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

Arthur Lazzaretti and Charalampos Papamathou

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 information about the index 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. 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, 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, but 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

In Private Information Retrieval (PIR), a server holds a public database DB represented as a string of N bits, and a client holds an index i. The goal of the protocol is for the client to learn $\mathsf{DB}[i]$ without the server learning i. Since the problem's introduction [11], PIR has become a building block for a myriad of privacy-preserving applications [1, 2, 20, 26, 37].

In order to circumvent PIR's well-known linear server time lower bound by Beimel et al. [4], Corrigan-Gibbs and Kogan propose a model with *client preprocessing*, where the client circumvents PIR's inherent lower bound over multiple queries by running an expensive preprocessing phase and storing hints. After one expensive *query-independent* offline phase, subsequent queries run privately in time sublinear in the database size. This model has shown to have many useful applications in practice, and brings PIR query times substantially closer to the non-private query baseline.

The core idea of the initial scheme is to process the parities of random sets in the offline phase, interacting with one server. Then, during an online query to index i, we find a preprocessed set S that contains i, and send to the second server $S \setminus \{i\}$. The server returns the parity of the set $S \setminus \{i\}$, and the client can compute the value of i through the difference of its preprocessed parity and the new parity.

To achieve better efficiency, Corrigan-Gibbs and Kogan use small-domain PRP keys to define the preprocessed sets. This ensures sublinear offline communication complexity, and that at query time, we can find a set S that contains i without having to enumerate every set. However, online, we have to send the set without i in plaintext, which means $O(\sqrt{N} \log N)$ communication, since the PRP key leaks information about i, and puncturing a PRP key is impossible [6]. Also, instantiating small-domain PRPs that are efficient turns out to be a non-trivial problem [31, 35, 38, 39]. A second consideration is that this scheme must fail with small probability to ensure that sequential queries to the same index do not leak information by omission. This means that the scheme must be run λ times in parallel to ensure overwhelming correctness. Throughout the paper, we will denote this initial scheme by Corrigan-Gibbs and Kogan PRP-PIR.

A second proposal, by Kogan and Corrigan-Gibbs [26] achieves logarithmic communication through representing their preprocessed sets with Puncturable Pseudorandom Functions [19, 25]. This approach does not directly support fast membership testing, due to the non-invertibility of PRFs, which means that finding a set with your query index takes $O(N \log N)$ expected time. In their work, they solve this problem by using a separate data structure of size proportional to N to help with membership testing. For many usecases of PIR, using $O(N \log N)$ client storage is unfeasible.

A third proposal, by Shi et al. [36] instantiates these sets using Privately Puncturable PRFs from LWE [5, 10]. Although this scheme boasts good complexities relative to the database size, as of yet, there is no implementation of these primitives, and our calculations for the concrete efficiencies of this scheme instantiated with secure parameters show very large overheads due to large factors in the security parameter, 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.

1.1 Our Contribution

Our contribution is two-fold:

1. A new two-server PIR scheme that achieves polylog bandwidth and sublinear server time and client storage, from DDH.
2. An implementation of our new PIR scheme benchmarked against the previous state-of-the-art.

Protocol	Server Time	Client Time	Client Storage	Bandwidth
TreePIR, Lemma 4.1	$O(\sqrt{N}\log^2 N)$	$O(\sqrt{N}\log N)$	$O(\sqrt{N})$	$O(\text{poly}\log N)$
Shi et al. [36][β]	$O(\sqrt{N}\log^5 N)$	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log N)$	$O(\text{poly}\log N)$
Checklist [26]	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log N)$	$O(N\log N)$	$O(\log N)$
PRP-PIR [12]	$O(\sqrt{N}\log N)$	$O(\sqrt{N}\log N)$	$O(\sqrt{N})$	$O(\sqrt{N}\log N)$

[β] The big O notation hides factors very large in the security parameter for this scheme.

Fig. 1. Amortized Complexities over $\sqrt{N}$ queries for a database of size N.

We refer to Figure 1 for an overview of how our new construction compares to previous PIR protocols asymptotically. In Section 5 we go into more detail into the scheme's concrete performance. Our main contribution is a scheme with optimized communication-complexity tradeoffs that is also very fast in practice. To achieve this, we introduce a new primitive we call a Weak Privately Puncturable Pseudorandom Function. Our primitive is defined broadly and can find applications outside the scope of PIR.

Our Weak Privately Puncturable PRF 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 says that one is only able to compute the PRF values from the punctured key if they know the point that was punctured. The punctured index is an additional input the the evaluation algorithm for the punctured key. A second property that a Weak Privately Puncturable PRF must satisfy is that we can enumerate the whole domain of the Weak Privately Puncturable PRF for all 'potentially punctured points' in quasilinear time in the domain of the PRF.

We use Weak Privately Puncturable PRFs with domain and range $\sqrt{N}$ to construct pseudorandom sets. Because of its strong privacy property and fast evaluation over many different potential punctures, these are concise, remain concise after being punctured, and support fast membership testing. Using these sets, we can 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, we 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 and two-server PIR schemes.

Our techniques paired with previous results enable us to achieve PIR with polylogarithmic amortized bandwidth and sublinear amortized server time, paired with previous results in single server PIR.

Notably, paired with the result from Döttling et al. [15], 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.

We implement and benchmark our new scheme, TreePIR, against previous state-of-the-art schemes. TreePIR achieves an amortized query time three times faster than Checklist [26], using fifteen times less client storage. It also achieves speed-ups of over twenty times with respect to amortized query time when benchmarked against the state-of-the-art non-preprocessing two-server PIR, amortizing time over two thousand queries for a database of 268 million elements of 32 bytes each. We provide a full picture of comparisons against previous schemes in Section 5.

1.2 Related Work

The first PIR protocol to achieve non-trivial communication was introduced, along with the problem of PIR itself, by Chor et al. [11]. This scheme relies on a two-server assumption, where the database is replicated in two non-colluding services. This has proven to be a reasonable assumption in practice [20,23,26]. Later, it was shown that non-trivial communication can also be achieved without the two-server assumption [27], 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 bring PIR closer to being practical [3,13,14,16,17,24,28,29,40].

In 2000, Beimel et al. [4] 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, although this direction has not proven very successful, with all known schemes requiring super-linear storage to achieve noticeable reductions in computation time.

1.3 Notation

We define $\nu(\cdot)$ to be a negligible function, such that for every polynomial $p(\cdot)$, $\nu(\cdot) < \frac{1}{p(\cdot)}$. We define overwhelming probability to mean that an event happens with 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 a_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 $i \overset{R}{\leftarrow} S$ to denote that i is an element of S sampled uniformly at random from the set of elements of S. We use $\overset{c}{\approx}$ to denote computational indistinguishability.

1.4 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* $\mathsf{DB}[x_1]$*,. . . ,* $\mathsf{DB}[x_Q]$ *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:*

– ***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.*
– ***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 construction 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 [19], 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 $G : \{0,1\}^\lambda \rightarrow \{0,1\}^{2\lambda}$ 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*

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

We also define below the pseudorandomness property for a PRF.

Definition 2.4 (PRF). *A PRF $F : \{0,1\}^\lambda \times \{0,1\}^n \rightarrow \{0,1\}^m$ 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 $\{0,1\}^n \rightarrow \{0,1\}^m$, for any PPT adversary $\mathcal{A}$, there is exists a negligible function $\nu(\lambda)$ such that*

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

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 $F : \{0,1\}^\lambda \times \{0,1\}^n \rightarrow \{0,1\}^\lambda$ 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 [19], 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.

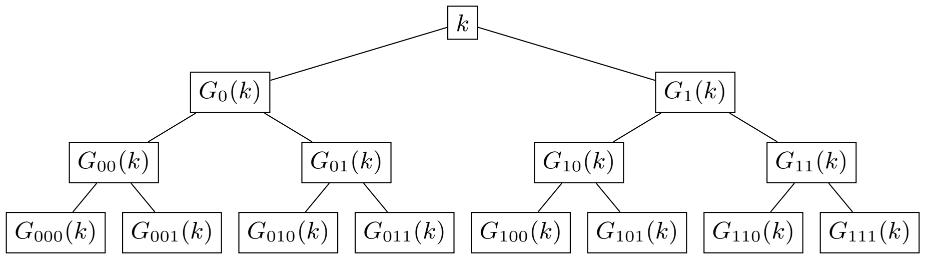


Fig. 2. The GGM PRF tree.

Figure 2 represents the tree for a GGM PRF with input length $n = 3$, key length λ and output length $m = \lambda$. [1] 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

[1] 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 indefinitely.

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)[2]. 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 $\nu(\lambda)$.*

- ***Expt***$_0$: $x \leftarrow \mathcal{A}(1^\lambda)$, $\mathsf{Puncture}(k, x) \to k_x$, $b' \leftarrow \mathcal{A}(k_x, \mathsf{Eval}(x))$.
- ***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 $\forall y \in \{0,1\}^n$ y not equal to x, $\mathsf{Eval}(k, y) = \mathsf{PEval}(k_x, y)$.*

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. [25]. 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 for this

[2] Other works have studied adaptively picked constraints for pPRFs [22, 34].

construction to be correct we require x to be sent along with 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. [25] conduct a formal analysis of this initial pPRF scheme and show that it satisfies the security and correctness properties above.

Next, we show how to modify this well-know GGM construction to achieve our new desired primitive.

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 [5, 8, 10, 33] 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) [25] 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 thinking of the tree)[3]. This punctured key satisfies our privacy and correctness definitions for the pPRF [25].

[3] It is clear that 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.

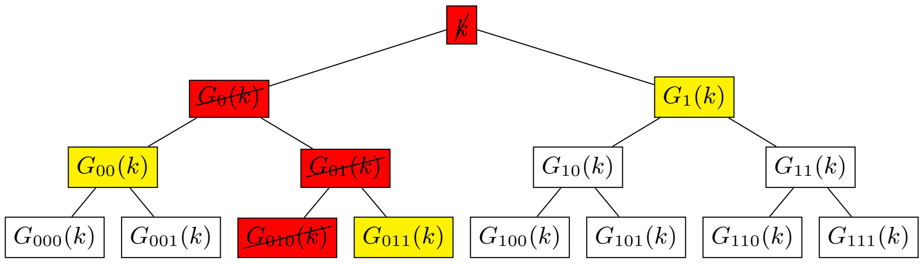


Fig. 3. Puncturing a GGM PRF.

Our first attempt to hide the punctured point is to simply remove it from the key. In our example, instead of outputting the tuple $(010, [G_{00}(k), G_{011}(k), G_1(k)])$ as our punctured key, we output only the ordered array $[G_{00}(k), G_{011}(k), G_1(k)]$. By security of our PRG (Definition 2.3), this should not leak any information about the punctured point—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 [25], the point is necessary to reconstruct the original function evaluations at the other indices.

One approach is to guess and take a punctured point as an additional input in the $P.\mathsf{PEval}(\cdot, \cdot)$ algorithm. A correct guess will enable us be able to evaluate the function as before, however any 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.

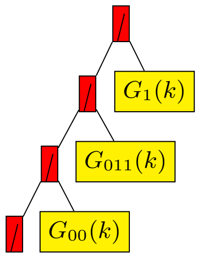


Fig. 4. Reconstructing attempted GGM tree from index and strings.

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 unchanged 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 $N \log N$ time. 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[4]). 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

[4] Note that since these sets are related, we can define all sets in $N \log N$ space, defining the first one 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 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 $O(N \log N)$ time using $O(N \log N)$ space. 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 using* $O(N \log^2 N)$ *space.*

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.

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.

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.

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 [19]. Correctness in weak puncturing follows directly by construction. Security in Puncturing follows from the pPRF security proof in [25], 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 an 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:

$$\mathsf{NumOps} = \sum_{h \in [1,\ldots,n]} \frac{2^n}{2^h} \times 2^h \tag{1}$$

$$= n2^n = N \log N \tag{2}$$

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 can be expressed in $O(N \log N)$ space, where we do not include factors dependent on the output size m. (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 applying 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 sets, and then use these sets to build TreePIR.

4.1 Constructing pseudorandom sets from wpPRFs

As we glanced over in the introduction, all current PIR schemes that achieve sublinear online time 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:

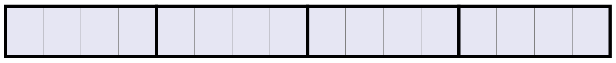
- Short description.
- Non-trivial membership testing.
- Maintaining the short description even after removing one element.

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, given a uniform random key $k \in \{0,1\}^\lambda$ as

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

Our set S will contain each element in $[N]$ with probability $1/\sqrt{N}$. Also the set will be "partitioned", in that it will contain exactly one element for each one of the $\sqrt{N}$ intervals.

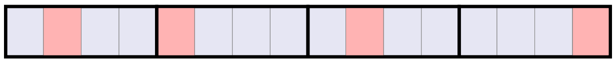
Let us take an example where $N = 16$. We represent the database with a box for each index below. Our set will contain one element within each of the dark boxes (one element within each range of four elements):



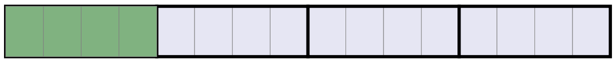
We pick some uniform k, and evaluate it at $\sqrt{16} = 4$ points, such that

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

Then, our set coverage with respect to the database would look as follows.



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 the server would know that the element is within the green elements below:



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. Then, given the set's punctured key, the server cannot guess the punctured index with probability better than $1/N$.

To summarize, our set satisfies the following properties:

1. It can be represented in λ bits by its key k.
2. We can check membership in one evaluation. For any $x = x^\ell || x^r \in [N]$, we simply check 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:

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

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 $O(\sqrt{N}\log^2 N)$ time[5].

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 the all $\sqrt{N}$ 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.

If we are willing to allow for a three-server assumption, we can also compute the online phase on *both* online servers and retrieve the parity in additional $\sqrt{N}$ time with $\log N$ total bandwidth and no encryption. We discuss the trade-offs of each in more detail in Section 4.3, but note that regardless of the approach we pick, our server time remains $\sqrt{N}\log^2 N$, since for the single-server PIR approach and the DPF approach, we are running these protocols on a database of size $\sqrt{N}$.

4.2 Our TreePIR Scheme

In Figure 6, we give the full scheme, based on the intuition above. On our implementation, we can make Step 1 of our Online Query this algorithm deterministic by adding *shifts* [12, 32] to our wpPRF scheme. We note that wpPRFs makes our scheme considerably simpler than previous schemes based on the same

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 $k_i \leftarrow F.\mathsf{Gen}(1^\lambda)$, $i = 1, \dots, M-1$, and sends $k_0, \dots, k_{M-1}$ to **server**$_0$.
2. For all $i = 0, \dots, M-1$ **server**$_0$ computes hints
$$h_i = \oplus_{j \in S_i} \mathsf{DB}[j]\,,$$
where
$$S_i = \{v || F.\mathsf{Eval}(k_i, v) : v \in \left[\sqrt{N}\right]\}\,,$$
and sends $h_0, \dots, h_{M-1}$ to **client**.
3. **client** stores pairs of keys and hints in a table
$$\mathsf{T} = \{T_j = (k_j, h_j)\}_{j \in [M]}\,.$$

- 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 $\mathcal{P}_b = [p_0, \dots, p_{\sqrt{N}}]$, where $p_i = \oplus_{j \in S_i} \mathsf{DB}[j]$,
$$S_i = \{\, v \,||\, F.\mathsf{PEval}(k_{punc}, i, v)\,\}_{i \in [\sqrt{N}]}, \text{ for } v \in \left[\sqrt{N}\right].$$
3. Return $\mathcal{P}_b$.

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

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

Fig. 6. Our novel PIR scheme, TreePIR.

paradigm [12, 26, 36], 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, our scheme given in Figure 6 satisfies correctness and privacy for multi-query PIR schemes as defined in Definition 2.1, 2.2 and runs with:*

- *$O(\lambda N \log N)$ offline server time and $O(\lambda\sqrt{N})$ offline client time.*
- *$O(\sqrt{N}\log^2 N)$ online server time and $O(\sqrt{N}\log N)$ 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 $O(\sqrt{N}\log^2 n)$ 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. We can make it deterministic $O(\log N)$ time by instantiating our sets with random shifts. The technique and proof of equivalence follow exactly as in [12, Appendix B.5].

Privacy with respect to ***server***$_1$. Offline, **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 **server**$_1$ and *serv*$_0$ who acts as **server**$_0$. **client** is invoked with query x_1, ignores x_1, picks some random $y \in [N]$ as its query index.

By Definition 2.6, for any $y \in [N]$, a query to $x_1^\ell || x_1^r$ is indistinguishable from a query to $x_1^\ell || y^r$, since $k_{x_1^\ell || x_1^r} \overset{c}{\approx} k_{x_1^\ell || y^r}$. The punctured key reveals nothing about the evaluation at the punctured index.

Then, by Definition 3.2, a query to $x_1^\ell || y^r$ is indistinguishable from a query to $y^\ell || y^r$, because $k_{x_1^\ell || y^r} \overset{c}{\approx} k_{y^\ell || y^r}$: the punctured key reveals nothing about the index that was punctured. Therefore, by the transitive property, we have that the punctured key for x and y (namely k_{x_1} and k_y) are computationally indistinguishable, and since this is the only thing the **server**$_1$ sees, it has no way of distinguishing between a query to its desired index x and a random y.

It follows that $\mathbf{Expt}_0$ and **Hyb** are indistinguishable to **server**$_1$[6]. It also follows in suit that **Hyb** and $\mathbf{Exp}_1$ are indistinguishable. Since **client** never uses the invoked query x_1 and instead picks a random y, we can define our simulator

[5] Database accesses incur an additional $\log N$ factor per element.

[6] Another way to show this is to look at the distribution of elements the server sees for each key and verify that they are indeed the same and independent of the index punctured.

Sim exactly as above. Note that for the next query, 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. It also follows from above that the server's view for the next query is *independent* of the previous query, since it always sees a uniform random punctured key from the same distribution.

Privacy with respect to ***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 **server**$_0$ can be argued symmetrically from the same arguments as privacy with respect to **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 **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:

$$\begin{aligned}
\Pr\left[x_1 \notin \{S_i\}_{i\in[\lambda\sqrt{N}]}\right] &= \Pr\left[\forall i \in [\lambda\sqrt{N}],\ F.\mathsf{Eval}(k_i, x_1^\ell) \neq x_1^r\right] && (3)\\
&= \left(1 - \frac{1}{\sqrt{N}}\right)^{\lambda\sqrt{N}} && (4)\\
&\leq \left(\frac{1}{e}\right)^{\lambda} \leq \nu(\lambda). && (5)
\end{aligned}$$

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 $x_1 \in S_j = \{i \,||\, F.\mathsf{Eval}(k_j, i)\}_{i\in[\sqrt{N}]}$, then:

$$\begin{aligned}
\mathsf{DB}[x_1] &= \left(\bigoplus_{i\in S_j} \mathsf{DB}[i]\right) \oplus \left(\bigoplus_{i\in S_j\setminus\{x_1\}} \mathsf{DB}[i]\right) && (6)\\
&= h_j \oplus \left(\bigoplus_{k\in S_{j,x_1^\ell}} \mathsf{DB}[k]\right) && (7)\\
&= h_j \oplus \mathcal{P}_1[x_1^\ell], && (8)
\end{aligned}$$

where:

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

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 hint 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 Driving communication complexity down to logarithmic

Prior works [15, 30] 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.

Lemma 4.1 (TreePIR with reduced bandwidth). *Theorem 4.1 implies a two-server PIR with the same complexities except with polylogarithmic bandwidth online.*

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 [9, 15, 17, 28, 30]. This means that we also have to introduce any assumptions used by the scheme selected. □

On Section 5 we benchmark the performance of our TreePIR paired with SPIRAL [30]. Note that we *cannot* recurse with a preprocessing PIR scheme (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 $O(\sqrt{N}\log^2 N)$ server time and $O(\sqrt{N})$ client space. If we tune our sets to be a different size, for example size $N^{1/D}$, then we can achieve faster server time of $N^{1/D}\log^2 N$ in exchange for a larger client storage of $N^{D-1/D}$. 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 $N^{1/2}$ tradeoff.

5 Performance

We implement TreePIR in 530 lines of C++ code and 470 lines of Go code. Our starting point was the previous optimized implementations of PIR by Kogan and Corrigan Gibbs [26] and Kales et al. [23].

We benchmark our scheme from Theorem 4.1 against PRP-PIR [12] implemented by Ma et al. [35], Checklist by Kogan and Corrigan-Gibbs [26], and the DPF-based PIR initially introduced by Gilboa and Ishai [7, 18, 21]. We run the analysis of amortized query time across two thousand queries for a database of four million elements of 256 bytes Figure 7 and a database of 268 million elements of 32 bytes. The tests are run on a single thread in an Amazon Web Services EC2 instance of size m5d.8xlarge.

Results for database of 2^{22} elements of size 256 bytes.

Protocol	**Amortized Query Time**	**Client Storage**
TreePIR	15ms	67MB
Checklist	18ms	78MB
PRP-PIR	315ms	67MB
DPF PIR	84ms	0

Fig. 7. Amortized query time for moderate database size.

Results for database of 2^{28} elements of size 32 bytes.

Protocol	**Amortized Query Time**	**Client Storage**
TreePIR	256ms	67MB
Checklist [26]	811ms	1GB
DPF PIR [7, 18, 21]	5480ms	0

Fig. 8. Amortized query time for large database of small elements.

The results seen in Figure 7 and Figure 8 represent TreePIR with no recursion. For either database size benchmarked, recursing with a DPF query would take less than 0.4ms, the time for a DPF query with a database of $\sqrt{2^{28}} = 2^{14}$ elements, which means that the query time would be basically unchanged, but we would require three servers. To recurse with SPIRAL, maintaining the two-server assumption, we would pay 61ms to recurse 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 would mean that in reducing bandwidth on the two-server setting, 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 of size 2^{28} elements of size 32 bytes. Also, any future improvements with single-server PIR or two-server PIR would also automatically imply a faster TreePIR. While we report amortized query time across two thousand queries, when measuring across more queries, TreePIR approaches an amortized query time of 23ms for the database of 2^{28} elements and 4.7ms for the database of size 2^{22}.

By coincidence, the client storage for TreePIR on both databases turned out to be almost exactly the same. This is because the client storage is proportional to both database size *and* element size. For larger databases, there is a significant discrepancy in the storage needed for TreePIR and Checklist.

On the absence of the scheme by Shi et al. [36] Note that we do not benchmark the Shi et al. scheme [36] because there is no known implementation of the Privately Puncturable PRF primitive. However, given the sample parameter instantiation of privately puncturable PRFs by [5], a conservative estimate on the punctured key size is of $\lambda^5 \cdot \log^3 N$. This means an online per query communication cost of over 30 billion bits given a security parameter of size 128 bits, for any database size. This makes the scheme impractical for most PIR usecases.

On the performance of PRP-PIR [12] To benchmark PRP-PIR [12], we use a separate library provided by Ma et al. [35]. 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 [31, 38] put overhead in the membership testing and evaluation, both of which are performed numerous times thoroughout the scheme. We do not benchmark PRP-PIR against the larger database sizes run the code by [35] run on a database larger that 2^{22} elements.

On the performance of Checklist [26] Running a very small number of queries, Checklist outperforms TreePIR in terms of query time by around a factor of approximately 8. However, when running many queries on large databases, such as was the case to benchmark, it turned out that Checklist was inconsistent. This is because 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 superlinear 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 $O(\lambda N \log N)$. 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 $N \log N$ in the worst case.

5.1 Supporting Mutable Databases

Techniques to support preprocessing in databases that change over time have been studied in previous work [26, 35]. These techniques can also be applied to TreePIR and are able maintain most of the benefits of preprocessing with small overhead.

5.2 Parallelizing Evaluations

Note that although our construction highly exploits sequential computation, many subsets of the computation can be parallelized. At a high level, it is easy to see that the first and last output set have no overlap, their tree configurations are completely different. This also holds true recursively within each subtree, and the computation could be optimized to reflect this.

References

1. Angel, S., Setty, S.: Unobservable Communication over Fully Untrusted Infrastructure. pp. 551–569 (2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/angel
2. 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
3. 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
4. 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
5. 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
6. Boneh, D., Kim, S., Wu, D.J.: Constrained Keys for Invertible Pseudorandom Functions. In: Kalai, Y., Reyzin, L. (eds.) Theory of Cryptography. pp. 237–263. Lecture Notes in Computer Science, Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-70500-2_9
7. 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

8. 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`
9. 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`
10. 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`
11. Chor, B., Kushilevitz, E., Goldreich, O., Sudan, M.: Private information retrieval. Journal of the ACM **45**(6), 965–981 (Nov 1998). `https://doi.org/10.1145/293347.293350`, `https://doi.org/10.1145/293347.293350`
12. 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`
13. 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
14. 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`
15. 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`
16. 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`
17. 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
18. Gilboa, N., Ishai, Y.: Distributed Point Functions and Their Applications. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology – EUROCRYPT 2014. pp.

640–658. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_35

19. Goldreich, O., Goldwasser, S., Micali, S.: How to Construct Random Functions (Extended Abstract). In: FOCS (1984). https://doi.org/10.1109/SFCS.1984.715949
20. Gupta, T., Crooks, N., Mulhern, W., Setty, S., Alvisi, L., Walfish, M.: Scalable and Private Media Consumption with Popcorn. pp. 91–107 (2016), https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/gupta-trinabh
21. 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
22. 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
23. 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/
24. 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
25. 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
26. Kogan, D., Corrigan-Gibbs, H.: Private Blocklist Lookups with Checklist. pp. 875–892 (2021), https://www.usenix.org/conference/usenixsecurity21/presentation/kogan
27. 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/
28. 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
29. Lipmaa, H., Pavlyk, K.: A Simpler Rate-Optimal CPIR Protocol. In: Financial Cryptography and Data Security, 2017 (2017), http://eprint.iacr.org/2017/722

30. 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
31. 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
32. Patel, S., Persiano, G., Yeo, K.: Private Stateful Information Retrieval. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 1002–1019. CCS '18, Association for Computing Machinery, New York, NY, USA (Oct 2018). https://doi.org/10.1145/3243734.3243821, https://doi.org/10.1145/3243734.3243821
33. 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
34. Pietrzak, Momchil Konstantinov, K.G.F., Rao, V.: Adaptive Security of Constrained PRFs (2014), https://eprint.iacr.org/undefined/undefined
35. Rabin, Ke Zhong, T.Y.M., Angel, S.: Incremental Offline/Online PIR (extended version). In: USENIX Security 2022 (2022), https://eprint.iacr.org/2021/1438
36. 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
37. 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
38. Stefanov, E., Shi, E.: FastPRP: Fast pseudo-random permutations for small domains. Cryptology ePrint Report 2012/254. Tech. rep. (2012)
39. Tessaro, Stefano, V.T.H., Trieu, N.: The Curse of Small Domains: New Attacks on Format-Preserving Encryption. Tech. Rep. 556 (2018), https://eprint.iacr.org/2018/556
40. 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