TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH

Arthur Lazzaretti * and Charalampos Papamanthou**

Yale University

Abstract. In Private Information Retrieval (PIR), a client wishes to retrieve the value of an index i from a public database of N values without leaking any information about i. In their recent seminal work, Corrigan-Gibbs and Kogan (EUROCRYPT 2020) introduced the first two-server PIR protocol with sublinear amortized server time and sublinear $O(\sqrt{N}\log N)$ bandwidth. In a followup work, Shi et al. (CRYPTO 2021) reduced the bandwidth to polylogarithmic by proposing a construction based on privately puncturable pseudorandom functions, a primitive whose only construction known to date is based on heavy cryptographic primitives such as LWE. Partly because of this, their PIR protocol does not achieve concrete efficiency. In this paper we propose TreePIR, a two-server PIR protocol with sublinear amortized server time and polylogarithmic bandwidth whose security can be based on just the DDH assumption. TreePIR can be partitioned in two phases that are both sublinear: The first phase is remarkably simple and only requires pseudorandom generators. The second phase is a single-server PIR protocol on *only* $\sqrt{N}$ indices, for which we can use the protocol by Döttling et al. (CRYPTO 2019) based on DDH, or, for practical purposes, the most concretely efficient single-server PIR protocol. Not only does TreePIR achieve better asymptotics than previous approaches while resting on weaker cryptographic assumptions, it also outperforms existing two-server PIR protocols in practice. The crux of our protocol is a new cryptographic primitive that we call weak privately puncturable pseudorandom functions, which we believe can have further applications.

Keywords: Private Information Retrieval · Puncturable Pseudorandom Functions · Privacy-Preserving Primitives.

1 Introduction

Private Information Retrieval (PIR) is classically a two-player protocol where the client holds an index $i \in \{0, \ldots, N-1\}$ and the server holds a public string DB of N bits. The goal of the protocol is for the client to learn $\mathsf{DB}[i]$, and for the server *not to learn any information related to* i. Since the problem was introduced [14], PIR has become a building block for a myriad of privacy-preserving applications [3, 4, 25, 33, 44].

* arthur.lazzaretti@yale.edu

** charalampos.papamanthou@yale.edu

PIR has been studied extensively over the years [2, 6, 9, 10, 14, 26, 34, 39, 40] [1] and unfortunately, all space-efficient PIR protocols are bound to a well-known linear server time lower bound by Beimel et al. [6]. Intuitively, linear server time is required since otherwise some index-specific portion of the database will remain untouched, and therefore information about the queried index will be leaked. To address this problem, Corrigan-Gibbs and Kogan [15] propose a model with *client preprocessing* and *two* non-colluding servers that store copies of the database: After one expensive *query-independent* offline phase where the client interacts with the first server and stores a small amount of information, subsequent queries run privately in time sublinear in the database size, resulting in *amortized* sublinear time per query. The online phase involves the second server and may or may not involve the first server. This model has shown to have many useful applications in practice, and brings PIR query times substantially closer to the non-private query baseline.

1.1 Client-Preprocessing PIR

The core idea of client-preprocessing PIR, as proposed by Corrigan-Gibbs and Kogan [15], is the following: In the offline phase, the client samples a certain number (in particular $\sqrt{N}$) of random index sets and asks the first server to compute parities of those sets, storing these parities, along with the respective sets, locally. In the online phase (query to index i), the client finds a preprocessed set S that contains i, and sends $S \setminus \{i\}$ to the second server. The server then returns the parity of the set $S \setminus \{i\}$, and the client can compute the value of index i through the difference of its preprocessed parity and the new parity. (The actual protocol is a little more complicated but we want to keep the exposition at a high level for now.) Note that since the online server time is a computation over a set of $\sqrt{N}$ indices, the server computation is sublinear.

Reducing communication complexity in client-preprocessing PIR. Several optimizations of the above idea have been proposed. For example, Corrigan-Gibbs and Kogan [15] observe that instead of sending the actual sets to the first server, one can send small PRP keys representing those sets, allowing the server to compute the sets itself—this ensures sublinear offline communication and sublinear client storage. Still, the online query cannot be further compressed and the client must send $S \setminus \{i\}$, as before. To reduce online communication, Kogan and Corrigan-Gibbs [33] propose representing their sets with keys derived from *puncturable* PRFs [23, 32]—such keys can be updated to represent a set with a removed element i while still (i) hiding which element i was removed; (ii) maintaining the small key size. Unfortunately, this approach does not directly support fast membership testing (which is crucial in order to find the preprocessed set that contains i during the online phase), due to the non-invertibility of PRFs. Therefore finding a set containing i during the online phase requires $O(N)$ expected time. (In their work, they propose a faster membership test by

[1] A non-exhaustive list, we provide more background on related work in Section 1.3.

using a linear-space data structure. For some use cases of PIR however, using linear client storage can be prohibitive.)

Client-preprocessing PIR via privately puncturable PRFs. In CRYPTO 2021, Shi et al. [43] addressed the above shortcomings by proposing *puncturable pseudorandom sets*. Their seminal construction achieves the following three properties: First, a set can be represented with a small key k; Second, this small key k can be updated to k_i to represent a set with a removed element i, while (i) hiding what i is and (ii) maintaining the small key size; Third, one can check membership of any element in key k efficiently, i.e., in polylogarithmic time. Their construction is based on *privately puncturable* PRFs, whose only instantiation is based on LWE [7, 13]). As such, although the Shi et al. scheme has excellent asymptotic complexities, it does not seem to have concrete efficiency. Our back-of-the-envelope calculations show communication overhead of hundreds of megabytes, which make it unusable in practice for now. (We discuss this further in Section 5.)

Therefore, we still do not have a suitable sublinear-time PIR scheme with concrete efficiencies and low communication. Our scheme, TreePIR, was developed to bridge the gap. We paint a full picture of the asymptotics mentioned above, including our new scheme TreePIR, in Figure 1.

1.2 Our Contribution

In this work, we present a new two-server PIR scheme that achieves polylogarithmic bandwidth and sublinear server time and client storage, from DDH. *To the best of our knowledge, ours is the first scheme achieving such complexities from such a well-established cryptographic assumption.* (For comparison with existing work, see Figure 1.) Our construction is simple and reuses ideas from the celebrated GGM PRF construction [23] in a novel way, introducing a new primitive that we call *weak privately puncturable pseudorandom functions* (wpPRFs). Due to its conceptual simplicity, our scheme lends itself to an efficient implementation, showing strong performance improvements over past schemes for many use cases of PIR. We now summarize our core technical ideas.

Weak privately puncturable PRFs. A wpPRF satisfies the strong notion of privacy of privately puncturable PRFs, where the punctured key hides both the point that was punctured and its evaluation, but with relaxed correctness. The relaxed correctness property states that one is only able to compute the PRF values from the punctured key if they know the point that was punctured. The punctured point is an additional input to the evaluation algorithm for the punctured key. A second property that a wpPRF must satisfy is to allow the enumeration the whole domain for all "potentially punctured points" in quasilinear time. We note here that previously, the only known way to construct puncturable PRFs that hide both the evaluation at the punctured point and the punctured index was using LWE with superpolynomial modulus, in conjunction with other inefficient cryptographic primitives [8, 13]. Instead, we show that we are able to construct a weaker version of puncturable PRFs (that fits our application) relying solely on the existence of one-way functions. This is shown in Section 3.

Protocol	Server* Time	Client Storage	Bandwidth*	Assump.
TreePIR, Theorem 4.1	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log N)$	$O(\sqrt{N})$	OWF
TreePIR, Lemma 4.1	$O(\sqrt{N}\log N)$	$O(\sqrt{N})$	$O(\text{poly}\log N)$	DDH
Shi et al. [43]$^{\beta}$	$O(\sqrt{N}\log^2 N)$	$O(\sqrt{N}\log^2 N)$	$O(\text{poly}\log N)$	LWE
Checklist [33]	$O(\sqrt{N})$	$O(N\log N)$	$O(\log N)$	OWF
PRP-PIR [15]	$O(\sqrt{N})$	$O(\sqrt{N})$	$O(\sqrt{N}\log N)$	OWF

$^{\beta}$ The big O notation hides factors very large in the security parameter for this scheme.

Fig. 1. Complexities for different PIR schemes over a database of size N. Server time and bandwidth are amortized over $\sqrt{N}$ queries (denoted amortized with a *). Client time is omitted because it is the same for all schemes, $O(\sqrt{N}\log N)$.

Applying weak privately puncturable PRFs to PIR. We use our new primitive, the weak privately puncturable PRF, with domain and range $\sqrt{N}$ to construct sets that are concise, remain concise after removing one element, and support fast membership testing. This is the first construction to achieve all three properties in unison from only one-way functions. The tricky part is that given the relaxed notion of correctness of our new primitive, it is not straightforward exactly how we can use these sets in PIR. We expand on this in Section 4 and show to use our primitive to reduce the problem of PIR on N elements to PIR on $\sqrt{N}$ elements during the online query, using sublinear time and logarithmic communication.

To reduce communication further, one can recursively apply a second PIR scheme to retrieve the element of interest from the resulting database, incurring the cost of the PIR scheme used on the database of size $\sqrt{N}$, because we know exactly which index is of interest within the smaller database. This means that TreePIR benefits from previous (and future) work on non-preprocessing PIR, since it is compatible with the state-of-the-art single-server PIR schemes. Our techniques paired with previous results enable us to achieve PIR with polylogarithmic amortized bandwidth and sublinear amortized server time.

Notably, paired with the result from Döttling et al. [19], our technique implies the first sublinear time PIR scheme with non-trivial client storage and polylogarithmic communication complexity from only the Decisional Diffie-Hellman (DDH) assumption.

A TreePIR implementation. As a second contribution, we provide an implementation of TreePIR. We benchmark our implementation in Section 5, and show strong evidence of its practical value. For many usecases, somewhat surprisingly, downloading the $\sqrt{N}$ bits without recursing presents itself as a better alternative to employing a second PIR scheme over $\sqrt{N}$ databases. This is because, in general, current single-server PIR schemes incur somewhat large baseline communication costs and cannot handle small elements well.

Our implementation of TreePIR shows an amortized query time of over three times faster than Checklist [33] over different tests, using up to 8,000× less

client storage. We also outperform the other known implementation of client-preprocessing PIR by Corrigan-Gibbs and Kogan and Ma et al. [15, 38]. The price we pay for the improvements in client storage and query time is increased communication, which we believe is still reasonable for many applications, given the other improvements. We provide a full picture of performance comparisons against previous schemes in Section 5.

1.3 Related Work

The first PIR protocol to achieve non-trivial communication was introduced, along with the problem of PIR itself, by Chor et al. [14]. This scheme relies on a two-server assumption, where the database is replicated in two non-colluding servers. This has proven to be a reasonable assumption in practice [25, 30, 33]. Later, it was shown that non-trivial communication can also be achieved without the two-server assumption [34], albeit paying a hefty computational price on the server. Subsequent to the seminal works on two-server PIR and single-server PIR, many works have inched towards bringing PIR closer to being practical [5, 17, 18, 20, 22, 31, 36, 37, 47].

In 2000, Beimel et al. [6] showed that a PIR scheme must incur at least linear work per query when the server stores no extra bits. In the same work, it was shown that we can decrease server work by storing additional bits at the server (the server-preprocessing model), since the client is not involved. To date, many efforts have been directed towards improving PIR in this model, which is also sometimes called Doubly Efficient PIR [10, 29]. These works can achieve very good server time and bandwidth (polylogarithmic). One drawback of these works is that many of them rely on non-standard assumptions, and we have yet to evaluate how well these perform in practice. However, it is an interesting direction to be able to store some small amount of preprocessed bits on the server that allow client queries to run in amortized sublinear time, since this would imply the possibility to amortize queries across different clients (as opposed to the client-preprocessing model, where each offline phase is client specific).

1.4 Notation

We define $\nu(\cdot)$ to be a negligible function, such that for every polynomial $p(\cdot)$, $\nu(\cdot) < 1/p(\cdot)$. We define overwhelming probability to be the probability $1 - \nu(\cdot)$. Unless otherwise noted, let $\lambda \in \mathbb{N}$ be the security parameter and $m, n \in \mathbb{N}$ be arbitrary natural numbers and $N = 2^n$. We index a bitstring x at index i using notation x_i and an array a at index i with notation $a[i]$, both are 0-indexed. For any bitstring x, we define x^ℓ, x^r such that $x = x^\ell || x^r$, where $|x^\ell| = |x^r| = |x|/2$. For any $q \in \mathbb{N}$, let $[q]$ denote the set $\{0, \ldots, q-1\}$. We use the notation $i \xleftarrow{R} S$ to denote that i is an element sampled uniformly at random from the set of elements of S. Unless explicitly stated, our big-O notation $O(\cdot)$ hides factors in the security parameter.

1.5 Paper Outline

On Section 2, we recall definitions and constructions from previous work that will be useful in constructing our scheme. On Section 3, we introduce our new primitive, the weak privately puncturable PRF, and show how to construct it from one way functions. Next, we provide our PIR scheme, TreePIR on Section 4, and prove its correctness, privacy and efficiency. Finally, we benchmark an implementation of our scheme against previous PIR schemes in Section 5.

2 Preliminaries

Here we outline definitions and primitives that we will need throughout the paper.

2.1 Security Definitions for PIR

We first formally define correctness and privacy for PIR.

Definition 2.1 (PIR correctness). *A PIR scheme (*$\mathbf{server}_0$,$\mathbf{server}_1$,**client***) is correct if, for any polynomial-sized sequence of queries* $x_1, \ldots, x_Q$*, the honest interaction of* **client** *with* $\mathbf{server}_0$ *and* $\mathbf{server}_1$ *that store a polynomial-sized database* $\mathsf{DB} \in \{0,1\}^N$*, returns* $\mathsf{DB}[x_1], \ldots, \mathsf{DB}[x_Q]$ *with probability* $1 - \nu(\lambda)$*.*

Definition 2.2 (PIR privacy). *A PIR scheme (*$\mathbf{server}_0$,$\mathbf{server}_1$,**client***) is private with respect to* $\boldsymbol{server}_1$ *if there exists a PPT simulator* Sim*, such that for any algorithm* $serv_0$*, no PPT adversary* $\mathcal{A}$ *can distinguish the following experiments with non-negligible probability:*

- $\boldsymbol{Expt}_0$*:* **client** *interacts with* $\mathcal{A}$ *who acts as* $\mathbf{server}_1$ *and* $serv_0$ *who acts as the* $\mathbf{server}_0$*. At every step* t*,* $\mathcal{A}$ *chooses the query index* x_t*, and* **client** *is invoked with input* x_t *as its query.*
- $\boldsymbol{Expt}_1$*:* Sim *interacts with* $\mathcal{A}$ *who acts as* $\mathbf{server}_1$ *and* $serv_0$ *who acts as the* $\mathbf{server}_0$*. At every step* t*,* $\mathcal{A}$ *chooses the query index* x_t*, and* Sim *is invoked with no knowledge of* x_t*.*

In the above definition our adversary $\mathcal{A}$ can deviate arbitrarily from the protocol. Intuitively the privacy definition implies that queries made to $\mathbf{server}_1$ will appear random to $\mathbf{server}_1$, assuming servers do not collude (as is the case in our model). Privacy for $\mathbf{server}_0$ is defined symmetrically.

We will need these when constructing our scheme in Section 4. Until then, we shift our focus slightly to other primitives we will require to build TreePIR.

2.2 Pseudorandom Generators (PRGs) and Pseudorandom Functions (PRFs)

Our core technique builds upon the celebrated construction of a PRF from a length-doubling PRG by Goldreich, Goldwasser and Micali [23], henceforth denoted the GGM construction. We introduce both the definitions of a PRG, a PRF, and give the GGM construction in the remainder of this section.

Definition 2.3 (PRG). *A PRG $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ satisfies security if, for any $k \in \{0,1\}^\lambda$ and $r \in \{0,1\}^{2\lambda}$ sampled uniformly at random, for any PPT adversary $\mathcal{A}$, there is a negligible function $\nu(\lambda)$ such that*

$$|\Pr[\mathcal{A}(G(k)) \to 1] - \Pr[\mathcal{A}(r) \to 1]| \leq \nu(\lambda)\,.$$

We also define below the pseudorandomness property for a PRF.

Definition 2.4 (PRF). *A PRF $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^m$ satisfies security if, for any $k \in \{0,1\}^\lambda$ sampled uniformly at random, for any function $\mathcal{F}$ sampled uniformly at random from the set of functions mapping $\{0,1\}^n \to \{0,1\}^m$, for any PPT adversary $\mathcal{A}$, there is exists a negligible function $\nu(\lambda)$ such that*

$$|\Pr[\mathcal{A}^{\mathcal{O}_{\mathcal{F}(\cdot)}} \to 1] - \Pr[\mathcal{A}^{\mathcal{O}_{F(k,\cdot)}} \to 1]| \leq \nu(\lambda).$$

2.3 The GGM PRF Construction and Puncturing

Given a PRG G as above, the GGM construction of a PRF F works as follows. Let us define for any output of G on input k, $G(k) = G_0(k)||G_1(k)$, where $|G_b(\cdot)| = \lambda$ for $b \in \{0,1\}$. To simplify sequential applications of G, we also define $G_{10}(\cdot) = G_1(G_0(\cdot))$. From G, we construct a PRF $F : \{0,1\}^\lambda \times \{0,1\}^n \to \{0,1\}^\lambda$ as follows. For key $k \in \{0,1\}^\lambda$ and input $x \in \{0,1\}^n$, let $F_k(x) = G_x(k)$. As shown in [23], this outputs a secure PRF with evaluation time n, assuming the PRG is secure. The construction can be visualized as a tree with k as the root with recursive applications of G split in half as its children.

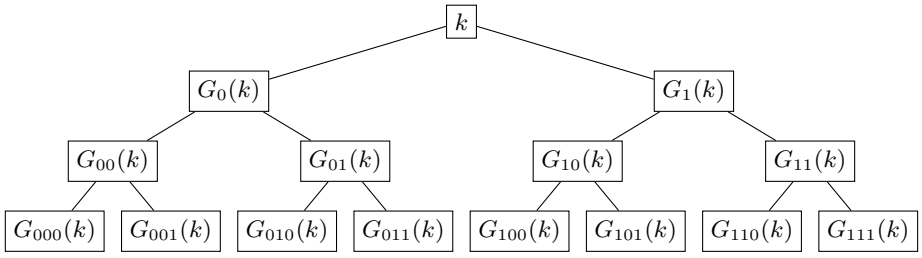


Fig. 2. The GGM PRF tree.

Figure 2 represents the tree for a GGM PRF with input length $n = 3$, key length λ and output length $m = \lambda$. [2] Now, this PRF construction is not ideal in terms of practical evaluation time, since it requires sequential applications of G linear in the size of the input. However, it is also very powerful since it allows us to

[2] We note that this construction is only secure for a fixed input length. Also, we can support any output length either truncating an output to be less than λ or reapplying G sequentially on the final leaf node to increase the output size.

constrain the PRF key so as to *disallow evaluation at one point*. In the literature this is commonly referred to as a puncturing constraint. The constraint can be picked selectively after the key generation. We denote a PRF that selectively allows for a puncturing constraint as a puncturable PRF (pPRF)[3]. We define a pPRF below and give additional security properties it must satisfy.

Definition 2.5 (Puncturable PRFs). *Let n and m be public parameters. A pPRF P maps n-bit inputs to m-bit outputs and is defined as a tuple of four algorithms.*

- $\mathsf{Gen}(1^\lambda) \to k$*: Generates key* $k \in \{0,1\}^\lambda$ *given security parameter* λ.
- $\mathsf{Eval}(k, x) \to y$*: Takes in a key* k *and a point* $x \in \{0,1\}^n$ *and outputs* $y \in \{0,1\}^m$*, the evaluation of* P *on key* k *at point* x.
- $\mathsf{Puncture}(k, x) \to k_x$*: Outputs* k_x*, the key* k *punctured at point* x.
- $\mathsf{PEval}(k_x, x') \to y$*: Takes in a punctured key* k_x *and a point* $x' \in \{0,1\}^n$ *and outputs* y*, the evaluation of* P*'s key* k_x *at point* x'.

Along with standard pseudorandomness (Definition 2.4), the pPRF P must satisfy the following additional (informal) properties.

1. The punctured key k_x reveals nothing about $P.\mathsf{Eval}(k, x)$, the evaluation of the point x on the unpunctured key.
2. For any point x' not equal to x, $P.\mathsf{Eval}(k, x')$ equals $P.\mathsf{PEval}(k_x, x')$.

We formalize these below.

Definition 2.6 (Security in puncturing). *A puncturable pseudorandom function* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval})$ *satisfies security in puncturing if for* $r \in \{0,1\}^m$ *sampled uniformly,* $k \leftarrow \mathsf{Gen}(1^\lambda)$*, there exists a negligible function* $\nu(\lambda)$ *such for any PPT adversary* $\mathcal{A}$*,* $\mathcal{A}$ *cannot distinguish between the following experiments below with probability more than* $\nu(\lambda)$.

- $\boldsymbol{Expt}_0$*:* $x \leftarrow \mathcal{A}(1^\lambda)$*,* $\mathsf{Puncture}(k, x) \to k_x$*,* $b' \leftarrow \mathcal{A}(k_x, \mathsf{Eval}(k, x))$.
- $\boldsymbol{Expt}_1$*:* $x \leftarrow \mathcal{A}(1^\lambda)$*,* $\mathsf{Puncture}(k, x) \to k_x$*,* $b' \leftarrow \mathcal{A}(k_x, r)$.

Definition 2.7 (Correctness in puncturing). *A puncturable pseudorandom function* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval})$ *satisfies correctness in puncturing if for* $k \leftarrow \mathsf{Gen}(1^\lambda)$*, for any point* $x \in \{0,1\}^n$*, for* $k_x \leftarrow \mathsf{Puncture}(k, x)$*, it holds that* $\forall x' \in \{0,1\}^n$ x' *not equal to* x*,* $\mathsf{Eval}(k, x') = \mathsf{PEval}(k_x, x')$.

A pPRF construction based on a GGM style PRF was widely referenced in the literature for many years before it was finally formalized by Kiayias et al. [32]. The construction goes as follows: When puncturing a point x, we remove the "path to x" from the evaluation tree created using k and output the keys so that the adversary can reconstruct all the other values except for x. We will be handing the adversary a key of size $n \cdot \lambda$ (instead of just λ), that allows evaluation of the pPRF in every point of the domain *except for* x. We also note that the

[3] Other works have studied adaptively picked constraints for pPRFs [21, 28].

punctured point x is part of the punctured key, so that the adversary is able to reconstruct the pPRF's structure. We expand on this in the next section. Kiayias et al. [32] conduct a formal analysis of this initial pPRF scheme and show that it satisfies the security and correctness properties above.

Then, we show how one can modify this well-know GGM construction to achieve our new desired primitive with stronger privacy guarantees.

3 Weak Privately Puncturable PRFs

In this section we introduce a new primitive called *weak privately puncturable pseudorandom functions* that is going to be useful for our final construction. Weak privately puncturable PRFs are privately puncturable PRFs [7, 11, 13, 42] that satisfy a weaker notion of *correctness*.

But first, let us see what a privately puncturable PRF is: privately puncturable PRFs satisfy a stricter security definition than the pPRF introduced in Section 2. Note that although the punctured key k_x of a pPRF P reveals nothing about $P.\mathsf{Eval}(k, x)$, it still reveals the punctured point, x. In fact, without revealing x, there is no way to evaluate the pPRF punctured key at the other points. This is not necessarily inherent to all pPRFs but it is certainly inherent to the GGM scheme. In contrast, privately puncturable PRFs also hide the punctured point! This very powerful primitive was built using techniques that depart significantly from the GGM construction, and current schemes employ heavy machinery, such as lattices with super-polynomial moduli and fully-homomorphic encryption to achieve private puncturing. Because of this, these are unfortunately very far from being practical, especially for smaller domains.

So, can we have Privately Puncturable PRFs from simpler assumptions, ones that would allow more efficient implementation? Let us take a step back and look at one specific goal, i.e., that of hiding the index that was punctured.

We examine how this could be achieved on a standard GGM pPRF. This requires a closer look into exactly what comprises a pPRF punctured key given our current GGM pPRF construction. Suppose that we take the pPRF P defined by the tree in Figure 2 and would like to puncture the point 010. In order to satisfy our Definition 2.6 we need to remove all the nodes on the path to 010, so that it cannot be computed given a punctured key. We are left with the tree in Figure 3.

After removing the nodes in red, note that the strings on the nodes highlighted in yellow, and the punctured point, 010 are necessary (and sufficient) [32] to reconstruct the remaining outputs of P. Put together, we require our punctured key for the pPRF to be the tuple $(010, [G_{00}(k), G_{011}(k), G_1(k)])$, where the array is *ordered* (in a left-to-right fashion with respect to the tree)[4]. This punctured key satisfies our privacy and correctness definitions for the pPRF [32].

Our first attempt to hide the punctured point is to simply remove it from the key. In our example, instead of outputting the tuple $(010, [G_{00}(k), G_{011}(k), G_1(k)])$

[4] This is equivalent to a depth-first ordering up to some deterministic shifting, however this ordering will be more intuitive for our approach moving forward.

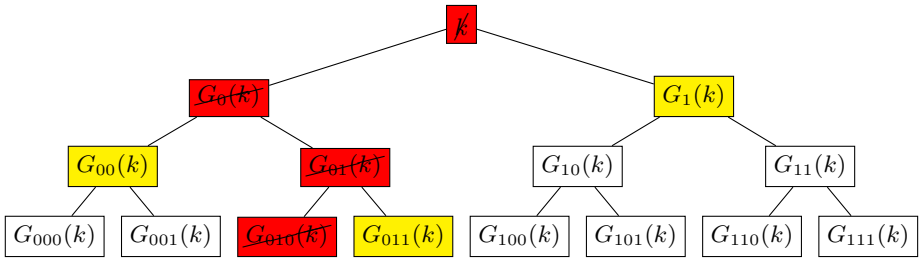


Fig. 3. Puncturing a GGM PRF.

as our punctured key, we output only the ordered array $[G_{00}(k), G_{011}(k), G_1(k)]$. By security of the PRG (Definition 2.3), this should not leak any information about the punctured point (intuitively, the array is just a sequence of random strings). We now have a construction that satisfies privacy in the point punctured! However, it is not clear as of now how this will be useful. How do we evaluate anything with this when not given the punctured point? After all, as was noted in [32], the point is necessary to reconstruct the original function evaluations at the other indices.

One approach is to guess the punctured point, meaning that evaluating a point in the punctured key now takes two inputs other than the key: both the point to be evaluated (as before) and the point at which the original key was punctured (new). A correct guess will enable us be able to evaluate the function correctly, since a correct guess means that we have exactly the same key as before. An incorrect guess will likely yield some other random string. For example, if we guess 000 as the punctured point, we can arrange our array $[G_{00}(k), G_{011}(k), G_1(k)]$ in a tree *as if* the punctured index was 000 (it is important that to note the ordering of the array). We construct this tree in Figure 4.

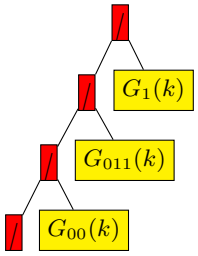


Fig. 4. Reconstructing attempted GGM tree from index and strings.

In Figure 4, although the first half of the tree is not consistent with our initial evaluation, $G_1(k)$ is placed correctly and therefore the evaluations of the

last four indices will be consistent with our unpunctured key. This is not good enough to satisfy any current definition of correctness, but it points us in the right direction. *Some evaluations will be shared across different guesses.*

The key observation required for our work is that if we are interested in *every evaluation in the domain* except the punctured point, the fact that different "puncture guesses" are related can be used to our advantage. By construction, we can evaluate the whole domain of input-output pairs for our initial guess of the punctured point 000 in N time (the tree has $2N$ nodes total). Let us denote this set S_{000}. Now, using this S_{000}, we can compute the entire domain of input-output pairs for the PRF on a "puncture guess" of 001, S_{001}, by only performing one removal and one addition to S_{000}.

Applying this observation across all possible punctured guesses, we iteratively obtain the set of all input-output pairs for every "potential punctured point" in just $N \log N$ time! Out of these N sets, one is correct (using correctness as defined in Definition 2.7[5]). In our example, this would be S_{010}. Crucially, the "correct evaluation set" still does not reveal the evaluation at the punctured point, by security of the pPRF construction we saw in Section 2.

In the remainder of this section, we will define our new primitive, the weak privately puncturable PRF (wpPRF), give its security definitions, and provide our construction. It follows a generalized version of the example above.

Definition 3.1 (Weak Privately Puncturable PRF). *We define a weak privately puncturable pseudorandom function (wpPRF) F as a tuple of four algorithms.*

- $\mathsf{Gen}(1^\lambda) \to k$*: Takes in a security parameter λ and returns the wpPRF key $k \in \{0,1\}^\lambda$.*
- $\mathsf{Eval}(k, x) \to y$*: Takes $x \in \{0,1\}^n$ as input and outputs the evaluation on key k at x, $y \in \{0,1\}^m$.*
- $\mathsf{Puncture}(k, i) \to k_i$*: Takes in the wpPRF key k and an input from the domain i and outputs the privately punctured key k_i punctured at point i.*
- $\mathsf{PEval}(k_i, j, x) \to y$*: Takes in a privately punctured key k_i, a guess j of the point that k_i was punctured on, and the point to be evaluated x, and outputs the evaluation of the point x for punctured key k_i with potential puncturing index j.*

First, note that our $\mathsf{Gen}(\cdot)$ and $\mathsf{Eval}(\cdot,\cdot)$ algorithms must satisfy the standard PRF pseudorandomness definition (Definition 2.4). We also require our wpPRF to satisfy the same notion of security in puncturing as the pPRF (Definition 2.6. Since the adversary *picks* and therefore *knows* x, it can evaluate $\mathsf{PEval}(k_x, x, \cdot)$ on every input except x, which is equivalent to the experiment on the original pPRF (Definition 2.5).

Our $\mathsf{Puncture}$ algorithm must satisfy an additional notion of privacy with respect to the puncture operation, aside from Definition 2.6. The puncture must

[5] These sets are related, and there are only $N \log N$ unique elements across all sets. We can exploit this, defining the first set in full and the following ones as set differences.

hide both the evaluation at the point punctured *and* the point punctured. We capture the second property below:

Definition 3.2 (Privacy in puncturing). *A weak privately puncturable PRF* $(\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval})$ *satisfies privacy in puncturing if given a uniformly random* $b \in \{0,1\}$, $k \in \{0,1\}^\lambda$ *there exists a negligible function* $\nu(\lambda)$ *such that for any probabilistic polynomial time adversary* $\mathcal{A}$, $\mathcal{A}$ *cannot correctly guess* b *with probability more than* $\frac{1}{2} + \nu(\lambda)$ *in the experiment below.*

- $k \leftarrow \mathsf{Gen}(1^\lambda)$.
- $(x_0, x_1) \leftarrow \mathcal{A}(1^\lambda)$.
- $k_{x_b} \leftarrow \mathsf{Puncture}(k, x_b)$.
- $b' \leftarrow \mathcal{A}(k_{x_b})$.

Finally, we also redefine correctness with respect to private puncturing, where, intuitively, we only require $\mathsf{PEval}(k_i, j, x)$ to be equal to $\mathsf{Eval}(k, x)$ on the unpunctured key k if i equals j. Note that by Definition 3.2 k_i gives no information about i. For i not equal to j, the output will look random, but will not necessarily map to the original PRF output.

Definition 3.3 (Weak correctness in puncturing). *A weak privately puncturable PRF (*Gen, Eval, $\mathsf{Puncture}$, PEval*) satisfies weak correctness in private puncturing if given* $k \leftarrow \mathsf{Gen}(1^\lambda)$, *for any point* $x \in \{0,1\}^n$, $k_x \leftarrow \mathsf{Puncture}(k, x)$, *it holds that* $\forall x' \in \{0,1\}^n$, $x' \neq x$, $\mathsf{Eval}(k, x') = \mathsf{PEval}(k_x, x, x')$.

Lastly, for our scheme to be useful, we require one final property, which we will denote *efficient full evaluation*. This will ensure that given some punctured key, we can evaluate our wpPRF on its full domain, for every possible punctured index, in $O(N \log N)$.The definition below captures this property.

Definition 3.4 (Efficient full evaluation). *Let* F *be a weak privately puncturable PRF (*Gen, Eval, $\mathsf{Puncture}$, PEval*) and let* $N = 2^n$. *Also let* $k \leftarrow \mathsf{Gen}(1^\lambda)$ *and* $k_i \leftarrow \mathsf{Puncture}(k, i)$ *for some* $i \in \{0,1\}^n$. *Define*

$$S_j = \{(x, \mathsf{PEval}(k_i, j, x)) \,|\, x \in \{0,1\}^n \wedge x \neq j\}\,.$$

We say that F *satisfies* efficient full evaluation *if all sets* $\{S_j\}_{j \in \{0,1\}^n}$ *can be enumerated in* $O(N \log N)$ *time.*

It is clear that to satisfy efficient full evaluation there needs to be overlap between the sets S_j, as will be the case with our construction. Otherwise, $\Omega(N^2)$ computation and space is needed. For our case, we will show that due to the tree construction, there are at most $N \log N$ unique elements that can be enumerated, and so if we define each set iteratively from the first, we can evaluate the whole domain of the PRF for every punctured guess in $O(N \log N)$ time.

Our wpPRF construction.

Let G be a length-doubling PRG (satisfying Definition 2.3).

- $\mathsf{Gen}(1^\lambda) \to k$:
  - Output a uniform random string of length λ.
- $\mathsf{Eval}(k, x) \to y$:
  - Let $y \leftarrow G_x(k)$, output y.
- $\mathsf{Puncture}(k, i) \to k_i$:
  - Output list of seeds *not* in path to i, ordered left to right, as shown in Figure 3.
- $\mathsf{PEval}(k_i, j, x) \to y$:
  - Let $y \leftarrow G_x((j, k_i))$. We denote with $G_x((j, k_i))$ the leaf node at position x of the tree reconstructed from (j, k_i) as shown in Figure 4. Output y.

Fig. 5. Our wpPRF construction.

3.1 A wpPRF Construction

Our construction follows exactly our earlier description in this section, slightly modified from the GGM pPRF to fit the new definitions. We give the full construction in Figure 5.

Theorem 3.1 (wpPRFs). *Assuming the security of the pseudorandom generator G (Definition 2.3), our wpPRF scheme (Definition 3.1) satisfies pseudorandomness (Definition 2.4), security in puncturing (Definition 2.6), privacy in puncturing Definition 2.6, weak correctness in puncturing (Definition 3.3) and efficient full evaluation (Definition 3.4).*

Proof. Note that pseudorandomness follows from the standard GGM construction and proof from [23]. Weak correctness in puncturing follows directly by construction. Security in puncturing follows from the pPRF security proof in [32], since our privately punctured key is a strict subset of the punctured key in the GGM construction.

Privacy in puncturing follows from directly from the security of G. Our punctured key is an ordered array of random strings, and therefore cannot leak any information about the index that was punctured. The key can be simulated by generating $\log N$ random strings of size λ, and by security of G that will be indistinguishable from our key for any probabilistic polynomial time adversary.

Finally, we show that our scheme also satisfies efficient full evaluation. Given a punctured key k_i, we enumerate all sets S_j using the following algorithm.

- Step 1: Compute S_{0^n}, as defined in Definition 3.4. This takes $O(N \log N)$ time.

– Step 2: For $j = 1, \ldots, N-1$:
  1. Let h be the height of the node between index $j-1$ and index j on the tree. We denote leaf nodes to have $h = 0$.
  2. $S_j = \{(v, F.\mathsf{PEval}(k_i, j, v)\}_{v \in \{j-2^{h-1}, \ldots, j+2^{h-1}\}}$.

Given that we run into a transition of height h with exactly $2^n/2^h$ times, we have that going through this loop we will take

$$\sum_{h=1}^{n} \frac{2^n}{2^h} \times 2^h = n2^n = N \log N$$

steps. Then, this whole process of evaluating every S_j takes time $2N \log N = O(N \log N)$. Note that each S_j as defined above has 2^h elements and so by a similar argument we have that this conjunction of all sets will have $O(N \log N)$ elements. (Intuitively, the first set will be constructed normally and the remaining sets will be constructed iteratively from the first, reusing evaluations.)

Finally, we have to show that each of this set of $\{S_i\}_{i \in [N]}$ does indeed represent the appropriate full evaluation for all potential puncture at points $j \in [N]$. We define the real set of mappings for a puncture guess of j to be $S'_j = S'_{j-1} + S_j$, where we define the $+$ operation to be the union of both sets, except when there are two mappings of the same index, we overwrite to the value to the latter value. As an example, if we have S contain the entry (x, y) and S' contain (x, y'), the set $S + S'$ contains only (x, y'). It is straightforward to verify that for any j, $S'_j = S_0 + S_1 + \ldots + S_j$ corresponds to the set of all evaluations of the domain of F given a puncture guess of j.

□

4 Applying wpPRFs to PIR

In this section we focus on showing how to utilize our new primitive, the wpPRF, to achieve a PIR scheme with the complexities outlined in Figure 1. We first show how our wpPRFs can be used to construct pseudorandom sets, and then use these sets to build $\mathsf{TreePIR}$.

4.1 Constructing Pseudorandom Sets from wpPRFs

As we glanced over in the introduction, all current PIR schemes in the client-preprocessing model use some notion of pseudorandom sets. Here, we explore how we can construct these sets from our new wpPRF primitive. In a general sense, we want sets that satisfy the following properties:

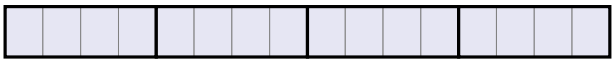
– Have a short description.
– Maintain a short description after removing one element.
– Support fast (non-trivial) membership testing.

Our approach. Here we show how to use wpPRFs to address the shortcomings of prior work. Suppose we have a wpPRF $F := (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval})$ whose domain and range is $\sqrt{N}$. We can then define a set S of $\sqrt{N}$ elements in $[N]$ using F. For exposition, let N be an even power of two. Given a uniform random key $k \in \{0,1\}^\lambda$, we define our set S as:

$$S = \left\{ i || F.\mathsf{Eval}(k, i) : i \in \left[\sqrt{N}\right] \right\} .$$

The $||$ notation concatenates the binary representation of both numbers. The set S will contain each element in $[N]$ with probability $1/\sqrt{N}$. Also the set will be "partitioned" within $[N]$, and will contain exactly one element for each interval of size $\sqrt{N}$ within $[N]$. We look at a small example below to aid in our exposition.

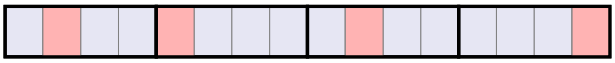
Let us take an example where $N = 16$. We represent the database with a box for each index below. The darker boxes represent chunks of size $\sqrt{N} = 4$ indices:



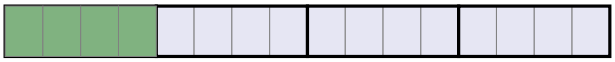
We pick some uniform k, and evaluate it at $\sqrt{16} = 4$ points, such that

$$F.\mathsf{Eval}(k, 00) = 01, F.\mathsf{Eval}(k, 01) = 00, F.\mathsf{Eval}(k, 10) = 01, F.\mathsf{Eval}(k, 11) = 11.$$

Then, our set would be $S = \{0001, 0100, 1001, 1111\}$. The coverage with respect to the database would look as follows:



As aforementioned, we have exactly one element within each of the darker boxes. This is intrinsic to our set definition. Now, assuming we were using a regular GGM style pPRF, a puncture to a point would reveal its 'box'. Let us say we want to puncture the element 0001 from the set. To do this, we run $F.\mathsf{Puncture}(k, 00) = k_{00}$. Using a regular pPRF, k_{00} does not reveal the evaluation 01, but it does reveal the punctured index. What this means is that given the punctured pPRF key, one can infer that the element removed from the set is within the green elements below:



In the context of PIR, this would enable the adversary to narrow down the query index to $\sqrt{N}$ indices.

Intuitively, because our wpPRF enables us to also hide the point that was punctured, we hide both the index within the partition *and* which partition we are puncturing. If we define the set using a wpPRF key, a punctured key would reveal no information about what element was removed from the set.

To summarize, our set, as defined above, initialized with a wpPRF key, satisfies the following properties:

1. It can be represented in λ bits by its key k.
2. We can check membership with one wpPRF evaluation. For any $x = x^\ell || x^r \in [N]$, x will be part of S if and only if $F.\mathsf{Eval}(k, x^\ell)$ evaluates to x^r.
3. If we puncture at a point x (by puncturing position x^ℓ as defined above), the punctured key remains concise, and reveals nothing about the punctured point or the punctured index (Definition 2.6, 3.2).

Applying our new sets to PIR We now explore how to use a punctured key to retrieve a desired database index value. Recall that our set is defined as:

$$S = \left\{ i || F.\mathsf{Eval}(k, i) : i \in \left[\sqrt{N}\right] \right\} .$$

We want to find the value $\mathsf{DB}[x]$ for some $x \in S$, note that for $x = x^\ell || x^r$, it follows from the set definition above that:

$$x \in S \iff F.\mathsf{Eval}(k, x^\ell) = x^r .$$

Suppose we happen to have the respective set parity:

$$p = \bigoplus_{i \in S} \mathsf{DB}[i] .$$

Let us now define :

$$p_t = \bigoplus_{i \in S \setminus \{t\}} \mathsf{DB}[i] .$$

To retrieve $\mathsf{DB}[x]$, where $x = x^\ell || x^r = x^\ell || F.\mathsf{Eval}(k, x^\ell)$, we first send $k_{x_\ell} \leftarrow F.\mathsf{Puncture}(k, x^\ell)$ to the server. Then, without revealing x to the server, we must have the server compute p_x. This would allow us to locally compute $\mathsf{DB}[x] = p \oplus p_x$. Since we are using a wpPRF, k_{x^ℓ} does not allow the server to compute p_x. However, because the wpPRF that we are using satisfies efficient full evaluation, per Definition 3.4, the server uses k_{x_ℓ} and computes all $\sqrt{N}$ values S_j (and thus all p_j) in $O(\sqrt{N} \log N)$ time.

We have successfully reduced the problem of fetching a record privately from a database of N records to fetching a record privately from a database of $\sqrt{N}$ records. There are two different ways we can proceed from here.

1. Download all the $\sqrt{N}$ parities.
2. Use a single-server PIR scheme to fetch the record p_x from the smaller database. The record we want from this smaller database p_x is exactly the x^ℓ-th index.

Using approach number two, our bandwidth then becomes the size of the wpPRF key (which is $\lambda \log N$) plus whatever bandwidth is incurred from the single-server PIR scheme used. Since the single-server PIR is run over a smaller database of size $\sqrt{N}$, the server cost of each query is still sublinear in N.

4.2 Our TreePIR Scheme

In Figure 6, we give the full scheme, based on the intuition above. For the scheme. We assume that N is an even power of two, so that $\sqrt{N}$ is a natural number and that x's bit representation can be split in half. We discuss how to generalize to any size of database in Appendix A. We note that wpPRFs makes our scheme considerably simpler than previous schemes based on the same paradigm [15, 33, 43], since we do not require "failing" with certain probability and executing a secondary protocol or executing λ instances in parallel.

We argue our scheme's privacy and correctness in Theorem 4.1.

Theorem 4.1 (TreePIR). *Assuming Theorem 3.1, TreePIR, for any $N \in \mathbb{N}$ which is an even power of two, and security parameter λ, our scheme given in Figure 6 satisfies correctness and privacy for multi-query PIR schemes as defined in Definition 2.1, 2.2 and its complexities are:*

- $O(\lambda N \log N)$ *offline server time and* $O(\lambda\sqrt{N})$ *offline client time.*
- $O(\sqrt{N} \log N)$ *online server time and* $O(\sqrt{N} \log N)$ *probabilistic online client time.*
- *No additional server space and* $O(\sqrt{N})$ *client space.*
- $O(\lambda\sqrt{N})$ *offline bandwidth.*
- $O(\lambda \log N)$ *upload bandwidth and* $\sqrt{N}$ *download bandwidth.*

Proof. Our efficiencies follow directly from construction and from Theorem 3.1. We specifically highlight that Step 2 of the Answer algorithm runs in $O(\sqrt{N} \log N)$ time by the efficient full evaluation property. In Step 1 of our online query phase, we run the $F.\mathsf{Gen}(\cdot)$ *until* we find the mapping from x^ℓ to x^r. As is, this runs in *probabilistic* $O(\sqrt{N})$ time.[6]

Privacy with respect to $\boldsymbol{server}_1$. Offline, $\mathbf{server}_1$ sees nothing, so we do only consider online privacy. We first show that for the first query, we satisfy the indistinguishability experiment. Then, since we show how to induct on this argument and extend it for any polynomial number of queries. Assume the adversary picked query index x_1 for query Q_1. Now consider the hybrid experiment below:

- **Hyb**: **client** interacts with $\mathcal{A}$ who acts as $\mathbf{server}_1$ and $serv_0$ who acts as $\mathbf{server}_0$. **client** is invoked with query x_1, and instead of finding a key that contains x to puncture and send to $\mathbf{server}_1$, it instead samples a fresh key $k^* \leftarrow F.\mathsf{Gen}(1^\lambda)$, punctures it at x_1, and sends $q_1^* \leftarrow F.\mathsf{Puncture}(k^*, x_1)$ to $\mathbf{server}_1$.

By security in puncturing (Definition 2.6), our wpPRF key punctured key reveals nothing about the evaluation at the punctured point. A distinguisher $\mathcal{D}$ that distinguishes $\mathbf{Expt}_0$ and **Hyb** can be used to break the security in puncturing experiment through the following steps:

[6] Client time is probabilistic because sampling a set that contains x takes $O(\sqrt{N})$ time probabilistically by naively sampling keys and testing until we find one that contains x. We discuss an optimization to this naive approach in Appendix A.

Our TreePIR construction.

Let $F := (\mathsf{Gen}, \mathsf{Eval}, \mathsf{Puncture}, \mathsf{PEval})$ be a wpPRF as defined in Definition 3.1 that maps $\sqrt{N}$ bits to $\sqrt{N}$ bits. Let $M = \lambda\sqrt{N}$ and DB be a vector of N indices.

• Offline: Preprocessing(**client**,**server**$_0$).

1. **client** sets $k_i \leftarrow F.\mathsf{Gen}(1^\lambda)$, $i = 1, \ldots, M-1$, and sends $k_0, \ldots, k_{M-1}$ to **server**$_0$.
2. For all $i = 0, \ldots, M-1$ **server**$_0$ computes parities

$$p_i = \bigoplus_{j \in S_i} \mathsf{DB}[j]\,,$$

where

$$S_i = \left\{ v || F.\mathsf{Eval}(k_i, v) : v \in \left[\sqrt{N}\right] \right\}\,,$$

and sends $p_0, \ldots, p_{M-1}$ to **client**.
3. **client** stores pairs of keys and parities in a table

$$\mathsf{T} = \{T_j = (k_j, p_j) : j \in [M]\}\,.$$

• Online: Query(**client**, x) $\rightarrow (q_0, q_1)$, $x \in [N]$.

1. Sample $k' \leftarrow F.\mathsf{Gen}(1^\lambda)$ until $F.\mathsf{Eval}(k', x^\ell) = x^r$.
2. Find T_j such that $F.\mathsf{Eval}(k_j, x^\ell) = x^r$.
3. Let $q_0 \leftarrow F.\mathsf{Puncture}(k', x^\ell)$.
4. Let $q_1 \leftarrow F.\mathsf{Puncture}(k_j, x^\ell)$.
5. Send (q_0, q_1) to **server**$_0$ and **server**$_1$ respectively.

• Online Answer(**server**$_b$, q_b) $\rightarrow \mathcal{P}_b$ for $b \in \{0, 1\}$.

1. Parse $q_b = (k_{punc})$.
2. Compute an array of parities $\mathcal{P}_b = [\phi_0, \ldots, \phi_{\sqrt{N}}]$, where $\phi_i = \bigoplus_{j \in S_i} \mathsf{DB}[j]$,

$$S_i = \left\{ v \,||\, F.\mathsf{PEval}(k_{punc}, i, v) \;:\; \text{for } v \in \left[\sqrt{N}\right] \right\}_{i \in \left[\sqrt{N}\right]}\,.$$

3. Return $\mathcal{P}_b$.

• Online Reconstruct(**client**, $\mathcal{P}_0, \mathcal{P}_1$) $\rightarrow \mathsf{DB}[x]$.

1. $\mathsf{DB}[x] \leftarrow p_j \oplus \mathcal{P}_1[x^\ell]$.
2. $T_j \leftarrow (k', \mathcal{P}_0[x^\ell] \oplus \mathsf{DB}[x])$.

Fig. 6. Our novel PIR scheme, TreePIR.

1. Send point x_1^ℓ to the security in puncturing experiment, get back $(k_{x_1^\ell}, u)$.
2. Send $k_{x_1^\ell}$ and $x_1^\ell || u$ to $\mathcal{D}$; if $\mathcal{D}$ outputs $\mathbf{Expt}_0$, output 0. Else, output 1.

Note that the probability that we output correctly is exactly the probability that $\mathcal{D}$ can distinguish whether u is uniform or the original evaluation of $k_{x_1^\ell}$ at x_1^ℓ. Then, since a PPT algorithm $\mathcal{D}$ that can distinguish between $\mathbf{Expt}_0$ and $\mathbf{Hyb}$ allows us to break the security in puncturing experiment, it follows that assuming security in puncturing, $\mathbf{Expt}_0$ and $\mathbf{Hyb}$ are computationally indistinguishable.

Next, we define our algorithm Sim as follows.

– Sim: Run $k \leftarrow F.\mathsf{Gen}(1^\lambda)$. Let α be an element sampled uniformly from $\left[\sqrt{N}\right]$. Output $q_{sim} \leftarrow F.\mathsf{Puncture}(k, \alpha)$.

Now, if we show that there Sim is computationally indistinguishable from $\mathbf{Hyb}$, we have shown privacy with respect to $\mathbf{server}_1$. Note that both k and k^* are sampled from Gen. Now, suppose there exists a distinguisher $\mathcal{D}$ that can distinguish between Sim and $\mathbf{Hyb}$. Then, we can use $\mathcal{D}$ to break our privacy in puncturing experiment Definition 3.2 as follows:

1. Send to the privacy in puncturing experiement the points $(\sigma_0 := \alpha^\ell, \sigma_1 := x_1^\ell)$ and get back a key k_{σ_b}.
2. Send k_{σ_b} to $\mathcal{D}$. If $\mathcal{D}$ outputs Sim, output 0, else output 1.

Again, we see that advantage in the experiment corresponds exactly to the privacy in puncturing experiment, and thus a PPT algorithm $\mathcal{D}$ that distinguishes between Sim and $\mathbf{Hyb}$ allows one to break the privacy in puncturing property of the underlying wpPRF. Then, by contrapositve, assuming privacy in puncturing of the wpPRF, Sim and $\mathbf{Hyb}$ are computationally indistinguishable from the point of view of $\mathbf{server}_1$. Finally, we have shown privacy with respect to $\mathbf{server}_1$ for the first query, since our protocol is computationally indistinguishable from an algorithm Sim that runs without knowledge of x_1.

For subsequent queries, we replace our used key k with a new key $k' \leftarrow F.\mathsf{Gen}(1^\lambda)$ *until* $F.\mathsf{Eval}(k', x^\ell) = x^r$. But since our key, k, that was used in the first query can also *be seen as* the output of $F.\mathsf{Gen}(1^\lambda)$ *until* $F.\mathsf{Eval}(k, x_1^\ell) = x^r$, because it was the *first* key generated that contained x_1, we note that k and k' are computationally indistinguishable and therefore swapping k for k' maintains the distribution of T. Then, by induction, since each query maintains the distribution of T, by the same argument as above we conclude that for any sequence of queries $\{1, \ldots, t\}$, our scheme satisfies privacy with respect to $\mathbf{server}_1$.

Privacy with respect to $\boldsymbol{server}_0$. Offline privacy follows directly from the fact that the keys are picked before any query, and therefore cannot leak any information. Online privacy with respect to $\mathbf{server}_0$ can be argued symmetrically from the same arguments as privacy with respect to $\mathbf{server}_1$. The only difference is that we have to be careful to pick fresh keys from a *different* randomness so they are independent from the keys sent to $\mathbf{server}_0$ offline.

Correctness. We argue correctness by construction and Theorem 3.1, using an induction argument on the client's state.

Let us first consider the first query Q_1 to index x_1. For any query index x_1, the probability that we *do not* find a set that contains x_1, for some negligible function $\nu(\cdot)$, is:

$$\Pr\left[x_1 \notin \{S_i\}_{i\in[\lambda\sqrt{N}]}\right] = \Pr\left[\forall i \in [\lambda\sqrt{N}],\, F.\mathsf{Eval}(k_i, x_1^\ell) \neq x_1^r\right] \tag{1}$$

$$= \left(1 - \frac{1}{\sqrt{N}}\right)^{\lambda\sqrt{N}} \tag{2}$$

$$\leq \left(\frac{1}{e}\right)^{\lambda} \leq \nu(\lambda). \tag{3}$$

This means that Step 2 in our Query algorithm will always succeed for the first query except with negligible probability. Then, by construction of our scheme and weak correctness of our wpPRF (Definition 3.3), it follows that, if $x_1 \in S_j = \left\{i \,||\, F.\mathsf{Eval}(k_j, i) : i \in \left[\sqrt{N}\right]\right\}$, then:

$$\mathsf{DB}[x_1] = \left(\bigoplus_{i\in S_j} \mathsf{DB}[i]\right) \oplus \left(\bigoplus_{i\in S_j\setminus\{x_1\}} \mathsf{DB}[i]\right) \tag{4}$$

$$= p_j \oplus \left(\bigoplus_{k\in S_{j,x_1^\ell}} \mathsf{DB}[k]\right) \tag{5}$$

$$= p_j \oplus \mathcal{P}_1[x_1^\ell], \tag{6}$$

where:

$$S_{j,x_1^\ell} = \left\{i \,||\, F.\mathsf{PEval}(k_{j,x_1^\ell}, x_1^\ell, i) : i \in \left[\sqrt{N}\right]\right\},\ k_{j,x_1^\ell} = F.\mathsf{Puncture}(k_j, x_1^\ell).$$

We have shown that the first query Q_1 to index x_1 is correct except with negligible probability. At the end of the query, we update T by setting $T_j = (k', \mathcal{P}[x^\ell] \oplus \mathsf{DB}[x])$. Correctness of the parity follows in a similar argument as above. Also, our updated table T maintains its distribution, and holds only sets never seen by $\mathbf{server}_1$, as we have shown in the privacy proof. Then, it follows that the next query Q_2 to index x_2 will also be correct by the same argument as above. By induction, this will hold for query Q_t to index x_t for any $t < \frac{1}{\nu(\lambda)}$, for any negligible function $\nu(\cdot)$. □

4.3 Sublinear Time, Polylog Bandwidth PIR from the DDH Assumption

Prior works [19, 39] have studied single-server PIR and have achieved schemes with polylogarithmic bandwidth. The bottleneck of these schemes is that the server time grows linearly with the database size. Applying TreePIR with one of these schemes, we can achieve a practical PIR scheme with sublinear time

and polylogarithmic bandwidth. Applying the scheme by Döttling et al. [19] we achieve our claimed result, a two-server PIR scheme with sublinear online time and polylog bandwidth, reliant only on the DDH assumption [16]. This is reflected on the following lemma:

Lemma 4.1 (Sublinear time, polylog bandwidth PIR from DDH). *Assuming the Decisional Diffie-Hellman problem is hard, Theorem 4.1 implies a two-server PIR with the same complexities except with polylogarithmic online bandwidth.*

Proof. We can replace the last step of the server answer in our protocol with a single-server PIR that has linear work and polylogarithmic bandwidth. This is because we know what index we want from the string of $\sqrt{N}$ words ahead of time, it corresponds to exactly x^ℓ. The protocol then replaces the last step of downloading $\sqrt{N}$ words with fetch the x^ℓ-th word using a single-server scheme. The privacy, efficiency, and correctness follow from Theorem 4.1 and previous work on single-server PIR [12, 19, 22, 36, 39]. This means that we also have to introduce any assumptions used by the scheme selected. By recursing with the single-server PIR scheme by Döttling et al., which has polylog bandwidth and relies only on DDH, we achieve the claimed complexities. □

On Section 5 we benchmark the performance of our TreePIR paired with SPIRAL [39]. Note that we *cannot* recurse with a PIR scheme that uses preprocessing based on the database elements (and this includes our TreePIR), since the $\sqrt{N}$ words from the last step of the Answer phase are dynamically generated and entirely dependent on the index we decide to query.

4.4 Tuning Efficiencies in TreePIR

We have picked wpPRFs of domain and range $\sqrt{N}$ so that we achieve $O(\sqrt{N}\log N)$ server time and $O(\sqrt{N})$ client space. This is not the only tradeoff supported by TreePIR. We can trade off client storage and online client time for online server time and bandwidth. If we change our set size to N^D, then this makes our online time and bandwidth N^D. In exchange, we get client storage and online client time proportional to N^{1-D}. This will work for any D in the range $(0, 1)$. Intuitively, this says that if we have smaller sets in the client, we get faster server time, but we need more sets at the client to ensure coverage of all indices. Conversely, to have less sets at the client, they need to represent more indices and we would have to pay for it in server time. For this work we fixed the $D = 1/2$ tradeoff. Ideal tradeoffs between the parameters depend largely on application and therefore for the rest of this work we fix this tradeoff.

5 Performance

In this section, we benchmark both TreePIR as introduced in Section 4 and TreePIR paired with SPIRAL to reduce the communication of the last step,

on databases of different sizes and with different size elements (not only bits as defined previously). As we will see, there are use cases where plain TreePIR performs well, specifically for databases with small elements.

Our tests. We implement TreePIR in 530 lines of C++ code and 470 lines of Go code. The source code is available on GitHub [1]. Our starting point was the previous optimized implementations of PIR by Kogan and Corrigan Gibbs [33] and Kales et al. [30]. Given that the only client-preprocessing scheme with comparable asymptotics does not have any known implementation, we benchmark PIR with two other client-preprocessing PIR schemes: PRP-PIR [15] implemented by Ma et al. [38], which has online bandwidth of $O(\sqrt{N}\log N)$ and requires parallel instantiations, and Checklist by Kogan and Corrigan-Gibbs [33], which requires persistent client storage proportional to the number of elements in the database ($O(N \log N)$ client storage). Across all tests, it will be clear that in many use cases of PIR, TreePIR provides the best alternative out of the three. The tests results reflect microbenchmarks run on a single thread in an Amazon Web Services EC2 instance of size m5d.8xlarge.

Comparison with Shi et al. [43]. The only known client-preprocessing PIR scheme with comparable asymptotics to TreePIR is the scheme by Shi et al. [43]. However, we do not benchmark the Shi et al. scheme [43] because there is no known implementation of the Privately Puncturable PRF primitive. Given the sample parameter instantiation of privately puncturable PRFs by [7], a conservative estimate on the online bandwidth is of at least $2\lambda^4 \log(\lambda) \cdot \log N$. This means an online per query communication cost of over 400 megabytes, given a security parameter of size 128 bits, for any database size. This means that by our estimates the communication using TreePIR presents a communication of 8,000x or more over the scheme by Shi et al. in all databases benchmarked (as will be clear later in the section). This large communication is largely due to the underlying primitive, the *privately puncturable PRF*, which means that improvements in the privately puncturable PRF construction imply improvements to their scheme. However, for now, the best known constructions yield the complexities discussed above.

5.1 TreePIR with No Recursion

First, we consider TreePIR as outlined in Figure 6, without recursion. Without using a second PIR scheme to recurse in the online phase, TreePIR incurs $O(\sqrt{N})$ online bandwidth, since we are required to download the parity for each "potential set". Although asymptotically suboptimal, in applications where the database elements are very small, it actually outperforms not only other schemes, but also the recursive solution, in both time and bandwidth.

One such application was recently introduced by Henzinger et al. [27]. Henzinger et al. study the use of PIR for secure certificate transparency (SCT) auditing. Their protocol for SCT requires the use of PIR over a database of

2^{33} elements of size only 1 bit. Another application that might involve large databases of 1 bit entries would be compromised credential checking services, among other usecases where we are basically 'checking membership' using PIR, but the query is sensitive.

The work by Henzinger et al. [27] considers the problem given only one server. Here we consider a similar-sized database in the client-preprocessing, two-server scenario. We provide evaluations for such scenario on how plain TreePIR (no recursion) compares to another two state-of-the-art two-server client preprocessing schemes, Checklist [33] and PRP-PIR [15], implemented by Ma et al. [38] in Figure 7. We benchmark using a similar sized database of 2^{32} 1-bit elements. We note that PRP-PIR query time is marked with a '−' because we were not able to benchmark PRP-PIR time in this experiment, the implementation did not support the large database size .

Results for database of 2^{32} 1-bit elements.

Protocol	**Amortized Query Time**	**Client Storage**	**Online Bandwidth**
Checklist	12574ms	8.6GB	0.51KB
PRP-PIR	-	1.05MB	33.5MB
TreePIR	3508ms	1.05MB	16.6KB

Fig. 7. Amortized query time for a large database of small elements. Query time is amortized per client, over 2000 queries.

After an initial offline phase, queries are run in $\sqrt{N}\log N$ server time.[7] Although not asymptotically optimal, for this usecase, and other usecases with very small elements, incurring $\sqrt{N}$ is better than recursing with a single-server scheme in practice. When comparing with previous approaches to client-preprocessing PIR, such as Checklist, TreePIR incurs additional communication, but makes up for it in both query speed and client space used. Note that using Checklist for this usecase would allow for queries with half a kilobyte of bandwidth, but since Checklist incurs client storage proportional to $N \log N$, this would mean persistent client storage larger than the size of the database (upwards of 8 gigabytes, as seen in Figure 7) to perform the queries. In contrast, TreePIR incurs persistent client storage of only around one megabyte to perform its queries. Therefore, in this case, compared to Checklist TreePIR reduces persistent client storage by over 8,000x. Additionally, TreePIR improves total query time by more than 3.5x. We pay for this in communication, but a price small enough to still be practical. When compared to PRP-PIR, we see a very large gain in online bandwidth, largely due to the fact that while TreePIR has bandwidth proportional to $\sqrt{N}$, PRP-PIR requires $\lambda\sqrt{N}\log N$ bandwidth in order to perform the same query. With respect to client storage, both store PRP-PIR and TreePIR store the same information, $\lambda\sqrt{N}$ parities and a random seed of size λ to generate the sets.

[7] This is amortized per client.

5.2 Recursing to Improve Bandwidth

Next, we run benchmarks for larger database sizes, where we use TreePIR paired with a single-server PIR scheme to reduce the bandwidth. We found SPIRAL [39] to be the most suitable scheme to recurse with in practice. We run the analysis of amortized query time across two thousand queries for a database of four million elements of 256 bytes (Figure 8) and a database of 268 million elements of 32 bytes (Figure 9). Running benchmarks on both databases with a large collection of small elements and a smaller selection of large elements is common practice to test the flexibility of the PIR scheme.

Results for database of 2^{22} 256-byte elements.

Protocol	Amortized Query Time	Client Storage	Online Bandwidth
Checklist [33]	140ms	78MB	0.7KB
PRP-PIR [15, 38]	315ms	67MB	721KB
TreePIR + SPIRAL	(89+61)ms	67MB	50KB

Fig. 8. Amortized query time for moderate database size. Query time is amortized over 200 queries.

Results for database of 2^{28} 32-byte elements.

Protocol	Amortized Query Time	Client Storage	Online Bandwidth
Checklist [33]	711ms	570MB	0.3KB
PRP-PIR [15, 38]	-	67MB	7.3MB
TreePIR + SPIRAL	(251+61)ms	67MB	50KB

Fig. 9. Amortized query time for large database of small elements. Query time is amortized over 2000 queries.

The times seen in Figure 8 and Figure 9 for TreePIR + SPIRAL represent the time that each takes, respectively. As shown in the figures, to recurse with SPIRAL, we pay 61ms on a database of 2^{11} elements of size 256 bytes, and the same 61ms to recurse on a database of 2^{14} elements of size 32 bytes. This means that, in reducing bandwidth, our amortized query time would be slower than Checklist's for the databases of 2^{22} elements of 256 bytes, but still considerably faster for a database with 2^{28} elements of size 32 bytes each. In total then, Checklist outperforms TreePIR for small databases of large elements. However, once the database is scaled, the client storage incurred by Checklist is upwards of half a gigabyte. For many client use-cases, such as mobile phones and even laptop computers, half a gigabyte of storage is extremely undesirable. In such

cases, TreePIR provides an alternative with faster query times and small client storage, at the cost of higher bandwidth per query. Furthermore, the overhead seen in the experiment is largely due to the security parameters, meaning that TreePIR with recursion can support larger database elements with much less overhead than these small ones.

In cases where the database size is small but its elements are large, Checklist still presents itself as a very good candidate. Whenever the size of the database is large or the elements are small, TreePIR provides the best trade-offs in practice, either through recursion or just using it plainly. Below we provide additional remarks with respect to the schemes that we compared against.

On the performance of PRP-PIR [15]. To benchmark PRP-PIR [15], we use a separate library provided by Ma et al. [38]. This library does not use optimized instructions to perform the xor operation and that could partially explain its poor performance. The other factor is that small-domain PRPs [41, 46] put overhead in the membership testing and evaluation, both of which are performed numerous times thoroughout the scheme. We were not able to successfully benchmark the times for PRP-PIR against databases with more than 2^{22} elements.

On the performance of Checklist [33]. Checklist will always have a shorter faster online server time than TreePIR by construction. However, when running many queries on large databases, such as was the case to benchmark, the Checklist query time is inconsistent. The hashmap used to find the set with the desired query index does not contain a full mapping of every set, it only contains one entry per index. If x and y are in the same set i with the map pointing both x and y to set i, a query to x will make the mapping of y invalid on the map with very high probability. This means that a query to y after a query to x requires enumerating $\sqrt{N}$ sets in expectation and therefore requires around linear client work. This explains why over many queries, TreePIR outperforms Checklist. This problem could be fixed by keeping a full mapping of all sets in the hashmap, although this would require an additional λ factor of client storage, bringing Checklist's storage up to $\lambda N \log N$. We do not benchmark this scenario since the client storage would be too large, but we note that the asymptotics reported in Figure 1 reflect this latter case, since without this extra λ factor, Checklist's client time is $O(N)$ in the worst case.

5.3 Supporting Changing Databases

Techniques to support preprocessing in databases that change over time have been studied in previous work [33, 38]. These techniques can also be applied to TreePIR and are able to maintain most of the benefits of preprocessing with small overhead.

In specific, the technique introduced by Kogan and Corrigan-Gibbs [33] is a waterfall-based approach to updates also used in other related primitives such as oblivious RAM [24] and searchable encryption [45]. We re-iterate the main ideas of the technique here, and refer to previous works for a complete analysis.

Update types. A changing database can be given three different kinds of operations: $\mathsf{Add}(i, v), \mathsf{Remove}(i)$ or $\mathsf{Edit}(i, v)$.

PIR by keywords. The first step in supporting updates via a waterfall approach is converting our classical PIR algorithm TreePIR algorithm. Chor et al. [14] showed that this can be done in a blackbox fashion with $O(\log N)$ overhead for almost all modern PIR schemes.

Initializing subdatabases. We initialize i 'subdatabases'. The i-th subdatabase is of size 2^i, for $i \in \{1, \ldots, \log N\}$ (for simplicity, again, we assume that N is a power of two). We will refer to the i-th subdatabase as the i-th layer. Initially, every layer is empty except for the $\log N$-th layer, which stores the whole database. For each query, the client must send its query to each layer. Here we incur another constant factor of overhead in client space, client time, server time and bandwidth, to query each layer.

Updating the database. For each update to the database, the update is directed to the 0-th layer. If the 0-th layer is empty, then the client runs the preprocessing over the 0-th layer and is done. Else, the update is directed to the next non-full layer j, along with the updates from all other layers from 0 to j, and the preprocessing is redone for the j-th layer, and zeroed out for layers 0 through i. The updates to the database defined by a tuple consisting of a key (the index to the updated) and value (the value of this index). If there are conflicts, they are resolved upon merging (with priority given to the newest values, the same is true when clients receive conflicting values from different layers for the same index). We can reserve a special value for deletion to support deletes as well. Note that in this manner, updates are amortized, and we go through N updates before having to re-run the preprocessing for the whole database again. In Checklist [33], they show that the update costs are manageable. Furthermore, [24] show how to deamortize these more expensive preprocessing steps over multiple updates.

Another approach suitable for some applications, outlined by Henzinger et al. [27] is to have a pre-determined update schedule. In the SCT application discussed before, for example, it is acceptable to update the certificates monthly, meaning that we would perform the expensive preprocessing phase only once a month for each client.

Acknowledgements. This research was supported by the National Science Foundation, the Algorand Foundation and Protocol Labs. We thank Samir Menon for a helpful exchange regarding SPIRAL and the reviewers for helping improve our work.

A Further Optimizations

We discuss here some further optimizations to TreePIR.

A.1 Deterministic Client Time

Our protocol in Figure 6 has probabilistic client time due to Step 1 of the Online Query algorithm, which is:

1. Sample $k' \leftarrow F.\mathsf{Gen}(1^\lambda)$ until $F.\mathsf{Eval}(k', x^\ell) = x^r$.

In practice, sampling several keys until finding one can be time consuming, and as N increases, the worst case run-time can be very expensive. To achieve both faster and more consistent run-times, it is desirable to have a fully deterministic PIR algorithm. This is achievable by introducing an additional parameter to each of our 'sets', a *shift*. The shift will permute every element in the set by a fixed offset (this technique was used before in [15]). We modify the TreePIR by include a shift $s \in \left[\sqrt{N}\right]$ to be a part of every pseudorandom set (which is now defined as a tuple of a wpPRF key and a shift).

Offline, the client now generates tuples (k_i, s_i) for $i = 1, \ldots, M-1$; where $k_i \leftarrow F.\mathsf{Gen}(1^\lambda)$ as before, and s_i is sampled uniformly from $\left[\sqrt{N}\right]$. Then, for all $i = 0, \ldots, M-1$, $\mathbf{server}_0$ computes the appropriate parities p_i as before, except we now define our set S_i as:

$$S_i = \left\{ v || \left(F.\mathsf{Eval}(k_i, v) \oplus s_i\right) : v \in \left[\sqrt{N}\right] \right\}.$$

The membership check also is changed accordingly. Finally, the reason for this change is that we can now run Step 1 of the online query as:

1. Sample $k' \leftarrow F.\mathsf{Gen}(1^\lambda)$. Let $s' = x^r \oplus F.\mathsf{Eval}(k', x^\ell)$.

Note that this guarantees that we generate a set with x sampling only a single k' instead of an expected $\sqrt{N}$ (and potentially many more) different keys. We sketch the privacy proof here and include refer to the full version of the paper [35] for a complete proof. For sets generated offline, the shifts are sampled uniformly at random, and therefore do not affect privacy or correctness for the initially generated sets, they only shift all elements of the initial sets by a fixed offset. However, in Step 1 of the Online Query, s' is dependent on x, our query index.

Then, we must now show that upon sending such tuple to $\mathbf{server}_0$ does not reveal any additional information. We must show that the tuple (q_0, s') can be simulated without knowledge of x. This follows from the fact that we can replace our q_0 by some freshly sampled key punctured at a uniformly sampled point in the functions domain. We denote u to be a point sampled uniformly from the range of the PRF. Then, since $F.\mathsf{Eval}(k', x^\ell)$ for a freshly sampled key is computationally indistinguishable from u given only q_0 (follows from Definition 2.6), and completely independent from x (since we are using a fresh sample key), from the server's view, $s' = x^r \oplus u$. Since the xor operation is randomness preserving, we can replace the whole s' by a uniformly sampled point in $\left[\sqrt{N}\right]$. Then, if we do this for the Sim algorithm, we have shown that our query is computationally indistinguishable from a query generated without knowledge of x. The rest of the proof follows as in Section 4.

A.2 Generalizing TreePIR to More Flexible Database Sizes

In Section 4 we assume that N is a perfect square *and* a power of two for simplicity and exposition. This allows us to use concatenations and splitting to go between our index x and the building blocks x^ℓ and x^r. With some extra steps, TreePIR can be generalized to work with any N that is a perfect square by replacing the concatenation operation by a multiplication by $\sqrt{N}$ and addition by the function evaluation value. Our sets S_i are therefore now defined as:

$$S_i = \left\{ v * \sqrt{N} + F.\mathsf{Eval}(k_i, v) : v \in \left[\sqrt{N}\right] \right\}.$$

Here, $*$ and $+$ are plain addition and multiplication over the natural numbers. Checking membership is done in the corresponding fashion. For an index $x \in [N]$, let $x^\ell = \left\lfloor x/\sqrt{N} \right\rfloor$ (where $\lfloor \cdot \rfloor$ denotes the floor function, rounding *down* to the nearest integer). We can check if x is in S_i by checking whether $x - x^\ell * \sqrt{N} = F.\mathsf{Eval}(k_i, x^\ell)$

If a database size N is *not* a perfect square, one can simply use the domain and range of F to be $\left\lceil \sqrt{N} \right\rceil$ with little to no overhead at the client or server and treat elements larger than N as 0-strings (when necessary for calculating parities). We use $\lceil \cdot \rceil$ to denote the ceil function, rounding *up* to the nearest integer.

References

1. Source code for TreePIR, `https://github.com/alazzaretti/treePIR`
2. Aguilar-Melchor, C., Barrier, J., Fousse, L., Killijian, M.O.: XPIR : Private Information Retrieval for Everyone. Proceedings on Privacy Enhancing Technologies **2016**(2), 155–174 (Apr 2016). `https://doi.org/10.1515/popets-2016-0010`, `https://petsymposium.org/popets/2016/popets-2016-0010.php`
3. Angel, S., Setty, S.: Unobservable communication over fully untrusted infrastructure. In: Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation. pp. 551–569. OSDI'16, USENIX Association, USA (Nov 2016)
4. Backes, M., Kate, A., Maffei, M., Pecina, K.: ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In: 2012 IEEE Symposium on Security and Privacy. pp. 257–271 (May 2012). `https://doi.org/10.1109/SP.2012.25`, iSSN: 2375-1207
5. Beimel, A., Ishai, Y.: Information-Theoretic Private Information Retrieval: A Unified Construction. In: Goos, G., Hartmanis, J., van Leeuwen, J., Orejas, F., Spirakis, P.G., van Leeuwen, J. (eds.) Automata, Languages and Programming, vol. 2076, pp. 912–926. Springer Berlin Heidelberg, Berlin, Heidelberg (2001). `https://doi.org/10.1007/3-540-48224-5_74`, `http://link.springer.com/10.1007/3-540-48224-5_74`, series Title: Lecture Notes in Computer Science
6. Beimel, A., Ishai, Y., Malkin, T.: Reducing the Servers Computation in Private Information Retrieval: PIR with Preprocessing. In: Bellare, M. (ed.) Advances in Cryptology — CRYPTO 2000. pp. 55–73. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2000). `https://doi.org/10.1007/3-540-44598-6_4`

7. Boneh, D., Kim, S., Montgomery, H.: Private Puncturable PRFs from Standard Lattice Assumptions. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 415–445. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_15
8. Boneh, D., Lewi, K., Wu, D.J.: Constraining Pseudorandom Functions Privately. In: Fehr, S. (ed.) Public-Key Cryptography – PKC 2017. pp. 494–524. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54388-7_17
9. Boyle, E., Gilboa, N., Ishai, Y.: Function Secret Sharing. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology - EUROCRYPT 2015. pp. 337–367. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46803-6_12
10. Boyle, E., Ishai, Y., Pass, R., Wootters, M.: Can We Access a Database Both Locally and Privately? pp. 662–693 (Nov 2017). https://doi.org/10.1007/978-3-319-70503-3_22
11. Brakerski, Z., Tsabary, R., Vaikuntanathan, V., Wee, H.: Private Constrained PRFs (and More) from LWE. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography. pp. 264–302. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_10
12. Cachin, C., Micali, S., Stadler, M.: Computationally Private Information Retrieval with Polylogarithmic Communication. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. pp. 402–414. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48910-X_28
13. Canetti, R., Chen, Y.: Constraint-Hiding Constrained PRFs for NC^{1}from LWE. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 446–476. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-56620-7_16
14. Chor, B., Gilboa, N., Naor, M.: Private Information Retrieval by Keywords (1998), https://eprint.iacr.org/1998/003, report Number: 003
15. Corrigan-Gibbs, H., Kogan, D.: Private Information Retrieval with Sublinear Online Time. In: Canteaut, A., Ishai, Y. (eds.) Advances in Cryptology – EUROCRYPT 2020. pp. 44–75. Lecture Notes in Computer Science, Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_3
16. Diffie, W., Hellman, M.: New directions in cryptography. IEEE Transactions on Information Theory **22**(6), 644–654 (Nov 1976). https://doi.org/10.1109/TIT.1976.1055638, conference Name: IEEE Transactions on Information Theory
17. Dong, C., Chen, L.: A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In: Kutyłowski, M., Vaidya, J. (eds.) Computer Security - ESORICS 2014, vol. 8712, pp. 380–399. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-11203-9_22, http://link.springer.com/10.1007/978-3-319-11203-9_22, series Title: Lecture Notes in Computer Science
18. Dvir, Z., Gopi, S.: 2-Server PIR with Subpolynomial Communication. Journal of the ACM **63**(4), 1–15 (Nov 2016). https://doi.org/10.1145/2968443, https://dl.acm.org/doi/10.1145/2968443
19. Döttling, N., Garg, S., Ishai, Y., Malavolta, G., Mour, T., Ostrovsky, R.: Trapdoor Hash Functions and Their Applications. In: Advances in Cryptology – CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA,

August 18–22, 2019, Proceedings, Part III. pp. 3–32. Springer-Verlag, Berlin, Heidelberg (Aug 2019). https://doi.org/10.1007/978-3-030-26954-8_1, https://doi.org/10.1007/978-3-030-26954-8_1
20. Efremenko, K.: 3-Query Locally Decodable Codes of Subexponential Length. SIAM Journal on Computing **41**(6), 1694–1703 (Jan 2012). https://doi.org/10.1137/090772721, http://epubs.siam.org/doi/10.1137/090772721
21. Fuchsbauer, G., Konstantinov, M., Krzysztof, P., Rao, V.: Adaptive Security of Constrained PRFs (2014), https://eprint.iacr.org/undefined/undefined
22. Gentry, C., Ramzan, Z.: Single-Database Private Information Retrieval with Constant Communication Rate. In: Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M.Y., Weikum, G., Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) Automata, Languages and Programming, vol. 3580, pp. 803–815. Springer Berlin Heidelberg, Berlin, Heidelberg (2005). https://doi.org/10.1007/11523468_65, http://link.springer.com/10.1007/11523468_65, series Title: Lecture Notes in Computer Science
23. Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions (Extended Abstract). In: FOCS (1984). https://doi.org/10.1109/SFCS.1984.715949
24. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM **43**(3), 431–473 (May 1996). https://doi.org/10.1145/233551.233553, https://doi.org/10.1145/233551.233553
25. Gupta, T., Crooks, N., Mulhern, W., Setty, S., Alvisi, L., Walfish, M.: Scalable and private media consumption with Popcorn. In: Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation. pp. 91–107. NSDI'16, USENIX Association, USA (Mar 2016)
26. Hafiz, S.M., Henry, R.: A Bit More Than a Bit Is More Than a Bit Better: Faster (essentially) optimal-rate many-server PIR. Proceedings on Privacy Enhancing Technologies **2019**(4), 112–131 (Oct 2019). https://doi.org/10.2478/popets-2019-0061, https://petsymposium.org/popets/2019/popets-2019-0061.php
27. Henzinger, A., Hong, M.M., Corrigan-Gibbs, H., Meiklejohn, S., Vaikuntanathan, V.: One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval p. 27 (2022)
28. Hohenberger, S., Koppula, V., Waters, B.: Adaptively Secure Puncturable Pseudorandom Functions in the Standard Model. In: Iwata, T., Cheon, J.H. (eds.) Advances in Cryptology – ASIACRYPT 2015, vol. 9452, pp. 79–102. Springer Berlin Heidelberg, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48797-6_4, http://link.springer.com/10.1007/978-3-662-48797-6_4, series Title: Lecture Notes in Computer Science
29. Holmgren, J., Canetti, R., Richelson, S.: Towards Doubly Efficient Private Information Retrieval. Tech. Rep. 568 (2017), https://eprint.iacr.org/2017/568
30. Kales, D., Omolola, O., Ramacher, S.: Revisiting User Privacy for Certificate Transparency. In: 2019 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 432–447. IEEE, Stockholm, Sweden (Jun 2019). https://doi.org/10.1109/EuroSP.2019.00039, https://ieeexplore.ieee.org/document/8806754/
31. Kiayias, A., Leonardos, N., Lipmaa, H., Pavlyk, K., Tang, Q.: Optimal Rate Private Information Retrieval from Homomorphic Encryption. Proceedings on Privacy Enhancing Technologies **2015**(2), 222–243 (Jun 2015). https://doi.

org/10.1515/popets-2015-0016, https://www.sciendo.com/article/10.1515/popets-2015-0016
32. Kiayias, A., Papadopoulos, S., Triandopoulos, N., Zacharias, T.: Delegatable pseudorandom functions and applications. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 669–684. CCS '13, Association for Computing Machinery, New York, NY, USA (Nov 2013). https://doi.org/10.1145/2508859.2516668, https://doi.org/10.1145/2508859.2516668
33. Kogan, D., Corrigan-Gibbs, H.: Private Blocklist Lookups with Checklist. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 875–892. USENIX Association (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/kogan
34. Kushilevitz, E., Ostrovsky, R.: Replication is not needed: single database, computationally-private information retrieval. In: Proceedings 38th Annual Symposium on Foundations of Computer Science. pp. 364–373. IEEE Comput. Soc, Miami Beach, FL, USA (1997). https://doi.org/10.1109/SFCS.1997.646125, http://ieeexplore.ieee.org/document/646125/
35. Lazzaretti, A., Papamanthou, C.: TreePIR: Sublinear-Time and Polylog-Bandwidth Private Information Retrieval from DDH (2023), https://eprint.iacr.org/2023/204, report Number: 204
36. Lipmaa, H.: An oblivious transfer protocol with log-squared communication. In: Proceedings of the 8th international conference on Information Security. pp. 314–328. ISC'05, Springer-Verlag, Berlin, Heidelberg (Sep 2005). https://doi.org/10.1007/11556992_23, https://doi.org/10.1007/11556992_23
37. Lipmaa, H., Pavlyk, K.: A Simpler Rate-Optimal CPIR Protocol. In: Financial Cryptography and Data Security, 2017 (2017), http://eprint.iacr.org/2017/722
38. Ma, Y., Ke, Z., Rabin, T., Angel, S.: Incremental Offline/Online PIR (extended version). In: USENIX Security 2022 (2022), https://eprint.iacr.org/2021/1438
39. Menon, S.J., Wu, D.J.: Spiral: Fast, High-Rate Single-Server PIR via FHE Composition. In: IEEE Symposium on Security and Privacy, 2022 (2022), https://eprint.iacr.org/2022/368
40. Mughees, M.H., Chen, H., Ren, L.: OnionPIR: Response Efficient Single-Server PIR. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 2292–2306. CCS '21, Association for Computing Machinery, New York, NY, USA (Nov 2021). https://doi.org/10.1145/3460120.3485381, https://doi.org/10.1145/3460120.3485381
41. Patarin, J.: Security of Random Feistel Schemes with 5 or More Rounds. In: Franklin, M. (ed.) Advances in Cryptology – CRYPTO 2004. pp. 106–122. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28628-8_7
42. Peikert, C., Shiehian, S.: Constraining and Watermarking PRFs from Milder Assumptions. In: Public-Key Cryptography – PKC 2020: 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4–7, 2020, Proceedings, Part I. pp. 431–461. Springer-Verlag, Berlin, Heidelberg (May 2020). https://doi.org/10.1007/978-3-030-45374-9_15, https://doi.org/10.1007/978-3-030-45374-9_15
43. Shi, E., Aqeel, W., Chandrasekaran, B., Maggs, B.: Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In: Advances in Cryptology - CRYPTO (2021), http://eprint.iacr.org/2020/1592

44. Singanamalla, S., Chunhapanya, S., Hoyland, J., Vavruša, M., Verma, T., Wu, P., Fayed, M., Heimerl, K., Sullivan, N., Wood, C.: Oblivious DNS over HTTPS (ODoH): A Practical Privacy Enhancement to DNS. Proceedings on Privacy Enhancing Technologies **2021**(4), 575–592 (Oct 2021). https://doi.org/10.2478/popets-2021-0085, https://www.sciendo.com/article/10.2478/popets-2021-0085
45. Stefanov, E., Papamanthou, C., Shi, E.: Practical Dynamic Searchable Encryption with Small Leakage (Jan 2014). https://doi.org/10.14722/ndss.2014.23298
46. Stefanov, E., Shi, E.: FastPRP: Fast pseudo-random permutations for small domains. Cryptology ePrint Report 2012/254. Tech. rep. (2012)
47. Yekhanin, S.: Towards 3-query locally decodable codes of subexponential length. Journal of the ACM **55**(1), 1–16 (Feb 2008). https://doi.org/10.1145/1326554.1326555, https://dl.acm.org/doi/10.1145/1326554.1326555