|$. As noted in [19], this is a property of standard FHE schemes [16, 29].

6.2 Near-optimal Single Server PIR from Fully Homomorphic Encryption

Assuming we have such a boolean batch parity circuit C and FHE scheme defined as above, for correctness exactly as defined in Section 5.1 and single-server privacy as defined for $\mathbf{server}_2$ in Section 5.1, our Theorem 5.1 implies the following theorem:

Theorem 6.1 (Single-Server PIR Protocol)**.** *There exists a single-server PIR protocol satisfies privacy and correctness as defined in Section 6.2 and has the following complexities:*

- $\tilde{O}(\sqrt{n} + Q)$ *client storage and no additional server storage.*
- $\tilde{O}(n/Q + \sqrt{n})$ *amortized server time.*
- $\tilde{O}(\sqrt{n})$ *client time.*
- $\tilde{O}(\sqrt{n}/Q)$ *amortized bandwidth.*

We defer the scheme and proofs to Appendix C, both which follow closely from the scheme and proof for Figure 3 and the primitives discussed in this section.

7 Conclusion

We have presented the *first* single-server PIR scheme for adaptive queries with *sublinear amortized server time and polylogarithmic bandwidth*. We re-iterate that given known lower bounds, Theorem 6.1 has optimal complexities up to polylogarithmic factors in every dimension except for client time.

7.1 Open Problems

We provide some future directions we consider interesting, given our results:

- Finding an implementation of puncturable random sets compatible with Linearly Homomorphic Encryption, or a compiler to port schemes to single server from weaker assumptions.
- Implementing a concretely efficient privately puncturable PRF from LWE.
- Constructing privately puncturable PRFs from core cryptographic assumptions such as one-way functions.
- Constructing a scheme that maintains our complexities but uses $\tilde{O}(1)$ client time, or proving that it is not possible.

References

[1] W. Aiello, S. Bhatt, R. Ostrovsky, and S. Rajagopalan. Fast Verification of Any Remote Procedure Call: Short Witness-Indistinguishable One-Round Proofs for NP. In *ICALP*, 2000.

[2] S. Angel, H. Chen, K. Laine, and S. Setty. PIR with compressed queries and amortized query processing. Technical Report 1142, 2017.

[3] S. Angel and S. Setty. Unobservable Communication over Fully Untrusted Infrastructure. pages 551–569, 2016.

[4] M. Backes, A. Kate, M. Maffei, and K. 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.

[5] A. Banerjee and C. Peikert. New and Improved Key-Homomorphic Pseudorandom Functions. Technical Report 074, 2014.

[6] A. Banerjee, C. Peikert, and A. Rosen. Pseudorandom Functions and Lattices. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, Lecture Notes in Computer Science, pages 719–737, Berlin, Heidelberg, 2012. Springer.

[7] A. Beimel and Y. Ishai. Information-Theoretic Private Information Retrieval: A Unified Construction. In G. Goos, J. Hartmanis, J. van Leeuwen, F. Orejas, P. G. Spirakis, and J. 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.

[8] A. Beimel, Y. Ishai, and T. Malkin. Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. page 19, 2000.

[9] J. Bell, K. A. Bonawitz, A. Gascón, T. Lepoint, and M. Raykova. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. Technical Report 704, 2020.

[10] D. Boneh, S. Kim, and H. Montgomery. Private Puncturable PRFs From Standard Lattice Assumptions. Technical Report 100, 2017.

[11] D. Boneh, K. Lewi, H. Montgomery, and A. Raghunathan. Key Homomorphic PRFs and Their Applications. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, Lecture Notes in Computer Science, pages 410–428, Berlin, Heidelberg, 2013. Springer.

[12] D. Boneh, K. Lewi, and D. J. Wu. Constraining Pseudorandom Functions Privately. Technical Report 1167, 2015.

[13] D. Boneh and B. Waters. Constrained Pseudorandom Functions and Their Applications. Technical Report 352, 2013.

[14] E. Boyle, S. Goldwasser, and I. Ivan. Functional Signatures and Pseudorandom Functions. Technical Report 401, 2013.

[15] Z. Brakerski, R. Tsabary, V. Vaikuntanathan, and H. Wee. Private Constrained PRFs (and More) from LWE. Technical Report 795, 2017.

[16] Z. Brakerski and V. Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and P. 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.

[17] R. Canetti and Y. Chen. Constraint-hiding Constrained PRFs for NC1 from LWE. Technical Report 143, 2017.

[18] B. Chor, O. Goldreich, and E. Kushilevitz. Private Information Retrieval. page 18, 1997.

[19] H. Corrigan-Gibbs, A. Henzinger, and D. Kogan. Single-Server Private Information Retrieval with Sublinear Amortized Time. Technical Report 081, 2022.

[20] H. Corrigan-Gibbs and D. Kogan. Private Information Retrieval with Sublinear Online Time. Technical Report 1075, 2019.

[21] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. 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.

[22] G. Di Crescenzo, Y. Ishai, and R. Ostrovsky. Universal Service-Providers for Private Information Retrieval. *Journal of Cryptology*, 14(1):37–74, Jan. 2001.

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

[24] Z. Dvir and S. Gopi. 2-Server PIR with Subpolynomial Communication. *Journal of the ACM*, 63(4):1–15, Nov. 2016.

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

[26] S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: Round-Optimal Oblivious RAM with Applications to Searchable Encryption. Technical Report 1010, 2015.

[27] C. 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.

[28] C. Gentry and Z. Ramzan. Single-Database Private Information Retrieval with Constant Communication Rate. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Automata, Languages and Programming*, volume 3580, pages 803–815. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.

[29] C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, Lecture Notes in Computer Science, pages 75–92, Berlin, Heidelberg, 2013. Springer.

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

[31] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, M. Walfish, and U. Austin. Scalable and private media consumption with Popcorn. page 18, 2016.

[32] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal Rate Private Information Retrieval from Homomorphic Encryption. *Proceedings on Privacy Enhancing Technologies*, 2015(2):222–243, June 2015.

[33] A. Kiayias, S. Papadopoulos, N. Triandopoulos, and T. Zacharias. Delegatable Pseudorandom Functions and Applications. Technical Report 379, 2013.

[34] D. Kogan and H. Corrigan-Gibbs. Private Blocklist Lookups with Checklist. pages 875–892, 2021.

[35] 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.

[36] H. Lipmaa. An Oblivious Transfer Protocol with Log-Squared Communication. page 15, 2005.

[37] H. Lipmaa and K. Pavlyk. A Simpler Rate-Optimal CPIR Protocol. Technical Report 722, 2017.

[38] M. H. Mughees, H. Chen, and L. Ren. OnionPIR: Response Efficient Single-Server PIR. Technical Report 1081, 2021.

[39] G. Persiano and K. Yeo. Limits of Preprocessing for Single-Server PIR. Technical Report 235, 2022.

[40] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. Acm*, 2009.

[41] E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. Technical Report 1592, 2020.

[42] S. Singanamalla, S. Chunhapanya, J. Hoyland, M. Vavruša, T. Verma, P. Wu, M. Fayed, K. Heimerl, N. Sullivan, and C. Wood. Oblivious DNS over HTTPS (ODoH): A Practical Privacy Enhancement to DNS. *Proceedings on Privacy Enhancing Technologies*, 2021(4):575–592, Oct. 2021.

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

[44] M. Zhou, W.-K. Lin, Y. Tselekounis, and E. Shi. Optimal Single-Server Private Information Retrieval. page 38.

A Proofs for New Puncturable PRSet with Addition

A.1 Scheme Analysis

Let $\mathbb{D}_n$ be defined as in Section 4.4. Proofs for the distribution of our PRSets and runtime of our algorithms. We use the notation $\mathbb{D}_n^x$ to represent a distribution where we sample from $\mathbb{D}_n$ until we find a set that contains element x. In the proofs, we consider $\mathbb{D}_n$ sampled from a random oracle, although it is easy to see that sampling from a PRF would differ only in a negligible function in the security parameter of such function.

A.1.1 Distribution of our PRSet

Proofs for Lemmas presented in Section 3.

Lemma 3.1 (Set Size). *The expected size of a set $S \sim \mathbb{D}_n$ is $\mathbb{E}[|S|] \leq \frac{\sqrt{n}}{\log^2 n}$.*

Proof. We have that

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

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

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

■

Let $\ell = \sqrt{n}(\log n)^3$

Lemma 3.2 (Primary set coverage). *Let $S_1, \ldots, S_\ell \sim (\mathbb{D}_n)^\ell$. For any $x \in \{0, \ldots, n-1\}$, the probability of not finding x over these sets,* $\Pr[x \notin \cup_{i\in[1,l]} S_i] \leq \frac{1}{n}$.

Proof. From Lemma 3.1, know x is included in each set S_i with probability $\frac{1}{\sqrt{n}(\log n)^2}$. Then,

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

■

A.1.2 Related Elements

Lemma 3.3 (Related elements in S)**.** *For any $x \in \{0, \ldots, n-1\}$, for $S \sim \mathbb{D}_n^x$, the expected number of elements in S related to x,* $\mathbb{E}[|\text{Related}(S, x)|] \leq \frac{1}{2 \log n}$.

Proof. Note that for any $k < \log n$, there are exactly $2^{\log n - k} - 1 \leq 2^{\log n - k}$ strings in $\{0, 1\}^{\log n}$ that share a suffix of length $\geq k$ with x that do not equal x. Note that since x is in the set, for any k, the probability that a string y that has a common suffix of 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 $k < \log n$, $y = \{0, 1\}^k || x[k:]$

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

Let N_k be the expected number of strings in the set that share a longest common suffix with k 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

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

Then, for our construction, where we only check prefixes for $k > \log n$, we can find that the expected number of related elements in the set can be the sum over such k that fit such criteria

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

■

Lemma 3.4 (Related elements almost in S)**.** *For a randomly sampled set $S \sim \mathbb{D}_n$, and any $x \in \{0, \ldots, n-1\}$, we define a related set $S_{almost,x}$, for $y \in \{0, \ldots, n-1\}$:*

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

Then, we can bound the expected size of the set $S_{almost,x}$ as: $\mathbb{E}[|S_{almost,x}|] \leq \frac{1}{2 \log n}$.

Proof. Note that for any $k < \log n$, there are $\leq 2^{\log n - k}$ strings in $\{0, 1\}^{\log n}$ that share a suffix of length $\geq k$ 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, for any $k < \log n$, $y = \{0, 1\}^k || x[k:]$. Then, since this is exactly equivalent to

the number of related elements to x for $x \in S$,the rest of the proof follows exactly as in Lemma 3.3, and we can say that the expected number of elements in $S_{almost,x}$,

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

■

A.1.3 Deterministic Time Bounds

We define a function $\texttt{time}: f(\cdot) \to \mathbb{N}$ to take in a function $f(\cdot)$ and output the number of calls made in $f(\cdot)$ to any PRF function.

Lemma A.1 (EnumSet runtime)**.** *Fix some $x \in \{0, \ldots, n-1\}$, $(msk, sk) \to$ Gen$(1^\lambda, n)$, $sk^x \leftarrow$ HardPunc(msk, sk, x), then*

$$\Pr\left[\texttt{time}(\textsf{\textit{EnumSet}}(sk^x)) > 6\sqrt{n}(\log n)^3\right] \leq \frac{1}{\log n} + negl(\lambda)$$

Proof. Our proof follows very much like described in the distribution analysis in Section 2.2, except that we require *two* function calls per poin evaluation. Also note that in this case we are bounding the enumeration time for sets sampled from a distribution *with x*.

From our argument earlier, if we start from strings of length $l = \frac{1}{2}\log n + 1$, check which strings evaluate to 1, and then iteratively build up to strings of size $\log n$ by appending 1 and 0 to remaining strings, we will have $\sqrt{n}$ remaining strings, in expectation, after every step (in expectation, only 1 from the two new string attempted remains, by pseudorandomness of the PRF used for sampling). Conditioned on having x, this shifts slightly. For every length l, we can upper bound the number of related elements with x of length l in the set to be $\leq \log n$ (see Lemma 3.3). This increases our expected set size for each lengh l to at most $\sqrt{n}\log n$.

Following our argument from before, this means that to enumerate a set with x, we will have to evaluate $2\sqrt{n}\log n$ strings per step and then at most $2\log n$ strings for each remaining string at the end so putting it all together, we will make at most $2(2\sqrt{n}\log n\frac{1}{2}\log n + 2\log\log n\sqrt{n}\log n) \leq 6\sqrt{n}(\log n)^2$ PRF calls in expectation. Now, in order to bound this, we use a Markov Inequality, and from this expectation, we have that

$$\Pr\left[\texttt{time}(\textsf{EnumSet}(sk^x)) > 6\sqrt{n}(\log n)^3\right] \leq \frac{1}{\log n} + negl(\lambda)$$

■

Lemma A.2 (HardPunc runtime)**.** *Let $(msk, sk) \leftarrow$ Gen$(1^\lambda, n)$ for some λ, n. Then, for any $x \in \{0, \ldots, n-1\}$, for our PRSet construction,*

$$\Pr[\texttt{time}(\textsf{HardPunc}(msk, sk, x)) > 2\sqrt{n}(\log n)^2] \leq \frac{1}{\log n} + negl(\lambda)$$

Proof. Note that the expected runtime of HardPunc is

$$2\sqrt{n}\log\log n \leq 2\sqrt{n}\log n$$

Then, by a simple Markov Inequality, we have that

$$\Pr[\texttt{time}(\textsf{HardPunc}(msk, sk, x)) > 2\sqrt{n}(\log n)^2] \leq \frac{1}{\log n} + negl(\lambda)$$

■

A.2 Proofs

A.2.1 Correctness Proofs

Lemma A.3. *Our PRSet scheme satisfies correctness. Assuming pseudorandomness of the underlying PRF, our scheme also satisfies pseudorandomness.*

Proof. Correctness follows from our construction and functionality preservation of the underlying PRF. Pseudorandomness follows from pseudorandomness of the underlying PRF. Both incur a negligible probability of failure in λ, inherited from the underlying PRF. ■

Functionality Preservation in Puncturing. *Assuming pseudorandomness and functionality preservation of the underlying PRF, our PRSet scheme satisfies the properties of Functionality Preservation in Puncturing. (Definition 4.5).*

Proof. 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 (we will only every puncture a prf evaluation of 1 to 0, not the other way around).
- From the point above, and construction of our $\mathsf{EnumSet}$, it follows that $\texttt{time}(\mathsf{EnumSet}(sk)) \geq \texttt{time}(\mathsf{EnumSet}(sk_x))$.
- By construction of our puncturing operation and `Related` function, it must be that

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

.

■

Functionality Preservation in Hard Puncturing. *Assuming pseudorandomness of the underlying PRF, our PRSet scheme satisfies the properties of Functionality Preservation in Hard Puncturing (Definition 4.6)*

Proof. For any $n, \lambda, x \in \{0, \ldots, n-1\}$, for $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$, $sk^x \leftarrow \mathsf{HardPunc}(msk, sk, x)$ 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 Puncturing above, it follows that

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

.

■

A.2.2 Security Proofs

Security in Puncturing. *Assuming pseudorandomness and privacy w.r.t. puncturing of the underlying PRF the following two distributions are computationally indistinguishable for any* $x \in \{0, \ldots, n-1\}$*:*

- $\textbf{Expt}_0$: Run $\mathsf{Gen}(\lambda, n) \rightarrow (sk, msk)$. Return sk.
- $\textbf{Expt}_1$: $\mathsf{Gen}(\lambda, n) \rightarrow (sk, msk)$ until $x \in \mathsf{EnumSet}(sk)$. Return $sk_x = \mathsf{Punc}(msk, sk, x)$.

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

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

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

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

Distinguisher D_0^*

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

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}.
3. Run PRF.$\mathsf{Gen}(1^\lambda, L, m) \rightarrow sk$, PRF.$\mathsf{Puncture}(sk, P_2) \rightarrow sk_{P_2}$.
4. Set our secret key $sk' = (sk_{P_2}, sk_{P_b})$.
5. Return $D_0(sk')$.

Note that in the case where $b = 0$, the experiment is exactly equivalent to D_0's view of $\textbf{Expt}_0$, since sk' is two random m-privately puncturable PRF keys punctured and m points starting with a 1-bit. Also, when $b = 1$, D_0's view is exactly equivalent to $\textbf{Expt}_1^*$, 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 puncturing. 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 $\textbf{Expt}_1^*$ and $\textbf{Expt}_1$ are indistinguishable, assuming pseudorandomness of the underlying PRF. Now, assume there exists a distinguisher D_1 that can distinguish between $\textbf{Expt}_1^*$ and $\textbf{Expt}_1$ 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 follows, for any $x \in \{0, \ldots, n-1\}$:

Distinguisher D_1^*

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

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.Gen$(1^\lambda, L, m) \to sk$, PRF.Puncture$(sk, P_1) \to sk_{P_1}$. Let $sk' = (sk_{P_1}, sk_P)$.

3. If $\forall i \in [1, m]$, PRF.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 contrapositive then, assuming pseudorandomness of the PRF, it must be that **Expt**$_1$ and **Expt**$_1^*$ are indistinguishable. This concludes our proof. ■

We now prove the **Security in Hard Puncturing** for our implementation

Security in Hard Puncturing. *Assuming privacy w.r.t. puncturing of the underlying prf (defined in Section 4.1), the following two distributions to be computationally indistinguishable for any $x \in \{0, \ldots, n-1\}$:*

- **Expt**$_0$: Run Gen$(1^\lambda, n) \to (sk, msk)$ and return $sk^x \leftarrow$ HardPunc(msk, sk, x).
- **Expt**$_1$: Run Gen$(1^\lambda, n) \to (sk, msk)$ until $x \in$ EnumSet(sk). Return sk.

Proof. 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 $x \in \{0, \ldots, n-1\}$:

Distinguisher D^*

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

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}.
3. Run PRF.Gen$(1^\lambda, L, m) \to sk$, PRF.Puncture$(sk, P_2) \to sk_{P_2}$.
4. Set our secret key $sk' = (sk_{P_b}, sk_{P_2})$.
5. **If** InSet(sk', x), output $D(sk')$, **else** output a random bit.

Consider the case where $x \in$ EnumSet(sk'). Now

- If P_0 was punctured, D's view is exactly equivalent to **Expt**$_0$ in his experiment, since in HardPunc we output a secret key $sk = (sk[1], sk[2])$ where the $sk[1]$ is punctured at x, $sk[2]$ is punctured at m random points starting with a 1, and InSet(sk, x) = true.
- If P_1 was punctured, D's view is exactly equivalent to **Expt**$_1$ in his experiment, by construction of Gen, P_1 and P_2, the sk outputted is equivalent to a key outputted by Gen$(1^\lambda, n)$ where InSet(sk, x) = true.

Then, we conclude that, conditioned on $\mathsf{InSet}(sk_{P_b}, x) = true$, D's view of the experiment is exactly equivalent to **Security in Hard Puncturing**, and therefore it will be able to distinguish between whether P_0 and P_1 was punctured with non-negligible probability. Since the probability Pr[$\mathsf{InSet}(sk', x) = true$] $= \frac{1}{\sqrt{n}} > negl(n)$ (we fix a random $sk[2]$ and the probability follows), then the D^* we constructed will break the privacy w.r.t. puncturing of the PRF with non-negligible probability. By contrapositive, assuming privacy w.r.t. puncturing, security in hard puncturing holds. ■

Security Corollary 1 (Randomness in puncturing). *In some distribution $\mathbb{D}_n$, the following the 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 $sk_x \leftarrow \mathsf{Punc}(msk, sk, x)$, 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(ϕ) where $\phi = 2^{-m}$. Output the tuple ($\mathsf{EnumSet}(sk)$, Bernoulli(ϕ)).

Note that $x \in \mathsf{EnumSet}(sk)$ denotes a boolean.

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 $sk^x \leftarrow \mathsf{HardPunc}(msk, sk, x)$. 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 hard puncturing, and puncturing a key with x at 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(ϕ) expect with negligible probability, and by security of hard puncturing as we saw above generating a set key until we find one with x is indistinguishable from generating a fresh key and hard puncturing x. Then, it follows that **Expt**$_1$ and **Expt**$_0^*$ are also indistinguishable, and this concludes our proof.

■

Security Corollary 2 (Efficient HardPunc). *The constructions of* $\mathsf{HardPunc}$ *defined in Section 4.6 and Section 4.6.1 are equivalent.*

Proof. Follows in a straightforward manner from the construction of privately puncturable PRFs that we use. ■

B New PIR Protocol proofs

We prove the following theorem:

Theorem 5.1 (PIR Protocol). *The scheme in Figure 3 satisfies privacy and correctness as defined in Section 5.1 and has the following complexities:*

- $\tilde{O}(\sqrt{n} + Q)$ *client storage and no additional server storage.*
- $\tilde{O}(n/Q + \sqrt{n})$ *amortized server time.*
- $\tilde{O}(\sqrt{n})$ *client time.*
- $\tilde{O}(\sqrt{n}/Q)$ *amortized bandwidth.*

The efficiencies in the scheme follow directly from construction of the scheme and efficiencies of the PRSet proved above.

For privacy and correctness, we will present proofs here for *one* instance of the PIR scheme. Specifically, we will prove overwhelming probability of privacy holding, and probability greater than $\frac{1}{2}$ of correctness holding. Then, the overwhelming probability of privacy of the final scheme follows from the privacy of each copy, and overwhelming probability of correctness follows from the Chernoff bound (seen in Section 2) when we take the majority vote over k instances.

B.1 Privacy Proof for our PIR scheme

We present the privacy proof for **server**$_2$ in our scheme, since as discussed in Section 6, **server**$_1$ privacy is trivial.

Privacy with respect to **server**$_2$, as per our definition, must be argued by showing $\exists$ `Sim`, a stateful algorithm that can run without knowledge of the query and be indistinguishable from an honest execution of the protocol, from the view of any p.p.t. `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 and runs $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$, and sends sk to **server**$_2$.

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

- **Expt**$_1^*$: For each query index x_t that **client** receives, **client** runs $(sk, msk) \leftarrow \mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk, x)$. **client** sends $sk_x \leftarrow \mathsf{Punc}(msk, sk, x)$ to the **server**$_2$.

Note that by **Security in Puncturing**, it follows straighforwardly that **Expt**$_1$ and **Expt**$_1^*$ are indistinguishable.

Next, we define another intermediate experiment **Expt**$_0^*$ to help aid 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 set key (msk_b, sk_b) from our secondary sets and running $sk_b' = \mathsf{HardPunc}(msk_b, sk_b, x_t)$, we generate a new random set using with $(msk, sk) \rightarrow \mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk, x_t)$ and replace our used set with sk instead.

First, we note that by **Security in Hard Puncturing** it follows directly that **Expt**$_0$ and **Expt**$_0^*$ are computationally indistinguishable.

Then, we move 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 $|T|$ random set keys. For the first query index, x_1, we either

- Pick a set key $(msk_j, sk_j) \in T$ from these random sets where $\mathsf{InSet}(sk_j, x_1)$.
- If the step above fails, we run $(msk_j, sk_j) \rightarrow \mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk_j, x_1)$.

Then, we send to **server**$_2$ $sk'_j \leftarrow \mathsf{Punc}(msk_j, sk_j, x_1)$. Note that the second case is trivially equivalent to generating a random set with x_1 and puncturing it at x_1. But in the first case, note that T holds a sequence of outputs from $\mathsf{Gen}(1^\lambda, n)$. As a matter of fact, looking at it in this way, (msk_j, sk_j) is the first output in a sequence of $\mathsf{Gen}(1^\lambda, n)$ that satisfies the constraint of x being in the set. Then, if we consider just the executions from 1 to j, this means (msk_j, sk_j) is equivalent to running $\mathsf{Gen}(1^\lambda, n)$ until $\mathsf{InSet}(sk_j, x)$, except we ran this in advance, which does not make a difference.

Because in both cases it holds, it follows that for the first query, q_1, $\mathbf{Expt}_0^*$ is indistinguishable from $\mathbf{Expt}_1^*$. 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 $\mathbf{Expt}_0^*$ is indistinguishable from each query in $\mathbf{Expt}_1^*$.

Base Case. Initially, our table T is a set of $|T|$ random PRSet keys generated independently from the queries, offline.

Inductive Step. After each query, a random set key $(msk, sk) \leftarrow \mathsf{Gen}(1^\lambda, n)$ that satisfies $\mathsf{InSet}(sk, x)$ with a new random set key $(msk', sk') \leftarrow \mathsf{Gen}(1^\lambda, n)$ that satisfies $\mathsf{InSet}(sk', x)$. 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.

Note that since, at every step t, we pick a set from a table T of fresh random sets, then by what we saw above, every query q_t from the **client** will be indistinguishable from a random set key. But that is exactly what $\mathbf{Expt}_1$ does. Then, we can say that $\mathbf{Expt}_1^*$ is indistinguishable from $\mathbf{Expt}_0^*$. This concludes our proof for experiment indistinguishability. Since we have defined a way to simulate our protocol *without* access to each x_t, such that the view is indistinguishable for any p.p.t non-uniform adversary $\mathtt{A}$, it follows that we satisfy **server**$_2$ privacy. ■

B.2 Correctness Proof for our PIR Scheme

Note that for any query q with query index x, we have three sources of error, as denoted in the introduction. $\mathsf{DB}[x]$ will be incorrect when we either

- Cannot find a set key that contains x in T
- When we puncture incorrectly during query, and we define 'puncturing incorrectly' to be any of the two mutually exclusive events
  1. The puncture operation does *not* remove x from the set.
  2. The puncture operation removes *more* than just x from the set.
- We hard puncture incorrectly during refresh, which is limited to one case: when we add *more* than just x to the set.

Note that the first two sources of error occur *before* refresh, and this means that, if these were the only errors, as long as we could bound the sum of their probability to occur with probability less than half, then our scheme would be correct due to the k iterations and Chernoff bound. However, the last source of error occurs *after* the majority vote and therefore potentially propagates some incorrect set key and parity pair $((msk_k, sk_k), p_k)$ to following queries. To deal with this, for each query, we consider the probability that the entry of T picked by the query is incorrect due to this error. Then, we instead consider the sources of error in the following way:

- Cannot find a set key that contains x in T.

- When puncturing yields incorrect parity:
  1. The puncture operation does *not* remove x from the set.
  2. The puncture operation removes *more* than just x from the set.
  3. Probability that the set we are using was hard punctured and yields incorrect parity.

Bounding probability of puncture yielding incorrect parity during query

We first consider a modified PIR experiment that is exactly equivalent to our scheme in Figure 3 in terms of correctness. We define it below:

Modified PIR Scheme

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

Offline phase

- **client** generates $\ell + Q$ PRSet keys $(msk_1, sk_1), \ldots, (msk_{\ell+Q}, sk_{\ell+Q})$ each with $\mathsf{Gen}(1^\lambda, n)$.
- **client** sends all the secret keys, $sk_1, \ldots, sk_{\ell+Q}$ to **server**$_1$ and gets back a set of bits $p_1, \ldots, p_{\ell+Q}$ where $p_i = \oplus_{j \in S_i} \mathsf{DB}[j]$. For each set key sk_i, if $\mathsf{EnumSet}(sk_i) = \perp$, **server**$_1$ sets $p_i = 0$.
- **client** stores hints $T = \{T_j\}$ where each $T_j = ((msk_j, sk_j), p_j, null, \{\})\}_{j \in [1,\ell]}$, $B = \{B_k\}$ where each $B_k = ((msk_k, sk_k), p_k)\}_{k \in [\ell+1, \ell+Q]}$.

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

• Query

1. **client** finds the smallest j such that $T_j = ((msk_j, sk_j), p_j, z_j, \mathtt{setRelated}_j) \in T$ and $(\mathsf{InSet}(sk_j, x) \vee (x == z_j) \vee (x \in \mathtt{setRelated}_j))$. 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** runs $sk_j^{z_j} \leftarrow \mathsf{HardPunc}(msk_j, sk_j, z_j)$
3. **client** sends $sk' = \mathsf{Punc}(msk_j, sk_j^{z_j}, x)$ to **server**$_2$, that returns $r = \oplus_{k \in \mathsf{EnumSet(sk')}} \mathsf{DB}[k]$.
4. **client** computes $\mathsf{DB}[x] = r \oplus p_j$.
5. **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 $sk_k^x = \mathsf{HardPunc}(msk_k, sk_k, x)$.
3. **client** computes

$$\mathtt{setRelated} = \mathsf{EnumSet}(sk_k^x) \setminus \mathsf{EnumSet}(sk_k) \setminus \{x\}$$

   and sets

$$T_j = ((msk_k, sk_k), p_k \oplus (\mathsf{DB}[x]' \wedge \mathsf{InSet}(sk_k, x)), x, \mathtt{setRelated})$$

   and also sets $B = B \setminus B_k$.

This modified PIR scheme is exactly equivalent to our PIR scheme in Figure 3, except we take extra steps in order to defer defer the hard puncture step to the query and with that better expose the potential query errors. We examine it in this light because it exposes potential errors in puncturing more clearly for our scheme.

Let us denote $\texttt{HardPuncRel}$ the event where $\mathsf{HardPunc}$ added *more than one element* to the set. It is clear to see that in our modified scheme, $\texttt{HardPuncRel}$ occurrs if and only if $\texttt{setRelated}_j \neq \{\}$. It is straightforward to see that if $\texttt{HardPuncRel}$ occurs, then the parity p_j is incorrect. From Lemma 3.4 and **Functionality Preservation in Hard Puncturing**, we can say that:

$$\Pr[\texttt{HardPuncRel} \leq \frac{1}{2\log n} + negl(\lambda)$$

Also note that if we puncture *more than just* x, our final $\mathsf{DB}[x]$ will be incorrect. We puncture more than just x when there exists some element in $y \in \mathsf{EnumSet}(sk_j^{z_j})$ s.t. $\texttt{Related}(y, x) = 1$ for $z_j \neq x$. We denote this event $\texttt{PuncRel}$ and separate it as:

$$\begin{aligned}\Pr[\texttt{PuncRel}] =& \Pr[\texttt{PuncRel}|\texttt{Related}(x, z_j) = 1 \wedge x \neq z_j] \\ &\cup \Pr[\texttt{PuncRel}|\texttt{Related}(x, z_j) = 0]\end{aligned}$$

Note that:

$$\begin{aligned}&\Pr[\texttt{PuncRel}|\texttt{Related}(x, z_j) = 1 \wedge x \neq z_j] \\ &\qquad \leq \Pr[\texttt{Related}(x, z_j) = 1 \wedge x \neq z_j] \\ &\qquad \leq \Pr[\texttt{setRelated}_j \neq \{\}] \\ &\qquad \leq \Pr[\texttt{HardPuncRel}] \leq \frac{1}{2\log n} + negl(\lambda)\end{aligned}$$

Now, let us consider $\Pr[\texttt{PuncRel}|\texttt{Related}(x, z_j) = 0]$. Since z_j and x are not related, the membership of x in the set is independent from z_j, and therefore we can bound this probability as:

$$\begin{aligned}&\Pr[\texttt{PuncRel}|\texttt{Related}(x, z_j) = 0] \\ &\qquad \leq \Pr[\texttt{Related}(x, z_j) = 0] \\ &\qquad \leq \Pr[\forall y \in \mathsf{EnumSet}(sk_j^{z_j}), \texttt{Related}(y, z_j) = 0] \\ &\qquad \leq \frac{1}{2\log n} + negl(\lambda)\end{aligned}$$

Given our **Security in Hard Puncturing**, last line follows from Lemma 3.3; since the expected number of elements related to z_j in a set generated with z_j is $\frac{1}{2\log n} + negl(\lambda)$, it follows that the probability of finding such element here is bounded by that. Given that x is independently sampled from z_j by construction, we can look at it in this light. Then, putting it together we have that:

$$\Pr[\texttt{PuncRel}] \leq \frac{1}{\log n} + negl(\lambda)$$

Lastly, we bound the probability of element x not being punctured on Punc, which would yield an incorrect $\mathsf{DB}[x]$. Let us denote this event $\texttt{NotRemoved}$. From a straighforward reduction from Security Corollary 1, we see that the output of the puncturing operation is independent from the set being punctured, and computationally equivalent to a bernoulli variable with $p = \frac{1}{\sqrt{n} * 2^B}$, and therefore, we bound the probability of not $\texttt{NotRemoved}$ as follows:

$$\Pr[\texttt{NotRemoved}] \leq \frac{1}{\sqrt{n}(\log n)^2} + negl(\lambda)$$

Then, putting together all the probabilities computed, we have the the probability that puncture yields the incorrect $DB[x]$, $\texttt{PuncErr}$,

$$\begin{aligned}\Pr[\texttt{PuncErr}] &\leq \Pr[\texttt{HardPuncRel}] \cup \Pr[\texttt{PuncRel}] \cup \Pr[\texttt{NotRemoved}] \\ &\leq \frac{1}{2\log n} + \frac{1}{\log n} + \frac{1}{\sqrt{n}(\log n)^2} + negl(\lambda)\end{aligned}$$

Bounding probability of not finding query index

Let us denote the event of not finding our query index x in our sets as $\texttt{NotFound}$. From Lemma 3.2 , we see that for the first query, the probability of not an index $x \in \{0, \ldots, n-1\}$ in any entry of T, denoted $\Pr[x \notin T] \leq \frac{1}{n}$. Then, from our privacy proof earlier in the section, the distribution of T is unchanged throughout our Q queries. From this, we conclude that for each query x_t for $t \in \{1, \ldots, Q\}$, $\Pr[x_t \notin T] \leq \frac{1}{n} + negl(\lambda)$, and so we say that, for each query:

$$\Pr[\texttt{NotFound}] \leq \frac{1}{n} + negl(\lambda)$$

Bounding probability of error in scheme

We denote $\texttt{IncQuery}$ as the event that our single copy scheme outputted an incorrect $\mathsf{DB}[x]$. We can say that:

$$\begin{aligned}\Pr[\texttt{IncQuery}] &\leq \Pr[\texttt{NotFound}] \cup \Pr[\texttt{PuncErr}] \\ &\leq \frac{1}{n} + \frac{1}{\sqrt{n}(\log n)^2} + \frac{3}{2\log n} + negl(\lambda)\end{aligned}$$

Then, we can loosely bound our single-copy scheme to be correct with probability more that $\frac{1}{2}$ for any $n > 100$. By running $k = \omega(\log \lambda)$ instances and taking a majority vote for $\mathsf{DB}[x]$, by the Chernoff Bound, our scheme is correct for Q queries with overwhelming probability in our security parameter λ.

B.3 Deterministic Time Bounds

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

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 Lemma A.1, 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.

HardPunc. To get deterministic run-time for $\mathsf{HardPunc}$, we can cap the hard puncturing introduced at Section 4.6 at $2\sqrt{n}(\log n)$ function calls, and it will fail with probability $\leq \frac{1}{\log n}$ (by Lemma A.2). For the improved implementation presented in Section 4.6.1, we can bound a similar error probability with only $2(\log n)^2$ with a similar proof. We show that these implementations are equivalent in Security Corollary 2. 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 as discussed in the notes in Section 5.2.2. Correctness follows as in $\mathsf{EnumSet}$.

C Single Server PIR

C.1 Batch Parity Retrieval Circuit and FHE

As was outlined in Section 6.1, one of the requirements to port our scheme to single server is a batch parity retrieval circuit. In [19], Corrigan Gibbs et. al showed a construction for a batch parity retrieval

circuit C that can compute the parity of l lists of size m in $\tilde{O}(l \cdot m + n)$ time, where n is the size of the database. This works in a straightforward manner for their pseudorandom sets from PRPs since the sets are of fixed size. To use this for our new PIR algorithm from Figure 3 that uses Extended Pseudorandom Sets introduced in Section 4, 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 PRSet.$\mathsf{EnumSet}$ algorithm to output a list instead of a set, and pad any list of size $< \frac{\sqrt{n}}{\log n}$ with the index n until the list is of size $\frac{\sqrt{n}}{\log n}$, and output $\perp$ if any set list is of size $> \frac{\sqrt{n}}{\log n}$. 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 PRSet enumeration algorithm and circuit C, along with the FHE scheme introduced in Section 6.1 we finally construct our single server PIR scheme in Figure 4.

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

Lemma C.1 (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* $|L| > w$*, and 0 otherwise. Let* $s = \ell + Q$ *for some* $\ell, Q \in \mathbb{N}$*. Then, for* $S_1, \ldots, S_{s\cdot(\log n)^2} \sim (\mathbb{D}_n)^{s\cdot(\log n)^2}$

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

We provide the proof in Appendix C.3. From this Lemma, we see that this restriction incurs an additional correctness failure of $\frac{1}{\log n}$ compared to our normal scheme.

The online phase runs in exactly the same way as our scheme in Figure 3.

C.2 Theorem Proof

Now, we set out to prove that our scheme satisfies Theorem 6.1. We re-state it here:

Theorem 6.1 (Single-Server PIR Protocol)**.** *There exists a single-server PIR protocol satisfies privacy and correctness as defined in Section 6.2 and has the following complexities:*

- $\tilde{O}(\sqrt{n} + Q)$ *client storage and no additional server storage.*
- $\tilde{O}(n/Q + \sqrt{n})$ *amortized server time.*
- $\tilde{O}(\sqrt{n})$ *client time.*
- $\tilde{O}(\sqrt{n}/Q)$ *amortized bandwidth.*

Proof. The efficiencies follow from the efficiencies in the scheme in Figure 3, except for extra polylogarithmic factors and λ 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 the privacy proof in Appendix B.

PIR Scheme

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

Offline phase

- **client** generates s PRSet 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.Dec into the parity p_i and stores the first ℓ hints where $p_i \neq \perp$ in $T = \{T_j = ((msk_j, sk_j), p_j)\}_{j \in [1,\ell]}$, and the next Q hints that $p_i \neq \perp$ in $B = \{B_k = ((msk_k, sk_k), p_k)\}_{k \in [\ell+1, \ell+Q]}$.

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 $sk' = \mathsf{Punc}(msk_j, sk_j, x)$ to **server**$_2$, 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 $sk_k^x = \mathsf{HardPunc}(msk_k, sk_k, x)$.
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 4: Our 1PIR protocol.

Correctness follows from the correctness proof in Appendix B and Lemma C.1, along with correctness of C and the FHE scheme. Note that for each single copy scheme, we incur exactly the same errors as in the 2PIR scheme, with the addition of the additional error factor offline when we do not have the right amount of sets. This only happens with probability $\leq \frac{1}{\log n}$ for any $n > 4$ (see Lemma C.1). Note that for larger n, we can potentially tighten this bound and require less additional sets. It is clear to see that this extra factor does not take the correctness probability of the single copy scheme to be $< \frac{1}{2}$ for relevant n, so by taking the majority vote for $\mathsf{DB}[x]$ over $\omega(\log \lambda)$ instances in each query, by the Chernoff bound argument and the arguments in Appendix B, this scheme is correct with overwhelming probability. ■

C.3 Lemma Proofs

Proofs for Lemmas used throughout the section.

Lemma C.1 (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* $|L| > w$*, and 0 otherwise. Let* $s = \ell + Q$ *for some* $\ell, Q \in \mathbb{N}$*. Then, for* $S_1, \ldots, S_{s\cdot(\log n)^2} \sim (\mathbb{D}_n)^{s\cdot(\log n)^2}$

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

Proof. From Lemma 3.1, we know that the expected size of S, $|S| = \frac{\sqrt{n}}{(\log n)^2}$. 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:

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

Now, if we just apply the Markov bound again:

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

■