Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time*

Elaine Shi[1], Waqar Aqeel[2], Balakrishnan Chandrasekaran[3], and Bruce Maggs[2]

[1] CMU runting@cs.cmu.edu
[2] Duke waqeel,bmm@cs.duke.edu
[3] Vrije Universiteit Amsterdam b.chandrasekaran@vu.nl

Abstract. Imagine one or more non-colluding servers each holding a large public database, e.g., the repository of DNS entries. Clients would like to access entries in this database without disclosing their queries to the servers. Classical private information retrieval (PIR) schemes achieve polylogarithmic bandwidth per query, but require the server to perform linear computation per query, which is a significant barrier towards deployment.

Several recent works showed, however, that by introducing a one-time, per-client, off-line preprocessing phase, an *unbounded* number of client queries can be subsequently served with sublinear online computation time per query (and the cost of the preprocessing can be amortized over the unboundedly many queries). Existing preprocessing PIR schemes (supporting unbounded queries), unfortunately, make undesirable tradeoffs to achieve sublinear online computation: they are either significantly non-optimal in online time or bandwidth, or require the servers to store a linear amount of state per client or even per query, or require polylogarithmically many non-colluding servers.

We propose a novel 2-server preprocessing PIR scheme that achieves $\widetilde{O}(\sqrt{n})$ online computation per query and $\widetilde{O}(\sqrt{n})$ client storage, while preserving the polylogarithmic online bandwidth of classical PIR schemes. Both the online bandwidth and computation are optimal up to a polylogarithmic factor. In our construction, each server stores only the original database and nothing extra, and each online query is served within a single round trip. Our construction relies on the standard LWE assumption. As an important stepping stone, we propose new, more generalized definitions for a cryptographic object called a Privately Puncturable Pseudorandom Set, and give novel constructions that depart significantly from prior approaches.

1 Introduction

Imagine that a service provider has a large public database, DB, and is serving clients who request records from DB. For example, in a search-engine scenario

* Please read the online full version [47] for complete details and proofs.

each entry in DB may be the search result for a specific keyword; in the DNS scenario, each entry contains the records for a specific domain name. Without loss of generality, we may assume that the database $\mathsf{DB} \in \{0,1\}^n$ is an array of bits indexed by $\{0,1,\ldots,n-1\}$, and a client's query is an index $i \in \{0,1,\ldots,n-1\}$ into DB[4]. Although the database itself is public, the clients wish to hide their queries from the server. This problem has been studied in a beautiful line of work called Private Information Retrieval (PIR), first formulated by Chor, Goldreich, Kushilevitz, and Sudan [18,19]. Since then, a rich line of work [4,9,10,13,16,17, 21,23,24,26,28,32,34,35,37,38,40,42–44] has improved the original construction of Chor et al. [18]. This paper focuses on *2-server PIR*, i.e., there are two non-colluding servers, and the goal is to prevent each individual server from learning anything about the clients' actual queries.

Single- or multi-server PIR schemes with *polylogarithmic* bandwidth (bits sent per query) and *linear server work* per query are well known [9,10,13,16,17, 24,28,32,34,37,38,42–44]. While these PIR schemes are elegant in construction and achieve non-trivial asymptotic bounds, the prohibitive server running time per query is a significant barrier towards practical deployment. For example, in our motivating applications, the database may have billions or trillions of entries. Unfortunately, in the original formulation phrased by Chor et al. [18], linear server work is required to achieve privacy [6] — intuitively, if there is a location that the server does not need to read, the query is definitely not looking for that location. To avoid this drawback, a promising direction has been suggested by a few recent works [6,20], namely, PIR with *preprocessing*. In PIR with preprocessing, clients and servers are allowed to perform one-time offline preprocessing. After preprocessing, the PIR scheme should support an *unbounded* number of queries from each client. The cost of the offline preprocessing can thus be amortized "away" over sufficiently many queries, and we can hope for *sublinear amortized (i.e., online) running time* per query.

Preprocessing PIR was considered in several prior works [6, 38, 44]. Beimel, Ishai, and Malkin [6] were the first to suggest using preprocessing to reduce the server's online computation. They constructed a statistically secure 2-server PIR scheme with n^ϵ online bandwidth and running time for some constant $\epsilon \in (0,1)$ by having the servers preprocess the n-bit DB into an encoded version of $\mathsf{poly}(n)$ bits. The line of work on preprocessing PIRs culminated in the elegant work by Corrigan-Gibbs and Kogan [20], who showed that, assuming one-way functions, there is a 2-server preprocessing PIR scheme with $O(\sqrt{n})$ online bandwidth and running time (ignoring the dependence on the security parameter). In their scheme the servers store only the original database DB and nothing extra, but each client needs to store a "hint" of size $O(\sqrt{n})$. Corrigan-Gibbs and Kogan [20] also proved that the $O(\sqrt{n})$ online computation is optimal, assuming that the client downloads only $O(\sqrt{n})$ amount of information from the server during preprocessing and that the servers store only the unencoded database (and the proof works by reducing PIR to Yao's Box problem [53]). The main drawback with

[4] If the query is a keyword or domain name, it can be hashed to an index, and if each entry has multiple bits, we can treat it as retrieving multiple indices.

their scheme is the significantly non-optimal $O(\sqrt{n})$ online bandwidth which is also much worse than classical PIR without preprocessing.

Given the state of affairs for preprocessing PIR, we ask the following question:

Can we construct a preprocessing PIR scheme that is simultaneously optimal in online bandwidth and online time?

Before we present our results and contributions, we point out a couple of important desiderata and clarify the problem statement:

- *Unbounded query setting.* First, we want the PIR scheme to support an *unbounded* number of queries after a one-time processing. This is necessary in the vast majority of conceivable applications (e.g., oblivious DNS [1, 49], oblivious Safe Browsing [2], the four excellent use cases in the Splinter work [52], and other applications [3, 4]). Unsurprisingly, state-of-the-art PIR implementations invariably support unbounded number of queries too [3, 4, 52]. Without the unbounded requirement, there is indeed a scheme with $O(\sqrt{n})$ online computation and $\widetilde{O}(1)$ online bandwidth shown in the same work of Corrigan-Gibbs and Kogan [20] — unfortunately, this scheme supports only a single query after the preprocessing, and thus the linear preprocessing cost should be charged to each query, and cannot be amortized over multiple queries.
- *No per-client server state.* Second, the server should not have to store per-client state. There are alternative solutions if we let the server store per-client state (and often $O(n)$ state per client). For example, one strawman candidate is to use an Oblivious RAM (ORAM) scheme [29, 31, 48]. During the offline phase, the client downloads the database from the server and uses a secret key to compile the database into an ORAM which is then stored on the server. This would allow queries to be supported in polylogarithmic running time and bandwidth per query, and constant roundtrips (provided the server can perform computation) [22, 25, 27, 39]. Unfortunately, $\Omega(n)$ per-client state on the server would clearly be a barrier towards practicality in some motivating applications. Similarly, the recent doubly-efficient (1-server) PIR constructions in the designated-client setting [11, 15] also suffers from the same drawback, although they remove the need for clients to store persistent state. A doubly-efficient PIR construction in the public-client setting promises to remove the $O(n)$ per-client state at the server. Unfortunately, the only known such construction relies on virtual blackbox (VBB) obfuscation which is known to be impossible [5]. We compare with additional related works in Section 7.

Besides the above, we also want the client-side storage to be small — if the client could store the entire database, then there is no need to talk to the server.

Our results and contributions. We answer the above question affirmatively, assuming Learning With Errors (LWE) [45]. Our scheme employs two servers, a "left" server and a "right" server and, at a high level, works as follows.

– During the offline preprocessing phase, each client sends a single message of size roughly $\widetilde{O}(\sqrt{n})$ to the left server[5]. The left server responds with a *hint* of $\widetilde{O}(\sqrt{n})$ bits, which is stored by the client. Then online queries begin.
– For each online query, the client sends a single poly-logarithmically sized message to each server in parallel. In particular, the message sent to the right server is used for answering the query. Using its locally stored hint and the right server's response, the client can reconstruct the correct answer to the query except with negligible probability. The message sent to the left server is used to partially "refresh" the client's hint. The client uses the answer from the left server and the outcome of the present query to update one entry in the $\widetilde{O}(\sqrt{n})$-sized hint it stores.

More formally, we prove the following theorem:

Theorem 1 (2-server preprocessing PIR). *Assuming the Learning With Errors (LWE) assumption, there exists a 2-server preprocessing PIR scheme that satisfies the following performance bounds:*

– *the offline server running time is $\widetilde{O}(n)$; the offline client running time and bandwidth is $\widetilde{O}(\sqrt{n})$.*
– *the online server and client time per query is $\widetilde{O}(\sqrt{n})$; the online bandwidth per query is $\widetilde{O}(1)$.*
– *each online query can be accomplished in a single roundtrip, that is, the client sends a single message to each server in parallel, and reconstructs the answer from the two servers' responses respectively; and*
– *each server needs to store only the original database DB and no extra information; each client needs to store $\widetilde{O}(\sqrt{n})$ bits of information.*

Due to the lower bound of Corrigan-Gibbs and Kogan [20], our scheme's total online time is *optimal up to poly-logarithmic factors*, assuming that the client downloads only approximately $\sqrt{n}$ amount of information from the server during preprocessing. In comparison, the prior state-of-the-art scheme [20] can achieve optimal online computation, but their $\sqrt{n}$ online bandwidth is significantly non-optimal. We improve their bandwidth consumption by a roughly $\sqrt{n}$ factor, and thus achieve near optimality in both online computation and bandwidth. Table 1 compares our result with the most relevant prior work.

Theorem 1 does not give the exact constant c in the hidden $\log^c n$ factor; however, in the online full version [47], we give a more careful analysis of the concrete constants. Specifically, we show that with some fine-tuning, we can get the following more precise asymptotical performance where $\alpha(\lambda)$ denotes an arbitrarily small super-constant function: the offline server time is $O(n \log^2 n \log \lambda) \cdot \alpha(\lambda)$, the offline client time is $O(\sqrt{n} \log^2 n \log \lambda) \cdot \alpha(\lambda)$, and the offline client bandwidth is $O(\sqrt{n} \log^2 n \log \lambda) \cdot \alpha(\lambda)$. Moreover, the online client time per query is

[5] The $\widetilde{O}(\cdot)$ notation hides polylogarithmic factors and dependence on the security parameter.

Table 1: Comparison with prior schemes. Includes only schemes where the servers need not store per-client state, has sublinear online time, and supports an unbounded number of queries (possibly after a one-time preprocessing). Sections 1 and 7 review additional related work in the broader design space, when we are willing to relax these desiderata. "C-Time", "S-Time", and "BW" denote client time, server time, and bandwidth, respectively. "OLDC" means oblivious locally decodable codes, and "VBB Obf." means virtual-blackbox obfuscation. $\epsilon \in (0,1)$ is a constant.
★: Beimel et al. [6] requires the servers to store a large $\mathsf{poly}(n)$ amount of state.

Scheme	#server	Assumpt.	**Offline** C-Time	S-Time	BW	**Online** C-Time	S-Time	BW
[6]★	2	None	0	$\mathsf{poly}(n)$	0	n^ϵ	n^ϵ	n^ϵ
[20]	2	OWF	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$	$O(\sqrt{n})$
	2	OWF	$O(\sqrt{n})$	$O(n)$	$O(\sqrt{n})$	$O(n^{5/6})$	$O(\sqrt{n})$	$\widetilde{O}(1)$
[11]	1	OLDC, VBB Obf.	0	0	0	n^ϵ	n^ϵ	n^ϵ
Our PIR	2	LWE	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(n)$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(\sqrt{n})$	$\widetilde{O}(1)$
LB [20]	-	-	-	-	n/β	-	β	-

$O(\sqrt{n}\log^2 n \log\lambda)\cdot\alpha(\lambda)$, the online server runtime is $O(\sqrt{n}\log n\log\lambda)\cdot\alpha(\lambda)$, and the online bandwidth per query is $O(\log n\cdot\log\lambda)\cdot\alpha(\lambda)$.

Furthermore, in the online full version [47], we also discuss how to tune the parameters to get near optimality of the online bandwidth and computation, for every choice of offline bandwidth, in light of the known lower bound [20].

Remark 1. Like in earlier works [20], for simplicity, in our asymptotical performance bounds, we hide a security parameter $\chi(\lambda)$ factor that is related to the strength of the LWE assumption. If we assume standard polynomial security, $\chi(\lambda)$ is polynomially bounded in λ; if we assume subexponential security, $\chi(\lambda)$ is poly-logarithmic in λ.

Technical highlight. Our 2-server preprocessing PIR scheme is inspired by the very recent work of Corrigan-Gibbs and Kogan [20]. At a high level, their work shows how to construct a 2-server preprocessing PIR scheme using a cryptographic object which they call a Puncturable Pseudorandom Set (PRSet). A PRSet scheme provides an algorithm for generating a secret key sk that can be used to generate a pseudorandom subset $\mathbf{Set}(\mathsf{sk}) \subseteq \{0,1,\ldots,n-1\}$; sk thus serves as a succinct representation of the set $\mathbf{Set}(\mathsf{sk})$. Further, there is an efficient puncturing algorithm: suppose some element $x \in \mathbf{Set}(\mathsf{sk})$, then $\mathbf{Puncture}(\mathsf{sk}, x)$ outputs a punctured key sk_x that effectively removes x from the set, i.e., $\mathbf{Set}(\mathsf{sk}_x) = \mathbf{Set}(\mathsf{sk})\backslash\{x\}$.

Unfortunately the Corrigan-Gibbs and Kogan [20] PRSet scheme is not efficient in all dimensions, namely, set enumeration time, membership test time,

and punctured key size. As a tradeoff, they opt for efficient set enumeration and efficient membership test, allowing their PIR scheme to achieve roughly $\sqrt{n}$ online running time. Their PRSet scheme, however, adopts a trivial puncturing algorithm. The punctured key is simply the entire punctured set itself minus the element x to be removed, which causes their online bandwidth to be roughly $\sqrt{n}$, which is asymptotically worse than classical PIR schemes without preprocessing [10, 13, 17, 24, 26, 28, 37, 38, 42].

To achieve our stated result, an important stepping stone is to construct a new Privately Puncturable Pseudorandom Set (PRSet) that is efficient in all dimensions. Unfortunately, as explained in Section 2, these requirements seem to be inherently conflicting, and we were not able to directly reconcile them — likely Corrigan-Gibbs and Kogan [20] encountered the same barriers.

Our key insight is to observe that the Corrigan-Gibbs and Kogan formulation of a PRSet scheme seems too restrictive. We generalize their PRSet abstraction in the following ways.

1. *Emulating a customized sampling distribution.* Corrigan-Gibbs and Kogan consider only PRSet schemes that emulate simple distributions, such as sampling a random $\sqrt{n}$-sized subset among n elements, or sampling each element at random with probability $1/\sqrt{n}$. By contrast, we generalize the PRSet definition to allow it to emulate an arbitrary distribution of choice. Later we discuss the challenges of choosing this distribution.
2. *Relaxed correctness definition.* Corrigan-Gibbs and Kogan's definition insists on almost-always correctness. We observe that a weaker notion, which we call "occasional correctness," is sufficient for obtaining a 2-server preprocessing PIR (since our PIR construction relies on parallel repetition to amplify the correctness to $1 - \mathsf{negl}(\lambda)$) where λ denotes the *security parameter* globally. Specifically, we want the puncturing algorithm to remove the point x being punctured, and only the point x — but we only need this to happen with considerable but not overwhelming probability.

Therefore, one technical contribution we make is to devise a more generalized/relaxed abstraction of a Privately Puncturable Pseudorandom Set (PRSet) scheme that is suitable and sufficient for constructing an efficient 2-server preprocessing PIR. To do so, we need to identify an appropriate sampling distribution that the PRSet should emulate. In our carefully chosen distribution, each element from $\{0, 1, \ldots, n-1\}$ is included in the set with roughly $1/(\sqrt{n} \cdot \mathsf{poly} \log n)$ probability, but the sampling is not completely independent among the elements. For example, if some element x is included in the set, it might make some other element y more likely to be included. As we explain in more detail in Sections 2 and 4.4, an independent distribution seems to facilitate an efficient membership test, but preclude efficient set enumeration; on the other hand, having more dependence and the right type of dependence can enable efficient set enumeration, but may destroy the efficiency of the membership test. We seek a middle ground by choosing a distribution that has a limited amount of dependence, and the right type of dependence.

We next show how to construct a PRSet scheme that emulates our carefully chosen distribution, and prove the construction secure under our new definitions. Our construction relies on the existence of Privately Puncturable PRFs which can be constructed assuming LWE [7, 9, 12, 14]. Our PRSet construction is remotely inspired by the line of work on designing block ciphers and format preserving encryption from pseudorandom functions [41, 46, 50], but our problem definition and solutions are novel and fundamentally different from prior works.

Finally, we use our PRSet scheme to construct a 2-server preprocessing PIR scheme and prove the PIR scheme correct and secure. Our construction is inspired by Corrigan-Gibbs and Kogan [20] but differs in several important details. The proofs are rather technical and involved. Perhaps somewhat surprisingly, proving correctness turns out to be the most technically challenging part of our proof, although proving privacy is also non-trivial. Our PIR scheme runs k parallel instances of a single-copy PIR scheme. We need to prove occasional correctness of each single-copy scheme, and use majority voting among all instances to amplify correctness. Unfortunately, we cannot easily argue occasional correctness of the single-copy PIR from the occasional correctness of the PRSet scheme. Part of challenge arises from the fact that conditioning on events that have taken place skews the distribution of the pseudorandom sets, and we need to make an occasional correctness argument even for this skewed distribution (which does not even have a clean and succinct description). At a very high level, to make the argument work, we make an involved stochastic domination argument that effectively shows that conditioning on the events that have taken place will not worsen the probability of certain bad events that could lead to incorrectness. We refer the reader to Section 4.4 for more detailed discussions on the technicalities in the proof.

Non-goals and open questions. Previous preprocessing PIR schemes in the unbounded query setting are significantly non-optimal in either online bandwidth or computation. Our work is primarily a theoretical exploration aimed at *bridging the important theoretical gap in our understanding. We do not claim immediate practicality of our scheme.* We believe, however, that achieving asymptotical near optimality represents an important step forward towards eventually having a practical PIR scheme. Specifically, we suggest the following possible future directions towards better concrete performance: 1) the parameters in our current theorems are not tight, therefore concrete security parameterization is a potential improvement; and 2) designing a concretely efficient Privately Puncturable PRF would be critical to concrete performance. For example, instantiations based on other assumptions might be more efficient than the current LWE-based schemes.

Besides improving concrete performance, there are also interesting theoretical open questions. One seemingly challenging question is whether we can asymptotically reduce the client online time — the lower bound by Corrigan-Gibbs and Kogan [20] shows that the server computation (or the combined server-client computation) must be at least $\sqrt{n}$ per query, assuming the client downloads $\sqrt{n}$ information from the server during pre-processing. The known lower bound does not rule out schemes with asymptotically smaller client online time.

2 Strawman Attempts

To understand our ideas, it helps to first illustrate a strawman scheme and see why it fails — the toy scheme below is a variant (and slight simplification) of the elegant 2-server preprocessing PIR scheme by Corrigan-Gibbs and Kogan [20]. This toy scheme is meant for illustrating the "core" of the scheme, and is not concerned about compressing storage or bandwidth.

An Inefficient Toy Scheme: Single-Copy Version

Offline preprocessing. (DB_k *denotes the* k*-th bit of the database*)

- Client generates $\sqrt{n}$ sets $S_1, S_2, \ldots, S_{\sqrt{n}}$. Each $S_j \subseteq \{0, 1, \ldots, n-1\}$ where $j \in [\sqrt{n}]$ is sampled by including each element $i \in \{0, 1, \ldots, n-1\}$ with independent probability[a] $1/\sqrt{n}$.
- Client sends the resulting sets $S_1, \ldots, S_{\sqrt{n}}$ to Left. For each set $j \in [\sqrt{n}]$, Left responds with the parity bit $p_j := \oplus_{k \in S_j} \mathsf{DB}_k$ of indices in the set.
- Client stores the hint $T := \{T_j := (S_j, p_j)\}_{j \in [\sqrt{n}]}$.

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

- **Query:** ($\mathsf{Client} \Leftrightarrow \mathsf{Right}$)
  1. Find an entry $T_j := (S_j, p_j)$ in its hint table T such that $x \in S_j$. Let $S^* := S_j$ if found, else let S^* be a fresh random set containing x.
  2. Send the set $S := \mathsf{Resample}(S^*, x)$ to Right, where $\mathsf{Resample}(S^*, x)$ outputs a set almost identical to S^*, except that the coin used to determine x's membership is re-tossed.
  3. Upon obtaining a response $p := \oplus_{k \in S} \mathsf{DB}_k$ from Right, output the candidate answer $\beta' := p_j \oplus p$ or $\beta' := 0$ if no such T_j was found earlier.
  4. Client obtains the true answer $\beta := \mathsf{DB}_x$ — the full scheme will repeat this single-copy scheme k times, and β is computed as a majority vote among the k candidate answers, which is guaranteed to be correct except with negligible probability.
- **Refresh** ($\mathsf{Client} \Leftrightarrow \mathsf{Left}$)
  1. Client samples a random set S containing x, and then lets $S' := \mathsf{Resample}(S, x)$, and sends S' to Left (notice that this is equivalent to just sampling a fresh set, but we write it this way for later convenience).
  2. Left responds with $p := \oplus_{k \in S'} \mathsf{DB}_k$. If a table entry T_j containing x was found and consumed earlier, Client replaces T_j with $(S, p \oplus \beta)$.

[a] The work of Corrigan-Gibbs and Kogan [20] samples a set of fixed size $\sqrt{n}$, whereas in our particular variant, the size of each set is a random variable whose expectation is $\sqrt{n}$.

In this toy scheme, during pre-processing, the client samples $\sqrt{n}$ sets each containing $\sqrt{n}$ randomly chosen bits, and downloads the parity of each set from

the left server. During an online query, suppose the client wants the index i, it finds a set S^* containing i. It then resamples the decision whether i should belong to the set, and the resampled set S removes i with high probability. It sends the resampled set S to the right server, which returns its parity. Now, if such a set S^* was found, XORing the parity of the set S^* and the set S gives client the correct answer with high probability. To support an unbounded number of queries, the client performs a refresh procedure with the left server to replenish the set that was just consumed.

Correctness amplification through parallel repetition. The above toy scheme guarantees correctness for the query x, provided that 1) an entry $T_j := (S_j, p_j)$ containing x is found, and 2) $\mathsf{Resample}(S_j, x)$ happens to remove x from the set S_j. It is not hard to prove that correctness is guaranteed with probability at least $3/5$ for sufficiently large n. To amplify correctness, we can run k copies of the scheme, and instead of calling the true-answer oracle to obtain the answer β, we set β to be the majority vote among the k candidate answers, which is correct with $1 - 2^{-\Theta(k)}$ probability due to the standard Chernoff bound. If we set $k = \omega(\log \lambda)$, then the failure probability would be negligibly small in λ.

Privacy. In the inefficient toy scheme, left-server privacy is easy to see: basically the left server Left sees $\sqrt{n}$ random sets during the offline phase. During each online query, it sees a random set as well.

Arguing right-server privacy is a little more subtle. The right server Right is not involved during the offline phase. We want to show that for each online query, Right sees a fresh random set. Recall that during a query for x, the client finds an entry $T_j := (S_j, p_j)$ such that $S_j \ni x$. It lets $S^* := S_j$ if such an entry T_j is found, else S^* is a fresh random set containing x. The client now sends $\mathsf{Resample}(S^*, x)$ to Right and if such a T_j was found and consumed, it replaces T_j with a fresh set containing x. We can prove right-server privacy by induction: suppose that conditioned on Right's view so far, the client's hint table T contains $\sqrt{n}$ independent random sets (note that this is true at the beginning of the online phase). Then, we can argue that during the next query for x, $\mathsf{Resample}(S^*, x)$ is distributed as a fresh random set conditioned on Right's view so far; and moreover, at the end of the query, the client's hint table T is distributed as $\sqrt{n}$ independent random sets conditioned on Right's view so far.

Performance bounds. In the toy scheme, the bandwidth and server runtime are $O(\sqrt{n})$ for each online query. If the client adopts an efficient data structure for testing set membership, the client's runtime can also be upper bounded by $O(\sqrt{n})$ per query, but its storage is $O(n)$. We want to reduce the online bandwidth to polylogarithmic and reduce the client-side storage to sublinear, while preserving the $\widetilde{O}(\sqrt{n})$ online time for both the server and the client.

Strawman ideas for improving efficiency. A failed attempt to improve efficiency is the following. Let us generate each set using a pseudorandom function (PRF) rather than using true randomness. Specifically, we may assume that the $\mathsf{PRF}(\mathsf{sk}, \cdot)$ outputs a number in $[n]$, and an element $i \in \{0, 1, \ldots, n-1\}$ is con-

sidered in the set iff $\mathsf{PRF}(\mathsf{sk}, i) \in [1, \sqrt{n}]$. Moreover, sampling a pseudorandom set would boil down to sampling a fresh PRF secret key.

In this way, a pseudorandom set can be succinctly represented by a PRF secret key, and we can improve the client's storage to $\sqrt{n} \cdot \chi(\lambda)$ where $\chi(\lambda)$ is an upper bound on the length of the PRF key. During the online phase, the client needs to resample the set at the point x where $x \in \{0, 1, \ldots, n-1\}$ is the current query. If we could represent this locally resampled set succinctly too, then we can reduce the online bandwidth.

To achieve this, our idea is to adopt a Privately Puncturable PRF [7,9,14]. A Puncturable PRF is a PRF with the following additional functionality: given a point x and the secret key sk, one can call the $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{sk}, x)$ function to obtain a secret key sk_x that allows one to evaluate the PRF correctly at any point other than x. In an ordinary Puncturable PRF construction [30], using the punctured key sk_x to evaluate over the point x could result in an invalid symbol $\perp$. In contrast, a *Privately* Puncturable PRF allows one to remove a point x and obtain a punctured key sk_x; however, the punctured key sk_x does not disclose the point x. For sk_x to hide x, it must be that using sk_x to evaluate over the point x yields a non-$\perp$ outcome r. Not only so, imprecisely speaking, to a computationally bounded adversary, calling $\mathbf{Puncture}(\mathsf{sk}, x)$ should behave just like resampling the PRF's outcome at the point x.

If we use a Privately Puncturable PRF to construct a pseudorandom set like above, during each online query, we obtain a construct which we call a *Privately Puncturable Pseudorandom Set*. Generating a pseudorandom set is achieved by sampling a PRF key sk. Further, given a set represented by sk that contains a specific element x, one can perform a puncturing operation at x to derive a punctured secret key sk_x — this puncturing procedure acts as if we resampled the coins that determine whether x is in the set or not.

With such a Privately Puncturable Pseudorandom set, during each online query, the client can find a secret key sk from its table T that contains the queried element $x \in \{0, 1, \ldots, n-1\}$ (or sample a random sk containing x if not found), puncture the element x from the set sk, and send the punctured key sk_x to the right server. Similarly, to perform a refresh operation with the left server, the client simply samples a key sk' such that the associated set contains x, puncture x from sk', and send the resulting punctured key sk'_x to the left server. This approach allows us to compress the online bandwidth to $O(\chi(\lambda))$ bits per copy (and recall that there are $k = \omega(\log \lambda)$ parallel copies), where $\chi(\lambda)$ denotes the length of a punctured key.

Unfortunately, this idea completely fails because to generate the set from a secret key sk, the server would have to do a linear amount of work! This defeats our original goal of achieving sublinear online runtime.

Corrigan-Gibbs and Kogan's variant and why it fails too. At this point, we also briefly overview the approach of Corrigan-Gibbs and Kogan [20]. They adopt a different PRSet construction that indeed allows efficient set enumeration in roughly $\sqrt{n}$ (rather than linear in n) time. Unfortunately, their scheme does not offer a puncturing procedure that achieves any non-trivial efficiency;

thus in each online query, the client has to send an entire $(\sqrt{n}-1)$-sized set (rather than a punctured secret key) to each server. More specifically, Corrigan-Gibbs and Kogan [20] use a Pseudorandom Permutation (PRP) on the domain $\{0,1,\ldots,n-1\}$ to sample a pseudorandom set. A secret key sk of a PRP scheme defines a corresponding set $\{\mathsf{PRP}(\mathsf{sk},i)\}_{i\in\{0,1,\ldots,\sqrt{n}-1\}}$. Thus, the definition of the set itself gives an efficient set enumeration algorithm. To determine whether an element $x \in \{0,1,\ldots,n-1\}$ is in the set generated by sk, simply check whether $\mathsf{PRP}^{-1}(\mathsf{sk},x) \in \{0,1,\ldots,\sqrt{n}-1\}$. Their approach samples the set from a different distribution than our earlier strawman — in particular, the sampled set is of fixed size $\sqrt{n}$, and therefore x being in the set is not independent of whether $y \neq x$ is in the set (even when the PRP is replaced with a completely random permutation). For this reason, during the online phase, they adopt a slightly different approach than our earlier strawman: after finding a set either from the table T or freshly generated that contains the queried element x, they remove x from the set with high probability, but with a small probability, they remove a random element other than x. In this way, the right server sees a random set of size exactly $\sqrt{n}-1$, and the same applies to the left server.

The main problem with their approach is that it is not amenable to puncturing (with non-trivial efficiency). In fact, Boneh, Kim, and Wu proved the non-existence of Puncturable PRPs [8]. In our case, the domain size n is polynomially bounded, and even if we punctured a point x from the PRP, the adversary can easily recover $\mathsf{PRP}(\mathsf{sk},x)$ by evaluating $\mathsf{PRP}(\mathsf{sk},\cdot)$ at all other points.

To get around the non-existence of puncturable PRP barrier, one might be tempted to compute the pseudorandom set as $\{\mathsf{PRF}(\mathsf{sk},i)\}_{i\in\{0,1,\ldots,\sqrt{n}-1\}}$ instead, i.e., essentially the "dual" of our earlier PRF-based strawman scheme. While this approach allows for efficient set enumeration, it precludes efficient membership testing which, in our context, would make the client's online runtime linear.

3 Generalized Privately Puncturable Pseudorandom Set

To summarize the above discussion, we would like to construct a Privately Puncturable Pseudorandom Set (PRSet) scheme with some non-trivial security and efficiency requirements which we shall state shortly after defining the syntax:

- $(\mathsf{sk},\mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$: given the security parameter 1^λ and the universe size n, samples a secret key sk and a corresponding master secret key[6] msk;
- $S \leftarrow \mathbf{Set}(\mathsf{sk})$: a deterministic algorithm that outputs a set S given the secret key sk;
- $b \leftarrow \mathbf{Member}(\mathsf{sk},x)$: given a secret key sk and an element $x \in \{0,1,\ldots,n-1\}$, output a bit indicating whether $x \in \mathbf{Set}(\mathsf{sk})$; and

[6] The secret key sk is needed to enumerate the set, whereas the msk contains extra secret information needed for computing a punctured key. Jumping ahead, in our PIR scheme, the secret key sk can be sent to the server whereas the master secret key msk is kept secret by the client.

– $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$: given a master secret key msk and an element $x \in \{0, 1, \ldots, n-1\}$, outputs a secret key sk_x punctured at x.

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

Efficiency requirements. Our goal is to use the PRSet scheme to sample pseudorandom sets of size roughly $\sqrt{n}$. For efficiency, we want that enumerating the set can be accomplished with the $\mathbf{Set}(\mathsf{sk})$ algorithm, taking time roughly $\sqrt{n}$ (rather than linear in n). Additionally, we want that the membership test algorithm, i.e., $\mathbf{Member}(\mathsf{sk}, x)$, completes in polylogarithmic time.

3.1 Security Definitions

For security, we want the following:

1. **Pseudorandomness w.r.t. some distribution** $\mathcal{D}_n$: given a randomly sampled secret key $(\mathsf{sk}, \_) \leftarrow \mathbf{Gen}(1^\lambda, n)$, the associated set $\mathbf{Set}(\mathsf{sk})$ is computationally indistinguishable from a set sampled at random from some distribution $\mathcal{D}_n$ — we shall specify the distribution $\mathcal{D}_n$ later;
2. **Security w.r.t. puncturing I**: we want the following two distributions to be computationally indistinguishable for any $x \in \{0, 1, \ldots, n-1\}$:
   – Sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $\mathbf{Set}(\mathsf{sk})$ contains x, and output $\mathbf{Puncture}(\mathsf{msk}, x)$.
   – Sample $(\mathsf{sk}, \_) \leftarrow \mathbf{Gen}(1^\lambda, n)$ and output sk.

   The above definition says that a key punctured at any point is computationally indistinguishable from an unpunctured key, which implies that a punctured secret key should be simulatable without knowledge of the point x being punctured. In our PIR scheme, we only need the latter property, i.e., that a punctured key is simulatable without knowledge of the point being punctured — but we define this slightly stronger version for simplicity.
3. **Security w.r.t. puncturing II** (defined w.r.t. $\mathcal{D}_n$): we want the following two distributions to be computationally indistinguishable for any $x \in \{0, 1, \ldots, n-1\}$:
   – Sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $\mathbf{Set}(\mathsf{sk})$ contains x, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, and output $(\mathbf{Set}(\mathsf{sk}), x \in \mathbf{Set}(\mathsf{sk}_x))$ where "$x \in \mathbf{Set}(\mathsf{sk}_x)$" denotes the boolean predicate whether $x \in \mathbf{Set}(\mathsf{sk}_x)$.
   – Sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $\mathbf{Set}(\mathsf{sk})$ contains x, and output $(\mathbf{Set}(\mathsf{sk}), \mathsf{Bernoulli}(\rho))$ where $\rho := \Pr_{S \xleftarrow{\$} \mathcal{D}_n}[x \in S]$.

   Intuitively, the above says that knowing the unpunctured set reveals nothing about whether x still belongs to the set after puncturing x from the set.

Remark 2. Jumping ahead, the "security w.r.t. puncturing I" property will be used in proving the privacy of our PIR scheme, and the "security w.r.t. puncturing II" property will be needed for proving correctness — it turns out that the correctness proof is rather technical (see Section 4.4 for further discussions).

3.2 Defining Occasional Correctness

From the strawman attempts described in Section 2, we are essentially faced with the following dilemma. Consider some distribution $\mathcal{D}_n$ which the pseudorandom set tries to emulate. On one hand, we want each element to be included in the set with *independent* probability, since this would enable puncturing and efficient membership test. On the other hand, we do not want complete independence among elements, since it would preclude efficient set enumeration. It seems like we have hit a wall, but what comes to our rescue is the observation that our single-copy scheme need not guarantee $(1 - \mathsf{negl}(\lambda))$-correctness. Since we can take majority vote among $k = \omega(\log \lambda)$ parallel copies, it suffices for each copy to have 2/3-correctness. We therefore hope to seek middle ground between the seemingly conflicting requirements by relaxing correctness.

Informally speaking, we want the following notion of occasional correctness: with $1 - o(1)$ probability over the choice of a PRSet secret key that contains an element $x \in \{0, 1, \ldots, n-1\}$, puncturing at an arbitrary point x would *remove the point x from the set, and only x*. Recall that earlier, we said that puncturing at x should behave as if we resampled the choice whether x is in the set or not, independent of the unpunctured set (see "security w.r.t. puncturing II"). Thus, the relaxed correctness requirement intuitively implies that the resampling that happens at puncturing should only choose to include x in the punctured set with small (but possibly non-negligible) probability. Furthermore, jumping ahead, in our construction, puncturing at x may occasionally end up removing other elements besides x from the set, but this should not happen too often.

It turns out that to formally prove our PIR scheme secure, we actually need a more refined occasional correctness definition. Specifically, our formal definition lets us specify exactly which set of elements are related to x such that they might accidentally get evicted from the set due to the puncturing of x. Further, we also need to define an extra monotonicity condition that the puncturing operation never adds an element to the set.

Formally, we define occasional correctness as below.

Functionality preservation under puncturing. To define functionality preservation, we introduce a symmetric boolean predicate $\mathsf{Related}(x, y) : \{0, 1, \ldots, n-1\}^2 \to \{0, 1\}$, that outputs whether two elements x and y are related or not. We may assume that $\mathsf{Related}(x, y) = \mathsf{Related}(y, x)$.

We say that $\mathsf{PRSet} := (\mathbf{Gen}, \mathbf{Set}, \mathbf{Member}, \mathbf{Puncture})$ satisfies functionality preservation w.r.t. the $\mathsf{Related}$ predicate, iff for any $\lambda, n \in \mathbb{N}$, with probability $1 - \mathsf{negl}(\lambda)$ for some negligible function $\mathsf{negl}(\cdot)$, the following holds: let $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$, then, for any $x \in \mathbf{Set}(\mathsf{sk})$: let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$:

1. $\mathbf{Set}(\mathsf{sk}_x) \subseteq \mathbf{Set}(\mathsf{sk})$;
2. $\mathbf{Set}(\mathsf{sk}_x)$ runs in time no more than $\mathbf{Set}(\mathsf{sk})$;
3. for any $y \in \mathbf{Set}(\mathsf{sk}) \backslash \mathbf{Set}(\mathsf{sk}_x)$, it must be that $\mathsf{Related}(x, y) = 1$.

Intuitively, the above requires that puncturing results in a subset of the original set; and the set enumeration time can only reduce once a set has been punctured.

Moreover, puncturing x can only cause elements related to x to be removed from the set. Later on, when we instantiate the distribution $S \xleftarrow{\$} \mathcal{D}_n$ that the PRSet scheme tries to emulate, we shall see that *most elements in the sampled set S likely do not have other related elements in S.*

3.3 Choosing a Sampling Distribution

Recall that each element wants to decide at random whether to be included in the sampled set. Our idea is to allow weak dependence in the coins chosen by different elements. Such weak dependence should be sufficient to allow efficient set enumeration, and yet without destroying efficient membership tests. Of course, we have to pay a price for introducing the weak dependence among elements, and indeed we pay in terms of the correctness of puncturing. In our PRSet scheme, puncturing a secret key msk at a point x may, with some small but non-negligible probability over the choice of msk, not only cause the coins for x to be resampled, but also the coins for some elements other than x. When this happens, puncturing at the point x may end up removing other elements from the set, and possibly lead to an incorrect output in our single-copy PIR.

Even with this high-level intuition, identifying a construction that works is challenging. To this end, our approach is very remotely inspired by the line of work on designing block ciphers and format-preserving encryption [41, 46, 50]. Despite the remote reminiscence, of course, our problem definition and solutions are fundamentally different from block ciphers.

To convey the intuition, let us first describe the distribution our PRSet scheme aims to emulate, assuming the existence of a random oracle[7] $\mathsf{RO} : \{0,1\}^* \to \{0,1\}$. Suppose we sample an RO at random which will determine a pseudorandom set of expected size roughly $\sqrt{n}/\log^2 n$. To determine if an element $x \in \{0, 1, \ldots, n-1\}$ is in the set associated with RO or not, write x as a $\log n$-bit string, i.e., $x := \{0,1\}^{\log n}$. We then say that x is in the set iff using RO to "hash" every sufficiently long suffix of $0^{2\log\log n}||x$ outputs 1. More formally, set membership of $x \in \{0,1\}^{\log n}$ is defined with the following algorithm:

1. let $z := 0^B||x$, i.e., prepend $B := \lceil 2\log\log n \rceil$ number of 0s in front of the string x;
2. we say that x is in the set iff $\mathsf{RO}(z[i:]) = 1$ for every $i \in [1, \frac{1}{2}\log n + B]$, where $z[i:]$ denotes the suffix $z[i : \log n + B]$ starting at the index i. For example, $z[1:] = z$, $z[2:]$ is the string z removing the first bit, and so on.

Toy example. Figure 1 gives a toy example: suppose that $n = 4$, and thus $B = 2\log\log n = 2$, and $\frac{1}{2}\log n + B = 3$. Then, the string $x = 10$ is in the set iff $\mathsf{RO}(0010) = \mathsf{RO}(010) = \mathsf{RO}(10) = 1$.

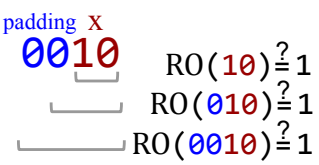

Fig. 1: A toy example.

The above sampling distribution has the following properties.

[7] Our final scheme does not need any random oracle, the RO is only for exposition.

Expected set size. Each $x \in \{0,1\}^{\log n}$ is included in the set with probability $2^{-(\frac{1}{2}\log n + B)} \approx 1/(\sqrt{n}\log^2 n)$, and the expected set size is roughly $\sqrt{n}/\log^2 n$.

Fast membership test. The definition itself gives a fast algorithm to test if an element $x \in \{0,1\}^{\log n}$ is in the set, by making $\frac{1}{2}\log n + B$ calls to RO.

Fast set enumeration. Enumerating all elements in the set can be accomplished by making roughly $\sqrt{n} \cdot \mathsf{poly}\log n$ calls to RO with at least $1 - o(1)$ probability. Let $\ell \geq \frac{1}{2}\log n + 1$, and let Z_ℓ be the set of all strings z of length exactly ℓ, such that using RO to "hash" all suffixes of z of length at least $\frac{1}{2}\log n + 1$ outputs 1. To enumerate the set generated by RO, we can start out $Z_{\frac{1}{2}\log n+1}$, which takes at most $2^{\frac{1}{2}\log n+1}$ RO calls to generate. Then, for each $\ell := \frac{1}{2}\log n + 2$ to $\frac{1}{2}\log n + B$, we will generate Z_ℓ from $Z_{\ell-1}$. This can be accomplished by enumerating all elements $z' \in Z_{\ell-1}$, and checking whether $\mathsf{RO}(0||z') = 1$ and $\mathsf{RO}(1||z') = 1$. In our online full version [47], we will prove that with at least $1 - o(1)$ probability, all the Z_ℓ sets encountered along the way will not exceed $\sqrt{n} \cdot \mathsf{poly}\log n$ in size. Thus, with $1 - o(1)$ probability, set enumeration can be accomplished by making at most $\sqrt{n} \cdot \mathsf{poly}\log n$ calls to RO.

Occasional correctness of "puncturing". Suppose that we sample an RO whose associated set contains the element $x \in \{0,1\}^{\log n}$. In this idealized world with RO, imagine that puncturing the point x from RO means that we resample the outcomes for $\mathsf{RO}((0^B||x)[i:])$ for every $i \in [1, \frac{1}{2}\log n + B]$. We want to make sure that with $1 - o(1)$ probability over the choice of the RO, puncturing the point x removes x and only x from the resulting set. We prove (a more refined version of) this statement in our online full version [47]. At a high level, to prove this statement, it suffices to prove that the expected number of related elements in the set is $o(1)$, where an element $x' \neq x$ is related to x, iff the longest common suffix of x and x' has length at least $\frac{1}{2}\log n + 1$.

3.4 Our PRSet Scheme

Given the above distribution $\mathcal{D}_n$, we can derive a PRSet scheme by replacing the RO with a privately puncturable PRF [7,9,14] — we review the formal definition for a privately puncturable PRF in the online full version [47]. Puncturing a point $x \in \{0,1\}^{\log n}$ simply punctures all queries we must make to the PRF to determine x's membership. For a punctured key to be indistinguishable from a freshly generated secret key, we puncture a set of "useless" points from a freshly generated secret key as well, since original keys and punctured keys may be trivially distinguishable in the the underlying privately puncturable PRF scheme. More formally, let $\mathsf{PRF} := (\mathbf{Gen}, \mathbf{Eval}, \mathbf{Puncture}, \mathbf{PEval})$ be a privately puncturable PRF scheme where **Eval** and **PEval** denote the evaluation algorithms using a normal key and a punctured key, respectively. Our PRSet scheme is described below:

Our PRSet scheme

- **Gen**$(1^\lambda, n)$: let $B := \lceil 2 \log \log n \rceil$,
  1. call PRF.**Gen** with the appropriate parameters to generate a normal PRF key sk'.
  2. let P be an arbitrary set of $\frac{1}{2} \log n + B$ distinct strings in $\{0,1\}^{\log n + B}$ that begin with the bit 1;
  3. call $\mathsf{sk} \leftarrow$ PRF.**Puncture**(sk', P), and output $(\mathsf{sk}, \mathsf{msk} = \mathsf{sk}')$.
- **Set**(sk): similar to the earlier set enumeration algorithm for the distribution $\mathcal{D}_n$, but replace $\mathsf{RO}(\cdot)$ calls with calls to PRF.**PEval**$(\mathsf{sk}, \cdot)$ instead;
- **Member**(sk, x):
  1. write $x \in \{0,1\}^{\log n}$ as a binary string, and let $z := 0^B || x$;
  2. if for every $1 \le i \le \frac{1}{2} \cdot \log n + B$, PRF.**PEval**$(\mathsf{sk}, z[i:]) = 1$, then output 1; else output 0.
- **Puncture**(msk, x):
  1. write $x \in \{0,1\}^{\log n}$ as a binary string, and let $z := 0^B || x$;
  2. let $P := \{z[i:]\}_{i \in [1, \frac{1}{2} \cdot \log n + B]}$ and $\mathsf{sk}_P \leftarrow$ PRF.**Puncture**(msk, P); output sk_P.

Performance bounds. Our privately puncturable PRF scheme must support puncturing $O(\log n)$ many points. As stated in the online full version [47], we can construct such a privately puncturable PRF with $\widetilde{O}(1)$ runtime for **Gen**, **Eval**, and **PEval**, and moreover, each punctured key is of length $\widetilde{O}(1)$. Using such a privately puncturable PRF, our resulting PRSet scheme achieves $\widetilde{O}(1)$ time for PRSet.**Gen**, PRSet.**Member**, and PRSet.**Puncture** operations, and the expected runtime for PRSet.**Set** is $\widetilde{O}(\sqrt{n})$. Bounding the runtime of PRSet.**Set** will require a probabilistic analysis of the distribution $\mathcal{D}_n$, which we defer to the online full version [47].

We also defer to Section 5 a detailed proof of security for our PRSet scheme.

4 Putting it All Together: Our PIR Scheme

4.1 Definitions: Two-Server Preprocessing PIR

In our definition below, the two servers are treated as stateful algorithms Left and Right, respectively (but in our construction, the only state they need to store is the database itself). The client is treated as a stateful algorithm denoted Client. Initially, all of Client, Left, and Right receive the parameters 1^λ and n.

- **Offline setup.** Client receives nothing and each of Left and Right receives the same database $\mathsf{DB} \in \{0,1\}^n$ as input. Client sends a single message to Left, and Left responds with a single message often called a *hint*.
- **Online queries.** The following can be repeated for a priori-unknown polynomially many steps. Upon receiving an index $x \in \{0, 1, \ldots, n-1\}$ to query,

Client sends a single message to Left and a single message to Right. It receives a single response from each server Left and Right. Client then performs some computation and outputs an answer $\beta \in \{0,1\}$.

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

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

Privacy. For privacy, we require the following.

- *Left-server privacy.* There is a probabilistic polynomial time (p.p.t.) stateful simulator Sim, such that for any arbitrary (even computationally unbounded) algorithm Right^*, for any non-uniform p.p.t. adversary $\mathcal{A}$, $\mathcal{A}$'s views in the Real and Ideal experiments are computationally indistinguishable:
  1. Real: The honest Client interacts with $\mathcal{A}$ who acts as the left server and may deviate arbitrarily from the prescribed protocol, and an arbitrary (even computationally unbounded) algorithm Right^* acting as the right server. In every online step t, $\mathcal{A}$ adaptively chooses the next query $x_t \in \{0, 1, \ldots, n-1\}$, and Client is invoked with x_t.
  2. Ideal: The simulated client Sim interacts with $\mathcal{A}$ who acts as the left server, and an arbitrary (even computationally unbounded) algorithm Right^* acting as the right server. In every online step t, $\mathcal{A}$ adaptively chooses the next query $x_t \in \{0, 1, \ldots, n-1\}$, and Sim is invoked without receiving x_t.
- *Right-server privacy.* Right-server privacy is defined in a symmetric way as above by exchanging left and right.

Intuitively, the above privacy definition requires that any single server alone cannot learn anything about the client's queries; further, this must hold even when both servers can behave maliciously. However, recall that we do not guarantee correctness if one or both server(s) fail to respond correctly.

4.2 Construction

We describe our PIR scheme below, where Client, Left, Right denote the client, the left server, and the right server, respectively.

Our PIR Scheme

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

Offline setup. For $i = 1$ to $\mathsf{lenT} := 6\sqrt{n} \cdot \log^3 n$ in parallel:

1. Client: Sample $(\mathsf{sk}_i, \mathsf{msk}_i) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$, send sk_i to Left.
2. Left: Run $S_i \leftarrow \mathsf{PRSet}.\mathbf{Set}(\mathsf{sk}_i)$. If the runtime of $\mathsf{PRSet}.\mathbf{Set}(\mathsf{sk}_i)$, measured in terms of $\mathsf{PRF}.\mathbf{PEval}$ calls, exceeds $\mathsf{maxT} := 6\sqrt{n}\log^5 n$, return

$p_i := 0$ to Client. Else, return the parity bit $p_i \in \{0,1\}$ of the set S_i to Client.

3. Client: Save $T_i := (\mathsf{sk}_i, \mathsf{msk}_i, p_i)$ where $T := (T_1, T_2, \ldots, T_{\mathsf{lenT}})$ denotes a table saved by Client.

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

– **Query** ($\mathsf{Client} \Leftrightarrow \mathsf{Right}$):

1. Client:
   (a) Sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$, append $(\mathsf{sk}, \mathsf{msk}, 0)$ to the end of the table T. $(\star)$
   (b) Henceforth parse $T_i := (\mathsf{sk}_i, \mathsf{msk}_i, p_i)$. Let j be the smallest entry in the table T such that $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_j, x) = 1$.
   (c) Call $\widetilde{\mathsf{sk}}_j := \mathsf{PRSet}.\mathbf{Puncture}(\mathsf{msk}_j, x)$. Send $\widetilde{\mathsf{sk}}_j$ to Right.
2. Right: Run $S \leftarrow \mathsf{PRSet}.\mathbf{Set}(\widetilde{\mathsf{sk}}_j)$. If the runtime exceeds maxT, return $p := 0$ to Client. Else, return the parity bit $p \in \{0,1\}$ of the set S to Client.
3. Client: Let $\beta' := p \oplus p_j$ be a candidate answer of this copy. Let β be the majority vote among the candidate answers of all k copies.

– **Refresh** ($\mathsf{Client} \Leftrightarrow \mathsf{Left}$):

1. Client:
   (a) Sample a new $(\mathsf{sk}', \mathsf{msk}') \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to the constraint $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}', x) = 1$. $(\star)$
   (b) Call $\mathsf{sk}'_x \leftarrow \mathsf{PRSet}.\mathbf{Puncture}(\mathsf{msk}', x)$, and send sk'_x to Left.
2. Left: Run $S \leftarrow \mathsf{PRSet}.\mathbf{Set}(\mathsf{sk}'_x)$. If the runtime exceeds maxT, return $p := 0$ to Client. Else, return the parity bit $p \in \{0,1\}$ of the set S to Client.
3. Client: Replace $T_j := (\mathsf{sk}', \mathsf{msk}', p \oplus \beta)$. Finally, remove the last entry from the table T.

Remark. To obtain *deterministic* performance bounds, we can have the client run the Step 1(a) of both the Query and Refresh phases, marked with $(\star)$, at the very beginning of each online query, and simply abort if the number of tries exceeds maxT — in this case, no message is sent and the client outputs a canonical bit 0 as the candidate answer. It is not hard to see that this change does not affect the privacy proof and adds only $o(1)$ correctness failure probability for each instance per query.

Intuitively, the idea here is to summarize the random sets in the earlier toy scheme with the keys of a PRSet scheme, i.e., the client stores the lenT number of keys to represent lenT sets; moreover, the client sends punctured keys to the servers rather than the full sets. By construction, in each copy, the client always obtains answers from the respective servers during the query and refresh phases, but the answers may be incorrect with some small probability. The $k = \omega(\log \lambda)$ parallel repetions make the overall error probability negligibly small.

Specifically, the answer from the right-server during the query phase may be incorrect if 1) the queried index x is not found in the lenT sets stored by the client; 2) x is found to be in some set represented by $(\mathsf{sk}_j, \mathsf{msk}_j)$, but the parity bit stored by the client is incorrect (see why the refresh phase may cause error shortly); 3) puncturing the key msk_j does not result in exactly the set $\mathbf{Set}(\mathsf{sk}_j) \backslash \{x\}$; or 4) the right server exceeds maxT set enumeration time.

The refresh phase can incur error with small probability too, and thus cause the client to store an incorrect parity bit for the refreshed set. Recall that during refresh, the client computes the parity bit of a newly refreshed set as $\beta + p$ where β is the client's belief of the answer to the present query, and p is the answer returned by the left server. If either β or p is wrong, the refreshed parity bit may be incorrect. Specifically, p can be wrong if the left server exceeded maxT set enumeration time. Moreover, if the punctured key sk'_x does not give exactly $\mathbf{Set}(\mathsf{sk}') \backslash \{x\}$, then the parity p returned by the left server could be incorrect.

4.3 Performance Analysis

For our performance analysis, we will assume that Step 1(a) of both the Query and Refresh phases, marked $(\star)$ in the PIR scheme, are capped at maxT runtime, since this will give us deterministic performance bounds — see the remark at the end of the PIR algorithm.

We now analyze the performance bounds for each instance of PIR — keep in mind that our final scheme involves $k = \omega(\log \lambda) = \widetilde{O}(1)$ parallel instances. Our analysis below also shows how to translate the runtime of the underlying PRSet scheme to the runtime of the resulting PIR scheme. Specifically, we will use our PRSet scheme whose performance bounds are stated in Section 3.4.

- *The offline bandwidth and client computation are $\widetilde{O}(\sqrt{n})$, the offline server computation is $\widetilde{O}(n)$.* The offline client computation is dominated by running $\mathsf{PRSet}.\mathbf{Gen}$ for $\mathsf{lenT} = \widetilde{O}(\sqrt{n})$ number of times; the bandwidth is dominated by transmitting lenT number of PRSet keys to the server and then for the server to transmit 1 parity bit back for each of the lenT keys; and the server computation is dominated by running the $\mathsf{PRSet}.\mathbf{Set}$ algorithm for lenT number of times, where each $\mathsf{PRSet}.\mathbf{Set}$ call is capped at $\mathsf{maxT} = \widetilde{O}(\sqrt{n})$ runtime.
- *The online server and client runtime is $\widetilde{O}(\sqrt{n})$, and the online bandwidth is $\widetilde{O}(1)$.* Specifically, during the "Query" phase, the client's runtime is bounded by the following: Step 1(a) is capped at maxT calls to $\mathsf{PRSet}.\mathbf{Gen}$ and $\mathsf{PRSet}.\mathbf{Member}$; Step 1(b) involves running $\mathsf{PRSet}.\mathbf{Member}$ at most lenT number of times; the runtime of Step 1(c) and Step 3 is dominated by other steps. During the "Refresh" phase, the client's runtime involves the following: Step 1(a) is capped at maxT calls to $\mathsf{PRSet}.\mathbf{Gen}$ and $\mathsf{PRSet}.\mathbf{Member}$, and the runtime of Step 1(b) and 3 is dominated by Step 1(a). Both the left and right server's runtime includes a single call to $\mathsf{PRSet}.\mathbf{Set}$ capped at maxT, and the cost of computing the parity of at most maxT number of bits. The online bandwidth involves the client sending a single PRSet key to each of the left and right server, and each server sending back one bit.

4.4 Proof Roadmap

Proving our PIR scheme secure turns out to be very much non-trivial. Somewhat surprisingly at first, the most challenging part is actually the proof of occasional correctness of the single-copy version of our PIR scheme (see Sections 5 and the online full version [47]) — even though we are only asking for a relaxed correctness requirement. At a high level, the challenge arises from the fact that the distribution of the PRSet key sk becomes *skewed*, when conditioning on the fact that the key sk is chosen during the online query phase, since it is the first entry in the client's hint table T that contains the queried element x. In one part of the occasional correctness proof, we need to argue that, imprecisely speaking, despite this skewed distribution, the selected secret key can provide a correct answer to the present query with $1-o(1)$ probability. In a key technical step, we make a *stochastic domination* type of argument that roughly speaking, proves the following: conditioned on the secret key not having been consumed so far and now being consumed by the present query, it makes it less likely, in comparison with a freshly generated PRSet key, for certain bad events to happen that might lead to incorrectness. To make this argument work, we rewrite the randomized experiment into an equivalent one where the sampling of a subset of the random coins is delayed to the point when they are consumed. In our scheme, multiple bad events can lead to incorrectness of a single copy of the scheme. Therefore, in our proof, we bound the probability of each of these bad events (see the appendices) — to do so, we often view the randomized experiment in different lights, to facilitate the analyses of different bad events.

5 Proofs for our PRSet Scheme

Lemma 1 (Correctness, pseudorandomness, and functionality preservation under puncturing). *The above* PRSet *construction satisfies correctness. Further, suppose that the* PRF *scheme satisfies pseudorandomness; then the* PRSet *scheme also satisfies pseudorandomness and functionality preservation under puncturing.*

Proof. Correctness follows directly from the construction. Pseudorandomness relies on the pseudorandomness of the PRF through a straightforward reduction. To see functionality preservation, let $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, and below we may ignore the negligible probability event that the underlying puncturable PRF violates its functionality preservation property. Notice that for every string z that is a suffix of $0^B||x$ of length at least $\frac{1}{2}\log n+1$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, z) = 1$, but there may exist such z where $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}_x, z)$ becomes 0 instead. For any string z that is not a suffix of $0^B||x$ of length at least $\frac{1}{2}\log n + 1$, $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, z) = \mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}_x, z)$. Given the above observation, "functional preservation under puncturing" is easy to verify.

Lemma 2 (Security w.r.t. puncturing). *Suppose that the* PRF *scheme satisfies pseudorandomness and privacy w.r.t. puncturing as defined in the online*

full version [47]. Then, the above PRSet *construction satisfies security w.r.t. puncturing.*

Proof. We need to prove two properties.

First property. We begin by proving the first property, that is, the following distributions are computationally indistinguishable for any $x \in \{0, 1, \ldots, n-1\}$:

- Expt_0: Repeat $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $x \in \mathbf{Set}(\mathsf{sk})$, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, output sk_x.
- Expt_1: $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ and output sk.

We define an intermediate hybrid experiment Hyb: sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, and output sk_x.

Claim 1 *Suppose that the puncturable* PRF *satisfies pseudorandomness as defined in the online full version [47]. Then,* Expt_0 *and* Hyb *are computationally indistinguishable.*

Proof. Suppose that there is an efficient adversary $\mathcal{A}$ that can distinguish Expt_0 and Hyb with non-negligible probability. We can construct an efficient reduction $\mathcal{B}$ that breaks the pseudorandomness of the PRF scheme.

Let P_x denote the set containing the $m = \frac{1}{2}\log n + B$ queries we need to make to determine whether x is in the set. $\mathcal{B}$ asks its own challenger denoted $\mathcal{C}$ for a PRF key punctured at P_x, and it obtains sk_{P_x}. It forwards sk_{P_x} to $\mathcal{A}$. $\mathcal{B}$ then obtains a vector of bits denoted $\boldsymbol{\beta} := (\beta_1, \ldots, \beta_m)$ as the purported outcomes for $\{\mathbf{Eval}(\mathsf{sk}, y)\}_{y \in P_x}$. If $\boldsymbol{\beta} = \mathbf{1}$, then $\mathcal{B}$ outputs whatever $\mathcal{A}$ outputs. Else, it outputs a random bit.

Case 1. If the challenger $\mathcal{C}$ is using real values for $\{\mathbf{Eval}(\mathsf{sk}, y)\}_{y \in P_x}$, then $\mathcal{A}$'s view is identically distributed as Expt_0. The probability that $\mathcal{B}$ outputs 1 is

$$p := \Pr[\mathcal{A}(\mathsf{Expt}_0) = 1] \cdot \Pr[\boldsymbol{\beta} = \mathbf{1}] + \frac{1}{2} \cdot \Pr[\boldsymbol{\beta} \neq \mathbf{1}]$$

Case 2. If the challenger $\mathcal{C}$ is using random values in place of $\{\mathbf{Eval}(\mathsf{sk}, y)\}_{y \in P_x}$, then $\mathcal{A}$'s view is identically distributed as Hyb. In this case, the probability that $\mathcal{B}$ outputs 1 is equal to

$$p' := \Pr[\mathcal{A}(\mathsf{Hyb}) = 1] \cdot \Pr[\boldsymbol{\beta} = \mathbf{1}] + \frac{1}{2} \cdot \Pr[\boldsymbol{\beta} \neq \mathbf{1}]$$

Note that in Case 1, $|\Pr[\boldsymbol{\beta} = \mathbf{1}] - 1/2^m| \leq \mathsf{negl}(\lambda)$ due to the pseudorandomness of the PRF; and in Case 2 $\Pr[\boldsymbol{\beta} = \mathbf{1}] = 1/2^m$. Moreover, $1/2^m$ is non-negligible due to the choice of m. Therefore, if $|\Pr[\mathcal{A}(\mathsf{Expt}_0) = 1] - \Pr[\mathcal{A}(\mathsf{Hyb}) = 1]|$ is non-negligible, then $|p - p'|$ would be non-negligible, too.

Claim 2 *Suppose that the puncturable* PRF *satisfies privacy w.r.t. puncturing as defined in the online full version [47]. Then,* Hyb *is computationally indistinguishable from* Expt_1.

Proof. Follows from a straightforward reduction to the privacy w.r.t. puncturing property of the PRF.

The computational indistinguishability of Expt_0 and Expt_1 now follows from Claim 1 and Claim 2.

Second property. We next prove the second property, that is, we want to show that the following two distributions are computationally indistinguishable:

- Expt_0^*: Repeat $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $x \in \mathbf{Set}(\mathsf{sk})$, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, and output $(\mathbf{Set}(\mathsf{sk}), x \in \mathbf{Set}(\mathsf{sk}_x))$ where $x \in \mathbf{Set}(\mathsf{sk}_x)$ denotes a boolean predicate.
- Expt_1^*: Repeat $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $x \in \mathbf{Set}(\mathsf{sk})$, and output $(\mathbf{Set}(\mathsf{sk}), \mathsf{Bernoulli}(\rho))$ where $\rho := 2^{-(\frac{1}{2}\log n + B)}$.

Let $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $x \in \mathbf{Set}(\mathsf{sk})$, and let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$. Observe that there is a deterministic, polynomial-time function $\mathbf{Reconstruct}$ such that $\mathbf{Reconstruct}(\mathsf{sk}_x, x) = \mathbf{Set}(\mathsf{sk})$. Essentially, $\mathbf{Reconstruct}$ uses answers to $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}_x, \cdot)$ calls to determine set membership, except that when encountering any string z that is a suffix of $0^B||x$ of length at least $\frac{1}{2}\log n + 1$, override the outcome of $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}_x, z)$ to 1.

We can therefore rewrite Expt_0^* as the following experiment Hyb: repeat $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathbf{Gen}(1^\lambda, n)$ until $x \in \mathbf{Set}(\mathsf{sk})$, let $\mathsf{sk}_x \leftarrow \mathbf{Puncture}(\mathsf{msk}, x)$, and output $(\mathbf{Reconstruct}(\mathsf{sk}_x, x), x \in \mathbf{Set}(\mathsf{sk}_x))$.

Due to the first property which we just proved, the above distribution Hyb is computationally indistinguishable from the following Hyb': $(\mathsf{sk}, \_) \leftarrow \mathbf{Gen}(1^\lambda, n)$, and output $(\mathbf{Reconstruct}(\mathsf{sk}, x), x \in \mathbf{Set}(\mathsf{sk}))$.

Now, consider the experiment Ideal which is defined just like in Hyb, except that sampling a PRF secret key is replaced with sampling an RO, and to determine set membership, any call to $\mathsf{PRF}.\mathbf{PEval}(\mathsf{sk}, \cdot)$ is replaced with $\mathsf{RO}(\cdot)$. In Ideal, $\mathbf{Reconstruct}(\mathsf{RO}, x)$ does not need to look at the coins that determine x's membership in the set. Based on this observation as well as the pseudorandomness of the underlying PRF, we conclude that Ideal is computationally indistinguishable from Expt_1^*.

Deferred contents. We defer the probabilistic analysis of the distribution $\mathcal{D}_n$, and proofs for the runtime of set enumeration to the online full version [47].

6 Proofs for our PIR Scheme

Single-copy varirant of our PIR scheme. In our proofs, we consider a single-copy variant of our PIR scheme. In the single-copy scheme, we set the number of parallel instances $k := 1$. Further, we imagine that the true answer β is obtained from some true-answer oracle rather than taking majority vote.

6.1 Privacy Proof

We focus on the single-copy variant, and prove its privacy.

Theorem 2 (Left-server privacy). *Suppose that the* PRSet *scheme satisfies (the first property in) "security w.r.t. puncturing". Then, the single-copy scheme satisfies left-server privacy.*

Proof. We define the following simulator Sim which fully specifies the ideal experiment Ideal:

- *Offline setup.* For $i = 1$ to $\mathsf{lenT} := 6\sqrt{n} \cdot \log^3 n$: sample $(\mathsf{sk}_i, \mathsf{msk}_i) \leftarrow \mathsf{PRSet.}\mathbf{Gen}(1^\lambda, n)$ and send $\{\mathsf{sk}_i\}_{i \in [1, \mathsf{lenT}]}$ to $\mathcal{A}$ acting as the left server.
- *Online queries.* For each online query, sample $(\mathsf{sk}', \mathsf{msk}') \leftarrow \mathsf{PRSet.}\mathbf{Gen}(1^\lambda, n)$ and send sk' to $\mathcal{A}$ acting as the left server.

The computational indistinguishability of $\mathcal{A}$'s views in Real and Ideal follow due to a straightforward hybrid argument relying on the "security w.r.t. puncturing" property of the PRSet scheme. Specifically, let Q be the total number of queries in the online phase. We define a sequence of hybrid experiments $\{\mathsf{Hyb}_i\}_{i \in \{0,1,\ldots,Q\}}$, where in Hyb_i, during the first i online steps, $\mathcal{A}$ (acting as the left server) receives a message constructed like in Ideal, and during the remaining $Q - i$ online steps, $\mathcal{A}$ receives a message constructed like in Real. Clearly, $\mathsf{Hyb}_0 = \mathsf{Real}$ and $\mathsf{Hyb}_Q = \mathsf{Ideal}$. It suffices to show that any two adjacent hybrid experiments are computationally indistinguishable, and this follows due to a straightforward reduction to the "security w.r.t. puncturing" property of the PRSet scheme.

Theorem 3 (Right-server privacy). *Suppose that the* PRSet *scheme satisfies (the first property in) security w.r.t. puncturing. Then, the single-copy scheme satisfies right-server privacy.*

Proof. We define the following simulator Sim which fully specifies the ideal experiment Ideal:

- *Offline setup.* $\mathcal{A}$, acting as the right server, receives nothing.
- *Online queries.* For each online query, sample $(\mathsf{sk}', \mathsf{msk}') \leftarrow \mathsf{PRSet.}\mathbf{Gen}(1^\lambda, n)$ and send sk' to $\mathcal{A}$ acting as the right server.

We now need to argue that any non-uniform p.p.t.$\mathcal{A}$'s views in Real and Ideal are computationally indistinguishable.

Real^*. First, we consider the following experiment Real^*.

- *Offline setup.* For each $i \in [1, \mathsf{lenT}]$, Client samples $(\mathsf{sk}_i, \mathsf{msk}_i) \leftarrow \mathsf{PRSet.}\mathbf{Gen}(1^\lambda, n)$, and lets $T_i := (\mathsf{sk}_i, \mathsf{msk}_i)$. The adversary $\mathcal{A}$, acting as the right server, receives nothing.
- *Online queries.* For each online query $x \in \{0, 1, \ldots, n-1\}$:
  a) Client samples $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathsf{PRSet.}\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet.}\mathbf{Member}(\mathsf{sk}, x) = 1$, and appends $(\mathsf{sk}, \mathsf{msk})$ to the end of the table T.

b) Client finds the smallest entry $T_j := (\mathsf{sk}_j, \mathsf{msk}_j)$ in T such that $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_j, x) = 1$. It sends $\mathsf{PRSet}.\mathbf{Puncture}(\mathsf{msk}_j, x)$ to $\mathcal{A}$ acting as the right server.
c) Client samples $(\mathsf{sk}', \mathsf{msk}') \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$, overwrites T_j with $(\mathsf{sk}', \mathsf{msk}')$, and removes the last entry from T.

Real^* is just a rewrite of Real throwing away terms that we do not care. Let $\mathsf{View}^{\mathsf{Real}}$ and $\mathsf{View}^{\mathsf{Real}^*}$ denote $\mathcal{A}$'s view and the client's table T (truncating the third field of each entry in Real) at the beginning of each online query, in the experiments Real and Real^*, respectively. We have that even for a computationally unbounded $\mathcal{A}$, $\mathsf{View}^{\mathsf{Real}}$ and $\mathsf{View}^{\mathsf{Real}^*}$ are identically distributed.

Fact 1 *In Real^*, for every online step t, even if $\mathcal{A}$ is computationally unbounded, and even when conditioned on $\mathcal{A}$'s view over the first $t-1$ steps,*

- *let $x \in \{0, 1, \ldots, n-1\}$ be the t-th online query, the message $\mathcal{A}$ receives in the t-th query is distributed as: sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$ and output $\mathsf{PRSet}.\mathbf{Puncture}(\mathsf{sk}, x)$;*
- *at the end of the t-th online query, the client's table T is a fresh uniform sample from $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)^{\mathsf{lenT}}$ independent of the message $\mathcal{A}$ receives during the t-th query, i.e., T contains a sample of lenT uniform, independent entries from the distribution $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$.*

Proof. We can prove by induction.

Base case. At the end of the offline phase (henceforth also called the 0-th query), indeed the client's table T is a uniform sample from the distribution $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)^{\mathsf{lenT}}$.

Inductive step. Suppose that at the end of the t-th step, the client's table T is uniform sample from the distribution $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)^{\mathsf{lenT}}$ even when conditioned on $\mathcal{A}$'s view in the first t steps. We now prove that the stated claims hold for $t+1$. Let $x \in \{0, 1, \ldots, n-1\}$ be the query made in online step $t+1$, the choice of x depends only on $\mathcal{A}$'s view in the first t online queries. Henceforth, for $i \in [1, \mathsf{lenT}]$, let $\alpha_{i,x}$ be the probability that in a random sample from the distribution $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)^{\mathsf{lenT}}$, the first entry that contains x is i. Let $\alpha_{\mathsf{lenT}+1,x} := 1 - \sum_{i=1}^{\mathsf{lenT}} \alpha_{i,x}$.

Consider the following experiment Expt:

- Client samples an index $u \in [\mathsf{lenT}+1]$ such that $u = i$ with probability $\alpha_{i,x}$.
- $\forall j < u$, Client samples $T_j := (\mathsf{sk}_j, \mathsf{msk}_j) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_j, x) = 0$.
- For u, Client samples $(\mathsf{sk}, \mathsf{msk})$ and $(\mathsf{sk}', \mathsf{msk}')$ independently from the distribution $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}_j, x) = 1$. It sends $\mathsf{PRSet}.\mathbf{Puncture}(\mathsf{sk}, x)$ to $\mathcal{A}$ and saves $T_u := (\mathsf{sk}', \mathsf{msk}')$.
- $\forall j \in [u+1, \mathsf{lenT}+1]$, Client samples $T_j := (\mathsf{sk}_j, \mathsf{msk}_j) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$.
- Finally, Client removes last entry from T.

Let $\mathsf{View}_{t+1}^{\mathsf{Real}^*}$ be the message $\mathcal{A}$ receives during the $(t+1)$-st query as well as the client's table T at the end of the $(t+1)$-st query in Real^*. Let $\mathsf{View}^{\mathsf{Expt}}$ be the message $\mathcal{A}$ receives as well as the client's table T at the end in Expt. Suppose that the induction hypothesis holds, then it is not hard to see that $\mathsf{View}^{\mathsf{Expt}}$ is identically distributed as $\mathsf{View}_{t+1}^{\mathsf{Real}^*}$ even when conditioning on the view of $\mathcal{A}$ in the first t queries in Real^*, and even when $\mathcal{A}$ is computationally unbounded.

In the experiment Expt, it is not hard to see the distribution $\mathsf{View}^{\mathsf{Expt}}$ is the following: T is sampled at random from $\mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)^{\mathsf{lenT}}$, and $\mathcal{A}$'s received message is distributed as: sample $(\mathsf{sk}, \mathsf{msk}) \leftarrow \mathsf{PRSet}.\mathbf{Gen}(1^\lambda, n)$ subject to $\mathsf{PRSet}.\mathbf{Member}(\mathsf{sk}, x) = 1$ and output $\mathsf{PRSet}.\mathbf{Puncture}(\mathsf{sk}, x)$.

Given Fact 1, we can prove that any non-uniform p.p.t. $\mathcal{A}$'s views in Ideal and Real^* are computationally indistinguishable through a standard hybrid argument, relying on the the "security w.r.t. puncturing" property of the PRSet scheme — the hybrid sequence is similar to the proof of Theorem 2.

6.2 Correctness Proof

Deferred to the online full version [47].

7 Additional Related Work

Beimel et al. [6] proved that in the original formulation of PIR, the servers must collectively probe all n bits of the database on average to respond to a client's query. Various techniques have been suggested to overcome this key performance bottleneck, e.g., encoding the server-side database, storing per-client or even per-query server state, batching, introducing assumptions like Virtual-Blackbox obfuscation which is known to be impossible [5], or having many non-colluding servers. We review this line of work below.

As mentioned in Section 1, the work of Beimel et al. achieves sublinear online computation by encoding the database into a $n^{1+\epsilon}$ to $\mathsf{poly}(n)$-sized string. The recent (designated-client) doubly efficient PIR schemes [11, 15] rely on encoding the database as well as having the server store $\Omega(n)$ state per client, which is a significant barrier towards practicality in our motivating applications. Boyle et al. [11] show that assuming Virtual-Blackbox Obfuscation which is known to be impossible [5] (and additional non-standard assumptions that are not yet well understood), one can indeed construct a preprocessing PIR with n^ϵ online runtime and bandwidth, without having to store per-client state at the server.

A related notion called *private anonymous data access* (PANDA) was recently introduced by Hamlin et al. [33]. PANDA is a form of preprocessing PIR which requires a *third-party trusted setup* besides the client and the servers (which is not necessary in our work); and moreover, the server storage and time grow w.r.t. the number of corrupt clients. In our motivating examples, the number of clients is essentially unbounded which makes known PANDA schemes unsuitable. Some works [6, 23] suggested having the server store *per-query* state to reduce

the online time. Specifically, the construction by Beimel et al. [6] requires a linear amount of server storage per query, and this is even worse than per-client storage. Other works [43] improve the online time by making the number of public-key operations sublinear, along with a still *linear* number of symmetric-key operations. Sharding has also been suggested to spread out the server work online [21] but the total work across all servers is still linear.

A couple of works [38, 44] construct preprocessing PIR schemes whose online runtime is marginally sublinear, e.g., roughly $O(n/\log n)$; and the complexity of these protocols is much larger than Corrigan-Gibbs and Kogan [20].

An elegant line of work suggested batching queries from the same client [3, 4, 32, 34] or among multiple clients [6, 35, 40] to amortize the linear server work among the batch. Our formulation can be viewed as a generalization of batched PIR, since we do not require the requests to come in a batch, and we can nonetheless achieve small online bandwidth and work. The work by Beimel et al. [6] also showed how to get a preprocessing PIR with polylogarithmic online bandwidth and cost assuming polylogarithmically many non-colluding servers, and $\mathsf{poly}(n)$ server space. Toledo et al. [51] consider how to relax the security definition and achieve differential-privacy-style security, to improve the server time to sublinear.

The concurrent of Kogan and Corrigan-Gibbs [36] gives a practical instantiation of their earlier work [20], with a clever trick to remove the k-fold parallel repetition. Their implementation is indeed in the unbounded query setting. For their particular application, i.e., private blocklist, it turns out that the dabase is somewhat small, and therefore, they are willing to incur $\Theta(n)$ computation per online query, in exchange for roughly $O(\sqrt{n})$ online time and logarithmic bandwidth. While their implementation is indeed a practical sweetspot for the private blocklist application, for larger databases, incurring linear client time per online query could be prohibitive. Their trick to remove the k-fold repetition does not seem to immediately apply to our construction because we have an additional source of error from our underlying PRSet scheme.

Deferred Contents

In the interest of space, we defer additional details and proofs to the online full version [47].

Acknowledgments

This work is in part supported by a DARPA SIEVE grant under a subcontract from SRI, an ONR YIP award, a Packard Fellowship, a JP Morgan Award, and NSF grants under the award numbers 2001026, 1901047, and 1763742. The authors would like to acknowledge Dima Kogan and Feng-Hao Liu for helpful discussions, and thank the anonymous reviewers for the detailed and throughtful comments.

References

1. Oblivious dns over https. https://tools.ietf.org/html/draft-pauly-dprive-oblivious-doh-04.
2. Private information retrieval with sublinear online time. Talk by Dmitry Kogan at Charles River Crypto Day, 2021.
3. A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication–computation trade-offs in pir. Cryptology ePrint Archive, Report 2019/1483, 2019. https://eprint.iacr.org/2019/1483.
4. S. Angel, H. Chen, K. Laine, and S. T. V. Setty. PIR with compressed queries and amortized query processing. In *S&P*, 2018.
5. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. In *CRYPTO*, 2001.
6. A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *CRYPTO*, pages 55–73, 2000.
7. D. Boneh, S. Kim, and H. W. Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, pages 415–445, 2017.
8. D. Boneh, S. Kim, and D. J. Wu. Constrained keys for invertible pseudorandom functions. In *TCC*, 2017.
9. D. Boneh, K. Lewi, and D. J. Wu. Constraining pseudorandom functions privately. In *PKC*, 2017.
10. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.
11. E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In *TCC*, 2017.
12. Z. Brakerski, R. Tsabary, V. Vaikuntanathan, and H. Wee. Private constrained prfs (and more) from LWE. In *TCC*, 2017.
13. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, pages 402–414, 1999.
14. R. Canetti and Y. Chen. Constraint-hiding constrained PRFs for NC^1 from LWE. In *EUROCRYPT*, pages 446–476, 2017.
15. R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *TCC*, 2017.
16. Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *ACISP*, 2004.
17. B. Chor and N. Gilboa. Computationally private information retrieval. In *STOC*, 1997.
18. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, 1995.
19. B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *J. ACM*, 45(6):965–981, Nov. 1998.
20. H. Corrigan-Gibbs and D. Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.
21. D. Demmler, A. Herzberg, and T. Schneider. Raid-pir: Practical multi-server pir. In *CCSW*, 2014.
22. S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In *TCC*, 2016.
23. G. Di-Crescenzo, Y. Ishai, and R. Ostrovsky. Universal service-providers for database private information retrieval. In *PODC*, 1998.

24. Z. Dvir and S. Gopi. 2-server pir with subpolynomial communication. *J. ACM*, 63(4), 2016.
25. S. Garg, P. Mohassel, and C. Papamanthou. TWORAM: efficient oblivious RAM in two rounds with applications to searchable encryption. In *CRYPTO*, 2016.
26. W. I. Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.
27. C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In *EUROCRYPT*, 2014.
28. C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP*, 2005.
29. O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *STOC*, 1987.
30. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *J. ACM*, 33(4), Aug. 1986.
31. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 1996.
32. S. M. Hafiz and R. Henry. Querying for queries: Indexes of queries for efficient and expressive IT-PIR. In *CCS*, 2017.
33. A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. Private anonymous data access. In *EUROCRYPT*, 2019.
34. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *STOC*, 2004.
35. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Cryptography from anonymity. In *FOCS*, pages 239–248, 2006.
36. D. Kogan and H. Corrigan-Gibbs. Private blocklist lookups with checklist. In *Usenix Security*, 2021.
37. E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *FOCS*, 1997.
38. H. Lipmaa. First CPIR protocol with data-dependent computation. In *ICISC*, 2009.
39. S. Lu and R. Ostrovsky. How to garble ram programs. In *EUROCRYPT*, 2013.
40. W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *FC*, 2015.
41. B. Morris and P. Rogaway. Sometimes-recurse shuffle - almost-random permutations in logarithmic expected time. In *Eurocrypt*, volume 8441, pages 311–326. Springer, 2014.
42. R. Ostrovsky and W. E. Skeith, III. A survey of single-database private information retrieval: techniques and applications. In *PKC*, pages 393–411, 2007.
43. S. Patel, G. Persiano, and K. Yeo. Private stateful information retrieval. In *CCS*, 2018.
44. P. Pudlák and V. Rödl. Modified ranks of tensors and the size of circuits. In *STOC*, 1993.
45. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6), Sept. 2009.
46. T. Ristenpart and S. Yilek. The mix-and-cut shuffle: Small-domain encryption secure against N queries. In *CRYPTO*, 2013.
47. E. Shi, W. Aqeel, B. Chandrasekaran, and B. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. Online full version of this paper. Cryptology ePrint Archive, Report 2020/1592, 2020. https://eprint.iacr.org/2020/1592.

48. E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In *ASIACRYPT*, 2011.
49. S. Singanamalla, S. Chunhapanya, 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, 2020.
50. E. Stefanov and E. Shi. Fastprp: Fast pseudo-random permutations for small domains. *IACR Cryptol. ePrint Arch.*, 2012:254.
51. R. R. Toledo, G. Danezis, and I. Goldberg. Lower-cost ϵ-private information retrieval. *PETS*, 2016.
52. F. Wang, C. Yun, S. Goldwasser, V. Vaikuntanathan, and M. Zaharia. Splinter: Practical private queries on public data. In *NSDI*, 2017.
53. A. C.-C. Yao. Coherent functions and program checkers. In *STOC*, 1990.